The Logical Design of Parallel Operating Systems

Wolfgang Schröder-Preikschat

German National Research Center
for Computer Science
GMD FIRST
Rudower Chaussee 5, D-12489 Berlin, Germany
wosch@first.gmd.de

June 8, 1994

Petra and Sophie

Contents

1 Introduction **1**
- 1.1 About This Book 4
- 1.2 Parallel Operating Systems 6
  - 1.2.1 Shared-Memory Systems 7
  - 1.2.2 Distributed-Memory Systems 10
  - 1.2.3 Massively Parallel Systems 17
- 1.3 Design Constraints 20
  - 1.3.1 Message Startup Time 20
  - 1.3.2 System Complexity 25
- 1.4 Design Principles 27
  - 1.4.1 Program Families 27
  - 1.4.2 Object Orientation 28
- 1.5 PEACE 31
- 1.6 Summary 35

2 Functional Decomposition **37**
- 2.1 Approaching the Family Concept 38
  - 2.1.1 Minimal Subset of System Functions 39
    - 2.1.1.1 Process and Message Abstractions . . . 40
    - 2.1.1.2 Latency Hiding 43
  - 2.1.2 Minimal System Extensions 45
  - 2.1.3 Exchangeable System Components 46
- 2.2 Macrostructure 47
  - 2.2.1 Functionally Dedicated Building Blocks 49
  - 2.2.2 Extensibility and Configurability 50
  - 2.2.3 User and Supervisor Modes 51
  - 2.2.4 Intra- and Inter-Domain Calls 53

2.3 Functional Hierarchy . . . 56
2.3.1 Invocation Schemes . . . 57
2.3.2 Actual Structure . . . 59
2.3.3 Kernel Entity Configurations . . . 60
2.3.4 Multithreaded Kernel Entity . . . 64
2.4 Microstructure . . . 65
2.4.1 Single-User/Single-Tasking . . . 67
2.4.2 Multi-User/Single-Tasking . . . 69
2.4.3 Single-User/Multitasking . . . 71
2.4.4 Multi-User/Multitasking . . . 72
2.5 Communication System . . . 72
2.5.1 Network Capabilities . . . 73
2.5.1.1 Data Security . . . 75
2.5.1.2 Segmentation and Reassembly . . . 76
2.5.1.3 Flow Control . . . 79
2.5.1.4 Sequencing . . . 81
2.5.2 Heterogeneity . . . 84
2.5.2.1 Different Network Subsystems . . . 84
2.5.2.2 Integrated Systems . . . 86
2.5.2.3 Representation-Independent Protocols . 87
2.5.3 Problem-Oriented Protocol Layers . . . 88
2.5.3.1 Inter-Nucleus Protocol . . . 89
2.5.3.2 Transport Protocol . . . 90
2.5.3.3 Network Device Protocol . . . 92
2.6 Separation of Concerns . . . 95
2.6.1 Vertical vs. Horizontal Isolation . . . 96
2.6.2 Normal vs. Exceptional Processing . . . 98
2.6.3 Static vs. Dynamic Configuration . . . 100
2.7 Summary . . . 103

3 System Composition 107
3.1 Native Operating System . . . 108
3.1.1 Configuration Aspects . . . 108
3.1.2 Configuration Variety . . . 109
3.1.2.1 User Nodes . . . 109
3.1.2.2 System Nodes . . . 113
3.1.2.3 Generic Nodes . . . 115

3.1.3 Configuration Interplay 117
3.2 Parallel Computing Platform 117
3.2.1 User-Level Runtime Executive 119
3.2.2 Host Interconnection 120
3.2.2.1 Asymmetrical Coupling 120
3.2.2.2 Logically Symmetric Coupling 122
3.2.2.3 Physically Symmetric Coupling 123
3.2.3 Logical Nodes 125
3.2.4 Applicability 125
3.3 Latency-Hiding Architecture 126
3.3.1 Dual-Processor Node Architecture 127
3.3.2 Coprocessing 129
3.3.2.1 Application Processor Tasks 130
3.3.2.2 Communication Processor Tasks 132
3.3.3 Asymmetric Multiprocessing 134
3.3.4 Symmetric Multiprocessing 137
3.4 Summary 139

4 Operating Principle 141
4.1 Interrupt Handling 142
4.1.1 Volatile and Nonvolatile Register Set 143
4.1.2 Event Synchronization 145
4.1.3 Serialization of Asynchronous Activities 148
4.1.4 Prologues and Epilogues 151
4.1.5 Locking Delays and Preemption Points 154
4.1.6 Integrating Busy Waiting 156
4.2 Abstract Protocol Processors 158
4.2.1 Abstract Machine Primitives 158
4.2.2 Protocol Sequence Expressions 160
4.2.2.1 Packet Transfer 161
4.2.2.2 Segment Transfer 162
4.3 Inter-Node Communication 163
4.3.1 Logical Structure of the Network Driver 164
4.3.2 Layout of a Network Data Unit 168
4.3.2.1 Service and Protocol Data Unit 168
4.3.2.2 Message Buffer Descriptors 171
4.3.3 Exploiting the Holder Abstraction 173

4.3.4 High-Volume Data Transfer 174
4.3.4.1 Sender-Initiated Data Transfer 175
4.3.4.2 Receiver-Initiated Data Transfer 177
4.3.4.3 Reassembly of Segmented Messages . . . 179
4.3.5 Communication Latency 180
4.3.6 Busy Waiting and Interrupt Handling 184
4.4 Approximating Active Messages 186
4.4.1 Active Messages 186
4.4.2 Epilogue Processing 190
4.4.3 Thread Model 191
4.4.4 Reactive Objects 192
4.4.5 System Integrity 194
4.5 Summary 197

5 Transient Objects 199
5.1 Extend-On-Use Property 200
5.1.1 Object Granularity 202
5.1.2 Incremental Loading 204
5.1.3 Garbage Collection 205
5.1.4 Exception Handling 207
5.2 Dynamically Alterable System 208
5.2.1 Service Access Point 209
5.2.1.1 Passive Object Reconfiguration 209
5.2.1.2 Active Object Reconfiguration 211
5.2.2 Third-Party Configuration 214
5.2.3 Bootstrapping 217
5.2.3.1 Master Nodes 217
5.2.3.2 Slave Nodes 219
5.3 Object Localization 220
5.3.1 Implicit Addressing Through Objects 220
5.3.2 Dual Objects 222
5.3.2.1 Private and Public State 222
5.3.2.2 Prototype and Likeness 224
5.3.2.3 Programming-Language-Level Support . 227
5.3.3 Object Placement 228
5.3.4 Unique Identifiers 231
5.3.4.1 Local Unique Identifier 232

5.3.4.2 Global Unique Identifier 237
5.3.4.3 Opaque Identifier 239
5.3.5 Communication Gates 241
5.3.6 Naming Abstractions 243
5.3.6.1 Access and Location Transparency . . . 245
5.3.6.2 Replication Transparency 247
5.3.6.3 Scaling Transparency 247
5.3.7 Name Space Partitioning 248
5.3.7.1 Name Domains 249
5.3.7.2 Entity to Name Domain Relationship . . 252
5.3.7.3 Functionally Dedicated Entities 254
5.3.8 Search Space Selection 257
5.3.8.1 Name Server Instantiation 258
5.3.8.2 Domain Clerk Addressing 260
5.4 Summary 263

6 Class Ensemble 265
6.1 Basic Abstractions 266
6.1.1 Queues, Stacks and Chains 266
6.1.2 Event Synchronization 268
6.1.3 Lock Variables 272
6.1.4 Critical Sections 274
6.2 Processor Abstractions 276
6.2.1 Propagation of Hardware Exceptions 277
6.2.1.1 Exception Handler Linkage 278
6.2.1.2 Trap/Interrupt Linkage Descriptor . . . 279
6.2.2 Kernel-Level Exception Handling 284
6.2.2.1 Passive Object Model 285
6.2.2.2 Active Object Model 287
6.2.3 Timer Interrupt Handler 291
6.2.4 Cross-Domain Call Trap Handler 293
6.3 Object Addressing 299
6.3.1 Modeling Opaque Identifiers 299
6.3.2 Hash Item 300
6.3.3 Hash Table 302
6.3.4 Process Gates 304
6.3.4.1 Modeling Communication Endpoints . . 305

6.3.4.2 Binding Active Objects 310
6.4 Processes 313
6.4.1 Active Objects 313
6.4.2 User-Level Active Objects 314
6.4.3 Threads Package 316
6.4.4 Kernel-Level Active Object 318
6.4.4.1 Process Control Block 319
6.4.4.2 Team Control Block 324
6.4.5 Extensions to the Minimal Basis 327
6.4.6 Installation of Active Objects 331
6.5 Messages 332
6.5.1 Locally Operating Nucleus 332
6.5.2 Remotely Operating Nucleus 335
6.5.3 Inter-Nucleus Protocol Data Units 338
6.5.3.1 Class Selection 339
6.5.3.2 Routing 341
6.5.3.3 Thread Control 342
6.5.3.4 Segment Transfer 342
6.5.3.5 Packet Transfer 345
6.6 Program Families Revisited 346
6.7 Summary 350

7 System Abstractions 353
7.1 Execution Model 354
7.1.1 Processes 355
7.1.2 Leagues 357
7.1.3 Synchronization 358
7.1.4 Communication 360
7.1.4.1 Packet Transfer 360
7.1.4.2 Segment Transfer 370
7.2 Software Backplane 376
7.2.1 Class Hierarchy 378
7.2.2 Interface Classes 381
7.2.2.1 Thread Localization 381
7.2.2.2 Packet Delivery 383
7.2.2.3 Packet Acceptance 384
7.2.2.4 Segment Transmission 388

7.2.2.5 Assessment 388
7.3 Message Streams 392
7.3.1 The Ideal Model 393
7.3.2 Typed Data Streams 395
7.4 Distributed Processes 400
7.5 Summary 405

8 Discussion 407
8.1 Microkernels 409
8.2 Programming Environments 413
8.3 Trends 416
8.4 Summary 420

Glossary 423

Bibliography 429

Index 441

List of Figures

1.1 Program families vs. object orientation 29
1.2 Peace family tree 33

2.1 Building blocks 50
2.2 Service access point of the nucleus 54
2.3 Inter-domain call stubs 55
2.4 Problem-oriented invocation schemes 57
2.5 Kernel entity in a single-address-space configuration . . . 61
2.6 Kernel entity in a multi-address-space configuration . . . 63
2.7 Single- vs. multi-address-space model 64
2.8 Nucleus family tree 67
2.9 Communication system architecture 89
2.10 Software vs. hardware transport protocol features 91
2.11 Coexisting transport protocol entities 94

3.1 Operating-mode variety 110
3.2 A platform for workstation clustering 118
3.3 Asymmetrical host coupling 121
3.4 Logically symmetric host coupling 123
3.5 Physically symmetric host coupling 124
3.6 Example of a dual-processor node architecture 128
3.7 Message-passing coprocessor 130
3.8 Asymmetric multiprocessing 135
3.9 Symmetric multiprocessing 138

4.1 Fundamental steps in interrupt processing 145
4.2 Structuring of event-handling subsystems 152
4.3 Network driver subsystem 165

4.4 Correlation between the SDU and PDU 170
4.5 Message holder abstraction 172
4.6 Sender-initiated high-volume data transfer 176
4.7 Receiver-initiated high-volume data transfer 177
4.8 Communication latency breakdown 183
4.9 Flow of control in the active-message model 188
4.10 Flow of control using reactive objects 193

5.1 Inter-domain call variants 210
5.2 Interaction with replicated service functions 212
5.3 Configuration by means of a third party 215
5.4 Relationship between prototype and likeness 225
5.5 Prototype placement 230
5.6 Structured communication endpoint address 233
5.7 Use of the communication endpoint address 234
5.8 Systemwide unique communication endpoint address . . 238
5.9 Representations of an opaque identifier 240
5.10 Gates used as forwarding identifiers 242
5.11 Naming abstractions 245
5.12 Name space with different name domains 250
5.13 Hierarchically arranged name servers 251
5.14 Problem-oriented name server ensemble 253
5.15 Name space structure for single-user mode of operation . 255
5.16 Name space structure for multi-user mode of operation . 256
5.17 Chaining of name servers 260
5.18 The root gate as domain clerk forwarding identifier . . . 262

6.1 Queue abstraction 267
6.2 Abstractions for prologue and epilogue handling 269
6.3 Abstractions implementing lock variables 272
6.4 A simple spin-locking strategy 273
6.5 Lock abstraction used by the nucleus 274
6.6 Basic abstraction to secure critical sections 275
6.7 Translating traps/interrupts into object invocations . . . 278
6.8 Linkage descriptor for trap/interrupt handlers 280
6.9 Assembly-level (MC68020) wrapper routine 283
6.10 Trap/interrupt handler as an active or passive object . . 285

6.11 The shepherd class 290
6.12 Timer interrupt handler as a passive object 292
6.13 VICE class specification 294
6.14 VICE (MC68020) nucleus service function stub 295
6.15 VICE trap handler 297
6.16 A nucleus service function carrier 298
6.17 Hybrid representation of an opaque identifier 300
6.18 Hash element composition 301
6.19 Hash table composition 302
6.20 A communication endpoint address 306
6.21 A communication/process gate 307
6.22 Interfacing to communication/process gates 308
6.23 User abstractions 315
6.24 Thread customization 317
6.25 Threads package primitives 318
6.26 The process control block 320
6.27 The team control block 325
6.28 Adding a new process class 328
6.29 Networkless nucleus 333
6.30 A segment container 334
6.31 Networking nucleus 336
6.32 NICE class hierarchy 340
6.33 A thread family 348

7.1 Process model 356
7.2 Thread synchronization 359
7.3 Rendezvous with a thread 361
7.4 Rendezvous with a team 363
7.5 Delegating a rendezvous request 365
7.6 Location transparency of mobile objects 367
7.7 Relaxed rendezvous 369
7.8 Segment transfer models 372
7.9 Monitored segment transfer 376
7.10 Nucleus as a software backplane 378
7.11 Information hiding using interface classes 379
7.12 Code template performing thread localization 382
7.13 Single-buffer asynchronous communication 385

7.14 Receive of a message packet 386
7.15 Ticked-based segment transmission 389
7.16 Virtual channel communication 394
7.17 Writer stream abstraction 396
7.18 Reader stream abstraction 397
7.19 A class hierarchy implementing virtual channels 399
7.20 A thread implemented as a dual object 401
7.21 The primer thread 402
7.22 The actor thread 403

List of Tables

1.1 Local vs. remote memory access times 11
1.2 Dimensions of message latency 22

2.1 Message startup time problem 66

3.1 Properties of user nodes 112
3.2 Properties of system nodes 113
3.3 Properties of generic nodes 116

4.1 Protocol processor communication primitives 159
4.2 Protocol processor management primitives 160

6.1 Methods for searching hash items 304

Foreword

Driven by the tremendous progress in microprocessor technology and architecture of recent years, parallel computers have come to offer very-high performance at maximum cost-effectiveness. CMOS technology has become available that allows several ten million transistors to be put on a single chip, while a sophisticated processor such as the Pentium has "only" about three million transistors. The next step will be to have several (actually four because of pin limitation) complete processors on a chip—for the price of one. Consequently, the standard building block of any kind of computer system, from workstations to supercomputers, will be the multiprocessor node in which several active threads will be executed in parallel.

While the processors of the node will communicate through the shared memory, the larger parallel computers built with such nodes will be message-passing architectures, as this is the only form that provides unrestricted upward scalability. The problem with the message-passing programming paradigm will be resolved either by virtual shared memory or high-level programming models and parallelizing compilers that free the programmer from having to deal with low-level aspects such as data distribution, communication, and thread coordination.

The key issue of message-passing architectures is communication latency. This problem ceased years ago to be a hardware issue. Modern technology allows for interconnection networks whose transmission latencies are negligible compared to the message startup time in the node, which is primarily an operating system problem. Making communication purely a hardware activity that is carried out below the operating-system level may be a solution in some "dedicated machines", but is certainly not the answer in general purpose computers—one should not try to throw away the computer world back by three decades! Hence, the communication startup overhead incurred by the operating system is unavoidable. However, what one can do is (1) make it as small as possible in the first place and (2) reduce the effect of the remaining latency by latency hiding.

More than a decade ago we introduced into computer architecture the concept of hiding the message-passing latency by executing in the nodes of a distributed memory system computation threads and communication threads in parallel. In 1981 we completed such an architecture, UPPER, but had to learn that the overhead of the monolithic operating system of the time (we used Motorola's Versados) largely defeated our attempt. In the late '80s we proposed that approach again, and by now it has become a standard in most commercial parallel computers.

In 1986 we began designing SUPRENUM, the first European supercomputer development. The objective was a massively-parallel number cruncher that would provide the highest possible performance at a cost-effectiveness better than that of the common vector machines. Before the advent of the superscaler processor, the only way to achieve substantial MFLOPS performance in the node was by adding a vector processor to the CPU. The coarse-grain parallelism of vector processing made the communication latency less critical. Nevertheless, Wolfgang Schröder-Preikschat, the designer of SUPRENUM's node operating system, PEACE, set out to minimize the startup times as much as the state of the art would permit. His endeavor was based on the new microkernel approach, which he was familiar with from developments such as *Thoth*, *V*, and *Amoeba*, as well as from his Ph.D. work. Consequently, PEACE became the first microkernel operating system implemented on any commercial parallel computer. Its 400 microseconds for a single message-passing activity, at that time, was the speed record.

In 1990 we started with the development of a "next generation" parallel computer, MANNA, based on the first available superscalar processor (Intel i860). MANNA was to become the prototype of a general purpose parallel computer for numerical and non-numerical applications, whose users would be given the choice of programming either in the—more efficient—message-passing paradigm or the—easier—shared-memory model. For the latter case, MANNA provides as operating system service an efficient virtual shared memory mechanism. Such a use demanded the efficient execution of fine-grain parallelism and, hence, a drastic reduction in startup latency. A first step towards achieving that goal was Wolfgang's innovative concept of a *family of operating system kernels* that makes the kernel configurable, to provide only the services the user wants to have. Thus, no overhead need be paid for unrequested functions. A second step was to revisit the concept of the dual processor node.

The underlying notion of the family concept is to make all components of the operating system—modules, threads, mechanisms—as lightweight as possible. This book demonstrates how this goal can be accomplished by modularizing even the operating system kernel which, even in microkernel systems, traditionally is monolithic (thus, microkernel systems deserve not really the attribute "micro"). The rationale for this extreme "decomposition of complexity" is elaborated in the book. The hierarchical decomposition of the kernel functionality moreover allows for their distribution over the processors of a multiprocessor node architecture. Only then will the concept of the dual processor node really bear fruit, as is strikingly demonstrated in the book by Table 1.2. Compared to the commercial systems that use the same concept, MANNA's effective message latency is an order of magnitude smaller! It has been brought down to the microseconds range, a figure so far obtained only on "naked hardware," i.e., without operating system involvement.

For all its striking advantages, I believe that the operating system of the future for distributed memory machines will exhibit the anatomy outlined in this book. The author not only explains convincingly the construction principles for a new generation of parallel operating systems by example of his "*objective* PEACE," he also discusses in detail all conceivable angles of such a design. This book will be seminal, strongly influencing the future operating system designs. For me, who had the privilege of years of inspiring work with Wolfgang Schröder-Preikschat, it was gratifying witnessing his work and see the concepts presented in this book evolve.

Del Mar, California, February 1994
Wolfgang K. Giloi

Preface

Forthcoming massively parallel systems will have distributed-memory architectures. They consist of several hundreds to thousands of autonomous processing nodes interconnected by a high-speed network. A major challenge in operating-system design for these architectures is to design a structure that reduces system bootstrap time, avoids bottlenecks in serving system calls, promotes fault tolerance, and is dynamically alterable and application-oriented.

Operating-system construction is primarily a design and structuring problem. This is true for all kinds of operating systems and spans the areas of distributed, parallel, real-time, and fault-tolerant computing. The situation does not change with massively parallel (distributed-memory) architectures. A highly scalable system structure is required. This structure must scale well with at least hundreds to thousands of nodes and should scale well with millions of nodes. Thus, the structure should meet the needs of both today's and future parallel architectures. In addition to all of these constraints regarding how to divide the complexity of the system, there is also the requirement of finding a system structure that guarantees the least possible message startup time for a given parallel application.

This work describes the logical design of state-of-the-art parallel operating systems that have to meet the needs of both massively parallel computer architectures and massively parallel applications. The focus is on how to organize and structure these operating systems; this is not a discussion of algorithms that implement specific system features. Thus, it is a software engineering course explaining operating-system structuring rather than implementation. By starting on an abstract level, the "big picture" of a parallel operating system is stepwise refined.

The refinement is carried out top-down to the implementation level, to give an impression of the message startup time overhead implied by a hierarchically composed system.

The design approach followed is to understand a parallel operating system as a *program family* and to use *object orientation* as the fundamental implementation discipline. The former concept (program family) helps prevent the design of a monolithic system organization, while object orientation enables the efficient implementation of a highly modular system structure. An in-depth presentation is given rather than a broad overview. Instead of discussing several designs more or less superficially, a single, elaborate case study is made on the basis of PEACE. The object-oriented design of the PEACE family of parallel operating systems is thoroughly discussed.

Acknowledgments

First of all, I would like to express my gratitude to Wolfgang K. Giloi. With him, I came to appreciate a very knowledgeable computer scientist who knows one's business and how to fill one with enthusiasm. Thanks a lot for your support and advice.

Peter Behr, the acting director of GMD FIRST, accompanied the PEACE project as a very fair and supporting head over the years (from SUPRENUM to MANNA and beyond). Thanks for your support, especially during the bad days of the project.

Vinny Cahill and Ellen Siegel helped me significantly improve this work. Besides making the manuscript readable not only for a German, but also Briton and American, they made a number of highly constructive remarks concerning the technical merit of the written material. Vinny, thanks for your help during the hot phase of the book project. Your comments have not been too negative. Ellen, thanks for your very constructive comments on the early manuscript of this work and discussions on PEACE.

PEACE would be unthinkable without the effort of a number of students and research associates at GMD FIRST. It took about 100 man-years of software development to make PEACE an object-oriented, family-based parallel operating system and port it to several distributed-memory machines. The main "PEACE-makers" were Ralph Berg, Katja Bienert, Lars Büttner, Jörg Cordsen, Arnulf Günther, Thomas Garnatz, Anne Gerischer, Jörg Heuer, Rüdiger Krahl, Marion Krauser, Nicolai Leymann, Andreas Luther, Eva Luther, Reinhard Meyer, Jörg Nolte, Bernd Oestmann, Michael Sander, Henning Schmidt, Friedrich Schön, Gerd Truschinski, Matthias Wilhelmi, and André Zehl. They all made PEACE a success story. Thanks for your patience with the project leader. Especially, Jörg "the Dane," Arnulf, "Stud" Thomas, Rüdiger, "Magic" Nolte, "Stud-keeper" Bernd, Michael, and Matthias did outstanding work, and therefore must receive my deepest acknowledgment.

Ulrich Büning and his hardware crew created terrific platforms and systems on top of which PEACE was able to mature into a true parallel operating system. The close cooperation with these highly qualified and very ambitious hardware developers was one of my greatest experiences in computer science so far. Ulrich, you and your group did a very great job.

Most of the material was produced during my research stay at the ICSI in Berkeley. It is safe to say that this work would not exist if I had not had the opportunity to escape from GMD FIRST (i.e., the daily life of exacting project work) for a while. California, the Bay Area, Berkeley and the ICSI are nice places to relax and launch a book project. Thanks to Jerry Feldman, Joachim Beer and Domenico Ferrari for your support during the six month at ICSI.

Last but not least, many thanks to my wife Petra and daughter Sophie. Without your love I would have been unable to make PEACE.

Chapter 1

Introduction

The design and development of parallel operating systems is quite a new discipline when compared to that of classical operating systems. At the end of the '80s, this discipline increased in importance with the breakthrough of parallel computer systems based on distributed-memory architectures. During the '90s, in order to fulfill the increasing demands of high-performance parallel computing, parallel operating systems will be even more important in the realm of massively parallel computer architectures.

In the past, parallel computing was dominated by two areas: hardware and (numerical) algorithms. The presence of distributed-memory parallel computers raised big challenges not only in these two areas but also in compiler technology. The emphasis in compiler development was—and still is—on automatic parallelization and data partitioning. Closely related to the compiler branch, the demand for new programming models became evident. Nevertheless, the two areas mentioned above still play the dominating role in the first half of the '90s—operating-system issues are hardly ever addressed.

> "A widespread misconception is that the two most important parts of the high performance field are architecture and algorithm [83]."

Since the introduction of parallel machines, there has seemingly been no energetic demand to provide specifically designed system software support other than compiler and runtime environments. The approach was (and is) to bypass the operating system—still further, to sacrifice the operating system. There was little enthusiasm for thinking about means by which an operating system for these machines could be designed such that bypassing the operating system never allows the integrity of the kernel and the application tasks to be violated, not to mention discovering operating-system designs with the view that bypassing the operating system could be a possible but never mandatory feature. In the meantime, this attitude changed. It is very well accepted now that forthcoming (massively) parallel systems must offer a complete solution to achieving high performance, programmability, and supporting availability and management. Modern computer architecture research, thus, means seeking for means by which "seamless transitions" between the various hardware, firmware, and software layers of the system are established.

Massively parallel systems consist of hundreds or thousands of cooperating processing nodes. The preferred paradigm for constructing parallel machines on that scale relies on distributed memory. At the hardware level, the view of a common global memory is sacrificed. This view may be reintroduced by software providing a virtually shared-memory space. A very-high-speed message-passing network serves as the "backbone" for interconnecting the nodes. Every node operates more or less autonomously. A *multiple-instruction/multiple-data* (MIMD) operating principle is established.

Almost every new (commercial) operating system that appears on the market is based on a microkernel architecture [30]. The most recent example of this is WINDOWS-NT^{TM} [24].[1] At first sight, this seems to suggest exploiting a microkernel as the common platform on top of which the parallel operating system should be built. It is quite feasible to port even microkernel-based UNIX^{TM} systems on distributed-memory parallel computers [97], with every node executing the microkernel, a subset of system servers, and at least one application task. However, the question arises as to whether this really is the right ap-

[1] WINDOWS-NT is the largest microkernel ever built. It is hard, if not impossible, to make it understandable that microkernel technology results in a highly modular system structure when the platform proves to be all but "micro." This holds for all microkernel-based systems currently on the market.

proach. Adapting portable operating systems to parallel machines and providing specific, portable operating-system support for parallel computing on these machines are two different matters. The adaptation of an existing operating system for use on a multinode parallel machine may fail to "*provide abstractions that are appropriate for certain applications*" and "*make efficient use of the hardware*" [81].

Very low latency and very high efficiency are still the predominant user requirements that must be fulfilled by a parallel operating system. Traditionally, parallel applications call for extremely high performance of both hardware and software. While in the past shared-memory architectures played the major role, it is now very well accepted that massively parallel computers are *tightly coupled distributed systems*. In order to improve programmability, and thus not only user acceptance but also maintainability, *transparency* [39] must not be sacrificed. However, this can be achieved only at the cost of *efficiency*. Transparency means to hide the distributed-memory organization from parallel applications, provide support for fault tolerance, make services globally available, and abstract from the actual number of nodes available for parallel processing. Efficiency means mainly reducing the message startup time for a given application to an absolute minimum. The former implies system software overhead, whereas the latter demands that this overhead generally be avoided—a "vicious circle" that parallel operating systems are required to break.

As a solution to this problem, two major aspects will be discussed. First, the design of a parallel operating system should lead to a *program family* [67], and second, the implementation should follow the paradigm of *object orientation* [91]. From the software-engineering point of view, the microkernel approach [30] is not appropriate for building a platform for high-performance parallel computing. Microkernels typically provide multitasking, and thus introduce not only unneeded functions but also overhead for many parallel applications which require only single-tasking support. Instead of microkernels, family-oriented software architectures are preferred. Applied carefully, the program family concept also results in a highly modular structure for microkernels and, beyond that, distributed operating systems; thus, it helps to overcome the complexity (and software maintenance) problem of operating systems in general. In this sense, the design principles employed to construct parallel operating systems also bridge the gap between distributed systems and parallel systems. These principles, therefore, will allow one to transfer concepts from distributed computing to the parallel computing area, and vice versa.

1.1 About This Book

This work describes the logical design of state-of-the-art parallel operating systems that have to meet the needs of both massively parallel computer architectures and massively parallel applications. The focus is on how to organize and structure these operating systems and not a discussion of algorithms that implement specific system features. Thus, it is a software engineering course explaining operating-system structuring rather than implementation.

PEACE [79] is used as a case study in order to clarify the various design aspects considered. The object-oriented implementation of a family of parallel operating systems is described. Related state-of-the-art designs covering these areas will be discussed. Every chapter concludes with a short summary. The chapter overview is as follows:

Chapter 1 gives the motivation for a family of object-oriented parallel operating systems. It serves as an *introduction* to the problem space.

Chapter 2 describes the *functional decomposition* of a system that optimally supports parallel computers based on distributed memory. It presents the organization of PEACE and discusses the functional hierarchy of the building blocks of the system. The design of the communication system is presented, followed by the refinement of those parts of the minimal basis that have to deal with network-wide data transfer.

Chapter 3 considers *system composition* for specific environments. The most significant configurations of PEACE for parallel, distributed computer systems are discussed.

Chapter 4 discusses the *operating principle* of the designed system. It describes the organization of the interrupt-handling subsystems and the synchronization of asynchronous events. The protocol interactions carried out by the network subsystem are explained. At the end of this chapter, active messages and their approximation by means of reactive objects are treated extensively.

Chapter 5 deals with the adaptability of object-oriented systems. System services are understood as *transient objects*. These objects are dynamically integrated into the system as requested and removed from the system when no longer needed. Interactions between the remote object invocation layer and the naming system to support transient objects are illustrated by means of PEACE.

Chapter 6 introduces the *class ensemble* used to implement object-oriented runtime environments for massively parallel systems. It demonstrates the degree of object orientation in PEACE by discussing the structure of the inheritance trees of the kernel implementation. A very technical presentation is made.

Chapter 7 presents application-oriented *system abstractions* by employing the class ensembles introduced before. This chapter discusses the object-oriented realization of a fundamental execution model for supporting parallel processing in the realm of distributed-memory machines.

Chapter 8 contains the concluding *discussion* and makes an analysis of the design decisions made. Operating-system concepts based on microkernels are compared to the design principles incorporated in PEACE. In addition, the influence of C++ on the design and implementation of the PEACE system is briefly discussed.

As far as possible, the chapters are self-contained. Cross-references are given where appropriate. Thus, depending on their interests, readers are not obliged to follow the order of chapters listed above. However, in order to get a complete impression of the design concepts (and of PEACE) and how these concepts have been applied in the realm of object orientation, adhering to the sequence of chapters presented in the overview is highly recommended.

The glossary contains definitions of terms used throughout this book. Sometimes, the meaning of these terms may be highly context-sensitive relative to the topics discussed in the book. This may lead to ambiguities and misunderstandings, especially if the reader is not familiar with distributed/parallel operating systems. Whenever a term seems to be misplaced in the particular context of the text, the glossary should be called on to get the right picture. The hope is that following the text will be quite possible without working through the glossary again and again—or, most desirably, without noticing the glossary at all.

In order to introduce the reader to the subject of system software construction for massively parallel systems in more detail, the characteristics of a parallel operating system are discussed first. After that, two of the most crucial design constraints are presented, followed by a brief overview of the design principles used to construct parallel operating systems. A short presentation of the case study system used to exemplify the concepts is given.

1.2 Parallel Operating Systems

Compared to distributed operating systems, the literature on parallel operating systems appears to be fairly modest. When taking a retrospective view over the past decade in the operating systems community, a thorough discussion on the many facets of parallel systems cannot be found. Most of the work, if any, was dedicated to classical shared-memory parallel computers. The modern approach in parallel computing, namely to rely on distributed-memory parallel computers, was almost not covered.

In order to understand the notion and highlight the peculiarities of a parallel operating system, a brief excursion into the areas of shared-memory multiprocessor systems, distributed-memory systems, and massively parallel systems is made. The goal is to try to answer the question of whether distributed (microkernel-based) operating systems are adequate for distributed-memory parallel computers or specific system software structures are required. If the latter is the case, how much will these structures differ from or have in common with distributed (microkernel-based) operating systems? The intent is to elaborate the requirements for an operating-system structure whose implementation need not be bypassed in order to achieve high performance. The section concludes with a brief discussion on the current state of the art in (commercially available) system software platforms for distributed-memory parallel computer architectures.

1.2.1 Shared-Memory Systems

For many years, parallel computing was closely related to *shared-memory multiprocessor systems*. This understanding was also a dominating issue in operating systems development over the past decade.[2] Multiprocessor operating systems like *Mach* [96] can be an appropriate basis for shared-memory architectures. But even here, there are limits when the number of processors increases significantly. One of the problems, for instance, in trying to bring *Mach* onto the 64 nodes of the RP3 shared-memory multiprocessor [13] was the incredibly high system initialization time due to synchronization patterns inside the kernel. Primarily, this proved to be a kernel-structuring problem. The arrangement and protection of critical sections were proper for small-scalable multiprocessor systems, but showed limits in an environment of only tens of processors.

In shared-memory systems, the granularity of program units that can be executed in parallel is determined by the performance of the process model. The most important aspects here are the overheads for process creation, synchronization, and switching. The vertical interactions between the (non-privileged) application process and the (privileged) operating-system kernel are always crucial points. These interactions are extremely overhead-prone when compared to local program activi-

[2] For example, refer to the *Symposium on Operating Systems Principles* (SOSP), which happens every two years. Since 1981, the conference proceedings (*ACM SIGOPS Review*, Vol. 15-25, No. 5) have failed to see sufficient contributions on operating-system structures for distributed-memory parallel computers. This is astonishing, as these types of computers were evolving, and achieving their breakthrough, exactly in this period. Some topics partly addressing parallel operating systems were dealt with, especially in Vol. 25. However, the majority of papers were on the topic of "standard" (small-scale) shared-memory architectures only.

ties.

Most recently, approaches have been aimed at reducing kernel interactions to an absolute minimum. An example is the *virtual processor* model of *Psyche* [54]. A problem-oriented process model implemented as a library package is the coupling between the application and operating-system kernel. Assignment of processes to virtual processors happens without intervention of the underlying kernel. As was demonstrated with *Topaz* [1] and *Mach* [26], comparable models cannot be realized without making changes in fundamental kernel components. Nevertheless, completely bypassing the kernel is not feasible, even with the *Psyche* approach. Although it is no longer expensive to perform a process switch within the same application context, it now becomes expensive to initiate rescheduling of one of the user-level thread schedulers due to some external event. These events are still handled first by the kernel and, occasionally, propagated via upcalls to the user level. The upcall, however, must cross protection boundaries.

The PANDA *pico-kernel* "provides only those functions that are relevant with respect to protection and monopolization" [4]. All other operating-system functions have been moved to user space. The hardware abstraction provided by the pico-kernel deal with *protection domains* and *virtual processors*. In this sense, PANDA is somewhat closely related to *Psyche*. A user-level thread package running on top of the pico-kernel enables the user to write highly efficient nonsequential programs. The strength of PANDA is its object-oriented implementation, which allows for user customization of fundamental system abstractions. Nevertheless, there are two properties that make the PANDA approach not suitable for all kinds of parallel applications: the pico-kernel can be accessed only via traps; and address space protection is a fundamental capability. These are features the pico-kernel and the microkernel have in common. Since parallel applications are often run in single-tasking mode on a node, both features may cause an increase of the message startup time due to the artificial boundary between the user level and pico-kernel.

The performance bottleneck introduced by this protection boundary can be mitigated only by means of specifically designed hardware that enables the user program to access the network device without kernel intervention. However, the question arises as to whether the user/kernel boundary is mandatory if the tasks of the application program are mapped in a one-to-one correspondence with the nodes. For some, not all, parallel applications a protection boundary between the user and kernel levels is required. For some, not all, parallel machines, this boundary does not impair the message startup time.

Except PANDA, which uses a pico-kernel, all these examples are based on a microkernel architecture. However, this fact alone does not cause the performance bottleneck. Even in a monolithic system organization the same problem arises—it is omnipresent.[3] Both, monolithic and microkernel-based operating-system architectures have in common that user and kernel space are physically isolated from each other. This also holds for the pico-kernel approach. The distinction between user and kernel space is necessary in traditional multitasking and timesharing systems, but is not a predominant requirement in the context of (massively) parallel computing. As discussed later, neither a monolithic nor a microkernel approach is promoted by the design, even though specifically designed features are provided that help to bypass the microkernel or operating system. These features generally increase kernel complexity. Large kernels, however, are stumbling blocks in bootstrapping a massively parallel system.

A well-known fact is that shared-memory computer architectures are no longer able to keep pace with the dramatically increasing performance requirements of parallel applications. The design of large scalable shared-memory multiprocessor systems requires tremendous efforts to construct a fast low-latency memory system.[4] This does not only result in a system that comes very close to the limits of technical feasibility, but also becomes prohibitively expensive. Even then, one only gains a very moderate improvement in sustained system performance.

[3] Experiments have shown that user-mode processing of digital continuous media applications on top of SunOS 4.1, which is indeed a monolithic system, becomes difficult if not impossible. In this particular case, performance was acceptable only when kernel changes were made, providing scheduling and inter-process communication mechanisms that help to reduce interactions between the user and kernel modes [35].

[4] The construction of a bottleneck-free memory system is not possible, unless one goes to distributed shared-memory architectures such as DASH [46]. However, even the latter are scalable only up to a few hundred nodes, because of the $O(n^2)$ increase in directory memory [32].

A fairly bad cost/performance ratio is the outcome. The scalability of shared-memory architectures is limited by the current state of the art in chip technology and is not expected to increase significantly in order to meet the needs of "Tera-FLOPS computing" and beyond. In addition to that, the *von Neumann bottleneck* [5] remains, and becomes more and more severe. Well-established computer manufacturers (e.g., Cray and Thinking Machines) identified this problem and decided to go the alternative way, namely, to bet on distributed-memory architectures. Faced with this situation, parallel operating-system design must not place the focus on "old-fashioned" shared-memory computer architectures—although the design must not completely ignore these architectures.

1.2.2 Distributed-Memory Systems

As was mentioned above, distributed-memory parallel computers are distributed systems. This is especially true from the hardware point of view, but does not hold entirely for the software that runs on this hardware. Parallel applications are often of the classical shared-memory type, having merely been adapted to and not specifically designed for distributed-memory hardware. They have not lost their shared-memory (i.e., von Neumann) characteristics just by being executed on a distributed-memory machine.

In order not to be misunderstood, of course, there are parallel applications that have been designed and implemented for distributed-memory architectures and, therefore, have little in common with the shared-memory paradigm. However, these applications are still the minority, not only because of the lack of proper programming models, but also because of the volume of Fortran programs that have been written by the scientific computing community in the past 30 years.

This situation is one reason why "shared virtual memory systems" [47], and all their variants, have become important in recent years. However, the approach of implementing a shared-memory virtual machine on top of a distributed-memory platform actually means reinventing the "*von Neumann bottleneck*" for a machine that exhibits nothing of the kind. The idea of a virtual, global common address space cannot be considered as an ultimate goal, but rather as an approach to provide a "gentle shift" from old-fashioned shared to modern distributed-memory parallel architectures.

Providing means by which existing shared-memory parallel applications can take advantage of a distributed-memory architecture was also one of the main aspects in the design and development of the DASH architecture [38]. The DASH project aimed at developing a scalable hardware implementation of a shared but physically distributed address space. In order to exploit the full power of these distributed-memory machines, MIMD-style programming and appropriate software development tools are still indispensable. The fundamental problem is to (explicitly/implicitly) define data localities in order to reduce the global interaction among the processing nodes (i.e., local memories) to an absolute minimum. Table 1.1 (taken from [38]) compares typical costs of local and remote memory accesses by considering various system configurations. The numbers give an idea of the granularity of either messages or parallel executable program units required in order to obtain positive speedup. They also point out that pure software solutions to transparently perform remote memory accesses implemented by the operating system will be confronted with message startup time limits that are expected to be in the range of 20 to 200 μs.[5]

Implementation	*Memory Access*	*Processor Cycles*
hardware	cache	1
	local	10–40
	remote	50–100
operating system	remote	1000–10000

Table 1.1: Local vs. remote memory access times

The parameters shown are confirmed, and made more precise, by [2] referring to the penalties of even scaled-down software solutions in order to access the network interface of state-of-the-art message-passing architectures. The average number of instructions for a network access is $\geq$ 100 for the Intel Paragon (i860-based) and $\sim$ 50 for the CM-5

[5]Assuming 50 MIPS processor speed and (state-of-the-art) RISC technology.

(SPARC-based). This amounts to a low-level setup overhead of a few microseconds. That is, at the lower end of the spectrum there is one order of magnitude startup time discrepancy when one compares operating systems to scaled-down software (i.e., runtime environment) solutions. At the higher end of the spectrum, a discrepancy of two orders of magnitude can be observed.

Parallel operating systems are somewhat closely related to distributed operating systems. If there is a certain similarity, the question arises as to why operating systems for distributed-memory computer architectures are not very well recognized so far. A satisfactory answer cannot be given. It seems that the proximity of distributed-memory architectures to distributed systems leads to the assumption that distributed operating systems are thus easily adaptable to, and practically fulfill the user demands for, these specific architectures. The fact that *Chorus* [71], for example, has been (partly) ported to Transputer^{TM} systems does not imply, however, that it also adequately supports the parallel applications for this type of parallel machine.

In order to improve system performance, major redesign aspects may be necessary. For instance, the operating-system designer must be aware of the fact that, in order to improve performance, quite a large number of parallel applications are configured with respect to the number of processing nodes provided by the hardware. A number of applications accommodate their degree of parallelism to the degree of parallelism provided by the hardware. Although this does not hold for all types of parallel applications, it holds for the majority. Hence, the operating system is not in all cases required to multiplex a single node between several tasks or even threads of control. Standard distributed operating systems, however, are supposed to do so.

The problem is not that operating-system kernels implementing only multitasking support should eventually be unable to process "well-shaped" parallel applications whose tasks can be mapped in one-to-one correspondence with the nodes. This is beyond question, particularly for microkernel-based systems. However, all the microkernels presently available on the market fail to efficiently support the above-mentioned type of parallel application: considering bidirectional communication (i.e., send-receive-reply patterns), 74% pure multitasking overhead was measured in a microkernel specifically designed for a distributed-memory parallel computer [78]. The main problem is the artificial boundary between user and kernel. This boundary is due to the microkernel approach and is not necessarily demanded by the parallel application. The boundary makes network access from user mode extremely heavyweight, even if the hardware supports direct network access and thus allows the kernel to be bypassed when global communication has to take place.

Overloading processing nodes with unneeded microkernel functions does not only waste local memory resources but also limits scalability in general. OSF/1 AD, the microkernel-based parallel operating system for the Intel Paragon, gives an example of this circumstance. In many installations, the per-node memory consumption of OSF/1 AD is about seven to eight megabytes [27]. Thus, in a standard 16-megabytes node configuration, about half of the memory resources are confiscated by the operating system. Although OSF/1 AD implements virtual memory on every node, the user tries to partition the application program (i.e., data) such that it fits entirely into physical memory. Typically, the designer of numerical (parallel) programs is not willing to pay for the paging overhead of virtual memory systems, and therefore tries to arrange the programs in a way that minimizes, or even avoids, paging. Occasionally, program partitioning is done by the compiler.

The effect of the before-mentioned use pattern of parallel machines is that virtual memory degenerates to an unneeded microkernel function.[6] This fact was and is true for contemporary vector machines a la Cray and holds for (distributed-memory) parallel computers a la Intel.[7] Consequently, the modules implementing virtual memory intro-

[6]Almost the same applies to multitasking support.

[7]Virtual memory on the processing nodes is in almost every case sacrificed, not only for technical reasons (such as the lack of a memory management unit). Data parallel programming of distributed-memory machines typically makes very large local address spaces in many cases superfluous. Moreover, an "unlimited" local address space is in contradiction to the MIMD principle, namely, to avoid the *von Neumann bottleneck* to the memory. Note that traditional virtual memory has little in common with virtual shared memory [47]. The latter was not introduced to solve the problem of memory overallocation, but rather to present a view of a global, common address space that is physically distributed. In particular, for performance reasons, this view can (and should) be implemented without relying on paging but rather on compiler and runtime system support.

duce nothing but overhead. The overhead is twofold. On one hand, memory space is wasted due to unneeded code and data fragments. On the other hand, the message startup time is increased due to lower-level (microkernel) functions which assume that the message transfer takes place out of or into the virtual memory and are unaware of the specifically tailored user program that does not require the presence of virtual memory. For example, if the network hardware interface is able to perform the data transfer only from/to the physical address space, the driver will have to translate a logical memory address (pointing to the user-level message buffer) into a physical memory address. Depending on the underlying processor, this translation procedure may become a time-consuming task and thus introduces a communication performance bottleneck.

Unnecessarily, the scalability of user application programs for the Intel Paragon is limited mainly by the memory-consuming OSF/1 AD operating system, less by the amount of physical memory available on every node. Simply increasing the capacity of the physical memory of every node is not the solution. Besides scalability issues, compared with shared-memory parallel machines, the attractiveness of distributed-memory parallel machines lies in the favorable cost/performance ratio of a single node. More physical memory simply increases the hardware costs of a single node, and thus of the entire multinode machine.[8] As opposed to this, one can observe that the majority of parallel applications for distributed-memory machines get on with an operating system that exhibits a per-node memory consumption of about 50–500 kilobytes, even in the case of RISC-based system architectures such as the Intel Paragon. Thus, in its present shape, OSF/1 AD is entirely oversized for many parallel applications.

[8] For example, upgrading the per-node memory from 16 to 64 megabytes does not necessarily imply that the hardware costs are only quadrupled [12]. It depends on the actual state of a new-generation memory chip and how many chip providers of a new chip technology are available. Using prototype chips, the costs for 64 megabytes memory may be about 10 times as high as the 16 megabytes configuration. The cost factor may be about 5–6 if one can fall back on only one or two providers of the new technology. In the case of incompatibilities between the old and new chip technology employed, a complete redesign of the memory subsystem will be the consequence. This may increase the hardware costs more than a factor of 10.

Similar problems arise when considering the communication system, especially communication protocols. A distributed-memory system is usually a tightly coupled distributed system based on a very specific network architecture with high-quality data transmission capabilities. During data transmission over the network, one can expect a bit error rate that is not significantly worse than the bit error rate in local memory-to-memory transfers. A number of communication protocol features that are indispensable for classical distributed systems tend to become irrelevant. This is especially true when taking into account the specific type of application to be supported by the system.

The operating principles of the parallel application program play a significant role in understanding which functions should and should not be provided by a communication system [72]. For instance, recovering from the potential loss of packets must not always be handled by the communication protocol, since a numerical application may implicitly perform error corrections due to its underlying algorithm. A lost packet or message might simply result in a rounding error, which has to be covered by the (numerical) application anyway. Moreover, a missing intermediate result of an iterating numerical computation may be of no consequence for the final result. The produced error "dies out" in the iteration process. Similarly, it may be irrelevant to lose a single picture out of a sequence of pictures generated by an application performing computer animation.[9] All these examples demonstrate that the need for specific communication protocol capabilities depends on the type of parallel applications, not only on the type of computer architecture being used.

User-level (end-to-end) communication in

[9] A brief excursion into the "classical" communication protocol sector underpins this situation. For example, continuous multimedia applications, especially the video branch, are often hampered by data security properties of lower-level communication protocols or communication hardware. At these levels, checksum generation/verification in the course of transferring video data are unnecessary functions and only reduce performance (i.e., continuity). Once disregarding the rareness of bit flaws in state-of-the-art network technology, the most crucial point is that the transport system will drop packets that have not passed the checksum test successfully. These packets, however, may be meaningful for further processing of the video data stream. A single bit error will cause packet retransmission, although this error will have absolutely no impact on the final video quality.

the context of distributed-memory systems may be fairly different, not to say strange, when compared to the communication patterns known from the distributed systems arena. A brief example elucidates this. The most recent approach, so-called *active messages* [90], assumes that each message packet carries as header information the address of a remote handler routine, which gets immediately called when the packet arrives. This handler is furnished with the message body as parameter and serves to integrate the received information into the ongoing computation being carried out by its task. That is, the sender task encodes within the message not only the destination segment address but also a receiver task code address, and directs the communication system to continue processing at this user-level address upon message receive.

The active-message approach works well for distributed-memory parallel machines and is one of the most encouraging contributions in trying to overcome the message startup time problem. It assumes that the distributed application forms a single program, with identical code on every node required by the computation. In that case, the application is of the *single-program/multiple-data* (SPMD) type. Thus, the remote code address is actually drawn from a local identical copy of the message handler. This fact limits applicability, since not every parallel/distributed application corresponds to the SPMD model. Software extensions are necessary to make active messages suitable even for distributed programming.

The purpose of the discussion was to show that communication paradigms and protocols for distributed-memory parallel systems may have little in common with approaches discovered in the "classical" distributed systems area. Nevertheless, this does not necessarily mean totally abandoning communication principles known from that area. It might be appropriate to rely on transport protocol functions such as message segmentation and reassembly, flow control, and sequencing, for example, to reduce the potential of blocking delays within a circuit-switching network and support more sophisticated parallel applications. In particular, real-time communication protocols [29], presently being developed to meet the needs of high-speed and high-bandwidth multimedia applications, may also evolve as a solid basis for interconnecting the nodes in a distributed-memory computer architecture. Adhering to at least some of the principles of real-time communication may be sensible for improving message throughput. However, the system design must reflect the fact that the need for these features depends on the network architecture and the structure (i.e., topology) of the parallel application. This, at least, is definite: instead of trying to present a single solution for all possible cases, it will be necessary to provide a *protocol suite* that offers individual solutions.

Although there are some significant differences from distributed operating systems, a parallel operating system must still take care of the management of a (possibly) very large number of nodes, be able to tolerate node failures, provide I/O services, enable virtualization of nodes by implementing a proper process and address space model, and ease the programming and debugging of parallel applications. Thus, it must provide many services a distributed operating system typically provides. However, the operating system should provide these services only on an *extend-on-use* basis and not at all times, as this will imply a potential performance bottleneck and an increase of system complexity.

1.2.3 Massively Parallel Systems

A system that consists of several thousands of processing nodes is usually called a massively parallel system. From the hardware point of view, a distributed-memory organization is the only chance to build these systems, and ultra-high-speed networking will play a major role in interconnecting the nodes. Global access to the local memories of the individual nodes is exclusively on a message-passing basis—preferably performed by hardware.

Programming massively parallel systems enters new dimensions of computational science in general. This does not only hold for application programs, but also for those system programs that implement resource management for such a machine. A multitude of nodes must be controlled, and a minimum of availability must be achieved. For portability reasons, specific details of how to access the network must be hidden from the applications. Semantic gaps must be bridged, for example, by raising the data transfer security provided by the hardware to a level demanded by the (application) software. The gaps are there, not only with respect to different communication semantics [8].

Site-transparent execution of application tasks must be supported, such that (static/ dynamic) load balancing becomes feasible. This calls for access transparency [60] to files, I/O devices, and other system services, but also to application processes. Bootstrapping the system and networkwide loading of processes must be supported. In this context, one of the major problems will be detecting the minimal functioning of the system. Centralized approaches are to be rejected. Last but not least, traditional system services are required, such as process, memory, and address space management.

In contrast to shared-memory systems, in distributed-memory architectures the granularity of program units that can be executed in parallel is determined by the performance of the process and communication model. The problem of designing a lightweight or even *featherweight process model* is almost the same as was discussed in the context of shared-memory systems (see Section 1.2.1). Since processes have to exchange messages in order to carry out some parallel computation, the process model will be significantly influenced by the communication model. Both models are somewhat interlocked with each other.

A *single-address-space* configuration is appropriate for a number of parallel applications, and it makes most of the discussion on user-level thread packages obsolete. This model assumes no artificial boundary between the user task and the message-passing kernel. It thus proves to be a sound basis that should be supported by a parallel operating system. In order to avoid misunderstandings, one is willing to sacrifice the separation between user and supervisor modes only for specific applications [78]. For those applications whose tasks can be mapped in one-to-one correspondence with the nodes, heavyweight interactions between user task and kernel are no longer given. For others, the user/supervisor mode boundary will be present to enable the sharing of a single node between the tasks and to introduce *scaling transparency*. These types of parallel applications are supported by providing a *multi-address-space* configuration. The optimal solution is to offer a system architecture suitable for either configuration.

For all these reasons and because of the lack of properly designed parallel operating systems, parallel computer manufacturers developed their own system software platforms. These platforms, however, cannot always be regarded as operating systems, but rather are runtime environments. Nevertheless, they were specifically developed for a certain distributed-memory parallel computer and for a certain set of applications. Examples are *CS/Tools* [55], *PARIX*[10] [68], and *UBIK* [88]. Except for *CS/Tools*, they all are systems developed for Transputer-based architectures. The *iPSC/2 hypercube*, with its 128 nodes, is controlled by *NX/2*, a parallel operating system providing virtual shared memory [48].

The operating system for the CM-5 [23], called CMost, is a SunOS variant. This variant, however, is only executed on the control processors. Processing nodes run a runtime executive and normally are subjected to a single-tasking mode of operation. Nevertheless, there is also support for multitasking whereby the control processor (i.e., CMost) preempts all tasks belonging to the same application remotely "at the same time." Multitasking is not an autonomous feature of the processing nodes.[11]

These systems have two things in common: highly efficient network-wide communication and poor functionality. They all suffer from a design concept that makes it impossible to scale up with the manifold demands of parallel applications. Similarly, state-of-the-art distributed (microkernel-based) operating systems suffer from a design concept that makes it impossible to scale down functionality properly. This is where an innovative parallel operating system has its niche. The design must support scalability in both directions. Functional enrichment must be possible. At the same time, it must be demonstrable to a user what the expense of functional enrichment in terms of the loss of MFLOPS or GFLOPS is. The design of parallel operating systems must provide a framework for the composition of application-oriented parallel computing platforms.

Distributed-memory (massively) parallel systems have left the spadework phase and have become serious competitors in the supercomputer market. However, it cannot be denied that these computer architectures exhibit the dichotomy of being both mature and imma-

[10] Currently, a new generation of *PARIX* is designed and implemented for the PowerPC processor family.

[11] This somewhat strange scheduling strategy is the result of providing direct user-level access onto the network hardware interface. However, the problem is not in granting the access but rather the lack of a properly designed hardware interface that supports multitasking on the node and enables user-level access.

ture. When compared to the state of the art in parallel software developments, the hardware branch is quite mature and advances more rapidly, while the software shows a very large backlog. Moreover, "spiteful tongues" say that the programming of distributed-memory machines is at a level at which computer programming was 30 years ago—they equate state-of-the-art programming models and languages with assembly language programming of the '60s. This might be somewhat exaggerated, but somehow shows the sore point. Manufacturers are aware of this situation and understand that merely providing runtime systems is by far not sufficient to make their products work. They call for proper operating-system structures that are as lightweight as their runtime environments and, at the same time, aid *programmability*, *efficiency*, *availability*, *configurability*, and *extensibility* (PEACE).

1.3 Design Constraints

Parallel computing defines its own "book of rules," even if the parallel computer architecture utilized is a distributed system. No matter which principle is used for designing a parallel operating system, the resulting implementation must *never impede performance*. One of the most crucial performance-limiting factors is embodied in the communication system and addresses the message startup time problem. Another important factor is the complexity, and thus immense difficulty, of programming massively parallel systems. This calls for new programming models and still involves large efforts in theory, language design, and compilation techniques. It is well known that programming these systems must be "*liberated from the von Neumann style*" [5], but there are still no satisfactory programming models available. Compared to that, the operating system can only provide a very modest contribution to the challenge of coping with the complexity of a machine consisting of thousands of interconnected nodes.

1.3.1 Message Startup Time

The communication performance problem in massively parallel computer architectures is dominated by the *message startup time*. Basically, the message startup time determines how many processor cycles are lost for the application programs by issuing a request for remote inter-process communication. The loss of "number-crunching" power caused is not primarily incurred by the communication protocol overhead, but rather is a question of the actual operating mode of the node [78]. A single-tasking mode of operation does not imply any form of address space isolation. There is no need to protect either the tasks (since there is only one per node) or the kernel (since it only keeps track of the resources of a single task). Thus, the kernel is nothing but a communication library, sharing the same address space with the application task. This organization results in smaller code sequences for initiating a message transfer request. The consequence is faster message startup when compared to a kernel supporting contemporary multitasking. This is one of the problems that arise with the attempt to utilize state-of-the-art microkernels for distributed-memory parallel machines.

The choice of the proper operating mode depends on constraints defined by the application and the kernel architecture. There are a number of parallel applications that scale very well with the actual number of nodes. These applications call for single-tasking support on the nodes. The microkernel architecture promotes the encapsulation of system services by user-mode tasks. This approach calls for multitasking support on the nodes even if only a single application task must be locally processed. In other words, the microkernel is the proper platform only for those applications that call for a many-to-one correspondence between tasks and nodes.

Another performance-critical factor is the *communication latency* (also termed *message latency*). The communication latency determines the time for either transmitting or receiving and delivering a message transfer request, and includes the message startup time. Latency is caused by all "vertical activities" needed at the source and destination node to send, receive, and deliver a message request (not necessarily the message contents) to a process, and all "horizontal activities" between the nodes needed to execute the communication protocol. That is, communication latency is the product of *node latency* and *network latency*. In massively parallel systems, node latency is caused primarily by software overhead (i.e., by the message-passing kernel), while network latency depends on the hardware (i.e., the interconnection network). Table 1.2[12,13] gives an impression of the message latency mainly produced by the operating system of typical distributed-memory architectures. The timings shown stand for one-way communication.

Machine	*Operating System*	*Latency (μs)*
iPSC	NX	1100
SUPRENUM	PEACE	392
iPSC/2	NX	350
Paragon XP/S	OSF/1 AD	240
nCUBE/2	Vertex	160
nCUBE/2	PUMA	110
iPSC/860	NX	100
CM-5	CMOST	65
Paragon†	PUMA	50
nCUBE/2	Active Messages	32
Paragon‡	PUMA	30
MANNA†	PEACE	23

† 1 CPU, ‡ 2 CPU node configuration.

Table 1.2: Dimensions of message latency

It is worth noting that not all the systems listed can be directly compared to each other. The communication functions and their semantics are often extremely different. Table 1.2 draws a somewhat distorted picture, but nevertheless documents the message-passing performance limits of state-of-the-art parallel operating systems or runtime environments. In particular, the numbers given for the i860-based (dual-

[12] Most of the numbers were obtained from the `comp.parallel` news group and not by benchmarking. In particular, one author posted that the message latency of OSF/1 AD was benchmarked with 140 μs (half of the request-reply time with zero-length messages) and 130 μs (timing the send call only). The different results were not further discussed.

[13] *PUMA* supports two node configurations. The 30-μs timing is the result of having the second processor in charge of handling all message traffic. In contrast, the 50-μs timing is on the basis of a single-processor solution.

processor node) machines Paragon and MANNA[14] [33] show a notable factor of 8–10 in performance loss.[15] One of the Paragon system configurations used an OSF/1 (i.e., *Mach*) port, while the other three i860-based configurations relied on specifically designed parallel operating systems (*PUMA* [92] and PEACE, respectively). This timing discrepancy cannot be explained only by the different communication models implemented by the investigated operating systems. Rather, the microkernel architecture (and implementation) of OSF/1 AD is certainly an additional—and possibly the dominating—performance-limiting factor.

The way parallel programs are organized and implemented determines whether only the message startup time or the message latency as well is crucial for the overall performance. For example, multithreading may be employed by the programmer to perform *latency hiding*. In this case, the message startup time remains the only performance-critical factor, and basically defines the latency before a ready-to-run thread different from the communicating thread can be started. The communication thread is blocked in a message-passing operation, causing the thread scheduler and/or dispatcher to automatically select another thread for further program execution. This model strongly supports the overlapping of computation and communication. The time until the thread is blocked is mainly the message startup time. With the scalar performance of a 50-MIPS processor (RISC technology) in mind, this time must be below 10 μs. As simulations of a number of parallel numerical algorithms revealed, only then is a balanced ratio between the per-node computing power and network performance achieved [56].

In order to speed up communication, functionally dedicated hardware was developed, such as the Transputer [49] or the *iWarp* [11]. The Transputer performs well when it is exclusively allocated to the user application. As soon as the processor must be shared by application and operating-system tasks (i.e., threads), an extremely overhead-prone system software architecture is the outcome. This holds for all

[14]MANNA stands for "*Massively-parallel Architecture for Numerical and Non-numerical Applications.*" The project was funded by the Ministry of Research and Technology (BMFT) of the German Federal Government, grant no. ITR 9002 2.

[15]Referring to footnote 12, a factor of approximately 5–6 is given when using the alternate timing of 140 μs—which is still too large for a number of parallel applications.

types of Transputers developed so far. Similar considerations hold for the *iWarp*. Although this chip has a more sophisticated design, it does not support state-of-the-art operating systems technology. Both hardware designs are completely application-oriented and assume that no operating system but, at most, runtime system software need be executed by the processors. In contrast, the **T* processor (i.e., Motorola 88110MP) makes send and receive primitives machine instructions and strongly supports multithreaded address spaces [6]. These instructions are directly accessible from user mode, which decreases the startup time significantly. In order to ensure network (or message) integrity, *gang scheduling* is used to schedule all tasks simultaneously across the processors [6, 23]. This technique (i.e., the lack of a locally autonomous task scheduling) promotes only the processing of regular parallel applications in a single-user mode of operation. In addition, it makes it nearly impossible to exploit a not-fully-utilized parallel machine for the processing of operating-system services.

It becomes obvious that only specifically designed computer architectures will overcome the startup time problem in distributed-memory architectures. At the system software level, extremely lightweight operating-system structures are required—"*network bandwidth is rendered virtually insignificant*" [44]. Above all, the performance-limiting factor is the per-node operating-system software overhead. Taking into account state-of-the-art hardware technologies, it is not the limited network bandwidth that significantly increases the message startup time. This leads to the following conclusions:

1. A very careful operating-system design must be presented that, relative to the applications, reduces software overhead to an absolute minimum.

2. An extremely efficient implementation must be found that implements the design.

3. Only as the very last resort, one may argue that "*faster hosts are needed*" [44] to improve system performance—but it has to be kept in mind that this step does not really solve the message startup time problem, since the linear speedup effects all the software executed by the host (i.e., node).

Instead of calling for faster hosts, one must advance the development of innovative computer architectures, stating which hardware features are required to help software get faster [34]. However, before challenging new hardware developments, the software designer is asked to present a structure and implementation that helps to overcome, or even overcomes, the startup time problem in state-of-the-art parallel architectures. Furthermore, this structure should be designed under the goal of (possibly) directing future hardware developments.

1.3.2 System Complexity

Programming and management of thousands of interconnected nodes is a nontrivial task—and it does not get easier with millions of nodes. It is not only the problem of keeping track of all node resources and associating nodes with application tasks, but it also requires that a proper programming model exist, enabling the generation of many tasks or threads of control. Also, bootstrapping an operating system enters completely new dimensions. Getting an operating system instantaneously loaded over all the nodes is not only utopian, but will also significantly slow down the system startup time. As already mentioned, even in shared-memory multiprocessor systems, bootstrapping a fairly complex machine may become a critical issue (refer to Section 1.2.1). In addition to the difficulties arising from the complexity of the computer architecture, the main software problem is the complexity of the operating system and the procedure that must be executed during the initialization process.

The basic question is whether it is really mandatory to initially load and distribute the entire operating system over all the nodes. This is mainly a question regarding the organization of the operating system and the type of applications intended to be processed. Not every application demands the same set of system functions. Nevertheless, it still is a common approach to build operating systems that always provide a superset of services. These (monolithic) systems are not very well suited for large scalable parallel architectures. Operating systems are required that are capable of changing their shape with respect to the given set of applications. Even microkernels are not the ultimate approach, since the two examples of *Mach*[16] and *Chorus* demonstrate that the kernel itself is still far too complex. Rather, *dynamically alterable system structures* are needed.

Based on a dynamically alterable operating system, bootstrapping must only take care of the absolute minimal subset of system functions that need to be operable at any time. In addition, by relying on *incremental loading* features, the system will solely bootstrap a single node, and from here on continue bootstrapping of other nodes only on demand. Note that the nodes (and thus operating-system services) must only be in operation when they are needed by some application. In consequence, the application always takes responsibility (i.e., is the reason) for node bootstrapping. It turns out that, given an adequate operating-system structure, the majority of nodes in a massively parallel system need not be bootstrapped at all—incremental loading encompasses *incremental bootstrapping*. In such a context, *garbage collection* becomes indispensable as well. Just as system functions are loaded on demand, namely when invoked for the first time, they must also be deleted automatically when their presence is no longer required.

These considerations lead to a situation in which the emphasis in the system design must be placed on the ability to be almost arbitrarily configurable. It thus tends to model operating-system services by ("medium-grained") objects, trying to reduce the configuration problem to the question of how to generally map objects onto a parallel machine. Solutions to this problem, then, are very close to the solutions of problems arising in the course of mapping parallel applications onto the hardware. Load-balancing mechanisms are used to map not only applications but also the operating system. Mapping parameters and strategies are different. For example, the operating-system configuration needs to qualify objects as being site-dependent (such as disk drivers) or site-independent (such as file servers). The configuration of parallel applications, in contrast, has to keep track of nearest-neighbor relationships between the objects, for instance. In both cases, object attributes and mapping strategies are context-sensitive. What is com-

[16] Although often referred to as microkernel, *Mach* 2.5 is a monolithic operating system, with all main system functions executing in kernel mode. The OSF/1 version and *Mach* 3.0 are true microkernel-based systems.

mon is the mechanism, namely, to model program entities by means of objects.

Programming massively parallel systems then becomes mainly a configuration problem. This is true for both the user application and the parallel operating system. An object model used as the programming basis significantly eases the configuration task. In this sense, massively parallel systems and distributed systems share a common basis. Approaches stemming from the distributed systems area [43] become transferable to the area of massively parallel systems, namely, how to aid the mapping. Thus, the minimal requirement in the development of parallel operating systems is to design a structure that eases the configuration task. The operating system must show an actual representation that logically consists of a large number of "*transient objects*."

1.4 Design Principles

So far, the discussion has revealed that state-of-the-art distributed (microkernel-based) operating systems are still too heavyweight for distributed-memory (massively) parallel systems. Parallel operating systems must exhibit a lightweight or even *featherweight system structure* that is adaptable to the individual needs of both an application program and the hardware architecture.

The approach for solving the problems discussed in the previous sections is to understand a parallel operating system as a *program family* [67] and to use *object orientation* [91] as the fundamental implementation discipline. The former concept (program families) helps prevent the design of a monolithic system organization, while object orientation enables the efficient implementation of a highly modular system structure.

1.4.1 Program Families

The program family concept distinguishes between a *minimal subset of system functions*[17] and *minimal system extensions*. It does not dictate any particular implementation technique. The minimal subset of

[17]The notion of "*minimal basis*" will be used as synonym.

system functions defines a platform of fundamental abstractions serving to implement minimal system extensions. These extensions, then, are made on the basis of an *incremental system design* [36], with each new level being a new minimal basis (i.e., *abstract machine*) for additional higher-level system extensions. A true application-oriented system evolves, since extensions are only made on demand, namely, when needed to implement a specific system feature that supports a specific application. Design decisions are postponed as long as possible. In this process, system construction takes place bottom-up but is controlled in a top-down (application-driven) fashion.

Representing an operating system as a program family is always meaningful, even if only a paper design is made. A clean and highly modular system structure is the natural outcome of this procedure. Maintenance of an actual system is significantly improved. Moreover, the provided system components are always tuned to a specific application. These components introduce overhead only with respect to their attempt at mapping a specific application onto the lower-level (abstract) machine. They are system specializations for a specific application rather than any type of application. Thus, the overhead in the mapping process will be the same as if the mapping was done by the application itself.

In its last consequence, applications become the final system extensions. The traditional boundary between application and operating system disappears. The operating system extends into the application, and vice versa. An incremental system design also promotes a dynamically alterable system structure. Thus, this approach is the key to overcoming the bootstrap problem discussed earlier.

1.4.2 Object Orientation

Applying the family concept in the software design process leads to a highly modular structure. New system features are added to a given subset of system functions. Because of the strong analogy between the notions of "program family" and "object orientation" (Figure 1.1), it is almost natural to construct program families by using an object-oriented framework [21].

Both approaches are, in a certain sense, dual to each other (i.e.,

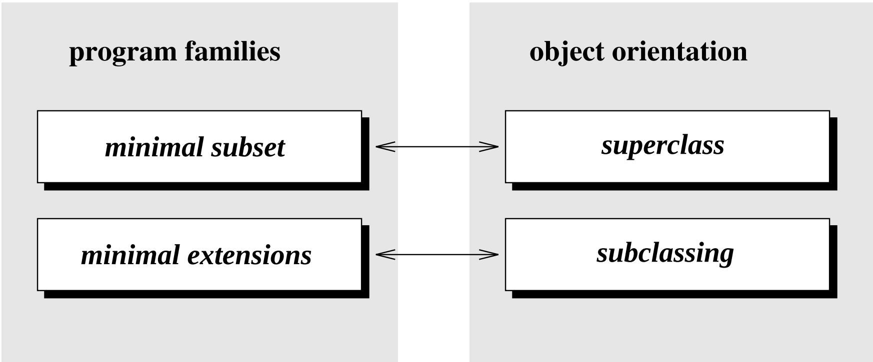

Figure 1.1: Program families vs. object orientation

exchangeable). The minimal subset of system functions in the program family concept has its counterpart in the superclass of the object-oriented approach. Minimal system extensions are thus introduced by means of subclassing. Inheritance and polymorphism are the proper mechanisms to allow different implementations of the same interface to coexist. Code reuse is significantly enhanced, increasing the commonalities of different family members:

> "We consider a set of programs to be a program family if they have so much in common that it pays to study their common aspects before looking at the aspects that differentiate them" [67].

For the development of parallel operating systems it is sensible to choose, above all, the program family concept as the fundamental design principle. Object orientation should primarily be used as an implementation, not a design instrument. Inheritance makes it much easier to implement and maintain an incremental system design, but is not mandatory for it.

Object orientation has many faces. In order to avoid misunderstandings, the "classical definition" [91] is employed.[18] This definition says

[18] A much more complete survey and fundamental discussion on object orienta-

that objects are instances of classes and that inheritance can be applied to build class hierarchies. Classes describe *abstract data types* [53], the composition of which happens through inheritance. Thus, object orientation means:

$$\textsf{abstraction} + \textsf{classes} + \textsf{inheritance}$$

Building hierarchies of classes also implies the specialization of their superclasses. Superclasses define fundamental object capabilities. Through inheritance, these classes are adapted to given application demands. In this way, specialized subclasses inherit and possibly extend the capabilities of their superclasses. Specialization differentiates between *single inheritance* and *multiple inheritance*. The former case always uses only a single superclass to compose the new level in the hierarchy (i.e., subclass), whereas the latter case does so with several superclasses [10].

Thus, a superclass may have many appearances, a circumstance that is described by the notion of *polymorphism*. Being virtually polymorphic is not only a capability of classes, but also of the methods defined by the classes. In this case, a superclass provides methods (functions or operations) that can be redefined by subclasses to achieve user customization. If instances (objects) of the specialized classes are created, *dynamical binding* (or *late binding*) takes place to associate the methods contained in subclasses with the superclass containing the original (deferred) definition. By applying the superclass scope, method invocation automatically leads to the invocation of the associated subclass method. This works transparently to the calling instance, namely, the object client.

An object-oriented system does not only consist of a very large number of objects. It must also provide "tunable" class abstractions. Above all, object orientation means *inheritance*, and thus specialization. Systems that do not show these properties from here on are not called object-oriented but class-based or object-based [10]. This is the basis for the following definition of an *object-oriented operating system*:

tion in operating systems can be found in [85]. The notion of "*extensibly object orientation*" comes very close to the idea of (dynamically alterable) object-oriented program families.

> An object-oriented operating system is designed and implemented in an object-oriented fashion. In addition, it offers an object-oriented interface by means of which operating-system abstractions can be specialized through inheritance to achieve user customization.

Because of their customizability, object-oriented operating systems are open and application-oriented. Distributed object-oriented operating systems must support inheritance networkwide. It does not suffice to use inheritance only internally to achieve an object-oriented design and implementation. Applications must be able to benefit from the reusability and adaptability of existing classes, even in a distributed environment. Thus, inheritance must have a global meaning and must bypass traditional operating system as well as node boundaries. Only very few of the existing (also often called object-oriented) operating systems fulfill these requirements.

The classification stated above intimates how neatly language, compiler, runtime system, and operating system must fit together to support object orientation networkwide. Inheritance must be applicable even in the case of crossing address space, node, and network boundaries. This classification is fundamental for this work.

1.5 PEACE

An example of the *family-oriented* approach in parallel operating system design is PEACE [79], the parallel operating system developed for SUPRENUM [31]. PEACE will be used as a case study in the course of exemplifying the various design approaches. Therefore, it seems proper to give a brief introduction to the development of this parallel operating system, presenting some background information.

Between 1986 and 1990, the SUPRENUM project[19] was conducted to design and develop a 256-node distributed-memory parallel computer aimed at supporting the execution of parallel numerical applications.

[19] SUPRENUM stands for "*Superrechner für numerische Anwendungen.*" The project was funded by the Ministry of Research and Technology (BMFT) of the German Federal Government, grant no. ITR 8502 A 2.

Every node was under the control of an instance of the same message-passing (micro-) kernel, with the entire multinode machine being run by an ensemble of server processes jointly constituting the distributed operating system [7].[20] A batch mode of operation was envisioned. The result was a distributed *process execution and communication environment* (PEACE) [76] attaching the multinode computer system to a UNIX front-end.

The PEACE development was faced with two major problems. The first problem was to build a new parallel computer system from scratch, concerning VLSI chip design and the development of hardware, firmware, and software (i.e., operating system, runtime system, compiler, tools, and numerical algorithms). Thus, SUPRENUM was an extremely ambitious project and covered almost every area of computer science. All these activities were more or less interlocked with each other. In particular, the operating-system development was requested to rely on hardware components that were still in the process of becoming stable—the non-stable operating system was compelled to fix hardware bugs. The second problem was to build a system that would fulfill extremely hard performance requirements: 500-μs message latency for user-level and networkwide message passing in a multitasking environment, based on nodes with relatively poor scalar performance (2 MIPS) compared to their vector performance (20 MFLOPS). Note that the performance of operating-system software usually depends on the scalar, not the vector, performance of a processor. The key to success for this development was to consider the operating system as a family of program modules and follow the idea of an incremental system design rather than build a "monolithic dinosaur."

Although the "genesis" of PEACE was in 1986, the history of development can be dated back as early as 1982 (Figure 1.2). In 1982 the development of an object-based (i.e., process-structured) operating-system family was started, based on a *message-oriented operating-system environment* (MOOSE). The outcome of this development was *Advanced uniX* (AX), a timesharing operating system that provided

[20] In reality, SUPRENUM is a 320-node computer, with 256 nodes dedicated to numerical processing and the additional 64 nodes dedicated to system administration purposes. Thus, the operating system was responsible for the management of the entire 320-node machine.

services of QNX^{TM} [70][21] and UNIX [89] by means of a program family. AX was the prototype implementation of the MOOSE concept of a dynamically alterable and process-oriented operating-system family [75]. MOOSE was the first approach aimed at extending the procedure-oriented idea of operating-system families, as followed by FAMOS [36] and DAS [73], into process-oriented environments. The fundamental process model to construct family members was taken from *Thoth* [15].

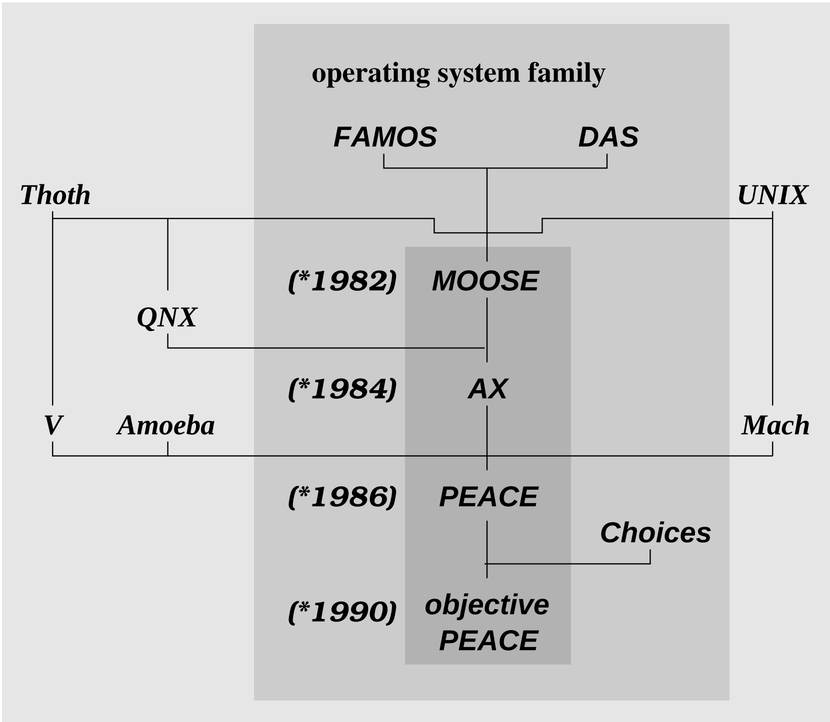

Figure 1.2: PEACE family tree

PEACE is the evolutionary successor of AX. The design strongly followed the idea of MOOSE (i.e., AX) and relied on an extremely efficient microkernel as a minimal basis for the operating-system family.

[21]It is worth noting that QNX was the first commercially available microkernel. The system is still quite successful. But strangely enough, *Mach* (i.e., OSF/1) and *Chorus* seem to monopolize this "tradename."

Important influences came from *V* [16], *Amoeba* [58], and *Mach* [96]. Compared to its "ancestors," the PEACE microkernel was significantly less complex and higher in performance.

The microkernel approach (of the 1986 PEACE) was successful in solving the problem of managing the SUPRENUM system so that all the nodes jointly did "number crunching" for the parallel applications quite well. However, this approach failed in the presence of "well-shaped" applications that have been tuned with respect to the actual number of nodes (i.e., degree of parallelism) provided by the hardware. With mapping application tasks in a one-to-one correspondence with the nodes, the performance—in particular the message startup time—was not satisfactory [78]. Although it was explicitly expressed by the user community that multitasking features (e.g., as provided by a microkernel) were a predominant issue in order to increase the scalability of parallel numerical applications, these features were utilized only in very rare cases. In reality, the majority of SUPRENUM applications were prepared in such a way that the nodes merely ran a single task. In other words, tasks were preferably mapped in one-to-one correspondence with the nodes.

Reflecting the experiences of the SUPRENUM development, a redesign was started in 1990, exploiting object-oriented design principles [80]. The result was "*objective* PEACE"[22] and an approach that extended the ideas of *Choices* [14] into parallel computing environments. In contrast to *Choices*, a microkernel was no longer used as the minimal basis for a family of parallel operating systems—without having abandoned the microkernel approach entirely [22]. The original concept of a family of parallel operating systems was not changed. This concept inherently supports customization, extensibility and maintenance of software systems in general and operating systems in particular. In this sense, the family idea was seen to be very close to object-oriented design principles [21].

The object paradigm was not only motivated by its suitability for the modeling of program families in general and operating-system fam-

[22]If not stated otherwise, "*objective*" is omitted hereafter. PEACE then always denotes the object-oriented variant of 1990. The development from 1986 to 1990, therefore, is also referred to as object-based PEACE.

ilies in particular. The expectation was also that object orientation is a proper instrument to make traditional operating-system boundaries disappear and thus help to decrease the message startup time and increase communication performance. This expectation came true. Inheritance lets applications appear as specializations of fundamental operating-system abstractions. Boundaries at which the operating-system domain begins and the application domain ends become indistinct. On the other hand, application patterns that require a clear separation (e.g., between the user and supervisor modes) are supported just as well, or even better, as in a non-object-oriented approach.

Polymorphism is one of the most essential aspects of object-oriented systems, and an operating-system family is clearly polymorphic. Thus, the metamorphosis from an object-based to an object-oriented PEACE appeared to be "gentle." The major differences, in fact, were in having used C++ instead of C as the implementation language and a different discipline for the implementation of system components (using classes instead of modules). The *design* (i.e., structuring) of PEACE was and is family-oriented, while the *implementation* was object-based and is now object-oriented.

1.6 Summary

Forthcoming massively parallel systems will have distributed-memory architectures. They consist of several hundreds to thousands of autonomous processing nodes interconnected by a high-speed network. A major challenge in operating-system design for these architectures is to design a structure that reduces system bootstrap time, avoids bottlenecks in serving system calls, promotes fault tolerance, and is dynamically alterable and application-oriented.

A massively parallel computer, by its nature, is a *tightly coupled distributed system*. Parallel applications being run by such machines are thus a specific type of distributed application. These applications demand systemwide message passing with very low latency and very high efficiency. It is this performance issue that, from the operating-system design point of view, is one of the most significant characteristics distinguishing a parallel application from a distributed application.

Distributed operating systems, even those based on a microkernel, are far too overhead-prone. Rather, specifically designed parallel operating systems are required which also exhibit distributed-operating-system characteristics.

Therefore, the major goal in parallel operating-system design is to combine transparency with efficiency. *Transparency* means hiding from parallel applications typical problems originating in distributed systems—but without enforcing transparency in all cases. This requires almost every system service a distributed operating system provides—but not necessarily at all times. *Efficiency* means reducing the message startup time for a given application to an absolute minimum. Message passing is a fundamental system capability supporting inter-node communication in massively parallel systems, and hence must be provided by a proper platform—but without all applications being obliged to use the same platform. Thus, system services are to be provided on an *extend-on-use* basis.

State-of-the-art parallel operating-systems design must obey the maxim to not punish an application by system functions that will never be used. This includes the entire spectrum of computer resources, ranging from memory space to processor cycles. An open, application-oriented operating-system structure is required, with the application and not the operating system deciding which functions must be supported and which must not be. An approach must be followed in which an operating system is understood as a *family of program modules* and not as a monolith of more or less related components. In such a context, the parallel application is an integral part of a *family of parallel operating systems*, and *object orientation* then is the natural choice for the design and development of such a family. Thus, an application becomes the final system extension.

Chapter 2

Functional Decomposition

Operating-system construction is primarily a design and structuring problem. This is true for all kinds of operating systems and spans the areas of distributed, parallel, real-time, and fault-tolerant computing. The situation does not change with massively parallel (distributed-memory) architectures. A highly scalable system structure is required. This structure must scale well with at least hundreds to thousands of nodes, and should scale well with millions of nodes. Thus, the structure should meet the needs of both today's and future parallel architectures. In addition to all of these constraints regarding how to divide the complexity of the system, there is also the requirement of finding a system structure that guarantees the least possible message startup time for a given parallel (MIMD) application.

The following sections discuss the design of PEACE, explaining how the concept of program families can be applied to developing a highly modular operating-system structure that especially meets the needs of massively parallel (hardware/software) systems. Following the principle of *stepwise refinement* [94], the macrostructure of the system is designed first. This structure reflects the overall system architecture and deals mainly with splitting an operating system into individual distribution units. Afterwards, the microstructure of the system is designed. The question considered is how to design the minimal subset of system functions in order to meet the needs of a very large number of parallel applications. The microstructure design focuses on performance and serves to implement an adaptable parallel computing platform for distributed-memory machines. In contrast, the macrostructure addresses the problem of how to divide system complexity by identifying loosely coupled but closely related building blocks. Finally, important design aspects are summarized.

2.1 Approaching the Family Concept

The first design step is to identify a *minimal subset of system functions*, which defines a platform of fundamental abstractions aimed at supporting parallel computing. Such a minimal basis solely introduces *mechanisms* from which more enhanced system functions can be derived. Relying on this basis, functional enrichments are made stepwise by means of *minimal system extensions*. These extensions introduce *mechanisms* and/or *strategies*. By either repeatedly adding system extensions or recursively applying the design principle to further refine a system component, a multilevel hierarchy of virtual machines emerges.

Whether or not extensions are necessary depends on the application and on the functions currently provided by the system. The following rule of thumb points out which role applications play in a family-based system design:

> Applications are always the best extensions one can think of—
> they are the ultimate system extensions.

Extensions serve to close the semantic gap between the application layer and some virtual machine layer of the hierarchically composed system. If there is a virtual machine definition that fulfills the demands of a specific application, obviously nothing needs to be done. This might be the virtual machine defining the minimal subset of system functions. In that case, this subset can be assumed to have been developed with a specific area of application in mind. Therefore, it may exhibit more complexity and functionality than the "pure family-oriented" approach would lead one to expect. This platform will be referred to as the "*minimal but perfect basis.*"

In approaching the family concept and taking the initial step, one has to weigh pragmatism vs. purism. The purist will always start with the most fundamental virtual machine one can think of. It may then take a number of extensions before a minimal subset of system functions emerges that is acceptable by some application. This is the logical consequence of a pure bottom-up approach. The practitioner is more top-down-oriented and starts with a virtual machine that provides the "optimal" platform for a given set of applications.

In the approach described here, the pragmatic way is pursued first by designing a minimal basis that is suitable for parallel processing in a distributed-memory environment. The goal is to develop a parallel computing platform that focuses on programmability. The refinement of this virtual machine then leads back to the pure approach and yields the tailoring of the minimal basis to improve efficiency. This will also open the door to employing the platform for completely different environments, such as distributed computing or real-time processing.

2.1.1 Minimal Subset of System Functions

Parallel processing in the context of distributed-memory machines calls for at least two fundamental abstractions: *processes* and *messages*. It is therefore obvious that the minimal basis must provide for some sort of process and communication model. Furthermore, the process model must not assume a specific address space model. This constraint increases portability and is fundamental to the family idea, making address space management a minimal system extension to process management.

A virtual machine with these capabilities will already support a large number of parallel applications. It could even be considered a software abstraction of a Transputer-like processor.[1] As the Transputer community demonstrates, logical address spaces are not necessarily required to execute parallel programs. However, under certain circumstances the same community also worries about the lack of an address space model and demands enhanced system features. These circum-

[1] To make it clear, it is not intended to open a discussion of whether the Transputer is an exemplary or impractical processor architecture for parallel computing. As matters stand, the Transputer (at least in Europe) achieved a remarkable market penetration. Therefore, the comparison is reasonable in order to illustrate and make understandable the functionality of the minimal subset of system functions envisioned.

stances are given, for example, when the application (dynamically) comprises more parallel executable processes than there are processors available. Hence, for some processes a pseudo-parallel execution is required. The introduction of logical address spaces can be postponed and is a case for minimal system extensions. This is where, for example, Transputer-based applications will benefit from the family concept. Some applications may still rely on a more or less familiar platform, while others are not constrained by the hardware architecture. Each class of application will be supported by a different member of the parallel operating-system family. Thus, one is neither forced to pay for unneeded system functions nor restricted in the attempt to increase scalability.

2.1.1.1 Process and Message Abstractions

Processes introduce scaling transparency. They make the modeling of parallel applications independent from the actual number of processing nodes. In fact, a virtual processor model can be easily implemented by processes. On the other hand, scaling transparency is not only an issue in the programming of massively parallel systems, but also improves availability in the case of permanent node failures. The crash of a single node can cause the premature end of application processing if the system does not support the migration of program activities onto still-functioning nodes.[2] For this purpose, the system needs an abstraction for modeling program activities, which is the process. Thus, a process serves as the common abstraction for both the application and the operating system. The application uses processes mainly to express parallelism, and the operating system uses processes to administer applications and keep them running and to implement the services required.

Communication based on message passing is a must when the processing nodes have direct access only to their local memory, while ac-

[2] A node crash can be disastrous for long-running numerical applications. The restart of a parallel program from the beginning is not acceptable when it takes two days to finish it, and a node crash may force the premature end just five minutes before its termination. For these applications, (hardware/software) fault tolerance is absolutely mandatory.

cess to nonlocal memory (i.e., to the local memories of other nodes) involves the execution of a network communication protocol. Since the process is the basic abstraction used to model (user/system) activities, *inter-process communication*, not inter-node communication, is required. This interlocks the communication model with the process model and requires a (logically) indirect addressing scheme to identify communicating entities. Process identifiers are introduced instead of relying on node numbers.

Whether synchronous or asynchronous communication must be supported depends mainly on the process model [8]. From the operating system (i.e., bottom-up) point of view, synchronous inter-process communication is the best choice in order to achieve high-performance and high-bandwidth data transfers. In contrast to asynchronous communication, intermediate buffering, and hence the additional overhead of message copying and resource (i.e., buffer and memory) management, is not implied by the communication model; at most, it will be implied by the network hardware interface. This enables bulk data transfers directly between (the address spaces of) the communicating processes without operating-system intervention.

The choice of synchronous inter-process communication implies a potential loss of parallelism at the application level. This must be compensated for by a process model that allows for concurrent programming even on a single node (i.e., supports a *multithreaded address space*). The implementation of such a model must result in a process switching time significantly smaller than the copying and buffering overhead involved in asynchronous communication. Data copying overhead may be very small when done by dedicated hardware—which has become state-of-the-art [32]. In fact, the crucial point in asynchronous communication is the implied buffer and/or memory management function in order to temporarily store the data at either the sender or receiver site. There is little or nothing of this function that could be accelerated by hardware. It is the software which has to implement a model that helps to either overcome or hide the implied overhead of asynchronous communication.

Asynchronous communication relies on copying and buffering to allow reuse of the original message buffer for the assembly of the next message. The optimal solution, namely to avoid copying/buffering even in that model, is a synergetic approach addressing compiler (and programming language), runtime environment, and operating system. The implementation of the send primitive (i.e., the runtime environment) could pass a message descriptor along with a send request down to the operating-system kernel, which "immediately" returns after having stored—or even executed—the request (but not the data). Immediately executing this request would mean instructing DMA hardware to start the networkwide transfer of a user-level memory segment. In that case, the DMA transfer would run in parallel[3] to the return from the kernel and the execution of the application program. The compiler then has to implement a "*lazy synchronization*" scheme, inserting synchronization points into the application program that ensure the integrity of the message buffer. These points are the output of the data dependence analysis of the application program carried out by the compiler. They represent those places where the application program is going to store variables in the (memory region establishing the) message buffer. The synchronization point, then, is a check of the status of the transfer process. This could mean to either query the kernel or verify a local flag maintained by the runtime environment. In the latter case, the kernel has to provide the means by which this flag can be set concurrent to the processing of the application program.

The concept of *lightweight processes* [40] is very suitable for implementing such a runtime environment. However, lightweight processes, also called threads,[4] display power only as members of a *team* [15] whose purpose is to push on ahead with the nonsequential[5] execution of a single task. The minimal basis then strongly promotes a process-

[3]This assumes that there is sufficient memory bandwidth for both the CPU and the DMA logic.

[4]Sometimes, in the literature, a distinction is made between a lightweight process and a thread. The former implies a single-address-space model, whereas the latter relies on multiple address spaces, that is, the separation between the user and supervisor modes of execution. Hereafter, lightweight process and thread will be used synonymously. A *featherweight process* then denotes a lightweight process (i.e., thread) that will be subjected to a single-address-space model.

[5]The term "nonsequential" does not only mean "parallel" but also "pseudo-parallel" (or concurrent). Even if there is only one processor in the node, a pseudo-parallel execution of a task becomes feasible by preemptive scheduling of the threads of that task.

ing model in which parallelism (or concurrency) is not a side effect of communication, but is expressed explicitly by means of threads.

2.1.1.2 Latency Hiding

There is another aspect that makes lightweight processes an important instrument to speed up parallel programs. It is a common approach in many distributed-memory parallel systems that a process issuing a message transfer is suspended at least as long as the transfer request takes to pass over to, and be stored at, the destination node.[6] In fact, compared with primitives for distributed computing [52], asynchronous communication in (most of) these systems is not of a true "*no-wait-send*" semantics, but rather is closely related to a pattern based on some sort of "*synchronization-send*" strategy.

In message-passing architectures, process suspension is often not understood as the means of switching to another thread of control, and thus possibly keep on with the execution of the same or a different application program. Rather, process suspension is implemented as "busy waiting." It stands for the temporary inability to proceed with the execution of the actual application program. This pattern applies irrespective of the communication model, that is, whether synchronous or asynchronous communication is provided. It is motivated mainly by the type of network used to interconnect the nodes of the parallel machine.

The communication system reflects a very tightly coupled architecture that enables extremely lightweight remote data transfers. In these systems, sending a message across the network is often much simpler and less expensive than transferring the same message into another address space on the same node. It is basically a question of whether the network controller or the memory management unit can be programmed more efficiently. In many cases, the network controller provides the prompt service.

Although straightforward, the busy waiting approach may have a significant drawback on the further processing of the sender task if only a single thread of control can be modeled. Network latency then becomes a performance-limiting factor and determines how long a process

[6] Usually, this is the approach in multithreaded architectures, not in message-passing architectures [32].

is delayed from making progress with "number-crunching." In order to compensate for this performance drain, *latency hiding* became one of the most important buzz words in parallel systems.

Latency hiding can be implemented in two ways. One approach is to design dedicated communication hardware, such as a separate (software programmable) *communication processor* that takes care of the execution of the communication protocol required for networkwide message transfers [34]. The other approach relies on specifically designed software functions: as long as the sending thread is blocked, another thread of the sender team is used for further program execution. It is obvious that only lightweight processes are suitable to implement such a communication and processing pattern. However, this kind of latency hiding requires that there are sufficient threads at the application level that can be dispatched by the system. Thus, a proper programming model is required to obtain the multiple threads of control through the appropriate partitioning of the application program. It turns out that the development of such a model (including programming language and compiler) is the real challenge in the software implementation of latency hiding on a threads basis.

Lightweight processes experienced a breakthrough in the parallel computing community only with the introduction of the Transputer—which, unfortunately, is the main reason why many application programmers still regard the Transputer as an exemplary processor architecture. Using the Transputer, an application programmer is forced to think in terms of threads, and thus in most cases implicitly performs latency hiding. Nevertheless, it is almost a tragic case that lightweight processes appeared in the parallel computing area only when a specific physical processor was available. Even a virtual machine implementation is suitable to achieve latency hiding. This virtual machine will perform better than the Transputer if a node must be shared by operating system and application tasks. In any case, providing a minimal basis that supports threading is the right choice. It can either be (more or less completely) replaced by a proper hardware architecture or provide an optimal solution if operating system and application tasks must be equally supported.

2.1.2 Minimal System Extensions

A minimal basis which supports threading and communication can be considered as the only operating-system support that needs to reside in the majority of processing nodes in order to execute tasks locally and provide the coupling with remotely residing tasks. In this sense, the minimal basis might already be the optimum for certain applications—it represents the minimal but perfect basis.

Nevertheless, extensions will be necessary, but certainly not on every node. For example, being able to both process and manage threads are two very different things. In contrast to thread processing, thread management is concerned with the dynamic creation and destruction of process objects. It is sufficient for thread processing to know about the threads that must be executed, and it is not necessary at this stage to solve the problem of how these threads can be added to or removed from the system. Thus, compared to thread management, thread processing tends to be a trivial function.

Having in mind that many parallel applications need not dynamically create threads, machine bootstrapping can be exploited to get user programs running in batch mode. Other applications are highly dynamic in nature, and thus must be supported by a proper thread management package. Another example is I/O. Many parallel machines have dedicated I/O nodes to which disks, graphic displays, or hosts are connected. These nodes must be equipped with proper system components that perform I/O on behalf of remote threads. The threads merely need to have access to these services, which can be readily achieved on a *remote procedure call* (RPC) [60] basis. As a consequence, only the minimal basis of the I/O nodes should need to be extended appropriately.

By adding minimal extensions, an operating-system family is constructed bottom-up, whereby construction is controlled top-down: lower-level components are introduced only when required by higher-level components. According to this maxim, system functions for scheduling, security, process management, (virtual) memory management, exception handling, I/O, file handling, checkpointing and recovery are designed. An open, application-oriented and evolutionary system organization is the consequence.

2.1.3 Exchangeable System Components

Understanding functional enrichment as a procedure that is used to add components is only one aspect. It also includes *component replacement*. During the design phase, a commitment on the minimal subset of system functions must be made. This introduces the risk of making wrong design decisions early in the software design process. One of the most important decisions is identifying the proper operating mode of the node, that is, whether single-tasking or multitasking is to be supported.

The processing of parallel applications by a massively parallel machine always implies communication; hence, the need for communication functions. The application might also call for a single- or multithreaded address space (i.e., task) on a node. Another application demands multitasking, which is then a functional enrichment of multithreading. Should the minimal basis therefore support multitasking? If the design decision advocates multitasked nodes, and tasks are mapped in one-to-one correspondence with the nodes, then a significant degradation of the message startup time is obtained [78]. Multitasking is not free of charge. Even if not exploited, these functions can be sources of some (communication) performance loss because of their interdependencies with other system functions—they are not called (i.e., needed) but "used" [67] by the application. A multitasking kernel running a single-tasking application program is the typical example of this situation. In this case, the single task has to pay for protection measures (such as CPU and memory protection) that are only needed if multiple tasks are requested to share the same physical processor. A design decision to support solely multitasking nodes will handicap single-tasking applications, and thus not conform with the idea of program families.

In order to overcome this problem, a number of implementations of the minimal basis must be available, with each implementation being dedicated to a specific type of application. For transparency reasons, all applications must see the same external (abstract) interface of the minimal basis. What differs is the internal behavior (i.e., the concrete implementation). The external interface is mainly concerned with communication, while the internal behavior mainly dictates the process model and the operating mode of the node. Thus, there is not a single minimal basis, but rather a single basic interface.

With the minimal basis being an *abstract data type*, a number of implementations of the same interface can coexist. This makes the minimal basis exchangeable at least from the design point of view. Flexibility is maintained, although the minimal subset of system functions must have been fixed early. However, note that transparency can be achieved only on a syntactic and not a semantic level. It is obvious that an application demanding a multitasking node is not very well supported by a single-tasking mode of operation. But this is not the key issue, as one is not trying to let a multitasking application run in a single-tasking environment. The key issue is that single-tasking applications have the opportunity to improve system performance by exploiting a functionally scaled-down component. Concerning the message startup time problem, it is the semantic and not the syntactic difference between the variants that determines the performance bottleneck.

The abstract data type "minimal basis" tends to be fairly complex. It is therefore reasonable to make its design family-oriented, not only to reduce complexity but also to let different family members take care of different operating modes. As a consequence, the overall structure exhibits a family within a family: the *operating-system family* is built on top of a minimal basis that is implemented as a *kernel family*, with the kernel primarily being an abstract data type providing communication functions and defining the operation modes of the nodes.

2.2 Macrostructure

The maxim for the design and development of operating systems for distributed-memory parallel computers is to reduce the number of site-dependent components to an absolute minimum. Hereafter, a site denotes either a cluster, a node, a processor, or an address space. Thus, the design must lead to an almost arbitrarily configurable system structure. Whether a system component is sharing a specific site with other system components should be a matter of configuration and not implementation. Only by these means can the problem of bootstrapping very large scalable computer architectures be tackled.

A highly modular system structure is required. This structure must enable, but not enforce, the decoupling of functional units such that a high degree of decentralization becomes possible. In order to design a structure that meets the needs of (massively) parallel systems, the following three observations give some directions:

- *The hardware architecture relies on a message-passing system.* This requires some kind of message-passing kernel acting as a software backplane to interconnect all the entities (i.e., user and/or system tasks) that will be executed by the parallel machine.

- *The parallel machine is of limited scalability.* Applications that were designed to operate in an environment with a larger (or even unlimited) number of nodes must be supported by a parallel computing platform offering any degree of parallel processing. Instead of dealing with physical processors (or nodes), provisions for a virtual processor model must be made.

- *The computing platform is a dedicated system.* Not all nodes are equipped with the same set of peripherals and executing the same system software—however, they may execute the same application. Most of the nodes are used for number-crunching purposes, and not every node is directly connected to a mass storage device, for example.

These facts suggests a system organization in which each aspect is covered by a separate building block, leading to three major subsystems. Two basic rules motivate this step. The first rule is that splitting up a system into subsystems leads to a general division of complexity, thus making the problem of constructing a parallel operating system more understandable and manageable. The second rule is that a modular structure of well-defined subsystems reduces the number of node-bound system components to an absolute minimum.

With these aspects as the ulterior motive, applying the idea of a program family in order to find the overall system structure is almost straightforward. Since entities must be able to cooperate with each other, the software backplane has to act as the minimal subset of system functions. Providing the notion of virtual processors is done by introducing minimal extensions to the minimal basis, since not every application will require more nodes than are physically available. Similarly, services that depend on the availability of specific devices are qualified as system extensions. The same holds for services that are only used in the context of specific applications.

2.2.1 Functionally Dedicated Building Blocks

The result of these considerations is a coarse-grained family organization. Every family member is constructed from three major building blocks (Figure 2.1). These building blocks are the *nucleus*, the *kernel*, and POSE, the *parallel operating system extension*. In addition to the system components, the *application* is considered as the fourth integral part of this architecture. The application largely determines the complexity of a family member and the distribution of the building blocks over the nodes of the parallel machine.

The nucleus implements systemwide inter-process communication and provides a runtime executive for the processing of threads. It acts as the minimal basis and is part of the kernel domain, with the kernel being a multithreaded system component that encapsulates minimal nucleus extensions. These extensions implement device abstractions, dynamic creation and destruction of process objects, the association of process objects with naming domains and address spaces, and the propagation of exceptional events (traps and interrupts). Application-oriented services such as naming, process and memory management, file handling, I/O, load balancing and (inter-) networking to provide some host access are performed by POSE.

The kernel and POSE services are built by active objects implemented by lightweight processes. In contrast, the nucleus is an ensemble of passive objects that schedule active objects. Kernel and POSE are *multithreaded entities*, and thus support concurrent service processing. An entity provides a common execution domain for active objects and is considered as the *unit of distribution*, while an active object is considered as the *unit of execution*.

The nucleus and kernel constitute the *kernel entity*. This entity defines an abstract processor for the execution of possibly multithreaded tasks. The sole purpose of the kernel entity is to provide hardware abstractions that make (parallel) applications and POSE independent

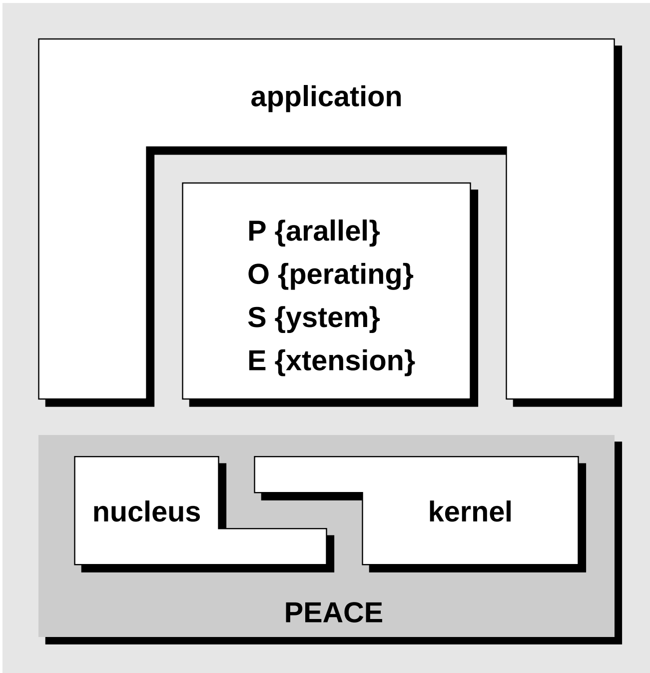

Figure 2.1: Building blocks

from the physical characteristics of a given processing node.

2.2.2 Extensibility and Configurability

Entities are system extensions. They are loaded on demand and (in most cases) can be arbitrarily distributed over the nodes of the parallel machine. Loading a kernel entity on demand means bootstrapping a node. The need for having a specific entity running on a particular node is expressed by either the application or a system administrator. In the former case, the entity is loaded when the service it represents is called for the first time. The entity vanishes when this service is no longer needed. This may happen, for instance, upon termination of the application entity that demanded the service. Note that in this model, application entities are not distinguished from system entities such as the file manager, process manager, or external pager. In other words, provided the application is properly structured, incremental loading of parallel applications works exactly the same as incremental loading of the operating system. Hence, it is free of charge.

In the case of the system administrator, an initial configuration description specifies which entities must be initially loaded onto which nodes when bootstrapping. This procedure loads "opaque" entities from some bootstrap device (usually a disk or network) and stores them into the local memories of the nodes. Understanding parallel programs as groups of related entities, bootstrapping of these opaque entities can be used to get applications, not only system components, running. The purest form of batch processing is thus achieved.

Either way, POSE is a dynamically alterable building block. The design specifies that POSE can alter its shape depending on the applications that are to be executed by the parallel machine. Thus, applications determine the actual complexity of POSE. In consequence, they are responsible for bringing (i.e., bootstrapping) the operating system on the machine.

Incrementally loading the operating system does not mean placing POSE entities on the node where the demanding application is residing. Since these entities may be qualified as site-independent, they could also be loaded on any appropriate node. In order to achieve an "optimal" mapping, expressing some nearest-neighbor relationship between the entity and either a higher-level application or a lower-level hardware/software component may be sensible. This is where the configuration aspect comes in. The entity must be attributed properly, and the attributes must be interpreted by some low-level load-balancing service.

2.2.3 User and Supervisor Modes

Logically, both the applications and POSE are assumed to be executed in the nonprivileged processing mode (*user mode*) of the underlying processor. In contrast, the nucleus and kernel are designed for execution in the privileged mode (*supervisor mode*). Thus, the logical design defines an artificial boundary isolating these two components from other entities. This does not mean that the actual system representation must always consist of isolated system components. Rather, the purpose of introducing the boundary is to guarantee downward scalability.[7] Therefore, it is of importance to clearly identify those components that, under certain circumstances, must be executed in the privileged mode and others that can be executed in the nonprivileged mode. Note that supervisor-mode components also constitute the set of (logically) node-bound entities. The complexity of this set has a significant impact on how long it takes to get the parallel machine started. By designing a system structure that exhibits a "vague" boundary between the user and supervisor modes, one is able to determine the number of node-bound entities dependent on the application that must be supported.

The design defines a *family of parallel operating systems* that is extended by a *kernel family*. The kernel family contains members that mainly distinguish between the separation of the user and supervisor modes. Basically, these members implement a number of different operating modes. The members are all node-bound, and therefore classified as site-dependent. The operating-system family is composed of site-independent entities. It is a configuration matter as to whether or not these entities share the same site (i.e., node, processor or address space) with the kernel and the nucleus. Similar considerations hold for application entities.

This design especially promotes address-space sharing between all entities of the same node, provided that the structure of the application allows it. In a single-tasking mode of operation, the application, POSE, kernel and nucleus all may execute within a single address space.[8] However, it must be clear that only in the presence of secure programming languages and the absence of temporary node failures can the model of a single address space be considered safe. Nevertheless, this model is still applicable for mature applications—and, of course, for mature system components—to prevent per-node overhead in crossing "artificial" subsystem boundaries. As a consequence, since it is not possible to safely protect user domains, the entire parallel machine must be

[7]The system design must be complete in order to be able to later integrate a version of a system component of significantly less (i.e., scaled-down) functionality.

[8]How big the resulting (nonsequential) program then becomes depends exclusively on what the application demands from the system. This issue will be discussed later.

operated in batch mode.

This short discussion of a very specific case points out that it is the application and not the system that must forbid certain representations. This is a design rule that is predominant and thus of significant importance for parallel systems.

2.2.4 Intra- and Inter-Domain Calls

Whether the application or POSE must cross address space boundaries to invoke the nucleus depends on the implementation of the calling sequence. The service access point of the nucleus is represented as an abstract data type with two different implementations: an *intra-domain call*, with the callee sharing with the nucleus both the same node and address space segment, and an *inter-domain call*, with the callee sharing with the nucleus the same node but not the same address space segment (Figure 2.2).[9]

The intra-domain call implementation is automatically generated as a side effect of compiling the nucleus interface. This calling sequence thus corresponds to normal (object-bound) procedure calls. Consequently, all components are executed within a single address space.

The inter-domain call implementation distinguishes between two main categories of isolating independently generated programs. One category introduces the physical separation of the user and supervisor modes, with the application and POSE mapped to *user space* and the nucleus and kernel mapped to *supervisor space*. In this situation, a *trap* mechanism is used to invoke nucleus services. POSE and kernel are then invoked on an RPC basis. The other category relies on a logical isolation between the higher-level application/POSE complex and the lower-level nucleus/kernel complex. All components reside within a single (possibly very large) address space. That is, isolating the different components becomes primarily a function of the linker and is not enforced by hardware. The difference from the intra-domain variant is that the invocation of nucleus, kernel, and POSE happens indirectly and not directly. Kernel and POSE services are still invoked through an RPC mechanism, while nucleus invocation happens through a "*jump table*"

[9]An inter-domain call is also denoted as a cross-domain call.

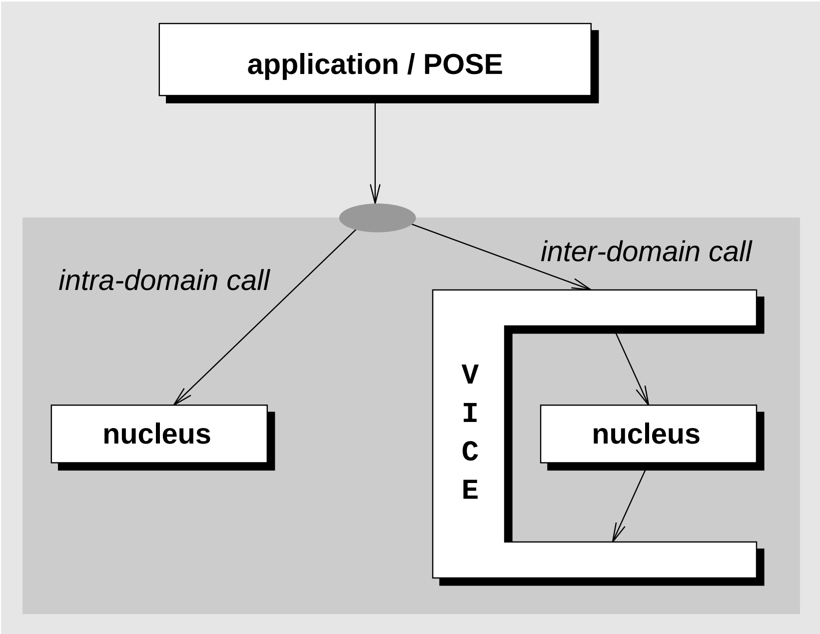

Figure 2.2: Service access point of the nucleus

mechanism. The jump table approach may be implemented by the dynamic binding feature of object-oriented programming languages such as C++ or by specifically designed "carrier" objects. In both cases, as a consequence of calling a nucleus function, costly address-space (i.e., context) switching is avoided.

Every nucleus primitive that is exported, and thus can be invoked by ordinary (user/system) processes, is also represented by a *stub routine* (Figure 2.3). In order to invoke its "vis-à-vis" in the nucleus domain, the *nucleus stub*[10] routine executes the calling sequence of an inter-domain call. It marshals input parameters, encodes the nucleus primitive, hands over the request packet to the nucleus, decodes the return value and unmarshals output parameters after having been resumed. The receiving

[10]The names used in Figure 2.3 indicate the component represented by a stub. That is, in the user context, the nucleus is represented by a *nucleus stub*. In the nucleus context, the user is represented by a *user stub*.

party takes the role of a *user stub* and decodes the nucleus primitive, unmarshals input parameters, invokes the nucleus primitive, encodes the return value, marshals the output parameters and returns from the inter-domain call.

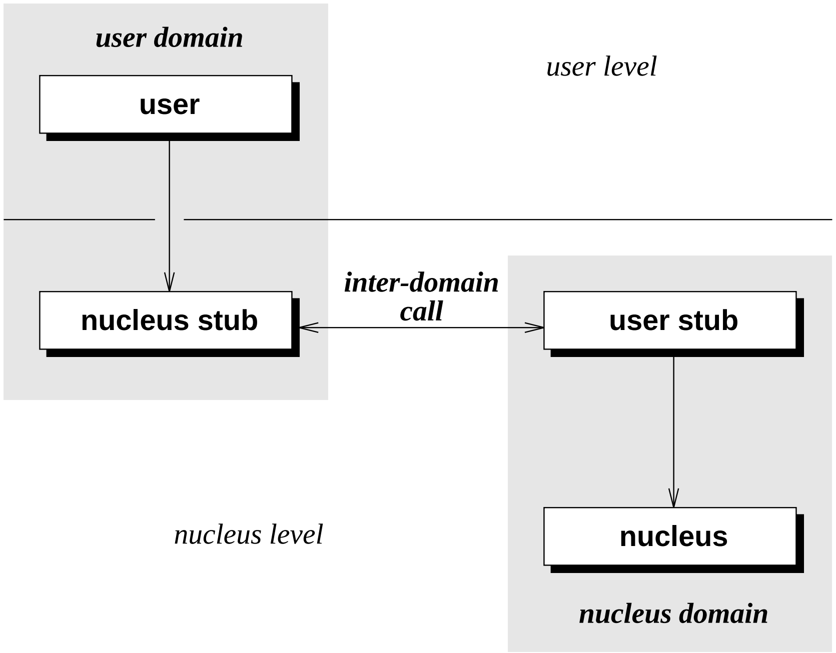

Figure 2.3: Inter-domain call stubs

In fact, the functions executed by the stubs are comparable to those needed to perform an RPC. The main difference is that a very straightforward binding phase takes place, and the nucleus request is not transmitted via a network. The binding phase is nothing but statically fixing the function code, which happens at compile time. The transmission phase means activating the stubs and exchanging parameters through processor registers and/or memory areas. This similarity of operation motivates the use of a *stub generator* that translates the nucleus interface specification into executable code for the stub routines.

The stubs are part of VICE (see Figure 2.2). VICE is a *versatile intercommunication executive* that takes the form of a "sandwich" [67], with the nucleus being the "spread." It accepts inter-domain calls, dispatches the corresponding nucleus primitive, and finishes inter-domain calls. All these activities are performed under the control of the currently executing process (i.e., callee). In addition to handling cross-domain calls, VICE adopts those functions that are used by the nucleus to transfer data from/to higher-level entities. These transfer functions exist in two different versions, distinguishing whether or not the nucleus possesses a private address space and is thus vertically isolated from user/system entities. Therefore, the same nucleus implementation can be used in two different environments.

Since VICE "uses" [66] the nucleus when invoking the requested function, and the nucleus "uses" VICE when transferring data from/to higher-level entities, a cyclic "uses relation" between the two independently callable instances would be given. This results in a highly interdependent structure that is difficult to manage and tends to cause hidden performance penalties. The cycle is broken up by slicing the VICE instance into two parts, which leads to the structure shown in Figure 2.2.

2.3 Functional Hierarchy

The functional hierarchy defined between POSE, the kernel and the nucleus makes possible a very high degree of decentralization (Figure 2.4). All components are encapsulated by (active/passive) objects, so that object invocation schemes are required to request service execution. Well-defined interfaces enforce a clean separation of all building blocks. Different styles of calling sequences to cross the (nucleus, kernel and/or POSE) interface enable various system configurations. These configurations particularly distinguish between a single or multiple address space model being implemented on a node.

The following subsections discuss the invocation schemes used for the interaction with and between the building blocks shown in Figure 2.4. A presentation of possible kernel configurations is given.

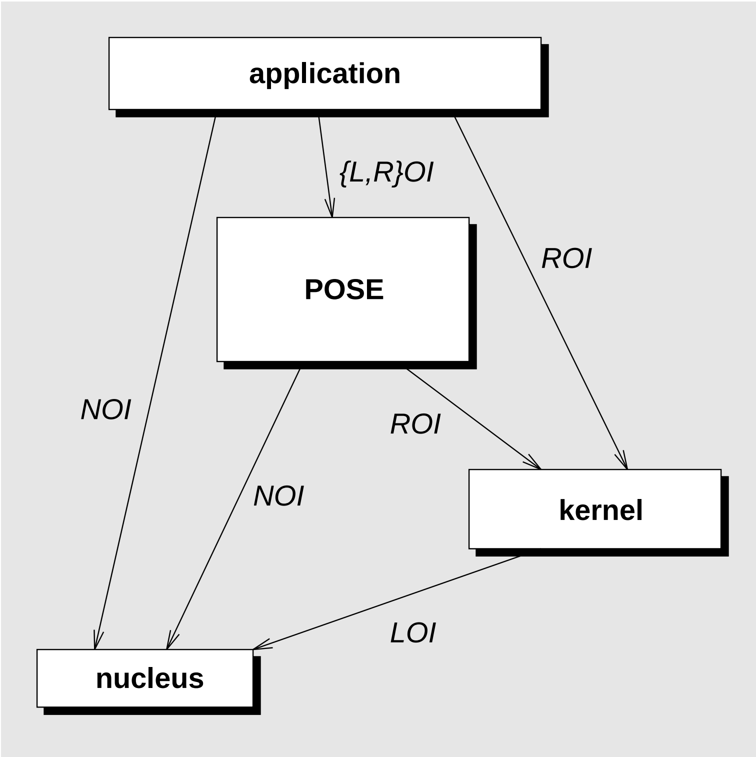

Figure 2.4: Problem-oriented invocation schemes

2.3.1 Invocation Schemes

Nucleus services are made available to the application and Pose via *nearby object invocation* (NOI). The logical design assumes a separation of the nucleus from the application (and Pose), which may call for the use of traps to invoke the nucleus and for address space isolation. This is the place where cross-domain calls may happen. The nucleus is "nearby" the using entity. It shares the same node, but not necessarily the same address space segment, with the entity. Since the kernel and nucleus together reside in the same address space, the kernel performs *local object invocation* (LOI) to request nucleus services. LOI causes the thread to continue execution within the same address space (or domain). In contrast, NOI causes the thread to continue execution within a different address space on the same node.

Logically, kernel services are made available via *remote object invocation* (ROI). The ROI scheme always implies context switching, but not necessarily address-space switching [61]. A separate thread of control is used to execute the requested method. In contrast, NOI implies the activation/deactivation of the nucleus address space via local system call traps. The implementation of ROI takes advantage of the networkwide message-passing services provided by the nucleus, and thus is logically based on NOI. However, since the nucleus and kernel together define a single program, nucleus primitives invoked by the kernel to either issue or accept ROI requests hence will not be activated via NOI but LOI. Keeping in mind Figure 2.2, NOI means requesting a call to a nucleus function from VICE. Once activated via NOI, VICE calls the nucleus function on behalf of the requesting thread by employing LOI. That is, the nucleus is always accessible via LOI, but may also be accessible from a high-level application domain via NOI to VICE. The kernel will never invoke the nucleus through VICE.

Services of POSE are requested via LOI and ROI. The former scheme is used to interact with the POSE runtime system library, whereas the latter is used to interact with POSE active objects. In certain situations the POSE library directly transforms a LOI into one or more ROI requests to the kernel. The POSE service, then, is not provided by an active but by a passive object (i.e., on a library basis).

The NOI and ROI patterns are nothing but specific RPC variants. Both implement a synchronous control flow transfer via a "narrow channel" [60]. For example, the narrow channel is represented by a trap handler (in the case of NOI) or the message-passing system (in the case of ROI). Thus, NOI and ROI indicate the two technical means of carrying out an object-bound RPC. In both cases, the control flow transfer entails logically changing protection domains (e.g., address spaces or segments). However, different from NOI, ROI causes a (lightweight) process switch. NOI forces the current thread of control to continue processing in a different domain; that is, NOI corresponds to an object-bound *lightweight RPC* [9].

2.3.2 Actual Structure

From the design point of view, neither the kernel nor POSE need to be present on every node, only the nucleus. In a specific configuration, the majority of the nodes of a massively parallel machine are equipped with the nucleus only. Some nodes are supported by the kernel, and a few nodes are allocated to POSE. All nodes can be used for application processing, but they are not all obliged to be shared between user tasks and system tasks. The functional hierarchy of the three building blocks expresses the logical design of PEACE, but not necessarily the physical representation. However, it depends on the actual operating-system family member whether the various schemes of object invocation (Figure 2.4) become effective as specified by the design or can be replaced by a simpler, more efficient alternative.

Although the functional hierarchy assumes NOI for the interaction between application (POSE) and nucleus, the LOI scheme is used for those members of the kernel family that place their focus on performance. As was discussed in Section 2.2.4, the entrance to the nucleus is represented as an abstract data type with two implementations. The first implementation sacrifices vertical and horizontal isolation. Thus, there is neither a separation between the user and supervisor modes of operation (vertical isolation) nor a separation between competing tasks (horizontal isolation). In this case, NOI actually means LOI—VICE is not used to call the nucleus. Horizontal isolation means that user/system entities have a private address space, that is, operate in a private protection domain. The second implementation assumes complete (i.e., vertical and horizontal) isolation and requires a trap-based activation of the nucleus. NOI then becomes a cross-domain call to be handled by VICE. The variants basically distinguish between single-tasking (no isolation) and multitasking (isolation) modes of operation. They implement different members of the kernel family.

A single-tasking mode of operation implies that only a single address space is supported on the node. As pointed out above, NOI then takes the form of LOI. This also means that all interactions with the local residing kernel happen via LOI too. Thus, instead of ROI, as specified by the design (see Figure 2.4), the more efficient LOI alternative is applied in that case. Note that, in this case, no address space boundary exists between higher-level entities and the nucleus. Since the nucleus is part of the kernel address space, sacrificing nucleus isolation also implies sacrificing kernel isolation. As the kernel "uses" the nucleus, and thus depends on its correctness, it makes little sense to isolate the kernel in this situation. Consequently, locally invoking the nucleus or kernel in the single-tasking configuration causes no context switch.

In the single-tasking configuration, the kernel is of *hybrid structure*. Locally issued requests are handled on an LOI basis as mentioned above. However, since there might be remotely residing system components requesting the allocation of local kernel resources (e.g., enabling the interaction with devices attached to the node), kernel services must be provided via ROI also. In other words, there are different invocation schemes existent at the same time to interact with the kernel. This aspect is discussed in detail in the next section.

2.3.3 Kernel Entity Configurations

The introduction of VICE, and LOI as the alternative representation of NOI and ROI, creates various kernel configurations. These configurations implement a single and a multiple address space model for a node. The corresponding refinements of the kernel entity are discussed in the following paragraphs.

A refinement of the kernel entity for a single-address-space configuration (Figure 2.5) reveals that neither the kernel nor nucleus must be specifically implemented for this case. In this configuration, the kernel entity provides three service access points, one for the nucleus and two for the kernel. The nucleus access point and one of the two kernel access points are implemented by LOI, whereas the second kernel access point is implemented by ROI. Kernel access on the ROI basis then means requesting a *clerk* thread to perform LOI on behalf of the calling client thread. That is, the kernel access point is represented by one or more lightweight processes (i.e., active objects), depending on how many services the kernel must actually provide and whether some of these services shall be executed concurrently.

The *object invocation layer* (OIL) shown in Figure 2.5 plays a role similar to VICE (see Figure 2.2). It defines a "sandwich," with the kernel

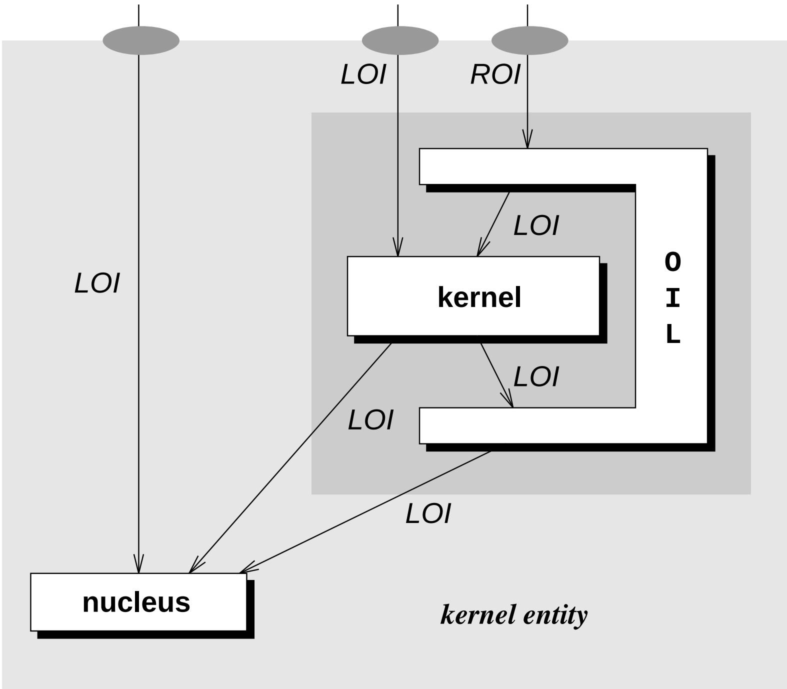

Figure 2.5: Kernel entity in a single-address-space configuration

as "spread."[11] OIL makes the kernel configuration-independent. As part of OIL, the clerk implements the externally and remotely accessible kernel interface. Kernel access to the client address space is handled by dedicated OIL functions, too. Whether client access is done directly (through the nucleus) or indirectly (through OIL) is an attribute of the objects on which either LOI or ROI is performed. In the LOI case, the nucleus is in charge of transferring data between client and kernel entity. In the ROI case, this becomes the responsibility of OIL itself.

A refinement of the multi-address-space configuration (Figure 2.6) shows only two service access points: one for the nucleus and one for the kernel. In this configuration, the kernel is isolated from higher-level

[11]This technique is used whenever system components need to be logically addressed on an ROI basis. See also Section 2.2.4.

entities. NOI and ROI are required to invoke the nucleus and kernel, respectively. Again, the implementation of nucleus and kernel does not change in order to support multiple address spaces. Rather, the entire kernel entity is (horizontally) extended by appropriate service functions without affecting its present implementation. Even the implementation of this new service does not depend on protected address spaces, merely on the availability of a *memory management unit* (MMU) to enforce kernel entity isolation. The MMU service partially implements an extension of those VICE functions used by the nucleus to transfer data across protection domains (refer to Section 2.2.4).

In a multi-address-space configuration, ROI must be executed to enter the protection domain of the kernel. This also holds when threads are collocated with the kernel entity on the same node. In this configuration, the kernel entity takes the form of a *microkernel architecture*. The objects representing POSE services are encapsulated by their own protection domains. They become active objects. A POSE service, then, is typically represented by a team of lightweight processes, with the team executing in a private address space. These teams will then have to compete with one or more application teams (i.e., tasks) for processor allocation.

The two configuration modes are able to coexist in a multinode environment. This is illustrated by Figure 2.7. In the single-address-space (or single-tasking) model (left node in Figure 2.7), the kernel anticipates a *hybrid structure*: all local requests are handled via LOI, and all remote (incoming) requests via ROI. Thus, even in a single-tasking mode of operation, kernel services can be invoked remotely. LOI is used for threads that are collocated with the kernel entity. An additional ROI interface makes the same kernel entity globally accessible by remotely residing threads.

The right node of Figure 2.7 shows a typical microkernel architecture. A POSE process manager service, for example, is able to control active objects executed by the single-tasking (left) node. Similarly, a POSE file management service residing on the right node is enabled to access a disk device that may be attached to the left node. Another example is a POSE service implementing virtually shared memory on a paging basis. This service, allocated to the right node, will be able to manipulate the page table located at the left node through the MMU

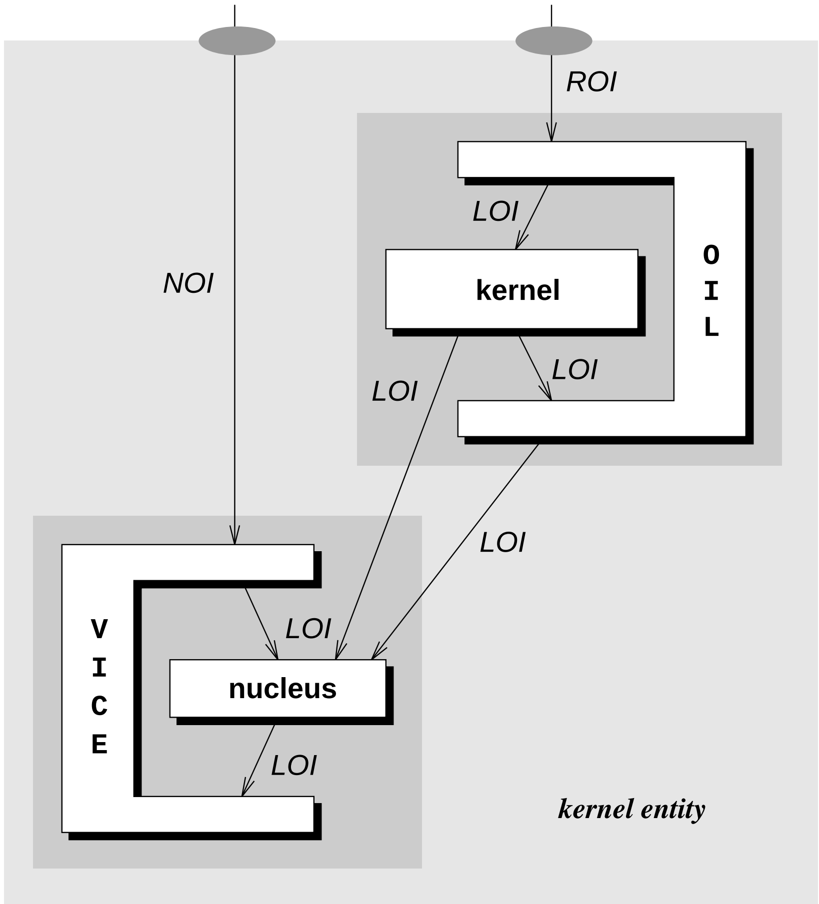

Figure 2.6: Kernel entity in a multi-address-space configuration

service provided by the kernel of that node. Last but not least, the application task(s) mapped to the right node may control the processing of the application task mapped to the left node, provided that all tasks belong to the same user domain. Since the right-node kernel offers an ROI interface, all of its services are accessible from the left-node entities too.

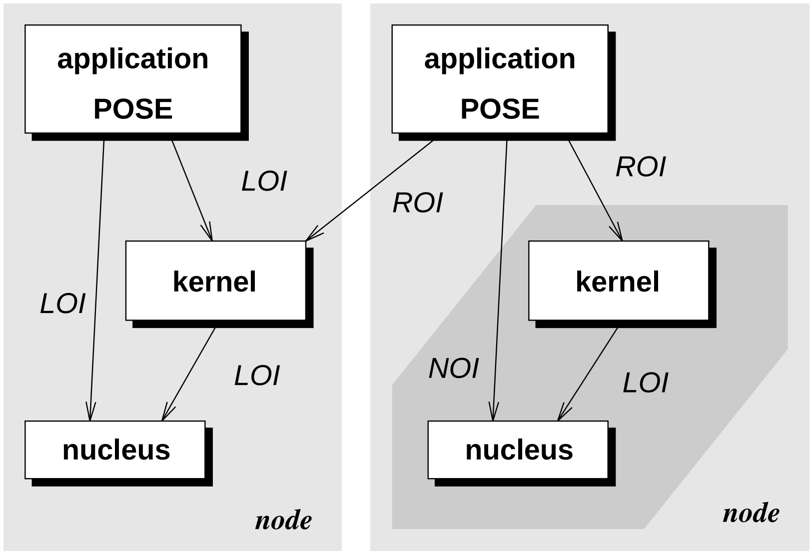

Figure 2.7: Single- vs. multi-address-space model

2.3.4 Multithreaded Kernel Entity

If a node is controlled by a kernel entity in addition to the nucleus, a multi-threaded address space model is required to invoke kernel services. The reason for requiring multiple threads of control is not only because ROI interfaces are typically controlled by lightweight processes (i.e., *clerks*). The main reason is that even in the single-address-space model, nodes must be remotely controllable, for instance, to aid debugging or to enforce bootstrapping ("cold startup") and reinitialization ("warm startup") of the node. In that case, to avoid depending on the application thread, some separate thread of control is needed on this node as linkage to a remotely residing component.

The thread of control used to manage the kernel ROI interface must not necessarily be implemented by a lightweight process; however, this greatly improves the system structure. No other specific mechanism is needed; the only purpose of which would be to remotely invoke kernel functions. The *V* kernel [16], for example, provides such a specific mechanism. It uses predefined process identifiers to address *pseudo processes* residing in kernel space and providing kernel services to user tasks. The inter-process communication layer of the *V* kernel interprets these identifiers differently than those that designate real processes. It calls the associated module when a message tagged with such an identifier comes in, and it does not deliver the message to a real process.

This approach interlocks the communication layer with the kernel in a way that is not advantageous for the system structure. The nucleus would have been assigned to "use" the kernel, which already "uses" the nucleus. "Sandwiching" becomes necessary to resolve this cyclic uses relation [67]. Despite these additional structuring measures, the minimal subset of system functions (as provided by the nucleus) would no longer be minimal. It then must be extended by exactly those kernel functions that are remotely invokable (see also Section 2.2.1).

2.4 Microstructure

The minimal subset of system functions defined so far is a compromise between scaling transparency and efficiency. The compromise was made with respect to the interface specification of the abstract data type which represents that subset. The internal behavior of this data type is manifold and basically distinguishes between the single- and multitasking mode of operations. That is, the functions provided by the minimal basis and specified by the interface of the abstract data type(s) define a "minimal but perfect" subset. It is *minimal* because it does not anticipate any particular design decision that would be a handicap for the implementation of higher-level user/system abstractions. It is *perfect* because it already supports a large number of parallel applications.

Offering dedicated single-tasking and multitasking kernel implementations enables the user to deal with the trade-off between performance and functionality on an individual basis. Parallel applications whose tasks can be mapped in a one-to-one correspondence with the nodes are not punished by multitasking overhead, because these applications can be supported by a single-tasking kernel. Experiences have shown that this overhead may take up to 74% of the time used to execute a user-level (networkwide) *send-receive-reply* sequence [78]. Table 2.1

shows how the 74% multitasking overhead is distributed over typical microkernel functions. Note that a state-of-the-art microkernel, however, supports only multitasking, and thus unnecessarily draining computing power from a single-tasking application.[12]

Multi-Tasking Overhead	
Operation	*Proportion*
inter-domain calls	5%
vector unit housekeeping	8%
task scheduling	14%
interrupt handling	22%
packet movement	25%
total	74%

Table 2.1: Message startup time problem

The minimal subset of system functions is represented by a *nucleus family* that implements four different modes of operation (Figure 2.8).[13] Each mode is represented by a subfamily. The entire family tree shows different nucleus versions, with the root (top) the simplest and the leaf (bottom) the most complex instance.[14] As functionality and complexity increases, overhead, and thus communication latency, increases.

[12]For the SUPRENUM implementation of PEACE, which is based on a 2-MIPS MC68020 processor, systemwide bidirectional inter-process communication using a 64-byte packet takes about 784 μs in the multitasking mode and 204 μs in the single-tasking mode.

[13]Also referred to as "*kernel family*" or "*microkernel family*." Nevertheless, the notion of a "nucleus family" is more appropriate since the variety of the nucleus implementation causes the different operating modes of a node. In contrast, depending on the supported operating mode, kernel service functions are either present or missing. For example, multitasking requires kernel functions to create and destroy address spaces. These functions are not present when the nucleus implements single-tasking.

[14]The arrows in Figure 2.8 indicate an inheritance relationship between the members of the nucleus family. However, not classes (or objects), but functions, are set in relation to each other. The nucleus family tree shows the functional relation between family members.

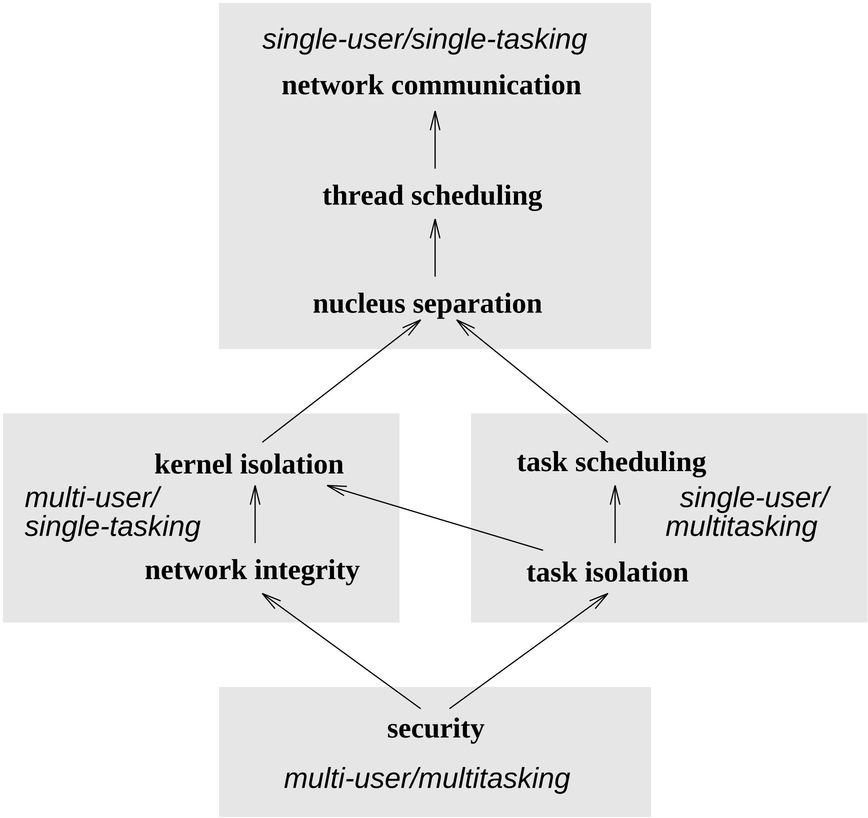

Figure 2.8: Nucleus family tree

This organization of the nucleus, and hence the kernel entity, is one of the major differences from state-of-the-art microkernel designs. The kernel entity defined is not a single microkernel, but a "*microkernel family.*" It can be adapted to the individual needs of parallel and distributed applications by providing family members that exhibit different system characteristics regarding functionality and performance.

2.4.1 Single-User/Single-Tasking

The root of the nucleus family tree shows three different members supporting the single-user/single-tasking mode of operation, with tasks possibly being multithreaded. This mode of operation allows for only one user application at a time and merely supports the processing of one task per node. On every node, a single-address-space model is implemented. Consequently, the entire parallel machine is subjected to batch processing. This enables the most efficient execution of the parallel program. At the same time, it may lead to the most inefficient utilization of the parallel machine, since the machine cannot be shared between a number of applications.

Minimal support for (distributed-memory) parallel applications means providing *network communication*. This feature is provided by the most fundamental member of the nucleus family. At this stage, thread dispatching is nonpreemptive and completely under the control of the application task. This nucleus representation is extremely lightweight and supports the notion of *featherweight processes*. Featherweight processes are a scaled-down version of lightweight processes. Each of them implement the purest form of a unit of execution, without any consideration of protection and security measures. NOI is technically represented as LOI.

Thread preemption is the minimal extension to network communication. This results in *thread scheduling* on a timer basis, actually introducing a second scheduling level. The thread-scheduling member implements CPU protection [69], since a single thread (or task) is no longer able to monopolize the processor. The additional level knows the *bundle* as the scheduling unit and allows safe integration of kernel-level lightweight processes that implement the service access point for ROI. These threads are then combined into a *kernel bundle*. All other threads form the *application bundle*. Since bundles are subjected to preemptive scheduling, kernel service processing works independent from the processing of the application task. Consequently, threads no longer execute under the control of the application task but of the system (i.e., nucleus). This ensures that remotely requested services are processed the latest, when the thread scheduler has gained control and selected a clerk for execution—provided that the application task sharing the same address space with the kernel entity is "well-behaved."

The next nucleus family member does not really introduce any additional functionality but provides the platform for the multitasking and/or multi-user mode of operations. These modes of operation call for isolation and protection measures. Thus, the *nucleus separation* member introduces NOI and ROI patterns as minimal extensions to thread scheduling in order to request services from the nucleus and the kernel, respectively. This kernel family member relies on inter-domain calls (refer to Section 2.2.4) to invoke the nucleus.

Higher-level entities are physically uncoupled from nucleus code. Since every nucleus and kernel takes the form of an abstract data type, these entities are also logically uncoupled from kernel-level data. This supports the dynamical change of the operating mode of a node by replacing the actual kernel entity with a different implementation. However, note that only code separation is supported, but not memory protection. As a consequence, the passing of complex data structures between the kernel entity and higher-level entities is still straightforward and involves no MMU programming.

2.4.2 Multi-User/Single-Tasking

In a distributed-memory parallel machine, the multi-user mode of operation is feasible even if only a one-to-one correspondence between tasks and nodes is supported. The entire multinode machine can be allocated to different users at the same time. For this purpose, different *user partitions* are provided, with every user application executing within a private partition. This does not require local ("on-board") security measures to protect the tasks from each other, but it requires network interface protection. With direct network access, the user task would be able to intrude into the network, thus into tasks mapped to different user partitions. In principle, this violates the integrity requirements of each of the user applications.

In order to provide the multi-user function, the kernel entity must be entirely isolated. Memory protection is introduced as a minimal extension to nucleus separation. This yields a new family member that provides *kernel isolation*. Because the nucleus is part of the kernel entity, applying memory protection to the nucleus also implies the isolation of (at least parts of) the kernel address space. Vertical isolation is applied to protect the kernel entity. In this case, nucleus invocation (i.e., crossing the trap interface) does not usually increase the overhead. However, the passing of data structures becomes more heavyweight. It mainly depends on the MMU whether or not additional overhead is introduced (see Section 2.6.1). In any event, the increase of nucleus functionality is accompanied by a potential loss of communication performance.

Once the kernel entity is protected, tasks are no longer capable of intruding into lower-level software components thus bypassing protection domains. However, in order to communicate, tasks need to know communication endpoint (i.e., thread) identifiers. These identifiers must be passed down to the nucleus along with a communication request. In a multi-user configuration, tasks mapped to different user domains must therefore be prevented from being able to successfully use communication endpoint identifiers of other domains. This is the purpose of the minimal extension to kernel isolation, called *network integrity*.

With network integrity, communication firewalls between different user applications are introduced. Every user application builds a unique communication domain. The same holds for the set of system processes that constitute the operating system. Within the same domain, communication is unlimited. In order to invoke system services, however, application processes must communicate with system processes. The changing of communication domains must be possible. This requires different communication domains to overlap in order to enable communication. Hence, communication security does not mean complete isolation, but rather controlled access.

To model and enforce protection domains in the above-mentioned sense, a capability-based approach [28] proves to be the appropriate solution. This approach grants object access only if a thread (i.e., subject) has possession of that object or one of its proxies. An object must be created and bound to a systemwide unique identifier before it can be used. It is assumed that a unique identifier cannot be deduced [58]. In order to achieve this, the nucleus generates a random number which, combined with a global hash key, is used to make identifiers systemwide unique. Note that this procedure works autonomously and need not be controlled by a central system component. Once instantiated, the creator implicitly possesses the new object and henceforth acts as its owner. The owner has unlimited access rights to that object. It is the autonomous decision of the owner to make the object either globally known, by exporting its unique identifier, or even accessible, by exporting its *proxy object* [61]. Thus, the access domain of an object may be extended only by the owner. Consequently, the nucleus need not verify that communication to an (active) object is allowed but only whether this object exists—knowing the capability of this object enables communication. Verification of the existence of an object then means checking for the validity of the systemwide unique identifier of the object. This means that the nucleus only has to deal with these unique identifiers and not with the object actually being communicated with.

2.4.3 Single-User/Multitasking

The scheduling paradigm introduced by the thread-scheduling member does not distinguish between multiple autonomous program entities. Mere execution of a single application program (i.e., task) is supported. Having introduced nucleus separation, the processing of several application programs becomes possible by slightly extending the scheduling functions. Note that the nucleus can easily be shared by a number of application tasks. It exploits either a trap mechanism or a statically known "jump table" (see Section 2.2.4) to provide a common entry point for independently generated programs.

A task is represented as a *team* of lightweight processes, with the team possibly exhibiting several thread bundles that can be autonomously scheduled. The team owns resources such as message buffers, memory space for stacks and process descriptors; *task scheduling* then means keeping track of the resources owned in order to select the next task ready to run. A scheduling discipline that takes into account the current load and thus tries to improve global system utilization may be favorable. This further calls for minimal scheduler extensions to those already provided by nucleus separation.

At this stage, multitasking can be readily supported even if private address spaces are not provided. A private address space serves two basic purposes. On one hand, it implements memory protection, isolating programs from each other. On the other hand, it defines a logical address space in which to execute a program, enabling code/data relocation at runtime. Being relocatable is also a property of *position-independent* code, which then must to be generated by a compiler. In addition, the use of secure programming languages supports program isolation without the necessity of address space protection hardware.

For all these reasons an intermediate step is made, introducing the task-scheduling member as a platform for multitasking. A minimal extension to this platform is introduced with *task isolation*, which encorporates vertical and horizontal isolation. This extension is used to generally improve system availability and in those cases where neither the programming language nor the compiler provides adequate support for a single-address-space multitasking mode of operation. Note that task isolation extends not only task scheduling but also kernel isolation. The latter family member already provides vertical isolation to protect the kernel entity from higher-level user entities. With task isolation, the nucleus is also functionally enriched by measures of horizontal isolation to protect user entities from each other.

2.4.4 Multi-User/Multitasking

By combining the functions of network integrity and task isolation, a new nucleus family member emerges. This member introduces multi-user *security*, leading to the fourth operation mode supported by the kernel family. Multi-user security in a distributed environment demands communication firewalls and protected address spaces. The former is achieved by ensuring network integrity and the latter by enforcing task isolation. The peak of system functionality has been reached; however, only at the cost of system performance.

2.5 Communication System

The nucleus performs networkwide message passing between processes and address spaces. Hence, it encapsulates a *communication system* for this purpose. The communication system is asked to keep track of interactions between remote processes (i.e., active objects), execute some sort of transport protocol, and control the network hardware interface.

Communication performance is the most crucial factor in massively parallel systems. That makes it especially difficult to take advantage of communication protocol architectures which have been developed for distributed systems [87]. These architectures tend to be too overhead-prone for the type of parallel systems considered here. Even if performance were not the predominant issue, the principles of program families would be violated if unneeded functions were designed into the communication system. In order to figure out what functions the communication system should provide, an investigation of both the application and architectural demands of massively parallel systems is required.

This section starts with a discussion of the network capabilities of massively parallel systems and makes a comparison with the protocol features discovered in computer networks. This is followed by some thoughts on heterogeneity and underlies designing a lightweight communication system structure for (massively) parallel systems.

2.5.1 Network Capabilities

A communication system for distributed-memory parallel machines has to provide *transport protocol features*, but it need not provide these features for every type of application and every kind of network. The relevant features are:

- Data security
- Segmentation and reassembly
- Flow control
- Sequencing

The following discussion is concerned with whether a software implementation of the above-listed protocol functions is really mandatory in the context of parallel computing (i.e., parallel operating systems). One must keep in mind the extremely tightly coupled nature of distributed-memory parallel computers. This is where the main difference from classical distributed systems lies. Very high quality, functionality and performance are the outstanding properties of networks that are used to interconnect the nodes of a parallel computer architecture. Therefore, a typical declaration of what the design of the communication architecture is concerned with is the following [2]:

> Networks in massively parallel systems correspond to the vector unit of classical parallel machines, with the network interface being the processor registers. Both are inherent system components that must be made directly accessible to the application programs. Just as vector machines provide dedicated arithmetic instructions are massively parallel machines required to provide dedicated communication instructions. Compilers should be able to take advantage of the enhanced instruction set and generate code to exchange messages without operating-system intervention.

It is obvious that this opinion calls for specifically designed hardware platforms, but not every state-of-the-art distributed-memory parallel computer provides such capabilities. However, before a new "hardware battle" is fought out, one first has to look for properly designed software structures. Even an abstract processor may perform well, at least to identify those functions that are worth, and capable of, being migrated down into a hardware design. As the discussion will show, it is mandatory to have a *protocol family* from which members can be selected to enable the tailoring of high-performance communication systems. This also enables real-life experiments in which applications are supported by extremely scaled-down communication functions, and helps to find out how far functional degradation can be pushed forward in the attempt to increase performance while keeping parallel applications running.

Since communication performance is predominant in distributed-memory parallel machines, it is important to identify a minimal subset of system functions that can be subjected to downward migration. It might even be the case that the minimal extensions to the minimal subset are downward migratable too. The size of the *semantic gap* closed by these extensions (i.e., the added functionality) determines whether a hardware solution becomes feasible. Thus, approaching the family concept in the design and development of communication functions on one hand strongly promotes dedicated hardware solutions, and on the other hand presents an overall software framework into which these hardware solutions can be neatly integrated.

2.5.1.1 Data Security

Typically, *data security* tackles the problem of noisy communication wires. Message packets are provided with redundant information, enabling *error detection* or even *error correction* by the receiving site. A common technique is the *cyclic redundancy check* (CRC). The sending site generates the CRC by passing each data word through a CRC generator and transfers it as a message trailer. At the receiving site, the incoming message is processed following the same procedure. The generated CRC is then compared with the received one, indicating a transmission failure if the check delivers an error. Normally, all these activities are performed by hardware.

In state-of-the-art network technologies such as fiber optic, noisy communication wires are no longer a serious problem even in the distributed systems area. In parallel systems, it is not a problem at all. These systems make the probability of data falsification in network communication comparable to the probability of bit flaws in local memory-to-memory transfers. It can actually be neglected. A standard *data link layer* that takes care of secured bit stream transfer is not mandatory.

With this background, there is only one situation in which the necessity of data security arises, namely, when a parallel operating system is either ported onto or developed for a newly designed hardware platform. If data-link-layer functions are not provided by the hardware, they will be provided by low-level software modules and only for debugging purposes. Thus, even if the final system does not care about data security, intermediate prototypes will have to do so. Otherwise, the development of a parallel machine, including hardware and software, is likely to become a knotty point. It is also worth noting that it is the software which needs to be properly structured to aid the debugging phase. A data link layer is to be provided as a member of a protocol family. This layer only comes into play upon demand and, under normal circumstances, is not in charge of any communication activity.

The need for providing a means by which data security measures can be sacrificed was already discussed in Section 1.2.2. Continuous multimedia applications are indicative of somewhat "revolutionary trends" in the traditional communication protocol sector. State-of-the-art hardware technology makes low-level functions used for a secured bit stream transfer almost superfluous. The most important aspect is that these functions, if not temporarily disconnectable, impair user acceptance of multimedia products significantly. This situation is directly transferable to the parallel computing area.

2.5.1.2 Segmentation and Reassembly

Falsification of data can be compensated for by means of error-correcting codes, but lost messages cannot be reconstructed from anything. As a consequence, messages must be retransmitted, network load increases, and net bandwidth drops. In addition, the number of node-bound activities increases. These activities will have to take care of buffering, timeout handling, retransmission and receive acknowledgment. This further raises the probability of discarding messages because of temporary shortages such as:

- A receiver process that, although known to the system, cannot be immediately dispatched to accept the message
- Insufficient buffer space to store (at least) part of the message until the receiver process gets ready for message acceptance
- A lack of buffer descriptors (and not of buffer space) which are needed to direct the network hardware interface or the network driver as to where to place incoming messages

These temporary shortages cause a further increase of the above-mentioned node-bound activities used to carry out a data transport protocol. In addition to resource shortages, two other aspects significantly impair overall protocol processing. These aspects are:

- A nonpreemptive kernel that is not able to react promptly on events (i.e., interrupts) signaled by the network hardware interface when messages are pending for receive.
- A poorly performing processor—but this tends to be one of the minor problems, since all other items listed before, and thus the majority of problems, are due to software shortages.

In order to tolerate a sporadic resource shortage at the receiving site of a message, *segmentation* and *reassembly* of message streams is required. The sender site cuts arbitrarily sized messages into little pieces (i.e., segments or packets), which are transferred in sequence and *flow-controlled* through the "ether." At the receiver station, incoming packets are reassembled to reconstruct the original message. The flow-control aspect, although closely related to segmentation and reassembly, is discussed in the next section.

Performance Bottleneck Problems. In order to tackle resource shortage problems, a common approach is to rely on a *sliding window protocol* [87]. This protocol enables the sender to transmit a number of packets in advance without requiring a receive confirmation. Confirmations are transmitted back by either "piggy-backing" them on a packet that is going to be sent to the original sender station or replying with an explicit confirmation packet at the latest time, when the window size is about to get exhausted. Message and control packets whose receipt is not acknowledged within a defined time interval are retransmitted.

Additional overhead is introduced to reduce the problem of discarding an entire message to the problem of discarding single message packets. With large messages and, compared to processor speed, slow networks, segmentation and reassembly is a suitable technique to significantly speedup communication performance in the presence of transmission faults (i.e., non-correctable bit flaws or resource shortage). With messages that rarely get much larger than the packet size defined by the communication protocol, nothing but overhead is gained. Furthermore, nowadays network technology allows transfer rates in the gigabit range; this holds particularly for massively parallel systems. In these networks, the overhead for segmentation and reassembly may have a significant drawback on the overall performance (i.e., the message startup time). Hence, it might be much better to apply some sort of *blast protocol* [98] for transferring arbitrarily sized messages in one chunk and awaiting a single confirmation for the entire transaction.

In order to prevent resource shortages during the transfer, the blast protocol assumes some kind of connection between the sender and receiver stations. However, taking into account the runtime properties of many SPMD-type applications, it is not required to explicitly establish connections before communication may take place [72]. In many cases, the algorithm of the parallel program implicitly defines a "*logical connection*" at the right moment. This moment occurs when the tasks leave (the computation burst) and enter the communication burst. Especially in *nearest-neighbor* configurations, the communication burst means that a sender task can rely on the readiness of a receiver task to immediately accept messages. The entire message stream can be "blasted" without the risk of retransmitting the message.

Provided that such regular processing of parallel programs cannot be assumed, nonfragmented and "connection-oriented" data transfer is only applicable when dealing with relatively large message streams. In that case, the time it takes to establish a connection can be ignored. The message startup time, which is dominated by the connection establishment phase, will become the predominant performance-critical factor with message length getting shorter.

Network Blocking Problems. Segmentation and reassembly may not only help to prevent a communication performance bottleneck that may arise due to message packet loss at the receiving site. Both functions also significantly improve the blocking behavior of the network. Network blocking proves to be a serious problem even when exploiting high-speed communication hardware technology. In a *circuit-switching* network, blocking delays are directly proportional to the message length (i.e., the transfer time). Circuit switching implies the establishment of a (physical) connection between the sending and receiving nodes. The connection may span several hops and, in parallel architectures, is used to blast the entire message across the network. Connection teardown takes place when the message transfer has finished. The problem now is that, as long as the connection is active, the route between source and destination node can be neither shared nor crossed by other communication requests. Consequently, communication requests that are independent of each other and therefore could be processed in parallel get unnecessarily serialized by the network. This increases the turnaround time for message transfer jobs and decreases communication throughput.

The problem of network blocking in a circuit-switching environment can be lessened by establishing the connection for a single message packet only. Segmentation and reassembly of messages is then used as a technique to introduce small and bounded blocking delays. In addition to that, the network is enabled to transfer several messages (virtually) in parallel. Basically, this approach implements *packet switching* on top of a circuit-switching network to improve overall throughput.

When relying on a (virtual) packet-switching network, *virtual channels* can then be introduced to relate a sequence of packets to the same message transaction. A virtual channel is nothing but a systemwide unique identifier used for resource management purposes. Every packet is tagged with a channel identifier to tell the receiving site to which message stream that packet belongs and which state descriptor is to be used to control message reassembly and delivery. Since communication is between threads or address spaces related to threads, a systemwide unique communication endpoint identifier associated with the receiver threads may already serve as the virtual channel identifier. As the communication endpoint identifier must be known before the message transfer request can be issued by the sender thread, no virtual channel identifier need be generated dynamically with each message transaction. This generally reduces message startup time costs.

2.5.1.3 Flow Control

Segmentation and reassembly supports *flow control* between stations that are involved in the exchange of messages. Message packets will only be sent when they are likely to be accepted by the receiving site. That is, in the case of a temporary disability of the destination node to receive incoming packets, the sender is stopped from issuing further packets. For example, suppressing receive confirmations in a sliding window protocol when the window size gets exhausted will automatically delay the transfer process. A *credit mechanism*, by which the sender site is told how many packets can be transmitted to (and buffered by) the receiver site without delay, provides similar functions [87].

Flow control in networks of massively parallel systems is generally performed by hardware. During the data transfer phase, sending and receiving DMA or FIFO units run synchronously with each other. Basically, "handshake signals" are set to indicate when to transfer the next block of data. Thus, once the hardware is provided with a set of DMA descriptors (e.g., for both the sending and receiving site), flow-controlled data transfer works without software intervention. The transfer speed is then limited only by either network or memory bandwidth. However, the problem is to feed the hardware with a proper number of DMA descriptors such that data transfer need not be stopped. This (lower-level) problem can be mitigated if the compiler used to translate the (higher-level) application program performs *instruction scheduling*, as typically done for super-scalar processors, and the communication system provides dedicated methods for the user-level management of DMA descriptors. Logically, the network (including hardware and software) implements a pipeline. The goal would then be to try keeping the pipeline running by compiler-controlled restructuring of the application program. This requires global data flow analysis of a distributed, parallel program, and hence becomes by far a nontrivial task. A network hardware interface providing the user/system programs with FIFO registers to send and receive messages (or message fragments) overcomes the problem of guaranteeing the availability of DMA descriptors at the receiver site to allow a continuous data transmission. Nevertheless, even a FIFO register store is not of unlimited size.

It turns out that flow control is not a predominant issue in designing a (software) transport protocol entity for parallel architectures. It suffices to rely on *back-pressure flow control* [87], which is readily achieved when receiver processes do not accept, and thus delay, the delivery of messages. The network DMA logic at the receiver site will stop its peer at the sender site when a shortage of DMA descriptors occurs. Running out of DMA descriptors may be caused, for example, by a receiver process that does not remove messages from the low-level network driver input queue. The above-mentioned sliding window protocol technique of temporarily suppressing the confirmation of received packets is another example of the back-pressure flow control approach. Similarly, "inter-FIFO communication" may take place on the basis of hardware signals indicating an overflow condition. The potential for overflows arises due to a process that is not ready or even fast enough to receive data. This process then causes back-pressure flow control to become effective between the sending and receiving FIFO logics.

2.5.1.4 Sequencing

Using a window mechanism and based on a network that may provide several routes from a source to a destination node, packets sent in sequence may arrive out of sequence. This requires the sending site to tag every packet with a sequence number, and the receiving site to reestablish the correct relative ordering of all the packets describing fragments of the same message by observing the sequence number. The problem of different packets of the same message taking different routes exists with either load-balancing or fault-tolerance functions.

Load-Balancing Issues. In the load-balancing case, the network of a (massively) parallel system must be intelligent enough to automatically perform packet routing and thus packet sequencing. Network load balancing may become an operating-system issue (more specifically, an issue of the embedded communication system) only when it proves to be applicable before the entire message is sent. As a prelude to the message transfer, the communication system will then have to compute the currently "optimal" route. Once the message transfer is in progress, the route remains fixed for all packets sent out of the node. This corresponds to a connection-oriented model in which, for every message transaction, a new connection is established and torn down afterwards. The message route is then computed during the connection establishment phase.

Computing the route prior to each message transfer is quite feasible, but it also introduces a significant performance bottleneck and increases the message startup time. Also note that dynamic load balancing through a reorganization of the process topology (i.e., using process migration) aims at achieving a different network load, too. In this case, message routes will usually change due to the different nodes to which the processes are mapped after migration. If process topologies are likely to change often, load balancing at the message level (i.e., within the network) becomes more or less obsolete, as the way processes are mapped to the nodes of the parallel machine primarily determines the actual network load.

All these aspects lead to the conclusion that packet sequencing must not belong to the minimal subset of software protocol functions. Rather, packet sequencing may be provided only through extensions to that subset.

Fault-Tolerance Issues. The second reason for routing and sequencing is fault tolerance. If a node crash were detected and application processing must be maintained even in that situation, the system needs to perform process migration, and thus changes the original process topology anyway. It will perform the migration procedure by querying a load-balancer subsystem to determine the most favorable nodes to host the processes that must be moved. The same occurs when either a node crash or communication link breakdown is detected while a communication activity is still in progress. In both cases, the (hardware/software) transport protocol notices that packet confirmations have not been received for a while. If even retransmissions are not succeeding, the protocol will raise a communication exception. The exception is typically handled by the fault-tolerance subsystem, which may then proceed in a manner similar to the case described above. It may probe the network to figure out the reason for missed confirmations and then try to initiate a recovery procedure.

The result of exception handling will be to either abort and restart or resume the communication activity in trouble. The former approach (abort and restart) will not cause packets of the same message going on different routes, since a restart implies the retransmission of all message packets in sequence. In contrast, the latter approach (resume) may cause packets to pass other packets. However, even in this case, it is very unlikely that packets run out of sequence, as the reconfiguration procedure takes orders of magnitude more time than supplying the network with new packets.

Message Integrity. There is a risk of packets being received twice at the destination node if a communication activity is either restarted or resumed. This can be regarded as a specific form of packets having been run out of sequence and must be solved by adding a *duplicate suppression protocol* to the communication system. On detection of a communication link or node breakdown, it is uncertain whether unconfirmed packets have been received. Retransmission of these packets will thus potentially produce duplicates at the destination node. Duplicates must be suppressed by the receiving site, as only this site knows what really came in. Any form of retransmission strategy may cause duplicates, independent of whether the entire message is retransmitted after restarting the aborted communication activity or packets are retransmitted after resuming this activity.

The handling of out-of-sequence packets is not a problem of maintaining sequence numbers but one of ensuring *message integrity*. Sequence numbers already play a role in the process of segmentation and reassembly. Thus, they have to be maintained anyway. Ensuring message integrity implies packet buffering. The message is delivered to the receiver process only when all packets have been received in sequence. This raises buffer management problems, provided that packet buffering is done within the communication system. Message reassembly (and buffering), however, could also be performed directly within the address space of the receiver process. In that case, message delivery actually means signaling the process that more data is pending for consumption. It no longer means transferring the data into the destination address space. From the communication system point of view, this procedure helps to avoid any packet buffering. Logically, packets are directly transferred from the network into the receiver address space. This may be accomplished by either network DMA or memory mapping.

Even in that case, the buffering of messages becomes unavoidable, unless specific programming paradigms and/or hardware support is provided. In many cases, a message is not the appropriate data object to feed directly into a computational process. The program executed by that process knows about variables of a specific type rather than untyped byte streams (i.e., messages). The message might contain vector elements (i.e., floating point numbers and the like), which need to be extracted before processing is possible. This implies (fine-grain) data copying out of the message into program variables. Thus, although the system performed unbuffered data transfer, the application program would have to read data from a "message buffer" in order to assign that data to program variables. From the application viewpoint, copying data from the network directly into these variables would be the better way to go. This is one reason why the parallel computing community calls for innovative programming paradigms and specifically designed networks and network interfaces for distributed-memory parallel machines.

2.5.2 Heterogeneity

Distributed-memory parallel machines are assigned to act as accelerators for specific types of applications. Thus, these machines are not necessarily autonomous. They often depend on the availability of a host computer. The host computer is then responsible for the global management of the parallel machine and also supports the development of parallel programs. In this scenario, the parallel machine takes the position of a special hardware device such as a floating point unit: it will be exploited only by dedicated application programs.

This sort of system organization potentially raises heterogeneity problems, although the parallel machine is of homogeneous nature. The problems may come into existence at both hardware and software level:

- The host computer is based on a different processor type than the parallel machine. Each of these processor types may introduce a different style of data representation. They may have a different byte order.
- The host computer is run using a different operating system than that provided on the parallel machine. These operating systems may provide a different set of services and make these services available through different interfaces. In particular, object-oriented and non-object-oriented operating-system interfaces may exist.

What impact these problems will have on the design of a parallel operating system greatly depends on the usage pattern of the parallel machine. Two main organizational forms are distinguished, depending on whether only the parallel machine or both the parallel machine and the host computer process application tasks.

2.5.2.1 Different Network Subsystems

If all application tasks are executed by the parallel machine only, the heterogeneity problem must (and can) be completely solved at the system software level. In this configuration of a parallel machine, also called *hosted system*, the application tasks have to communicate with the host computer only to request certain system services. That is, the communication between possibly heterogeneous systems would take place via a well-defined interface at the system level. However, this assumes that I/O produced by the application tasks is entirely processed by the parallel operating system. In that case, either the host computer is used as an intelligent I/O device or the parallel machine is equipped with proper I/O devices. Roughly speaking, the host computer is merely the console, used to interact with the parallel machine.

In this system configuration, *representation-independent communication protocols*[15] are needed to interconnect the host computer and parallel machine. The parallel operating system then has to provide a *gateway service* that maps commands issued by some host computer process (e.g., the shell or command interpreter) onto commands provided locally on the parallel machine and vice versa. More specifically, two different network subsystems are logically present: the "network" defined by the host computer[16] and the network used to interconnect the nodes of the parallel machine. A *heterogeneous network* exists, and in some sense, typical internetworking services are required.

Taking a more detailed look at the interaction pattern just mentioned, one will discover that providing representation-independent communication protocols alone is not sufficient. Each of the commands sent back and forth between both computers is nothing but an RPC. The crucial point here is how to represent parameter values of remote procedures and model a common naming domain. The parameter values may be of any type, depending on the procedure (i.e., service function)

[15]TCP/IP is a well-known communication protocol of that kind. However, this does not mean deeming TCP/IP the protocol of choice to make distributed-memory parallel machines work.

[16]With workstations serving as host computers, the notion of a "network" used in this context is more obvious. The host computer defines a different domain, and this may or may not imply the presence of an additional network system. Logically, the same problem exists; namely, to interconnect the host to the parallel machine by means of some network structure. In the system organization discussed here, "standard" networking techniques such as Ethernet or FDDI are assumed for this purpose. In almost every case, these techniques differ from those applied in order to establish a distributed-memory parallel system.

invoked at either site. As the RPC layer knows which member of an RPC message is of which type, a typical solution is to have the RPC client stub stamp that message with the byte order (i.e., CPU type) of the sending site. Only if the receiving site has a different byte order will the RPC server stub have to take care of data conversions. Exploiting stub generation tools is a common technique even in this case and improves the programmability of parallel machines in general.

2.5.2.2 Integrated Systems

The most common approach to parallel processing, however, is to have application tasks executing on the host computer also. Each of these tasks is then responsible for the application-oriented management of its own parallel program. The host tasks will create subtasks assigned to run on the parallel machine. They will distribute input parameters to and gather output parameters from the subtasks. This is also referred to as "*host/node programming.*" In that case, the parallel application itself is faced with the heterogeneity problem.

Considering a parallel application as an ensemble of communicating (active) objects and exploiting appropriate programming paradigms make heterogeneity an issue to be dealt with entirely by system software and stub generators. Reality, however, is very discouraging for the application programmer. The majority of parallel applications, especially for the numerical sector, were and still are written in Fortran. In this case, an integrated system organization is feasible only if both computer components together build a homogeneous system. A state-of-the-art example of such an organization is the CM-5 [23]. Control processors and processing nodes of the CM-5 rely on SPARC technology. The control processors run a SunOS variant, and the processing nodes execute a runtime executive supporting parallel processing. Roughly speaking, the control processor is nothing but a dedicated SPARC workstation used to supervise program execution on the processing nodes.

The integrated solution in attaching the parallel machine to the "outside world" has many advantages. Probably the most important aspect is the improved performance behavior of the system. An integrated solution does not only address software issues, such as the programming paradigm used to model parallel applications; but also hardware aspects, especially concerning the interconnection structure. In the integrated case, all hardware system components (i.e., the nodes) are interconnected by means of a *homogeneous network*. This makes specifically designed gateway services superfluous and is the reason for improved system performance.

A homogeneous network structure does not relieve the system designer of thinking about where and how to apply representation-independent communication protocols. As mentioned above, these protocols solve the byte-order problem when faced with incompatible CPU types. From the hardware viewpoint, there is nothing against interconnecting different processors on the same network.

2.5.2.3 Representation-Independent Protocols

This brief discussion has shown that system software support, in order to overcome potential heterogeneity problems, basically means two things: representation-independent communication protocols, and the use of the RPC paradigm to access operating-system services. Only the first aspect, however, is an issue in the design and development of a high-speed communication system for the nucleus. The RPC paradigm is mainly a programming language and runtime system issue.

Nevertheless, techniques applied by an RPC layer to cope with the problem of different data representations at the communicating sites are also applicable for nucleus-level communication protocols. Information that needs to be interpreted by the communication system in order to achieve systemwide message passing is tagged with the byte-order information of the sending site. This technique is also motivated by the following fact: From the implementation point of view, the protocol packets are often handled by an individual protocol function that encodes a "remote procedure call" to the peer protocol entity. The main difference from the classical RPC approach is in the semantics of communication [60] and not the process of encoding/decoding messages.

With this background, exploiting stub generators to overcome the representation problem becomes a possible and applicable solution. Unfortunately, it is not a common approach to build stub generators with a distinguished pass that encodes/decodes data only. The marshaling/unmarshaling pass could also work for communication protocols in general. Thus, the possibility of having the pass selectively run is deemed highly desirable. In many cases, "protocol stubs" need to be generated manually.

Representation-independent nucleus-level protocols are only required if an integrated system, as discussed above, is to be supported. Only in this case will the nucleus (and, possibly, other building blocks of the parallel operating system) also have to execute on the host computer to enable the cooperation between application tasks that are distributed across the entire hardware system. The bridge to the parallel machine can be readily achieved by means of a nucleus "guest-level" implementation on top of the host operating system. This aspect is explored in the next chapter. What is of importance at this point is to understand that the nucleus need not be integrated into the kernel of the host operating system to provide coupling with the parallel machine. Nucleus integration—thus kernel changes—is required only if the host operating-system interface does not provide functions to access the network in raw mode. That is, the host operating system must allow user-level access to the network device.

2.5.3 Problem-Oriented Protocol Layers

As the previous discussion pointed out, relying on a single transport protocol system cannot be the ultimate approach. Since the message startup time and end-to-end message latency depend greatly on the demands of the parallel application and the capabilities of the network architecture, *problem-oriented protocol layers* must be used to structure the communication system. Moreover, the communication system should exhibit a structure that also supports interconnecting the parallel machine with a host computer.

In order to perform networkwide inter-process communication, three main functions must be provided: first, determine the locations of active/passive objects; second, enable data transport between (thus interconnection of) locations; and third, attach the locations to the network interface. A communication system emerges that comprises three problem-oriented protocol layers (Figure 2.9).

Interactions between the layers happen via *downcalls* and *upcalls* [18]. Queues are used where possible to decouple the different flows

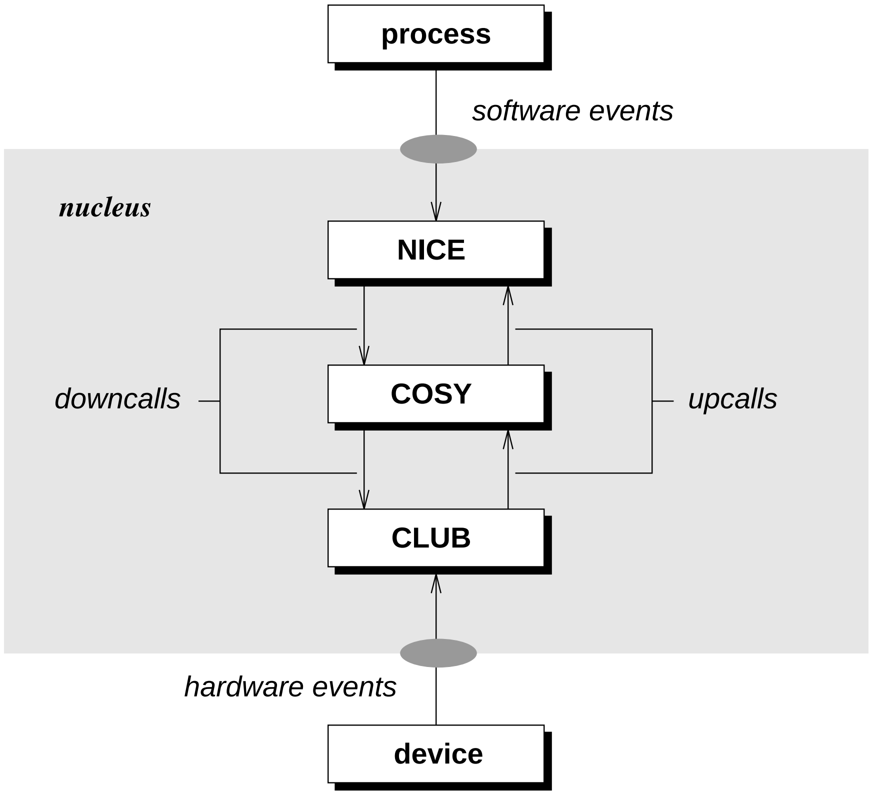

Figure 2.9: Communication system architecture

of control. Calls in either direction are virtually asynchronous, since it depends on the actual load whether message transfer requests must be queued or can be immediately carried out. The layers implement the NC^2 *protocol suite* and are discussed in the following subsections.

2.5.3.1 Inter-Nucleus Protocol

The nucleus supports networkwide communication between both processes (i.e., threads) and address spaces (i.e., teams). Inter-process communication is synchronous and *packet-oriented*, whereas the communication between entire teams is *segment-oriented* and asynchronous. In order to communicate, the location of processes (active objects) and/or team address spaces (passive objects) must be known. This is the task of the top layer of the communication system. It determines the correlation between active/passive objects and the nodes where these objects are residing.

Networkwide communication between objects is supported by NICE (*network independent communication executive*). NICE implements the *inter-nucleus protocol*. This protocol is responsible for global, networkwide process and address-space control. It activates and verifies the presence of remote processes and address spaces. Network communication is not done by NICE, but by some lower-level communication system (i.e., COSY). Instead, state transitions of processes and address spaces are controlled to logically enable end-to-end data transfers without the need for intermediate buffering. In this sense, NICE takes more the form of a management, rather than communication, layer.

The base function of the inter-nucleus protocol is to verify the presence of communication endpoints. This measure is to ensure the availability of address spaces that are either the source or destination of a data transfer. As mentioned, the basic inter-nucleus protocol primarily encapsulates resource management strategies. It does not implement security measures. Since the parallel machine is not always run in a multi-user mode of operation, security is understood as a minimal extension to a minimal subset of NICE functions. Nevertheless, the overall nucleus design is complete, and thus logically takes responsibility for multi-user security.

2.5.3.2 Transport Protocol

Data transfer is accomplished by COSY (*communication system*). This layer encapsulates *transport protocol* functions and provides an abstraction from the actual network capabilities. Depending on these capabilities, COSY is more or less complex. It provides a *protocol suite* (i.e., family), not only a single implementation, for all possible system configurations.

Logically, COSY takes responsibility for a secured data transport of arbitrarily sized messages. However, "logical" also implies a configuration in which COSY is not required for any network activities (Figure 2.10). That extreme situation arises when the network hardware itself is capable of transferring message streams in a manner required by parallel applications. In those cases, COSY simply forwards all request from and to NICE, without interpretation (right part of Figure 2.10).

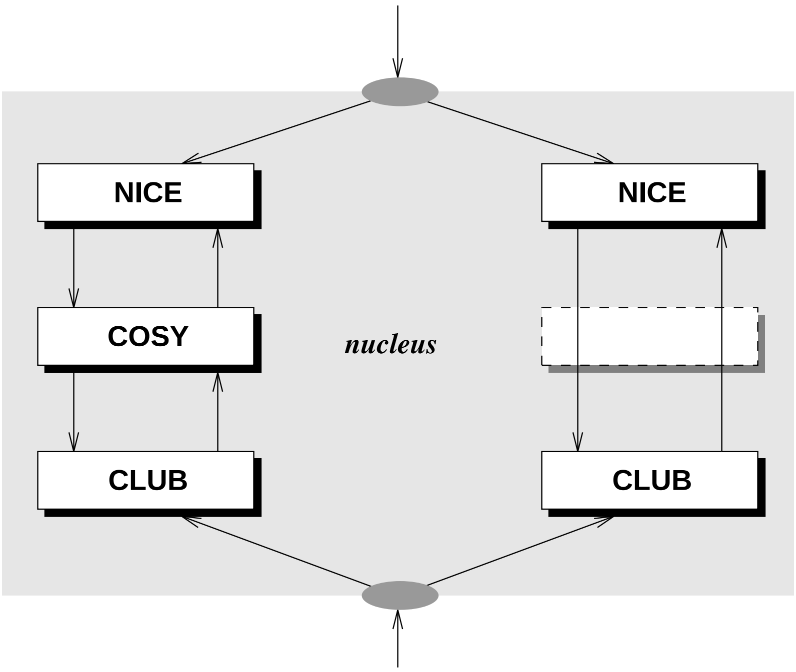

Figure 2.10: Software vs. hardware transport protocol features

COSY supports the concept of *virtual channels*. Downcalls from NICE that have network relevance and thus imply a message transfer include the logical addresses of the nodes to which these calls are directed. A logical connection between the sending and receiving nodes is established and made known by generating a virtual channel identifier. In reality, a virtual channel is nothing but a number that is used as hash key to locate protocol objects that manage a specific connection. Upcalls from CLUB deliver virtual channel identifiers as part of the COSY packet that was just received.

The protocol objects COSY implements distinguish between different data transport functions. These functions are tuned with respect to the given class of network, but without having knowledge of how this network is really accessed. Access transparency in this case is rendered possible by CLUB. Functions for segmentation and reassembly are provided when required by the network or even by the application. For instance, an Ethernet-type of network makes these functions necessary to allow the transfer of messages that are larger than the actual frame size as predefined by the hardware. Applications whose message sizes do not exceed this parameter make message segmentation and reassembly, even in Ethernet environments, unnecessary.[17]

2.5.3.3 Network Device Protocol

Abstraction from the physical network interface is done by CLUB (*cluster bus*).[18] Thus, the bottom layer of the communication system encapsulates the network device and attaches the nucleus physically to the network. This layer implements the *network device driver*.

CLUB provides the view of an abstract network device that may have several physical representations. The variety encompasses:

- *Loopback* implementations, to aid testing, debugging and porting of PEACE
- Network interface simulations on top of (commercial) operating systems such as UNIX or MS-DOS^{TM} to make PEACE a common object-oriented parallel computing platform

[17]Based on Ethernet, the maximal physical frame (i.e., packet) size is 1536 bytes. In contrast, relying on an ATM network, for example, a single cell (i.e., packet) is 48 bytes in size. These are the sizes of a single physical network data unit. It depends on the communication protocol how many bytes remain for a single user-level data unit. Obviously, ATM cells are capable of transferring at most very few words. However, this may be sufficient for fine-grain parallel applications based on a communication pattern that assumes the exchange of scalar values (typically of 32- or 64-bit size, taking into account 64-bit processor technology) and not entire vectors.

[18]This is "SUPRENUM lingo," since in the SUPRENUM architecture up to 20 nodes were interconnected through the *cluster bus*, thus building a cluster. A second-level network system was then used for interconnecting up to 16 clusters. The low-level communication protocol was known as the "cluster bus driver."

- *Shared-memory* drivers, to extend PEACE into a shared-memory multiprocessor environment

- Real network device drivers, for example, for Ethernet, as one extreme, to make PEACE a distributed operating-system framework, and for crossbar switches, as the other extreme, to make PEACE a parallel operating-system framework

The CLUB abstraction makes COSY independent from the network device actually used, no matter whether this device is a physical or a logical one. By that means, the portability of COSY protocols is supported. CLUB also provides a framework for the development of communication protocols, as it becomes possible to test and debug new implementations under the control of some host operating system instead of being compelled to use the bare hardware.

Another important aspect of CLUB is to allow coexisting COSY entities to share a single network device (Figure 2.11). A sending COSY entity provides additional demultiplex information used by the receiving CLUB instance to select the proper transport protocol for the incoming message. In principle, CLUB performs *dynamic binding* of upcalls to COSY across the network. A message object that is going to be transmitted by means of CLUB encodes the upcall handler (i.e., method) to be used for the delivery of that object to the peer COSY entity. That is, message objects are *polymorphic* and, as will be discussed later, composed through inheritance.

The *transport protocol* entity shown in Figure 2.11 provides the COSY functions discussed in the previous sections. In particular, this entity is "void" if a transport protocol can be sacrificed. A special case is the *user channel* entity, which can be selected by CLUB. The user channel entity provides user-level access to the network device (i.e., CLUB).

The user channel takes the form of a nucleus extension and is part of the kernel. Since kernel functions are provided to the user by means of ROI, the user channel entity is actually an active object. The task of this object is to forward incoming messages to *user protocol* entities and to forward outgoing messages to CLUB. In the upward direction, the user channel accepts and buffers CLUB upcalls. These upcalls are

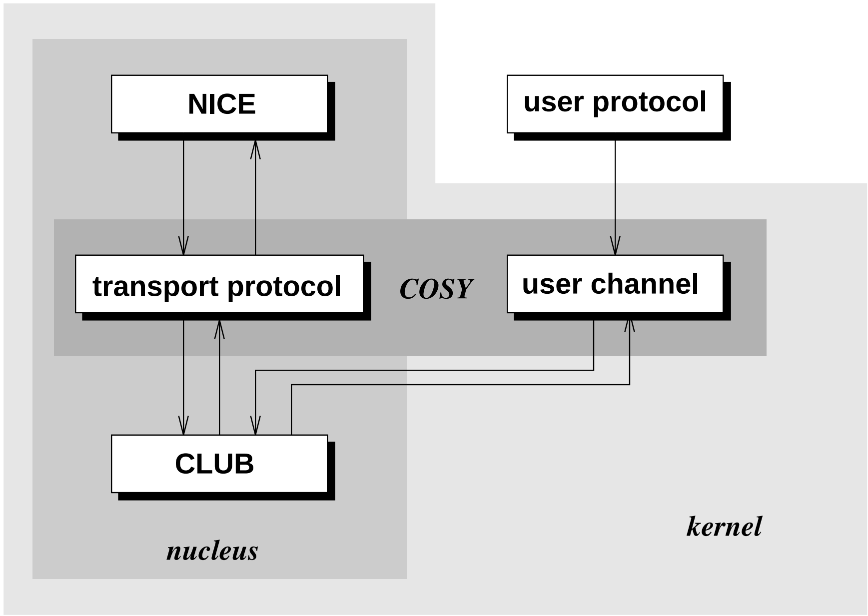

Figure 2.11: Coexisting transport protocol entities

delayed until a user protocol entity requests delivery. This may cause intermediate buffering of arbitrarily sized messages. In order to keep kernel complexity small even in this situation, the buffer space is provided by the user-level entity. That is, the user channel has direct access to some user address space, and a high-performance user-level communication system can still be achieved when applying appropriate techniques (e.g., as described in [35]). Different transport protocol classes may coexist at the same time. In this case, the user channel entity takes care of multiplexing/demultiplexing the data stream accordingly.

The user channel is mandatory for at least one reason and is a meaningful component for several more reasons.[19] If the network is the only means by which nodes can be bootstrapped, raw network device access

[19]Other application areas are, for example, testing and debugging of new communication protocols under real conditions, but without the necessity of reconfiguring the nucleus.

is demanded. The bootstrapping site cannot assume that a nucleus is initially running on the nodes. In this case, the user protocol entity actually acts as the bootstrapper and uses the user channel to transfer the nucleus (and, eventually, other system entities, but at least an application entity) to the "virgin" node. In order to provide a minimal basis that makes bootstrapping feasible, only the downward direction of the user channel need be supported. However, this assumes that the node being bootstrapped does not rely on a handshake protocol with the bootstrapping site. Messages sent back must then be distinguished from "normal" messages, which need to be forwarded to NICE. The bootstrapping site may be a multithreaded node, where even tasks other than the bootstrapper may be involved in normal communication patterns to still-bootstrapped nodes. For this purpose, CLUB, at the bootstrapping site, must execute the demultiplex function, whereas all other running nodes will have to make arrangements for proper multiplexing.

Finally, it should be mentioned that the multiplexing/demultiplexing functions of CLUB are optional. These functions are not free of charge, and will therefore cause an increase in the message startup time. Runtime support for coexisting COSY entities is provided only by some, not all, members of the protocol family. The bootstrap example mentioned above demonstrates how a specific (and only initially executed) protocol will imply overhead in the normal communication process. If a handshake protocol is used during the bootstrap phase and the user channel entity only required for getting the system started, and if that entity is the only alternative anyway, then applications are forced to pay for unneeded communication functions. This would clearly violate the family idea.

2.6 Separation of Concerns

The macrostructure of the system described in Section 2.2.1 suggests a possible separation of user and supervisor mode. If the separation is made, a *microkernel architecture* emerges. More precisely, the nucleus and kernel constitute a *microkernel family*, which provides a number of problem-oriented implementations of the abstract data types used to build the minimal subset of system functions. The microkernel approach implies service encapsulation by means of system processes that are executed in user mode. These processes will then have to (logically) compete with application processes to be assigned to the processor for execution. Thus, a microkernel always assumes a multitasking mode of operation.

2.6.1 Vertical vs. Horizontal Isolation

In the multitasking case, kernel entity and user-level processes need to be protected. The kernel address space (supervisor mode) must be isolated from the user address space (user mode). Since the kernel entity and processes are hierarchically arranged with respect to each other—processes "use," and are thus hierarchically layered above, the kernel entity—the need for *vertical isolation* of address spaces arises. A distinction is made between the kernel and user address spaces. As several processes may share the same node, but without necessarily being hierarchically arranged with respect to each other, the *horizontal isolation* of address spaces is required as well. In this case, the different user address spaces must be protected from each other. If the separation between the two processing modes is sacrificed, kernel and nucleus will take the form of a runtime library. Consequently, nucleus/kernel activations always happen without crossing protection boundaries. Application, POSE, kernel and nucleus together constitute a nonsequential program that is executed in (privileged) supervisor mode.

Since it makes a difference in the implementation and system performance, a distinction between these two isolation measures is made. Both cases have in common that an MMU is employed to enforce address space integrity. They vary in the way the MMU is programmed and used to protect address spaces, and thus exhibit different performance characteristics.

Vertical isolation means denying a user-mode entity access to the kernel address space, but granting the (supervisor-mode) kernel entity access to the user-mode address space. Thus, while a user-mode thread is executing in supervisor mode, it may "directly" access its user-mode address space. Whether or not this really happens without additional (kernel) software intervention, however, depends on the

MMU capabilities.[20] At best, vertical isolation implies runtime overhead that is dominated by executing a cross-domain call to enter the nucleus (i.e., kernel space). The need to access a user address space arises with either passing arguments to the system (i.e., nucleus) or performing message transfers out of or into the user address space.

In contrast, horizontal isolation means prohibiting user-mode entities from accessing each other's address space. The kernel is granted complete access rights. However, in many cases, the kernel is required to dynamically map in an address space when a user-mode thread, while executing in kernel mode, requests access to a segment of a different task.[21] Again, what needs to be done to virtually interconnect address spaces with each other depends on the MMU capabilities. A typical scenario is to first validate the logical address that denotes the address space segment which needs to be mapped in; second, set up the kernel page/segment table to enable access to the address space segment (i.e., actually map in); and third, flush the address translation buffer cache of the MMU to make sure that out-of-date mapping information is invalidated. Given a logical address and a process identifier, selecting the per-process page/segment table can be extremely heavyweight. These are all mostly software activities that, for example, must be executed to locally copy a message between user address spaces.

Overhead gets even worse as flushing of the address translation buffer cache entails expensive memory accesses to load page/segment descriptors upon future cache misses. Note that MMU cache flushing is not necessary only to finish the mapping of address spaces. It also has to be performed along with task switches. A task has its own address space. Switching between threads that belong to different tasks also

[20] For example, in the SUPRENUM implementation, the Motorola PMMU had to be explicitly directed to fetch a data word from user space, then move the data word into some CPU register and afterwards save the register contents into the destination location in kernel space. Transferring data from kernel into user space works in the opposite order. User mode access can only be expressed by special assembly-language-level instructions, which requires a subroutine representation to achieve the linkage with the C/C++ language level. This makes a simple move instruction an overhead-prone subroutine.

[21] This does not necessarily mean to page in the address space, supporting virtual memory. Rather, the MMU must be programmed to enable the kernel to access a specific user-level (in-core) memory segment.

requires switching address spaces. Provided that the runtime stack of a thread is not protected by the MMU, the address translation buffer cache must not be flushed upon thread switches as long as the other thread continues execution in the same address space.

Protecting the runtime stack of a thread is sometimes the absolute minimum that must be achieved. However, stack isolation may be as heavyweight as address space isolation if the stack segments are mapped to the same region of the team address space. In this case, thread switching implies stack segment switching by instructing the MMU to invalidate the stack segment descriptor of the unscheduled thread and validate the stack segment descriptor of the scheduled thread. Thus, processes of a team are lightweight only if the stack segments are nonoverlapping, that is, mapped to disjoint (logical/physical) team address space regions. This mapping strategy does not enforce MMU setup procedures when switching between threads of the same task (i.e., team) takes place. Hence, in order to implement lightweight processes, either must stack protection (and, most important, control over when a thread runs out of stack space) be sacrificed or must stack segments be scattered through the logical team address space. In both cases, thread switching within the same task (i.e., team) becomes lightweight.

2.6.2 Normal vs. Exceptional Processing

In the context of hierarchically structured abstract machines, the distinction between intra- and inter-domain calls in order to invoke the nucleus is a question of whether normal or exceptional processing takes place. Normal processing means that a thread of control does not leave its execution or protection domain, whereas in the case of exceptional processing, it does.

By what method a protection domain can be left depends mostly on the capabilities of the (abstract) processor that executes the program. When considering state-of-the-art processor architectures being used in commercial systems, a *trap mechanism* serves to cross protection boundaries. In the case discussed here, the nucleus defines an abstract processor and is invoked through a *trap instruction* in order to request a change of the protection domain of the currently executing thread. In very few cases, physical processors are able to provide comparable services. The (abstract) processor must verify that the thread is allowed to change protection domains. More generally, the thread must possess the capability to access a specific resource (i.e., object) that is mapped into some address space. Protection domains are such objects. These domains may isolate files, devices, processes, communication channels or buffers, memory segments, or even user-defined entities. Thus, the (abstract) processor, in some sense, must define a *capability-based architecture* [28].

Cross-domain calls of these categories can be much more easily realized by an abstract processor (i.e., software) as opposed to a physical processor (i.e., hardware). The abstract processor "nucleus" (or "kernel") then must provide primitive instructions which enable access to objects that are mapped in a context different from the one which accommodates the accessing thread (i.e., the subject). If relying on a software processor, in order to gain object access, boundaries due to vertical and horizontal isolation must be crossed. Note that vertical isolation ensures the integrity of the software processor "nucleus." The user/nucleus interface is the place where inter-domain calls are settled. The more general approach that supports threads entering (almost) any protection domain is not pursued at this level, since it would significantly increase the complexity of the minimal basis. Rather, an object-bound RPC mechanism enabling inter-domain remote object invocation is used. This mechanism is implemented as a runtime system library package and layered above the nucleus (but below the kernel).

An inter-domain call leads to the *partial interpretation* of the trapped program through the nucleus. The notion of partial interpretation stems from structured computer organization [86], based on a hierarchy of physical and abstract processors. The logical structure specifies a nucleus level between the parallel program level and the *conventional machine level* (i.e., physical processor). That is, the parallel program "uses" the nucleus, which in turn "uses" the conventional machine.

The nucleus implements an abstract processor that is dedicated to the processing of (distributed) parallel programs. Primitives of this processor take care of systemwide communication between cooperating active objects. Executing these primitives means partially interpreting the parallel program. It is said to be partial because in all other cases the conventional machine is in charge of program execution.

Thus, the normal situation is that a (parallel) program is directly run by the physical processor—during execution, the parallel program level is adjacent to the conventional machine level. In order to invoke the nucleus level, an "exceptional event" must be raised that directs the conventional machine to temporarily stop processing of the parallel program. The program is trapped, and the nucleus takes charge of the further execution by interpreting the encoded nucleus primitive. Processing of the instruction stream produced by executing the nucleus primitive is done by the conventional machine. Since the conventional machine cannot distinguish between instructions that implement the nucleus and those that implement the parallel program, it must be explicitly directed to stop the nucleus and continue processing of the parallel program when the nucleus primitive has completed its operation. For this purpose, the nucleus must raise another "exceptional event" in order to tell the conventional machine that the interpretation of the parallel program through the nucleus has been finished. A return from trap is indicated, which causes the conventional machine to further process the parallel program.

Usually, all of these activities happen transparently to the application program. However, it should be clear that partial interpretation (i.e., executing an inter-domain call) implies additional overhead.[22] In the design discussed here, only the different calling sequence introduces overhead, not the nucleus primitive that is going to be called. This enables an easy switch between the two different nucleus representations, without changing the nucleus implementation (see Figure 2.2). In this sense, *virtually partial interpretation* is performed, as the intra-domain call representation is not based on trap handling to execute a nucleus primitive.

2.6.3 Static vs. Dynamic Configuration

Nucleus and kernel are technically represented in two different ways. On the one hand, all functions (i.e., object modules) are kept in a *kernel library*. The majority of the object modules contain a single func-

[22]The overhead may become prohibitively large in state-of-the-art RISC architectures. This is discussed in a later section.

tion only, even when the module (or class) provides several functions. On the other hand, both building blocks together form an executable supervisor-mode program, referred to as the *kernel entity*. In this case, a *kernel stub library* stores those stub routines that correspond to exported nucleus and kernel functions. More precisely, the kernel stub library takes care of the calling sequences necessary to invoke the nucleus and kernel, respectively.

These two schemes are the first step towards a concrete representation of program families, that is, a kernel family. The family is constructed by means of *static configuration* as supported by a standard linker utility. It is the kernel application that determines which (nucleus/kernel) function must actually be present in a specific configuration. In the single-address-space model the kernel library, and in the multi-address-space model the kernel stub library, is used in the binding process. The former case results in the assembly of a true application-oriented kernel, as only those functions will be bound that are referenced by the given (user/system) application. In the latter case, what the actual shape of the kernel entity will be primarily depends on VICE (see Section 2.2.4).

As VICE does not know which of these functions are likely to be called by the application, a straightforward approach is to simply reference all functions defining the nucleus and kernel interface. The outcome will be a large kernel entity, often providing more services than actually required to support higher-level entities. However, this only wastes memory space but does not lead to a loss of communication performance. A sophisticated approach is to dynamically install a kernel service function on demand, namely when being called for the first time. In that case, VICE has to implement a feature known as the "*trap-on-use*" property of Multics [64]. This kind of dynamically alterable system organization is more costly when compared to the incremental loading of POSE entities. Individual modules will have to be integrated or replaced "on-the-fly." That is, an existing and running kernel program will be subjected to changes on a module basis, and an operating system is not only dynamically restructured by "simply" adding new processes.

Kernel entity interface functions define a horizontal extensible set of service primitives that are not necessarily interlocked with each other.

There may be a well-defined "uses-relation" [67] between these primitives, but in most cases these extensions are not inter-independent. The reasons they are still part of the kernel entity are:

- Their proximity to the hardware, which requires supervisor-mode execution in most state-of-the-art processor architectures
- Their proximity to some internal data structures, which requires some sort of address-space sharing

The first aspect typically refers to device abstractions, such as I/O, CPU and MMU drivers. Driver modules often use privileged instructions. This demands supervisor-mode execution. It does not suffice to map, for example, the I/O space into the logical address space of some higher-level entity—unless the processor is capable of temporarily granting privileged mode when a process is executing within a specific memory segment.[23] The second aspect is the typical situation with thread processing, as performed by the nucleus, being extended by thread management, which is the task of the kernel. A new system component is introduced that allocates and deallocates process objects from the same data type the nucleus uses to execute threads.

Even if the MMU and CPU allow one to take advantage of address-space sharing—or even manipulation—in the above-mentioned sense, it nevertheless will not be used at this stage of system design. The reason is that these features would have to be added to the minimal subset of system functions discussed so far. This form of *dynamic reconfiguration* would make the minimal subset oversized for any parallel application.

Dynamic configuration will be introduced on the basis of minimal system extensions. That is, a memory management service is not part of the minimal subset of system functions. In particular, it is neither part of the nucleus nor the kernel. The kernel at most provides an MMU driver as minimal system extension. This would be the case, for example, to support multitasked nodes and/or virtually shared memory. Similarly, VICE features enabling dynamical restructuring of the

[23]In this case, it will be the task of the address space management system to verify that the process is allowed to run through the privileged segment. This verification will be done at segment-mapping time and not upon every segment access.

nucleus will take the form of minimal VICE extensions. Thus, the minimal subset of VICE functions in order to invoke a nucleus primitive on behalf of the requesting thread assumes a statically configured nucleus. Specializations (i.e., extensions) of this VICE-level invocation mechanism will then take care of dynamically loading a nucleus function on demand, namely, when the function is called for the first time (see also Section 2.2.4).

2.7 Summary

The functional decomposition of a parallel operating system as presented led to three major building blocks: the *nucleus*, *kernel* and POSE (*parallel operating system extension*). A *highly modular system structure* has been presented, which not only ends up in a bottleneck-free implementation but also tackles the problem of how to bootstrap hundreds or thousands (or even millions) of nodes.

Distributed-memory parallel computers are organized very differently. Some are comparable to a very tightly coupled system of mini-workstations, and others have extremely scaled-down node functionalities and merely enable the processing of a single task per node. In order to demonstrate *applicability* for both (and other) types of systems, it was made a matter of configuration and not primarily implementation in order to adapt a parallel operating system to either of these environments. Dedicated building blocks have been identified that need not reside together on every node. The presented design does not require every node to be equipped with the same set of system functions, but rather that services are made accessible from any node.

In this sense, state-of-the-art microkernels were discovered to be unsuitable for building a common platform for every node on top of which operating system and application tasks can then be processed. Current microkernels still exhibit a complexity that makes it extremely difficult to overcome both the message startup time and bootstrapping problems in (massively) parallel systems. They suffer from not being downward-scalable, and were designed only with the goal in mind of supporting upward scalability of operating-system structures. In principle, this is a very sensible goal, but it is not challenging enough to meet the very specific needs of high-performance parallel (distributed) applications.

A modern microkernel design means providing specific support for single-tasking applications too. The microkernel should be understood as a *kernel family* and not as a single solution for all possible cases. This family then introduces *operating mode variety*. It includes state-of-the-art microkernels as well as dedicated runtime libraries assigned to implement a scaled-down *process execution and communication environment* for parallel applications.

Tailoring of the operating system to given user demands and hardware constraints must be the maxim. This is reflected by the functional hierarchy defined between the three building blocks, and by having introduced the notion of a kernel family. The functional hierarchy enables a number of system configurations. Both single- and multi-address-space models are supported.

With respect to performance, one of the most critical components of operating-system software for distributed-memory machines is the communication system. In addition to the performance aspect, the design of the communication system is faced with a variety of user demands and hardware facts. The underlying hardware architecture may require fairly complex transport protocol functions, but it may also call for extremely simple protocols more or less corresponding to low-level device driver functions. Similarly, different types of parallel applications imply service parameters of different qualities to be handled. There are applications that implicitly tolerate a sporadic loss of message packets and, although the network may not meet quality constraints as dictated by "classical" distributed applications, make a transport protocol (at least partly) obsolete. Other applications may have to rely on features such as message segmentation/reassembly, flow control and sequencing to ensure safe processing and support sharing of the parallel machine by different user applications.

The design of the communication system follows the same direction that was pursued in designing the kernel and the operating system. A *protocol family* makes the communication system adaptable to the various demands. The global structure defines three *problem-oriented protocol layers*, with each layer having several representations. At the top, the *inter-nucleus protocol* is responsible for remote scheduling of processes and address spaces. In the middle, a *transport protocol* is placed. At the bottom, a local *device protocol* is driven, mainly executing the network device driver. The transport protocol is virtually present, since it is sacrificed when the network hardware already meets the needs of parallel applications. If always present, the transport protocol could potentially cause a performance decrease, as unneeded functions are executed. This would clearly violate the family idea.

Chapter 3

System Composition

The functional decomposition explained in the previous chapters led to a system structure that consists of three main building blocks. These are the *nucleus*, the *kernel* and the *parallel operating system extension* (POSE). This structure was designed to ease *configurability* and *extensibility* and provide the utmost communication *efficiency*. The macrostructure given by the three building blocks defines an operating-system family, with POSE providing the variety of system services needed. In contrast, the microstructure defined by the nucleus (and partly by the kernel) introduces a microkernel family and provides a variety of operating modes. The macrostructure stresses configurability and extensibility, while the microstructure focuses on efficiency.

This chapter presents possible system configurations, emerging from the combination of the three building blocks in a problem-oriented manner. A distinction is made between a native operating system for distributed-memory parallel computers and a parallel computing platform for (high-performance) distributed systems. It is demonstrated that the system architecture is applicable in completely different environments. The summary at the end of this chapter draws some conclusions.

3.1 Native Operating System

A *native operating system* runs on the bare hardware. It is designed to control all hardware resources without the assistance of some other operating system. An important point is how to isolate and implement the hardware-dependent system components. The term "hardware-dependent" also encompasses site-dependent; for example, a device driver has to reside on the node to which the device it controls is attached. As is discussed later, replacing carefully isolated components may not only imply the porting of the native operating system onto completely different hardware platforms, but also transforming it into a user-level runtime executive.

3.1.1 Configuration Aspects

There are two system components encapsulating site-dependent functions: the nucleus and the kernel. The nucleus is site-dependent because it acts as the minimal basis needed to interconnect cooperating active objects of a parallel/distributed application. Since the nucleus supports networkwide communication and preemptive scheduling, hardware-dependent modules are used as abstractions of a clock device and a network device. In addition, the nucleus includes hardware-dependent modules that implement processor abstractions needed to perform process switching and interrupt synchronization.

The kernel is site-dependent because it encapsulates additional device driver functions used to support specific application and/or system configurations. Examples are the MMU driver and the disk driver. The MMU driver is needed to implement multiple address spaces or virtual shared memory. The disk driver is required for file-handling purposes. In addition to these hardware-dependent features, the kernel is also responsible for implementing a dynamic process model. This calls for process management functions that are not necessarily hardware- but site-dependent. For example, process creation means allocation of a process control block on a specific node. This can be done only by kernel modules residing on that node.

POSE is classified as portable, meaning site-independent and hardware-independent. Although providing memory management and file-handling functions, POSE is hardware-independent since it is based on hardware abstractions introduced by the kernel. In both cases, POSE services depend on abstractions from low-level hardware capabilities. That is, these services are not required to be collocated with the hardware devices needed to effectively carry out the management functions such as setting up some address-space segment or reading/writing a disk block.

3.1.2 Configuration Variety

For the reasons mentioned above, POSE is arbitrarily distributable across the parallel machine. Since the presence of the kernel is only required in the context of specific applications or system structures, nodes may be equipped only with the nucleus. This leads to a number of possible node configurations, as illustrated in Figure 3.1.

The design proposed makes it possible to distinguish between three different types of nodes. There are *user nodes*, which are mainly allocated to user applications, and *system nodes*, which are exclusively used for operating-system functions. Furthermore, *generic nodes* are supported, meaning that application and operating-system entities are collocated with each other. In all possible configurations, the common subset of system functions that must reside on each node is specified by the nucleus.

In the following, the different node configurations are described. Referring to Figure 3.1, the discussion proceeds clockwise. This also goes with increased system complexity, especially in terms of kernel and/or POSE services. The symbols used in the figure correspond to those introduced by Figure 2.1. In particular, shaded boxes denote application entities, white boxes, system entities.

3.1.2.1 User Nodes

A user node is equipped with a minimal subset of system functions only. The focus is on application processing, and this means entirely sacrificing POSE. More specifically, POSE services implemented by active objects are provided elsewhere, but never on a user node. Since

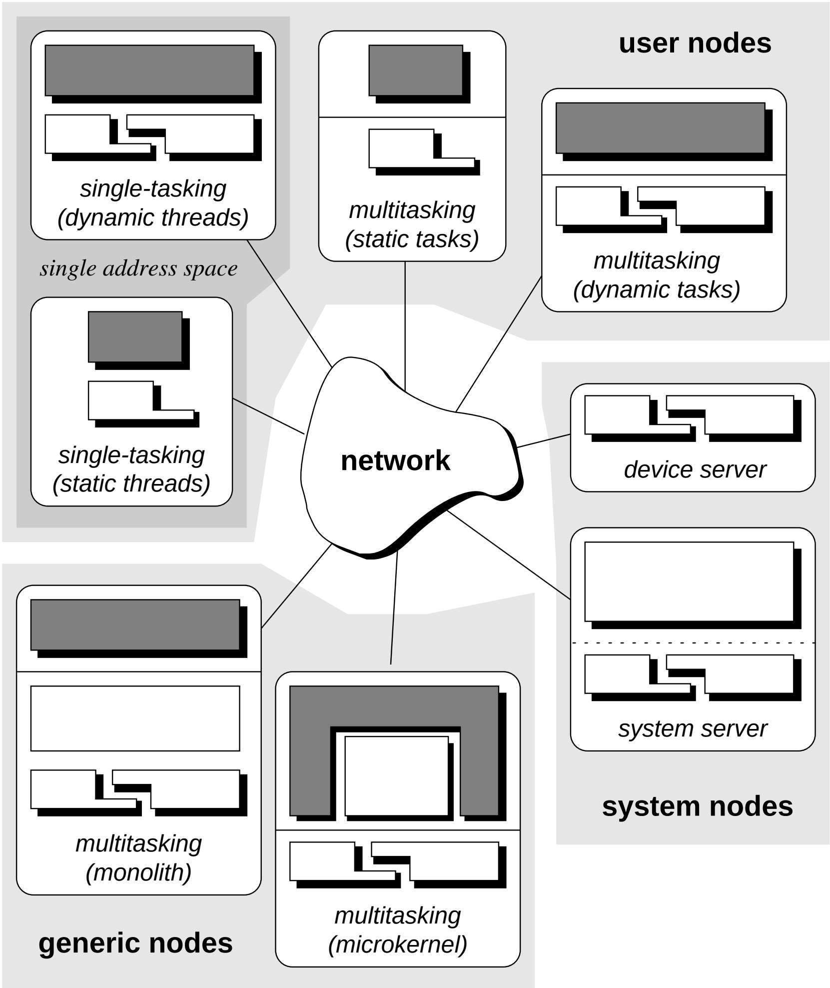

Figure 3.1: Operating-mode variety

these services are requested on an ROI basis, *access transparency* is maintained even for application entities.

With this background, only the nucleus and kernel are provided as local system support. However, the kernel is required only under any of the following circumstances:

- The node has devices attached that must be made globally accessible

- The node is managed/controlled from a remote node
- The application calls for *scaling transparency*

In the first case, the device driver simply exhibits an ROI interface. The second case works similarly and, for instance, enables remotely residing system components to initiate a "warm startup," switch on/off debugging, stop/continue the execution of active objects and freeze/revive address spaces. The third case assumes support for dynamic process management to establish a many-to-one correspondence between application tasks and nodes. Table 3.1 gives a brief overview of the properties of user nodes.

Applications that scale well with the actual number of available nodes call only for a single-tasking mode of operation. In this mode, a further distinction is made between configurations supporting *static threads* and *dynamic threads*, respectively. In the static case, multiple threads of control are established as a "side effect" of bootstrapping the node. These threads then exist as long as the node is executing the single application task. Consequently, only the nucleus is required for application processing. In the dynamic case, multiple threads of control are dynamically created and destroyed. This feature is provided by the kernel. The nucleus remains unchanged, and the kernel adds minimal nucleus extensions used for thread management.

The main characteristic of the static and dynamic thread model, however, is the *single address space* on a node. Hence, nucleus and kernel take the form of a runtime library that is directly linked to the application task. All local system functions are invoked on an LOI basis. The kernel additionally offers an ROI interface to remotely residing system components. This single-address-space approach reduces the message startup time to an absolute minimum, since protection domains are not implemented and therefore need not be bridged in the course of processing communication primitives. The single address space is thus the most important prerequisite for making lightweight processes featherweight.

Scaling transparency calls for a multitasking mode of operation. This implies *multiple address spaces* per node. A further refinement

Variant	*Property*
static threads	Single-tasking (multithreading) mode of operation. A single address space is supported. Threads are set up at application bootstrapping time and exist as long as the node is in charge of application processing. The node is solely equipped with the nucleus.
dynamic threads	Single-tasking (multithreading) mode of operation. A single address space is supported. Threads are dynamically created and destroyed by means of kernel extensions to the nucleus.
static tasks	Multitasking (multithreading) mode of operation. Multiple address spaces are supported. Tasks and threads are set up at application bootstrap time and exist as long as the node is in charge of application processing. The node is equipped solely with the nucleus. The nucleus executes in supervisor mode and application tasks execute in user mode.
dynamic tasks	Multitasking (multithreading) mode of operation. Multiple address spaces are supported. Tasks and threads are dynamically created and destroyed by means of kernel extensions to the nucleus. Nucleus and kernel execute in supervisor mode and application tasks execute in user mode.

Table 3.1: Properties of user nodes

leads to the distinction between *static tasks* and *dynamic tasks*. This is a similar distinction as the one made in the case of single address spaces. A library implementation of nucleus and kernel, as in the single-address-space variant, is no longer feasible. Consequently, the message startup time is increased, since a protection domain must be crossed to invoke the nucleus.

The distinction between the single- and multi-address-space variant of a user node is not made by either sacrificing or demanding the kernel entity. In both cases, the main purpose of the kernel is to give the node a dynamic process model. The single-address-space variant may exploit dynamic threads, whereas the multi-address-space variant may exploit dynamic tasks (with each task possibly being multithreaded). Introducing a dynamic process model can be readily achieved without any physical address space isolation. The distinction between these two address space models is due to the kernel family (see also Section 2.4). In contrast to multiple address spaces, in the single-address-space variant a different implementation of the same nucleus/kernel interface is employed. Each of these implementations represents a unique kernel family member.

3.1.2.2 System Nodes

A system node is used as server and provides globally accessible operating-system services. These nodes provide services that either have been off-loaded from user nodes or are required to control devices. That is, system nodes may host site-dependent as well as site-independent services. Table 3.2 gives a brief overview of the properties of system nodes.

Variant	*Property*
device server	The node exhibits kernel-level device driver functions enabling remote device access on an ROI basis. A single address space is supported.
system server	The node is extended by POSE services provided on an ROI basis. Depending on POSE, a separation between user and supervisor mode may be made. Logically, multiple address spaces are supported and nucleus and kernel execute in supervisor mode and POSE executes in user mode.

Table 3.2: Properties of system nodes

One scenario in establishing a system node is to have only the nucleus and kernel running, with the kernel encapsulating one or more node-specific device drivers. In this case, a *device server* node is configured. This node provides remote device access on an ROI basis. The nucleus is used to enable a (user/system) process to cooperate with the device driver (an active object) residing on that node. A typical example is a disk node. The main purpose of the kernel-level disk driver is to implement disk block caching on that node and provide cache read and write operations via ROI. In this case, file system functions are provided elsewhere, and may even be available on a library basis. Since ROI is used to interact with the device, applications are unaware of the fact that disk I/O is not really being performed by the user node. This is also true with file management functions taking the form of a user-level library, thus hiding the disk driver (i.e., cache) interface.

The disk driver example explained above may cause problems if several application tasks are going to access the same disk partition (i.e., file). In order to prevent data inconsistencies, a file cache consistency protocol must be executed. This can be readily achieved on a library basis as well. Another, more straightforward, approach, however, might be to add a file manager to the device server node. This introduces a *system server* node, which not only makes devices globally accessible but is also assigned to provide general (site-independent) operating-system services.

File management is only one example of establishing a system server node. Naming, process management, address space management, paging (e.g., in a virtual-shared-memory context), trap handling, incremental loading, load balancing and networking are further examples—the list could be continued. That is, almost any POSE service is a candidate for being off-loaded from user nodes, and thus for establishing a system server node. All these services are provided on an ROI basis; therefore, access to them may cross node boundaries, since ROI maintains access transparency.

The device server configuration also implies a single-address-space model. Only the kernel entity (i.e., nucleus and kernel) is executed on the node. In the system server case, address space isolation may be required. However, whether this is really necessary depends on the number and type of POSE services executed by the node. Separation between user and supervisor modes could be an issue, just as address space protection is. Logically, a system server node implies a multi-address-space model.

For a system server configuration, a single-address-space model may be applicable. For example, providing access to some front-end (host) computer with TCP/IP can be achieved by having the network software execute as a single POSE application of the kernel entity, with the latter being directly represented as a library for the POSE entity [63]. In this scenario, the kernel may provide access to Ethernet, thus encapsulating the network device driver. The POSE networking entity may additionally provide a socket interface, with ROI applied to socket objects in order to make host operating-system services accessible even for tasks residing on user nodes.

3.1.2.3 Generic Nodes

A generic node executes user as well as system tasks. That is, in addition to the nucleus and kernel, generic nodes also include the POSE and application building blocks. Table 3.3 gives a brief overview of the properties of generic nodes.

With POSE being executed in user mode, a configuration comparable to a typical *microkernel* organization emerges. In this case, the nucleus and kernel provide a (supervisor-mode) platform on top of which microkernel-based user and system applications can be processed. For obvious reasons, vertical and horizontal isolation measures are required (see also Section 2.6.1). The kernel entity needs to be isolated from higher-level (user and POSE) tasks, and the tasks need to be isolated from each other. A distinction between the user and supervisor modes of execution is demanded.

The second configuration of a generic node is to have only application tasks executing in user mode. The POSE, kernel and nucleus are then executed in supervisor mode. In addition, operating-system services are provided by passive objects. This leads to a traditional *monolith*, that is, a procedure-based operating system in which all service functions are mapped into the same (supervisor-mode) address space. Note that both the ROI paradigm used to interact with POSE and/or the kernel and the NOI paradigm used to interact with the nucleus are

Variant	*Property*
microkernel	The node is shared by application and system tasks. System tasks implement POSE entities. Multiple address spaces are supported. A separation is made between user and supervisor mode, with nucleus and kernel executing in supervisor mode and applications and POSE executing in user mode.
monolith	The node is shared by application and operating system. POSE entities and the kernel are represented by passive objects. Multiple address spaces are supported. A separation is made between user and supervisor mode, with nucleus, kernel, and POSE executing in supervisor mode and applications executing in user mode.

Table 3.3: Properties of generic nodes

of a procedure-based nature anyway. Creating a microkernel-based or monolithic system is mainly a question of how these calling principles are carried out by an actual system representation.

The duality of process-oriented and procedure-oriented operating-system structures [45] makes it possible to present a design that can be implemented either way. However, having a process-oriented implementation (by means of active objects) in mind is a good way to prevent the resulting system structure from developing performance bottlenecks. It is important to clearly identify the system components possibly associated with an autonomous thread of control, protected by a separate address space and executed on a different node, and therefore accessed via ROI. On this basis, it becomes a matter of configuration to generate an actual system representation that really provides active objects as service-providing entities. The other way around, namely, assuming that all components will share the same address space, may end up in a highly inefficient implementation if decentralization must be achieved in a later development phase. This may hold even though, from the software technology point of view, either representation is possible.

3.1.3 Configuration Interplay

All configurations discussed above may coexist. The nucleus is used as the common communication platform. Since POSE and kernel services are provided by means of ROI, user nodes may request services from either system nodes or generic nodes. Conversely, system nodes can be used to manage the parallel machine as a *processor bank*, allocating user nodes for the processing of parallel applications.

It actually depends on the application whether user nodes can be run as a combination of the configurations shown in Figure 3.1 and explained above. For example, a load balancer (residing on a system server node) may decide to allocate as many tasks as possible in a one-to-one correspondence with nodes. If the number of tasks exceeds the number of available nodes, some but not all of the nodes are driven in a multitasking mode of operation. This assumes that the number of tasks is already known at load time. A configuration description (in the sense of a batch job specification) can be used to instruct the load balancer accordingly. In this way, parallel applications are mapped onto a multinode machine with the goal of keeping the message startup time for the majority of tasks as small as possible. Note that a one-to-one correspondence between tasks and nodes also implies a single-address-space model, thus resulting in a low message startup time.

3.2 Parallel Computing Platform

In the previous section a monolith was characterized by the fact that all three building blocks (nucleus, kernel, and POSE) share the same address space and are executed in supervisor mode. The alternative view, however, may also be sensible, namely, to let the building blocks execute in user mode. In this configuration, a so-called *guest-level* implementation of the native operating system on top of some other (host) operating system emerges (Figure 3.2).

In fact, if the designer decides to construct a highly portable and configurable operating system, a guest-level implementation is feasible in almost every case. It is mainly a question of how to isolate hardware-dependent system components—and this, all in all, is the key question

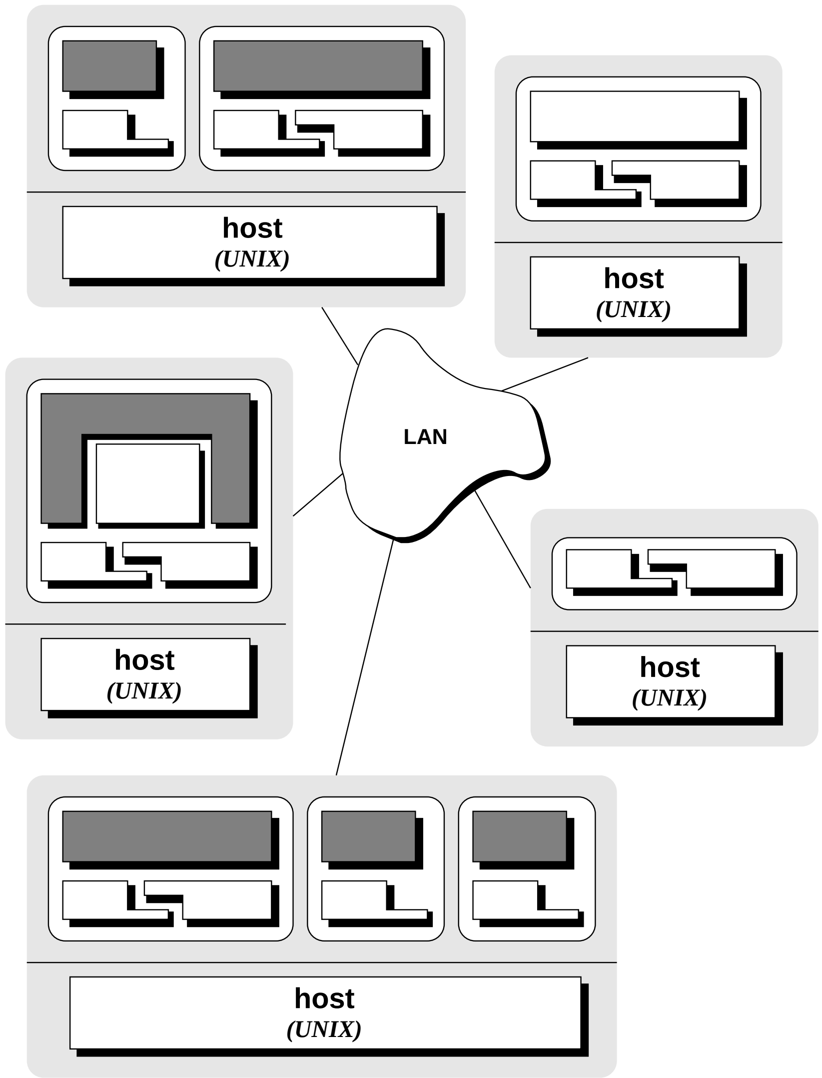

Figure 3.2: A platform for workstation clustering

in achieving portability.

3.2.1 User-Level Runtime Executive

In the case of PEACE, the guest-level implementation provides a *run-time system executive* for the processing of (object-oriented) parallel applications across a network of UNIX workstations.[1]. In order to switch between the runtime system and the native operating system, the following hardware abstractions of the minimal basis need to be replaced:

- The CPU *register set*, which is used to perform lightweight context switching.
- The CPU programming model for *trap/interrupt handling*, which is used to catch exceptional events and to perform low-level synchronization.
- The *timer device*, which is used to enable preemptive thread scheduling and to implement a watchdog service.
- The *network device*, which is used to support systemwide communication.

In the case of the UNIX guest-level version, only the first abstraction listed is really processor-dependent. The C-library functions `setjmp()` and `longjmp()` can be used to implement portable thread-switching code. However, upon thread creation, the initial runtime context (i.e., CPU register set) must be set up. This requires knowledge about the CPU-dependent layout of the jump buffer used by `setjmp()/longjmp()`. The three remaining (above-listed) abstractions are implemented using typical UNIX features such as signal handling, alarm clock and TCP/IP or UDP. That is, a guest-level implementation is nothing but a PEACE port that caters to the capabilities of a certain abstract, rather than physical, processor.

Hardware-dependent functions other than those mentioned above (e.g., I/O and memory management) may be provided in a similar way. Hardware abstractions maintained by the (PEACE) kernel must

[1] Presently, there is also a guest-level implementation available for *PARIX*, a runtime executive for Transputer-based parallel computers. A port onto WINDOWS-NT is in progress.

be replaced accordingly. Even these functions can be simulated on top of a (UNIX) host operating system, as demonstrated by the virtual-shared-memory subsystem executing under control of the PEACE guest level [19].

3.2.2 Host Interconnection

There is another aspect that makes a guest-level implementation of a parallel operating system sensible and, in some sense, even mandatory. This is the case when a parallel machine is attached to some host computer, on top of which quite a different operating system is probably being executed. In this situation, software packages need to be available not only for interconnecting the two different hardware components but also for coping with the heterogeneity issues raised by two different operating systems. Section 2.5.2 already discussed this aspect in terms of establishing the requirements that the communication system of the nucleus must meet. This discussion is revisited by presenting different patterns of how to integrate a parallel machine into an existing system environment.

Three approaches for host interconnection can be distinguished. These approaches differ in the way a parallel application is mapped onto the entire hardware complex. They also exhibit different techniques for physically interconnecting both host and parallel computer. The following subsections discuss this aspect.

3.2.2.1 Asymmetrical Coupling

A straightforward way to achieve host coupling is to provide *utility programs* that help users of the host to utilize the parallel machine for their own purposes. These programs run under control of the host operating system. They are started by the host command interpreter (e.g. the UNIX *shell*).

The parallel operating system must then provide a communication service that relies on protocols dictated by the host operating system and service functions implementing the bridge to the host-resident utility programs. For example, in the UNIX case TCP/IP must be run by some node of the parallel machine. Typically, the coupling is implemented by a *gateway node*. This node may also be assigned to execute software that enables interaction with the utilities controlling the parallel machine. In that scenario, the host computer acts as a *front-end* only. Application tasks are only executed by the multinode machine. Figure 3.3 illustrates the resulting configuration.

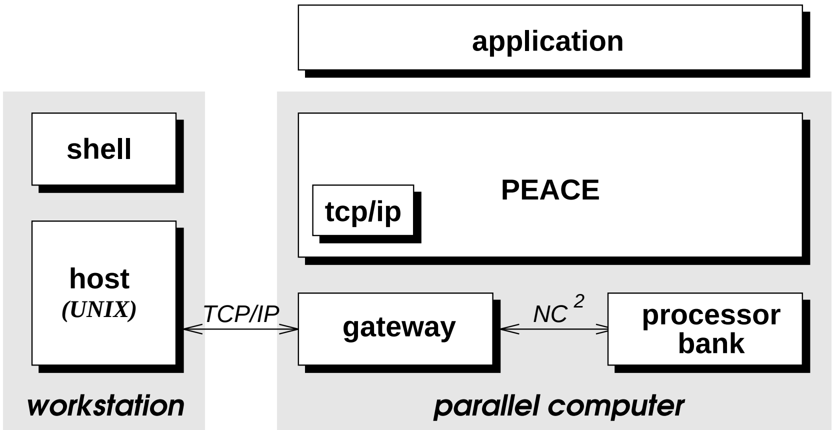

Figure 3.3: Asymmetrical host coupling

Although straightforward and undoubtedly applicable, the pure front-end approach has its limits for many (distributed-memory) parallel applications. Many of these applications follow the pattern of "*host/node programming*" and distinguish between a *host task* and a *node task*. There will be only one host task, but a number of node tasks. The host task typically creates the node tasks, distributes initial parameters and awaits final results jointly computed by these subtasks. Node tasks actually represent the computational aspect of the parallel application, whereas the host task corresponds to the management aspect.

The host task often exploits a number of services provided by the host operating system and, for this purpose, runs on the host computer. Host/node programming assumes that host and node tasks can easily and efficiently communicate with each other. In fact, the parallel program must be distributed not only over the nodes of the parallel machine but also over the host.[2] Because of the potential heterogeneity problem and I/O bottleneck, the asymmetrical coupling approach does not provide the appropriate support for these types of applications.

3.2.2.2 Logically Symmetric Coupling

In order to overcome the limits of an asymmetrical coupling between the host and parallel computers, the host task should be supported by the same software platform used to process the node tasks. This can be achieved by either making changes in the host operating system, that is, adding new kernel functions that mirror those provided on the nodes, or letting the host operating-system kernel remain unchanged and providing a user-level software package that simulates the nodes of a parallel machine.

Following an approach that relies on changes to be made in the host operating-system kernel to achieve the interconnection limits the applicability and availability of the parallel machine. In addition to that, even with such kernel modifications an efficient coupling is not guaranteed. The more promising approach is to entirely rely on user-level software packages.

This is where the guest-level implementation of a parallel operating system demonstrates its full power. It is not only a means to make parallel computing feasible while relying on local-area-network technology. Rather, this implementation defines an *abstract parallel machine*, hiding different hardware subsystems from the application (Figure 3.4).

The tasks of the parallel application all see the same abstractions. The host task is able to cooperate with the node tasks on an ROI basis. In particular, the host task can directly exploit the services provided by the parallel operating system. The parallel operating system can take advantage of the services provided by the host operating system. The host-resident part of the guest-level implementation takes care of mapping the requests dedicated to the parallel machine, and vice versa.

Nevertheless, the coupling between the host and parallel machines is only logically symmetric. This is because the interconnection is still

[2] See also Section 2.5.2 concerning the heterogeneity problems raised when the processor of the host computer and those of the parallel machine have different understandings of how data is represented in memory cells.

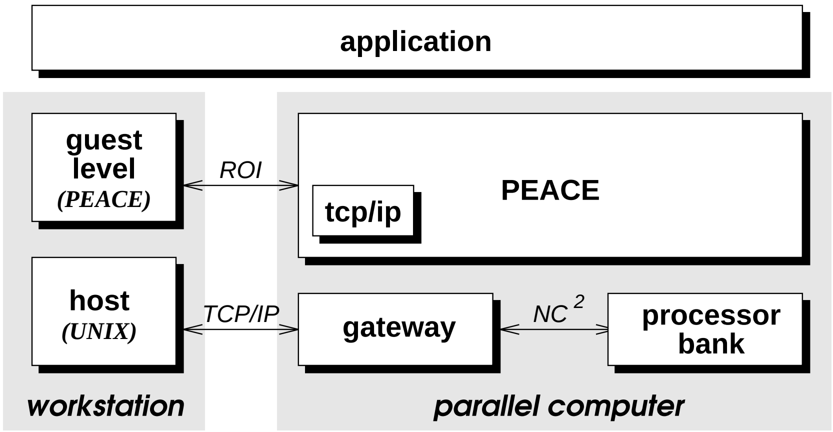

Figure 3.4: Logically symmetric host coupling

based on two different networking schemes. The host is reachable via a standard connection such as Ethernet or FDDI and TCP/IP. The nodes of the parallel machine are networked by means of a problem-oriented communication system (e.g., the NC^2 suite). As a consequence of these quite different networking techniques, both gateway node and services are still indispensable. In the example shown in Figure 3.4, the gateway function is to translate TCP/IP messages into NC^2 messages, and vice versa.

3.2.2.3 Physically Symmetric Coupling

Although application tasks can be equally distributed when relying on the symmetrical host coupling approach, the optimal coupling of host and parallel machines has not yet been achieved. It is obvious that the gateway node of the parallel machine may easily become a serious bottleneck if heavy traffic is encountered. For example, this is the case when only the host provides output devices adequate to display the results of a parallel computation. A typical device is the graphics display. Such a display is usually shared by a number of different applications and not necessarily used by parallel applications only. Its proximity to the host is thus quite reasonable.

Overcoming the potential I/O bottleneck defined by the gateway node, however, has far-reaching consequences. First, additional hardware support is required, namely to directly attach the host computer to the network that interconnects the nodes of the parallel machine. Second, standard communication protocols as used by the host operating system for LAN environments must be sacrificed (Figure 3.5).

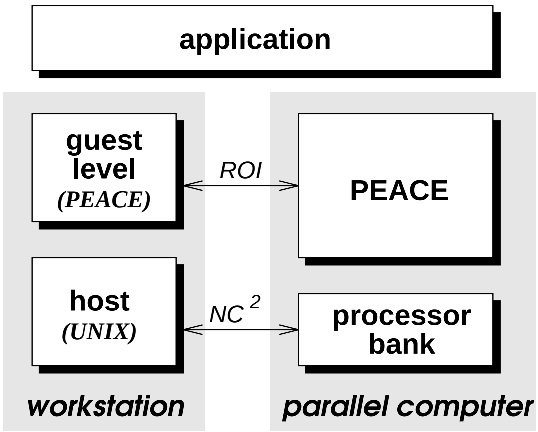

Figure 3.5: Physically symmetric host coupling

This approach leads to a fully *integrated system*. It does not require re-implementing (or even porting) of parts of the parallel operating system, except for the COSY protocols. The guest-level implementation may still be a user-level software package. However, instead of accessing TCP/IP sockets, raw network device access is required. The guest level has then to be extended with the same COSY implementation that is actually used by the native nuclei.

In fact, from the software point of view, the difference between the logically and physically symmetric solutions is whether to sacrifice or exploit COSY. Logically symmetrical host coupling means that the host-residing COSY entity takes advantage of the TCP/IP functions provided by UNIX, for instance. These functions define a set of typical transport protocol services. In other words, the PEACE guest level can rely on a network that makes the implementation of additional transport services superfluous. As a consequence, COSY can be sacrificed and is only virtually present (see also Section 2.5.3.2 for this aspect). The purpose of CLUB, then, is to map NICE directly onto the TCP/IP socket interface.

3.2.3 Logical Nodes

The main abstraction introduced by the guest-level system is the *logical node*. That is, the process model of the host operating system is used to implement logical nodes and to support the one-to-one correspondence between tasks and these nodes. Each logical node is assumed to provide only a single address space. The POSE, nucleus and kernel thus take the form of a runtime system library that is directly linked to the application task.

A logical node can be readily implemented by means of a heavyweight process, such as a UNIX process, that possesses its own address space. The guest-level system then basically implements a *user-level threads package* for that process and enables systemwide communication. In the case of UNIX, most of the POSE services are already provided by the underlying operating system. This makes service functions appear as simple compatibility libraries, primarily coping with syntactic and not semantic mapping problems.

Consequently, the nucleus and kernel are the two building blocks that remain almost completely unchanged with respect to the original implementation of a native operating-system platform. From the POSE building block, the naming subsystem is the only notable part that remains almost entirely intact. This subsystem specifically supports the ROI layer in locating the sites with which objects are associated.

3.2.4 Applicability

Designing a (native) operating system that can actually be run as a library package is extremely beneficial for the development phase and far more than a "toy system." Testing of operating-system software becomes possible without having to use bare hardware for execution. Tools provided by the host operating system can be exploited without the need to first implement debugging support or rely on some low-level monitor.

With high-speed local-area-network technology now available, parallel computing on distributed systems is gaining importance. In many parallel applications, the performance provided by these "distributed-memory parallel machines" is sufficient, or can be made sufficient by adopting a coarser grain of parallelism. It is important for the users to be able to rely on an environment that is also available on the parallel machine. Again, testing and debugging may be the motivations for having the same parallel computing platform running on some distributed system.

Beyond the somewhat parasitic approach of making parallel applications run by taking advantage of host operating-system capabilities, this approach does not only extend the applicability of (distributed-memory) parallel applications but also provides almost unlimited scalability with respect to performance. A distributed-memory machine is used only as an accelerator and need no longer be considered as just the mandatory hardware support provided for program execution. This is probably the most significant feature that a parallel operating system has to provide: an architecture and a hardware-independent platform for the processing of parallel applications.

3.3 Latency-Hiding Architecture

The rationale for composing members of the operating-system family from the three main building blocks (nucleus, kernel, and POSE) was to end up with a framework that spans a wide spectrum of possible system configurations. These global configurations, which basically determine the operating mode of the parallel machine, were discussed in the previous two sections.

By considering the internal organization of the nucleus, this section deals with another configuration aspect. As described in Section 2.5.3, the nucleus consists of three abstract protocol processors, NICE, COSY, and CLUB. These hierarchically arranged components together form the NC^2 (i.e., NICE-COSY-CLUB) protocol suite. The rationale for this structuring was not only portability; the main aspect was to develop a nucleus structure that supports specifically designed hardware architectures in the goal of NC^2 of reducing the message startup time to an absolute minimum and hiding the latency of networkwide message-passing operations.

Two state-of-the-art examples of such hardware architectures are the Intel Paragon and the MANNA [33]. The outstanding feature of both parallel machines is that they implement a so-called *dual-processor node* architecture. The following subsection briefly describes the MANNA approach. Afterwards, three different node configurations are presented. These configurations become possible due to a problem-oriented mapping of the kernel over the two processors of a node. Each configuration provides the user with a different level of abstraction and with different performance characteristics concerning the message startup time and latency.

3.3.1 Dual-Processor Node Architecture

As the name suggests, a dual-processor node consists of two software-programmable processors. The basic idea behind this architecture is to have one processor in charge of application program processing and utilize the second processor for global communication (i.e., inter-node message passing). In this setting, the former plays the role of an *application processor* (AP) and the latter plays the role of a *communication processor* (CP). Figure 3.6 sketches the architecture of the MANNA node.

Basically, the MANNA node defines a small-scale shared-memory multiprocessor system. The per-node memory system was designed to allow for a wait-state-free operation of the two 50-MIPS RISC processors (i860XP) used as AP and CP, respectively. The effective memory access rate is 400 MB/s. In addition to the memory, both processors share the I/O units used for attaching peripherals to the node; they also share the two communication links. The links connect the node to a byte-wide 16×16 crossbar switch. Sending and receiving messages through the links is accomplished by reading/writing data from/into FIFO registers, respectively. The total (physical) crossbar throughput is 2×50 MB/s, with an effective byte transfer latency of four clock ticks (i.e., 80 ns).

A crossbar hierarchy is used to interconnect more than 16 nodes. For this purpose, some crossbar links serve to establish the hierarchy, that

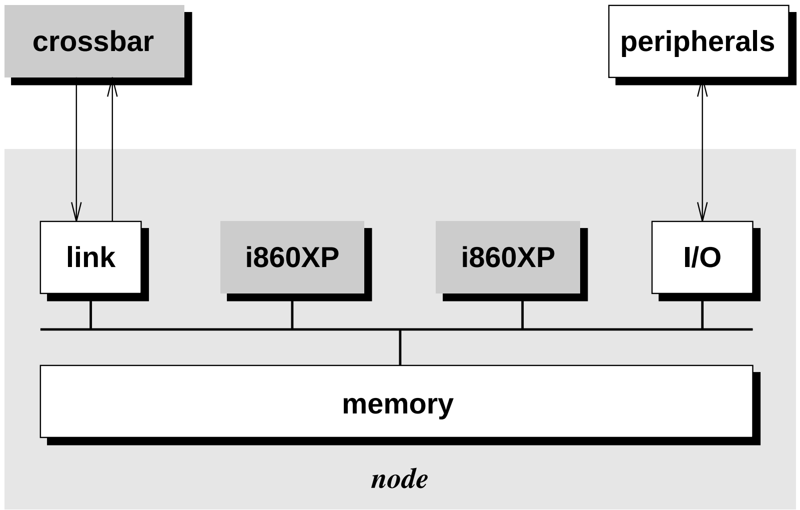

Figure 3.6: Example of a dual-processor node architecture

is, connecting a crossbar to other crossbars. The MANNA interconnection structure may also include crossbar switches that are used only for crossbar interconnection. Crossbars to which nodes are attached form a *cluster*. At the hardware level, inter-node communication within a cluster is faster than between clusters. The performance difference is in nanoseconds and will "die out" with more and more user/system software coming into play.

The idea of the CP is to relieve the AP of all functions necessary for driving the networkwide communication protocol. In particular, the CP is able to receive messages from the network and deliver the messages to the processes executed by the AP without interrupting the AP (and thus ongoing computations on that processor). In case of outgoing messages, the AP performs only a few setup operations and then hands over the message to the CP. Message setup means preparing a message descriptor and placing the descriptor on the request queue maintained by the CP.

This specific node organization and the presence of a software-programmable CP that takes over typical parts of a message-passing kernel supports several AP operating principles. Latency hiding can be efficiently implemented by guaranteeing a minimal message startup time. The AP merely sets up a message descriptor but no longer executes the communication protocol to transfer the message. Furthermore, message latency can be reduced since, once started, the CP proceeds in parallel to the AP. That is, the time spent by the AP to return to the sender thread can be utilized by the CP to perform message passing. Thus, not only user-level computation but also nucleus-level processing can overlap communication.

Before the three main node configurations are discussed, it is important to understand that the distinction between AP and CP is a software measure and not fixed by hardware. From the software point of view, the MANNA node architecture is largely symmetric. Both processors are able to directly access the network hardware interface. In particular, which processor will be responsible for handling interrupts is software-programmable. These outstanding node features define the matrix for exploring various solutions to the message startup time problem. In the following subsections, three software experiments of this sort are described.

3.3.2 Coprocessing

A more or less straightforward nucleus configuration is to view the CP as a *message-passing coprocessor*. The task of the CP is carrying out transport protocol functions on behalf of the AP and driving the network hardware interface. In particular, the nucleus implements an abstract machine that hides the dual-processor node architecture from the application tasks. Applications are concerned with only a single CPU, namely, the AP. This approach is shown in Figure 3.7.

As indicated in the figure, NICE is executed by both processors. In contrast, COSY and CLUB are executed only by one processor. In other words, the NICE-COSY-CLUB (i.e., NC^2) processor becomes the CP, and the processor executing both NICE and the application becomes the AP.

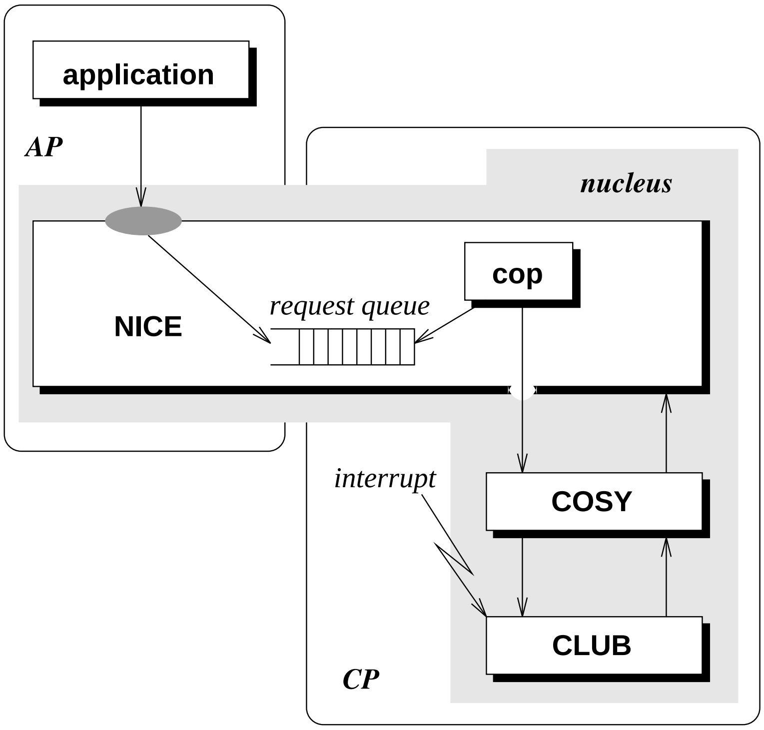

Figure 3.7: Message-passing coprocessor

3.3.2.1 Application Processor Tasks

The NICE portion executed by the AP is responsible for setting up and queuing message transfer requests. Under control of the application thread that requests a networkwide message transfer, NICE initializes a *protocol data unit* (PDU) and prepares a message descriptor. The descriptor is placed on a request queue.

The AP-to-CP connection is made up of a queue, storing packet and segment transfer requests and implementing two fundamental queuing strategies. The first strategy performs only enqueuing, providing both *first-in-first-out* (FIFO) and *last-in-first-out* (LIFO) semantics. In order to be aware of a non-empty request queue, this strategy implies that the CP polls the queue from time to time. The second strategy is a specialization of the first strategy. It sends an interrupt if the first element has been queued and the CP is busy executing software other than polling the request queue.

Sending a processor interrupt increases the message startup time for the AP slightly, but may decrease message latency since the CP is notified about pending transfer requests. Whatever the CP is executing at the moment the request queue becomes non-empty, the interrupt is the "writ of summons" for transferring AP messages. The overhead for signaling the processor interrupt is highly hardware-dependent. In the case of MANNA (i.e., the i860XP), this takes about 1 μs. Compared to the 4-μs AP message (queuing) startup time,[3] sending the processor interrupt is a fairly costly operation when the goal is to reduce the message startup to an absolute minimum.

The 4-μs startup time can be achieved only by sacrificing address space isolation between the application and kernel. Thus, the kernel for the message-passing coprocessor configuration discussed so far takes the form of a user library (see also Section 2.4). Exploiting NOI (i.e., *traps*) to invoke the nucleus adds an overhead of about 13 μs per call. Consequently, isolating the nucleus from higher-level user/system entities by drawing a user/supervisor mode protection boundary results in a message startup time of about 17 μs. In other words, nucleus protection quadruples the startup cost of a networkwide message transaction. The increased startup time also impairs the performance of local message passing. Nucleus isolation entails kernel isolation and calls for ROI to invoke kernel services, even locally. In this case, the time for a single ROI increases from 31 μs (nonisolated nucleus) to 70 μs (isolated nucleus); however, the ROI protocol overhead in both situations is the same and amounts to 5 μs. The ROI subsystem is entirely implemented on a user level and exploits the nucleus only to drive the request-response protocol between client and server. A *send-receive-reply* pattern of communication is carried out, invoking three nucleus primitives, thus resulting in three nucleus traps.

[3] More specifically, 4 μs is the delay for a user-level thread when issuing a nonblocking message transfer request. The delay expresses the nucleus overhead produced by running straight through the entire vertical execution path established when setting up a message transfer request.

3.3.2.2 Communication Processor Tasks

The NICE portion executed by the CP is responsible for processing the request queue, delivering incoming message segments directly into the address space of the AP destination thread, and unblocking AP threads waiting on incoming messages (i.e., packets or segments). In order to be able to carry out the latter two functions, NICE_{CP} has to share common data structures with NICE_{AP}. These are the ready list, the per-thread message (sender) queue and, depending on the address space model supported, the page table.

NICE_{CP} autonomously manipulates the ready list when AP threads must be unblocked due to the reception of messages. More specifically, the CP schedules but does not dispatch AP threads. Thread dispatching involves exchanging the nonvolatile CPU register set and therefore must be performed by the processor selected for executing a particular thread. In the configuration illustrated by Figure 3.7, the processor executing the (user) threads is the AP and not the CP. Depending on the scheduling strategy, the AP may or may not be interrupted by the CP in order to propagate a thread-dispatching event. How to proceed in this situation is closely related to the question of whether or not the CP should be interrupted by the AP when the first message transfer request has been queued (see previous subsection).

Sacrificing AP interruption by the CP has the advantage of allowing the AP to be fully in charge of executing the application tasks. In this situation, from the AP point of view, the reception of messages is completely transparent. The AP idle loop, entered when no more (AP) threads are ready to run, polls the ready list to determine when to dispatch which thread. Note that the scheduling decision has already been made by the CP. In order words, the AP computing resources are completely under application control.

Running the AP without such interrupts does not necessarily cause an unpredictable message delivery latency[4] for the receive function carried out by the CP. This will solely cause an unpredictable thread dispatching latency. The system cannot guarantee that a particular thread will be executed; nor is it possible to determine in advance when this might happen. The reception of arbitrarily sized segments causes NC^2_{CP}

[4]The notion of "message delivery latency" will be described in Section 4.3.5.

to transfer incoming data streams directly into the team (i.e., task) address space of the destination thread. This does not require NICE_{CP} to schedule threads, but only to verify the validity of the target address space segment. In contrast, end-to-end synchronous communication depends on thread scheduling and thus dispatching. Therefore, inhibiting the propagation of thread-dispatching events to the AP may impair synchronous but not asynchronous communication.

Nevertheless, there are a couple of applications demanding *preemptive communication*, that is, indicating the reception and delivery of a message by interrupting the AP. In order to adequately support these applications, the AP must be interrupted by the CP if, and only if, idle loop processing is currently not in progress.

As was mentioned earlier, the AP-to-CP queue stores packet and segment transfer requests. NICE_{CP} removes the requests from the queue and invokes the next lower (abstract) protocol processor (i.e., COSY). More specifically, `charge()` is called for every packet transfer request, and `stream()` is called for every segment transfer request (see Section 4.2). Queue processing is the task of the *cop* instance executed by the CP. Basically, the cop is the continuation of all AP threads that have issued transfer requests. The cop is a thread; but, in the majority of system configurations, it is the only CP thread, in contrast to the possibly many AP threads. The task of this single CP thread is nothing but listening in on the issuing of message transfer requests. That is, the cop waits on the transfer queue to become non-empty. Since there is only one CP thread, the cop mostly executes the CP idle loop.

The cop is the mediator for message transfers out of the node. This does not hold only for messages leaving the AP. In the configuration shown in Figure 3.7, the cop also handles CP messages to be sent out of the node. This second aspect comes into play only when the CP performs more operating-system tasks than message passing.

For example, in the case of specific MANNA configurations, the CP hosts kernel and POSE components implementing *virtual shared memory* on a paging basis. In such a situation, invalidation of the MMU_{AP} descriptors of replicated pages is done by the CP, impairing AP processing with only one exception. The exception is page invalidation, which usually entails flushing the MMU descriptor cache. Therefore, the AP must be forced to flush its MMU cache when the CP performs page invalidation. This will be accomplished by sending a processor interrupt to the AP. In addition to page invalidation and the propagation of MMU_{AP} cache-flush events, the CP drives (parts of) the page consistency protocol. It maintains information about replicated pages, such as the addresses of the nodes holding the page duplicates. All these functions will be carried out by higher-level kernel and POSE clerks (refer to Section 2.3.2) allocated to the CP.

3.3.3 Asymmetric Multiprocessing

From the application viewpoint, the second configuration described in the following introduces only minor changes. That is, the dual-processor node architecture is still hidden. The application has the view of a single, abstract processor. However, in contrast to the coprocessor approach discussed in the previous subsection, this abstract processor is implemented by a *runtime executive* layered above the nucleus and kernel, instead of by the nucleus itself. Figure 3.8 illustrates this configuration.

The application area of a kernel implementing such a configuration is a (multi-user) multitasking environment. Requesting the CP to carry out networkwide message transfers no longer involves the nucleus at the AP site. This is an important, overhead-reducing aspect when the kernel (and thus the nucleus) address space is isolated from the address space of the application task. Crossing protection boundaries may cause a serious degradation of the message startup time.

Logically interconnecting AP and CP, again, is by means of a *transfer request queue*. As indicated in Figure 3.8, the request queue is maintained by the runtime executive. A *giver* module implements the interface to the application task. It accepts and enqueues message transfer requests similar to NICE_{AP}, discussed before. The transfer queue is processed by an *agent*, executed by the CP. The agent plays almost the same role as the cop, except that it is a first-class thread and not implemented as a specialization of the CP idle loop.

In the case of a single-tasking mode of operation,[5] transparently

[5]This means sacrificing kernel isolation, that is, executing application task and kernel within the same unprotected address space.

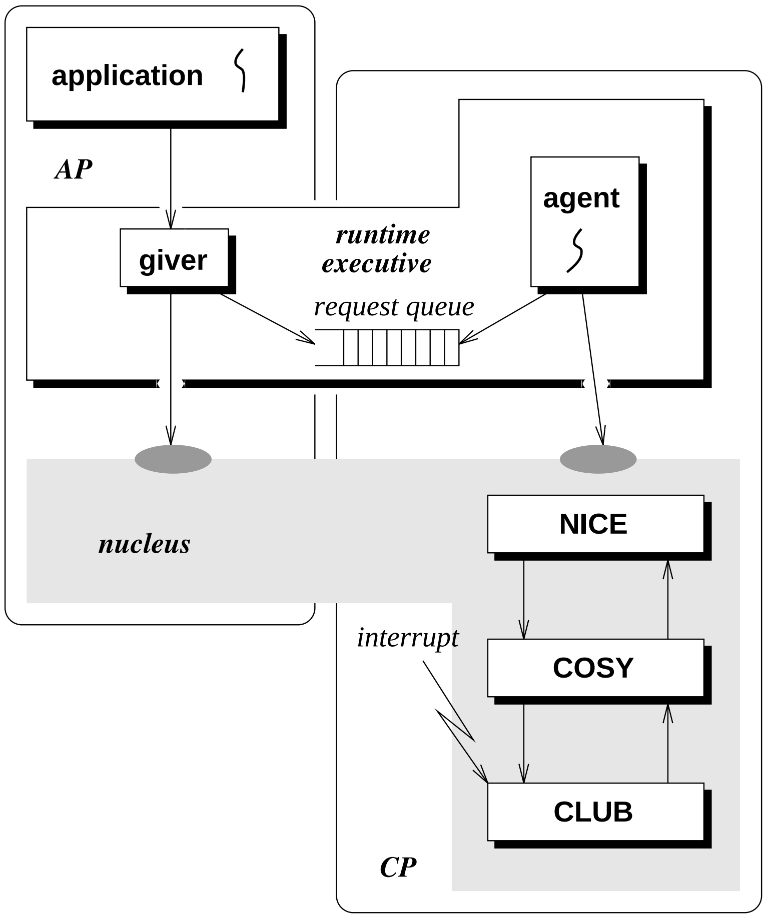

Figure 3.8: Asymmetric multiprocessing

to the application, the runtime executive is a truly *multithreaded program*. The program is executed in parallel by an *application thread* and a *communication thread*. Communication between both threads is via the common, shared team address space of the application task and happens entirely without nucleus intervention. In the case of a multitasking mode of operation, every AP task shares a communication segment with the CP agent. This segment hosts the request queue. The agent becomes responsible for processing several request queues, one queue for every application task executed by the AP. No thread communication, only synchronization, is done by the AP portion of the nucleus. Unless the agent operates in a polling mode, the giver sends a signal to the agent when enqueuing causes the request queue state to change from empty to non-empty. The nucleus provides the appropriate synchronization features.

This approach, in symbiosis with a dual-processor node, not only results in extremely low message startup costs but, compared to Figure 3.7, the execution path for performing the enqueue operation is shorter. In the case of MANNA, the startup cost amounts to approximately 2 μs. The message startup cost is decreased but the synchronization cost increased. This is because sending the CP an interrupt signal is more efficient than sending a signal to, and possibly readying and starting, a thread. Thus, compared to the coprocessor approach illustrated in Figure 3.7, the asymmetric multiprocessing case will be superior only if signaling the agent is a rare event. However, the frequency of agent synchronization events is highly application-dependent. This fact makes the asymmetric multiprocessing configuration basically nontransparent for specific application tasks. In order to obtain a positive speedup, the task may be compelled to figure out the optimal ratio between the issuing of message transfer requests, the agent processing speed (i.e., the average length of the request queue) and the thread synchronization delay.

By relying on a *shared-library* representation, the second aspect of the asymmetric multiprocessing approach is that handling the reception of messages now becomes subject to the user-level runtime executive even in a multitasking environment. That is, the AP is not involved in any networkwide message-passing activity, all of which is done by the CP. The AP solely executes nucleus code responsible for local thread synchronization. Nevertheless, thread synchronization must be executed by the AP—and, compared to the pure coprocessor solution, this might leave behind all the speedup achieved with the more efficient user-level implementation of the AP-to-CP message transfer request queue.

From the nucleus point of view, the functional dedication is more a question of the functions being called by user-level threads. Since the AP threads are no longer compelled to call the nucleus in order to request transmission or reception of messages, the AP can be relieved from any networkwide communication activity. The agent, which is only executed by the CP, is a thread specifically dedicated to communication. This thread invokes the nucleus communication primitives in order to transfer messages on behalf of the application thread (i.e., the AP).

In addition, both agent and application thread are bound to CP and AP, respectively. Thus, the CP idles even if there is at least one more AP thread ready to run. In this configuration, the nucleus implements only asymmetric multiprocessing, on the basis of functionally dedicated threads. Consequently, the assignment of these threads to the two processors determines which one will act as AP and which as CP—it becomes a user-level configuration decision.

3.3.4 Symmetric Multiprocessing

The third node configuration makes the dual-processor architecture completely visible to the application task. It relies on the presence of *multithreaded applications*. By "tearing down the wall" between the application and the node architecture, a further decrease of the message startup time may be the outcome. This is because the queue (or whatever) used for communication between AP and CP is moved closer to the application—the queue might even be implemented by a compiler.

Reflecting the discussion in the previous subsection, there is another consequence arising from bringing the architecture closer to the application. The consequence is sacrificing functional dedication of the threads, which leads straight to symmetric multiprocessing. It was mentioned that only the per-thread usage pattern of the nucleus primitives determines whether or not the AP is involved in network communication. This aspect still holds for the node configuration discussed here. Moreover, if both processors become equally able to execute threads, the distinction between AP and CP is no longer a static but a dynamic fact. Both processors share the entire kernel and are enabled to dispatch ready-to-run threads as long as the ready list is non-empty. This configuration is illustrated by Figure 3.9.

The symmetric approach supports parallel applications of two fundamental categories. It promotes the processing of fine-grain and communication-intensive applications as well as coarse-grain and com-

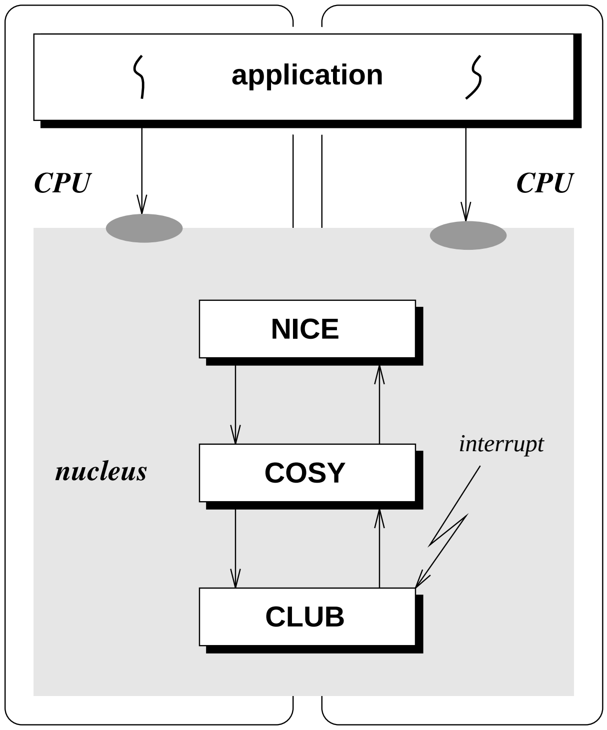

Figure 3.9: Symmetric multiprocessing

putationally intensive applications. In the first case, the application task will consist of dedicated computation and communication threads, similar to the asymmetric approach discussed before. In the second case, every thread of the application task executes the communication and computation bursts alone. That is, once the computation phase has been finished, the thread invokes the communication functions provided by the nucleus. If this is done by two threads at the same time, both processors logically execute the NC^2 protocol in parallel. A distinction between AP and CP is no longer made. However, due to the need for mutual exclusion when accessing common (nucleus-level) data structures, a true parallel NC^2 protocol execution will not take place.

It is worth noting that the symmetric multiprocessing approach also supports the other two node configurations. The difference is that symmetric multiprocessing is a general-purpose solution, in contrast to the special-purpose solution offered by the coprocessor and the asymmetric multiprocessing approach. Supposing a communication-intensive application and the requirement of keeping the message startup time extremely low, a nucleus implementing symmetric multiprocessing is clearly more overhead-prone than a nucleus implementing asymmetric multiprocessing or even the coprocessor solution. For example, the locking mechanism necessary for securing nucleus-level critical sections is alone more costly when comparing the symmetrical approach to the coprocessor approach. Similar comments hold for the scheduling strategies. Thus, a single solution is not adequate, since it cannot provide optimal support for the different application types. Rather, in order to support the dual-processor node architecture, the microkernel family introduced in Section 2.4 is to be enriched by specifically designed members implementing the various node configurations.

3.4 Summary

A parallel operating system must provide *operating-mode variety.* This not only includes a number of configurations of a *native operating system* for a bare parallel hardware. It also includes the possible representation as a *user-level runtime executive.*

Different native operating-system variants present a pure software solution to overcoming the message startup time problem in distributed-memory architectures. A distinction was made between three node classes. A *user node* is exclusively allocated to the processing of tasks belonging to a parallel application. In contrast, a *system node* performs global operating-system functions that are made available to user nodes. Finally, *generic nodes* are shared by user and system tasks and provide some sort of timesharing service. Depending on the requirements stated by the parallel application, different system configurations can be provided. It is worth noting that the distinction made between the above-mentioned node classes is a question of how to compose an operating system for the nodes of a parallel machine. Thus, it is a software and not a hardware configuration matter.

Understanding the parallel operating system as a user-level runtime executive is an important issue not only in testing and debugging parallel applications, but also in attaching a parallel machine to some host computer. In the latter case, heterogeneity problems (potentially) raised by using a different operating-system platform than that on the parallel machine are hidden to the application. That sort of "*guest-level implementation*" of a parallel operating system also supports parallel computing in a high-speed local-area-network environment. At the same time, downward migration of the parallel application to some real parallel architecture is possible.

The idea of a dual-processor node is to provide architectural support for latency hiding. This places specific constraints on the internal organization of a message-passing kernel. A dual-processor node, in which both processors are able to equally access the node resources, can be supported by a kernel implementing symmetric multiprocessing. However, this is only one of many possible and meaningful approaches. A symmetric multiprocessing kernel implements a general-purpose solution. The focus here is on a more or less balanced node resource utilization. Due to locking and scheduling overhead, the message startup time improvement achievable cannot be regarded as optimal. Other, special-purpose solutions must be supported by the kernel as well. Examples are a pure message-passing coprocessor configuration and asymmetric multiprocessing.

All these configurations are made feasible by having established a *family of parallel operating systems*. Each configuration is supported by a dedicated family member. All these configurations are able and, under certain circumstances, required to coexist. The operating-system family is extended by a *microkernel family* to be able to manage dual-processor nodes in an application-oriented manner.

Chapter 4

Operating Principle

It was mentioned earlier that communication performance is the most crucial factor in massively parallel, distributed-memory (MIMD) systems. The "big picture" of a high-performance parallel operating system for these architectures was given in the previous two chapters. An application-oriented system design was presented. This design (and its implementation) lets applications pay only for the functions they really need. The remaining risk of a communication performance bottleneck can be narrowed down to the "minimal but perfect basis" of the operating-system family (i.e., the nucleus). In order to get a feeling about the overhead of the minimal basis, the operating principle standing behind the nucleus is analyzed.

This chapter begins with a presentation of how an efficiently operating interrupt-handling subsystem can be implemented. Following that, the internals of the communication system structure are explained, as well as how this structure fits into the nucleus architecture. Service primitives are introduced and elucidated. A description of the network driver subsystem then helps to understand the operating principle behind the process of sending/receiving messages across/from the network. Finally, a model that approximates *active messages* is explained. In this model, user-level message handlers are described by means of *reactive objects*. All these discussions are with the goal of designing not only an upward- but also downward-scalable communication system.

4.1 Interrupt Handling

One of the most crucial problems in kernel design is to safely integrate device driver subsystems into the overall system structure. Usually, these subsystems will have to handle asynchronous events (i.e., *device interrupts*). They thus make the kernel entity nondeterministic, even if multiprocessing is not being used.

A kernel assigned to control a node of a (massively) parallel machine must be prepared to logically deal with two interrupt sources. The first one is caused by a *clock device* used to support preemptive (thread or task) scheduling. The second interrupt source comes with the actual *network device* that physically attaches the node to some interconnection system. When enabled, asynchronous events produced by both sources imply additional overhead for at least the following two reasons:

- Saving and restoring the processor state
- Synchronization with ongoing kernel-level activities

When disabled, a performance loss may occur due to the potential loss of events such as the signal which indicates that messages are pending for receive. The loss of a message event may be compensated for by polling the network driver when the next clock interrupt occurs. Since clock interrupt intervals are typically in the range of several tens to hundreds of microseconds or even milliseconds, the message latency will be increased—decreasing communication performance. In this situation, a message loss will be caused by the inability to compensate for the loss of a message-receive event by polling. Message loss requires the activation of retransmission procedures, and thus decreases communication performance and increases network load. Here, a trade-off exists. The designer of an operating-system kernel must decide which to pursue—whether to allow or disallow interrupts.

The subsequent sections discuss several aspects that play an important role in designing a lightweight interrupt-handling subsystem for the kernel entity. The first aspect addresses the problem of efficiently saving and restoring the processor state in order to start and finish interrupt handling. Afterwards, some event synchronization techniques are discussed, to motivate the approach of serializing asynchronous events by maintaining a schedule of when to start what activity. Finally, some remarks are made on how to let a busy waiting scheme coexist with interrupt handling.

4.1.1 Volatile and Nonvolatile Register Set

The conventional machine usually provides specific functions that support the handling of asynchronous events. Using state-of-the-art processor architectures, these functions, however, are by far too primitive to be directly exploited by an operating-system kernel without providing intermediate software abstractions.

Saving and restoring the *nonvolatile processor state* are typically the prelude and postlude of interrupt handling. The nonvolatile processor state is made of data sets that must be kept consistent when changing flows of control or processing modes (e.g., switching between the user and supervisor modes and vice versa). Most processors keep this state as small as possible. From the hardware point of view, it often encompasses only the program counter, stack pointer and status register. In contrast, the software may also require the inclusion of a the nonvolatile register set (explained below), used to store permanent program values. In addition, kernel-level software may also demand adding global variables such as the pointer to the currently active exception stack frame.

Different levels of abstractions will have different understandings of what the nonvolatile processor state ought to be. In the course of handling asynchronous events, it thus strongly depends on higher levels (i.e., application, problem-oriented language, assembly language and operating-system levels [87]) whether it suffices to save/restore the nonvolatile processor state as predefined by the conventional machine level.[1]

By using *problem-oriented programming languages* [86] to implement operating systems—which nowadays is standard procedure— it

[1] Note also that event handling means the partial interpretation of the interrupted program (refer to Section 2.6.2).

is mainly up to the compiler to determine what else must be kept consistent when asynchronously changing flows of control. For instance, optimization techniques try to schedule processor registers for frequently used program variables. These registers are exploited to store more or less permanent program data and therefore will constitute the *nonvolatile register set*. In contrast, the *volatile register set* will consist of processor registers that are used to store intermediate results and, as in the case of C and C++, function return values.

In order to invoke a subroutine, a compiler typically generates code that automatically saves (parts of) the nonvolatile register set into the *activation record* of that subroutine. This code is part of the runtime environment of the programming language used. Inside the subroutine, the entire register set can be exploited for speedup purposes. If that subroutine, in turn, calls another subroutine, the runtime environment establishes a chain (i.e., stack) of activation records. Upon return, (generated) code sequences will take care of restoring the register set properly. The volatile register set is not saved and must be handled explicitly.

Since it is assumed that the kernel entity is implemented in a problem-oriented programming language, one must only keep track of the volatile register set in order to safely invoke an interrupt handler, that is, to start handling an asynchronous event.[2] An assembly-level *wrapper routine* is required to perform all necessary functions of keeping the volatile register set consistent (Figure 4.1).

Usually, the wrapper routine is of very fundamental shape, so that it can even be automatically generated by some tool and need not be manually implemented. Serving an interrupt then works as follows:

1. The conventional machine level saves its minimal nonvolatile processor state and then passes over control to the wrapper routine.

2. The wrapper routine saves the volatile register set and invokes the interrupt handler.

[2]Actually, the designer is confronted with a dichotomy between optimized and non-optimized code: (1) the smaller the volatile register set, the faster the interrupt handler activation; and (2) the smaller the nonvolatile register set, the faster thread switching. In many cases, a small volatile register set entails a large nonvolatile register set.

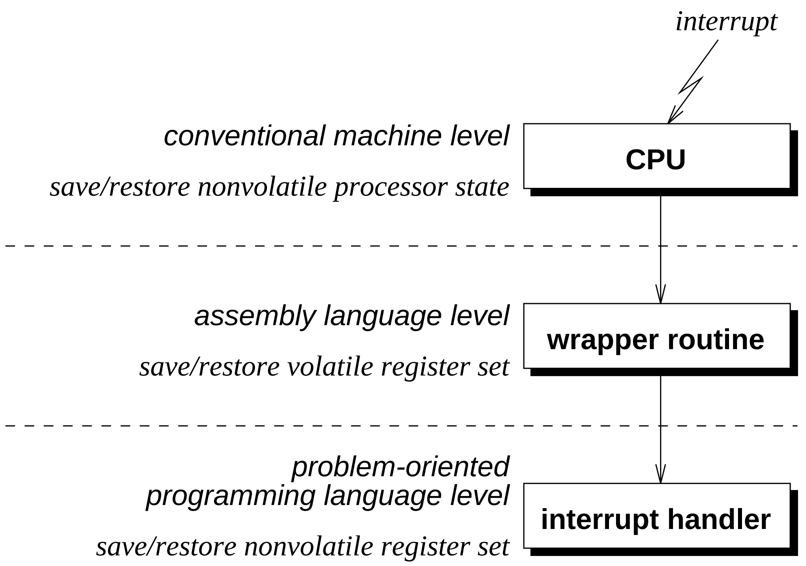

Figure 4.1: Fundamental steps in interrupt processing

3. Since it is written in a problem-oriented programming language, the interrupt handler automatically executes the compiler-generated code to save those nonvolatile registers likely to be changed.

Return from interrupt handling works vice versa. In a certain sense, to start an interrupt handler is nothing but performing a dedicated cross-domain call. The wrapper routine basically represents the procedure used to marshal and unmarshal function call arguments, except that the function to be called need not be furnished with actual parameter values. Procedure binding is static and corresponds to the index of the exception vector table that is related to the interrupt being served.

4.1.2 Event Synchronization

An event handler performs its task asynchronously to the interrupted program—it preempts the currently executing program. More precisely, event handling usually takes place under the control of the process that actually was in charge of program execution when the interrupt occurred. Since the event handler is part of (and the interrupted process may currently execute within) the kernel entity, synchronization is required when both parts access common data structures. Event handling means, for example, serving a clock interrupt in order to reschedule the currently executing thread. It may also mean serving a communication interrupt in order to deliver a message packet to a process and possibly set this process ready to run. In all these cases, event-handling procedures are assigned to share data with at least the nucleus.

Note that synchronization of an event handler and a process (i.e., a passive object and an active object) is required. There is no need to synchronize two (or more) processes. As a matter of fact, one cannot apply techniques such as *semaphores* [25] or *monitors* [37], which rely on a model that calls for the temporary suspension of processes when they try to enter an already active critical section.[3]

A simple and straightforward scheme is to perform a *hardware lock* on the entire kernel entity upon entrance by either processes or the event handler. This approach makes the kernel entity nonpreemptive and deterministic, but may raise serious problems in event processing. Unless the hardware provides sufficient buffer space to remember pending events and possibly associated data, the fairly long phase during which hardware locks are effective may cause the loss of further events, such as incoming messages.

The advantage of this scheme is that only one lock per kernel entity entrance is performed. Thus, synchronization overhead is reduced to the minimum. Also note that locks need not be explicitly released when performing a process switch from within the critical section. The new process that is switched on will have to leave the critical section anyway, since it must have entered the section somewhat earlier in order to have been rescheduled. Once scheduled again, a formerly suspended process

[3]In the case of a *semaphore*-style of locking, a `P()` may cause the calling process to block. As the event handler executes under control of the currently interrupted process, it is not allowed to use a `P()` for synchronization. Otherwise, a deadlock will occur when the process has already performed its `P()` and the event handler is trying its `P()`, both on the same semaphore, while the process is still inside the critical section and has not yet executed a `V()`. Similar considerations hold for *monitors*, as both approaches may block the currently executing process.

releases the lock that was acquired by the process that just relinquished control of the processor.

In order to overcome the problem of potentially losing events, an enhanced approach might be to place locks only at those places that really need to be secured. That is, instead of having only one large critical section, the kernel entity is partitioned into several small critical sections. This is a common approach often discovered in real-time kernels. There are three major problems with following this approach:

1. One must be sure not to have missed a single critical section. Otherwise, race conditions may produce undesirable effects or even nonreproducible errors. Debugging erroneous, highly concurrent and nondeterministic software becomes extremely difficult, if not impossible. This is not a problem if the entire kernel entity works in a monitor-like fashion.

2. The number of lock operations increases and thus may cause a performance bottleneck problem. Note that hardware locking in almost every case must be accomplished by using assembly-level instructions. When the high-level language in which the kernel entity is programmed does not provide appropriate locking features (such as is the case with C and C++), lock and unlock operations must be represented by subroutines enabling switching between different programming language levels. This further decreases performance.

3. Different nucleus/kernel layers get unfavorably interweaved with each other, leading to a more complex design and implementation. For instance, if a process has locked the nucleus and will be rescheduled afterwards, the process dispatcher is requested to save the current locking state of that process and to restore the locking state of the process that gains control. Thus, the per-process locking state is to be added to the nonvolatile processor state. In addition, this approach does not meet the goals of the family concept. It tends to let applications pay for unneeded service functions. For example, a process that does not relinquish control from within a critical section will nevertheless have to save and restore its locking state. A kernel-level process used to control

a device and immediately resumed upon an interrupt (without scheduler intervention) will be slowed down accordingly.

A compromise between both approaches is to place *software locks* on the entire kernel entity. This does not prevent event handlers from being called at the time an interrupt is signaled by the hardware. The kernel entity becomes able to quickly react to these interrupts, while still representing a single critical section. Consequently, event handlers are potentially able to interrupt kernel-level activities and, in turn, are enabled to initiate further ones. Activities with global meaning, namely, those accessing kernel data structures, then must be serialized in order to avoid race conditions.

4.1.3 Serialization of Asynchronous Activities

The design presented takes into account the fact that the nucleus must encapsulate critical data structures used for process scheduling and message passing. Consequently, only the nucleus must be secured properly, not the entire kernel entity.

Nucleus activities are manifold. They may be continuations of user-level threads that have called nucleus service routines, or they may be continuations of device interrupts. All these activities execute synchronously with their initiating source but potentially execute concurrently with each other. A device interrupt that occurs while a nucleus call is in progress potentially starts another nucleus activity, executing concurrently with the service routine that implements the call. In a shared-memory multiprocessor system, several nucleus calls may be issued in parallel, leading to a number of (possibly) different nucleus activities executed at the same time. Even in a single-processor system, preemptive scheduling may cause the same problem, except that only a pseudo-parallel execution of these activities takes place.

In all these cases, activity serialization may become necessary when the nucleus is invoked by either a thread or a device (i.e., software or hardware process) and an interrupt occurs. One can also argue that invocation on behalf of the thread is *normal*, whereas invocation on behalf of the device is *exceptional*. This promotes an approach in which activity serialization tries to minimize overhead when threads invoke the nucleus. In this case, the nucleus is not activated asynchronously but synchronously with respect to the currently running thread.

Serialization is split into two phases. These phases are *activity notification* and *activity propagation*. Activity notification will be the responsibility of the threads when entering the nucleus. Activity propagation is to be recognized by device interrupt handlers. When an interrupt handler wants to enter the nucleus and detects that there is still another activity in progress, the new activity is made pending. Pending activities are processed when the current activity leaves the nucleus. That is, the nucleus is immediately reentered by starting a pending activity. This procedure is repeated with each pending activity.

Activity notification is more basic than activity propagation. It may be accomplished by simply maintaining an *activity counter*, whereas activity propagation must set up an *activity schedule*. This separation of concerns reduces the synchronization overhead for the threads (i.e., normal invocation patterns) to a minimum and therefore strongly corresponds to the family idea.

The activity counter is incremented upon nucleus entry and decremented upon leaving. For this purpose, most processors provide indivisible instructions. An activity is made pending when the counter indicates at least one more activity in progress. Otherwise, the activity gets immediately started by both encountering and invoking the corresponding *activity object* of the nucleus. A schedule of pending activities remembers when to invoke which activity object. This schedule is processed by the current activity when it leaves the nucleus, provided that the activity counter has reached zero. The activity schedule is implemented by a queue (i.e., *waiting list*) of activity objects. As asynchronously executing event handlers will occasionally request nucleus service (i.e., may cause activities to be made pending) and interlocked handler invocations may take place, the waiting list must be secured properly. Some processors provide indivisible instructions that take care of a secured queue update.

The synchronization scheme just explained may be applicable without issuing any hardware locks by disabling interrupts. An exception are RISC processors based on a load-store architecture. This means that, for example, incrementing and decrementing the activity counter is not a single machine instruction but encompasses at least three basic operations:

1. Load the variable from memory into some processor register.

2. Increment/decrement the contents of this register.

3. Store the register contents back into memory.

In contrast, every CISC processor provides machine instructions to increment/decrement memory cells. Moreover, when applied on variables of storage class "*static*," standard C/C++ compilers are likewise trying to translate the unary operators "++" and "−−" into exactly a single machine instruction. These instructions are then indivisible, unless the nucleus executes in a shared-memory multiprocessor environment. Similar considerations hold for queue manipulations. Either way, even when interrupts are to be disabled to enforce atomicity, this happens only at very few, well-defined places. Also note that performance is (logically) improved by having defined specific primitives used to maintain indivisible counter and queue functions. Especially in the light of RISC architectures, these primitives then represent so-called *leaf functions* and can be subjected to an optimized calling sequence.[4] Based on a CISC processor that exhibits proper machine instructions (such as the MC68020), the abstraction introduced by these primitives must be made anyway, namely to integrate assembly-level (nonportable) code into a high-level language (portable) nucleus program.

In fact, the scenario just described delivers strong arguments against exploiting the most commonly used system programming language, namely C or C++, to implement a high-performance kernel for parallel processing. There are even more arguments related to programming aspects in general. The more appropriate system programming language supporting the implementation of indivisible operations without any overhead would provide integrated abstractions that are interpreted by the compiler. A state-of-the-art example of such a language is Oberon-2 [57]. This programming language issue raises another trade-off, namely, *suitability* versus *spreading*. The trade-off would not exist

[4] Using a SPARC processor, for example, leaf functions are not required to save and restore register windows.

with a totally integrated solution in which the operating system and the software development environment employed for implementation form a single "port package." However, such a solution will easily blow up the often limited manpower capacity of an operating-system project.[5] Under these circumstances, the best compromise is to employ a programming language, such as C++, which brings about a fairly good spreading of the operating system and causes only minor performance drawbacks.

4.1.4 Prologues and Epilogues

The scheme described so far burdens those parts of the system that potentially raise synchronization problems, namely the asynchronously executing event (i.e., interrupt) handlers. In addition, the scheme mandates the clear specification of activity objects that will be started in the course of interrupt handling. This assumes a proper structuring of event-handling subsystems, as demonstrated in Figure 4.2.

In the scheme presented, an interrupt-handling subsystem will consist of at least two distinguished components: *prologue* and *epilogue*. The prologue corresponds to an asynchronously executing handler module. It will be immediately activated when the interrupt is acknowledged, and thus served by the processor. Basically, the above-mentioned wrapper routine is the prologue entry point, and the high-level language routine called after having saved the volatile register set is the prologue continuation. Since the prologue executes with the priority of the currently served interrupt, only a very limited set of functions should be performed.[6] That is, a prologue must complete as soon as possible, so

[5]As demonstrated by the Oberon project, developing an integrated system consisting of compiler, runtime environment and operating system may be a successful undertaking, even for a small project team [95]. However, the big challenge would then be to neatly integrate different programming environments into a single system without violating the integrity of the underlying system software platform. Note that Fortran is the dominating programming language in the field of parallel (numerical) computing. An increasing acceptance of C and C++ can be observed. Employing Oberon-2 for scientific programming is far from near.

[6]It is an old operating-system principle that interrupt handlers should exhibit an extremely short execution path, not only to prevent deadlocks but also to avoid the loss of hardware events.

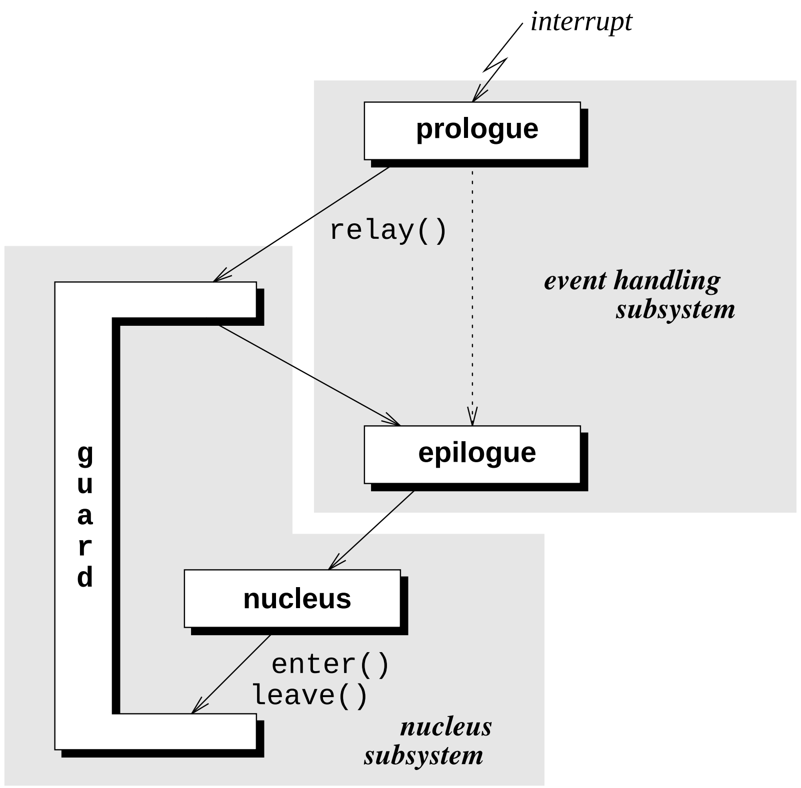

Figure 4.2: Structuring of event-handling subsystems

that interrupts of the same or a lower priority can be served again. Usually, a prologue merely drives a device protocol and occasionally queues data for later processing.

The epilogue corresponds to a synchronously executing handler module and actually represents an activity object. Its activation happens in a serialized manner (i.e., when no other nucleus activity is in progress). Once running, the epilogue may initiate unscheduling of and message delivery to threads, the only functions the nucleus performs. In fact, the synchronization scheme introduced serves to execute exactly these two functions safely and guarantees their integrity. In addition to the synchronization aspect, an epilogue always executes with a lower interrupt priority than its corresponding prologue. More precisely, when epilogue activation is granted, the execution proceeds with no interrupt priority being effective at all, just as if normal thread processing took place.

Cooperation between prologue and epilogue is problem-oriented and depends on what type of event is going to be handled. Handling of I/O events, for example, may cause the prologue first to queue data and then to activate the epilogue (provided that the epilogue is not yet running). In contrast, handling a clock event may get the prologue to immediately request epilogue processing in order to invoke the scheduler. If the prologue and epilogue will have to communicate via shared data, local synchronization becomes necessary. This means temporarily raising the interrupt priority. Note that only the epilogue is required to take care of synchronization proper, since only the corresponding prologue may cause data inconsistencies. The model assumes that, under normal circumstances, a prologue will not be interrupted by itself, but the epilogue may be interrupted by its prologue. Thus, in the worst case, local synchronization between prologue and epilogue means that the epilogue temporarily disables its prologue. The interrupt priority must be raised to a level that prevents the invocation of a specific prologue.

Switching from prologue to epilogue, and indicating that some nucleus activity is already in progress, is controlled by a *guard* object. Notification of an activity is accomplished via `enter()`, and cancellation happens by `leave()`. The purpose of `enter()` is to perform software locking (i.e., increment the activity counter). In order to request epilogue activation, the prologue applies `relay()` to the guard object. The invocation of `relay()` is furnished with the activity object representing the epilogue. This object is then made pending and placed on the epilogue waiting list. Waiting list processing, and thus the activation of delayed epilogues, takes place when the last prologue terminates or `leave()` is executed. With `leave()`, a check for pending activities is made and, if need be, locked processing continues with each activity found on the waiting list. Once finished, or if the urgency of the currently active thread does not permit processing the waiting list, the activity counter is decremented.

A similar procedure applies when a prologue terminates. However, an additional check is required. This is because the invocation of prologues may occur in an interlocked fashion, namely when the hardware implements several interrupt priority levels. Epilogues are processed with interrupts enabled; that is, on the lowest interrupt priority. Prologues always execute with a priority that prevents a recursive invocation due to the same interrupt type. The invocation of pending epilogues at the end of a prologue thus requires (re-)enabling the interrupts (i.e., lowering the priority level). As an interlocked prologue invocation is possible, (re-)enabling the interrupts and processing the epilogue waiting list must be done when no other ("stacked") prologue is active. The additional check means, for example, to verify the previous interrupt level or compare the previous and actual processing mode.[7] In this case, the last (or initial) prologue is found if the modes are different (i.e., the previous processing mode indicates "user mode") or the previous interrupt level indicates "interrupts enabled." This prologue then takes care of epilogue propagation, provided that pending epilogues are encountered.

It is worth noting that the guard abstraction explained also allows easy integration of other locking patterns, such as the global disabling of interrupts. In order to implement hardware locking, one must replace activity counter management (`enter()` and `leave()`) with instructions that raise and lower the interrupt priority accordingly, and activity schedule management (`relay()`) with the calling sequence to immediately invoke an epilogue. It also allows ignoring any synchronization measures when tolerated by the actual nucleus implementation.

4.1.5 Locking Delays and Preemption Points

The per-guard waiting list is a critical (shared) data structure, since `relay()` may be executed concurrently with `leave()`. An epilogue is just leaving the critical section via `leave()`, which processes the event list, while a higher-priority prologue is started. This prologue may request switching over to its epilogue by a call on `relay()`, which will add an activity object to the event list. The opposite case of concurrent

[7] Which method to pursue depends on the underlying processor. In the case of an i860, for example, the previous interrupt level is used.

execution will never hold. This is because `leave()` is always executed from within a synchronized part and, for this reason, cannot interrupt `relay()`. As a consequence, if the processor implements only a single interrupt priority level, then only `leave()` must take care of proper synchronization and temporarily exclude `relay()` from possibly being called. In the other, common case, namely when being confronted with several interrupt priority levels, both functions are subjected to mutual exclusion. That is, `relay()` must make sure that it cannot be (recursively) called while placing an epilogue object onto the waiting list. For these reasons, both `relay()` and `leave()` will have to perform a hardware lock, but only before attempting list manipulation or access.

This is the only place in the nucleus subsystem where all interrupts are temporarily disabled. More precisely, the locking is performed by the nucleus guard instance and not by the nucleus itself—the nucleus implements *interrupt transparency*, which is an important precondition for adding real-time capabilities. The inability to serve interrupts is thus bound by the complexity of the critical queue operations. In the queue implementation pursued,[8] only two operations need to be mutually excluded, namely advancing the queue tail pointer and resetting the queue head and tail pointer to the initial state. The critical data item is the tail pointer. This pointer is manipulated through a call on `relay()` and may be manipulated through a call on `leave()` (i.e., when waiting list processing takes place). Removing an epilogue object from the queue requires checking for the queue tail pointer and occasionally implies initialization of the queue descriptor. The queue descriptor needs to be initialized if the last epilogue object has been removed. In fact, these are two very primitive (critical) operations. Thus, enforcing mutual exclusion of these primitives will result in very low locking delays.

It depends solely on the event-handling subsystem whether parts of the kernel entity other than the nucleus guard must totally disable

[8]The tail pointer is a pointer to a link pointer and is initialized with the head pointer address. This makes an append operation straightforward and does not require an explicit check for the head pointer along with each append operation. An element is appended to the queue by making the link the tail pointer points to refer to the link of the new element. This ensures that the head pointer gets automatically updated with the first append operation.

interrupts. The need for local synchronization between prologue and epilogue mentioned above only means temporarily raising an interrupt priority to a level that prevents a specific prologue from being called by the hardware. Either way, if queues need to be secured and a 50-MIPS processor is assumed, the locking delay possibly caused by the guard object is far below 1 μs. If the processor provides indivisible operations for queue manipulation, the situation gets even better.

The discussed model leads to very short delays implied by hardware locks. Software lock delays, however, are orders of magnitude larger. They are determined by the longest execution path through the nucleus. The length of this path depends on the actual nucleus representation (see Section 2.4). In fact, the execution path is bound by the deepest calling hierarchy. Any time loops are executed, the nucleus voluntarily executes `leave()` immediately followed by `enter()`, thus polling the guard object. This function pair implements a *preemption point* and, being properly encapsulated, needs to be set accordingly. However, the delay implied by software locks will never exceed the current message startup time; this will be significantly below 10 μs when using a 50-MIPS processor.

Thus, the delays implied by software locks are not very serious even when compared to the constraints often seen in hard real-time systems. What may be serious in these systems is the queuing discipline to make epilogues pending. A FCFS strategy suffices to meet the needs for parallel processing, while a priority or deadline scheme may be preferable in the realm of real-time processing. The guard representation merely need be adapted accordingly. Except for the scheduling strategy, the nucleus may remain unchanged.

4.1.6 Integrating Busy Waiting

Interrupt handling implies overhead and is not necessary in all cases. Suppose no process is ready to run. In this situation, the processor sits idle. It will leave this state only when some external event causes a process to get ready again. However, since the processor has nothing else to do, it may also perform *busy waiting* on the occurrence of these events. In order to prevent interrupt-handling overhead in this case, interrupts are to be disabled while busy waiting is in progress.

Some processors provide a `stop` instruction for this purpose. However, in the presence of a *programmable interrupt controller* (PIC) chip, the more sensible approach is to let the *idle loop* of the scheduler instruct the processor to shut off all interrupts. Afterwards, the idle loop starts polling the PIC for some interrupt signal. If a signal is flagged, the corresponding driver subsystem is activated in order to acknowledge this "virtual interrupt" [63]. The effect might be that an epilogue is started and places one or more processes onto the ready list again. Once returned to the idle loop, the scheduler will discover ready-to-run processes, leave its idle loop and continue processing as usual (i.e., selecting a process for execution).

This approach becomes significant in the light of high-performance message-passing systems and parallel applications that exhibit a *compute and communicate* style of processing. All tasks of these applications execute their computation and communication bursts practically synchronously and at the same time, although the nodes of the distributed memory machine may work asynchronous to each other. They have to communicate in order to exchange intermediate results and thus to make progress in their computations. This means that each of these tasks sends out a message carrying results and then, in turn, receives messages (i.e., results) to continue local processing. Consequently, the schedulers (or processors) of all nodes involved in that computation are likely to enter their idle loops when the tasks begin to receive messages. Such particular circumstances may cause networkwide message passing to be executed almost interrupt-free, although the nucleus in principle assumes interrupt processing. Thus, the combination of interrupt handling and busy waiting is a meaningful strategy in order to specifically decrease the message startup time and generally speed up application processing.

There is another strong argument for exploiting busy waiting in order to listen for interrupt signals. In the realm of state-of-the-art RISC processors, the processing of hardware exceptions in general and interrupts in particular may result in prohibitively large delays in actually starting the handler routine. For example, the hardware exception handler for the i860 may easily result in 2000 lines of C code and has to take into account a number of obstacles defined by the pipeline architecture of this processor. In the case of the PEACE port for a 40-MIPS i860, a single trap/interrupt may cause a delay of approximately 50 μs before the first kernel-level C++ handler statement is executed. This is about three times slower than is the case of a 2-MIPS MC68020, which is a CISC processor. Effectively, there is a factor of 55 in trap/interrupt handling startup time decrease, although the i860 has 20 times higher scalar performance than the MC68020.

4.2 Abstract Protocol Processors

The nucleus consists of three main protocol layers (refer to Section 2.5.3). A *network independent communication executive* (NICE) controls systemwide communication and cooperation between threads. The *communication system* (COSY) implements a message transport protocol. This layer rests on a "*club of network drivers*" (CLUB) implementing physical access to the communication device.

Each of the three layers—NICE, COSY, and CLUB—is implemented by an *abstract machine*. Between these abstract machines, a functional hierarchy is defined that makes COSY only logically present. If the "COSY processor" is working, it maps all its primitives on the corresponding ones of the "CLUB processor" by executing a transport protocol. Otherwise (i.e., when no transport protocol is required), these primitives are directly passed through from CLUB to NICE without an intervening transport protocol (see also Figure 2.10). It is a static configuration decision whether COSY (i.e., a transport protocol) must be present and, if present, will increase the message startup time.

4.2.1 Abstract Machine Primitives

The elementary operations (i.e., primitives) of each abstract protocol machine are separated into two classes. These classes distinguish between *global significance* and *local significance* of a given primitive. Global significance means that processing the primitives causes network activities. This may imply the execution of a COSY transport protocol, unless the hardware is capable of doing everything needed to transmit and receive arbitrarily-sized message streams. Table 4.1 lists the primitives that have global significance. Local significance only implies the execution of local management functions, primarily for resource control purposes. Table 4.2 shows the corresponding primitives. Every primitive is applied to a buffer object that carries packet and segment descriptors. That is, a protocol processor is implemented by a class, and the instances created from this class are the buffer objects. The methods of these classes implement the elementary operations of the protocol processors.

Primitive	*Description*
`charge()`	Transfer of a packet with fixed, maximal size. The actual packet size may be smaller and is specified by an argument. The transfer request is logically queued. A second argument specifies the logical node address that designates the destination of the message packet.
`stream()`	Like `charge()`, however with a logically following segment of arbitrary size. The segment is placed ready for transmission, but it depends on the transport protocol layer whether it will be immediately transmitted or made pending until the peer site enables the transfer.
`accept()`	Acceptance of a segment pending for delivery. The segment is directly transferred into the destination address space.
`cancel()`	Abort of a transfer request either of `charge()` or `stream()`.

Table 4.1: Protocol processor communication primitives

All communication primitives provided by the protocol processor have been optimized for networks that support a flow-controlled message transfer. Cosy takes care of the proper virtualization if the underlying network of a concrete system configuration does not provide these capabilities. Additionally, the semantics of the primitives are defined such that a most efficient implementation of the communication model offered by the nucleus becomes feasible.

Primitive	*Description*
`upcall()`	Request some higher-level protocol processor to continue with the processing of a received packet.
`active()`	Check the processing state of a transfer request. Yields true if the request is still pending, i.e., in progress.
`donate()`	Add a packet buffer descriptor to the free list. The packet was either received earlier from some lower layer in the course of handling an upcall or was created by some management instance to overcome a temporary shortage of receive buffers.
`pilfer()`	Remove a packet buffer descriptor from the free list.

Table 4.2: Protocol processor management primitives

The primitives shown specify the minimal subset of communication system functions. Each of the three abstract machines logically implements the same subset, and may add system extensions if required. In an object-oriented sense, NICE becomes the specialization of COSY, which is a specialization of CLUB. NICE redefines the minimal subset provided by COSY, and so does COSY with respect to CLUB. If no transport protocol layer is needed (i.e., if COSY need not take charge of communication activities), then NICE is directly connected to CLUB and vice versa.

4.2.2 Protocol Sequence Expressions

Two main communication abstractions are provided by the nucleus. The `parcel` abstraction implements packet-based synchronous interprocess communication. In contrast, the `region` abstraction implements segment-based asynchronous inter-team communication. Exchanging packets implies short-term buffering in order to set up protocol headers and execute the communication protocol between the peer entities. Segment transfer, however, is based on a model that helps to avoid intermediate buffering and enables end-to-end communication between peer address spaces across the network. It basically extends fundamental primitives of the conventional machine layer, namely to load/store memory cells into a distributed-memory environment.

Both abstractions are implemented by a combination of the protocol processor primitives introduced in the previous section. The possible order in which these primitives are executable is specified by *protocol sequence expressions*. These sequence expressions manifest the vertical interactions between NICE and COSY (i.e., between NICE and CLUB if COSY is sacrificed).

4.2.2.1 Packet Transfer

Transferring a packet (`parcel`) end-to-end between processes normally implies two actions: a packet to be sent is placed ready for transmission by means of `charge()`, and a packet that has been received is made pending for delivery by means of `upcall()`. Thus, the issuing of `charge()` causes the invocation of `upcall()` at the peer (i.e., remote) protocol entity. This is described by the following sequence expression:[9]

$$\texttt{parcel} \ ::= \ \downarrow\texttt{charge}() \tag{4.1}$$
$$\uparrow\texttt{upcall}()\rightarrow(\downarrow\texttt{cancel}()\ \emptyset) \tag{4.2}$$

In order to transmit a packet, the `parcel` abstraction performs a `charge()` downcall to NICE (4.1), which hands over the function to COSY, which, in turn, downcalls CLUB accordingly. In order to process the receive of a packet and thus deliver a packet to the destination thread, low-level device driver routines will have to invoke `upcall()` to start packet processing by CLUB. In the course of being processed, the packet is forwarded to COSY and finally to NICE (4.2). At each level, protocol processing takes place, deciding whether to invoke `upcall()` in order to continue with the receive procedure at the next higher protocol level. Recall that in the normal case (i.e., a void COSY instance), `upcall()` is directly forwarded from CLUB to NICE.

[9] "↓" denotes a send and "↑" a receive action. By means of "→," events are chained that must be processed in sequence. The enumeration "(...)" groups a set of operations, from which exactly one may be chosen for execution. Thereby, "∅" implies no further activity (i.e., a void operation).

Having successfully passed the NICE checks, no further communication activities are required ($\emptyset$ in 4.2), and the packet will be placed onto the per-thread message queue. If a serious failure is detected, `cancel()` is called (4.2). Serious failures are, for example, the receipt of invalid NICE commands. In that case, NICE will also release the packet buffer descriptor immediately. This is accomplished by a call to `donate()`, applied to the packet in trouble. If the packet was successfully received and put onto the message queue, `donate()` is called by the `parcel` instance to finish the delivery request of the receiver thread. Thus, under normal conditions, `donate()` is called by that thread and indicates the end-to-end transfer of a packet.

4.2.2.2 Segment Transfer

A segment (`region`) is transferred by executing either `charge()` or `stream()`, depending on whether the segment is to be received or sent, respectively. Segment transmission, in either direction, implies handing over to the peer protocol entity a packet that announces a segment transfer request. The sequence expression describing segment-based data transfer is as follows:

$$\texttt{region} ::= \downarrow\texttt{charge}() \tag{4.3}$$

$$\downarrow\texttt{stream}() \tag{4.4}$$

$$\uparrow\texttt{upcall}()\rightarrow(\downarrow\texttt{cancel}()\downarrow\texttt{stream}()\uparrow\texttt{accept}()) \tag{4.5}$$

With `charge()` the packet encodes a load request (4.3), whereas `stream()` lets the packet carry a store request (4.4). In either case, `upcall()` is invoked at the peer site, indicating that a packet was received and is ready to be processed (4.5). The receive procedure for this packet works exactly the same as discussed before in the case of packet-based communication. It differs in interpreting the protocol information carried in the packet, namely sending or accepting the specified segment.

In order to load a segment from a remote address space under NICE control, `upcall()` simply invokes `stream()` (4.5). Segment loading is implemented by directing the peer site to store a segment at the site from which the request comes. The packet received carries all information to identify not only the destination of the segment but also which segment is to be sent. Thus, before `charge()` is applied, the `region` abstraction sets up a packet by explicitly saying which remote address space segment shall be loaded and stored into which local address space segment.

Hence, the main work in transferring segments is done by `stream()`. With `stream()`, a packet and a segment are prepared for transmission. The packet is used to encode the information needed to locate the per-thread address space segment to be overwritten, and to indicate that the segment data is pending delivery. That is, the model assumes a data stream with the packet as message header directly followed by the segment as message trailer. Again, the peer protocol entity executes `upcall()` first, upon receiving the header packet. Forwarded to NICE, the header packet requests verification of the validity of the destination address space segment. Having passed the check successfully, the address space segment is locked and filled by means of `accept()` (4.5).

Just as in the packet transfer case, `upcall()` may also lead to the execution of `cancel()` (4.5). With `cancel()`, the presence of an abnormal event is indicated to the next lower layer, causing the communication activity (related to the buffer to which `cancel()` is applied) to be aborted. This may involve end-to-end signal propagation, depending on the activity cancelled. Typically, the abnormal event corresponds to the error conditions mentioned above (i.e., an invalid NICE command).

4.3 Inter-Node Communication

The communication system is an integral part of the nucleus. It is activated upon two events (see also Figure 2.9): nucleus requests to send/receive messages on behalf of a process, and network requests to receive messages from the network device. The former class refers to *software events*, which usually happen synchronous to the current thread of control and will cause the activation of NICE. In contrast, the latter class denotes asynchronous *hardware events* signaled to CLUB and possibly propagated via COSY to NICE.

The sole function of the communication system is to connect the nucleus to the network. The purpose of NICE, then, is to translate nucleus requests of global significance into one or more COSY actions, and

COSY upcalls into one or more nucleus actions. The purpose of CLUB is to translate network (hardware) signals by means of an upcall into one or more COSY actions. Similarly, COSY takes advantage of CLUB to execute transport protocol functions in a hardware-independent manner. As discussed earlier in Section 2.5.3.2, whether COSY is really involved in the communication process mostly depends on the capabilities the network provides.

In fact, under "normal" circumstances, when relying on massively parallel systems with networks that are capable of transferring arbitrarily sized data streams, most of the work in making the nucleus operate networkwide is done by CLUB. Provided that COSY is not in charge of network communication (see Section 2.5.3.2), the message startup time is mainly determined by the structure and implementation of CLUB. Therefore, the internals of CLUB and the operating principle of this layer will be discussed briefly.

The following subsections discuss the operating principle of the communication system and take a more detailed look at what happens when messages are sent across or received from the network. For this purpose, the logical structure of CLUB is described first. Second, protocol interactions are considered, taking advantage of the specific structure of the message-holder abstraction. Third, *high-volume data transfer* as a means of asynchronously exchanging arbitrarily sized messages between address spaces is discussed. Finally, an issue already discussed in Section 4.1 is revisited, showing how busy waiting and interrupt handling are able to coexist within the network driver subsystem.

4.3.1 Logical Structure of the Network Driver

The CLUB structure, as illustrated in Figure 4.3, is defined by a *functional hierarchy* of modules. Note that a functional hierarchy does not prescribe how to implement the modules or module interactions [36]. The CLUB design is hierarchical and uses modules as the basic abstraction, but an actual implementation of the same structure is not mandatory.

The inter-module relationships expressed by the arrows in Figure 4.3 do not stand for procedure calls or the like. The presented design corresponds to the CLUB implementation for the MANNA distributed mem-

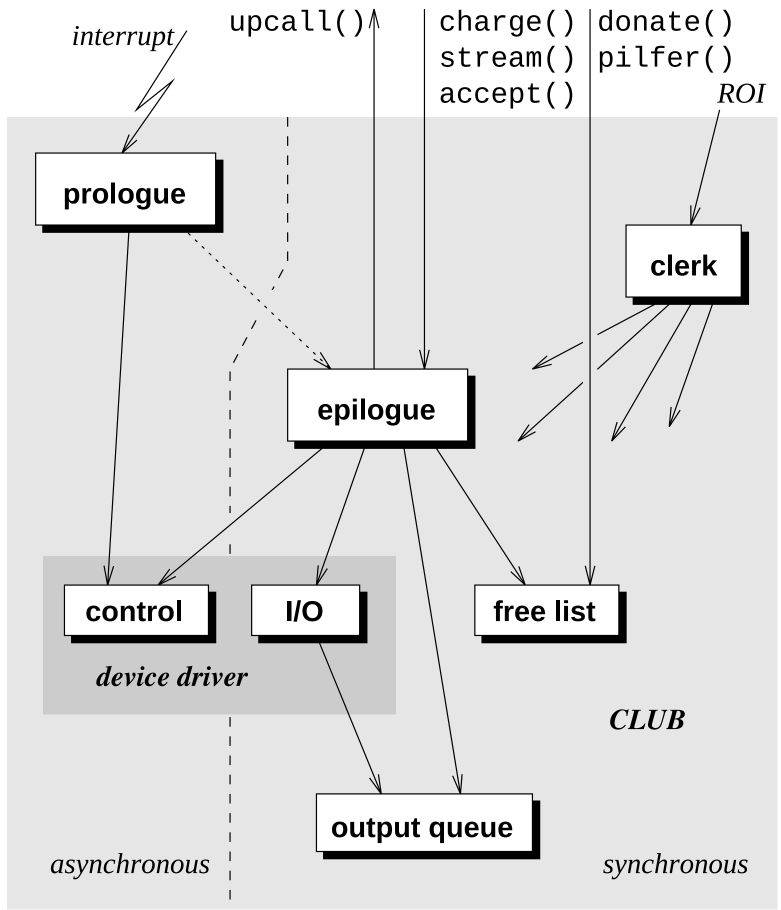

Figure 4.3: Network driver subsystem

ory machine [33] and identifies the following *functional units*:

prologue The interrupt-handling module. A communication interrupt causes this module to read the device status and acknowledge the interrupt request. Device access is via the **control** submodule. The epilogue is thus activated by the prologue only if the epilogue is not yet running (i.e., has not been interrupted by the prologue), and the device status indicates a valid interruption. The dashed arrow intimates that the prologue logically calls the epilogue (see also Section 4.1) after interrupt handling has taken place.

epilogue The CLUB main module. In order to process a received message packet, the epilogue removes a buffer descriptor from the free list, instructs the **I/O** submodule to fill the message buffer, and executes `upcall()` to initiate message processing at a higher level. The readiness to receive a message segment is indicated by `accept()`, applied on a message buffer descriptor previously forwarded by `upcall()` to COSY (or directly to NICE). Usually, `accept()` involves reading data from the network device (i.e., input FIFO) by instructing the **I/O** submodule accordingly. Initiated by the epilogue (on behalf of `charge()` or `stream()`), **I/O** is also responsible for transferring a message (into the output FIFO) out of the node.

device driver The device interface module. Basically, the device driver is structured into two submodules: **control** and **I/O**. The latter is invoked only during the epilogue phase, while the former is invoked during the prologue and epilogue phase. This structuring ensures that the **I/O** submodule need never be synchronized with the rest of the CLUB subsystem. The **control** submodule abstracts from device registers and does not implement a critical section. Thus, **control** may be called asynchronously in the prologue phase. In contrast, the **I/O** submodule uses the **output queue**, which implements a critical section. Calling **I/O** only in the synchronous epilogue phase implicitly guarantees the integrity of the output queue.

free list The pool of unused message buffer descriptors. The descriptors are free for use by the epilogue module in order to receive incoming message packets. A buffer descriptor is released (by `donate()`) by higher-level instances when the received message packet has been fully processed.

output queue The pool of accepted but not yet entirely processed message output requests. Epilogue gives the reception of a message preference to the transfer of a message. That is, if a COSY (or NICE) message is to be transmitted and, at the same time, a message is receive-pending at the network hardware interface (i.e., input FIFO), the incoming message is processed first. This may delay the transmission of a message by inserting the buffer descriptor into the output queue. The output queue is processed if the network hardware interface is not receiving incoming messages.

clerk The ROI interface module. This module makes CLUB remotely controllable. For example, it allows one to extend or shrink the free list, report the device status or monitor the output queue.

The CLUB structure shown in Figure 4.3 demonstrates the strengths of the interrupt-handling model introduced in Section 4.1. Unlike typical device driver subsystems, the CLUB implementation requires no specific synchronization pattern other than the distinction between prologue and epilogue processing. This is because the prologue and epilogue modules do not share common data structures. Rather, they only share the network device via the device driver module. That is, the CLUB modules need not temporarily disable the prologue (i.e., the interrupt handler) by raising the interrupt priority in order to secure a critical section—there is no critical CLUB section. Consequently, all CLUB functions execute without the artificial delay typically caused by interrupt synchronization primitives.

When detecting a receive-pending message in the input FIFO at the time a message is to be sent, the device driver delays sending the message by placing it on the output queue. Delayed messages are processed by the epilogue (i.e., I/O) only when no message is to be received. If the device driver (more specifically, the send function) is interrupt-driven, the epilogue would be forced to temporarily disable the CLUB prologue whenever output queue processing takes place. However, the send function of the device driver is part of the vertical, synchronous execution path that was established when a user-level thread called the nucleus for transferring a message. The send function is implemented by the I/O submodule shown in Figure 4.3. The asynchronous part of the device driver is technically represented by the control submodule, which only accesses the network device. Thus, removing elements from the output queue by epilogue need not be explicitly synchronized with the insertion of elements into the output queue because both functions are performed in the synchronous phase of the CLUB subsystem. The only synchronization measure is to switch over from prologue to epilogue (see also Section 4.1), as indicated by the dashed arrow in Figure 4.3.

4.3.2 Layout of a Network Data Unit

The data transferred by the network is never unstructured. In addition to the raw user data, the network data unit consists of information that directs the receiving site as to how to process the received message. Only this information is interpreted by the communication protocols. This raises the problem of how to logically represent and physically describe network data composed from (at least) two independent data sets. Both aspects are discussed in the following subsections.

4.3.2.1 Service and Protocol Data Unit

Typically, communication protocols distinguish between two sorts of data. First, a *service data unit* (SDU) carries information that is to be transported to a peer protocol entity without further interpretation by the communication protocol in charge of the transmission procedure. For instance, the SDU may represent the message originally generated by some application task. Second, a *protocol data unit* (PDU) is used to actually carry out protocol processing. A PDU is always interpreted by entities of the same protocol layer—"speaking the same language." It will tell the receiving site that, for example, an SDU is pending for delivery either to a thread or into some address space. As a consequence of this structuring measure, a single network data unit logically consists of scattered ("in-core") data fragments. The PDU and SDU need not be stored in adjacent memory regions in all cases.

Directly feeding the network user data without (software) protocol intervention and the need for generating a (software) PDU may be possible if the network hardware enables user-level access to some low-level output channel. However, even in this case the communication system has to deal with a structured network data unit consisting of a PDU and an SDU. This holds unless the network interface hardware is also able to manage virtual channels between user-level threads or address spaces. For scalability reasons, processes, not processors, must be the addressable units. It is exactly the function of NICE to dispatch processes (i.e., threads or address spaces) that are assigned to receive messages from the network. For this purpose, the sending NICE entity will have to instruct its peer what to dispatch and where to place the data, which obviously requires at least a PDU. Hardware that more or less provides NICE functions is presently not known.[10]

In the case study design presented, the maximal size of a hardware PDU that is submitted to and received from the network interface basically depends on the following aspects:

- If COSY can be bypassed, the largest SDU is defined by the user-level `parcel` abstraction. The `parcel` abstraction specifies the maximal size of a user-level message packet.

- If COSY cannot be bypassed, the largest SDU is defined by the nucleus-level COSY abstraction that describes a single `region` fragment. The size of this fragment corresponds to the maximal size of a packet created by the segmentation procedure of a COSY transport protocol.

The second aspect assumes that COSY implements message segmentation and reassembly in order to transfer arbitrarily sized (user-level) memory regions. These memory regions are described by `region` objects, carrying base address and length (in bytes). In this case, a single `region` fragment will always be larger than a single `parcel` object.[11] Taking into account the three protocol layers of the networking subsystem of the nucleus, a hardware PDU organization is given as shown in Figure 4.4.

At the NICE level, the SDU field carries a `parcel` or a `region` object. Which of these objects is stored in the SDU field depends on the nucleus message-passing primitive called. At the COSY level, the SDU field is large enough to hold the NICE PDU or a single segmentation unit.

[10]Even the Transputer does not provide these functions. It supports communication between threads, but not communication between address spaces.

[11]The current implementation uses packets (i.e., parcels) of 64-byte size.

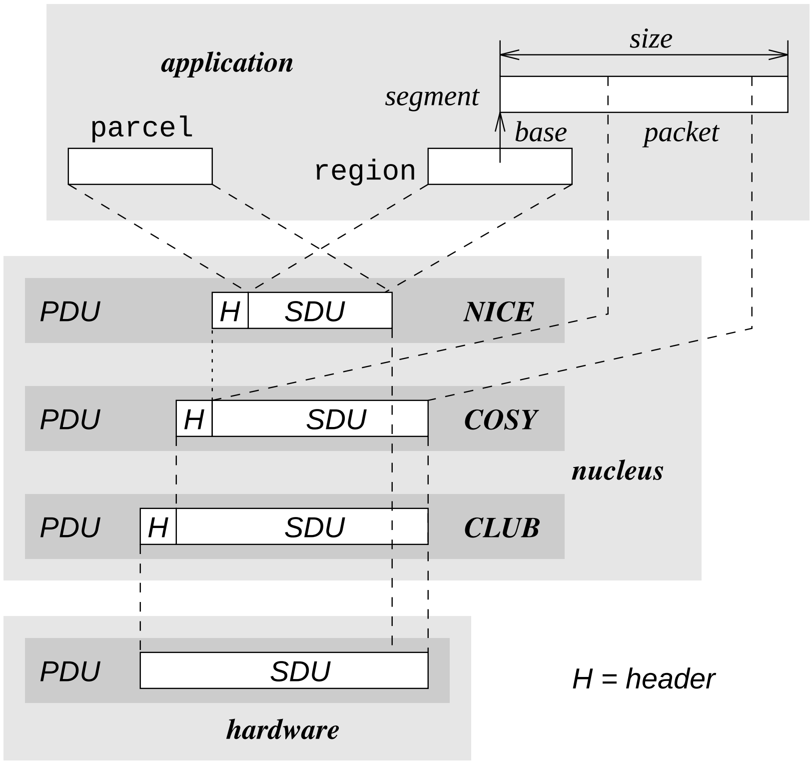

Figure 4.4: Correlation between the SDU and PDU

However, if Cosy need not perform segmentation and reassembly of an arbitrarily sized message segment, the SDU field at this level is only as large as the largest PDU defined by Nice. This is indicated by the dashed vertical lines in Figure 4.4.

Each of the three protocol layers adds header information to the SDU defined by the next higher layer. Nice and Cosy headers are manifold and depend on the actual implementation of the inter-nucleus and transport protocols, respectively. These headers are of variable size, depending on the protocol function they describe. The Club header basically carries as information only the size (in bytes) of the SDU to be transmitted across the network (i.e., is to be received by a peer node).

In specific configurations, the CLUB header additionally stores a *protocol entity pointer*. This pointer is used to multiplex the same network device driver between different protocol machines. With each network transaction initiated by the sender site, the CLUB header is set up with a protocol entity pointer that addresses the entity issuing that transfer request. By means of this pattern, the receiving CLUB entity is able to determine the COSY instance that is assumed to be able to process the incoming message. Transport protocol multiplexing and demultiplexing by CLUB supports a system configuration in which different COSY entities may coexist at the same time. Each COSY entity is then tailored to a specific class of applications that demand different data transfer capabilities from the transport protocol layer.

4.3.2.2 Message Buffer Descriptors

The problem for a communication system is how to compose a single logical data stream that consists of a PDU and SDU without the need for copying. For example, one solution is to reserve some space for the protocol header in the user-level message buffer. However, this will make user-level data structures independent from protocol-level data structures only if the protocol allows header information to be appended to the SDU it actually describes. This becomes a knotty point with an SDU being of arbitrary size. The other solution is to enable protocol entities to gather scattered information at the time a single data stream needs to be produced. The CLUB output module is the place where this function may come into play in order to submit user-level messages to the network. With proper hardware support, one may also let DMA units perform the gathering process "on the fly." In this situation, the output module will only have to set up the DMA units accordingly.

For all these reasons, CLUB data to be sent out to the network must be presented in a manner that eases the output process. Therefore, a *message descriptor* at the network-device-driver level will have to address at least two different memory regions, corresponding to the SDU and protocol headers, respectively. As Figure 4.5 illustrates, CLUB messages are managed by a *holder abstraction*, with the *header tank* describing where to find the protocol header and the *sequel tank* specifying a possibly present SDU.

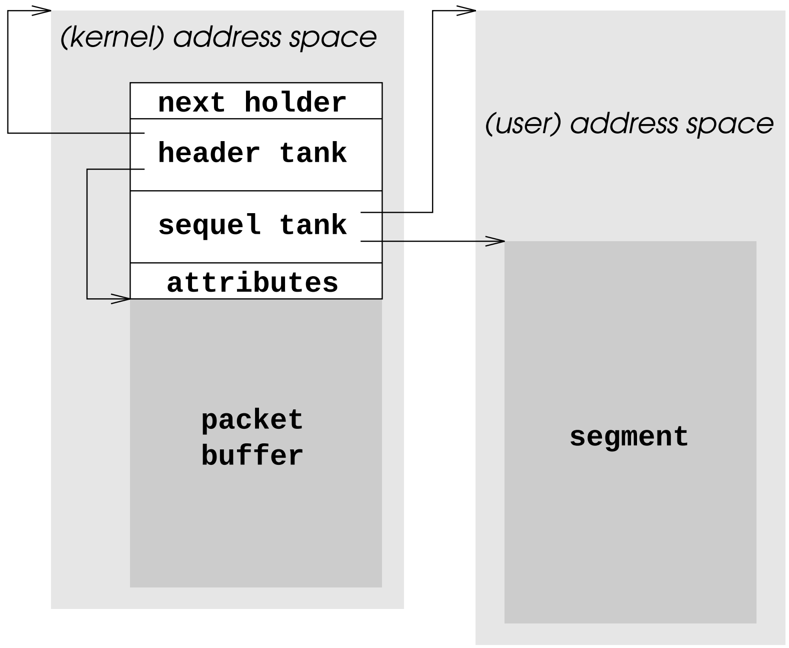

Figure 4.5: Message holder abstraction

In fact, a message holder stores a message descriptor followed by a PDU (packet) buffer object, with the header tank referring to that buffer object. The header tank is *invariant* once its enclosing holder object has been allocated. This makes buffer management more lightweight, since one must only keep track of a single kernel-level memory resource that stores both a message descriptor and a PDU. In addition, the extra space reserved is large enough to buffer the biggest PDU defined by NC^2 (i.e., the NICE-COSY-CLUB protocol).[12] The sequel tank, however, is *variant* and occasionally refers to an SDU.

As shown in Figure 4.5, the tanks of the holder abstraction keep track of the address space of a message fragment. Provided that the nucleus supports multiple address spaces, each tank stores the pointer to a page or segment table. In addition, the message fragment is specified

[12]If COSY is present and, for example, drives a transport protocol across Ethernet, the maximal size of the buffer object will be 1536 bytes.

by base address and size in bytes. A tank is a "logical DMA descriptor." In fact, this is also the right place to store physical DMA descriptors when network data transfer is carried out by exploiting DMA hardware.

4.3.3 Exploiting the Holder Abstraction

The specific holder organization explained above becomes significant at the receiving site of a NICE message. As was mentioned, outgoing data is processed by gathering the information addressed via header and sequel tanks and transferring the different data sets as a single logical message stream. Thus, at the receiving site a single message object comes in. When network data is actually received in DMA mode, a properly sized message buffer is needed to store the data. The holder object provides precisely this buffer space.

Packet-based user-level communication (provided by the `parcel` abstraction of the nucleus) is always supported with a single network transaction if COSY is not in charge of communication activities. With COSY enabled, and provided that data loss is not encountered, at most two network transactions are required: send a COSY packet that carries the NICE packet as SDU to a peer COSY entity, and let this entity return a receive confirmation. A "lazy mechanism" such as *piggy-backing* can be exploited by COSY to avoid extra network transfers.

In order to set up packet transfer, the `parcel` abstraction at the sending site defines the sequel tank of a holder object to refer to the user-level buffer which stores that packet. Every thread possesses a private holder object, and that object is used in the setup procedure. In the process of sending a packet, NICE is furnished with the holder object and accesses its protocol header (or PDU) by means of the header tank. The protocol header is initialized as required by the inter-nucleus protocol. Subsequently, `charge()` is applied on the holder object. Finally, a CLUB `charge()` is executed by either NICE or COSY (if enabled). This causes CLUB to transfer header and sequel data in sequence out of the node.

Unlike `charge()`, `stream()` and `accept()` support the transfer of arbitrarily sized data (i.e., segments). Since kernel-level buffering of segments at the receiving site does not only blow-up NICE but will also slow down communication performance, these primitives anticipate a model of end-to-end data transfer between user-level address spaces. NICE then has to check for the validity of these address spaces before the transfer can be enabled. Even if these checks are not required, as the actual nucleus representation does not implement address space protection, NICE will have to be in charge in order to start data transfer, since it knows the destination address at which to place the incoming segment. The destination address is taken from the NICE protocol header and set up by the `region` abstraction (more precisely, by the user-level thread) at the sending site.

Thus, segment transfer causes at least two NICE receive actions, namely `upcall()` and either `accept()` or `stream()`. By means of `upcall()`, the transfer request is forwarded to NICE on behalf of COSY (if enabled) and/or CLUB. NICE invokes `accept()` when the received PDU carries a write request and it invokes `stream()` in case of a read request. This leads to a two-phase inter-nucleus protocol in order to transfer a segment between peer address spaces across the network. Note that an active COSY instance may cause even more actions, depending on the segment size and the transport protocol supported.

4.3.4 High-Volume Data Transfer

Requesting the COSY protocol processor to accept a segment and directly transfer it into the destination address space may involve the execution of software responsible for message segmentation and reassembly, flow control, and sequencing. However, the other extreme may also be the case, namely that `accept()`, called by NICE, directly instructs CLUB (i.e., the DMA logic) to blast the data received from the network into the corresponding memory region without further processing. If the hardware implements a transport protocol sufficient to support parallel applications, COSY disappears, and all work is done by NICE and CLUB. Otherwise, `stream()` and `accept()` are carried out by COSY.

COSY has two choices for transferring a segment. It may execute a stream-like protocol by following the pattern mentioned above, namely setting up a message stream consisting of packet header and segment trailer. The second choice is to transfer only the packet and instruct the peer protocol entity to signal when to send which memory fragment. In the former case, `stream()` takes care of message composition and, possibly, generating packets that are to be transferred in sequence over a virtual channel. Actually, it may also start transferring all the packets. In the latter case, `accept()` tells the peer protocol entity to start the packet transfer, unless this has been done already. The purpose of `stream()` is to provide COSY with all the information necessary to go either way.[13] Whether `stream()` or `accept()` plays the active role in order to yield an end-to-end segment transfer strongly depends on the network capabilities

4.3.4.1 Sender-Initiated Data Transfer

A typical case in massively parallel systems is to let `stream()` play the active role. The message header packet then contains a segment descriptor, and the message trailer actually represents the contents of that segment (Figure 4.6).

Upon receive at the destination node, `upcall()` causes NICE only to verify the destination address space selected by the segment descriptor. While this verification is in progress, the message trailer is already being received by the low-level network device. The trailer (or parts of it) streams into the receiver FIFO and remains there until `accept()` takes place. NICE will issue `accept()` as soon as address space verification succeeds. The call guarantees the presence of the destination address space that was assigned to receive the segment. Finally, it directs the network device to empty the FIFO and forward data accordingly.

The transfer procedure works slightly differently when DMA hardware is exploited. In this case, the DMA descriptor used for delivering the message trailer must be set up from the information received by the message header. This is what `upcall()` does. The execution of `upcall()` due to the receive of the message header packet extracts the necessary information from that packet. Called by `upcall()`, `accept()` then gets furnished with a properly initialized holder object and starts

[13]This is one of the major differences from the object-based PEACE kernel developed for SUPRENUM. In that system, the inter-nucleus protocol decided how to achieve segment transfer. The lower-level protocol had no choice to efficiently carry out a stream-like data transfer. In fact, this is the main reason for having introduced separate layers in the new design, namely, NICE and COSY. Previously, both layers constituted the inter-nucleus protocol.

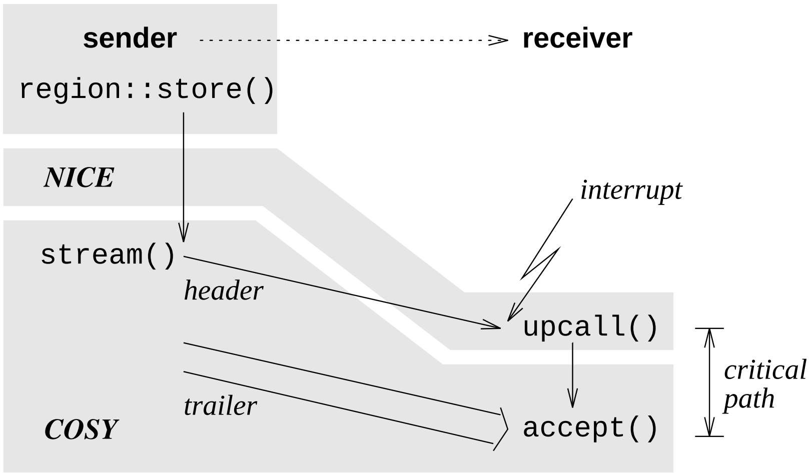

Figure 4.6: Sender-initiated high-volume data transfer

the DMA receive procedure.

In both cases (i.e., both the DMA and FIFO models), the sender-initiated data transfer assumes that (back-pressure) flow control is performed entirely by hardware. The sender site will temporarily stop the data transfer if the receiver site is unable to process the incoming data in time.[14] In particular, as indicated in Figure 4.6, the *critical path* that may cause the activation of flow control is from signaling the (header packet) receive interrupt until calling `accept()`. Most of the delay is due to software, that is, carrying out the NC^2 protocol. This means receiving and decoding the message header and making a memory region ready to accept the incoming message trailer. Basically, these measures are driven by network events (bottom-up), but must be controlled by Nice (top-down). Only Nice knows about threads and their associated address spaces. Therefore, Nice must be in charge of delivering the incoming message trailer directly into the destination address space.

[14]In the case of (massively) parallel computers, the network itself implements flow control. For example, using crossbar switches, hardware signals generated by the receiving end indicate the half-full or full state of a receiver FIFO. The signal propagation delay typically amounts to a few clock ticks, meaning nanoseconds and not microseconds.

4.3.4.2 Receiver-Initiated Data Transfer

The sender-initiated data transfer works efficiently in two cases. The first case is that in which the network hardware interface provides sufficient FIFO buffer space to compensate for the delays at the receiver site. The second case is that the nucleus is designed and programmed by paying attention to the receive latency between message header and trailer. That is, a sender nucleus knows when to initiate the transfer of the message trailer such that its receive becomes possible in time. This sort of *rate control* is discussed in the next subsection. If either of these two cases does not hold, the more sensible approach is to have the receiver, not the sender, site play the active role in the transfer procedure (Figure 4.7).

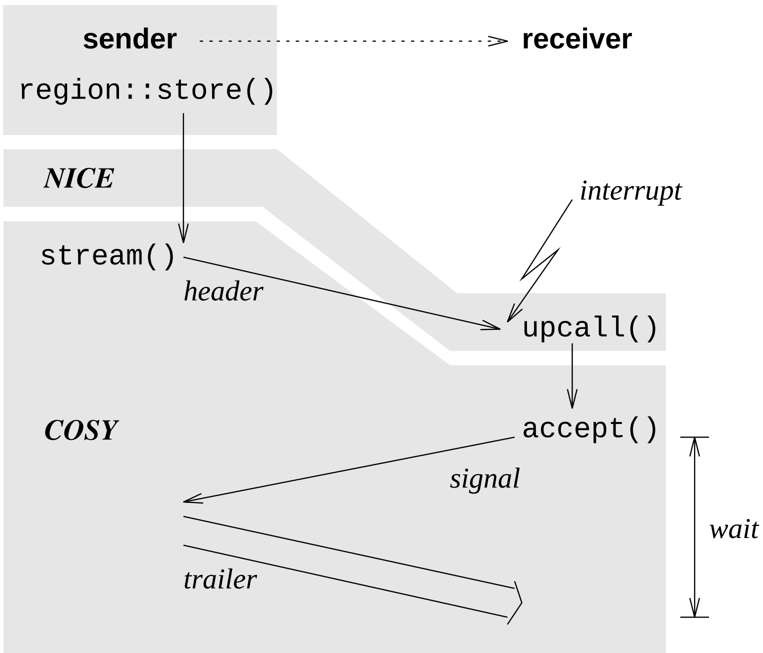

Figure 4.7: Receiver-initiated high-volume data transfer

In the receiver-initiated case, `accept()` will be used to synchronize the beginning of a data transfer with the readiness of the receiver process (or address space) to receive data from the network. Again, the sender issues `stream()` to request the transfer of an arbitrarily sized message. However, in contrast to sender-initiated data transfers, only the message header is transmitted. Upon receive at the destination node, the header packet is forwarded to NICE by means of `upcall()`, which takes care of the verification process. In particular, the call of `accept()` signals to COSY (and/or CLUB) that the destination segment is ready to receive data from the network.

At the COSY level, `accept()` causes the transfer of a control PDU to the site from which the message header was received. This PDU enables the transfer of the trailer segment corresponding to the message header. In fact, NICE applies `accept()` on the message header that perfectly describes the remote segment which needs to be read. The COSY-level PDU carries this information to the sender site. There, COSY starts transferring the entire segment, either in one chunk or fragmented into packets (depending on the transport protocol used). The message trailer arrives at the destination node and is forwarded into the destination address space.

The receiver-initiated transfer strategy implements a three-phase protocol, whereas in the sender-initiated case, a two-phase protocol is executed. The additional phase is signaling when the transfer of the message trailer may happen. This phase confirms the "connection establishment request" stated by the first phase (message header transfer). The connection guarantees the reception and delivery of the incoming (arbitrarily sized) data segment. It is implicitly torn down at the sender site when all data has been transferred. The implicit connection teardown at the receiver site takes place when all data has been received.

In the presence of "intelligent" DMA logic, the connection establishment phase could be exploited to prepare a DMA descriptor that will be automatically selected by the receiving node when the message trailer arrives. The message trailer is tagged with a hash key uniquely addressing a DMA descriptor at the destination node. Upon receive, the hash key is used by the receive logic to look up the DMA descriptor corresponding to the message trailer. The hash key is generated by the receiving COSY entity during the connection establishment phase and passed to the sending COSY entity with the signal message indicating readiness to accept the message trailer. Thus, the DMA hardware must only interpret the hash key. The sending COSY entity will encode the hardware PDU to be interpreted by the receiver DMA.

COSY performs the proper virtualization if the DMA logic just described is not implemented by the actual network interface hardware. In that case, the arrival of the message header causes COSY (and not NICE) to perform the DMA descriptor lookup. This scenario compares with the NICE-level sender-initiated data transfer described previously. However, the difference is that COSY, rather than NICE, carries out the actual message trailer acceptance procedure. The critical path to invoke `accept()` gets shorter. A verification of the validity of the data transfer is not required. This was already done during the connection establishment phase. What remain are interrupt handling and rehashing. Interrupt handling can be avoided if COSY polls CLUB in order to receive the message trailer. This is indicated by "*wait*" in Figure 4.7.

The discussion showed that flow-controlled data transfer between the sending and receiving nodes cannot be entirely sacrificed. However, the probability of the necessity to activate flow control can be lowered. A hardware-implemented DMA descriptor lookup will be the optimal solution. Nevertheless, a software-implemented COSY-level lookup is a quite reasonable and efficient DMA hardware virtualization. Finally, it is worth noting that this virtualization becomes a meaningful COSY property when DMA cannot be exploited in the transfer procedure at all.

4.3.4.3 Reassembly of Segmented Messages

NICE merely sets trigger points for when to begin data transfer. This approach leaves to COSY whether buffered data transport takes place. Buffering at the sender site to have COSY set up a PDU can be avoided, provided that the network hardware interface supports atomic data transfer of a sequence of packets. The holder abstraction enables COSY to perform segmenting without the need for intermediate buffering. It may exploit the sequel tank to select user-level message segments, while the header tank keeps track of the protocol header (i.e., PDU) used to transfer a single segment by means of COSY.

Whether message reassembly is possible without buffering depends on the hardware capabilities at the receiver site. If COSY is not supported by a FIFO-based interface, a buffer is required to temporarily hold the PDU for further processing. The same holds also when a DMA is used to transfer data from FIFO registers into memory or the FIFO store is too small. Otherwise, COSY may perform just as NICE performs to deliver an arbitrarily sized message to the destination address space. First, it gets activated via `upcall()` (issued by CLUB), which actually forwards the PDU to COSY. Second, the PDU is interpreted by COSY by driving the transport protocol and reading information taken via the header tank of the holder object. Third, COSY may apply `accept()` to finally perform message reassembly within the (user-level) destination address space.

By that means, COSY is also able to cope with the problem of out-of-sequence packets arriving at the destination node. Instead of reassembling messages according to the order specified by the sequence number, COSY is able to compose the original user-level message in the order in which packets are going to be received by the network interface. In this case, the relative position in the destination address space of every packet is to be explicitly specified in the PDU and used by the reassembling COSY entity to set up a sequel tank. Once set up, COSY invokes `accept()` and causes CLUB to transfer the next `region` fragment from the network directly into the right place of the destination address space.

4.3.5 Communication Latency

The sender-initiated high-volume data transfer requires fewer communication phases than the receiver-initiated transfer. There is no need for an additional network transaction to signal when to transfer the data stream. However, this approach also assumes synchronism between both sending and receiving parties. The problem is to initiate the segment transfer at a time that makes it highly probable that the receiving site is prepared to accept the incoming data. Roughly speaking, the latency of COSY in transferring a segment must be greater (or equal to) the latency NICE causes when settling the readiness for segment delivery.

The delay between data transfer issued by the sender site and delivery at the receiver site depends on a number of factors. At the sender site, on behalf of `stream()`, the network hardware interface needs to be provided with both message header and trailer. At the receiver site, on behalf of `upcall()` and `accept()`, the network hardware interface needs to be instructed to deliver arriving data into the destination address space. The elapsed time from starting execution of `region::store()`[15] to transmit a user-level message segment until finishing `accept()` to deliver that segment to the receiving thread (i.e., destination address space) is the *communication latency* (see also Section 1.3.1).

The communication latency depends on the *node latency* for sending and receiving a message to/from the network, and the *network latency* for actually transferring the message to the peer node. Node latency is mainly a software issue, whereas network latency depends primarily on the communication hardware. Thus, in order to reduce the message startup time, the node latency should be kept as small as possible—and this calls for a proper design and implementation of the nucleus and the embedded communication system.

The "coarse-grain" approach is to rely on a kernel family. That is, node latency is reduced by letting applications pay only for the service functions they really need. This approach is refined based on a communication paradigm by means of which arbitrarily sized messages are exchanged in a pipelined manner and directly between the address spaces of the communicating threads, without intermediate buffering. The transfer of a memory segment means basically sending a message header immediately followed by a message trailer. At the receiving site, the message header is used to announce the arrival of a message segment and to enable the delivery of the incoming data to the destination address space. The idea is the following:

- Overlap the header transmission with the procedure locally executed to set up the message trailer.
- Overlap the trailer transmission with the procedure remotely executed to deliver the message segment.

Trying to overlap the above-mentioned communication phases is motivated by the "short-term buffering" capabilities of the network.

[15]The `store()` primitive is used by the threads to write directly into a remote address space `region`. This primitive is implemented by `stream()`.

Typically, there are two categories of buffers. These are the "communication wire" and the sending and receiving FIFO registers of the network hardware interface. If DMA is used in the transfer procedure, an additional overlap makes sense if the memory bandwidth is not fully utilized. As long as the CPU can run memory cycles and/or primarily performs code/data cache accesses, the fetch-execute cycle is not delayed and operates at full speed. This makes the processing of CPU instructions (in particular, nucleus software) appear in parallel to the networkwide transfer of memory cells.

The scenario of trying to overlap message setup and transmission phases is illustrated in Figure 4.8, which shows a fine-grain breakdown of the communication latency. Network latency comprises two parts: *packet latency* (i.e., message header transmission) and *segment latency* (i.e., message trailer transmission). It depends on the network architecture whether both parts will accumulate to the same delay. Note that the network may be tuned to specifically support the transfer of small and fixed-size packets [23]. Node latency is the sum of *sender latency* and *receiver latency*. The former is due to the header and trailer setup procedure, referred to as *header latency* and *trailer latency*, respectively. The latter is due to (1) signaling and handling a communication interrupt, (2) receiving a packet from the network, (3) forwarding this packet to NICE, (4) verifying the destination address space segment, and (5) accepting the message segment from the network. The last four steps determine the *delivery latency* of a segment and are the most crucial factors in trying to reduce the communication latency. In principle, these steps are performed by the portable part of the nucleus. The first step listed relates to the nonportable nucleus part and is highly processor-dependent. The latitude for tuning the receiver latency is thus the fraction made up by the delivery latency.

Achieving message delivery on time is a question of the receiver latency. The longer the execution path between signaling the communication interrupt (in order to receive the message header) and accepting the message segment, the longer the receiver latency. The length of this path depends on the processor architecture and on the operating mode of the receiver node. State-of-the-art RISC processors may cause a prohibitively large performance decrease of interrupt-driven

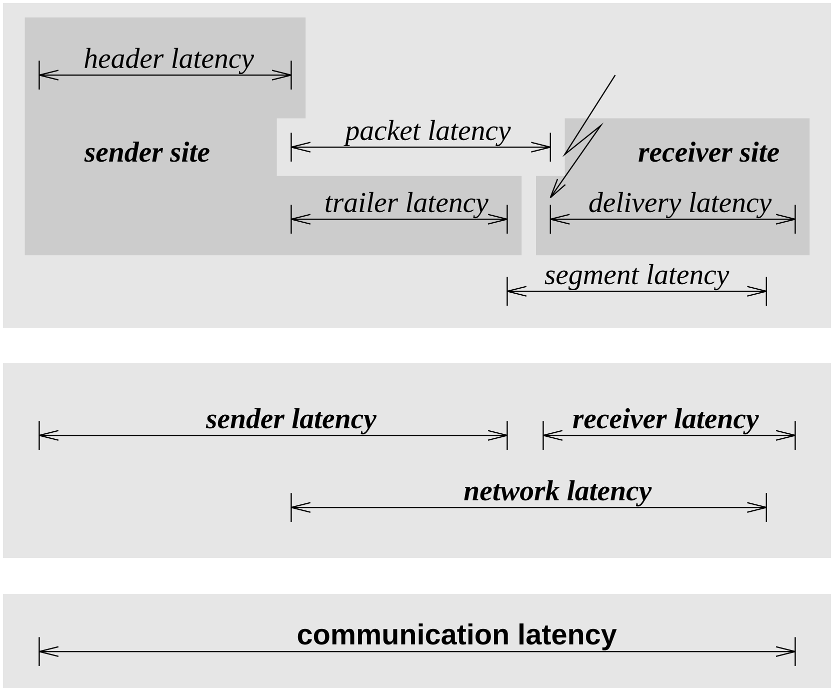

Figure 4.8: Communication latency breakdown

communication systems (see also Section 4.1).[16] A single-tasking mode of operation calls for significantly fewer nucleus functions in the receive procedure than a multitasking mode. Consequently, the delivery latency of a single-tasking node is smaller.

In order to overcome the receiver latency problem, two solutions are possible. The first solution applies a different communication pattern and relies on receiver-initiated high-volume data transfer. This pattern causes the COSY entity at the receiver site to explicitly signal when to transfer the message trailer, which requires the execution of a *handshake protocol*. Similarly, the COSY entity at the sender site only transmits the header and then waits on the signal to start the trailer transfer.

[16]This is the main reason for a polling mode of operation to perform networkwide message transfers in the CM-5 architecture [23]. The CM-5 is based on SPARC technology.

The second solution is to still rely on the sender-initiated high-volume data transfer but introduce a *delay slot* between the transfer of message header and trailer. This basically means that the COSY performs *rate control*. The delay slot is computed according to the following formula:

$$\begin{aligned} T_{packet} &= L_{packet} + L_{delivery} \\ T_{segment} &= L_{trailer} + L_{segment} \\ \\ T_{delay} &= T_{packet} - MIN(T_{segment}, T_{packet}) \end{aligned}$$

However, introducing sender delay slots is a meaningful undertaking only if message delivery within a known and fixed time frame can be guaranteed. This calls for *real-time capabilities* of all nucleus instances involved in networkwide high-volume data transfer. In particular, it requires NC^2 to implement a (scaled-down) *real-time communication protocol* [29]. In addition to that, the network hardware must provide guarantees for the allocation of packet transfer slots. There must be a fixed and known (worst-case) delay from which the occurrence of the remote communication interrupt due to a transmitted header packet can be deduced. This delay is the packet latency. Distinguishing between the prologue and epilogue phases and the introduction of *preemption points* (see also Section 4.1) are the main aspects that make the nucleus efficiently responsive to high-volume data transfer requests.

4.3.6 Busy Waiting and Interrupt Handling

Relying on a pure interrupt-driven approach in order to react to network events may cause unnecessary performance bottlenecks (this issue was already discussed in Section 4.1). Therefore, if the scheduler enters the idle loop, interrupts are disabled and the PIC is polled for external events. Flagging an interrupt signal causes the idle loop to invoke the corresponding device driver subsystem. In the case of a network event, the network device driver (and thus CLUB) is requested to serve that virtual interrupt.

CLUB activation from within the idle loop simply means invoking the prologue. In principle, there is no difference from invoking the prologue on behalf of an interrupt, except for the following cases:

- Time-consuming RISC housekeeping measures are omitted (this does not apply for CISC-type processors).

- The nonvolatile processor state need not be saved and restored.

- The volatile register set need not be saved and restored.

- The wrapper routine need not be activated to pass between different programing-language levels.

For all these reasons, prologue activation from within the idle loop is significantly faster than activation by interrupts. Once activated, the prologue (usually) is not aware of the real trigger function. The prologue proceeds independent of whether the idle loop or the interrupt requested device attention.

There is a second kind of "out-of-order" CLUB prologue activation. This is the case when a *watchdog timer* is used to enforce cyclic CLUB attention. The watchdog enables the handling of network events if network device interrupts are lost. This feature is indispensable in the process of porting the nucleus to a different hardware platform or helping to make newly developed hardware stable.

Thus, the CLUB prologue may be executed on three different interrupt priority levels, as defined by the network device, the clock device and the idle loop. Nevertheless, in any case only a single CLUB epilogue incarnation will be active. As discussed along with Figure 4.3, the CLUB prologue "calls" its epilogue only when the epilogue is not running. A flag indicates the epilogue processing state. However, checking the flag and, occasionally, activating the epilogue must be an indivisible operation. A prologue called by the idle loop may be interrupted by a real network or clock event and thus recursively activated. The attempt of switching over from the CLUB prologue to the epilogue has to take care of this fact in a problem-oriented manner.

4.4 Approximating Active Messages

The communication protocols discussed so far provide minimal overhead in order to exchange arbitrarily sized user data.[17] This particularly holds for the sender-initiated data transfer. However, the discussion has not yet addressed the issue of how to bind incoming messages to the data (i.e., program variables) processed by the application program. A similar problem is how to bind program variables to outgoing messages. In both cases, the binding process calls for specifically designed system software features and proper programming models. The first aspect is closely related to parallel operating systems and is therefore investigated in the following. The starting point of the discussion is the *active message* [90] model. Advantages and disadvantages of this approach are presented. The discussion will lead to a generalized model based on *reactive objects*.

4.4.1 Active Messages

As was mentioned in the introduction, the active-message approach allocates a user-level message handler for the processing of a communication interrupt. The model is highly preemptive. It enables a message handler (i.e., a normal user program) to either integrate the received data stream into or reply with a data stream out of the still-ongoing (i.e., interrupted) computation. The idea is to assign to program variables data values (more or less) directly received via a communication channel without intermediate buffering. Similarly, on request of a remote task, the contents of some program variable should be transferred (more or less) directly via a communication channel.

In the active-message model, data stream processing is entirely application-oriented and, depending on the implementation, may cause an indefinitely long phase during which subsequent communication interrupts are ignored and possibly lost. This is because the user-level message handler proceeds without interrupt and until completion, which generally raises problems in maintaining system integrity.

[17] For example, on a single 50-MIPS i860, the communication latency for a 64-byte packet is about 23 μs and for an 8-byte segment, 32 μs.

Even if safety is not of primary importance for a given system configuration, the model is quite hardware-dependent. In order to work efficiently and without buffering, active messages assume user-level network access to read data from some input FIFO register store (i.e., directly from the wire). A DMA-style data transfer would not provide the proper basis, although the model is implementable even in such an environment. In that case, the DMA would have to perform on a kernel-level buffer, which is then to be consumed by the user-level message handler. Thus, achieving the original goal, namely to avoid buffering, is not primarily a function of active messages but of the network hardware interface.[18]

Provided that the underlying processor does not support active messages, kernel-level software is needed to invoke the user-level message handler each time a communication interrupt occurs (Figure 4.9). This software must perform an upcall that is assigned to cross protection domains. The interrupt causes the currently executing (computation) thread to transfer from user to kernel space, and the upcall has to negate exactly this "side effect." A kernel-level interrupt handler is invoked, the sole function of which is to transfer control back to a user-level procedure.

In order to execute the upcall, a user stack frame must be created. This stack frame stores the nonvolatile processor state of the user-level message handler. More specifically, the stacked information simulates a downcall executed by the message handler. The stack frame is needed to set up the return path for the kernel-level interrupt handler and to start the user-level message handler. In fact, the kernel-level interrupt handler fakes a kernel call issued by the user-level message handler. The upcall is then executed by directing the processor to pop the newly created stack frame and "continue" user-level processing (i.e., begin executing the message handler). One of the first instructions performed

[18] A message handler is defined as `void handler(void* buffer, int length)`. This interface specification assumes that incoming messages have already been buffered in memory before the handler is invoked. By the way, it is also a question of the application program whether active messages can be exploited for speedup purposes. It provides a very primitive (yet powerful) programming paradigm that cannot be assumed to meet the needs of any application. Therefore, an important issue is reliance on parallel-language compilers that generate active-message code.

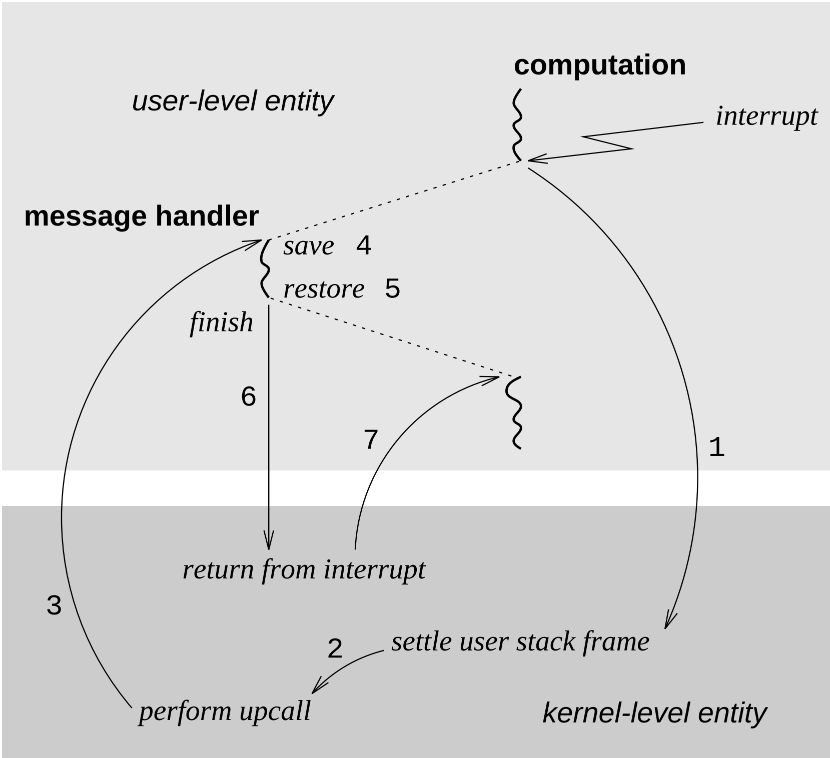

Figure 4.9: Flow of control in the active-message model

by the message handler is to save the nonvolatile register set. Then, either the incoming/received message is fed into the ongoing (i.e., interrupted) computation or a reply is sent back. Finally, the nonvolatile register set is restored. The message handler finishes processing by logically executing a return from interrupt.

All these steps (4–6 in Figure 4.9), except for message handling, will usually be carried out by a user-level *wrapper routine* (see also Section 4.1.1) assigned to invoke the problem-oriented message handler. This generally improves portability of the message-handling code. The start address of the wrapper routine is the destination of the upcall—and the target of an active message.

Whether the return from interrupt is a privileged instruction and, in that case, needs to be executed in supervisor mode (i.e., by the kernel) is highly dependent on the CPU. In the privileged case, this requires the wrapper routine to trap into the kernel when the message handler has finished. The stack frame generated by the trap is then replaced by the stack frame that was originally generated by the communication interrupt. To achieve this, a copy of the original stack frame must have been made. This copy must remain valid until the user-level message handler traps by means of the wrapper routine. Thus, if a stack cannot be used to save the original stack frame, further communication interrupts will have to be temporarily disabled as long as message handling is in progress. The duration of this phase then depends on the run-time behavior of a procedure (namely the message handler) of a user program.

In the nonprivileged case,[19] user stack frame setup is often carried out by extending the user-mode stack of the interrupted (computation) thread. The user stack frame generated by the interrupt is copied from the kernel to the user-mode stack, and then the original stack frame is changed to refer to the message handler (i.e., wrapper routine). This may involve MMU programming, page faults and the potential loss of interrupts because of address-space segment violations.[20] After return from kernel-level interrupt handling (which is the upcall), the original stack frame is still saved on top of the user-mode stack. A return from user-level interrupt handling then causes the pop of the processor state that was saved due to the communication interrupt. Restoring this processor state means continuing with normal application processing.

[19]This excludes single-address-space configurations. In these configurations, applications may be processed in supervisor mode only and are thus implicitly enabled to execute a privileged return-from-interrupt instruction.

[20]Interrupts are lost if the user stack segment cannot be extended. Stack extension may fail not because there is no more memory but due to an adjacent (code/data) segment of the same logical address space. Stack extension may not only fail because of memory shortage but also because of a user stack pointer addressing unavailable memory regions. Note that in such a situation the functioning of the nucleus, particularly the communication system, depends on the correctness of user programs. This defines a questionable and extremely dangerous "uses relation" [67].

4.4.2 Epilogue Processing

A critical issue in the active-message model is when to reenable communication interrupts. The upcall to invoke (and return from) the user-level message handler introduces additional overhead. If the message handler behaves like an ordinary interrupt handler, this overhead will lengthen the duration during which communication interrupts are disabled. Typically, interrupt handlers are executed on the priority level that corresponds to the interrupt being served. This is primarily to avoid interrupting the same handler by the same type of external event and thus prevent the potential overrun of the kernel stack (of the interrupted thread). The longer interrupt handling takes, the higher the probability that the same event occurs interlocked. A user-level (message) interrupt handler increases this problem, as interrupt processing becomes completely application-oriented. The advantage of active messages is that incoming data can be processed in a manner demanded by the ongoing (and interrupted) computation. However, the disadvantage is that this scenario raises significant problems concerning the delay during which further communication interrupts are to be temporarily disabled.

An alleviation of this problem is to understand the interrupt as a signal that announces the arrival of a (possible) sequence of messages. This signal is raised when the first message comes in. In that scenario, the overhead for a single (user-level) communication interrupt is (logically) dispersed over a number of messages. Before the message handler terminates, a check for additional (i.e., pending) messages takes place. The communication interrupt will then be interpreted as a signal to poll for further messages. In addition, the message handler executes with interrupts enabled (i.e., not on the priority level corresponding to the communication interrupt), and the communication device is temporarily disabled from delivering interrupts. The device is enabled when the message handler really terminates.

This approach is applicable for both FIFO and DMA-based network hardware interfaces, with the former being more suitable than the latter. However, even in the DMA case, there is no need to let the message handler proceed without first lowering the priority level established by the communication interrupt. The interrupt signals that DMA has finished and a message packet was buffered. Serving the network input channel then means to make another DMA descriptor available, unless the hardware is capable of processing chains of DMA descriptor lists. The input channel itself need not be read in order to receive data and thus enable the delivery of follow-up messages. That is, in order to provide an active-message model, a kernel-level interrupt handler will serve the (DMA) interrupt, lessen the priority level and perform the upcall to invoke the user-level message handler.

This is almost the same pattern as followed by a prologue; namely, to switch over to its epilogue in order to continue processing in a manner similar to user-level threads. An epilogue always executes with interrupts enabled. The epilogue will never preempt the nucleus, but it may preempt a user program. As was explained in Section 4.1, if the prologue interrupts a thread that currently is not in charge of executing a nucleus function, epilogue activation is straightforward and happens without significant delay. With the epilogue solely performing a user-level upcall, a message handler may be started, which is able to proceed asynchronous to the ongoing computation. In this approach, the active-message model is only approximated, since it is not the prologue (i.e., the interrupt handler itself) that acts as the lift.

The upcall mechanism takes the form of a wrapper routine that takes care of the invocation of the message handler. In fact, the most appropriate solution would be to invoke the handler as (ordinary) "subroutine." A return from this subroutine then automatically reactivates the kernel, which takes charge of the activities required to reenable the communication device to deliver interrupts.

4.4.3 Thread Model

The active-message model implies that a message handler either delivers or extracts data concurrent with the execution of some thread. Hence, this *computation thread* is not required to explicitly issue communication requests to the nucleus in order to receive data. Data will be automatically placed into the program variables. In this scenario, a message handler is nothing but a *communication thread* the sole function of which is to keep the computation thread running.

A *pipeline-based* mode of operation between the two threads must be anticipated. Active messages are of little benefit when the computation thread has to wait on data produced by the message handler (i.e., communication thread). In particular, the model will lose all its advantages if waiting means instructing the scheduler to execute a different thread (or task) and thus utilizing the idle time to start another computation thread. In this case, kernel-level critical sections need to be activated and therefore synchronized. Performance degradation will be the consequence, and this is what active messages are trying to avoid. Thus, under normal circumstances, one can assume that the epilogue, and thus the communication thread, are immediately started without any synchronization measure.

With the distinction between the two types of threads, further consequences arise. First, it is a good choice to give the message handler its own execution context; namely, to assign it to a real thread (i.e., to associate it with an *active object*). Second, this thread only gets activated when a communication interrupt occurs. The initial action of the communication thread is to request from the nucleus exclusion from normal scheduling and, instead, enable interrupt-driven activation. Communication thread processing, then, is involved with the execution of the message handler and accepting the next communication interrupt.

Exclusion from normal scheduling basically means that thread activation is possible without worrying about nucleus-level critical sections. In the case of the nucleus, neither the dispatcher nor the scheduler will intervene, which would require ensuring the integrity of the state maintained by these two components. The thread simply enters the nucleus in order to accept a communication interrupt and, if signaled, is immediately resumed. The thread will be synchronized to hardware and not software components. This approach provides an autonomous mechanism by which a message handler becomes able to poll on communication interrupts.

4.4.4 Reactive Objects

Because threads are used to implement active objects, the kind of thread activation discussed so far is based on so-called *reactive objects*. A reactive object provides a runtime context for a user-level interrupt handler (Figure 4.10). In the particular case discussed here, the reactive object encapsulates the message handler—it acts as a "wrapper routine" or, better, "wrapper object."

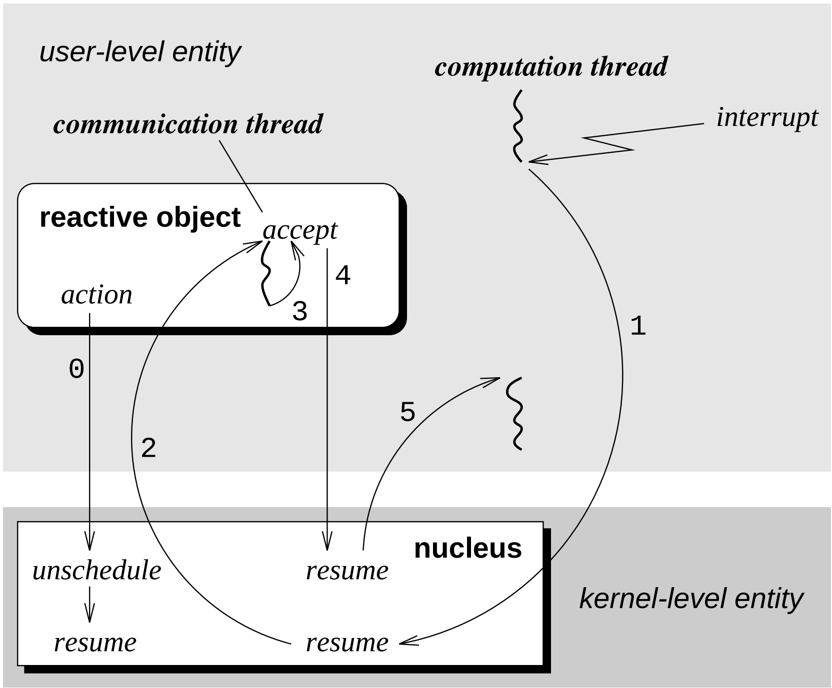

Figure 4.10: Flow of control using reactive objects

Resumption of a reactive object is accomplished by simply activating the nonvolatile register set of the associated thread. No other state information is saved or need be restored. The thread implementing the reactive object returns from the nucleus and executes the message handler. A stack frame need not be set up explicitly in order to perform the "upcall." Rather, the stack frame is automatically established whenever the thread accepts (i.e., synchronizes on) the next communication interrupt. Having called the nucleus and synchronized on an interrupt, the stack stores activation records that are keeping track of the proper return path. The nucleus call is then nothing but a request to poll for interrupts; and the thread (i.e., reactive object) provides a runtime context that enables the message handler to poll for communication interrupts.

Thus, the upcall to the message handler using reactive objects can execute faster than in the active-message model. In both approaches, the runtime environment of the message handler is responsible for preparing the task for user-level interrupt handling. However, the reactive object approach does not require the dynamic creation of a runtime context (i.e., stack frame) for the message handler. In this model, the kernel-level interrupt handler merely has to activate an already existing context (i.e., thread).

4.4.5 System Integrity

The fundamental technical problems of active messages and their approximation by means of reactive objects are when to allow for subsequent communication interrupts to occur and how to provide access to the network device. Both may limit applicability, since they raise severe difficulties in maintaining *system integrity*. In the reactive-object model discussed before, the first aspect is not really a problem. Reactive objects are executed in the epilogue phase, and an epilogue is always processed with interrupts enabled.

The problems disappear in a single-tasking/single-threading mode of operation. In that case, the node resources need not be multiplexed between several tasks and/or threads. This makes user-level interrupt handling and network access straightforward, even if a distinction between user- and supervisor-mode processing is made. If anything, the *interrupt latency* will impair only the task (or thread) for which a communication interrupt was signaled. Unless the sender applied a wrong communication endpoint identifier, the task executing on the node must be the receiver for the incoming message.[21] In that case, the communication endpoint identifier also stands for a node address. Network access becomes trivial, since this mode of operation promotes a single address space. Protection boundaries need not be bridged to transfer

[21] More precisely, with a single-thread model it is assumed that neither a system thread nor a system procedure will act as the message-receiving instance. In this configuration, after having been bootstrapped with some application, the node only receives user messages and consequently cannot be remotely controlled.

data between the user-level message handler and the network device. In addition, the network device is only bound to a single task. Thus, the message handler can seize that device since there is no other competitor.

If multithreading is supported by the system, active messages (i.e., reactive objects) may introduce the need for demultiplexing received messages. For example, kernel threads that are set up to receive system call requests on an ROI basis, for example, must not be obstructed by user-level message handlers (i.e., an application program). The problem is how to prevent a user-level program from reading kernel-level messages. This is solved only by either the network interface distinguishing between user and system channels for communication or the (kernel-level) network device driver always being in charge of message transportation. The former calls for hardware, and the latter for software, support. Hardware support means granting user-level access on user channels only. The nucleus then uses the system channel to carry out "normal" communication requests.

Software support for both active messages and reactive objects in a multithreading environment means to basically implement the network device as an *abstract data type*. The abstract data type provides to the application (i.e., message handler) only two sets of primitives, which will have the following semantic:

- The *input operations* (i.e., primitives to receive messages from the network) must guarantee that the user-level message handler cannot read beyond the message and thus discard system-relevant data. This also calls for two sorts of output primitives, distinguishing between user and system messages. In order to make demultiplexing feasible, each of these primitives must add channel information to the message.

- The *output operations* (i.e., primitives to send messages to the network) must use the active-message model as well. These primitives then have to address remote kernel-level message handlers that will take care of the processing of system-relevant data.

Exploiting active messages in the nucleus implementation assumes that each node provides a unique kernel-level entry point for all other nodes of the parallel machine. However, since in a family-based design not every node is equipped with the same kernel implementation, this entry point should not be a code address. Code addresses of the various message handlers may vary. All message handlers, and not only those of the kernel, are addressed indirectly by indexing via some "virtual interrupt vector." The index number may then serve as a logical remote handler address.

Another technique is to use a standard message-passing mechanism in order to make message handler code addresses globally known. This can be readily achieved by exploiting, for instance, a name service. A task that wants to issue an active-message queries the name service to retrieve the code address. Searching the name space for a given message handler address can typically be done as part of the initialization procedure of a task. In particular, the initialization sequence of a reactive object may export its address into the global name space. The binding phase to this object is then used to import that address from the name space. All these functions can be automatically performed by the runtime system and need not be explicitly called by the application program. Since one can assume that both the producer and consumer tasks are somewhat closely related to each other, the consumer task may also initially pass a reactive object reference to the producer. By means of this reference, the producer becomes able to send an active message and resume the message handler encapsulated by the addressed object. A reactive object thus becomes an instance variable of the program executed by the consumer task. By exporting this object, the consumer task grants access to its address space.

In all these cases, a further consequence arises: every message must be part of the PDU that triggers the activation of a reactive object. Thus, a nucleus-specific communication protocol must be executed. This protocol corresponds to the semantics of the input operations mentioned above. It guarantees that an input operation always knows about message boundaries.

4.5 Summary

This chapter explained the operating principle of the nucleus. Interrupt synchronization relies on the serialization of asynchronous activities (i.e., activity objects). A distinction between the *prologue* and *epilogue* of interrupt handling was made. The prologue implements the asynchronous phase of an interrupt handler. In contrast, the epilogue implements the synchronous phase and is represented by an activity object. Interrupt handling begins with the prologue and may be continued by starting the corresponding epilogue. An epilogue will never be interrupted by epilogues, but may be interrupted by prologues. This ensures nucleus integrity, since critical sections are implemented as epilogues.

Abstract protocol processors implement the nucleus communication system. The interactions between the protocol layers have been optimized to meet the needs of nucleus abstractions for systemwide inter-process communication. These abstractions support a packet- and segment-oriented data transfer. Packet transport can be accomplished by executing a single network transaction, while segment transport (i.e., the transfer of arbitrarily sized messages) assumes a two- or three-phase communication protocol.

Active messages are approximated, leading to shorter delays during which interrupts are physically disabled. In addition, the proposed thread-based model establishes a runtime context for the interrupt-driven user-level message handler. This context need not be set up at the time the interrupt occurs. The thread model introduces "*reactive objects*" that are immediately activated on behalf of an interrupt and used to encapsulate the user-level message handler.

Chapter 5

Transient Objects

Distributed-memory parallel systems consisting of several thousands of autonomous processing nodes are reality. In such massively parallel systems, bootstrapping an operating system becomes a serious problem. For obvious reasons, centralized approaches to getting a parallel operating system started are not feasible. These approaches result in bottlenecks and also raise difficulties in maintaining at least a minimal degree of fault tolerance. Bootstrapping time must be in a range that does not hamper the work of system administration staff. Last but not least, parallel machines are meant to process user applications. They should never waste computing power, as will particularly be the case with a time-consuming system initialization procedure. The delays between power-up and system operation may be in the order of several tens of minutes and can easily last hours, even in a small-scale (shared-memory) parallel system [13]. Undoubtedly, a parallel operating system implementing such a time-consuming initialization procedure is not scalable; scalability, however, is one of the most important aspects in an environment consisting of thousands of nodes.

The question of whether bootstrapping the entire multinode machine before the first application is started is really mandatory arises. A more promising technique seems to be *incremental loading* of the operating system and *incremental bootstrapping* of the parallel machine. In this context, operating system services become *transient objects*. These objects are installed on demand (i.e., when requested for the first time) and removed when no longer used. On this basis, nodes of the parallel machine are actually put into operation when required by the application and not by the operating system. This significantly reduces initial system startup cost in terms of the time it takes to get the machine running for a particular application. The burden of bootstrapping all nodes, in fact, is split among the applications to be run. An exception to this are applications seizing all nodes. However, even in that case, node bootstrapping may also take place in parallel with the execution of user tasks.

In the following, the implementation of transient objects is discussed. The means of making incremental loading transparent to the application tasks is described. The focus lies on the interactions that must take place between the ROI layer (see also Section 2.3) and the naming subsystem to signal when to install a transient object. For this purpose, an in-depth discussion of how to locate objects in a parallel/distributed environment is provided. These considerations present a detailed insight into the internal organization of POSE, the parallel operating system extension layered on top of the kernel. They demonstrate the means by which a dynamically alterable operating system building block can be established. Refer to Section 2.2.1 for a description of POSE and the other two fundamental building blocks, the nucleus and kernel.

5.1 Extend-On-Use Property

All sorts of system services provided by the POSE, nucleus and kernel are encapsulated by one or more objects. The logical design assumes that only the nucleus provides services by means of *passive objects*. POSE and the kernel are logically built with *active objects*; however, both may exhibit an actual structure in which passive objects are used for system composition as well. These different configuration aspects were discussed in Chapter 3.

The approach to adapting the operating system to different applications, that is, to establish dynamically alterable system structures, relies on an active object model. This is motivated by the following facts:

- Introducing a new system service can basically be reduced to the problem of dynamically creating processes and integrating them into an existing system structure. Releasing the service works in the opposite direction and is based on the capability to destroy processes.

- Sophisticated memory management schemes are not required, as it would be the case when attempting to dynamically introduce a passive object and restructure an existing address space. These schemes would have to be added to the minimal subset of system functions.
- Relying on active objects (i.e., a process model) comes closer to the needs of distributed-memory machines. As far as possible, system services must be off-loaded from the nodes assigned to application processing. In this case, message passing needs to be exploited for service invocation. This means that there must be some process that is in charge of service execution on behalf of the requesting process.

Thus, employing the active object model to implement a dynamically alterable operating-system family for distributed-memory parallel machines strongly corresponds to the program family idea. In this sense, the active object model is more fundamental than the passive object model. It supports the construction of a *minimal but perfect basis* for the processing of parallel applications.

Dynamic reconfiguration by adding or replacing passive objects at runtime (i.e., on the fly) becomes subject to the nucleus. For the POSE and kernel, dynamic reconfiguration is based only on active objects. More precisely, making the nucleus dynamically reconfigurable means designing into the minimal (but perfect) basis mechanisms that will trigger the loading of a function or passive object upon its first use (i.e., invocation). This implies the introduction of "joints" at places where system dynamics of that quality could have a positive effect on the overall complex. Thus, not every passive object, but only a few qualified ones, of the nucleus will be given the extend-on-use property. Even in this case, adding or replacing nucleus-level passive objects will actually be a function of POSE. Moreover, the functioning of the dynamically reconfigurable nucleus will not depend on the presence of a POSE service implementing dynamic restructuring. Just as the nucleus plays a mediator role to possibly propagate hardware exceptions (i.e., traps and interrupts) to higher-level system components, it also plays the mediator role for propagating software exceptions. A typical software exception will occur when calling a nonresident but existent nucleus function.

The motivation behind designing (and implementing) a dynamically alterable parallel operating system is to avoid unused but resource-consuming service functions. However, the gain obtained by that strategy depends on the granularity of the objects that are subject to dynamic restructuring—and this, basically, is a design issue. This aspect will be discussed in the next subsection. Afterwards, the extend-on-use property of POSE services (i.e., active objects) is explained. The discussion focuses on incremental loading, garbage collection and exception handling as the means to extend and shrink the operating system on demand.

5.1.1 Object Granularity

System restructuring on the basis of small (fine-grain) objects is only applicable in a passive object model. When relying on the active object model in that case, the overhead incurred by requesting and carrying out a system service may become prohibitively high. Note that a multilevel hierarchy of active objects may be the outcome of the restructuring procedure, with message passing (i.e., ROI) between the objects to make progress in service execution.

Compared to the fine-grain approach, coarse-grain restructuring yields the opposite effect. It may improve system performance, but only at the cost of memory resources required to store the coarse-grain service object. Moreover, this will not actually solve the bootstrap problem, as fairly large system program entities still need to be loaded. A coarse-grain active object may be, for example, a file server or a process server. It really depends on the application whether all functions provided by the complex server need be available. On the other hand, one also may argue that on the basis of virtual memory the size of a (system) program should no longer be a problem. This is true, but also implies at least the following two drawbacks. First, the minimal basis needs to be extended with virtual memory management functions. The approach pursued, however, is to make the virtual memory management subsystem incrementally loadable as well. Second, not every CPU (or node) used in distributed-memory machines supports virtual memory. Typical examples are Transputer-based architectures. For these reasons, coarse-grain objects as typically found in microkernel-based (process-structured) distributed operating systems are not the right choice.

Some sort of *medium-grain active object* provides the appropriate basis. A brief look at a file system service serves as an example. There, one will discover main functions such as disk I/O, caching, disk block and directory management. All of these functions are complex enough to be understood as a subsystem, and they are autonomous enough to be represented by means of active objects. In terms of an operating-system family, there is one more reason for making these distinctions. The disk I/O subsystem implements the low-level device driver, and therefore represents the minimal subset of file system functions. Caching is already an application-oriented subsystem, providing different strategies for different applications (distinguishing between different file access patterns such as random or sequential access). The disk block subsystem implements the physical file system structure, for example, differentiating between a UNIX or MS-DOS organization. Finally, directory management solves the problem of naming. Logical file identifiers (i.e., names) are mapped onto physical identifiers (i.e., descriptors).

Considering the above-mentioned file service, different application domains call for different subsystem representations. The difference might be due only to a UNIX or MS-DOS disk block subsystem, while the other features are still sharable between all applications. Moreover, not every application of the same (UNIX or MS-DOS) domain will necessarily call for all of these subsystems. A database task, for example, may exploit only the caching subsystem to improve performance, while other tasks use the disk block and directory management subsystem as well. In particular, a virtual memory system can be supported by providing efficient low-level disk access by means of the caching subsystem. In addition, with subsystems being implemented as active objects, it is almost natural to exploit the parallel machine to speed up service-processing in general. This even includes having different nodes in charge of executing different file service subsystems.

The file system is only one example. Similar considerations can be made with name space management (a number of name servers), process management (kernel-level active objects providing mechanisms and POSE-level active objects providing strategies), memory management (much the same organization as process management) and so on. Medium-grain active objects as service-providing entities, in conjunction with a distributed-memory machine, will outperform any monolithic system organization. A performance increase becomes possible since service functions, such as name space, process or I/O management, consist of several (medium-grain) active objects distributed across the parallel machine. Thus, to speedup service processing, a single (formerly coarse-grain) server has been parallelized to exploit the parallelism provided by the hardware. In other words, the service is implemented by a parallel system program and obeys the principle of parallel processing, similar to a parallel user application.

5.1.2 Incremental Loading

The features provided by the nucleus, namely message passing and thread scheduling, already suffice to allow operating-system (i.e., POSE) restructuring. By exploiting these features, a service-requesting process, hereafter referred to as the *client*, is able to issue a system request to install the service-providing process, hereafter referred to as the *server*. That request is handled by a system component residing on a system node (refer to Section 3.1.2.2). This management instance may either collocate the server on the same node or allocate the server on a different (system) node.

Collocating a server on a system (or even generic) node entails *incremental loading*. Allocating an unused system node for that server basically means triggering *incremental bootstrapping*. That is, incremental loading encompasses incremental bootstrapping of a distributed-memory machine. The decision concerning which way to go depends on the type of service requested and the number of nodes that are free for restructuring measures. It is obvious that a site-dependent service such as a disk server must be situated on the same node as the hardware resource to be controlled. This may cause collocating the new server with one or more servers already existing on that node.

In either case, the client is in charge of issuing the system restructuring request. Consequently, it is the application task that causes incremental loading of an operating-system service. An *application-oriented system structure* emerges because only required service functions are present.

5.1.3 Garbage Collection

Introducing operating-system services on demand is only one side of the story. These services are also required to be automatically removed from the system when no longer needed. A *garbage collection* problem arises, namely, deciding when to really delete an active object implementing an unused service.

One approach is to have the client be in charge of *initiating withdrawal* of the service function. Just as the client issues a request to install the service, a request for service deletion may be sent out. The server responsible for dynamic restructuring simply has to look up a registry to find out where the service resides, of how many active objects the service consists, and whether the service was acquired and not yet released by other clients.[1] If removal is possible, that server initiates service deletion accordingly.

Unfortunately, this approach is not always applicable. The problem is clients that have run out of control. It is very unlikely that a "crashed" task (i.e., node) will be able to ask for service release. In that case, a *third party* is required to take over the part that was originally assigned to the client. The third party then has to find defective tasks (i.e., clients). It will typically query the registry for tasks, actually acting as a service initiator. The processing state of these tasks is then determined by querying the kernels via ROI. In the case of a reply that indicates a functional disturbance, the third party releases the service on behalf of the task (i.e., client).

[1] The registry needed for implementing service withdrawal is more than a name service used to associate actives objects with logical names and physical addresses. It takes the form of a specialization of, or minimal extension to, the name service. Beyond that, the registry need not be persistent. Persistence is a feature introduced as a minimal extension as well.

A combination of both strategies is meaningful. The normal situation is that clients will announce the withdrawal of an unused service function. This may happen any time, the latest as part of the termination procedure of the client. Viewing the service (better, service access point) as an "instance variable" of the application program allows one to draw an analogy to block-structured programming languages. The service access point—and so the service itself—is associated with a specific scope. It is only visible within that scope. Leaving the scope implicitly results in the destruction of all its instance variables. Instance variables are objects. Hence, the modeling of operating-system services by means of objects strongly assists in maintaining a dynamically alterable structure, provided that proper system programming language support is given. The use of a service is then expressed by the declaration (i.e., construction) of an object. Service activation means applying methods on this object, with each of these methods representing a service function. Finally, the implicit release of a service is triggered when the object is destructed. Under normal circumstances, all these steps are performed under the control of the service user (i.e., the client or user process).

A meaningful strategy is to provide automatic garbage collection on a per-service basis. This helps to solve several problems. First, the third party (i.e., garbage collector) is running only when the corresponding service is in use or has been (incrementally) loaded. There is no need for a separate mechanism to activate the third-party component. Second, the registry is naturally implemented in a distributed and service-oriented manner. Different services may provide different information and strategies for determining their in-use states. Third, problem-oriented strategies for polling or signaling the clients may be introduced. Fourth, service release becomes an active procedure of the server itself. This helps to avoid synchronization problems that might arise when the service release is initiated asynchronously from "outside."

A standard (i.e., default) implementation can easily be provided by the ROI layer. Assuming that a server automatically receives the client identification with every incoming service request, it is a simple procedure to become aware of the clients that have used a specific service function. In particular, if the ROI layer implements a connection-oriented client-server model, client registration takes place only during the connection establishment phase and not with every request. If required, a specialization of the default implementation provided by the ROI layer is then possible on a per-server basis, employing service-dependent state information.

5.1.4 Exception Handling

The previous subsection discussed a "third-party garbage collection facility" to announce the withdrawal of a no-longer-used POSE service. Even if only a temporary disability of the client task (or its node) is discovered, service release in that case may still be practicable. However, when the task continues normal processing, it may get into trouble due to a previously existent but presently nonexistent server. As discussed in the following, this problem can be dealt with on an exception-handling basis.

A service function is requested by means of ROI, exploiting the nucleus. The message-passing primitive used for this purpose (by the ROI layer) has then to provide an actual parameter that denotes the communication endpoint address of the server of the service function. If service release took place, that server may no longer exist. The nucleus will detect this due to a *hash fault* when attempting to lookup the communication endpoint address and deliver an error code to the client. This error code then causes the client site to raise an ROI exception indicating that the requested service (function) is not yet present. In the course of handling this exception, incremental loading is used to enforce service availability.

An orthogonal approach, thus, is to always trigger incremental loading as a result of ROI exceptions. In the initial case, that is, when the service is requested for the first time, an exception may be raised due to some *naming fault*. As we will discuss later on, an active object implementing some service function is bound to a global name. The name is related to a communication endpoint address and registered in a global name space. Before initial service invocation is possible, the name space needs to be searched for the name of the active object. If found, the communication endpoint address of the object is retrieved and ROI to the appropriate server may take place. If not found, incremental loading is requested. Provided that loading was successful, the ROI layer performs dynamic binding between client and server site and then instructs the server to carry out the requested service function.

Incremental loading is initiated due to two ROI exceptions, namely, a *naming exception* and a *communication exception*. The former covers initial server faults, the latter server faults due to garbage collection. Note that handling the communication exception in the manner described above also copes with "stale" communication endpoint addresses that may come into existence with server migration and a possible inability of the kernel to maintain forwarding addresses. A kernel may provide services to establish *forwarding identifiers* and to set up chains of these identifiers.[2] Thus, the illusion of a static system is presented. However, in order to achieve this illusion the kernel must acquire memory resources to store forwarding addresses. If the kernel runs out of memory, access transparency in the case of process migration can no longer be guaranteed.

5.2 Dynamically Alterable System

It was mentioned above that incremental loading is actually initiated by the client who wants to get access to some server. Usually, this client represents an application task invoking a POSE service function. Incremental loading is transparent to the application proper. A POSE library module (i.e., a passive object) residing within the address space of that task effectively takes care of all the steps necessary to carry out and/or trigger dynamic system restructuring. These restructuring measures are invisible to the application programmer.

This passive object implements the *service access point* of the requested service function. In the following, the functioning of a service access point is discussed, and the interactions with other POSE components needed to have a service incrementally loaded are described.

[2] Forwarding identifiers are stale communication endpoint addresses. They are locally associated with forwarding addresses specifying the new communication endpoint address of a (migrated) thread. If a thread migrates several times, forwarding addresses may become forwarding identifiers, thus establishing a forwarding chain.

5.2.1 Service Access Point

Dynamic reconfiguration applies to active objects of the POSE and kernel complex, and to passive objects or functions of the nucleus complex. In both cases, the service access point is the source of eliciting incremental loading of the requested service function. The following subsections describe the organizational aspects of the two main sorts of service access points, interfacing to the nucleus and to the POSE or kernel.

5.2.1.1 Passive Object Reconfiguration

Dynamic nucleus reconfiguration starts from the inter-domain call approach (refer to Section 2.2.4) for invoking nucleus service functions. The inter-domain call implementation of a nucleus service access point has four different variants (Figure 5.1), whereby two variants use a *trap* for switching from user to supervisor mode, and thus from user to kernel address space. Both trap variants conform to the contemporary system call mechanism used for requesting operating-system services. The other two variants only logically separate the nucleus and kernel from higher-level entities, without distinguishing between the user and supervisor modes. They support a single-address-space configuration in which many autonomously generated application/POSE programs will be able to share the single nucleus/kernel program.

In the single-address-space configuration, the basic distinction made is to whether an inter-domain call supports dynamic nucleus restructuring or a static approach is pursued. The static nucleus relies on a simple *jump table* the entries of which are the addresses of all global accessible nucleus functions (Figure 5.1, case a). This jump table may be employed directly by the *user* program or by the VICE *trap handler*, depending on which service access point representation is preferred. The function code serves as the jump table index to locate the procedure body implementing the requested service function. The alternative (Figure 5.1, case b) to the static approach is to use a so-called *joint* object for connecting programs to the nucleus, supporting on-demand nucleus (re-)configuration. As with the jump table (case a), the joint object may be employed directly by the *user* program or by the VICE trap handler. In the non-trap-handling case, both the joint object and the jump table reside at a unique location in the (physical/virtual) address space shared by all the programs.

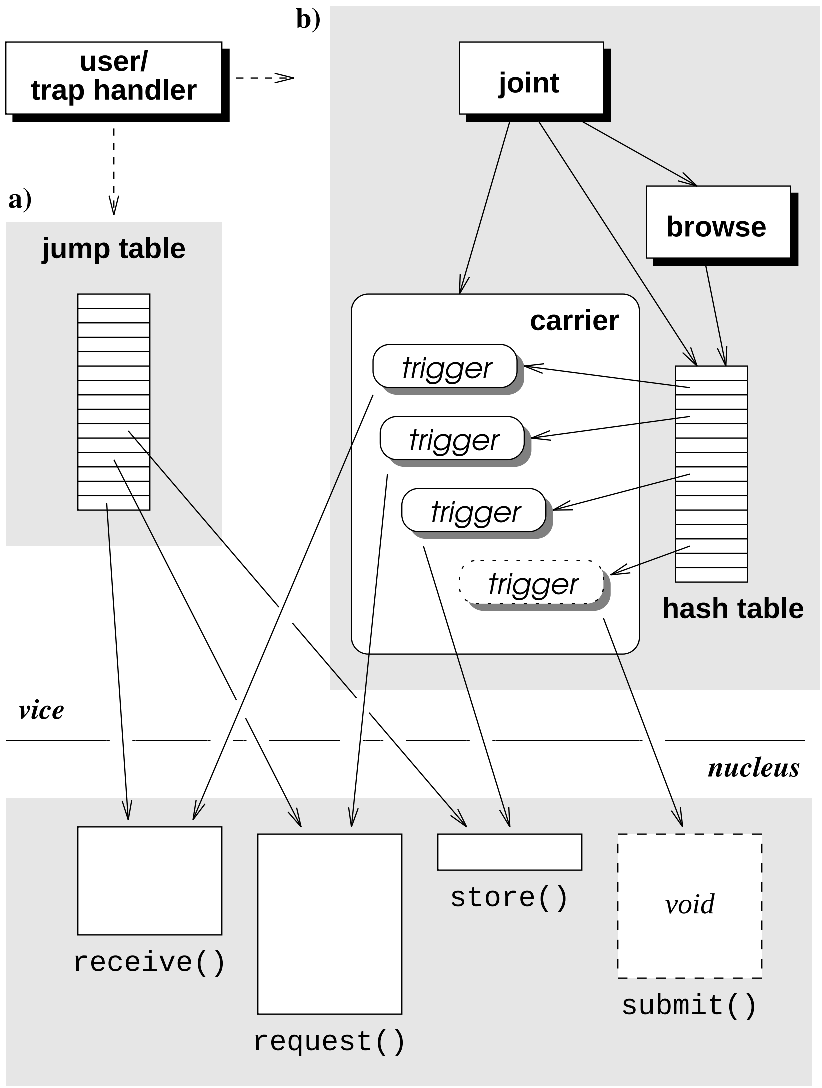

Figure 5.1: Inter-domain call variants

As illustrated in Figure 5.1, a joint call attempts to first look up a *trigger object* from a *hash table*. In the dynamic approach, every nucleus service function has a trigger object. A trigger object knows the signature of its associated nucleus function and is responsible for (input/output) parameter passing between disjoint address spaces—a trigger object is the "*carrier*" of a nucleus function call. The *browse* instance is instructed to allocate a trigger object and initiate the instantiation of the corresponding nucleus function if the joint instance encountered a hash table miss. Roughly speaking, the joint takes care of the presence of the requested nucleus function by initiating loading when an access fault is recognized. It does not perform the loading itself, but rather raises an exception that may or may not be handled by some POSE entity. An error will be returned to the thread that requested the nonexistent nucleus function if loading fails.

Although the jump table introduces an indirection step in order to invoke a nucleus function, this alone is not sufficient to make the nucleus configurable in an application-oriented manner. First, the jump table must be initialized with the addresses of all exported nucleus functions. In this case, the linker yields a nucleus representation consisting of all possible but not the few necessary service functions—note that not every application necessarily uses the same set of nucleus functions. Second, the jump table is of limited size, and thus defines an upper bound on the maximal number of exported nucleus functions. The second aspect is crucial even if the jump table stores the addresses of routines triggering the incremental loading of nucleus functions. Both problems do not exist when taking the joint variant and exploiting a hash table of trigger objects.

5.2.1.2 Active Object Reconfiguration

An application[3] always gains access to POSE by means of LOI. The LOI scheme is used to activate the service access point related to the service requested by the application task. This access point then forwards the service request to a server capable of providing the function requested by the client. Forwarding the service request logically means performing

[3] If not otherwise specified, "application" does not only cover user programs. System programs are included as well.

ROI. Figure 5.2 gives a rough idea of the function of a service access point.

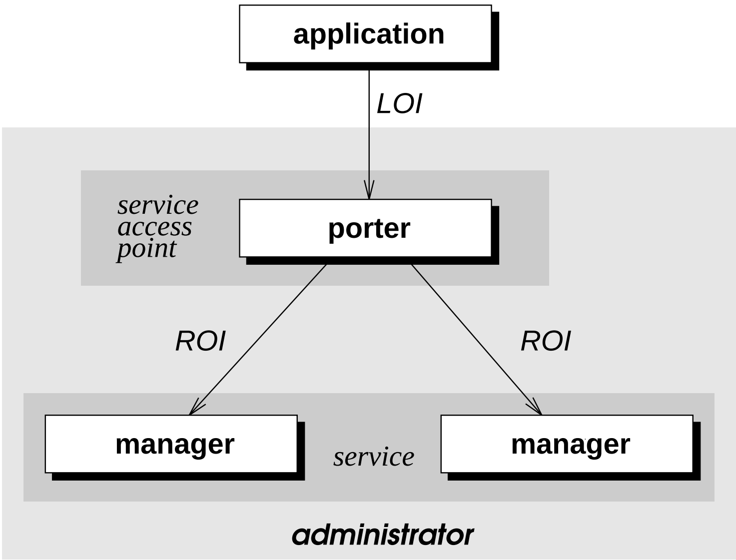

Figure 5.2: Interaction with replicated service functions

The service access point is logically represented by a *porter* module. As the name suggests, the porter knows what server to dispatch for a service request. Basically, the porter acts as a service-dependent signpost. In its simplest form, the porter is nothing but the ROI *client stub*, in which case the ROI layer, by utilizing the name service, is exclusively in charge of server selection. In a more sophisticated form, the porter may decode actual parameters of the requested service function to make a decision about which server to contact. For example, a UNIX file system porter may interpret the path name argument of the `open()` request as a (structured) name that is going to be resolved by means of a general naming facility. The porter extracts the path name and submits it to some name server, requesting delivery of a "shorthand file identifier." This identifier may then be used to direct the ROI to the file server managing the file addressed by the path name—the identifier could even be the systemwide unique communication endpoint address of a thread, thus modeling a file as an active object.

A service function is carried out by a *manager* object. Once the porter knows which manager is to be used, the service request is finally issued on an ROI basis. All the activities performed by the porter take place under control of the currently active thread and are executed within the address space of the client application task encapsulating the thread. Thus, on behalf of the porter, and since ROI implies a synchronous transfer between separate flows of control, the thread gets implicitly blocked when contacting the manager instance. The thread is unblocked when the manager instance terminates the ROI request by returning results or whatever; and application-level thread processing continues when the porter returns.

A porter and several managers jointly constitute an *administrator*. The administrator stands for a distributed service. Again, the file system is a good example. A parallel machine that exhibits several disk nodes may be configured with one file manager for each of these nodes. The file administrator is then physically distributed over a number of disk nodes and takes care of the management of files stored in all attached disks. The file porter then decides which file manager to use, and thus which disk node to employ. There are as many file managers as disk nodes, and there are as many file porters as applications requesting file access.

Administrators are used to construct the POSE and kernel building blocks. The porter is implemented by an ensemble of POSE/kernel library functions. For example, compared with contemporary UNIX installations, a porter is the logical equivalent of the system call functions contained in `libc.a`. A manager is implemented by an ensemble of POSE/kernel service functions, which are made accessible through corresponding porter-level library functions. Reflecting the functional hierarchy discussed in Section 2.3, a manager does not necessarily compare with a server as known from microkernel-based systems. In Figure 5.2, ROI denotes the logical means by which the porter and manager interact. Whether the logical view matches the actual system structure is a matter of the concrete system configuration. For example, in a single-user/single-tasking system, ROI is technically replaced by LOI. The possibility of replacing ROI by LOI does not change the hierarchical relationship between porter and manager.[4] Rather, under certain circumstances, it allows one to change the performance characteristics of a service access point.

Designing the service access point as a configurable component effects the interface specification of a service function. In order not to introduce a potential performance bottleneck, the interface must be "ROI-capable." This is the most restrictive requirement—and paves the way for lightweight or even featherweight interaction patterns. As discussed in Section 2.3, the lightweight case means NOI,[5] and the featherweight case means LOI. ROI capability of a service interface calls for a design guaranteeing minimal overhead when crossing protection boundaries. Overhead minimization of parameter passing and control transfer in a networked environment is the primary goal. Obviously, such an interface can be represented on an LOI basis without performance loss but rather an increase in performance. In contrast, relying on LOI (i.e., single-address-space contexts) implies the risk of designing an overhead-prone service interface when a configuration exhibiting some sort of physical protection boundary (e.g., isolated address spaces or networked nodes) must be supported by the system.

5.2.2 Third-Party Configuration

Usually, the POSE entity responsible for incremental loading resides on some system node. This entity is technically represented by one or more active objects. That is, incremental loading will be initiated on behalf of a service-related porter module by issuing ROI requests to some manager. This manager acts as a *plumber* and tries to install the requested service entity (Figure 5.3).

The plumber provides a "third-party connection" facility. It accepts installation requests, loads the desired service and integrates the corresponding server object into the existing system structure. Actually, server integration proves to be the most difficult function and is primar-

---

[4]This aspect is the reason why Figure 5.2 introduced the notion "manager" instead of "server." A manager need not be implemented as an active object. A server, however, is often used as a synonym for a service-providing process (i.e., active object).

[5]That is, object-bound *lightweight remote procedure call* (LRPC).

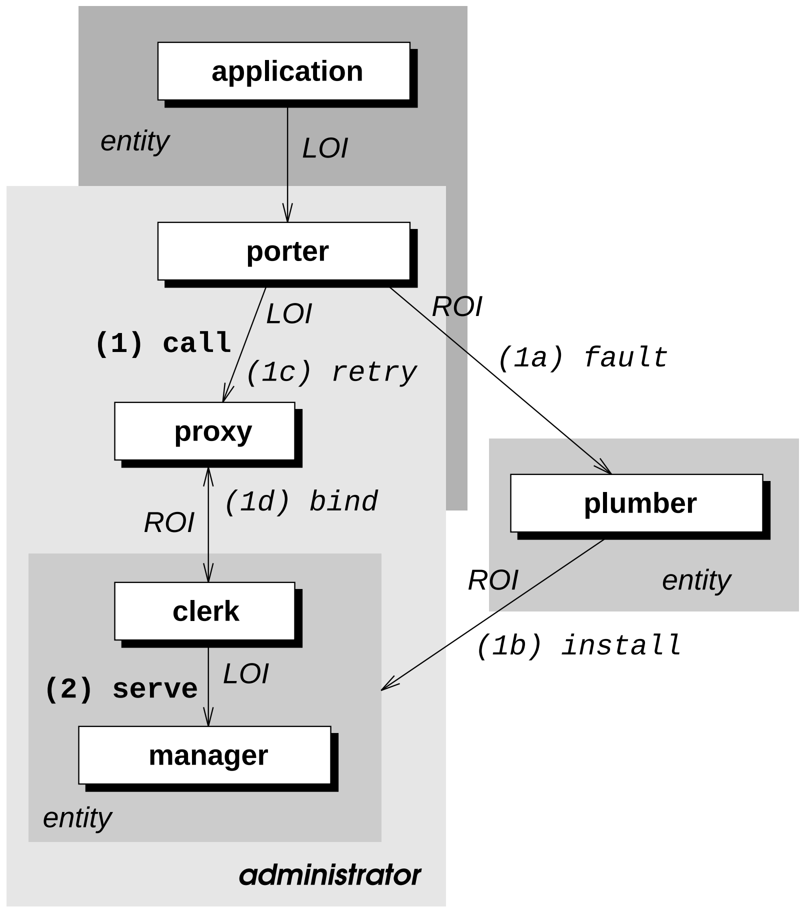

Figure 5.3: Configuration by means of a third party

ily related to "editing" the global name space. The steps that take place in order to dynamically load service entities, along with ROI requests, are as follows (refer also to Figure 5.3):

1. The porter issues ROI requests, each one corresponding to service functions called by the application. Each request is passed over to the ROI client stub that acts as local *proxy* of the service. The proxy always exists, implementing an abstraction for accessing a certain manager instance. A proxy *call* results in an ROI message

to the server. The server provides a thread of control to accept and execute received ROI requests. This thread of control acts as service *clerk* and shares the server address space.[6]

(a) The porter detects either a naming or communication exception. Handling the exception means signaling an entity *fault* to the plumber. This is done on an ROI basis, too.

(b) Having received the entity fault message, the plumber will *install* the nonexistent service entity and deliver the result of the installation procedure back to the porter. Note that the thread of control of the application task which requested the service function is implicitly blocked until the installation procedure completes (either normally or exceptionally).

(c) If the installation succeeds, the porter will *retry* the original proxy call. If the installation fails, an exception is raised, which usually causes the termination of the application task.

(d) A retry requests the proxy to *bind* with the clerk. The result returned by the plumber may already have delivered the communication endpoint address of the per-server thread representing the clerk. Otherwise, the proxy asks for name resolution by calling the name service.

2. Having received a service request, the clerk invokes the corresponding manager by means of LOI. In fact, the clerk will *serve* the ROI request and instruct the manager to carry out the encoded service function.

Basically, the proxy and clerk represent nothing but the client and server stub, respectively, which are assigned to handle function requests on a *remote procedure call* (RPC) [60] basis. Since service functions are represented by means of (active) objects, some sort of object-based RPC must take place—and this actually means ROI. Thus, there is

[6]The ROI subsystem of PEACE supports the modeling of co-resident server instances. That is, the proxy and its clerk may share the same address space, yet cooperate on an ROI (message-passing) basis. Employing LOI in this particular configuration will be the decision of the porter instance.

nothing special concerning the proxy and clerk, except the object-oriented view of a POSE interface. In particular, the normal situation is always that the proxy and clerk immediately execute the RPC protocol once the porter issues the ROI request. Service installation is considered an exceptional event and usually takes place only once.

The approach of incremental loading described has much in common with the "*trap-on-use*" facility of Multics [64]. However, the trigger mechanism is not implemented by a trap handler executing in privileged supervisor mode. Rather, a user-mode library (the porter) is used. This is close to the family concept since it does not unnecessarily increase the complexity of the minimal basis and—of primary significance—strongly corresponds to the fact that server selection in many cases tends to be both application-oriented and service-related. Implementing the trigger mechanism on a library basis is a typical case for postponement of design decisions.

5.2.3 Bootstrapping

On the basis of system restructuring mechanisms comparable to those that have just been introduced, getting a distributed-memory parallel machine running may imply only the bootstrapping of a single node. This node will be a system (or generic) node referred to as the *root node*. The root node is bootstrapped with all service functions needed to carry out incremental loading, and thus incremental bootstrapping of the rest of the machine [74].

During the bootstrapping phase, the nodes of the parallel machine are categorized into *master nodes* and *slave nodes*. Typically, there will be many more slave than master nodes. An extreme but still workable case would be to have only one master node and let all other nodes play the slave role. As the following discussion will show, distinguishing between both types of nodes is mainly a software, not hardware, question.

5.2.3.1 Master Nodes

A master node is assumed to have access to some bootstrap device, such as a disk or even a host computer. The bootstrapper of this type of node actively tries to read and, once finished, execute a boot block. It corresponds to a first-level bootstrap as commonly used in almost any computer system, from notebook to mainframe. The boot block contains a second-level bootstrapper, which gets started immediately after having been loaded into main memory. The bootstrapper reads a boot description from which it obtains information about which and how many entities are to be loaded next. The following entities are required to establish a service for incremental loading:

- The *kernel entity*, consisting of nucleus and kernel. The nucleus is required to provide remote access to the locally present service functions. In order to provide a systemwide unique name space entry point, kernel extensions to the nucleus are required. The name space initially keeps track of what service function is provided by which service entity.[7] It will be used by the ROI layer at remote nodes to trigger the resolution of server addresses.

- At least one *namer entity*. This manager instance provides access to the initial system name space and will be contacted by the ROI layer.

- A *loader entity*, responsible for both incremental loading and bootstrapping. Basically, the loader takes the position of the plumber as shown in Figure 5.3. It accepts entity faults and tries to install the requested entity by reading an executable file from the bootstrap device and selecting a node for execution. This may involve a number of intermediate steps [74], such as to first load a kernel entity if the selected node has not been bootstrapped yet. Typically, the local bootstrap device is a disk or the network. Which of these holds depends on the particular architecture or configuration of the parallel machine. That is, bootstrapping means either local or remote disk access.

If the user wants to have direct interaction with the parallel machine, a *shell* needs to be initially loaded as well. The shell implements

[7] Name space initialization is done by the service-providing entities themselves. After having been bootstrapped (i.e., loaded), each entity associates service function names with the communication endpoint identifier of a clerk instance and exports the tuple into the name space.

a user interface to the loader. If all user interactions take place at some host (i.e., front-end) computer, only a *gateway* is needed. This gateway, for example, implements the linkage between "guest-level" entities running on the host and entities running on the root node of the parallel machine. By that means, ROI to the loader from the host is enabled. This further implies that entity faults raised by some guest-level program can be propagated to the parallel machine, which causes the loader to handle these faults accordingly.

5.2.3.2 Slave Nodes

The majority of the nodes of the parallel machine, however, will act passively and play the slave role. That is, once switched on, the local bootstrapper of each of these nodes listens for bootstrap requests received from the network. These requests will be issued by the root node if the loader decides to incrementally bootstrap another node. Making the nodes listen for bootstrap requests is achieved by a bootstrapper that polls the network interface locally. This avoids flooding the network with messages that carry instructions, for example, to read boot block information from somewhere. One can imagine that the network may easily become a serious bottleneck when all nodes actively issue a request for bootstrapping to some—in the worst case, the same—bootstrap device. This approach becomes even worse when thinking in terms of several thousands of nodes.

All slave nodes constitute a *processor bank*. The processor bank acts as a repository of functioning but unused nodes. Whenever resources are required for the processing of either user or system entities, the proper number of nodes are taken from the processor bank. When the entities terminate, their nodes are added to the processor bank again. In this configuration, a master node runs software that associates slave nodes with (user/system) entities. Initially, this causes the bootstrapping ("cold startup") of slave nodes. Later on, the same nodes are subjected to entity loading ("warm startup") only. The processor bank comprises slave nodes that are at the disposal of master nodes to achieve application processing.

5.3 Object Localization

Objects are used to encapsulate operating-system services. Hence, a client must know about these objects in order to gain service access. A service is logically implemented by one or more active objects (i.e., servers). This raises the problem of how to address a service that is built by replicated and (almost) identical server processes. Assume that the hardware architecture comprises a number of disk I/O nodes. Each of these system nodes is then to be equipped with a disk server. Disk servers are active objects of the same "type." They are all instances of the same class, providing the same set of (service) functions.

The following subsections discuss how to tackle this kind of addressing problem in a transparent and efficient manner. A brief introduction to the notion of *dual objects* [61] is given, followed by a description of the structure of systemwide unique identifiers suited to efficiently locate objects. After that, naming abstractions are introduced. These abstractions make the locality of objects transparent to clients and ease system configurability. The naming and addressing scheme plays an essential role in the modeling of an operating-system family that is composed of a multitude of active objects.

5.3.1 Implicit Addressing Through Objects

In order to overcome the addressing problem raised by multiple instances of the same server class, one solution might be to add to each service function a parameter that uniquely identifies a server. Although straightforward, this approach has several drawbacks. The need for explicitly providing the server address violates the principle of transparency. A user program becomes responsible for determining and keeping track of that address. The address known to the client may change if a server migrates (see also Section 5.1.4). In this case, the user program must handle communication exceptions in order to obtain the new server address. Since a dynamically alterable system structure is promoted, the user program must also explicitly trigger incremental loading of service functions to obtain knowledge of the server address. Thus, even approaches by which a parallel programming language compiler could determine server addresses statically at compilation time will not work. Server addresses will only be known at runtime. All of these problems with which a user program would be concerned limit programmability and are rarely acceptable.

Not only are user programs poorly supported by such an addressing scheme, but for the server, the additional address parameter makes no sense at all. When the server is going to execute a service function, the service request must obviously have been successfully transmitted. The address parameter plays a role only in the transmission, not in the execution, of service functions and thus will be ignored by the server. The maxim in designing service interfaces is *software compatibility*. Compatibility can be achieved by relying on the RPC paradigm [60]. The client stub mirrors exactly the same external interface implemented by some server and made available through the server stub. This is the prerequisite for the transparent switch between single- and multi-address-space representations of the same system configuration. For this reason, the address parameter must also be provided by the server functions that constitute the client/server interface—but this parameter plays absolutely no role in the process of function execution.

Since the service function is invoked on an RPC or ROI basis, it turns out that this addressing scheme also introduces additional overhead in addition to limiting programmability. The server address needs to be provided by the user-level entity (i.e., application program) and will be shipped over to the system-level entity, where it is discarded. Unnecessary parameter copying occurs, and marshaling/unmarshaling operations must be executed. The sole purpose of this address is to inform the client stub where to send the service request. That is, this address is only of local significance. It has no end-to-end relevance.

Transportation, and thus marshaling/unmarshaling, of the address parameter can be avoided by presenting two sets of service functions. The signature of the functions contained in the first set exhibits the extra addressing parameter. This parameter is not included in the corresponding signature of the functions contained in the second set. The functions in the first set are always accessed on an LOI basis. They use the addressing parameter to direct the ROI to the corresponding function in the second set. However, this approach makes software maintenance more difficult. Changes made in one interface always entail changes in the second interface. Additional overhead is introduced when the functions in both sets are represented as subroutines. The set offering the extra addressing parameter defines a new subroutine level. A specific stub generator could overcome this problem. This generator needs to be different from "standard" RPC stub generators. The stub generator must interpret either a user-defined addressing parameter or specific addressing instructions/keywords and exploit the information obtained in the coding phase of a client stub.

For all of these reasons, server addressing must work implicitly. Note that a server is an active object and, at the same time, manages a number of (active/passive) objects. For example, a file server manages file objects. The user program refers to a file object by means of a file descriptor or file handle, which is also an object. Usually, once opened, the user is not responsible for where this file object resides and by which server it is managed. That association must be established by the file handle object. In a distributed environment, the file handle should also be applicable to uniquely identify the managing file server instance. The idea is that applying a file handle implicitly causes the client to interact with the right server.

5.3.2 Dual Objects

Server addressing must work implicitly through the objects exported to the clients and managed by the server. That is, when calling a method, the applied object must automatically find the way to the right managing server. In principle, the file server example discussed above is applicable to any other type of system service. Strictly speaking, following this addressing pattern is mandatory for supporting dynamically alterable system structures. Services for process, address space and memory management, to name only a few, are addressed implicitly through objects. In particular, this approach is applicable to any sort of object and not only limited to system objects. The subsequent sections discuss in some more detail how the user/system interaction works.

5.3.2.1 Private and Public State

The file service example is continued. Before the client is able to perform I/O, it must have access to the file object. Initially, the file object must be created. This may be accomplished by a UNIX-like `open()` system call (e.g., issued by the user-level program). As a result, the program receives a file handle, which will typically be assigned to some program variable. This procedure takes place even if the program is given file access implicitly, for example, through default program parameters. In that case, the runtime system of the program takes care of all necessary steps. From the operating-system point of view, however, nothing has changed: a user-level program stores the file handle object somewhere.

The file handle is generated by the file service and refers to some internal data structure used to store information about the amount and location of disk space reserved for user data. For instance, the file handle provides an abstraction of a UNIX (in-core) `inode`. It takes the form of a *capability* to access and manipulate the associated file object (i.e., the user and system data of a file).

Uniquely addressing the file object requires the file handle to carry at least two kinds of addresses: the global address of the server and the local address of the file object within the server address space. Both addresses must be known at the client site, but they must not be known by the user program. These addresses are exported by the file service after access to the file object has been granted. Together, they represent the *public state* of the file handle. Note that the public state may also consist of the server address only. This is feasible if every file object is implemented as an active object. The public state may be even larger; for example, by adding read/write pointers or access permissions.

At the server site, a file handle will also have to keep track of information such as linkage pointers, creation and modification dates, file size, and number and addresses of disk blocks used. This kind of information constitutes the *private state* of a file handle. The private state stores safety-sensitive data and therefore must be present only at the server site.

As mentioned above, the conclusion drawn from the file server case study must be transformed to any other operating-system service. A general distinction between service-relevant data that must reside at the client site and data that must be kept local to the server site is required. Thus, one has to differentiate between the public and private data of a server. Consider a process service. Process identifier and location and size of the user-level runtime stack are typical examples of public data of a *process control block* (PCB). Private data encompasses the location and size of the kernel stack (in case of user/supervisor mode separation), various linkage pointers (for waiting lists, message buffers, page/segment tables, etc.), process attributes, a list of resources acquired, and so on. An address space service could make the logical segment descriptor public and keep the physical MMU descriptor private. A name service must make the search key public, while keeping mapping information and attributes private. The list could be continued and covers every system service. Thus, it turns out that the distinction between the public and private states of operating-system objects is quite natural. Making this distinction is indispensable for distributed operating-system structures.

Thus, the client view of a system object is to have access to the public state and to keep track of the server address. The server view, in contrast, is to have access to the public state but also to the private state. In particular, the server should not worry about the address information. The server sees the entire object state, while the client only sees an excerpt (or subset) of that state. This kind of object is referred to as a *dual object* [61]. The model of dual objects places the focus on the separation of replicated and nonreplicated state information, without prescribing consistency measures between replicated parts. This model is briefly described in the following subsections.

5.3.2.2 Prototype and Likeness

A dual object consists of two closely related but physically separated parts: the *prototype* and at least one *likeness* (Figure 5.4). Conceptually, the prototype is a mobile object. At any given time, each dual object has only one prototype, but it may have more than one likeness. The prototype logically has a private state and a public state. A likeness is built by extracting the public state and representing the extract as a *proxy object* [82], and exporting the proxy into the context from which allocation of the dual object was requested. In addition to the extract (i.e., the proxy instance), the likeness carries a systemwide unique identifier as the *reference* to the site (i.e., the server) storing the prototype. This identifier is interpreted as a hint to where the prototype is located.

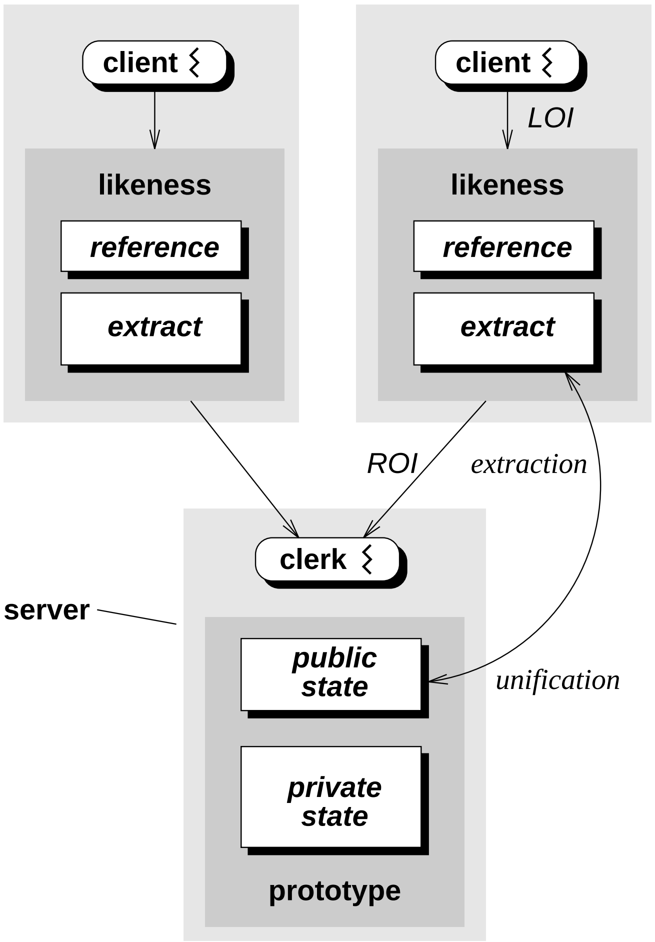

Figure 5.4: Relationship between prototype and likeness

Operations on a likeness (i.e., the extract of a prototype) are performed by means of LOI (i.e., without crossing protection domains). Operations on a prototype (i.e., the private state) by means of the likeness are performed via ROI. In that case, ROI will be performed even if both likeness and prototype are assigned to the same address space.[8]

[8]Employing ROI to communicate between collocated likeness and prototype instances is a design decision of the dual object model. This decision was made to

Operations on a prototype without using the likeness are performed via LOI. Whether LOI or ROI takes place is transparent to the calling entity, except for different execution times. The ROI request is sent to the prototype *clerk*. The clerk is an active object (i.e., a thread). It can be regarded as a "typed processor" capable of managing remotely accessible objects that are all instances of the same class.

The modification of an extract leads to a (tolerated) inconsistency with its original public state. When the prototype is invoked, the extract may be unified with the private state to reestablish consistency. Unification is done at the site holding the prototype. First, the public state is replaced by the extract, and then the operation on the prototype is executed. When the operation terminates, a new extract may be generated, replacing the extract at the storing site of the corresponding likeness. A new likeness is built, and *vertical consistency* is established. The need for generating new extracts can be specified on a per-invocation (i.e., method) basis.

The likeness is a *prototype capability*. A prototype is accessible remotely only via its likeness. As mentioned above, when a likeness is drawn from its prototype the public state is replicated by exporting a proxy. It is an individual decision of the entity that receives the proxy whether to initiate additional replication of the encapsulating likeness. Thus, under control of the entity that possesses the original likeness, prototype access may be granted to others. Note that a proxy encapsulates a replicated extract of the prototype state, while the likeness encapsulates the proxy and a reference to the clerk managing that prototype. This gives every instance possessing a likeness the right for prototype access and modification. However, the right for prototype destruction can never be passed on.

The extract of a prototype may be empty, meaning that the prototype has a private state only. Consequently, likeness replication will never evoke extract inconsistencies. In this case, ROI is the only means of interacting with the dual object. Only in the case of a non-empty extract can likeness replication result in possible inconsistencies between the various likeness instances of the same prototype and also with the

---

provide the user with a virtual machine supporting object-oriented parallel programming within single-address-space system architectures.

public state of the prototype itself. In order to (re)establish *horizontal consistency* among these instances, the model motivates customization (i.e., specialization) on a per-likeness basis. If required, a likeness, not its prototype, "inherits" the capability for consistency maintenance. Establishing consistency among replicated likenesses introduces additional overhead.[9] By making likenesses customizable on a per-application basis, this overhead only comes into play when horizontal consistency is a must. Thus, likeness specialization in that manner conforms with the goals of the family idea.

5.3.2.3 Programming-Language-Level Support

Dual objects are used to model the user/system interface—they can be used for parallel/distributed programming in general [62]. In addition, the distinction between single-tasking and multitasking, that is, between single- and multi-address-space representations, becomes a configuration decision. This decision must have no influence on the user program. From the viewpoint of using a specific system service and thus applying a dual object, the different operating modes must be transparent to the user. Applicability of an operating-system family is null and void when reprogramming of user routines is required to take advantage of a different family member. At most, recompilation may be acceptable under certain circumstances.

Dual objects support the modeling of a family of (distributed/parallel) operating systems, since the appearance of a common object state is created. The appearance is that both client and server reside within the same address space. Implementations are enabled which assume that a user task shares the same address space with operating-system functions. On the other hand, the same system implementation is feasible in a microkernel-based manner, for example, without reprogramming either user or system routines. Not only are user programs made independent of a given operating-system representation; in addition, operating-system modules are made available to them that are applicable for a broad range of family members. However, achieving this kind of system configurability requires more than

[9] A prototype instance does not maintain a list of likenesses and is thus unable to force "universal" consistency if desired.

just exploiting dedicated object models. Programming-language-level support is indispensable as well.

For each representation of a dual object, a class specification is used. Both specifications are automatically generated from a *dual class* by means of a class generator [61]. The dual class is the original class enriched by annotations (enclosed in comments) to describe the behavior of objects of that class. Object orientation is supported by allowing the dual class to take the position of a subclass. That is, a dual class can still be constructed through inheritance. Consequently, the operating-system interface built by such classes is open and strongly application-oriented, even the interface to (logically) remote service instances. A user program is enabled to specialize operating-system functions networkwide (i.e., crossing protection boundaries).

Basically, a generated client class is a subset of the generated server class. Both client and generated server class contain the public classes of the original class hierarchy, specified by the dual class, and they are extended by additional classes to support hashing and server selection. The server class, furthermore, contains all the private classes of the original class tree. Moreover, a clerk class is generated for the server site. This class describes active objects that are capable of managing dual objects of the dual class from which the clerk description was generated.

5.3.3 Object Placement

The programming language chosen for the implementation of dual objects is C++ [84]. One of the main reasons for using C++, besides its efficiency and widespread use, was the constructor/destructor principle embodied in the language. Classes may specify user-defined constructor operations that are automatically executed when instances of these classes are created. Similarly, destructors are automatically executed when an instance is destroyed. This allows object-specific initialization and finalization code to be transparently integrated, which is of particular benefit for the implementation of operating systems.

Thus, a dual class is an annotated C++ class, with the annotations enclosed in C-like comments. The class generator output is pure C++. One of the most notable aspects of class generation is the extension of the constructor signature. An additional formal parameter is appended to the user-defined constructor interface specification. This parameter determines where to create a prototype instance. Basically, the parameter refers to the communication endpoint address of a clerk of the dual class. These manipulations are made with the likeness class only. When an object of the likeness class is declared, a prototype instantiation request will be sent out to the clerk. The clerk installs and initializes the prototype, extracts the public state, and delivers that state to the likeness.

The address parameter has a default assignment, which is the logical location of the clerk. The logical location of a clerk is identified by the name of the dual class from which the clerk specification has been generated. In the default case, at likeness instantiation time the naming system is employed to resolve the actual physical location of a clerk instance.[10] The actual clerk location is remembered by the reference component of a likeness—the likeness is a reference to a clerk. By exploiting the clerk reference and using it as an actual parameter value for the constructor (i.e., overwriting the default assignment), the new prototype gets collocated with the prototype that corresponds to the likeness from which the reference component was taken. Either way, object (i.e., prototype) placement is directed by the client site (Figure 5.5).

A client thread creates a likeness locally. This results in an ROI request to a server thread, the clerk. The ROI request is implicitly sent out by the client thread, as the result of executing the likeness constructor. Having received a construction request, the clerk locally creates a prototype and finishes construction by passing the public state back to the client. There, the likeness construction will be resumed—the likeness constructor returns from ROI. The prototype extract is remembered, which finishes the likeness construction. Likeness destruction works similarly. In particular, destruction of a likeness instance automatically causes the destruction of the corresponding prototype. This greatly eases resource release in a distributed environment and is one of the most important features of dual objects. However, the destruction request only succeeds when issued by the client thread that originally

[10]The naming system is described later.

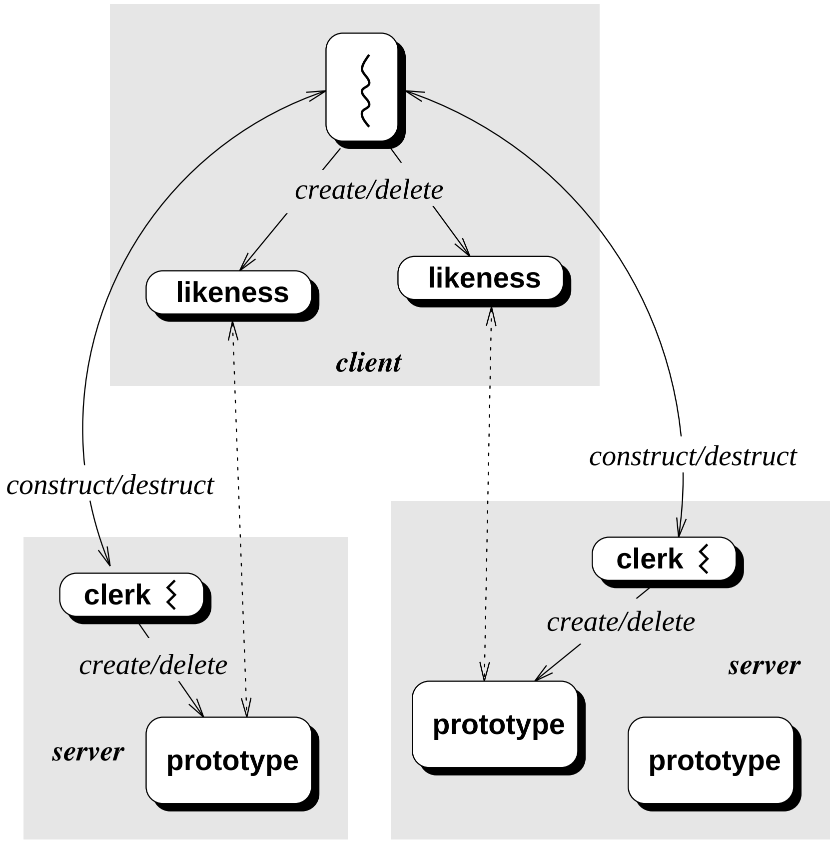

Figure 5.5: Prototype placement

requested prototype construction. Likeness instances may be passed to other threads to enable the sharing of prototypes. Thus, threads receiving likeness copies will be unable to delete the prototype; but they only succeed in the destruction of their private likeness copy.

In the dual object model, the clerk basically implements a "*typed virtual processor*." Object placement on such a processor is achieved by furnishing the constructor with a (logical/physical) processor address. The default assignment is a logical address of the clerk (processor). In that case, the likeness constructor requests the name service to deliver the location of an appropriate clerk. Load balancing is introduced by asking the load balancing service to determine where to place a prototype (i.e., to deliver a clerk address), and using the delivered systemwide unique clerk address as the target for the construction procedure. Technically, load balancing is object-specific and not necessarily a class property. It can easily be made a class property by designing a subclass that inherits the likeness class. The constructor of that subclass is then able to take care of object placement and load balancing in a class-specific manner.

Exploiting dual objects is not only a case for the system programmer. It is especially applicable for fine-grain and irregular parallel programs. Every application task then defines a virtual processor, which primarily processes threads. The initial thread of control in each of these tasks is a "thread clerk." The thread clerk accepts construction/destruction requests on an ROI basis and dynamically extends/shrinks its team of threads. These requests are sent by other tasks of the same parallel application. Application-specific placement strategies can be integrated on either a class or object basis, similar to those described above.

5.3.4 Unique Identifiers

The likeness of a dual object stores a systemwide unique identifier. This identifier refers to an active object, the clerk, that is able to extract/unify the public state from/of the prototype corresponding to that likeness. Since extraction and unification takes place on an ROI basis, the identifier represents the *communication endpoint address* of the clerk involved.

Communication endpoint addresses are interpreted by the nucleus. Such addresses must be designed with the following goals in mind. First, they must be easy to use for both the nucleus and the entities relying on the nucleus. The most desirable case would be to make them fit into a single CPU register, that is, to represent them as a *simple data type*. Second, these addresses must make object location straightforward, with negligible overhead. This requires proper structuring of the address rather than understanding it as a flat numerical value. Third, the address must be unique in the sense that regeneration of the same bit pattern becomes highly improbable within a specific time period.

The time period must be at least as long as the lifetime of a given application—desirably, equal to the system lifetime.

Obviously, these goals are somewhat contradictory. Making a communication endpoint address unique over the lifetime of the entire system is only feasible if very large and sparse addresses are used. Using small addresses and relying on a central system service that ensures the generation of unambiguous (i.e., presently unused) identifiers is impractical. In a distributed-memory machine, this service can be requested only on a message-passing basis, which expects that the communication endpoint address of the corresponding server is already known. More crucial, in the realm of massively parallel systems any central approach must be avoided. Rather, large sparse addresses must be used. Large addresses, however, do not always fit into a single processor register. Obviously, the kind of processor technology used to build the parallel machine makes a difference. A CPU with 64-bit (or even wider) registers will not cause very much trouble. The situation is completely different when relying on a 32-bit register machine. A 32-bit register is sufficient to hold a structured address that eases object localization, but is far too small to make that address unique systemwide over a long period of time.

A tradeoff exists that calls for the distinction between two different sorts of unique identifiers: local and global. These two kinds of identifiers are briefly discussed in the following subsections.

5.3.4.1 Local Unique Identifier

Basically, a *local unique identifier* (LUI) is nothing but a hash key for locating threads. The hash key is used to retrieve objects out of some object store. In the particular case of the nucleus, the hash key serves to select active objects that are involved in a communication activity. The hash key is structured not only to ease the hash procedure that must be executed locally by each nucleus, but also to efficiently identify the locations of nuclei (i.e., hash tables). Every nucleus encapsulates a hash table that keeps track of all collocated active objects. Thus, the location of a nucleus directly gives the location of the enclosing hash table (Figure 5.6).

The identifier structure shown in Figure 5.6 distinguishes between

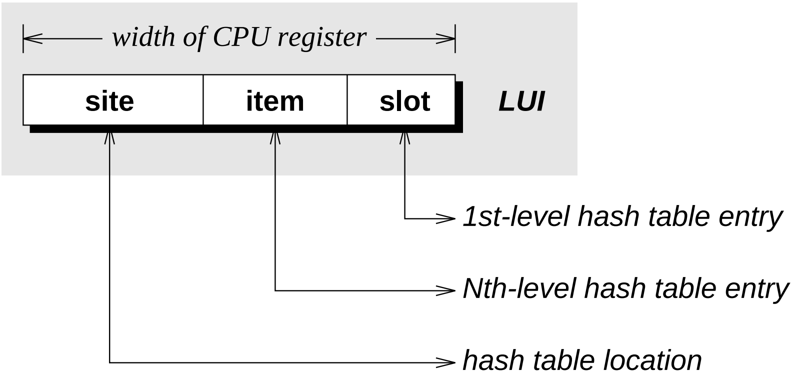

Figure 5.6: Structured communication endpoint address

three fields. A `slot` field serves as an index to the hash table. Every hash table entry is a (singly linked) list of hash items. An `item` field is provided to identify a single hash item relative to the per-slot hash list. It differentiates between a number of hash items associated with the same hash table index. Finally, the hash key includes a `site` field. This field represents a (logical) node number and gives a hint to where the hash table related to that LUI resides. In fact, the `site` field is the logical node address of a nucleus.

Normally, the hash table stores pointers to locally known active objects. Each LUI contained in the hash items then contains the same `site` field. More specifically, this field stores the logical node address of the local nucleus.[11] In particular, each LUI of the same per-slot hash list has a different `item` number. This correspondence is shown in Figure 5.7.

As indicated in Figure 5.7, the hash table entry stores only a list head pointer. This makes indexing fast for two main reasons [15, 75]. First, the size of a pointer value is a power of two, which results in a small scaling factor (and a simple left-shift operation) to generate a

[11] As will be discussed later, a hash item may take the form of a *forwarding identifier*. This may result in LUI patterns the `site` value of which is different from the local node address. Thus, the hash table is also used to locally store remote object references.

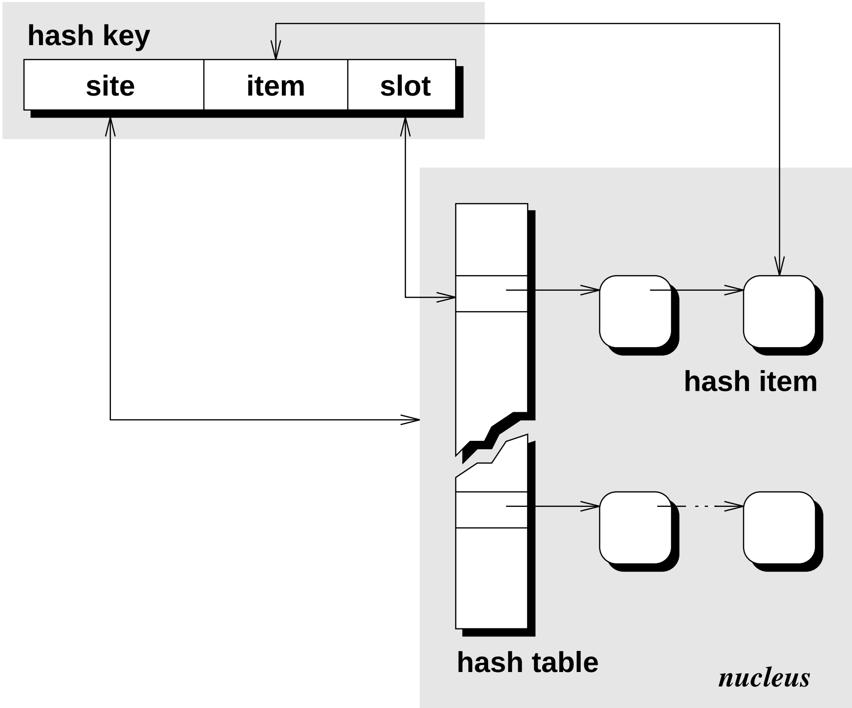

Figure 5.7: Use of the communication endpoint address

hash table offset for selecting a certain pointer instance. Second, since pointers are small compared to hash items, each hash table entry can be initialized with a pointer to a single default hash item. The default hash item acts as a "stopgap" and prevents communication from succeeding if an invalid LUI is used. The stopgap corresponds to a hash key that is never generated and exported to some thread. Passing a stopgap during the lookup procedure always raises a hash fault. On this basis, a hash key need not be explicitly checked for validity, that is, whether the `slot` value lies within the index range. There are always as many defined hash table entries as specified by the index range of the hash table.

The triple {`site,item,slot`} defines a unique route to reach an active object. Whenever the nucleus either accepts a local message-passing request or receives a message packet to be delivered to a thread (i.e., active object), LUI decoding takes place. The hash table is indexed, using the extracted `slot` number.[12] This gives access to the first-level hash table entry and a list of hash items. Every hash item at least stores an LUI and a linkage pointer. If the LUI of the first-level hash item does not match the LUI provided by the request/message, a search through the entire hash list takes place. If the search fails, a remote object reference is assumed. In this case, the communication system is asked to forward the request/message to the nucleus whose logical node address is specified by the `site` field of the LUI. An explicit `site` field check in order to determine the possible location of an active object is not performed. The reason is to allow for the installation of locally established forwarding identifiers, supporting the migration of active objects (i.e., threads).

Note that in the worst case, only a single hash list must be processed per hash table. This is ensured by the LUI structure, in particular by the `slot` field. The limit of the list size is determined by the `item` field. Whether hash lists are present at all depends on a number of factors:

- First of all, the size of the bit pattern reserved for the `slot` field determines the length of the hash table. The LUI structure allows for an index range of `0..slot-1`, and thus a maximum of `slot` hash lists. In general, the larger the `slot` field the more unlikely the need for hash lists. Under normal circumstances,[13] every hash table entry corresponds to an active object. An 8-bit `slot` field, for example, means a total of 256 first-level hash items, and possibly threads, per node. An application which exhibits that large a number of local threads is not a typical case for parallel computing, unless one is dealing with the virtualization of a massively parallel machine.

- Second, it depends on the number of threads generated by the application and on the number of nodes provided by the hardware whether hash lists are needed. Assuming a 12-bit `site` field and an 8-bit `slot` field, more than $2^{12} * 2^{8} = 2^{20}$ threads must be

[12] A simple logical **and** operation is performed on the LUI.

[13] The exceptional case is discussed later and deals with forwarding identifiers.

running at the same time before hash lists come into play. This proves to be a fairly massively parallel application.

- The third factor is the need for dynamic load balancing. In that case, task (and thus thread) migration may take place. In order to not change already known communication endpoint addresses, *forwarding identifiers* must be installed. These identifiers are hash items that are assigned to map old (i.e., stale) communication endpoint addresses onto new ones: they map old thread addresses onto new thread addresses. Forwarding identifiers always take the position of Nth-level hash table entries, with $N > 1$ when a first-level entry is already defined. This ensures a fast lookup procedure in the normal (i.e., nonforwarding) case.

- Finally, new hash items are inserted into the hash table on a *round-robin* basis. This leads to hash lists only if all hash table entries are in use.

Typically, the triple {`site,item,slot`} is varied to reflect the specific demands of the underlying hardware architecture. Meaningful parameter values could also be a 16-bit `site` and a 4-bit `slot` field, supporting a parallel machine of up to 2^{16} nodes. All these examples still assume a 12-bit `item` field to make the LUI fit into a single 32-bit register. A smaller `item` field makes room for a larger `site` field, for instance. Thus, a machine consisting of a million nodes could easily be supported by using bit field sizes as specified by a parameter triple of {`20,8,4`}. Forthcoming massively parallel systems, however, will be based on 64-bit technology. This demonstrates that the proposed LUI structure scales well with the technology used to build future parallel machines.

In principle, an LUI has only local significance, although it may also provide for a certain degree of global uniqueness. Limited global uniqueness is made possible through the `site` field. However, an LUI can be considered globally unique only if static or near-static systems are to be supported. These systems require no additional provisions and can thus be efficiently supported on a pure LUI basis. For others, the LUI must be extended properly.

5.3.4.2 Global Unique Identifier

In a highly dynamic and massively parallel system, a 32-bit LUI may soon become ambiguous. Suppose an LUI was known by some application task. If the active object associated with that LUI is destroyed, and afterwards, other active objects are created and destroyed on the node, to which the LUI refers, a regeneration of the same LUI bit pattern may become possible. This causes a serious problem when the application holding that LUI value is going to send a message using the LUI. Then, the nucleus cannot distinguish between the old and new LUIs, and thus cannot reject the communication request.[14] This problem can be mitigated by a process creation strategy that tries to allocate new active objects onto different nodes. In that case, the `site` field of an LUI varies, although all other fields may remain equal.[15] New LUI values can then be distinguished from old ones, and formerly valid LUI-to-thread relationships can be detected as stale. However, this is not applicable in all cases, especially for regular parallel applications assuming a nearest-neighbor model of node and thus task mapping.

The solution to this problem is known as "*revocation of capabilities*" [28]. An LUI is a capability for an active object. Subjects possessing that capability (i.e., knowing the LUI) are allowed to communicate with that object. Now, if an LUI becomes stale, all replicas must be invalidated—all capabilities must be revoked. This would require keeping track of all instances of the same LUI, that is, maintaining capability lists somewhere. Moreover, copying an LUI must then be done by a system primitive to ensure an up-to-date list of capabilities.

All these aspects do not allow direct application to scalable distributed systems. The mechanism to ensure the uniqueness of low-level object identifiers must rely on a decentralized approach and reduce user/system interactions to an absolute minimum. The LUI must be made a *sparse capability* [58]. This can be readily achieved by extend-

[14] Also note that the problem discussed here is not restricted to distributed or parallel systems. Even in a local system the same situation may arise—but it tends to be highly improbable. As long as user-level tasks know about process identifiers, the system is potentially in trouble due to ensuring the uniqueness of newly generated identifiers.

[15] Note that this only increases the likely time interval between a successive generation of the same identifier; it does not really solve the problem.

ing the LUI by a *random number* of meaningful size. In this approach, the LUI still serves as a fast addressing key, while the random number extension ensures uniqueness. That LUI specialization leads to a *global unique identifier* (Figure 5.8).

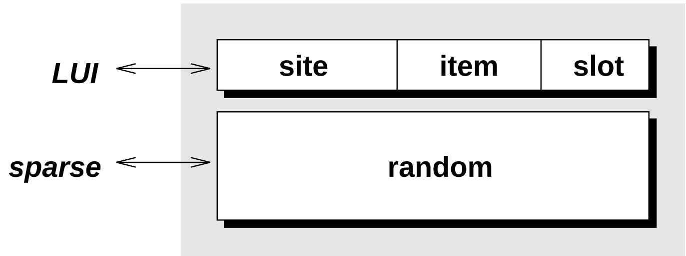

Figure 5.8: Systemwide unique communication endpoint address

The random number field makes the LUI sparse. The size of this number is problem-oriented and depends on the quality of the random number generator. Highly dynamic and long-lived applications and systems require a larger number space than modestly dynamic and short-lived applications and systems. The larger the number space, the sparser the individual numbers.

Using a random number scheme to make an LUI sparse has the decisive advantage of making the generation of systemwide unique communication endpoint addresses an autonomous procedure of every node. Every node is equipped with a software-implemented random number generator, implemented by the nucleus. The only provision required is to initialize the generators with different seed values. For example, this may be achieved by employing the physical address of a node. The interconnection network must rely on hard-wired and unique node addresses. These addresses are often readable by system software via specific hardware registers or devices. Consequently, this initialization procedure can be executed autonomously by every node.

5.3.4.3 Opaque Identifier

A random number scheme to generate sparse capabilities usually has to rely on fairly long bit patterns per generated value. 32-bit sparse numbers are not in all cases satisfactory. Rather, 64-bit or even 256-bit representations may be required. Thus, by additionally taking the size of a single LUI into account, a systemwide unique identifier of this sort may easily come close to or even exceed the size of most of the messages exchanged. The result is a poor ratio between system and user data per message. There are fine-grain parallel programs that generate messages of about 16 to 32 bytes each. At least a unique identifier must be encoded within each nucleus-level message to select an active object that is designated to play the receiver role. Thus, using, for example, a 256-bit sparse identifier might be a good choice for loosely coupled, large, scalable distributed systems, but not for massively parallel systems. In the latter case, the messages to be sent are at least twice as large as those usually generated by many parallel applications.

The size of a systemwide unique identifier determines the communication overhead in massively parallel systems, and therefore must be tunable with respect to the applications. In the case of static applications, the LUI representation works well. Support for dynamic applications may call for an LUI extension, and the size of this extension need not be the same for all types of applications. A solution must be provided that deals with variable-size communication endpoint addresses. The solution is to represent a communication endpoint address as a *virtual pointer*. The assumption made here is that a pointer value will always be of a simple data type. That is, a pointer always fits into a single CPU register. If 32-bit processor technology is given and 64-bit systemwide unique identifiers must be supported, the pointer refers to a properly sized data object stored in main memory. If a 32-bit-sized identifier meets the needs of the given application, the pointer is exactly the value of that identifier. These two representations of a virtual pointer are illustrated in Figure 5.9.

The virtual pointer is referred to as the *opaque identifier*. Applications dealing with opaque identifiers are made independent of the actual size of the corresponding unique identifier. A nucleus implementation depends on that size, but not on a different user interface specifica-

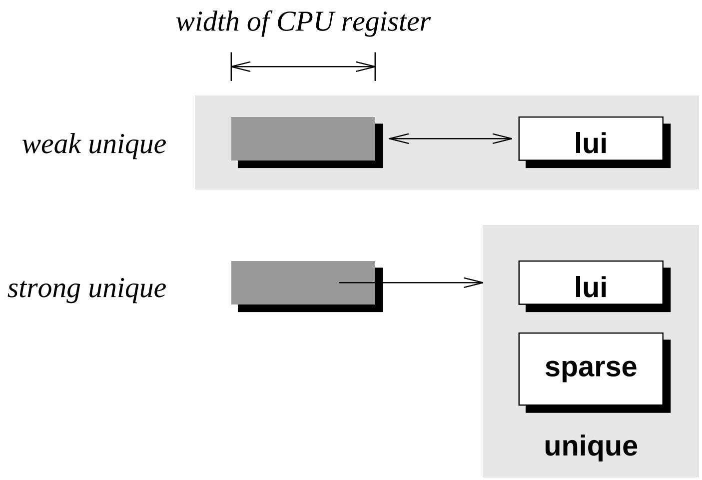

Figure 5.9: Representations of an opaque identifier

tion. Different nucleus implementations result from different strategies of how to handle, access and transfer opaque identifiers. If an opaque identifier is the LUI itself (i.e., *weak unique*), the simplest and thus most efficient nucleus implementation is given. Identifiers need not be read from user-level memory, but can be exchanged through CPU registers. In contrast, an opaque identifier that is a pointer to a unique identifier (i.e., *strong unique*) entails costly memory-to-memory copies.[16] This becomes even worse if user/supervisor mode separation is made on the basis of individual protection domains. That case may involve the programming of MMU descriptors in order to access user-level data from nucleus space.

In the PEACE approach presented, the different nucleus implementations dealing with different opaque identifier representations define individual members of the kernel family. In particular, a nucleus that supports strong unique identifiers provides the basis for kernel family

[16]Experiments have shown that, on the basis of a 40-MIPS i860, the distinction between a 32-bit and a 64-bit unique identifier (including LUI) did result in a communication performance degradation of approximately 15% when serving a *send-receive-reply* sequence by the PEACE nucleus.

members which introduce either network integrity or security (refer also to Section 2.4). All other members mainly use weak unique identifiers.

5.3.5 Communication Gates

An LUI encodes a node address. This may be a logical or physical address, dependent on how the (nucleus-level) communication system interprets that information. In the case of a physical address, the `site` field of the LUI must be large enough to keep track of all possible node numbers of the parallel machine. However, if the parallel machine exhibits a number of nodes that is not representable by the `site` field, an additional mapping is necessary. Basically, this means interpreting `site` values as logical node addresses and mapping these addresses onto a subset of physical node addresses. Each subset is then referred to as an individual user partition.

Anyway, the LUI encodes some hardware-dependent data. In order to support a dynamically alterable system structure, the hashing mechanism also takes care of *forwarding identifiers*. Every communication endpoint address is technically represented by a so-called *communication gate*. Each of these gates is a specialized hash item. It stores not only the linkage pointer and a unique identifier as a hash key, but also a second hash key as an alias. If not null, the second key implements a forwarding identifier. That identifier is nothing but an LUI-based hash key and refers to a (local/remote) hash table entry. Whenever the hashing mechanism detects a forwarding identifier, that identifier is used as the new hash key, and lookup continues. This procedure is repeated until a null forwarding identifier is found (Figure 5.10).[17] A null forwarding identifier denotes a communication gate that is encapsulated by an active object descriptor (i.e., PCB). As a matter of fact, the hash key related to that gate is not only a communication endpoint address but also the systemwide unique identifier of the thread modeled by the enclosing PCB.

[17]In order to ease the destruction process of forwarding identifiers, communication gates also carry a backward pointer to the preceding gate. Since the chain of forwarding identifiers usually proceeds through the network, the backward pointer must be a systemwide unique identifier too.

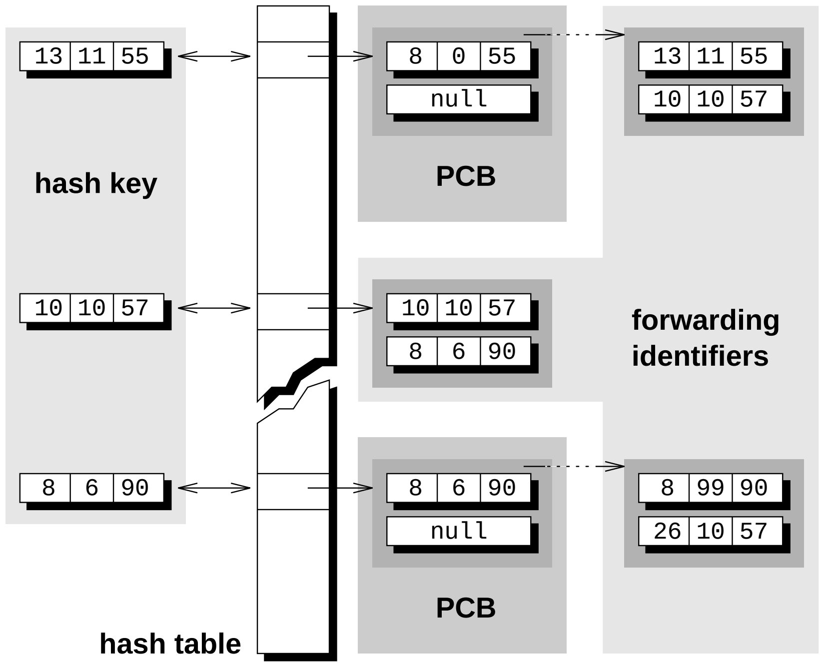

Figure 5.10: Gates used as forwarding identifiers

Gates take the form of port-like communication endpoint addresses. They are ports with routing, but not buffering, capabilities. The example shown in Figure 5.10 demonstrates that forwarding identifiers may take various forms. The most likely case is to store references to remotely residing active objects ({`26,10,57`}).[18] That form is used to keep track of migrated threads. Another common situation is to associate a single thread ({`8,6,90`}) with more than one communication endpoint address ({`13,11,55`} and {`10,10,57`}). The default scenario is to create a gate in one-to-one correspondence to active objects, giving a thread one single communication endpoint address. A further case, for example, is to locally remember forwarding identifiers to virtually remotely residing active objects ({`13,11,55`}).

[18]Here and in the following examples, the numbers do not specify the size but the value represented by each LUI bit field.

Forwarding identifiers are a simple mechanism for the interpretation of communication endpoint addresses. Maintaining these identifiers, however, may be quite complicated. Replicated data structures must be kept consistent. Furthermore, provisions must be made to help overcome the potential problem of either cyclic or broken chains of identifiers. This motivates a separation by which the nucleus plays the passive role and must solely interpret forwarding identifiers, whereas the kernel and POSE play the active role and take care of all necessary maintenance tasks.

5.3.6 Naming Abstractions

Systemwide unique identifiers specify the communication endpoint address of a message transaction. They are numeric, structured low-level identifiers. This representation is used mainly for performance reasons. The low-level addressing scheme, however, has a drawback for network transparency in general: the `site` field is a location-dependent value. Its presence relieves the nucleus (or kernel) of keeping track of local communication gates that point to remote tasks or servers. Local gates storing forwarding identifiers to remote tasks implement local communication endpoint addresses, and thus network transparency concerning object localization. However, this addressing scheme would entail the necessity of replicating the communication endpoint addresses of receiver tasks at the sites that lodge the corresponding sender tasks. Not only replication but also maintenance of a replicated (i.e., distributed) data set would be required. The LUI `site` field makes local communication gates unnecessary under normal circumstances, that is, when dynamic restructuring caused by a node crash does not take place. They make the lookup procedure fast but the modeling of dynamically alterable system structures difficult.

Even forwarding identifiers are not the final solution to this problem. They introduce transparency only at the price of kernel-level memory resources. Transparency is weakened when the kernel runs out of memory for storing communication gates to install a new forwarding identifier. Garbage collection techniques may be applied to detect unused forwarding identifiers. For example, each time a forwarding identifier is found during the LUI lookup procedure, the corresponding communication gate is marked. A kernel-level thread, preferably activated from within the idle loop, or the installation procedure makes a complete run over the hash table and figures out which forwarding identifiers were not used since the last collection. The memory resources acquired by communication gates of unmarked identifiers can then be reused to install new forwarding identifiers. This, however, will cause future hash faults when applying a communication endpoint address that was previously mapped by a forwarding identifier.[19] These faults must be reported to higher levels as communication exceptions.

In order to maintain transparency, the communication exceptions must be handled accordingly. This means figuring out the new communication endpoint address (i.e., location) of the active object with which communication just happened to fail. A logical address of this object is required. The address must not include any location-dependent information; in particular, it must not be an LUI. Note that even nucleus-level broadcasting techniques will not provide a solution to this problem. A new forwarding identifier must be installed instead of updating existing forwarding identifiers by broadcasting a query for the actual location of a given thread. Besides implementation aspects, this is the main reason why the nucleus does not support one-to-many communication. Rather, an approach is pursued that relies on "user-level" support to keep track of up-to-date communication endpoint addresses and to reduce the number of forwarding identifiers. This approach is based on the naming scheme illustrated in Figure 5.11.

Symbolic identifiers, represented as user-defined character strings, introduce location transparency. Moreover, they are applicable not only to address active objects, that is, objects which are capable of actively receiving messages. A symbolic identifier may refer to any kind of object. Thus, it serves as a general addressing mechanism. Symbolic identifiers are stored in *name spaces* and associated with systemwide unique identifiers. Name servers will take care of the mapping of symbolic to systemwide unique identifiers: they are used to map logical server ad-

[19] In addition, the potential of *orphan forwarding identifiers* must be taken into account. Simply discarding an unmarked identifier is not sufficient. All preceding/succeeding identifiers in the chain must be mapped to the successor entry to avoid orphans. This usually entails POSE/kernel activities following the chain in both directions until a null forwarding identifier is detected.

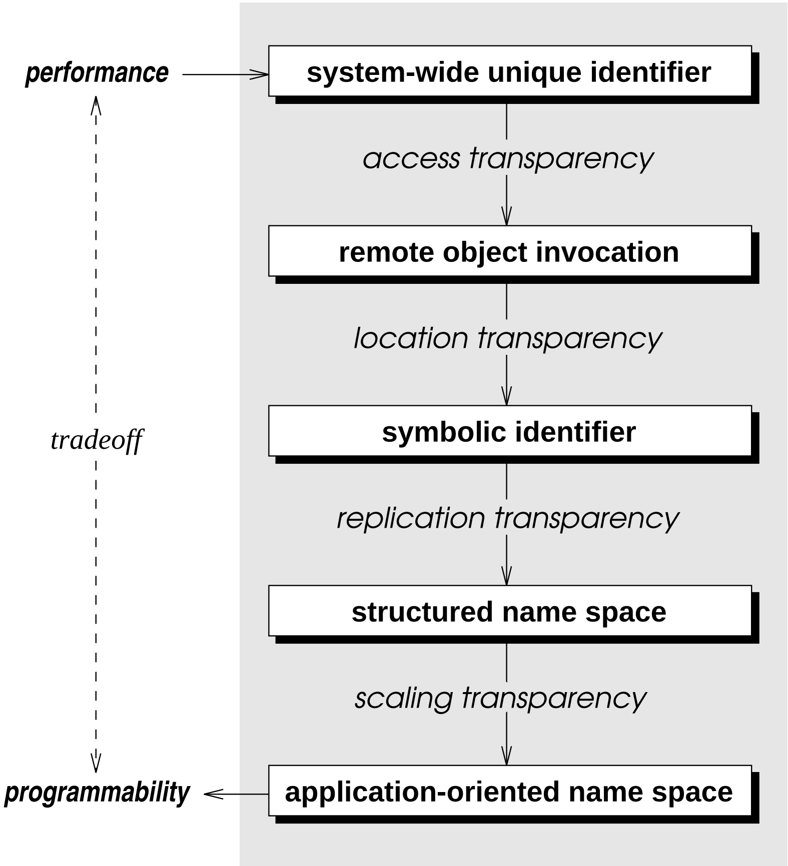

Figure 5.11: Naming abstractions

dresses onto physical server addresses. As shown in Figure 5.11, this mapping must reflect various aspects that deal mainly with the modeling of a family of (parallel) operating systems. The following subsections describe the mappings shown in Figure 5.11 in detail.

5.3.6.1 Access and Location Transparency

The nucleus operates with systemwide unique identifiers for performance reasons. In addition, the nucleus implements the minimal subset of system functions that must reside on every node of the parallel machine. This is the starting point of the following considerations—and what the mapping process is aimed at.

With the decision to implement system services using ROI (or RPC), *access transparency* is introduced. The user sees only a procedural interface. There is no way to apply a systemwide unique identifier unless demanded by the interface of the service function. In principle, this leaves it solely up to the ROI layer to cope with "physical server addresses." Thus, a good design makes sure that the service interface specification does not contain physical object addresses.

The use of symbolic identifiers to name system components that provide services hides the presence of systemwide unique identifiers. The logical object addresses introduce *location transparency*, which is achieved by the dynamic mapping applied to obtain systemwide unique identifiers from symbolic identifiers. At this level, naming and communication exceptions are handled. If the use of a stale communication endpoint address has been signaled by the nucleus, the user-level naming subsystem is requested to resolve a symbolic identifier. The result is the actual communication endpoint address of an active object. As was outlined in previous sections, that address is nothing but a systemwide unique identifier.

A symbolic identifier is a name that does not include "hard-wired" address information such as the `site` field of an LUI. In particular, the naming system does not interpret the character strings, and therefore cannot benefit from encoded site information. Site information may be encoded in these strings but will have no effect on the mapping. Symbolic identifiers are only significant end-to-end. They are interpreted by either the client or server. A name server takes the character string, applies the mapping function and thus returns the associated systemwide unique identifier. In reality, even a name server does not know anything about systemwide unique identifiers. Rather, it maps a character string onto a byte string, whereby the byte string in most cases represents a bit pattern that stands for a systemwide unique identifier. Thus, both input and output data of a name server are completely user-defined, with name servers implementing "*associative memory.*"

5.3.6.2 Replication Transparency

The naming approach offers an interface to active objects by which only location-independent symbolic identifiers are supported. The motivation is to make active objects configurable by a third party. Given a site-independent active object, the third party decides where to place it. The naming system does not benefit from site-dependent information, such as the node address of an active object encoded into a symbolic identifier. As a consequence, replicated and code-identical active objects cannot be distinguished by name. All these objects could use the same symbolic identifier (i.e., character string) to make themselves globally known. This issue is revisited below. The potential for name clashes due to code-identical instances must be solved by means of a structured name space.

The global name space is partitioned into a number of *name planes*. Each name plane implements a unique naming domain within which a single symbolic identifier is guaranteed to be unique. The same identifier may result in different outputs of the mapping function (when all name planes are taken into account). A structured name space of this sort (i.e., relying on different name planes) introduces *replication transparency*. The third party, for example, associates active objects with name planes and thus, logically, with individual naming domains. Objects can be migrated to different name planes, extending or reducing the scope. Active objects may or may not share the same name plane. For instance, tasks of the same application will be bound to a single name plane, to enable network-transparent object localization. As a consequence, tasks belonging to different applications cannot identify each other. This is a prerequisite for maintaining multi-user security.

5.3.6.3 Scaling Transparency

A family of operating systems implies giving every application, logically, its "private" ensemble of server objects. This not only means providing different kinds of services, such as process management for one and file management for another application. It also includes providing different implementations of the same service interface. Servers are abstract data types, their interface being described by an "*abstract class*." The class name is then the almost natural choice for use as the symbolic identifier to address a given instance of that class.

For these reasons, a number of different objects may be bound to the same logical name. Using name planes helps to reestablish the uniqueness of each name. However, this alone does not solve the problem that a given application must have access only to those name planes that are keeping track of (server) objects belonging to the application domain. The name space must be structured in an application-oriented manner. Name planes are to be configured on a per-process basis. This introduces *scaling transparency*, that is, it supports a scalable (parallel) operating-system architecture. Scalability means the property of an operating system that is at least statically adaptable to given user environments.

5.3.7 Name Space Partitioning

The main goal in the modeling of a name space is to keep track of service locality. That is, operating-system services that are used by a specific application shall be accessible only by that application. The possible one-to-one correspondence between an application and its service is the consequence of a family-based operating-system design. Several operating-system family members may be active at the same time, but their services are meaningful only for certain applications. Note that a family member exists only in the context of a specific application. For example, a Unix file service assumes Unix and not MS-DOS applications.

Another goal is to design a fundamental (i.e., minimal) naming system that does not rely on multicast communication services in order to find out where a specific object resides. There are two reasons for this. First, multicasting is not necessarily supported by the underlying network hardware. A software solution for multicasting would thus have to be provided if the naming system relied on these services. The software solution then needs to be implemented by the kernel entity and will impair almost every functional unit of the nucleus. This is because the "uses relation" [67] amongst nucleus modules becomes more complex. In addition, the more complex nucleus organization would increase the risk for a communication performance loss. Making the shift from one-to-one to one-to-many communication in the *V* system, for example, definitely deteriorated one-to-one communication, although the network hardware supports broadcasting [17]. Such a minimal basis will not conform to the family idea. Second, a dynamically alterable operating-system family will not take much advantage of multicasting.[20] Most of the operating-system services are part of a specific application domain. More specifically, these service are loaded when the application domain comes up. This makes servers known at least to the loader instance, and it is an easy undertaking to pass the server address on to the instance that raised the service fault.

Entities of one application must be logically isolated from entities of a different application. This avoids name clashes when unrelated entities use the same symbolic identifier to address an application-related entity. That identifier may refer to different implementations of the same service interface. Moreover, user tasks may apply symbolic identifiers to identify each other in a logical and network-transparent manner. These identifiers, however, are bound only to a specific and application-oriented scope (Figure 5.12).

The global name space is structured into *name domains*. The number of these name domains depends on the process topology defined by the applications, the set and structure of commonly used system services, and the hardware architecture. Every name domain is implemented by one or more name servers. Each of these name servers implements a single name plane. The name plane stores symbolic identifiers (i.e., character strings) and their associated systemwide unique identifiers (i.e., byte strings) and represents a problem-oriented registry of logical object addresses.

5.3.7.1 Name Domains

Names are unique within a name plane only. In the same name domain, different mappings from symbolic to systemwide unique identifiers are allowed. These mappings are necessary, for example, to link name servers by following a simple procedure. Name servers themselves have a logical address—they are associated with at least one symbolic

[20] Multicasting to support user applications is a case for minimal system extensions.

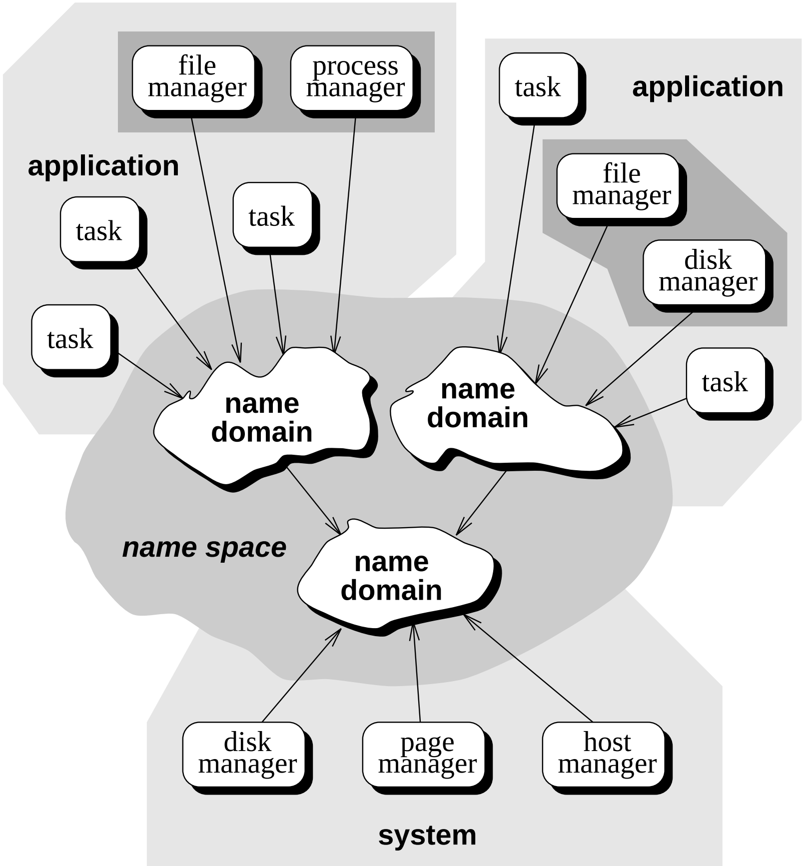

Figure 5.12: Name space with different name domains

identifier. As shown in Figure 5.13, in order to establish a hierarchical name space, a succeeding name server is remembered by its predecessor using the "`..`" character string. This symbolic identifier, also referred to as the *intra-domain link*, is associated with the systemwide unique identifier of the succeeding name server.

Thus, name lookup becomes a simple procedure. A *name porter* instructs a name server to deliver the byte string associated with a given user-defined character string. If the string cannot be found, the name porter follows the intra-domain link by instructing the same name

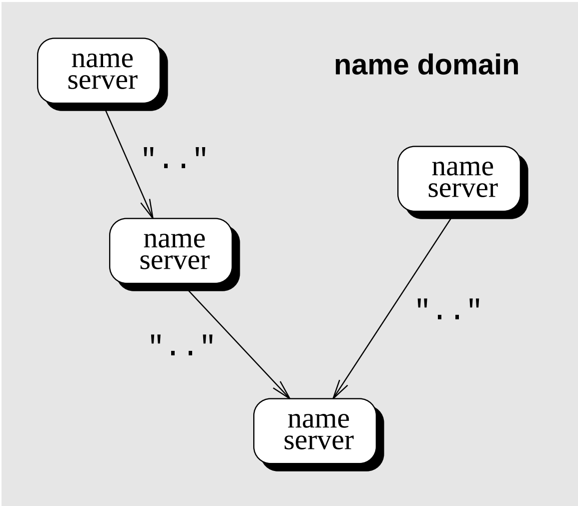

Figure 5.13: Hierarchically arranged name servers

server to deliver the byte string associated with "..". In the case of a defined link entry, the porter interprets the received byte string as the systemwide unique identifier of another name server at which to continue the search. Thus, a standard lookup procedure is provided that corresponds to the scope rules often discovered in block-structured programming languages: the search for a variable declaration begins in the innermost block and, upon a miss, is extended to enclosing blocks until the variable declaration is found or the outermost block reached. If, on the path from the innermost to the outermost block (i.e., scope), the variable declaration is not found, a declaration fault will be encountered. Name servers correspond to these blocks. The name porter will detect that a given name is not (yet) defined. A *naming exception* is raised and may cause the incremental loading of an entity corresponding to the requested name. This takes place when the name porter tries to dereference the intra-domain link and does not succeed.[21]

[21]An optimized procedure is to furnish the name server not only with the user-defined string, but also with the character string ("..") denoting the intra-domain link. In that case, the name server will automatically try to follow the link pointer if the user-defined string is not found. If a linkage is defined, the current name

5.3.7.2 Entity to Name Domain Relationship

Logically, every entity is given a *private name domain*. This is accomplished by associating the entities to the systemwide unique identifiers of "their" name servers. That is, every entity may possess a private namer server. Since name servers are working on disjunct name planes, each one can freely define and use names. If an attempt is made to gain register the same name in a name plane, a registration fault is signaled. By allowing entities to share name planes, inter-entity communication becomes possible, which means associating a number of entities with the same name server or with the same name server ensemble. These entities are then associated with the same systemwide unique name server identifier. A typical case is to let tasks (i.e., entities) of the same application share name servers (Figure 5.14).

In the scenario shown in Figure 5.14, the process manager, for example, provides services only to the four entities (i.e., the three user tasks and the file manager) situated at the upper left corner. Together, they constitute a specific application (refer also to Figure 5.12). The process and file manager then provide application-oriented services that are only visible in a certain scope of application. The five entities of this application also have access to services provided by the disk, page and host manager entities located at the bottom. The upper right part of Figure 5.14 shows a second application, which also has its own file manager and, additionally, its own disk manager. In total, there are three different name domains, whereby the name domain situated at the bottom of the picture is shared by the other two. This "system domain" provides commonly used operating-system services, that is, services which can be shared amongst the different applications.

Name domains are built by associating either user or system entities with name servers. The association is made dynamically. For example, incrementally loading the process manager may entail the installation of a new name server. This name server is then referred to by means of the intra-domain link kept by the name server of the user task that caused the entity fault. The new name server acts as a registry for

server forwards the original name porter request to the next name server. This reduces porter/server interactions to an absolute minimum and keeps system load low during the name lookup procedure.

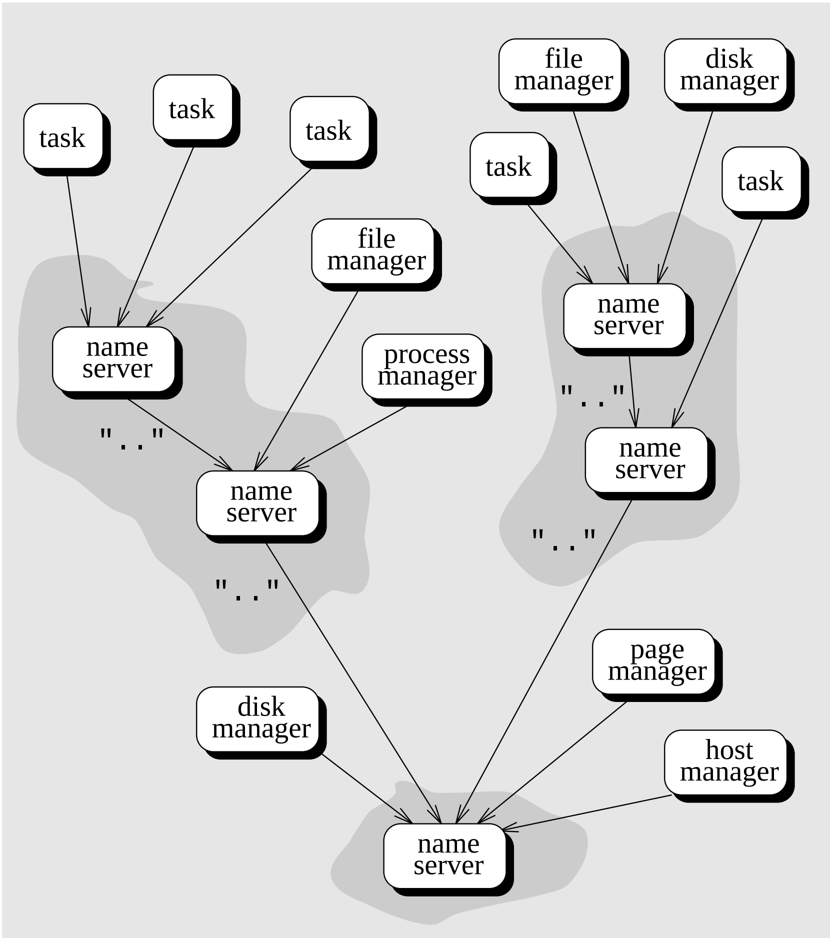

Figure 5.14: Problem-oriented name server ensemble

the services provided by the process manager and—the most common case—may also act as a registry of all application-specific services. Basically, the process manager name server is inserted into a name server chain. It carries over the intra-domain link definition of the user task name server and instructs that server to update the entry properly. Therefore, name scopes within a single name domain are established. Name scopes are used to separate different domains, that is, to separate user-defined names from system-defined names. A newly installed system entity is thus enabled to export its service names without the risk of name clashes.

5.3.7.3 Functionally Dedicated Entities

Name space construction becomes slightly more complicated when providing site-dependent services by dedicated entities, specifically when the nodes are shared by different applications. A typical example is the kernel entity. The kernel provides fundamental services in order to create/destroy active objects. These services must be available on each node that is going to be multiplexed between several tasks. Moreover, the service names must be registered by some name server. If the node is subjected to a single-user mode of operation (i.e., shared by tasks belonging to the same application), the same name server can be used to store both kernel-level and user-level names. Two name servers would separate the different name scopes. Name server interconnection may then take place as described above, namely on the basis of the intra-domain link (Figure 5.15).

The figure shows a single domain, combining both user and system name planes. In particular, replicated and code-identical kernel entities can be easily integrated without causing name clashes. The name scopes of each of these kernel entities are disjunct, and the intra-domain link defines a unique path through the global name space.

The situation changes if nodes are allowed to be run in a multi-user mode of operation. This implies that a node will be shared by tasks belonging to different user applications. Note that this is different from the multi-user mode of operation at the machine level. In the latter case, different user partitions are usually created, whereby each of the partitions runs the nodes in a single-user mode, leading to a name space organization as shown in Figure 5.15. Now, if tasks of different user applications share a single node, the question arises as to what happens with names denoting kernel services. Each application defines a unique naming domain and has at least one name server. Kernel names, however, cannot be easily stored in these application-specific name servers. The problem is how to tell the kernel entity into which name domain names are to be exported.

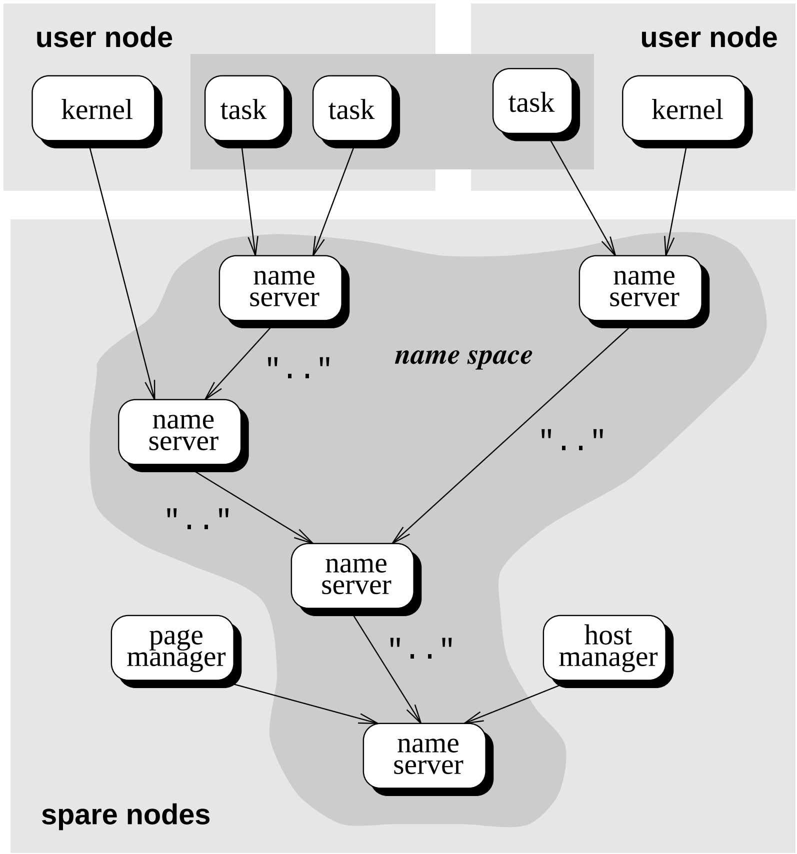

Figure 5.15: Name space structure for single-user mode of operation

The solution to this problem is to give nodes that are hosting site-dependent entities and run in multi-user mode their own name servers. Note that this does not mean giving every node its own name server: not every node is equipped with a kernel, and kernels neither provide the same set of service functions nor are required to export services at all (see also Chapter 3). These node name servers are referenced by all entities of the node. In addition to node-related service names, these servers also store *inter-domain links* to application name servers in order to distinguish between different user domains. Thus, user domains are united by binding to the same name server and divided by means of the inter-domain link.

For obvious reasons, "`..`" is not a good choice as the inter-domain link name. The name must be different for different applications and is thus an application-dependent character string. Since each task must have a unique ancestor, the systemwide unique identifier of the "root task" is used for this purpose. This identifier is converted into a character representation, associated with the unique identifier of the application name server and registered by the node name server. Name clashes are omitted (Figure 5.16).

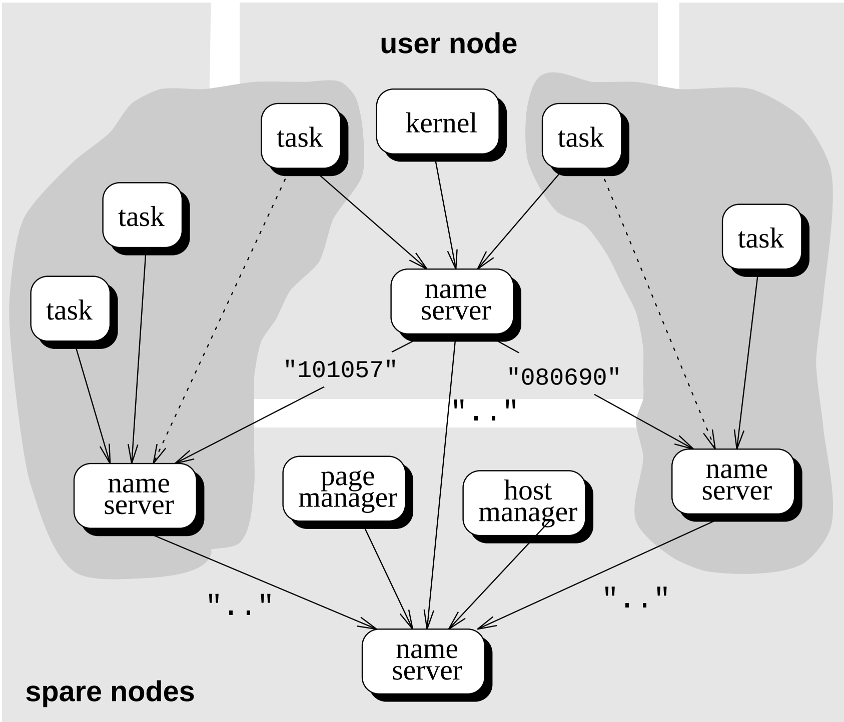

Figure 5.16: Name space structure for multi-user mode of operation

This approach calls for a slightly different name lookup procedure than discussed earlier. The first name server selected is the name server to which the entity that requests name resolution is attached. So far, there is no difference except that the node name server is selected in this case. If the name is not found within the scope of the node name server, the search continues by querying the name server pointed to by the inter-domain link of the requesting entity. This link addresses the first domain-specific name server. Within the same domain, the search continues by following the intra-domain link. The same occurs if an inter-domain link is not defined. In that case, the name-requesting entity is assumed to belong to the domain of the node name server.

An inter-domain link (i.e., the root task name) is stored in the environment of every entity and takes the form of a UNIX-like *environment variable*. This variable is set up by the loader and extracted by the name porter. Hence, a generally applicable lookup procedure is made feasible. Note that this sort of name server configuration applies only when tasks of different applications are allowed to share the same node. This is not a common approach, even in small-scale distributed-memory parallel machines. Anyway, even sharing, if enabled, does not raise security problems. Name servers can only be asked for a symbolic identifier presented by the requesting entity. Thus, this identifier must already be known. The inter-domain link, however, is the character representation of a systemwide unique identifier, and guessing these identifiers is almost impossible. Above all, it depends on the random number generator. The only chance is to try any possible number and submit it as a character string. This trial-and-error approach has hardly a chance to succeed, since ROI is to be used to query the name service. ROI, however, is orders of magnitude slower than LOI. Consequently, it becomes highly probable that a successful guess will not occur during the lifetime of the application. In any event, security problems can be avoided by *encryption/decryption* of the inter-domain link. However, this may only happen on a per-application basis, that is, when required by the particular application field.

5.3.8 Search Space Selection

As discussed, name resolution starts with the name server attached to the entity that asked for a certain service function. For this purpose, the PCB provides space to hold a single systemwide unique name server identifier. Thus, the name porter must simply ask for the delivery of this PCB instance variable. The request is carried out on an ROI basis, since the instance variable must be accessible from other nodes also. A kernel clerk (more specifically, a *domain clerk*) handles the ROI. This raises the problem of how to address the domain clerk. Note that ROI uses name servers if name resolution needs to be done and the domain clerk is used to make name servers known.

The solution to this addressing problem must also work in configurations in which name servers are initially loaded as part of the bootstrapping procedure. Therefore, name server instantiation is discussed first. This will reveal that, because of the bootstrapping case, the addressing problem becomes more serious.

5.3.8.1 Name Server Instantiation

The typical scenario for a server is to initially associate its systemwide unique identifier with one or more service names and announce these symbolic identifiers accordingly. A name server communication endpoint address is required in order to contact the service name registry. This address, however, is only known at runtime, namely, when name server initialization is finished. Name servers are normal processes. In particular, they all are code-identical (i.e., instances of the same abstract data type). Becoming aware of a name server address is free if the name server is created along with the server that is attempting to announce its service names. If name server creation is not feasible, because dynamic process management is not (yet) supported by the system, name servers must be bootstrapped. Name server initialization then requires the following:

- Requesting from the domain clerk the assignment of a systemwide unique identifier to the appropriate instance variable of all active PCB objects on the node. The identifier is typically the communication endpoint address of the requesting name server. This request succeeds only once, namely, when the first name server issues the request. All other name servers receive the systemwide unique identifier of the name server that succeeded in the PCB setup procedure.

- If the PCB setup was not successful, the name server tries to create a "`..`" entry by requesting the name server whose communication endpoint address was returned by the domain clerk. A name server installs a "`..`" link only if not yet defined. Otherwise, the request is forwarded to the name server associated with the existing "`..`" link entry.

The procedure described above results in the creation of a list of bootstrapped name servers (Figure 5.17). A name server identifier can thus be obtained by requesting the domain clerk to deliver the PCB name domain instance variable associated with the kernel entity. Basically, this variable is the head pointer of a name server chain. As part of the global system initialization procedure, a *name usher*[22] reads the head pointer from the kernel, removes the tail from the chain and remembers the tail as a chain of available name servers. The name server identified by the head pointer becomes the node name server.

In fact, the domain clerk also plays an important role during name server initialization. Obviously, the domain clerk cannot be made known by exploiting the name service, but must be known in the first place to install the name service. A name server is able to apply a local lookup procedure on its own name plane. However, the question arises, in that case, as to what system component requests the registration of the domain clerk service names. The name server address required for registration could be obtained by that "magic" instance by requesting the domain clerk to deliver the appropriate communication endpoint address. Again, the domain clerk address must be known for this purpose. Thus, the problem is only shifted away from the name server to the "magic" instance, but has not really been solved.

A fixed, systemwide unique communication endpoint address is needed to reach the domain clerk. More specifically, a number of different addresses must be provided, namely, one address for each node equipped with the kernel entity. The domain clerk address must be computable by the name porter and the name server.

[22]The name usher is another component of the *name administrator*. A name administrator additionally consists of all name servers and name porters.

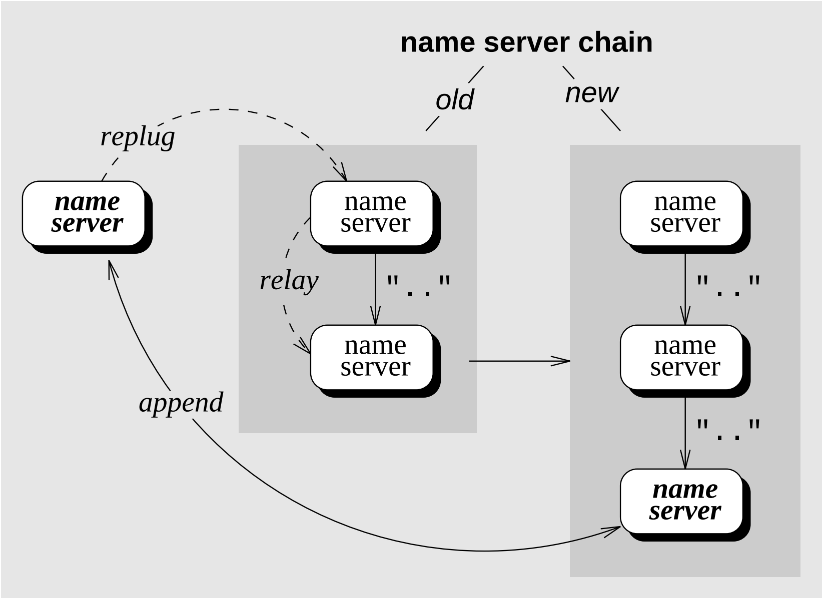

Figure 5.17: Chaining of name servers

5.3.8.2 Domain Clerk Addressing

The domain clerk is a normal thread. Therefore, its default communication endpoint address is not known in advance. However, threads may have more than one communication endpoint address by using forwarding identifiers. Every nucleus initially creates a *root gate* whose associated systemwide unique identifier can be computed. The LUI part of this identifier has the value {`site,0,0`}, with `site` representing the logical node number where this gate is stored. The sparse field of this identifier is also zero. Thus, all that the name porter needs to know is the number of its hosting node. The domain clerk, if present, attaches the root gate and lets the forwarding identifier point to its own communication gate. Every communication request sent via the root gate is thus automatically forwarded to the domain clerk. The root gate address is the only fixed and computable communication endpoint address in the system.

In reality, the name porter does not have to know the precise root gate address. The nucleus provides a service function, `gateway()`, which delivers the root gate address corresponding to a systemwide unique identifier. An opaque identifier is passed back to the caller. This identifier is then used to direct the ROI to the domain clerk. The name porter thus determines its systemwide unique identifier, which, in turn, is used as an argument of `gateway()`. Fixing its own unique identifier is enabled by the `address()` nucleus call. Again, an opaque identifier is returned.

The actual initialization of the root gate depends on whether the domain clerk is locally or remotely installed. That is, the initialization procedure defines a forwarding identifier to a local or a remote thread. If the domain clerk is locally present, a kernel component must be running on the node (case a of Figure 5.18). The root gate thus contains a local forwarding identifier. ROI to the domain clerk via the root gate involves no network activities, but rather causes the change between two protection domains sharing the same node. The domains are the user entity and the kernel entity.

In the second configuration, the domain clerk is physically separated from the name porter. This means that the domain clerk was allocated to a different node. As a consequence, a kernel component need not be present locally except when demanded for other reasons (device handling, process and address space management, etc.). Only the nucleus is required. The root gate then stores a remote domain clerk reference (case b of Figure 5.18).

These two configurations are transparent to higher-level entities. Interaction with the domain clerk is always carried out on an ROI basis and by applying the same communication endpoint address denoting the root gate. If a local domain clerk is provided, that clerk takes care of installing the forwarding identifier. The initialization sequence of the nucleus makes the necessary definitions if the domain clerk resides in a remote location. In that case, the bootstrapping procedure sets up a nucleus environment variable that contains the communication endpoint address of the remotely situated domain clerk. After having been bootstrapped, the nucleus reads this environment variable and installs it as the forwarding identifier.

A remote domain clerk will typically be a single thread of a *domain*

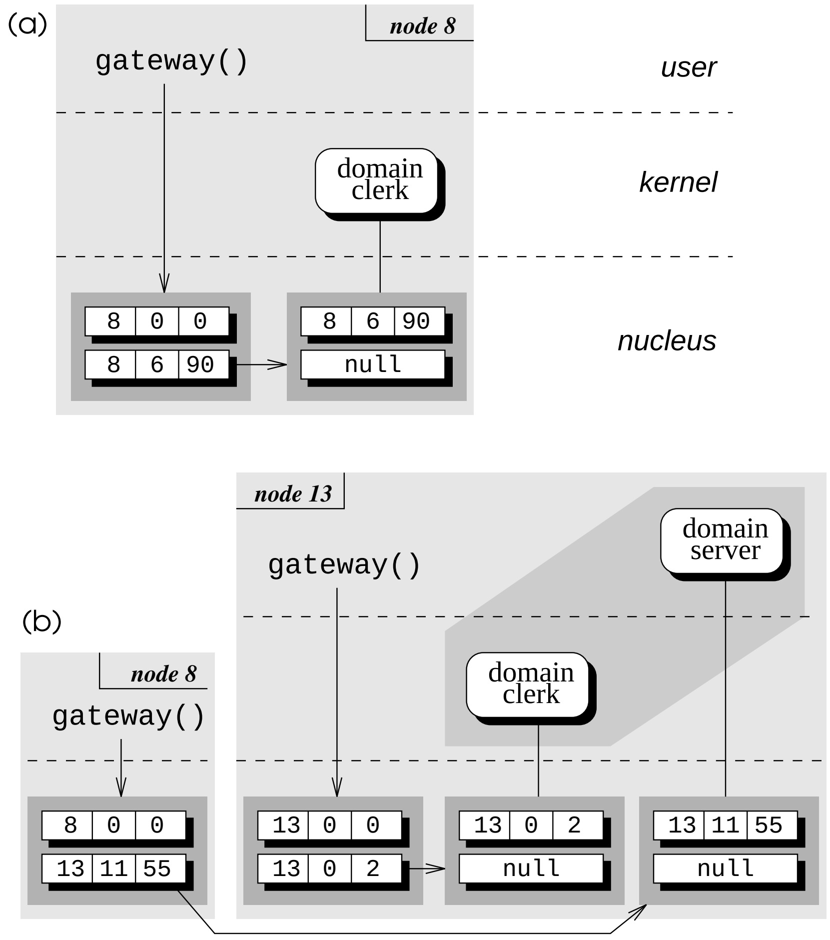

Figure 5.18: The root gate as domain clerk forwarding identifier

server. Thus, the domain server is a multithreaded entity, with each thread being a domain clerk for a node. The domain server keeps track of communication endpoint addresses which refer to node name servers of remote nodes. Name servers in general, and node name servers in particular, may reside anywhere. Node name servers are only logically associated with a specific node and need not be physically present on that node. In addition to the "user-level" domain server, configuration b of Figure 5.18 also shows a kernel-level domain clerk. This domain clerk plays the same role as the domain clerk of configuration a. It is responsible for maintaining the per-process name server communication endpoint addresses of all collocated entities. The system node (node 13 in Figure 5.18) thus hosts a *domain administrator*, with domain server and domain clerk as the two distinguished manager instances.

5.4 Summary

This chapter presented basic building blocks of a dynamically alterable parallel operating system. The fundamental idea is to understand operating-system services as *transient objects*. Transient objects are loaded on demand and removed when no longer needed.

A discussion of *incremental loading* as the means of establishing application-oriented operating-system structures at runtime was given. By the same pattern, *incremental bootstrapping* is employed to get a parallel machine started without the risk of large delays caused by global system initialization. Nodes are put into operation only when required by the given parallel application(s).

Object localization and naming is the dominating issue in maintaining a family of parallel operating systems and supporting the model of transient objects. Objects have low-level addresses which are used to facilitate systemwide communication. These are referred to as *systemwide unique identifiers*, which encode a hint as to the node where the addressed object will probably reside. High-level addresses are used to introduce transparency. These addresses use *symbolic identifiers* to denote abstract data types, classes, objects and/or operations. Thus, symbolic identifiers are used to identify almost anything that could be of interest to applications.

The naming scheme presented does not rely on one-to-many communication (i.e., multicasting or broadcasting). These techniques are not suited to properly supporting operating-system families. The naming scheme relies on a set of name servers that jointly maintain a *physically fragmented name space*. Name servers are allocated on a per-process basis. They are used to establish a *name domain*, which implements a name space for a given application. Different applications are related to different name domains. Name servers of the same domain build a chain of *name planes*, whereby each name plane ensures a unique mapping from symbolic to systemwide unique identifiers.

Mappings made by name servers are completely user-defined. Therefore, the symbolic identifier is represented by a character string, and the systemwide unique identifier is represented by a byte string. Every name server implements "associative memory" that translates character strings into byte strings if the mapping function is defined. The naming scheme is applied by the ROI layer, but is also designed to be applicable by the user tasks of the same distributed application.

Name servers are accessible via ROI, too. For this purpose, specific mechanisms are provided at the nucleus and kernel levels to help identify per-process name servers. The model assumes that name servers can reside elsewhere. In particular, they are not forced to share the same node with the tasks to which they are attached. This decision was made to support user nodes that are assigned solely to process application tasks. Thus, the naming system allows nodes that are only controlled by the nucleus to be safely integrated into a distributed environment.

Chapter 6

Class Ensemble

As mentioned in the introduction, object orientation is somehow the natural answer to the question of how to properly implement program families [21]. The minimal subset of system functions can be best implemented as one or more superclasses. Subclassing (i.e., inheritance) is suited to implementing minimal system extensions. Both inheritance and (function) polymorphism ease the implementation of a system in which different representations of the same abstract interface are allowed to coexist.

In the subsequent sections, fundamental class hierarchies are discussed. These classes serve to implement the basic building blocks of which a family of parallel operating systems for distributed-memory architectures is constructed. The object-oriented design and/or implementation of the kernel entity is presented. More precisely, the focus will lie on the minimal basis used to construct nucleus and kernel. Except for a few examples, hardware-dependent modules such as device drivers are not considered. Mainly, the portable part of the kernel entity is discussed. Nevertheless, the presentation begins at the conventional machine layer and moves towards the user/system interface. The aim of this bottom-up course is to demonstrate by what means a non-object-oriented physical processor can be transformed into an object-oriented abstract processor.

Being familiar with the principles of object orientation will be very beneficial in order to understand most of the issues addressed in this chapter. The discussion is on a very technical level, describing a fairly large number of classes and giving the rationale for the chosen structure. This also presents an experiment in *pure object orientation*. Every feature the minimal basis provides is encapsulated by a class. The only exception to this approach is made when introducing CPU abstractions that are used to bridge the gap between the non-object-oriented and the object-oriented world.

Before starting with the discussion on the class ensemble implementing the minimal (but perfect) subset of system functions, some general and clarifying remarks concerning the class hierarchies presented in this chapter are necessary. Inheritance relationships are expressed by upward-pointing arrows. In contrast, reference or pointer relationships will be expressed by either downward-pointing straight arrows or curved arrows. Inheritance is not applicable in every case.[1] Where required, both reference and inheritance relationships are used together.

Another important aspect is how inheritance is technically carried out by the programming language or the compiler. The class hierarchies shown assume the modeling of inheritance trees on the basis of *multiple inclusion*. That is, when two subclasses inherit the same superclass, and both subclasses, in turn, are inherited by another class, objects of the latter will contain two instance variables of the superclass. Logical relationships between classes, not between objects, are discussed.

6.1 Basic Abstractions

Fundamental abstractions used to build kernel-level software deal with the modeling of dynamic data structures and prologues and epilogues for event synchronization (see also Section 4.1). Both aspects are discussed in the following subsections.

6.1.1 Queues, Stacks and Chains

Roughly speaking, the functions of an operating system kernel are closely related to the processing of dynamic data structures. Nearly ev-

[1] For example, to include abstract interfaces, introduce logical boundaries, model mutual relationships, or when making transformations between non-object-oriented and object-oriented levels of abstraction.

ery kernel object is remembered in some kind of (single-/double-linked) list. Sometimes objects are held in more than one list at the same time. Typical examples are waiting lists, hash lists and resource lists. Thus, queues (FIFO lists), stacks (LIFO lists) and other kinds of object chains are definitely one of the most fundamental abstractions to be provided (Figure 6.1).

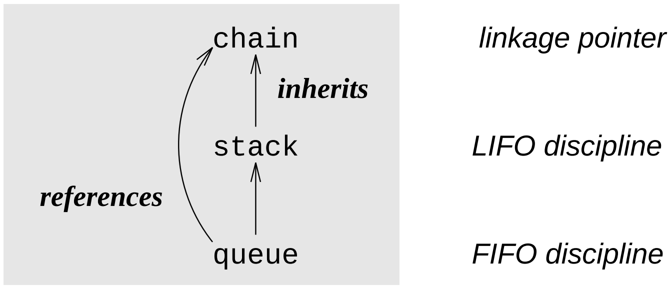

Figure 6.1: Queue abstraction

As shown, a `queue` "is a" `stack`, which "is a" `chain`. Queues and stacks thus do not only describe lists, they can also be kept in lists. In other words, this technique allows, for example, the treatment of list fragments as stacks and the application of stack operations such as `push()` and `pop()` to manipulate these fragments. This also holds when representing list fragments as queues, that is, having the tail pointer refer to the queue descriptor (which "is a" `chain` element) of the subsequent queue.

The `chain` abstraction implements a linkage pointer and nothing else. The linkage pointer refers to objects of type `chain`. Any object of a class derived from `chain` possesses the property of being generally queuable. A `stack` is a specialized `chain`, introducing the LIFO queuing discipline. This class adds methods only. The `queue` abstraction adds methods and one more instance variable. The methods introduce the FIFO queuing discipline, and the instance variable is a pointer to a `chain` pointer and denotes the list tail. This pointer address is described by the reference relationship between `queue` and `chain`. The list head pointer is represented by the `chain` element included through inheritance. This element also serves as "stack pointer" for the `stack` abstraction.

This approach to modeling a queue is a simple example of how object orientation can be used to share common, and introduce new, functionalities, which is an inherent property of program families. The `queue` abstraction shown provides two queuing disciplines, namely, LIFO and FIFO. It inherits the LIFO discipline from `stack` and adds a FIFO discipline. In addition to that, the specific tail pointer representation was chosen to ease and speedup the enqueue operation.[2] The queue implementation proposed makes the tail pointer refer to the linkage pointer of an element [63]. Moreover, the tail pointer is initialized to point to the head pointer. This makes element insertion straightforward and requires no check for the head pointer. The tail (or initially, head) linkage pointer which is referred to by the tail pointer gets assigned the address of the linkage pointer of the element that is queued. Then the tail pointer gets assigned the same address. Note that this automatically advances the head pointer when the first element is queued.

6.1.2 Event Synchronization

The synchronization model of asynchronous events (i.e., interrupts) is extensively discussed in Section 4.1. Here, the classes and class hierarchies used to implement a pair of prologue and epilogue objects, termed in Section 4.1 an *activity object*, are explained. The modeling of an activity object can be obtained from Figure 6.2, part a.

A prologue object is implemented by the `event` abstraction and an epilogue object is implemented by the `annex` abstraction. Both classes are derived from `chain`, which makes prologue and epilogue objects queuable. Both `event` and `annex` are *deferred classes* [84]. The `event` abstraction introduces the *pure virtual function* `prologue()`,

[2] A straightforward and very common queue implementation makes the tail pointer refer to the last queue element. This causes every enqueue operation to check for the initial state of the head pointer. However, only when the first element is placed on the queue must the head pointer be initialized to point to that element as well. As a matter of fact, the check for the initial state in most of the situations is a typical case for the execution of an unneeded function.

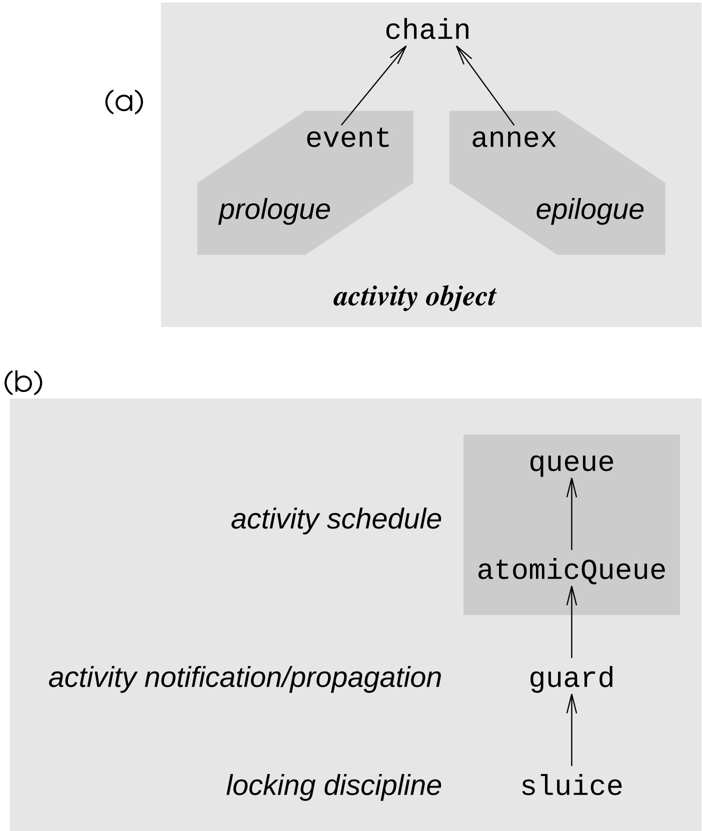

Figure 6.2: Abstractions for prologue and epilogue handling

while `annex` introduces `epilogue()` as a pure virtual function.[3] The `epilogue()` entry is called whenever epilogue processing of that activity object is granted (see Section 4.1 for more details). By means of `event` and the `prologue()` entry, the polling of interrupt handlers is

[3]A *virtual function* [84] is a method subject to dynamic binding and to having a default implementation. A *pure virtual function* is a virtual function without implementation. Classes containing pure virtual functions are *deferred classes*. These classes must be derived and specialized in order to enable object instantiation. Specialization then means to provide implementations of the pure virtual functions specified at the superclass level.

made feasible. Asynchronously executing prologue objects can be placed onto a "call-out queue" and are activated by invoking `prologue()` when a certain condition (e.g., clock interrupt) holds.

The model of interrupt synchronization is defined by the `sluice` abstraction (part b of Figure 6.2). This abstraction specifies the locking discipline, that is, whether software locking, hardware locking or no locking at all is performed. The main work of maintaining the schedule of pending activities is done by `guard`. An activity schedule is implemented by an `atomicQueue`. The latter ensures the integrity of `queue` manipulations even in a shared-memory multiprocessor configuration.

The `guard` introduces specific queuing disciplines and takes care of activity notification and propagation. It provides the methods `enter()`, `leave()` and `relay()`. These methods are redefined by `sluice` to reflect the various locking variants. The nucleus applies a `sluice` instance for synchronization purposes, and thus is easily adaptable to the different styles of locking. In the software locking case, `sluice` is basically empty, that is, it maps all its primitives directly on the `guard` primitives. If one must take care of a shared-memory multiprocessor context, `enter()` performs spin-locking. Note that software locking means to advance the `counter` instance implementing the lock variable. The spin-lock implementation is a loop in which the lock variable is atomically incremented and then atomically decremented if the counter value indicates that the update was performed several times (by competing processors). The `leave()` implementation decrements the lock variable atomically. Pending epilogues are propagated by `relay()` as provided by the `guard` object. In the hardware locking case, the `sluice` implementation of `enter()` disables, and the implementation of `leave()` enables, all interrupts. In that case, `relay()` always invokes `epilogue()` since no other nucleus activity could have been interrupted. If no locking is performed, `enter()` and `leave()` are empty, and `relay()` behaves as in the hardware-locking case.[4]

[4]The methods `enter()`, `leave()` and `relay()` should not be represented as virtual functions. Performance is the main reason for this decision. Invocation of virtual functions is about four times slower than that of nonvirtual functions (depending on the processor). This becomes a serious performance problem when the functions execute only a handful of instructions, which is the case with these `guard` methods.

In many cases, *class-based object orientation* [10] is used in the implementation of the kernel entity. This means that most of the methods are taking the form of *inline functions*. The methods provided by the basic abstractions described so far are all directly implemented in the class definition. They can all be regarded as "typed" macros. Consequently, the resulting nucleus code is flat concerning these classes, although the source code implementation is well structured. Note that "*it is the system design which is hierarchical, not its implementation*" [36]. Applying class-based object orientation where performance is the predominant issue, as in the nucleus case, results in extremely efficient implementations without sacrificing a highly modular system structure.

This is one of the virtues of C++, namely, that the system programmer is able to explicitly instruct the compiler about where to take care of runtime efficiency at the cost of a "slight" increase[5] of code size. This argument does not hold entirely in the presence of modern RISC processor architectures, where function calls are more or less free. However, making all methods external for this reason would mean making assumptions that will no longer hold when the software is to be ported onto "old-fashioned" CISC processors. Similar considerations hold when trying to develop software whose user acceptance should not be constrained by the compiler used for translation into executable code. State-of-the-art compiler technology also automatically takes care of function inlining where appropriate, but not every (especially C++) compiler on the market provides these features. Having method implementations within the class specification is, at least when regarding C++, the best compromise. However, this applies only to functions that are mostly one or two lines of C++ code.

---

[5] However, the code size increase must not be underestimated. For example, only making all methods of the `queue` abstraction and their ancestors external (i.e., sacrificing class-based object orientation) resulted in a decrease of the kernel text segment for the T800 port of PEACE of about 12% [93]. The code size problem, however, was mainly due to the compiler employed. Complex methods, especially those containing loops and interlocked conditions, were not inlined but rather coded as static functions of every compilation unit that includes the corresponding class definition. The PEACE kernel consists of a very large number of these compilation units.

6.1.3 Lock Variables

The previous subsection mentioned that, under certain circumstances, `sluice` performs spin-locking in order to synchronize parallel kernel activities in a shared-memory multiprocessor context. In the following subsection, the modeling of lock variables and the implementation of a simple spin-locking strategy are presented.

First of all, the lock variable is assumed to be a counter that can be incremented to activate the lock and decremented to deactivate the lock. This suggests a `counter` abstraction containing an instance variable of type `int` and defining a number of unary and binary operators to manipulate and verify the object state. Some of these operators must be subjected to *mutual exclusion*. This feature is introduced through subclassing, that is, an atomic lock variable update is considered as a specialization of the `counter` abstraction. The specialization is implemented by an `atomicCounter` class. This class redefines the critical `counter` operators and ensures atomicity.

Once the abstraction for an atomic lock variable has been designed, synchronization strategies may be added as a further specialization of `atomicCounter`. An example is the implementation of spin-locking. A `hurdle` class may be derived from `atomicCounter`, introducing `enter` and `leave()` as the synchronization primitives. The evolving class hierarchy is presented in Figure 6.3.

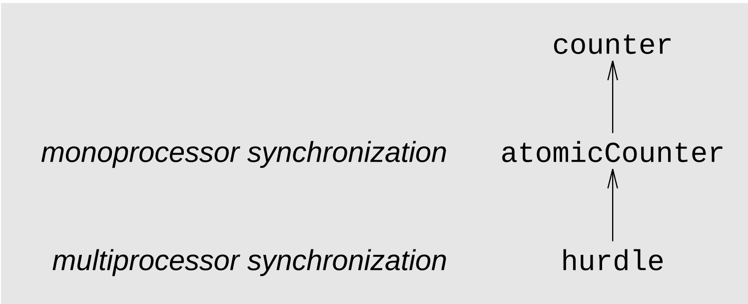

Figure 6.3: Abstractions implementing lock variables

In the class hierarchy shown, `atomicCounter` is a more or less processor-dependent class. More specifically, it typically employs low-level (i.e., C) routines implementing the atomic update of an `int` variable. This variable is the `counter` instance variable. By relying on an atomic increment/decrement, spin-locked synchronization schemes may be implemented as demonstrated in Figure 6.4.

```
 1: #include "atomicCounter.h"
 2:
 3: class hurdle : public atomicCounter {
 4: public:
 5:    hurdle () : atomicCounter(-1) {}
 6:
 7:    void enter () {
 8:       while ((*this)++ >= 0) {
 9:          (*this)--;
10:       }
11:    }
12:
13:    void leave () { (*this)--; }
14: };
```

Figure 6.4: A simple spin-locking strategy

In this implementation, releasing the lock (line 13) is a trivial task. Method `leave()` simply (atomically) decrements the lock variable. This gives other threads (or processors) the chance to successfully pass the `enter()` implementation (lines 7 to 11). As shown, `enter()` keeps the invoker checking the lock variable if the (atomic) increment indicates that at least one more increment has been performed previously. That is, `enter()` blocks the invoker if the corresponding `leave()` was not yet executed. The loop (lines 8 to 10) increments and decrements the lock variable atomically. Thus, in effect the lock variable state remains unchanged. The first invoker of `enter()` passing the loop is the winner (i.e., holds the lock) and is assumed to call `leave()` some time.

The counter is initialized with -1 (line 5). Thus, entering a critical section leaves behind a counter value of 0. As line 8 shows, `enter()` loops as long as the counter has a positive value, meaning that the critical section is still active. At that time, the counter value gives an estimation on the number of threads attempting to acquire the lock.

6.1.4 Critical Sections

The nucleus performs locking of critical sections by exploiting the `hurdle` and `sluice` abstractions. This makes the nucleus largely independent of the actual locking technique used and eases configurability. Nevertheless, the main problem remains, namely, to identify in the code where to set the locks and, even more important, to guarantee the release of locks after having left the critical section. Once having identified the critical section, from the programming point of view, placing the lock is much easier than releasing the lock. This is because critical sections may be encapsulated by functions exhibiting multiple exit points. In fact, the nucleus is only concerned with the problem of properly marking a critical section as active by setting a lock. Lock release is automatically performed by applying the class shown in Figure 6.5 and a specific C++ feature.

```
1: #include "locker.h"
2:
3: class secure : public locker {
4: public:
5:     secure () { enter(); }
6:    ~secure () { leave(); }
7: };
```

Figure 6.5: Lock abstraction used by the nucleus

The `secure` abstraction provides a constructor and destructor method. As line 5 makes clear, constructing a `secure` object entails the setting of a lock. Similarly, destructing such an object (line 6) implicitly causes the release of the lock again. In C++, class constructors are automatically invoked by the runtime system when objects of these classes are declared. Destructors are automatically called when objects are deleted. Object deletion may be explicit or implicit. Dynamically allocated objects usually must be explicitly deleted when no longer used. The language defines no garbage collection. Statically allocated objects or objects declared as local variables, in contrast, are automatically destroyed when the scope of their declaration is left.

From the perspective of the nucleus, this means the following: A `secure` object is locally declared within the scope defined by a nucleus primitive that defines a critical section. As soon as this primitive is left, the C++ runtime system calls the destructor for that object. Thus, a typical declaration pattern is {`secure nucleus; ...`}, with { opening and } closing the critical section. Any return from this block leads to the implicit invocation of the `secure` destructor. As a matter of fact, a critical section syntactically may correspond to any sort of C++ block.

Locking and unlocking is carried out by `enter()` and `leave()`, respectively. Both methods are implemented by the `locker()` class, as shown in Figure 6.6. The `locker` class is inherited by `secure` to implement locking constructor and unlocking destructor operations (refer to Figure 6.5).

```
 1: #include "hurdle.h"
 2: #include "sluice.h"
 3:
 4: extern hurdle bolt;     // multiprocessor lock
 5: extern sluice hall;     // pro/epilogue lock
 6:
 7: class locker {
 8: public:
 9:    void enter () { hall.enter(); bolt.enter(); }
10:    void leave () { bolt.leave(); hall.leave(); }
11:
12:    void check () { leave(); enter(); }
13:};
```

Figure 6.6: Basic abstraction to secure critical sections

The `enter()` method (line 9) first acquires a local *epilogue lock* (`hall` instance) and then a global *processor lock* (`bolt` instance). In contrast to the processor lock, the epilogue lock has only local significance. The epilogue lock is used for per-processor interrupt synchronization (see also Section 4.1), whereas the processor lock serves to synchronize the activities carried out by several processors in parallel. Unlocking the critical section is done by `leave()` (line 10). First, the multiprocessor lock and then the epilogue lock are released. This prevents deadlocking from initiating processing of pending epilogues with the processor lock still on. Releasing the epilogue lock (by `hall.leave()`) may cause the immediate propagation of pending epilogue requests. Epilogue processing in a multiprocessor environment means acquiring the multiprocessor lock before epilogue activation takes place. Consequently, if the multiprocessor lock has not been released before the epilogue lock, a deadlock will occur.

The `locker` abstraction also provides a method used to mark a *preemption point*. This method is `check()` (line 12). As shown, the sole function of `check()` is to leave and (re-)enter the critical section. With `leave()`, possibly pending epilogue requests are processed (see also Section 4.1). By that means, "interrupt handler continuations" can be started and are allowed to proceed by executing critical nucleus code. For example, this may result in message delivery to a thread and/or thread scheduling.[6] Similar to the (possible) activation of interrupt handler continuations (i.e., epilogues), in a multiprocessor environment, `check()` gives other processors the chance to enter the nucleus.

The `check()` method is invoked when the currently executing thread processes loops inside a critical section. These loops may occur, for example, when the hash function is performed in order to locate the data structure that models a thread. Whenever a forwarding identifier is detected, which entails another run of the hash function, `check()` is invoked to avoid too-large locking delays.

6.2 Processor Abstractions

The abstractions discussed here in detail are dealing with the propagation and handling of hardware exceptions. These exceptions are *traps* and *interrupts*. Since kernel-level exception handlers are (active/passive) objects, a transformation from non-object to object level

[6] Note that `check()` does not activate the thread scheduler. It merely temporarily suspends further processing of a critical section and allows interrupt handlers to enter the nucleus. Entering the nucleus may entail the activation of a thread scheduler.

is necessary. The mechanisms implemented by the presented class ensemble are exemplified by presenting the design of the timer interrupt handler and the VICE trap handler.

6.2.1 Propagation of Hardware Exceptions

In state-of-the-art processor technology, low-level exception handlers are addressed via a program counter value only. There is no means to implicitly provide the handler with a specific data set on which the handler should operate. That is, hardware support suitable for describing per-handler state information, similar to *own variables*, is typically lacking. A trap/interrupt vector table is the usual way of mapping hardware exceptions onto trap/interrupt handlers. In order to handle these exceptions, the kernel stores the start code addresses of the handler routines in vector table entries. Vector table entries correspond to certain hardware exceptions.

The program counter value stored in a vector table entry is typically the start address of a *wrapper routine* the only purpose of which is to save/restore the volatile register set and invoke the original exception handler (see also Section 4.1). If the programming language employed does not provide adequate support for low-level interrupt and trap handling, which is the case with C++, the wrapper routine must be written in assembly language. This makes the wrapper routine a nonportable part of the nucleus. However, the trap/interrupt handler is usually written in a problem-oriented programming language and may be portable. In order to support models in which trap/interrupt handlers are implemented by methods of specific objects, two bridges must be crossed. The first bridge is implemented by the wrapper routine, and the second bridge is implemented by the routine that gets called by the wrapper routine. Taking the bottom-up view, wrapper routines invoke trap/interrupt handlers directly. From the top-down (object) view, however, a routine called by the wrapper routine maps the call into an object invocation. This second bridge knows about an object that corresponds to a specific hardware exception. It applies this object and invokes a method representing a trap/interrupt handler.

6.2.1.1 Exception Handler Linkage

A further and very important aspect is to implement the trap/interrupt handler linkage procedure independent of the actual handler. The linkage procedure must be generally applicable for any type of hardware exception. Thus, the propagation of hardware exceptions to objects has to take care of the following tasks:

1. Specifying a wrapper routine
2. Setting up a "lifter routine" that gets called by the wrapper routine
3. Associating an object with each possible hardware exception
4. Performing *dynamic binding* to install the trap/interrupt handler

The last point listed enables abstraction from the actual trap/interrupt handler that is to be associated with a given hardware exception. Figure 6.7 shows the relationship between classes that implement the linkage procedure.

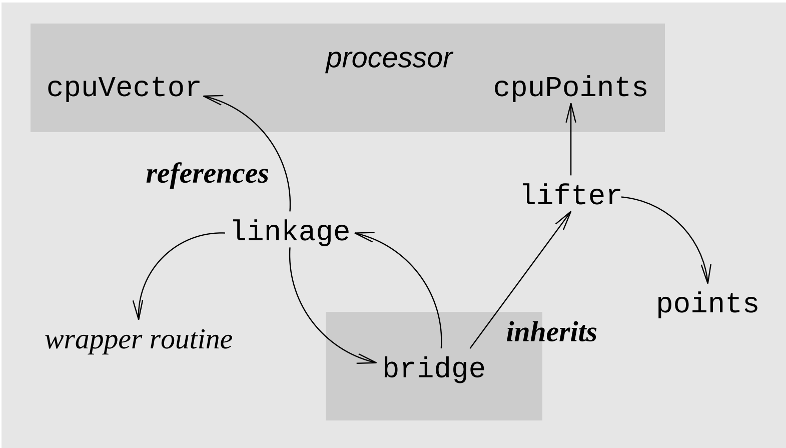

Figure 6.7: Translating traps/interrupts into object invocations

The `bridge` abstraction plays the central role. It helps to translate traps/interrupts into object invocations. A `bridge` object stores a pointer to a trap/interrupt handler linkage descriptor. This descriptor is defined by the `linkage` abstraction. As indicated in Figure 6.7, the linkage descriptor remembers the base address of a trap/interrupt vector table, the vector table index corresponding to the hardware exception (not shown), the start address of the wrapper routine and the pointer to an associated `bridge` instance. A trap/interrupt vector table is implemented by the `cpuVector` abstraction, and a single vector table entry is a `cpuPoints` object.

A `bridge` "is a" `lifter` and, by that means, provides information of how to map traps/interrupts onto objects. Furthermore, a `lifter` is the specialization of a `cpuPoints`, that is, of a trap/interrupt vector table entry. The `lifter` also introduces an instance variable that refers to a `points` object. The `points` abstraction defines a *deferred class* [84] and is used to dynamically bind trap/interrupt handler methods. These higher-level methods are thus responsible for "moving the points"[7] for problem-oriented trap/interrupt handling.

6.2.1.2 Trap/Interrupt Linkage Descriptor

The linkage descriptor is used to dynamically associate an object with a trap/interrupt. This descriptor actually consists of two objects: one of type `linkage`, and a second one of type `bridge`. A `linkage` instance stores all necessary information needed to program a trap/interrupt vector table entry. A `bridge` instance provides the coupling to the object dedicated to handling traps/interrupts.[8] In addition to these two

[7] *(Am.)* "shifting the switch"

[8] The reason for this separation was mainly dictated by C++ and not by the model. A linkage descriptor is assigned to store initial definitions made known at compile time. Class initialization, however, is only possible at runtime and depends on when the constructor of globally (i.e., statically) declared objects gets to be called. This makes the initialization of global objects nondeterministic and may cause trouble when a trap/interrupt is signaled before the constructor was called by the runtime system. The order of constructor calls for statically declared objects is not defined in C++. Therefore, the separation between `linkage` and `bridge` exists. More specifically, `linkage` is not a class but a structure only. Structure objects can be initialized at compile time.

data objects, a linkage descriptor also encapsulates the wrapper routine and a "lifter routine" used to propagate traps/interrupts to objects (Figure 6.8). The task of the lifter routine is to activate an object associated with a trap/interrupt vector table entry. The lifter routine is called by the wrapper routine.

```
 1: #include "assert.h"
 2: #include "bridge.h"
 3: #include "linkage.h"
 4:
 5: extern "C" void WrapLifterNOP();
 6: extern "C" void LiftPointsNOP(escape*);
 7:
 8: static  bridge gate;
 9: linkage pointsNOP = { 0, NOP, WrapLifterNOP, &gate };
10:
11: void LiftPointsNOP (escape* esf) {
12:    assert(**(lifter*)&gate);
13:
14:    (***(lifter*)&gate)(esf);
15:
16:    esf->verify();
17: }
```

Figure 6.8: Linkage descriptor for trap/interrupt handlers

Lines 8 and 9 of Figure 6.8 contain the declarations of the two objects just mentioned. Only the `linkage` instance is globally known. The `bridge` instance is referenced indirectly by means of the `linkage` instance. Lines 5 and 6 contain the compiler directives to make the assembly-level wrapper routine known to the C++ level and the C++-level lifter routine known to the assembly level. Compile-time initialization of a `linkage` instance means the following (refer to line 9):

- Definition of the vector table base address. The `null` pointer (i.e., 0) causes the linkage procedure to use the default table.
- Definition of the vector table index. The index number is specified by `NOP` ("number of points").

- Definition of the start address of the wrapper routine obtained from the `linkage` instance. The start address is specified by `WrapLifterNOP`.

- Definition of the linkage to the static `bridge` object.

The globally declared `linkage` instance is used by trap-/interrupt-handling subsystems for dynamically attaching them to entries of the trap/interrupt vector table. Logically, these subsystems are transient objects. For each source of hardware exception, its own `linkage` instance is used. When the trap-/interrupt-handling subsystem applies a `linkage` instance, the selected `cpuPoints` entry of the `cpuVector` instance is saved into the `cpuPoints` part of the `bridge` instance. Then the vector table entry is defined with a `cpuPoints` that was initialized with the address of the wrapper routine. The old vector table entry is restored when the trap/interrupt handling subsystem releases control.[9]

When a hardware exception occurs, the assembly-level wrapper routine passes over control to the lifter routine, which activates the object pointed to by the `lifter` instance. The lifter routine is specified in lines 11–17 of Figure 6.8. The `assert()` statement (line 12) merely ensures the presence of a valid linkage when the kernel is specifically configured, for example, to aid porting. Under normal circumstances (i.e., in a "mature" system configuration), `assert()` does not come into play when executing the lifter routine. Line 14 shows the statement used to activate the object currently used for exception handling. Effectively, the function call operator of class `points` is invoked.

The purpose of `verify()` (Line 16) is to take care of the activation of pending epilogues. This procedure is processor- and/or kernel-dependent. One approach is to manipulate the saved processor state in such a way that return from interrupt handling entails an *asynchronous*

[9]Note that this changes the main system trap/interrupt vector table. The mechanism presented is fundamental and, in correspondence with the family idea, does not try to solve the problem of multiplexing a single vector table entry between several trap/interrupt handlers. Solutions to this problem are implemented as extensions of the minimal basis. In particular, the solutions can be transparently integrated by employing specifically designed objects. In this case, the lifter routine (i.e., `LiftPointsNOP()` in Figure 6.8) simply activates an object that takes care of the multiplexing task.

system trap (AST) [69]. Note that the `escape` instance, applied to `verify()`, stores exactly this state. This makes processor state manipulation, and thus AST signaling, straightforward—unless the processor does not implement the AST feature. Handling the AST means epilogue propagation, but the trap is signaled only when pending epilogues are encountered. Another approach is to let the final active interrupt handler (out of a possibly stacked set of interrupt handlers) initiate epilogue propagation. In this case, it is the function of `verify()` to ensure that interrupts are enabled and check for pending events. Lowering the interrupt priority becomes necessary since the synchronization model (as discussed in Section 4.1) assumes epilogues processing with interrupts enabled, and the final interrupt handler executes on a certain interrupt priority. This second approach comes into play if the underlying processor does not support the modeling and handling of an AST.

For every wrapper routine there exists a corresponding lifter routine: a `WrapLifterNOP` is accompanied by a `LiftPointsNOP`. The formal argument of this routine is a pointer to the *exception stack frame* created by the CPU. This pointer is passed through to the object and can be used to access the nonvolatile processor state (see Section 4.1).

The problem-oriented programming-language-level lifter routine is called by the assembly-level wrapper routine. There are many wrapper routine templates, one implementation for each type of CPU, but only a single lifter routine template. The lifter routine is portable, and the wrapper routine is not. In order to exemplify how the linkage between the different levels of abstractions works and to get some idea about the overhead involved with trap/interrupt handling, a wrapper routine for the Motorola MC680x0 family is shown in Figure 6.9.

The naming convention of both wrapper and lifter routine was already demonstrated in Figure 6.8.[10] Lines 3 and 11 deal within saving and restoring the volatile register set. The nonvolatile register set is saved by the C++ runtime system once `LiftPointsNOP()` is being called (line 7). In lines 4 and 5, the user-mode stack pointer of the trapped/interrupted thread is saved onto the kernel stack. The user

[10]In the assembler used, the underscore denotes symbols that are also usable at the C/C++ level. In this particular case, `_WrapLifterNOP` corresponds to `WrapLifterNOP` and `_LiftPointsNOP` corresponds to `LiftPointsNOP` of Figure 6.8, respectively.

```
 1:    .globl _WrapLifterNOP
 2: _WrapLifterNOP:
 3:    moveml d0-d1/a0-a1,sp@- | save volatile reg.
 4:    movl   usp,a0           | save user sp
 5:    movl   a0,sp@-
 6:    pea    @sp              | pass esf pointer
 7:    jsr    _LiftPointsNOP   | enter C/C++ level
 8:    addql  #4,sp            | remove esf pointer
 9:    movl   sp@+,a0
10:    movl   a0,usp           | restore user sp
11:    moveml sp@+,d0-d1/a0-a1 | restore volatile reg.
12:    rte                     | finished...
```

Figure 6.9: Assembly-level (MC68020) wrapper routine

mode stack pointer is part of the `escape` abstraction modeling the nonvolatile processor state. Lines 9 and 10 contain instructions used to restore this stack pointer from the saved nonvolatile processor state. Note that these two instruction pairs are only present if the nucleus implements the separation between the user and supervisor modes. Thus, the wrapper routine shown in Figure 6.9 assumes user/supervisor mode separation. In line 6, the exception stack frame pointer (i.e., `escape*`) is generated and passed as an actual parameter to `LiftPointsNOP()`. This parameter is removed from the stack in line 8. The CPU is directed to restore its nonvolatile processor state and to finish trap/interrupt handling by executing the instruction of line 12.

Wrapper routine overhead is minimal. This is also true for the lifter routine overhead, which is determined by the C++ runtime system and the single C++ statement executed by `LiftPointsNOP()`, as shown in line 14 of Figure 6.8 (`assert()` is only enabled when the nucleus/kernel is ported). The instructions used to save/restore a user-mode stack pointer are only required for specific nucleus members of the kernel family. As a matter of fact, there are only two instructions (lines 6 and 8) that are implied by the exception-handling model. All other instructions are mandatory anyway, in order to handle traps/interrupts safely.

The names "`WrapLifterNOP`," "`LiftPointsNOP`," "`pointsNOP`" and

"`NOP`" are actually preprocessor parameters. The same source file is used to generate a linkage descriptor for each trap/interrupt vector table entry. At binding time, these descriptors must have different names. This is accomplished by making multiple compilations of the same source file and likewise specifying different assignments of the preprocessor parameters.[11] For example, the linkage descriptor for entry (index) 7 (i.e., with `NOP` = 7) uses the names "`wrapLifter7`," "`liftPoints7`" and "`points7`" accordingly. All generated linkage descriptors are kept in a *points library*. There are as many descriptors as entries of the actual trap/interrupt vector table. However, only as many descriptors as are required (i.e., referenced) by trap/interrupt handling subsystems are actually used. The linkage editor is responsible for statically configuring these subsystems. External references to `linkage` objects are resolved by exploiting the points library, which causes the binding of linkage descriptors.

6.2.2 Kernel-Level Exception Handling

The classes described so far implement the propagation of hardware exceptions to objects. It remains open how these objects are structured. Figure 6.7 introduced the `points` abstraction. This is where to start off with the modeling of active/passive objects assigned to handle hardware exceptions (i.e., traps or interrupts).

As mentioned above, the `points` abstraction is a deferred class. This class provides a single member function only, a *pure virtual function* in terms of C++. The member function is represented as a function call operator and specified as `void operator()(escape* = 0)`. When called, it usually receives a pointer to the currently active exception stack frame as an argument. The modeling of trap/interrupt handler objects then begins with introducing a subclass derived from `points`.

[11] The implementation does not rely on `generic.h` being shipped with standard C++ compilers. Relying on that macro package made the PEACE system non-portable. Different C/C++ preprocessors generated different output. A `Makefile`-based approach is used instead, submitting a template source file to the compiler in a loop controlled by the `NOP` (environment) variable. The `NOP` variable iterates from 0 to $N-1$, with N specifying the number of trap/interrupt vector table entries defined by the hardware.

This is exemplified by Figure 6.10.[12]

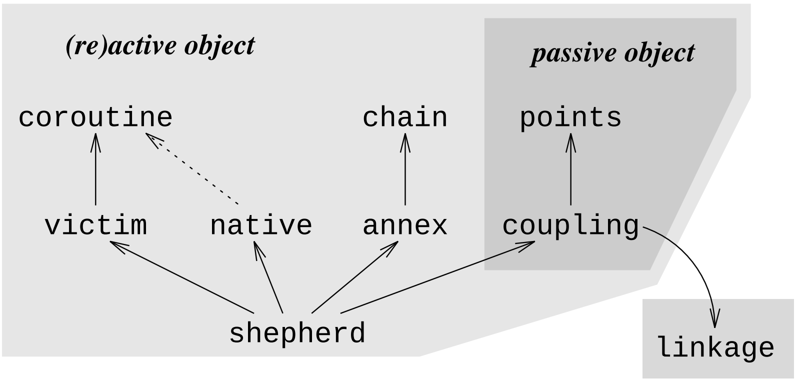

Figure 6.10: Trap/interrupt handler as an active or passive object

There are two models provided. The models distinguish between the implementation of a trap/interrupt handler as a passive or active kernel-level object. Both models, however, use the same fundamental linkage procedure described above. The distinction between the passive and active models is made by means of subclassing. In particular, the active object model is a specialized variant of the passive object model. Traps are typically handled by passive objects, whereas interrupts are candidates for handling by active objects. Especially, active objects must be used in the implementation of interrupt-handling subsystems based on Transputer technology. This is the basic rationale of these two models, namely, to provide abstractions that help the system programmer to design highly portable kernel-level software.

6.2.2.1 Passive Object Model

A passive object is modeled by means of the `coupling` abstraction. A trap/interrupt handler is then represented by a class that inherits `coupling`. The `coupling` constructor expects a `linkage` ob-

[12]The dashed arrow from `native` to `coroutine` shall indicate that the inheritance branch at this place is more elaborate than shown in the figure. The complete picture is drawn and discussed later with Figure 6.26.

ject as actual parameter. This object tells `coupling` which type of trap/interrupt shall be handled. The constructor performs the binding to the trap/interrupt vector table entry specified by the `linkage` instance. In order to also enable decoupling, the `coupling` abstraction keeps track of the actual linkage descriptor by encapsulating a pointer to the `linkage` object. This pointer is set up by the constructor. The `coupling` destructor uses this pointer to restore the previously defined linkage when the trap/interrupt handler object is destroyed.

The trap/interrupt handler method is dynamically bound when the trap/interrupt handling object is created. For this purpose, the user-defined[13] class that inherits `coupling` must redefine `operator()()` of the `points` abstraction. The effect is that, once this redefinition has taken place, a user-defined function call operator will be automatically invoked when the hardware exception corresponding to the `linkage` instance occurs (line 13 of Figure 6.8). This method then implicitly operates on the trap/interrupt handling object—the trap/interrupt handler has a `this` pointer. Thus, device driver programming, for example, means the following:

1. Design a class that introduces a device abstraction
2. Derive this class at least from `coupling`
3. Create an object of the newly designed class
4. Attach this object to a `linkage` descriptor, that is, pass that descriptor over to the `coupling` constructor

The mechanism of how propagating hardware exceptions as dictated by the actual CPU is made completely transparent to the system programmer. There is no need to deal with assembly-level and CPU-dependent programming unless demanded by the given problem to be solved. However, in many cases the device abstraction does not depend on CPU but rather on device capabilities. Even the formal parameter of `operator()()`, namely `escape*`, does not necessarily cause CPU dependencies, although the exception stack frame is highly CPU-intrinsic.

[13]The phrase "user-defined" refers to a kernel-level class that is derived from some superclass—in this particular case, from `coupling`.

The `escape*` can be considered as *opaque pointer* which will be transparently passed through to other modules (i.e., classes).

The user-defined class is often CPU-dependent if a trap and not an interrupt needs to be handled. For example, a page fault handler may be required to force the CPU to rerun the instruction that caused the page fault trap. A "rerun flag" is typically part of the processor status register saved by the exception stack frame at the time the page fault occurred. This is made feasible by the `escape*` parameter. After having finished page fault handling, the rerun flag is set by dereferencing the `escape*` pointer and manipulating the processor state.

6.2.2.2 Active Object Model

A slightly different view arises when making interrupt handlers active objects. The major distinction from passive objects is that, in the active object model, interrupt handlers are executing on their own stack. Interrupt handlers are associated with a private runtime context and do not share the stack of the currently executing (i.e., interrupted) process. This generally improves kernel safety. In the passive object model, the dimension of the (kernel) stack of a process depends on the interrupt model supported and on the runtime behavior of the interrupt handlers. The standard interrupt model approach is to implement multiple levels of priority and allow higher-priority interrupt handlers to preempt lower-priority interrupt handlers. In order not to run out of (kernel) stack space and thus raise the potential of segmentation faults signaled by the MMU, interrupt handlers are often flat and unstructured.

An active object model is not concerned with all these problems. Every interrupt handler has its own stack, which is of problem-oriented size. Consequently, before the interrupt handler is activated, a context switch must take place. Similarly, finishing interrupt handling means switching back to the context of the previously interrupted process. However, this sounds more overhead-prone than it proves to be in reality. Context switching only means exchanging the CPU register set (i.e., performing a *coroutine switch*). Address spaces are not (yet) switched. Since an interrupt already causes a (logical) context switch from user to supervisor mode, a simple (intra-address space) coroutine switch suffices to start the interrupt-handling active object. The active object processes in supervisor mode, too. It then depends on that object whether interrupt handling shall be performed in user space or carried out entirely in kernel space. Also note that a coroutine switch will only exchange the nonvolatile register set. This register set, however, usually needs to be (implicitly) saved by the interrupt-handling routine anyway.

An active object that is assigned to handle interrupts is an instance of the `shepherd` abstraction (Figure 6.10). As shown, the `shepherd` "is a" `coupling` too. Thus, associating the object with a specific hardware exception works exactly the same as discussed above. What makes the `shepherd` different from a `coupling` is shown on the left side of Figure 6.10: the two abstractions `victim` and `native`. The `native` abstraction implements kernel-level active objects in general. This is discussed in a later section. What must be kept in mind now is that a `shepherd` instance possesses its own runtime stack and can be scheduled, since it inherits all the necessary features from the `native` class.

Thus, a `shepherd` is a specialized `native` thread of the kernel entity. This thread is subjected to event-driven activation, which implies the coroutine switch upon interrupt. As shown in the figure, a `native` inherits `coroutine` capabilities. These capabilities are a save area in which to store the nonvolatile register set and a `resume()` operation to actually perform context switching (i.e., to exchange nonvolatile register sets). When an interrupt occurs, the nonvolatile register set of the currently active thread must be saved somewhere and then the nonvolatile register set of the `shepherd` object can be activated. This all happens completely asynchronous to kernel-level activities.

The currently active (i.e., interrupted) thread is also implemented by a `native` abstraction and is therefore provided with a save area for the nonvolatile register set. However, that part of a `native` instance cannot be used for register saving when interrupt propagation to an active object may be performed. The interrupted thread itself might perform a "regular" context switch, controlled by the scheduler. Besides the problem of yielding an inconsistent ready list, asynchronously activating the `shepherd` object under scheduler control, and therefore using the same save area instance, would raise the potential of overwriting an already or partly saved register set. This would lead to the destruction of the runtime context of the interrupted thread.[14]

The modeling of `shepherd` objects assumes software locking of nucleus critical sections (Section 4.1). For this purpose, the `shepherd` abstraction includes a second save area which is destined for storing the nonvolatile register set of an interrupted thread. This save area is implemented by the `victim` abstraction, which is merely used to rename the `coroutine` abstraction.[15] As a consequence, a `shepherd` object is not reentrant. On this basis, event-driven activation of interrupt-handler threads is implemented as shown in Figure 6.11.

The `shepherd` abstraction redefines `operator()()`, which is the object-bound entry point of an interrupt handler. This method performs two functions: remembering the exception stack frame pointer (line 12), and starting the active object assigned with interrupt handling (line 13). If required, the exception stack frame pointer can be obtained through `status()` (line 21). Line 13 selects the `resume()` method of the `coroutine` abstraction via the `victim` branch. This causes `resume()` to save the nonvolatile register set of the interrupted thread into the `victim` save area. The actual parameter of `resume()` in line 13 instructs the `coroutine` abstraction to restore the nonvolatile register set associated with the `native` thread that implements the `shepherd` active object.

When interrupt handling is finished, the `resume()` method of the `shepherd` abstraction applies, provided that this method has been called accordingly. Thus, the "return from interrupt" instruction in the active object model corresponds to the `resume()` that instructs the `coroutine` abstraction to switch back to the interrupted thread. This is shown in line 17. The difference to line 13 is that the arguments of `resume()` are simply exchanged. A switch back to the interrupted thread shall be performed, and this requires saving the register set of

[14]This fact places a restriction on the interrupt-handling subsystem but not on the interrupted (user/system) thread. The restriction is that, in order to activate a `shepherd` object, a separate save area for the nonvolatile register set of the interrupted thread must be provided. That is, transparency and/or modularity must not be compromised.

[15]Introducing classes that merely inherit superclasses and provide nothing else is the only means for renaming in C++. Renaming is necessary when the same superclass is used more than once in an inheritance path.

```
#include "annex.h"
#include "native.h"
#include "notice.h"
#include "victim.h"
#include "coupling.h"

class shepherd : private victim, private native,
                 public annex, public coupling {
   notice boot;
   escape* esf;
   void operator() (escape* esf) {
      this->esf = esf;
      ((victim*)this)->resume((coroutine*)(native*)this);
   }
protected:
   void resume () {
      ((coroutine*)(native*)this)->resume((victim*)this);
   }
   void    listen ();
   void    vanish ();
   escape* status () { return esf; }
public:
    shepherd (linkage&, unsigned ass = 0);
   ~shepherd ();
};
```

Figure 6.11: The shepherd class

the `native` thread (i.e., the `shepherd` active object) and restoring the register set of the `victim` (interrupted) thread.

There are two more notable methods of the `shepherd` abstraction. With `listen()`, the `shepherd` object requests exclusion from normal scheduling. As a consequence, the `native` instance is no longer controlled by the scheduler and the dispatcher. This ensures that event-driven thread activation does not cause race conditions since the scheduler is bypassed upon interrupt. Scheduler bypassing is released by calling `vanish()`. This method is invoked by either the user-defined interrupt thread or the `shepherd` destructor. The effect of `vanish()` is to achieve resynchronization with kernel-level activities. Note that the shepherd active object executes as an interrupt prologue (refer to Section 4.1).

The only function of vanish() is to ensure that the termination of the active object takes place in the epilogue phase. This may cause the shepherd object to be placed onto the activity schedule when another kernel-level activity is still in progress. The shepherd instance is thus an activity object that potentially must be serialized with other activity objects. In order to achieve this, annex is inherited. The annex abstraction makes shepherd objects queuable when they are not yet allowed to enter the epilogue phase. This applies independent of the termination procedure just discussed and can be generally exploited to switch over from prologue to epilogue processing.

Finally, some remarks about the shepherd constructor (line 23). The constructor has two formal parameters. One parameter specifies a linkage object in order to attach the active object to an entry of the trap/interrupt vector table. This parameter is directly passed through to the coupling constructor. The second parameter specifies the additional stack size (ass) of the native thread, giving a shepherd instance an autonomous flow of control. By that means, user-level classes derived from shepherd are able to tune the extent of the runtime stack needed for interrupt processing.

6.2.3 Timer Interrupt Handler

This subsection presents a concrete case study of a device driver subsystem used by the kernel to trigger timeout handling in a problem-oriented manner. The case study is on the timer interrupt handler and relies on the passive object model (Figure 6.12). Kernel-level timeout handling is required for a number of purposes. There are at least three cases in which the nucleus applies timeout handling under certain circumstances. These are clock-driven rescheduling, COSY retransmission control and user-level thread synchronization.[16]

First of all, a clock device driver is required. This driver is represented by the clockDevice abstraction. As indicated, a clockDevice

[16]However, note that due to the family organization, these features are not provided at all times and by all members of the kernel family. Refer to Section 2.4.

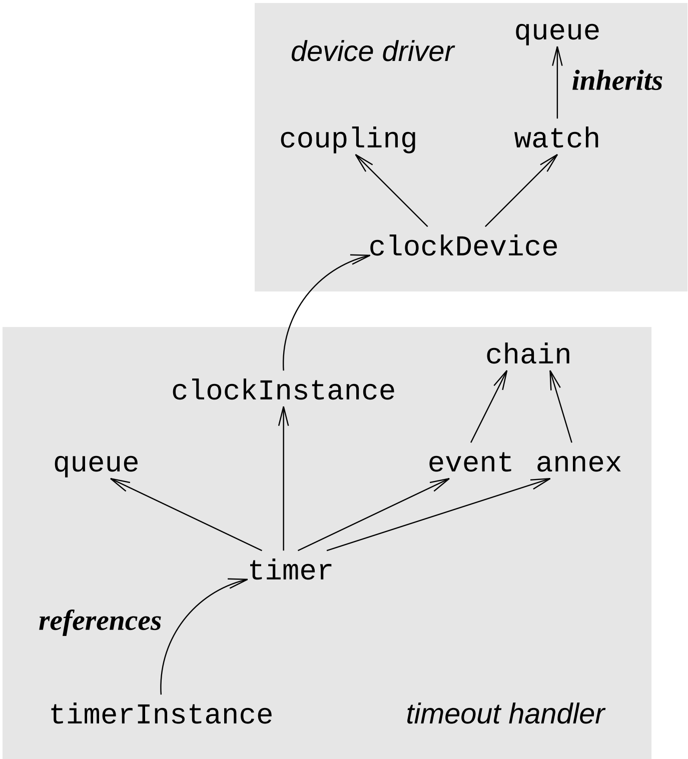

Figure 6.12: Timer interrupt handler as a passive object

is directly derived from `coupling` and redefines `operator()()`, which actually implements the clock interrupt handler. This makes the device driver a passive object. The `clockDevice` also implements a prologue queue, which is achieved by inheriting `watch` capabilities. The `watch` abstraction "is a" specialized `queue`, providing specific queuing disciplines. Every clock interrupt causes the clock interrupt handler to instruct `watch` to process the prologue queue. What the `watch` instance does is nothing but call `prologue()` for all queue elements (of type `event`) at every clock interrupt.

A call on `prologue()` may then result in thread/bundle rescheduling, packet retransmission, or timeout processing. In fact, it depends on the actual `event` object, and thus on the `prologue()` implementation, which of these functions is performed. Timeout processing causes the activation of a `timer` instance. As shown in Figure 6.12, `timer` is derived from `event` and, by that means, represents a prologue object processable by the `watch` abstraction. A `timer` also represents an epilogue object because of the `annex` relationship. Thus, a `timer` instance is a passive object that is subjected to activity serialization when the prologue requests epilogue processing. By specializing the `queue` abstraction, `timer` implements a timeout queue sorted by waiting time relative to the head element. Timeout queue processing takes place in the epilogue phase, that is, it is carried out by a specialized implementation of the `epilogue()` method of `annex`. The only thing the timer prologue does is to switch over to the epilogue and thus synchronize with other kernel-level activities.

The linkage to the clock device driver is accomplished via `clockInstance`. This abstraction ensures that every `timer` instance is bound to the same `clockDevice` instance. There may be many timeout handlers, but only a single clock device. A similar technique is used to attach kernel-level modules (i.e., objects) to the default timeout handler. There is a `timerInstance` abstraction which ensures that the default `timer` instance is created only once. Both `clockInstance` and `timerInstance` implement a pointer to the `clockDevice` and `timer` object and a reference count, respectively. The constructor of these *instance classes* allocates the object when not yet present and sets the reference count to one. Each further constructor call increases the reference count and copies the pointer into the new instance object. The destructor decrements the reference count and deallocates the instance object when the reference count drops to zero.

6.2.4 Cross-Domain Call Trap Handler

A nucleus service access point is represented in two different manners. The implementation distinguishes between a single- or multi-address-space model. In the former case, exported nucleus functions are called on a LOI basis. The latter case employs NOI. Refer to Section 2.3 for an explanation of the terms LOI and NOI. In the case of NOI, a *cross-domain call* is carried out to invoke the nucleus in a multi-address-space configuration. This was extensively explained in Section 2.2.4. Here, the cross-domain call implementation exploiting a traditional "system call trap" mechanism is discussed.

In the multi-address-space configuration, nucleus service functions are made accessible by Vice (see Section 2.2.4). The Vice "sandwich" [67] accepts traps issued by user-level entities when the nucleus is asked to carry out a specific service function. This class specification of the Vice trap handler is shown in Figure 6.13.

```
 1: #include "coupling.h"
 2: #include "hashSimple.h"
 3:
 4: class vice : private coupling, private hashSimple {
 5:    void operator() (escape*);
 6:    int  browse (int);
 7: public:
 8:    ~vice ();
 9:     vice ();
10: };
```

Figure 6.13: Vice class specification

Class `vice` is derived from `coupling`, making `vice` instances passive trap-handling objects. As line 4 indicates, `vice` also derives from `hashSimple`. This second base class will be described in detail later on in Section 6.3.3. The important aspect here is to understand that `vice` also serves as a hash table specialization. This specialization keeps track of the objects used to implement access points of nucleus service functions. These objects act as "carriers." They translate user-level Vice requests into nucleus-level service calls and make Vice itself independent from the interface specifications of these services.

There are two fundamental primitives provided by `vice`. The `browse()` method is used to dynamically install a nucleus service function. This takes place when Vice is properly configured[17] and a hash

[17] The minimal subset of system functions implemented by the nucleus does not allow for dynamically extending/shrinking the kernel entity. Rather, the kernel has to provide minimal system extensions, which is a configuration matter. Basically, `browse()` is aware of these extensions and triggers dynamic restructuring accordingly.

fault is signaled. The second primitive provided is `operator()()`, that is, the actual trap handler routine. This routine is activated when a user-level thread issues a "nucleus trap." The trap is caused by a *stub routine* implementing the user-level representation of a nucleus service function. An example of such a stub routine is shown in Figure 6.14.

```
1: #include "region.h"
2:
3: int region::store (const opaque*, const opaque*, volume*) const {
4:    asm("movl #1,d0");
5:    asm("trap #0");
6: }
```

Figure 6.14: VICE (MC68020) nucleus service function stub

The instructions for issuing a nucleus trap are shown in lines 4 and 5 of Figure 6.14. First, the service function code is moved into a CPU "scratch" register (line 4). Second, the trap is performed (line 5). Every nucleus service function is assigned to a unique service function code. The stub implementation presented assumes that the actual parameters of a service function have been stored on the runtime stack of the trapped thread. Thus, the parameter-passing mechanism implemented by the compiler for "traditional" (i.e., non-RISC) processor architectures automatically results in the creation of a stack frame storing the parameters for the nucleus service call.[18] This stack frame is made accessible to `vice::operator()()` via `escape*` (refer to line 5 of Figure 6.13). More specifically, the stack frame base address is computed from the nonvolatile processor state saved by the assembly-level wrapper routine and stored as an `escape` object. A wrapper routine that creates such an `escape` instance was already discussed along with Figure 6.9. Which `escape` instance variable will be used to generate the stack frame base address depends on the actual CPU. In the CPU example shown here (MC68020), the saved *user-mode stack pointer* is the

[18] In the case of RISC processors, which often use CPU registers to store actual parameters, the stub routines declare an actual parameter descriptor object locally on the runtime stack. This object is then initialized from the actual parameter list of the specific method representing the nucleus service function to be called.

best choice.

Executing the trap instruction encoded by the nucleus stub routine activates a wrapper routine, which, in turn, invokes the problem-specific trap handler. The implementation of the VICE trap handler activated in the particular case discussed here is illustrated in Figure 6.15. Trap handling begins with executing the (customized) function `vice::operator()()` (line 16), whose actual parameter `esf` points to the exception stack frame created by the wrapper routine. The `vice` constructor (line 11) is responsible for hash table initialization and the coupling to the trap vector table entry employed by the CPU for trap propagation, respectively. `VICE_POINTS` is used as the trap linkage descriptor (line 9). The destructor (line 14) releases the object allocated by the constructor. This object is used to specify an invalid hash table entry and is returned with every hash table miss. Its invocation automatically results in the delivery of an error code.

The basic idea of `operator()()` is to activate a `carrier` object that knows how to actually invoke the requested nucleus service function. Every carrier object implements the `expand()` method, which serves as a service-function-specific wrapper routine. Every exported nucleus service function is invoked by its own `carrier` specialization. Thus, `expand()` is a virtual function and has a customized implementation. This implementation copies the actual parameter list from user to nucleus space and invokes the requested nucleus function. The invocation of `expand()` is shown in line 33, with `apl()` delivering the base address of the actual parameter list. As indicated, the result of service function execution is passed back to the user-level thread by overwriting an `escape` instance variable referring to one of the saved volatile CPU registers.

The `carrier` object is taken from the `vice` hash table. In fact, the service function code serves as object identifier and hash key at the same time. Figure 6.15 shows two different implementations of the lookup procedure. The first implementation (`fameExtensible` disabled) does not support dynamic restructuring of the kernel entity. This static case assumes that the object identifier must refer to a valid hash table entry. The `access()` method is applied (line 20), yielding the pointer to a `carrier` object related to the specified object identifier (i.e., service function code). This method does not check whether the `carrier` object

```
 1: #include "vice.h"
 2: #include "assert.h"
 3: #include "escape.h"
 4: #include "linkage.h"
 5: #include "carrier.h"
 6: #include "pointsMap.h"
 7: #include "fameExtensible.h"
 8:
 9: extern linkage VICE_POINTS;
10:
11: vice::vice() : hashSimple(empty = new carrier),
12:                coupling(VICE_POINTS) {}
13:
14: vice::~vice() { if (empty) { delete (carrier*)empty; } }
15:
16: void vice::operator() (escape* esf) {
17:    assert(esf);
18:    register carrier* gate;
19: #ifndef fameExtensible
20:    gate = (carrier*)access((int)esf->voi);
21:    assert(gate);
22: #else
23:    gate = (carrier*)lookup((int)esf->voi);
24:    if (!gate) {
25:       if (browse(esf->voi) < 0) {
26:          esf->vrc = (unsigned)-1;
27:          return;
28:       }
29:       gate = (carrier*)lookup((int)esf->voi);
30:       assert(gate);
31:    }
32: #endif  fameExtensible
33:    esf->vrc = gate->expand(esf->apl());
34: }
```

Figure 6.15: VICE trap handler

is really bound to the specified identifier. It merely delivers a `carrier` object, which could be the `empty` carrier (line 11) as well. Invocation of an `empty` carrier results in an error code. That is, the default (i.e., initial) hash table entry is used to avoid an explicit check of whether a hash miss or a hash match was given.

The second implementation of `operator()()` (`fameExtensible` enabled) supports dynamic restructuring of the kernel entity. Instead of simply accessing the specified hash table entry, a hash table `lookup()` is performed. This procedures delivers a `null` pointer if the requested service function is not yet known (i.e., if the corresponding `carrier` object is missing). As a result of this hash miss, `browse()` is called (line 25) in order to incrementally load a nucleus service function. The service function to be loaded is identified by the service function code exploited as the hash key. If `browse()` succeeded, `lookup()` is retried to grab the newly installed `carrier` object (line 29).

Nucleus service function execution is thus achieved by invoking a specialized `carrier` object. As was mentioned above, every installed service function is related to its own `carrier` object. An example of the class specification of such an object is Figure 6.16. This class implements the Vice wrapper routine to expand a nucleus trap into the `store()` method provided by the `region` abstraction. Class `regionStore` corresponds to the `region::store()` stub routine shown in Figure 6.14.

```
1: #include "region.h"
2: #include "carrier.h"
3: #include "aplRegion.h"
4:
5: class regionStore : public carrier {
6:    int expand (void* apl) {
7:       return THIS->origin->
              store(THIS->client, THIS->server, THIS->target);
8:    }
9: };
```

Figure 6.16: A nucleus service function carrier

The actual parameter of `expand()` (i.e., `apl`) refers to the actual parameter list created by the nucleus stub. By means of `THIS`, the `apl` instance is temporarily interpreted as `aplRegion*` to correctly access the `store()` parameters (line 7). This coercion is implemented by the

`THIS` macro contained in `aplRegion.h`. That is, the actual parameter list of `region::store()` is described by an `aplRegion` structure. The structure member `origin` refers to the `this` pointer of the original user-level `store()` method invocation. The other three parameters correspond to the parameters shown in line 3 of Figure 6.14.

6.3 Object Addressing

Object addressing at the kernel level is concerned with the management of communication endpoint addresses. These addresses are technically represented by communication gates. Communication gates are held in the hash table; thus, so are hash items. The following subsections discuss the modeling of communication endpoint addresses in general and the structure of hash items in specific.

6.3.1 Modeling Opaque Identifiers

The nucleus interprets each communication endpoint address as an opaque pointer. These pointers make the actual structure (i.e., size) of a communication endpoint address transparent to the nucleus. They refer to so-called opaque identifiers (see also Section 5.3.4.3). There are two representations of an opaque identifier. One representation consists of an LUI (Section 5.3.4) only, and the other representation additionally consists of a `sparse` field which makes the LUI a systemwide unique identifier. These different representations also introduce different nuclei of the kernel family. The reason why this results in different nuclei is shown in Figure 6.17.

An LUI is a `lui` instance. This class describes the shape of an object identifier that fits into a single CPU register. More specifically, `lui` contains an instance variable that is of a simple C++ data type such as `int`. A unique identifier is implemented by the `unique` abstraction. As illustrated, `unique` inherits `lui` and `sparse`. This composition yields an LUI that is extended by a random number of problem-oriented size. The size of the random number is specified by the `sparse` abstraction.

An opaque identifier is an instance of class `opaque`. As indicated by the dotted arrows, the `opaque` abstraction is derived from either

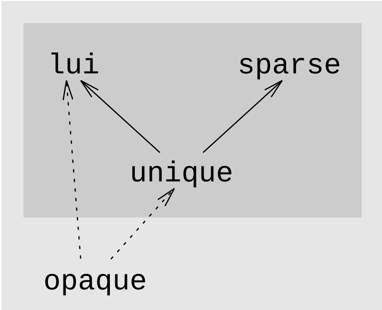

Figure 6.17: Hybrid representation of an opaque identifier

`lui` or `unique`. Thus, an `opaque` instance is of different size, depending on the superclass from which it is derived. Consequently, all kernel-level data structures that store `opaque` objects will have different sizes too. This especially holds for data structures used to describe messages exchanged between peer nucleus entities. Preprocessor flags control the generation of the various nucleus configurations.

Nucleus functions exported to user level, expecting communication endpoint addresses to be there, have formal parameters of type `const opaque*`. Each of these parameters takes the form of an opaque pointer. This makes the nucleus interface provided to the user independent of the actual composition of the `opaque` abstraction. In particular, the `opaque` abstraction introduces methods through which `const opaque*` objects are built by either using the `lui` value directly or generating a pointer to a `unique` object. The former corresponds to a weak unique identifier representation and the latter corresponds to a strong unique identifier representation (see Section 5.3.4.3). Weak unique identifiers make the nucleus light or even featherweight in communication, but may cause integrity problems for a specific set of applications. In contrast, strong unique identifiers result in heavyweight inter-nucleus communication but overcome the potential of integrity problems.

6.3.2 Hash Item

The kernel associates communication gates and threads with hash items. As discussed later, a kernel-level thread is nothing but a specialized communication gate, which is a specialized hash item. Anyway, hash items are stamped with systemwide unique identifiers, and thus are globally addressable. The structure of a systemwide unique identifier is designed to ease the object localization process in distributed environments. In addition, hash items are potentially stored in hash lists.[19] All these facts motivate the hash item structure shown in Figure 6.18.

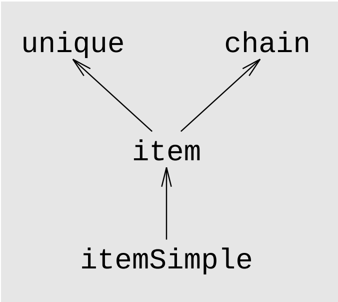

Figure 6.18: Hash element composition

There are two sorts of hash items, represented by the `item` and `itemSimple` abstraction, respectively. An `item` is composed of a `unique` and a `chain` abstraction. This basically makes a hash item a queuable unique identifier. The `item` abstraction, just like `itemSimple`, defines no further instance variables other than those provided by `unique` and `chain`. The difference between `item` and `itemSimple` is that the former supports hashing by exploiting the entire `unique` instance, while

[19] As discussed later, only hash table overallocation will result in hash lists. In this case, more hash items have been created than first-level hash table entries provided. A first-level hash entry is implemented as a hash table entry and is thus accessible through an index operation. If hash table overallocation arises, some Nth-level hash items must be stored in hash lists. A first-level hash item then serves as the head of a hash list, and an Nth-level hash item becomes accessible through a linear list search.

the latter employs the `lui` subfield of `unique` only. This results in different performance figures and directly reflects the idea of distinguishing between strong and weak unique identifiers.

A hash item always encapsulates a systemwide unique identifier. Thus, it may be of fairly large size, dependent on the size of the `sparse` field of that identifier. Note that it is not the larger object address stored in memory that makes object localization more overhead-prone. It is the size of the bit pattern that is to be looked for, and thus compared with the stored hash key, which causes the performance decrease. Therefore, different method implementations are used. Inheritance is applied to adapt the lookup method to the addressing model actually incorporated in the nucleus.

6.3.3 Hash Table

The hash table is a multilevel hierarchy of classes arranged in single-inheritance manner. The different classes are used mainly to reflect the idea of an ensemble of specialized nucleus members of the kernel family. There are different types of nuclei, each one providing dedicated hash table implementations. Figure 6.19 displays the hash table composition.

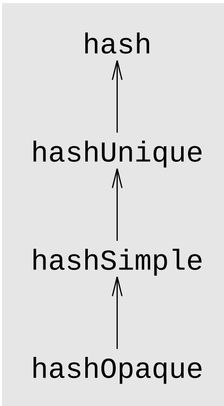

Figure 6.19: Hash table composition

Common to all variants is the `hash` abstraction. This class implements the hash table data structure only. A hash table consists of an array of pointers to a list of hash items, the index in which to store the next hash item and a pointer to a *null* hash item. The null hash item represents an invalid hash table entry or designates the end of the per-entry hash list. It is used to initialize the hash table at construction time and thus ensure that there is always a defined hash item pointer. This speeds up the hashing procedure, since no check for undefined hash item pointers must be made.

The next-item index variable is used to implement a round-robin hash table entry allocation strategy. This strategy is employed to vary the value of the `slot` field of newly created `lui` objects. By this means, two goals are met: first, lengthening the period until the same hash key will be created again; and second, trying to allocate new hash items as "first-level" hash items. A first-level hash item is always at the head of a hash list. Its lookup is faster than that of any other item in the same chain. Second-level through `item` − 1-level[20] hash items are linearly searched for in a loop, stepping through the per-slot hash list.

Hash item lookup is done via methods introduced by `hashUnique`, exploiting the `item` abstraction, and `hashSimple`, exploiting the `itemSimple` abstraction. In particular, `hashSimple` merely provides those three methods of `hashUnique` which are dealing with hash item access and lookup. Table 6.1 gives a brief description of the access methods. Methods used to insert items into or remove items from the hash table are inherited from `hashUnique`. These methods are not performance-critical, as they are not employed by the communication procedures of the nucleus.

The hash table access methods are overloaded. Their actual signature depends on whether `hashUnique` or `hashSimple` is looked at. In the `hashUnique` case, the hash key is represented as `const unique&`, whereas in the `hashSimple` case, `int` (i.e., the type of the `lui` instance variable) is used. Because of having exploited subclassing to compose the hash table, both sets of lookup methods are provided at the same time. The applying instance of `hashSimple` can decide which way to go.

[20]Here, "`item`" stands for the correspondingly named LUI subfield described in Section 5.3.4.

Method	*Description*
`access()`	Access first-level hash item. Use the LUI `slot` field to determine the hash table entry specifying the head of the hash list in which to look up the item.
`locate()`	Locate Nth-level hash item by starting search at a specific list position. Terminate if either a match or the list end was detected.
`lookup()`	Look up a hash item. First, access the first-level item and check whether it is associated with the specified hash key. If the check failed, try to locate the key in the per-slot hash list. The method is implemented using `access()` and `locate()`.

Table 6.1: Methods for searching hash items

With `hashOpaque`, the hash table specialization dealing with opaque identifiers is introduced. This class only (re-)introduces the `lookup()` method described above. Again, a different signature is used. The hash key to be looked for is represented as `const opaque*`, which is exactly the representation of a communication endpoint address the nucleus provides to the user. There are two implementations of `hashOpaque::lookup()`, depending on whether an `opaque` inherits `lui` or `unique`. If of `lui` shape, `hashSimple::lookup()` is invoked with the `int` representation of `const opaque*`. Otherwise, `hashUnique::lookup()` is invoked with `const opaque*` interpreted as the pointer to a `unique` instance. The distinction is made at compile time, on the basis of preprocessor flags.

The nucleus exploits `hashOpaque` to create a hash table instance. This results in the two different kinds of nuclei of the kernel family, distinguishing between strong and weak unique identifiers. Strong unique identifier variants have `hashOpaque` configured to perform the mapping to `hashUnique`. Weak unique identifier variants require mapping to `hashSimple`.

6.3.4 Process Gates

Both kernel-level threads and communication gates are somewhat synonymous. Initially, there is a one-to-one, and later on there might be a many-to-one, correspondence between gates and threads. The initial case makes threads compatible with gates: a thread always encapsulates one gate. The gates give directions on how and where to reach active objects (i.e., threads or processes). They generally ease process addressing, not only communication. For that reason, the term "*process gate,*" not only communication gate, is used. In the following subsections, the structure and function of process gates is explained in some more detail.

6.3.4.1 Modeling Communication Endpoints

As was mentioned above, a communication gate is a hash item. This indicates that there is again some inheritance relationship, in order to design a class that describes a gate. Communication gates are also related to forwarding identifiers. The actual implementation and structure of forwarding identifiers, however, can be manifold and depend particularly on whether garbage collection techniques are used by the kernel to reclaim resources held by stale communication endpoint addresses (refer to Section 5.1.4). On the one hand, communication gates should be provided at the user level to ease the implementation of problem-oriented communication and addressing schemes. On the other hand, the actual structure of forwarding identifiers must be kept hidden to ease kernel configurability. Figure 6.20 shows the class hierarchy that models a communication endpoint address (i.e., forwarding identifier), while Figure 6.21 describes the abstraction from this structuring measure.

A forwarding identifier is implemented by the `guide` abstraction. This class is derived from `item` and `counter`. The `unique` properties (see Figure 6.17) thus inherited implement the original and exported communication endpoint identifier of an active object. When an active object is migrated for some reason, or when communication to this object is to be redirected, the `guide` instance of that object is set

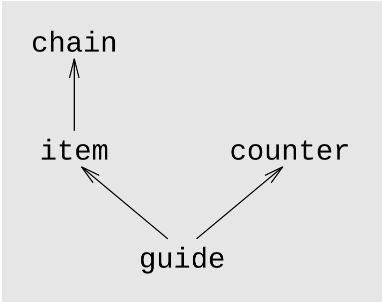

Figure 6.20: A communication endpoint address

up accordingly.[21] For that purpose, `guide` defines a `unique` instance variable that acts as the forwarding address. The `counter` abstraction implements a reference count for a `guide` instance.

Abstracting from the structure of a forwarding identifier is achieved by means of the `loophole` class. The `loophole` class is the abstract specification of a (process/communication) gate—it manifests the *gate concept*. A `loophole` stores a "`guide` pointer," that is, the `unique` identifier corresponding to the `guide` instance.[22] Derivation from `ticket` makes `loophole` compatible with the technical representation of communication endpoint addresses. A communication endpoint address is implemented as `const opaque*`, and the `ticket` abstraction provides the necessary transformations of the corresponding `unique` instance.

Communicating processes must have a "ticket." This ticket is derived from the systemwide unique identifier denoting the communication endpoint address, which is reflected by the class hierarchy shown. The `ticket` abstraction is implemented by the nucleus. It employs the

[21] If process migration is performed, only the `guide` instance implicitly associated with that process remains allocated on the node. As discussed later, a default process gate is part of a kernel-level active object and gives that object its systemwide unique address. Process migration technically means to locally destroy the active object, with the exception of the encapsulated `guide` instance. That is, the active object is transformed into a passive `guide` object.

[22] The name "loophole" was chosen to express the fact that by instantiating a `loophole` instance, the creator may gain access to unrelated active objects. These "unrelated" active objects were not created by the `loophole`-instantiating thread.

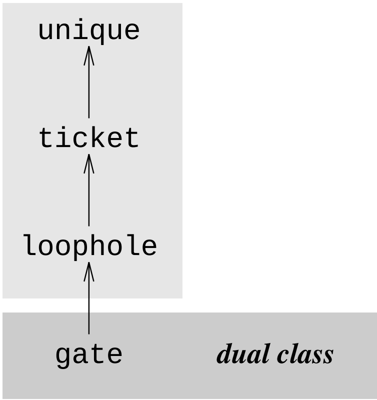

Figure 6.21: A communication/process gate

`opaque` abstraction (Figure 6.17) to convert a `unique` instance into a `const opaque*` and vice versa. At construction time, the `unique` instance is set up from the `unique` instance obtained from the associated `guide` object. After a `loophole` instance has been established, the user creates a communication endpoint address by using the methods provided by `ticket`. This address then enables communication with the active object referred to by the gate.

As indicated in Figure 6.21, a gate is a *dual object*. The corresponding *dual class* is represented by `gate`, which is derived from `loophole`. By that means, networkwide access to `loophole` instances is provided. Subclassing is used to make these instances globally accessible: `gate` introduces only the minimal extension to make `loophole` globally accessible. However, there is a second reason for this sort of specialization. Section 5.3.2 explained that a class generator tool is used to create, from a dual class, different classes for the client and server sites of a dual object. This limits configurability when using the generated classes directly as a superclass. Especially, classes derived from the likeness class become aware of the fact that there are two physically isolated fragments of the same logical object. Transparent transformation between single- and multi-address-space models are then only made feasible on a recompilation basis and cannot be achieved at binding time or even dynamically at runtime. In a single-address-space configuration, only the "virgin" (nonprocessed) class specification is required and thus need be visible to the client. This configuration provides the minimal subset of system functions, whereas the model of multiple address spaces is implemented as a minimal system extension. Inheritance and function polymorphism solve this problem. In order to make the (user/system) application independent from the particular kernel configuration, a "*smart pointer*" to a `loophole` instance is introduced. This pointer is implemented by `associate`. Its relation to a (process/communication) gate is shown in Figure 6.22.

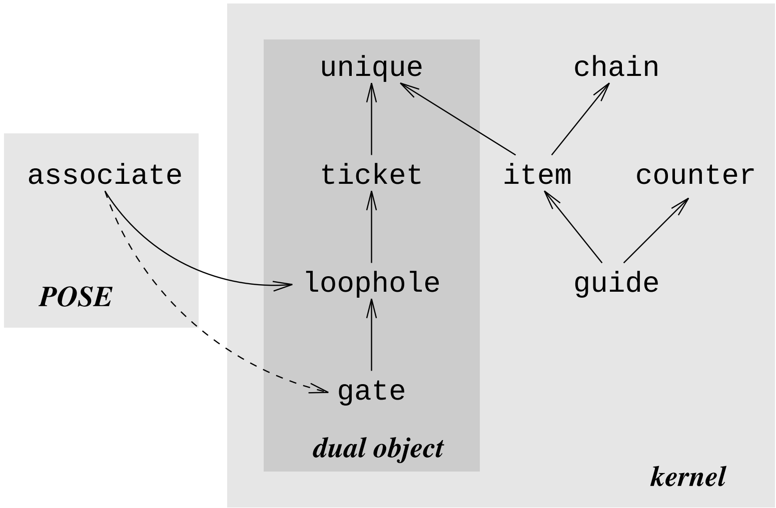

Figure 6.22: Interfacing to communication/process gates

The `associate` class serves as a client interface to `loophole`/`gate` (i.e., dual) objects. This class contains a `loophole` pointer only, whereby `loophole` is the original, non-annotated class specification. Consequently, the client sees no dual class, and thus never depends on classes that are produced by the class generator. All `loophole` methods and the destructor are specified as virtual functions that will be redefined accordingly by `gate`. Invoking a `loophole` method by dereferencing the `loophole` pointer thus causes the invocation of the corresponding local `gate` method. All local `gate` methods, however, take the form of "client stub procedures." Calling these procedures results in executing the ROI protocol with the remote `gate` clerk—this also applies when calling the constructor and destructor methods. Having received and decoded the ROI request issued by the client, this clerk locally invokes the corresponding `gate` method. However, in that case the "server stub procedure" is called, causing the execution of the real `gate` and thus the `loophole` method.[23]

Dealing with the creation of `gate` objects is the task of the `fasten()` and `settle()` methods provided by `associate`. Communication gates are created by `fasten()`, while process gates are created by `settle()`. The creation of a process gate means the creation of a (kernel-level) lightweight process implicitly bound to a communication gate. That is, every process receives a unique communication endpoint address at creation time. This aspect is revisited in the next section. It is important at this moment to understand that a lightweight process also has the appearance of a `guide` object, to which a `loophole` can be attached.

Both methods `fasten()` and `settle()` create either a normal or dual object, depending on whether the nonprocessed or processed `gate` class is employed. There are two different implementations of each of these methods: one for the single-address-space model and a second one for the multi-address-space model. As a matter of fact, the distinction between these two configurations is made at binding and not at compiling time. Refer to Figure 6.22 for the two different situations. The solid arrow from `associate` to `loophole` corresponds to the logical (user) view, whereas the dotted arrow indicates the actual (system) view of the `loophole` interface. This issue is discussed in the next section.

Considering the entire path from an `associate` instance (Figure 6.22) to a `guide` instance (Figure 6.20), two logical boundaries can be identified. One boundary is between `associate` and `loophole`, and the other is between `loophole` and `guide`. The `loophole/guide` boundary hides the structure of forwarding identifiers from the client. The `associate/loophole` boundary maintains software compatibility irrespective of the actual address space model employed and therefore

[23]In fact, `associate` and `loophole` can be considered as software-technical measures to hide the dual-object nature of `gate`.

also improves kernel configurability and scalability.

6.3.4.2 Binding Active Objects

Gates introduce the methods `replug()` and `portal()`, which are used by the naming system to bind name servers with processes and to determine the communication endpoint address of a per-process name server (refer also to Section 5.3.6). There is also the `assign()` method to define a forwarding identifier and the `lookup()` method to obtain the currently defined forwarding identifier. In addition, a couple of constructors are provided. The constructors are classified into two categories. The first category deals with communication gates and was extensively discussed in the previous section.

The other gate constructor category is concerned with the instantiation of gates that are part of an active object. Making a gate object part of an active object is achieved through subclassing. As discussed in the next section, at a certain level of abstraction, `guide` serves as a superclass to compose a class used to establish kernel-level active objects (i.e., processes). A communication gate referring to such a `guide` instance, hence, is also termed *process gate*. The creation of process gates causes the construction of kernel-level active objects. Similarly, process gate deletion yields active object destruction.

Communication Gates. A *communication gate* constructor is `gate(const unique&)`. This constructor tries to preallocate a gate with a specified unique identifier and thus reserve a communication endpoint identifier for whatever reason.[24] If this identifier is already known, the constructor simply sets up the linkage to the corresponding `guide` object—provided the `guide` object is not a forwarding identifier but part of an active object. Hence, a connection to a kernel-level active object is established.

[24] Recovering from a node crash, for example, usually requires the establishment of the process structure that was known when the last checkpoint was written, provided that the node can be reclaimed again. These processes are reestablished with and bound to the unique identifiers they were associated with at checkpointing time.

Processing of user-level threads is achieved by scheduling kernel-level active objects. Whether these threads take the form of active objects (i.e., are instances of a specific class) depends on the application. In particular, there is a one-to-one correspondence between user-level thread and kernel-level active object. Before the thread starts processing the user program, it begins executing inside the kernel—the thread bootstraps out of the kernel. It is the kernel-level active object that technically represents the thread at that time. This object then performs an upcall to the user level. That is, the active object continues processing in "user space" by jumping to a well-defined code address in the user program. This code address typically refers to a routine in the runtime system. If a Fortran (or C) program is started, no further system activities take place. Object-oriented (parallel) applications, however, make themselves known to the kernel by attaching a user-level active object to the corresponding kernel-level active object. This is the purpose of `gate(const unique&)`.

When a thread starts user-level processing, it may even decide to create a "wrapper object." This object is nothing but a user-level active object, except that its instantiation does not cause the creation of a new and autonomous flow of control. Rather, the flow of control and runtime stack of the currently executing (i.e., instantiating) thread is used to call the wrapper object into being. Wrapper object construction transforms a thread into an active object and thus switches from a non-object-oriented into an object-oriented context. This transformation is made top-down, for only the user program knows whether it is object-oriented, especially the C++ structure. The `gate` abstraction provides features to perform the binding. The current thread employs the `ticket` abstraction (i.e., method `address()`) to determine its own unique identifier. This identifier is the same for the kernel-level active object, which basically schedules the thread. Providing the `gate` constructor with this identifier finally yields the binding between user-level and kernel-level active object. All these steps are performed by the runtime system linked to the user program but must be initiated by a call from the user program. Transformation from a non-object-oriented to an object-oriented context is performed by the `peace()` procedure.

Process Gates. The object constructor for establishing a *process gate* is `gate(const unique&, const unique&, const germ&)`. The two unique identifiers constituting the formal parameter list specify the team into which the new (lightweight) process is to be created. From the implementation point of view, a team consists of one or more *thread bundles* and one or more *thread baskets.* A bundle defines an autonomous scheduling domain of a group of threads, and a basket serves the collection of threads ready to receive incoming messages (i.e., rendezvous requests). Bundles are subjected to preemptive scheduling. The basket is accessed whenever the message-passing system detects that the addressed receiver thread is currently unable to immediately accept the incoming message. If non-empty, a substitute thread is removed from the basket and instructed to process the pending message.

The first unique identifier of the process gate constructor selects the basket of the addressed thread. Since a basket also implements a team of threads, this first parameter is an alias for a *team identifier.* The second unique identifier of the process gate constructor specifies the scheduling domain of the addressed thread. In both cases, the identifiers technically address threads. Indirectly via these threads, the two fundamental data structures implementing a team are selected. If the identifier does not refer to an existing thread, a new thread with exactly this identifier will be created. In addition, a new team structure is established and linked to the new thread.

The `germ` parameter describes the initial execution context of a user-level thread (i.e., potentially active object). This context consists of a program counter, stack base address, stack length and pointer to a `points` object. If defined, the `points*` instance variable represents the entry point into a user-level active object. The other instance variables serve obvious functions. They are used to set up a user-level thread and to direct the kernel-level upcall accordingly. Thus, a `germ` object is used to establish a process. A kernel-level active object is created and instructed to continue user-level processing in the context specified. The `gate` constructor creates an instance of a specific process class (see next section) and remembers the `guide` part of that instance by means of the `unique` abstraction inherited by the `gate` class. Refer also to Figure 6.22 for the general `gate/guide` relationship.

6.4 Processes

In the previous section, one of the subjects under discussion was the distinction between user-level and kernel-level active objects. This discussion is continued by investigating how these objects are designed. Class hierarchies for kernel-level and user-level active objects are elucidated. Before entering the discussion, some introductory and general remarks on active objects are made.

6.4.1 Active Objects

Active objects are constructed from a class that contains an *autonomous method* (`action()`) to specify the (private) thread of control of instances of that class. The method must be redefined by a specialized (application-oriented) class to enable the construction of an active object. Redefinition and thus specialization are accomplished by means of subclassing. A parallel application becomes a system specialization. The user is required to design a new subclass and to specify its own `action()`. One or several subclasses may be designed this way, to generate one or more process types.

Dynamic binding (*virtual functions*) is used to make the autonomous thread of control known to the nucleus. This thread of control then implements a *lightweight process*. The private data space of the corresponding active object becomes *own variables* of that process. In principle, a lightweight process is created/destroyed by executing the constructor/destructor of the class of an active object.

An active object acts as either `native` of the kernel or as a `thread` of the application, POSE or kernel subsystem. The `native` class specifies active objects that *must* reside in kernel space. In contrast, the `thread` class specifies active objects that *may* reside in kernel space. The former class describes kernel-level active objects (or threads), whereas the latter class describes user-level active objects (or threads). For each case there are specific base classes whose constructors/destructors are responsible for the different ways to create/destroy the processes.

Basing classes on `native` makes derived classes dependent on internal (nucleus and kernel) abstractions and requires that these objects be subject to the supervisor mode of execution. It serves to implement *site-dependent entities* such as low-level device drivers (see also Section 6.2.2.2). Basing classes on `thread` supports the implementation of *site-independent entities*, the execution of which is independent of the actual processing mode (user vs. supervisor) and the effective destination node. Either way, the same `action()` specification is used to identify the entry point of the active object. This makes the association of a "user process" to either `thread` or `native` virtually transparent.

Although the kernel is implemented in an object-oriented fashion (exploiting C++), it must not impose on higher-level system components to correspond to the same programming discipline. That is, the kernel must not dictate higher levels to model and represent threads as active objects similar to kernel-level threads. For certain applications, a non-object-oriented model could be more appropriate. In particular, a number of parallel applications were and still are written in Fortran, which lets these programs have very little in common with object orientation. User acceptance will be zero if they are required to employ the same programming paradigm used to design and implement the kernel. The kernel must also support non-object-oriented applications—it must be open to different user programming disciplines.

6.4.2 User-Level Active Objects

Almost every active object is a `thread`, and thus is constructed from a class that is composed by *multiple inheritance* from POSE and nucleus classes (Figure 6.23). A `thread` instance is associated with a kernel-level active object, which is effectively implemented as a `native` thread—in fact, `native` scheduling yields `thread` processing. This coupling is implemented by the `associate` abstraction. As shown in Figure 6.23, `associate` "is a" `germ`, as introduced in Section 6.3.4.2. The `germ` abstraction keeps track of the initial runtime context of a thread. In particular, it stores a pointer to the `points` object used to bootstrap a `thread` instance. The thread constructors compute the start address of the wrapper routine that takes care of invoking `operator()()` in order to activate a thread. This routine requests from the nucleus the delivery of the `points` reference that has been passed down in the course of creating a `loophole` instance (i.e., a process gate).

The construction of a `thread` instance results in the construction

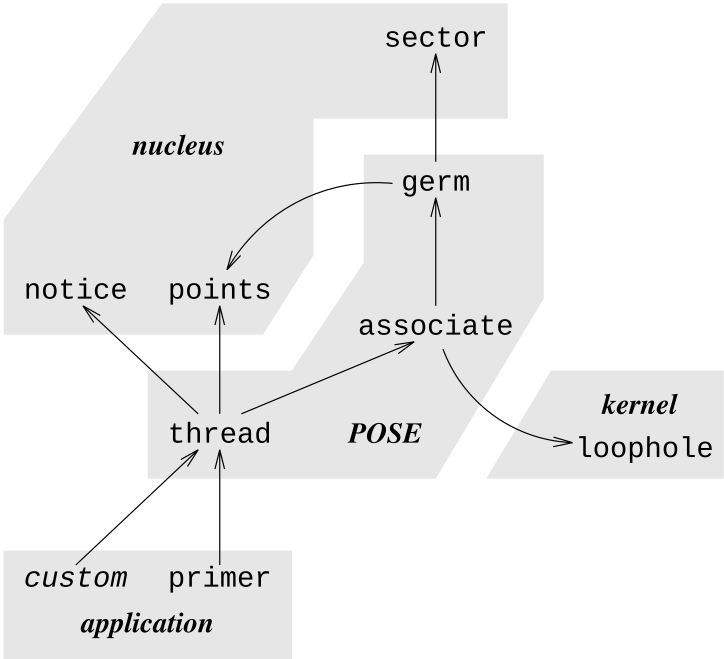

Figure 6.23: User abstractions

of an active object that is managed by the kernel and, usually, in the creation of a (user) stack. A user stack segment is described by the `sector` abstraction. The kernel performs an upcall to invoke the user-level active object when the corresponding kernel-level active object has been initially scheduled (i.e., “bootstrapped”). A specialized upcall handler then takes care of the invocation of `action()` in the `thread` context. This handler is implemented by `thread`. Upon return from `action()`, the handler stops further processing its enclosing `thread`. By exploiting the nucleus, it awaits a signal to invoke `action()` again and will never return to the kernel. The necessary features are provided by `notice`. This nucleus class implements thread synchronization on an event counter basis.

Dynamic binding is applied not only to introduce a specialized upcall handler, but also to specify the problem-oriented `action()` in the application context. Hence, `operator()()` leads to the activation of a customized `action()` of the application program, provided that the program employs object orientation and was written in C++. The application class is represented by the `custom` abstraction. In addition to these purely application-oriented classes, there is also one application-level class that is predefined by POSE. This is the `primer` abstraction. A `primer` is the initial active object of an entity, representing the initial flow of control. It transforms the C++ `main()` procedure into an active object. Thus, `primer::action()` is nothing but the object-oriented variant of `main()`. Actually, the wrapper object mentioned in Section 6.3.4.2 is nothing but a `primer` instance.

From the kernel (and nucleus) viewpoint, a `primer` instance is activated as an ordinary task. The corresponding entity is controlled by a single `native` thread only. As a matter of fact, at this point in time only a single thread of control executes inside the entity. Further threads must be created by the entity itself. Upon startup, the local runtime system creates the `primer` instance, which simply requests the binding with the currently executing kernel-level active object (see also Section 6.3.4.2). After that, the runtime system bootstraps the `primer` instance by performing an upcall exploiting the `points` object. This causes `primer::action()` to be started.

6.4.3 Threads Package

The procedure explained in the previous subsection also supports non-object-oriented applications, such as those written in C or Fortran. The mentioned local runtime system contains a default `main()` implementation which invokes `peace()` to create the `primer` root instance. A user may still use its own `main()`, with the effect that no root instance is created by default. In this situation, only kernel-level process abstractions are used to implement a user-level thread. Later, the user may decide to enter the "object-oriented world" by calling `peace()`.

In particular, a *threads package* that does not exhibit active objects at the user interface can be easily designed. This package is then nothing but a library that hides the `thread` abstraction and thus the object-oriented system interface. The library implements a specialized version of `thread` and at least two functions used to create and delete threads, respectively. Figures 6.24 and 6.25 sketch this `thread` customization.

```
 1: #include "thread.h"
 2:
 3: class Cthread : private thread {
 4:    void (*code)(void*);
 5:    void  *data;
 6:    void action () { (*code)(data); }
 7: public:
 8:    Cthread (void (*pc)(void*), void* ap, unsigned ass = 0)
          : thread(ass) {
 9:       code = pc;
10:       data = ap;
11:       settle();
12:    }
13: };
```

Figure 6.24: Thread customization

The `Cthread` class describes nothing but a wrapper object used to attach the C function specified by `pc` to a newly created lightweight process. When called, this function is furnished with the argument `ap` pointing to a user-defined data structure. The function invocation is shown in line 6. Thus, the sole purpose of `action()` is to act as a wrapper routine and assign a separate thread of control to a C function specified as `void foo(void*)`, for example. If this function should return, the attached lightweight process (i.e., current thread of control) will be stopped by `thread` (more precisely, by `operator()()` implemented by `thread`) after having called `action()`.

Construction of a `Cthread` instance only has to take care of telling the `thread` constructor the additional stack size (`ass`) required for safe `Cthread` processing (line 8), setting up the parameters for the `action()` code (lines 9 and 10), and creating the kernel-level thread (line 11). The `settle()` call results in the creation of a kernel-level active object (i.e., `native` thread), which schedules the user-level active object (i.e., the `Cthread` instance). `Cthread` destruction is entirely handled by the `thread` destructor and hence needs no further provisions (at least in the simple example discussed here).

```
#include "thread.h"

void* CthreadCreate (void (*pc)(void*), void* ap, unsigned sz) {
   return new Cthread(pc, ap, sz);
}

void CthreadDelete (void* victim) {
   delete (Cthread*)victim;
}
```

Figure 6.25: Threads package primitives

The non-object-oriented user interface is implemented by the two functions listed by Figure 6.25. Both functions merely trigger the instantiation and finalization of `Cthread` instances and thus active objects. Instantiation is requested by means of `CthreadCreate()` and simply results in the creation of a `Cthread` object. The constructor is furnished with the actual parameters of `CthreadCreate()`. The return value serves as the `Cthread` handle. This handle is nothing but an "opaque pointer" to the allocated active object. In contrast to instantiation, finalization is requested by making a call on `CthreadDelete()` and passing the handle of the `Cthread` instance to be destroyed. The implementation assumes that this handle was previously delivered by `CthreadCreate()`. As shown in the figure, the handle is interpreted as a `Cthread` pointer.

6.4.4 Kernel-Level Active Object

It was mentioned above that `native` scheduling yields `thread` processing. This corresponds to the traditional way of modeling processes in a multitasking environment that distinguishes between the user and supervisor modes of operation. The process (logically) has two stacks, one for each operation mode. Depending on its execution state, the process is referred to as a *user process* (user mode execution) or as a *kernel process* (supervisor mode execution), with each mode executing on a different stack.

A `native` thread is a kernel process. It has its own stack only in the case of vertical isolation, that is, when NOI is implemented as a cross-domain call. In a single-tasking environment without any isolation measures, the `native` executes on the stack of its user `thread`. Thus, in these configurations the `thread` is processed in supervisor mode too. It shares the same address space with the nucleus (kernel).

The process model implemented distinguishes between teams and threads. The former implements a common execution domain for the latter. For this reason, at least two fundamental data structures are required to describe a single process. The modeling of these data structures is explained in the following two subsections.

6.4.4.1 Process Control Block

A *process control block* (PCB) is used for each `native` thread to keep track of all process activities. It is a common approach in operating systems design that many different kernel modules share the same PCB for performance reasons. These modules, for example, perform dispatching, scheduling, synchronization, communication and resource management. They all store their process-related state information in a single data structure. Concerning PCB members, the sharing is often disjunctive, that is, the kernel modules usually access only their own state information. However, they still have global knowledge about all the information stored in the PCB and are thus interdependent—they "*use*" [67] each other, which leads to a very poor internal system organization. The UNIX `proc` structure is a typical example of this PCB representation. Moreover, note that it is only the physical representation as a *flat data object* that yields performance—and not necessarily the logical one.

Subclassing is used to design the PCB and to introduce new process-related state information. Each class in the hierarchy is an *abstract data type* and thus allows state manipulation only via (*inline*) methods. Since multiple inclusion is assumed, instances of (hierarchically composed) classes have a *static representation*. The consequence is a flat data object representation of a single PCB and, at the same time, a highly modular kernel organization.

The class hierarchy used to model kernel-level active objects (i.e., to construct a PCB) also demonstrates how to virtualize a single physical processor as an object-oriented abstract processor (Figure 6.26). At the lowest level (i.e., the root of the inheritance tree), a thread of control is ordinarily represented by a program counter and other CPU registers. At a higher level, the process becomes an active object encapsulating a private thread of control.

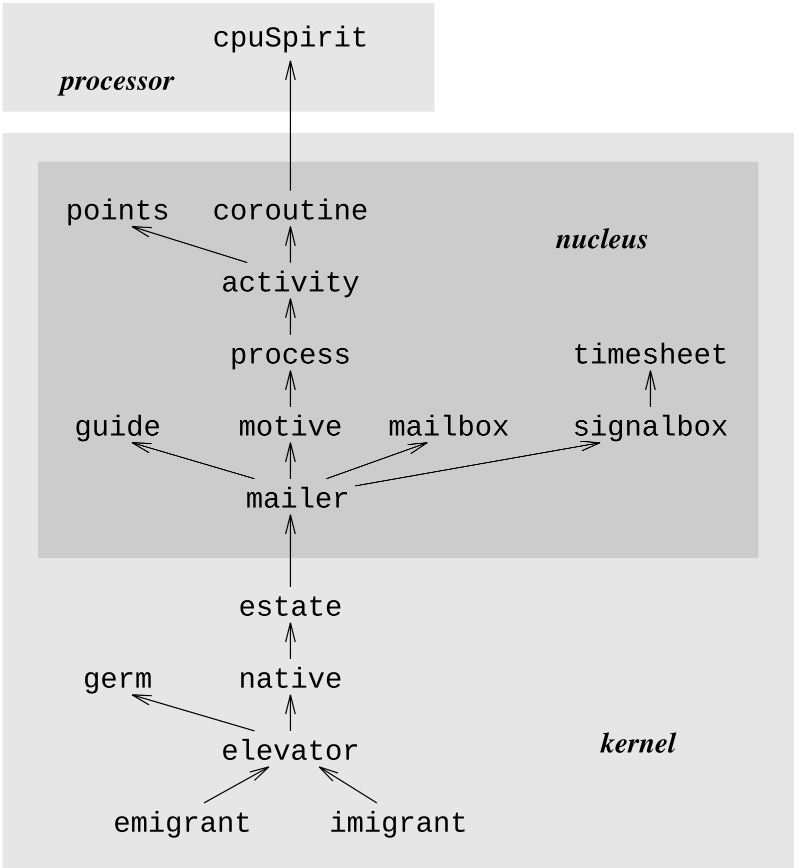

Figure 6.26: The process control block

The entire class hierarchy consists of ten levels (classes around the middle axis from top to bottom in Figure 6.26, with the top being the lowest level), each of which implements a *virtual machine* [36]. The corresponding class descriptions specify both the state (i.e., PCB fragment) and the operations of a virtual machine. The meaning of each level is as follows:

Level 0 *Abstraction from the physical processor.* Class `cpuSpirit` describes the structure and location of the area used to save/restore the nonvolatile register set upon process switches. Two models are supported. The first model assumes that the save area is located on the kernel stack of the thread. In this case, `cpuSpirit` stores the kernel stack pointer, which directly refers to the save area. The second model assumes that `cpuSpirit` is the save area. In that situation, `cpuSpirit` keeps track of the entire CPU register set, from which the nonvolatile registers are a subset only. That is, the save area is located on the kernel heap and not on the kernel stack.

Level 1 *Introduction of concurrency.* Coroutines are used as abstractions from a single-processor machine and provide the basis for low-level process switching. The `coroutine` abstraction is the proper mechanism. It provides the `resume()` method to exchange nonvolatile register sets. Independent of whether `cpuSpirit` stores a save area pointer or the save area itself, `resume()` always saves/restores only the nonvolatile register set as defined by C++.

Level 2 *Abstraction from non-object-oriented processors.* In `activity`, the autonomous method `action()` is declared as a pure virtual function. This class also inherits `points` to provide the entry into the active object. Method `operator()()` is redefined by `activity` and acts as wrapper routine for `action()`. The operator stops the processing of a thread when returning from `action()`.

Level 3 *Abstraction from the process dispatcher.* The dispatcher is adaptive and provides problem-oriented context-switching capabilities. Context switching may require saving and restoring processor registers other than the nonvolatile register set, for example, the register set of a floating point unit used by "number-crunching" processes. Class `process` applies dynamic binding at this level to reflect the various requirements. An extension of `resume()` is introduced. This extension keeps track of the globally declared `life` pointer to the currently active object.

Level 4 *Abstraction from the process scheduler.* The scheduler is adaptive and provides problem-oriented scheduling capabilities. Class `motive` applies dynamic binding at this level to introduce a per-process scheduler, thus providing a platform for a multischeduler environment.

Level 5 *Introduction of message-driven scheduling.* The `mailer` abstraction implements a refinement of the scheduler abstraction. A process is given a systemwide unique identifier and associated with a mailbox and a signalbox. The `mailbox` abstraction stores received but not yet delivered (i.e., consumed) message packets, whereas the `signalbox` abstraction keeps track of synchronization events that have been signaled by the team of the process. By means of `timesheet`, timeouts for threads waiting on events are managed. For this purpose, instances of `timesheet` are processed by the `timer` abstraction (see also Section 6.2.3). The unique identifier (i.e., `guide`) makes an active object hashable systemwide.

Level 6 *Introduction of kernel-level resource management.* Locally keeping track of all allocated threads of the node is done by the `estate` abstraction. More specifically, `estate` is used to remember all kernel-level resources owned by a thread. Hence, the `estate` destructor can ensure that these resources are released as part of the finalization procedure of the active object that implements the terminating thread.

Level 7 *Introduction of kernel-level active objects.* The `native` abstraction serves as the general basis for dynamically created active objects of any type. Constructor and destructor methods take care of the allocation/deallocation of the kernel stack of an active object.

Level 8 *Upcall linkage.* Class `elevator` stores the reference to a user-level `points` object. This object is used to invoke (i.e., bootstrap) the user-level active object. When a user-level thread is started and decides to transform into a user-level active object, this thread reads the `points*` instance variable and invokes `operator()()`. The `points*` value was passed down to the kernel by the `germ` object used in the construction procedure of a process gate (refer to Section 6.3.4.2). This `germ` object is used to initialize the `elevator` instance accordingly.

Level 9 *Upcall handling.* There are two abstractions at this level, distinguishing between user- and supervisor-mode threads. For a user-mode thread, `emigrant` keeps track of additional processor registers used to implement vertical isolation. For example, in order to transfer control from the supervisor to the user mode, the cross-domain call must be furnished with the user-mode software prototypes of both the stack pointer and the program counter of a user-level thread. Class `immigrant` is used for user-level active objects that will never leave the kernel space. In contrast to `emigrant`, `immigrant` objects are logically but not physically separated from the supervisor mode.

The design decision whether to distinguish between the user and supervisor modes is made at a very high level of the kernel design. Active objects always start processing in supervisor mode and may switch over to user mode afterwards. Objects that never leave supervisor-mode execution are considered kernel immigrants. Objects entering user-mode processing (i.e., kernel emigrants) are subjected to vertical isolation. In this case, a cross-domain upcall must be performed to transfer control accordingly. In either case, the object is given user-level visibility, making the PCB representation transparent even for kernel threads. This design decision is one reason for the fact that almost the complete kernel and nucleus implementation can also run as a guest level (see also Section 3.2) on top of a host operating system such as UNIX.

It was mentioned that `process` and `motive` are adaptive. This must be clarified in some respects. Both classes are "*virtually deferred.*" Instead of "virtual," "adaptive" is used as the prefix in the method declaration. This prefix is a preprocessor definition. It is set to "virtual" in order to generate a nucleus that supports (user-level) customization of the two PCB classes. If left unassigned, customization of these classes is not supported. Virtual functions are more expensive than nonvirtual functions. Since `process` and `motive` are low-level nucleus abstractions that are frequently employed by high-level nucleus abstractions, the decision to make these classes deferred was postponed.[25] The "adaptive" trick helps to generate a nucleus that supports applications whose focus is on very-low message startup times. On the other hand, a nucleus supporting applications that need to define and transparently integrate their own scheduling and dispatching strategy can easily be generated. These different nuclei become members of the kernel family.

6.4.4.2 Team Control Block

Threads are lightweight processes. Light weight results from the fact that related threads share the same address space, thus constituting a *team*. Therefore, intra-team thread switching does not cause the switching of address spaces. In order to model a team object, a *team control block* (TCB) is introduced. The TCB plays almost the same role as the PCB. In particular, a TCB keeps track of all the PCB definitions of the threads that are members of the same team. Conversely, every PCB contains information that uniquely addresses a TCB—every thread knows its team.

This motivates a structure in which TCB objects implement lists/queues of PCB objects, and PCB objects can be held in these lists/queues. More specifically, a single PCB may be held in three lists at a time. Figure 6.27 describes this structure. Note that the PCB part shown is only an excerpt from the complete implementation depicted in Figure 6.26.

In Figure 6.27, there are three classes that implement nothing but a linkage pointer. These are `competitor`, `colleague` and `kindred`. All three are derived from `chain`. The three linkage pointer classes have been introduced for renaming purposes only. The `kindred` class is used

[25] For example, a single blocking and local send message primitive causes two `motive` and one `process` invocation. In the default case, each of these invocations entails the execution of very few instruction sequences only. This makes a difference when using a possibly four times slower invocation mechanism (i.e., virtual functions). One would not follow the family idea if `motive` and `process` were deferred classes, although the application need not exploit this feature.

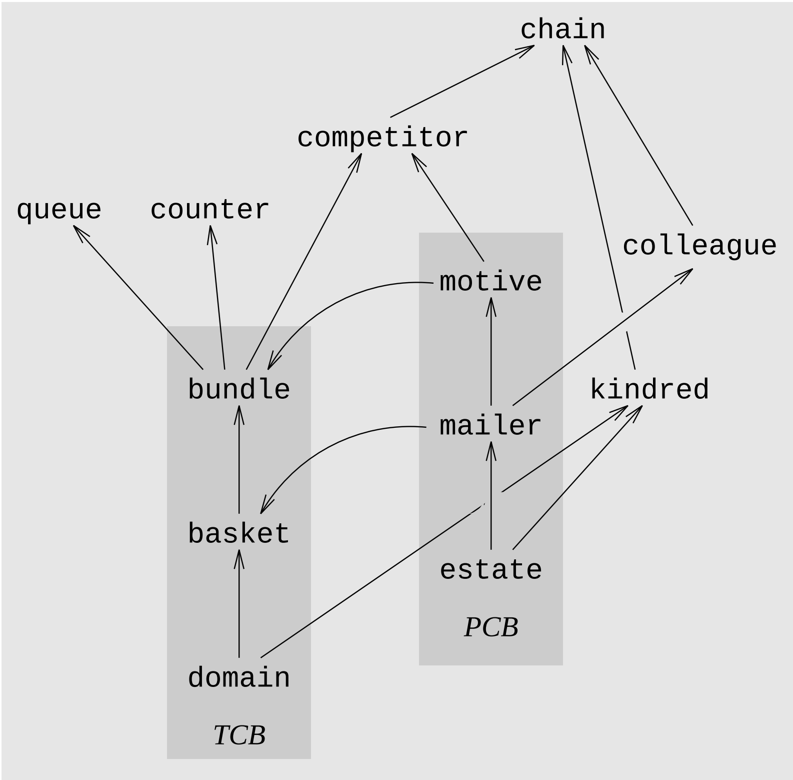

Figure 6.27: The team control block

to model a chain of all existing PCB objects. This class also chains up all known TCB objects. Thus, a `kindred` pointer is used for basic resource management. The `colleague` class serves to implement a stack of threads that are ready to receive messages for the entire team. By means of the `competitor` class, both the PCB and TCB can be held in per-team thread ready lists and per-nucleus team ready lists, respectively.

The `bundle` abstraction of a TCB basically implements a ready list of `motive` objects. Technically, the ready list is implemented by the `queue` abstraction inherited by `bundle`. In order to associate threads with bundles, the `motive` abstraction stores a pointer to its enclosing `bundle` object. Threads are allowed to change between bundles. This is accomplished by switching the `bundle*` entry contained in `motive`. By inheriting `counter` properties, the actual number of threads associated with a `bundle` instance can be remembered. A `bundle` object is released only if no more associated threads are encountered.

As mentioned above, a number of threads may have been made ready to receive messages for the entire team. These threads are put into a `basket` object. The `basket` abstraction implements a stack of ready-to-receive `mailer` objects. Whenever a message is sent to a thread that is not ready to immediately receive the incoming data, the nucleus tries to dispatch a substitute thread. The nucleus dereferences the `basket*` entry of the originally addressed `mailer` object (i.e., receiver thread) and takes the next ready-to-receive `mailer` from the top of stack. The message packet is made pending for receive (in the `mailbox` object associated with the original receiver thread) only if the basket is empty. Threads make themselves ready to receive messages for the team. They invoke a nucleus function which pushes the current thread (i.e., `mailer` object) onto the `basket` stack.[26]

In contrast to the `bundle*` member of `motive`, the `basket*` instance variable of `mailer` is invariant as long as the active object exists. This linkage is established at `mailer` construction time. Usually, both TCB pointers refer to the same team object. These pointers differ only when teams define multiple scheduling domains, for example, to map a single team onto a shared-memory multiprocessor system. In that case, all threads of the same team are associated with the same `basket*` value but (groups of threads) may have different `bundle*` values. This also enables the moving of a thread back into its original scheduling domain defined at `motive` construction time: the `bundle*` instance variable is set to the value of the `basket*` instance variable.

On this basis, `basket` also implements a TCB fragment which remains valid for the entire lifetime of all threads belonging to the same team. This property is inherited by the `domain` abstraction in order to perform basic resource management functions. By means of `domain`, all

[26]The nucleus identifies the current thread of control by exploiting the `life` pointer. This pointer is of type `activity*` and is properly coerced to become compatible with `mailer*`.

threads of the same team are known: the corresponding `estate` objects are held in a queue. A `counter` instance variable keeps track of the actual size of the queue. The `kindred` linkage is used to chain up `estate` objects. Similarly, this linkage pointer is also used to keep track of all currently known `domain` objects. Just as in the case of the ready list, a two-level resource list is formed: the per-nucleus team list and the per-team thread list.

6.4.5 Extensions to the Minimal Basis

The discussion of how active objects are modeled has covered the default case only. A "minimal but perfect" process implementation was shown. This implementation is sufficient for quite a large number of parallel applications. However, it needs to be extended to provide a more elaborate platform for functions such as fault tolerance, global scheduling and monitoring, just to name some examples.

Extensions are made on a subclass basis. The number of extensions needed, as well as where these extensions are to be made, depends on whether or not software compatibility and kernel transparency are major user requirements. By sacrificing compatibility and transparency, extending the minimal basis becomes straightforward. In this case, either `emigrant` or `native` will serve as a superclass for the new (kernel-level) process abstraction. Maintaining software compatibility and kernel transparency makes the introduction of new process abstractions slightly more complicated. This is a typical situation in which an operating system designer, due to the lack of proper programming (language) support, is sometimes forced to introduce several "obscure" things. Structural measures will have to replace clean programming features. Such a pure structural solution will be discussed in the following.

Extending the process model is done by introducing new classes for both the user and kernel levels. Although dedicated to different levels of abstractions, these new classes correspond to each other. That is, a new *user-level process abstraction* usually calls for a corresponding *kernel-level process abstraction.* In addition to subclassing, a new *linkage component* is required. This component is particularly designed to bind user- and kernel-level instances and thus perform a linkage between the different levels of abstractions. A fictitious class ensemble is shown in Figure 6.28.

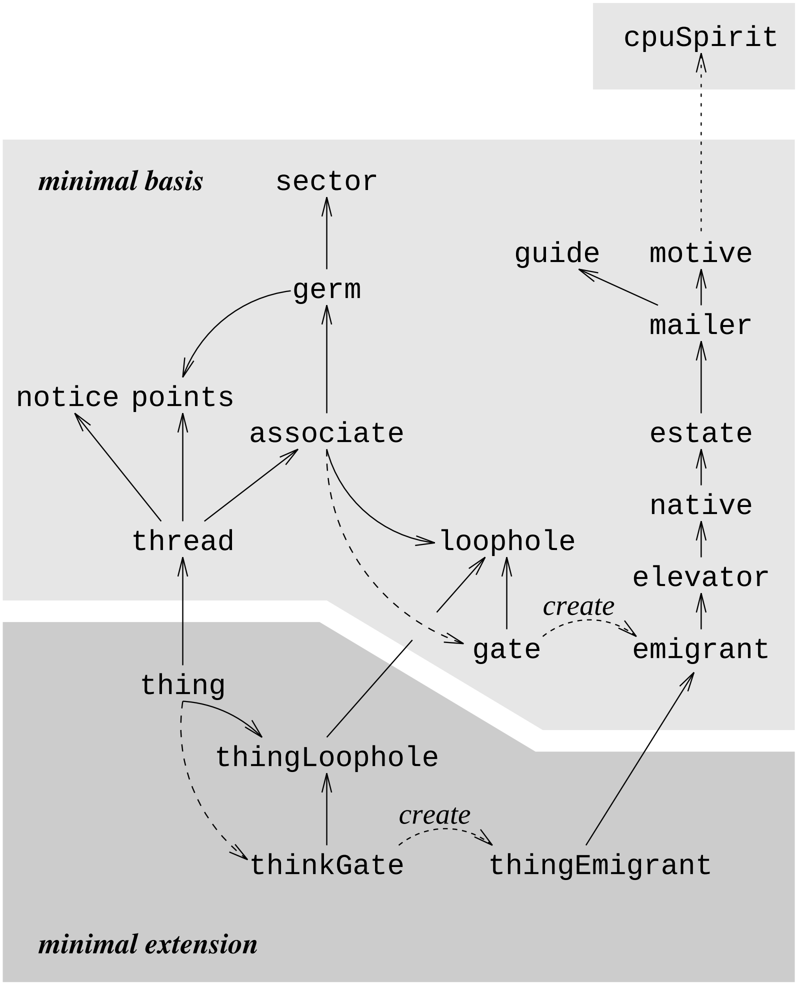

Figure 6.28: Adding a new process class

As shown in the figure, four new classes have been designed. The `thing` class specifies the new user-level process type and redefines `action()`. In particular, the constructor signatures as well as the structure of `thing` are entirely user-defined. Similar considerations hold for `thingEmigrant`, which is nothing but the kernel-level continuation of `thing`. The `thingEmigrant` abstraction may, for example, adopt all `motive` methods, thus introducing a new scheduler instance. In addition, to implement some priority-based scheduling scheme, `thingEmigrant` also may add instance variables specifying the urgency of a thread implemented by an instance of that class. If demanded by the implementation of the new process type, a `thing`-related TCB may also be designed. This could readily be done by introducing a `thingDomain` class derived from `domain` and associating this new class with `thingEmigrant` (not shown in Figure 6.28).

The classes `thingLoophole` and `thingGate` play a role similar to the `loophole/gate` pair introduced earlier (in Figure 6.20). They make user-level process abstractions independent from kernel-level process abstractions and actual kernel configurations (single- vs. multi-address-space model).

In the example presented, `thing` behaves like `associate`, by distinguishing between a normal and dual-object representation of `loophole`. The `thing` class only stores a `thingLoophole*`, whereas the `thing` constructor (or a customized `settle()` method) allocates a `thingGate` object and assigns a `thingGate*` to the `thingLoophole*` instance. By deriving `thingLoophole` from `loophole`, the basic features of name server and forwarding identifier installation are shared. Moreover, `thingLoophole` is free to add new features to this fundamental process gate implementation. The features will be used, for example, for communication with the associated `thingEmigrant` instance. By means of `thingGate`, all these features are then logically implemented and carried out with ROI. Taking into account the other abstractions shown in the example, `thing` and `thread` thus share the following commonalities:

- Communication gate and forwarding identifier management
- Kernel-level process management, except for those `process`, `motive` and `native` features redefined by `thingEmigrant`
- User-level process management, except for those `points` and `thread` features redefined by `thing`

The `thingGate` constructor is responsible for allocating a `thingEmigrant` instance as requested by the user-level `thing` constructor. At this point, dynamic binding will be applied to adopt `motive` and `process` features, for example. Triggered by the `thing` destructor, the destructor of `thingGate` takes care of releasing the associated `thingEmigrant` object accordingly.

This example demonstrates the strength of object orientation. The strength is that a really open system structure emerges. There is no need to overload fundamental abstractions such as `associate` in order to deal with a (possibly large) variety of problem-specific process types. This makes these abstractions independent of future and more elaborate system components and generally improves the "uses relation" [67] between the system modules. In particular, function interfaces need not be designed to cover any possible application demand—which is an impossible undertaking in any case.

The differences between the `thread` and `thing` levels of abstraction are mostly caused by the different problem levels. The `thing` level does more than just introducing new features by means of dynamic binding. Rather, this level must also take care of the possibly completely different data abstraction introduced by `thing` and the passing of instances of this abstraction down to the kernel (i.e., to `thingEmigrant`). In principle, that new abstraction is incompatible with the data abstraction handled by `associate` and passed over to `loophole`—the kernel may simply expect a different parameter set when creating either an `emigrant` or a `thingEmigrant`. However, in many cases, the actual algorithms used to implement the linkage component (represented by the `loophole` specializations shown in Figure 6.28) are the same. They only differ with respect to the data types. This motivates making `associate` and `thing` *generic classes* in order to allow for the sharing of even the same linkage component implementation. The modeling presented in Figure 6.28 assumes no generic classes, but rather a "hand-made" solution to this problem. Thus, this class ensemble is even processable by (C++) compilers that do not support generic classes.[27] Also note that the exploitation of generic classes will not result in a different

[27]In fact, the current PEACE implementation necessitated this approach, since the C++ compiler being used does not support "templates." As already mentioned earlier, using `generic.h` as a means of faking templates was strictly declined for

class hierarchy than the one presented in Figure 6.28. Both `thing` and `associate` would then share only the same class implementation.

6.4.6 Installation of Active Objects

The construction of an active object is carried out top-down. First, the user-level and then the kernel-level active object are created. Finally, the binding between both instances takes place. From the user point of view, instantiation of active objects is done solely by executing the various constructor methods. This is true for most of the classes used to model an active object, but it does not hold at the user/kernel boundary.

Before the kernel-level active object can be instantiated, construction of the user-level active object must already be finished. This is not satisfied when `associate`, which implements the user/kernel linkage, automatically creates a `gate`, and thus `native` thread, instance by executing its constructor method. The problem is that once the kernel `native` (scheduling the user `thread`) has been created, an upcall to the user level may occur immediately. When this upcall happens depends on the scheduler and the current load. Carrying out the upcall, however, must rely on the fact that dynamic binding of the `thread` instance was finished. Two user-level virtual functions are involved in executing the upcall. The first one is `operator()()` and the second one is `action()`, with the former invoking the latter. However, both virtual function calls will only succeed when `thread` instantiation has been completed.

In C++, object construction happens bottom-up. First the superclass and afterwards subclass constructors are called. Object destruction works vice versa, that is, top-down from the subclasses to the superclasses. As was shown in Figure 6.23, `associate` is a superclass of `thread`. Hence, the `associate` constructor gets executed before the `thread` constructor. When requesting the installation of a kernel-level active object from within the `associate` constructor, `thread` activation might take place even though the `thread` instance is not yet completely constructed.

portability reasons.

The solution to this problem calls for a dedicated method, namely `settle()`, which is in charge of active object instantiation. The `associate` and `thread` constructors only set up the parameters needed for kernel-level active object creation. These parameters are remembered by `germ` and used by `settle()` in order to continue process creation. That is, `settle()` will be called when `thread` construction, and thus the binding of both `operator()()` and `action()`, has been finished. The earliest point in time at which `settle()` can be invoked is when the `thread` constructor gets executed. However, this would assume that subclasses of `thread` are not employing dynamic binding for whatever reasons. As a consequence, `settle()` can only safely be called by a constructor of the most specialized class of `thread`. Placing the `settle()` call in that constructor retains the user-level view of how active object instantiation takes place, that is at object declaration time.

6.5 Messages

In addition to processes, the nucleus has to implement message abstractions in order to enable communication between threads. These abstractions distinguish between a locally and remotely operating nucleus. The following subsections discuss both nucleus implementations. Subsequently, the modeling of the protocol data units of the inter-nucleus protocol is explained. This is used as an example of how to generally structure communication protocols by means of subclassing.

6.5.1 Locally Operating Nucleus

The nucleus implements packet-based synchronous as well as segment-based synchronous and asynchronous communication. Packet communication is provided by `parcel`. Segment communication is provided by the `charge` and `region` abstractions, respectively. In the following, the description of the class hierarchy implementing a locally operating nucleus focuses on packet communication only.

Methods of the `parcel` class support rendezvous-based communication patterns. The `request()` method is used by a sender thread to send a small fixed-size `packet` object to a receiver thread. The receiver thread must then invoke `receive()` to request message delivery. It may invoke `respond()` to pass a `packet` object back as response and thus terminate the rendezvous with the sender thread. The receiver may also employ `forward()` to relay the received `packet` object to another thread or to itself. This suggests a fundamental subclassing relationship between `parcel` and `packet`. The `parcel` abstraction inherits `packet` and introduces the methods by means of which `packet` (i.e., `parcel`) instances can be exchanged between threads. Figure 6.29 illustrates this class hierarchy.

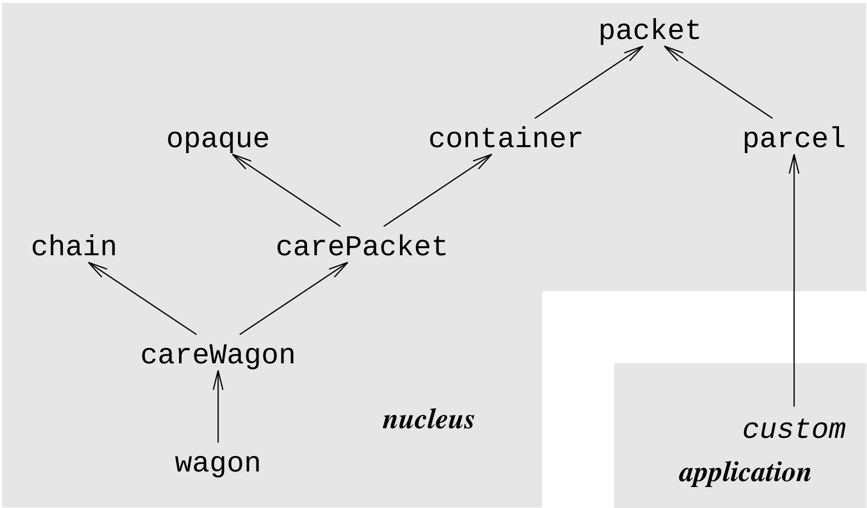

Figure 6.29: Networkless nucleus

Usually, in order to access the packet communication service the user designs a `custom` class and derives this class from `parcel`. Consequently, both application and nucleus are implicitly dealing with message objects of the same size. The objects are all of `packet` type.

Every thread possesses a private message buffer, which is allocated at thread creation time. The abstraction from this buffer is implemented by `wagon`. As can be seen from Figure 6.29, `wagon` also (indirectly) inherits `packet` properties, and thus is at least as large as a single `parcel` instance. The wagon class is used for nucleus configuration purposes. It distinguishes between a locally and remotely operating nucleus. As will be seen in the next subsection, wagon is derived from a different class when the nucleus is designed to perform global, not only local communication. All nucleus interface functions (such as the parcel methods) are working on wagon objects. These functions are thus (partly) independent of either nucleus configuration.

The base class of wagon, careWagon, is introduced to make kernel-level message buffers queuable. This is achieved by derivation from chain and a base class describing a single message packet. At this level, a kernel-level message packet is implemented by means of carePacket. This class was introduced to add to every message packet the sender identification as header information. The placeholder for the sender identification is of opaque type. That is, it may be either a weak or strong unique identifier (refer to Section 6.3.1), depending on the actual nucleus configuration (i.e., kernel family member).

A carePacket also "is a" container of sector objects. A sector object is a segment descriptor, and stores base address and length in bytes of a memory region. The container class specifies as many sector elements as can be stored in a single packet object. Thus, container makes packet an array of sector elements (Figure 6.30).

```
1: #include "packet.h"
2: #include "sector.h"
3:
4: class container : public packet {
5: public:
6:    sector& operator[] (int slot) {
7:       return ((sector*)this)
              [slot % (sizeof(packet) / sizeof(sector))];
8:    }
9: };
```

Figure 6.30: A segment container

This trick serves to support scattering/gathering of (a limited set of) message segments. Each of these segments is then described by a sector element. As shown in Figure 6.30, operator overloading (line 6) is employed to make `sector` elements accessible in an array-like manner. The *subscript operator* ensures array bound integrity. Moreover, this operator coerces a `container*`—which is the default type of `this` in this case—to appear as a `sector*`. This is a "dirty" but hidden cast and guarantees that the returned pointer (i.e., `sector&`) refers to a valid `sector` element within a `packet` object.

6.5.2 Remotely Operating Nucleus

As was mentioned above, the distinction between a locally and remotely operating nucleus is made by `wagon`—and, of course, by the `parcel` (and `region`) implementations dealing with `wagon` objects. As shown in Figure 6.31, this distinction is transparent to the application.

In this class hierarchy, `niceWagon` plays the same role `careWagon` did in the locally operating nucleus. It basically makes a NICE packet manageable by the `mailbox` abstraction (see also Figure 6.26). The `niceWagon` abstraction hides the strategy of the NICE packet implementation—keeping it in single- or double-linked lists, for example. In the case considered here, the packet is held in single-linked lists. This is achieved by making a `niceWagon` compatible with a single `chain` only.

With `nicePacket`, the maximal size of a NICE packet is specified. The initial size of the free list of receive messages buffers is determined by the CLUB layer (i.e., the network device driver). Upon startup, this layer creates as many buffers as are needed for the efficient handling of incoming message (i.e., NICE) requests. Therefore, CLUB must know the packet size demanded by NICE. A data type of exactly that size is introduced. CLUB refers to the `nicePacket` data type when creating message buffer objects.

A single *network data unit* is described by NICE, COSY and CLUB data structures, as indicated in Figure 6.31. Since NICE controls process-level and networkwide message passing, some sort of process identifier specifying the source and destination of a message transaction, respectively, is required. A receiver process is identified by the `opaque` field, whereas the sender process identification is encoded by a specific NICE data structure, which is explained in the next subsection. The important aspect at this point is to be aware of the fact that

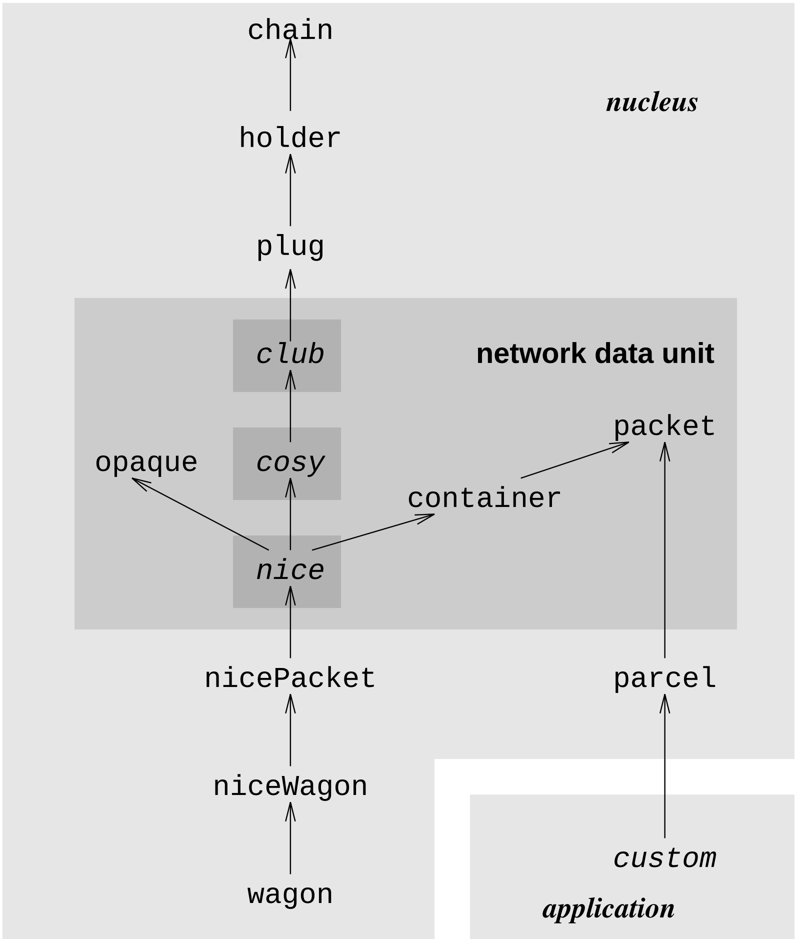

Figure 6.31: Networking nucleus

`opaque` serves a similar purpose, as discussed along with Figure 6.29. It is nothing but a placeholder for a communication endpoint address of a message transaction—in this case, however, specifying the target and not the origin. The abstractions `container` and `packet` play exactly the same role as mentioned in the previous subsection.

Section 4.3.2.2 explains that both message buffer descriptor and message buffer are allocated in one chunk to ease kernel-level buffer management. This makes a network data unit, which is typically held in the message buffer, a part of the message buffer descriptor. The descriptor is implemented by the `holder` abstraction. A `holder` instance is stored in either of three lists: the *free list*, the *I/O list*, or the per-process *receive list*. At any given point in time, this instance is in only one of these lists. The structure of a message buffer descriptor (i.e., `holder` instance) was already discussed in Section 4.3.2.2 and is not explained again here. It was mentioned that the so-called *header tank* refers to the message buffer that stores a *protocol data unit* (PDU) and the message buffer itself is adjacent to (i.e., part of) the message buffer descriptor. This structuring measure is made to generally ease buffer management, while hiding the specific buffer organization from low-level driver modules. There is always a one-to-one correspondence between a buffer descriptor and a buffer object.

Concerning the class hierarchy shown in Figure 6.31, the base address stored in the header tank of a `holder` instance points to the CLUB fraction of the network data unit. The size entry of that tank determines the actual length (in bytes) of the stored network data unit. Note that the (single) inheritance path in the design of a network data unit as depicted in the figure places CLUB data at the head, COSY data in the middle and NICE data at the end of a nucleus PDU. This corresponds to the conventional understanding that lower-level PDU information is followed by higher-level PDU information to ease the processing of incoming message streams.

Exploiting subclassing in the design of network data units, whereby each of these units is part of a message buffer descriptor, introduces some minor obstacles. This is because subclassing (on a *multiple inclusion* basis) leads to a flat object representation in memory. In the case discussed here, the network data unit is framed by data sets that are usually never intended to be sent across the network. More specifically, as `nicePacket` and its subclasses do not add further instance variables, the network data unit in this particular case makes up the second half of a message buffer descriptor.

Network data units are meaningful only in the process of executing a certain communication protocol. On the other hand, protocol processing relies solely on information exchanged across the network. That is, local management information such as queue pointers must be excluded from actual PDU objects. Management information that is of local significance at either the sending or the receiving node may thus be kept in subclasses of NICE abstractions or superclasses of CLUB abstractions, but never inbetween.

In the case of C++ (and other object-oriented programming languages), there is also the potential of "hidden" management information. An example is the virtual function table pointer contained in classes subject to dynamic binding. That pointer is dereferenced by the runtime system in the course of calling a (pure) virtual function. Carrying this pointer in a PDU does not only introduce unnecessary data transfer overhead, but may also cause serious integrity problems when protocol processing should exploit dynamic binding on a per-PDU basis. At the receiving site of the PDU object activated via a virtual function call, the corresponding virtual function table is to be referenced. However, the table addressed by the received virtual function table pointer is actually stored at the sending site. In the worst case, a memory protection fault might occur when the runtime system tries to apply this pointer. This fault may happen while executing the communication interrupt-handling procedure, and thus stands a good chance of crashing the nucleus (i.e., node). However, the other extreme may also hold, namely, successfully calling a virtual function—the question is only whether this was the right one.[28]

In order to clearly separate local management information from PDU data, the `plug` abstraction is introduced. This abstraction is used to merge site-dependent management data units with site-independent network data units and to locally keep track of both types of information by employing only a single buffer object.

6.5.3 Inter-Nucleus Protocol Data Units

As an example of how object orientation can be used to design communication protocols, the modeling of PDU objects required for carrying out the inter-nucleus protocol will be described. The function of the inter-nucleus protocol layer was explained in Section 2.5.3. Basically,

[28]Carrying virtual function table pointers within a PDU may also be intended by the nucleus. For instance, the side effect described will be a simple implementation trick to model active messages [90] in an object-oriented context.

this layer translates the nucleus service primitives into networkwide message-passing operations. The layer implements a *network independent communication executive* (i.e., NICE), by means of which various communication paradigms can be realized.

The minimal basis of communication functions provided at the nucleus interface is defined by three abstractions. These are `ticket`, `parcel` and `region`. The `ticket` abstraction serves to translate systemwide unique object identifiers into communication endpoint addresses. These addresses are demanded by `parcel` and `region`. The `parcel` abstraction implements synchronous and packet-based interthread communication, whereas asynchronous and segment-based interteam communication is carried out by `region`. The primitives provided by these three nucleus abstractions are implemented by specifically designed NICE abstractions. Every NICE abstraction is represented by a (C++) class. Figure 6.32 gives an overview of the resulting class hierarchy implementing the NICE layer.

6.5.3.1 Class Selection

The fundamental abstraction shared by all other NICE abstractions is `nice`. This class implements the interface to COSY and encodes the NICE function to be carried out by the receiving nucleus. The "function code" is nothing but an identification of a `nice` specialization. Thus, this code refers to NICE classes that are (directly/indirectly) derived from `nice`. For each NICE class shown in Figure 6.32, there is a unique *class code*.

An upcall from COSY to NICE results in the invocation of the NICE-level upcall handler `nice::operator()()`. This function interprets the class code and passes over control to `operator()()` of the class encoded in the NICE packet. In fact, the only purpose of `nice` in the receive process is to delegate upcall handling. Dynamic binding on the basis of (C++) virtual functions is not applicable in this case. This would assume that the text/data segment of the sending and receiving nucleus will be identical. The kernel family idea (see also Section 2.4), however, implies that different nucleus implementations are able to coexist, certainly not on the same node but still on the same parallel machine. As a consequence, a sending nucleus may have a different text/data segment

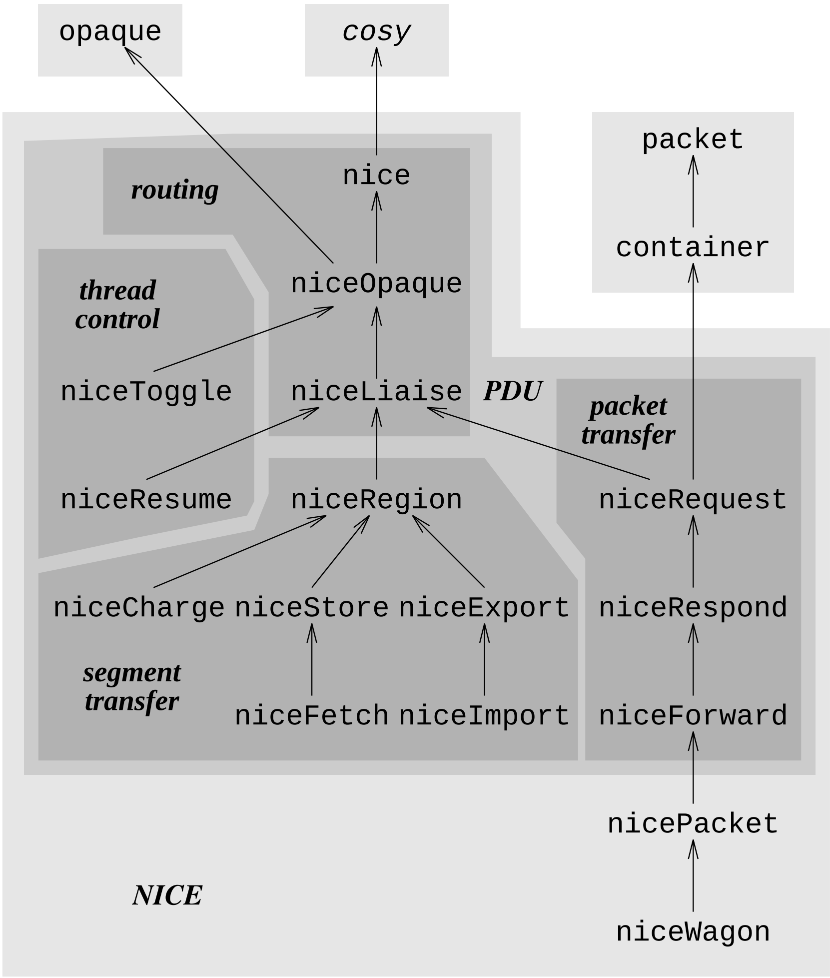

Figure 6.32: Nice class hierarchy

than the receiving nucleus. In terms of C++, this means that virtual function tables maintained by the sending node need not necessarily be located in the same memory region at the receiving node. However, the base addresses of these tables are stored as (hidden) runtime information in those objects that have been created from classes containing virtual functions. Using virtual functions in order to implement the upcall from COSY to a particular NICE "module" may thus result in a runtime exception at the receiving site. Preventing these exceptions requires "hand decoding" at the NICE-level, or the exploitation of a programming language that supports some sort of dynamic typing.

6.5.3.2 Routing

The target thread of a message transaction is specified by `niceOpaque`. This class only stores a single opaque identifier. As shown in Figure 6.32, the opaque identifier is included by inheriting `opaque`. An active "connection" between the sender and receiver threads is specified by the `niceLiaise` abstraction. The only purpose of `niceLiaise` is to extend `niceOpaque` by an opaque identifier specifying the origin of a message transaction. This class is fundamental to all other abstractions, because NICE always has to know the identifications of sender and receiver threads (and thus address spaces).

Besides describing the sender and receiver of a message transaction, `niceLiaise` is also responsible for *routing*. By providing the method `focus()`, specialized classes are enabled to verify the presence of the target thread. The function of `focus()` is to try to locate an active object by using the opaque identifier stored in the `niceOpaque` part of the NICE packet. A hash table lookup is performed. Three situations are handled by `focus()`:

- If the addressed object is known, the pointer to the process gate (i.e., a `guide*`) will be returned.
- If a remote forwarding identifier is associated with the specified target object, the `relay()` method of `niceLiaise` is invoked. This method sends the currently processed NICE packet back to the original sender, requesting the forwarding of the same packet to a different target. The new target is encoded in the origin field of the returned packet.
- If the addressed object is unknown, a hash fault is propagated back to the sender site.

Thus, the normal situation is to receive back a valid `guide*`. This indicates that the addressed active object is known on the current node.

The forwarding strategy described is chosen because the sender thread PCB keeps track of the identification of the actual receiver thread. Returning a NICE packet to the original sender site and having that site in charge of packet forwarding therefore also means updating the "blocked-on" field of the sender PCB. That PCB entry is used to remember a snapshot of the actual communication relationship of the sender thread and enable verification of whether unblocking of (i.e., replying to) a sender thread is allowed. In both cases, the rendezvous state of the sender thread is stored. The former case supports deadlock detection, whereas the latter case serves to implement communication security.

6.5.3.3 Thread Control

There are two classes used only for kernel-internal management purposes: `niceResume` and `niceToggle`. The `niceToggle` abstraction is used to force the addressed thread into a specific processing state. This state is described by a `niceToggle` instance variable. By exploiting `niceToggle`, a ready-to-run thread may be suspended (i.e., removed from the ready list), or a formerly suspended thread may be set ready-to-run again (i.e., placed onto the ready list). Thus, `niceToggle` supports the network-wide control of thread processing.

The `niceResume` abstraction exploits (i.e., inherits) `niceLiaise` to signal errors back to the sender site of a message transaction. That is, `niceResume` is used to force the premature end of the rendezvous between the two threads specified by the `niceLiaise` abstraction. Error codes are of negative values (i.e., of type `int`) and are carried by a proper `niceResume` instance variable. By handling a `niceResume` request, the addressed thread receives the specified error code and is set ready-to-run. Thus, a formerly established communication relationship is aborted. The resumed thread will return from the nucleus.

6.5.3.4 Segment Transfer

The notion of high-volume data transfer refers to the capability of exchanging arbitrarily sized memory segments between peer address spaces, even across the network. This requires the specification of a memory segment to be sent or received. A systemwide unique memory segment is described by `niceRegion`. Class `niceRegion` stores a `sector` member denoting the memory segment and takes advantage of `niceLiaise` to uniquely address that segment within a distributed environment.

Segment transfer distinguishes between exporting and importing a memory region. The export function is supported by `niceExport`, and the import function is supported by `niceImport`. Segment-based data transfer may be issued by a thread different from the original server of a rendezvous request. In order to check whether a segment transfer can be granted to the issuing thread, `niceExport` encodes the opaque identifier addressing the original server thread. The assumption is that the issuer and server are staying in some "trustworthy" relationship that makes the latter known to the former. This is implicitly true when both threads belong to the same team. The server identifier is then used at the target site of the segment to verify the validity of the issuer. At that site, the opaque identifier encoded by `niceExport` is checked against the blocked-on field of the destination thread storing the server identifier. However, note that all these checks are only carried out by a nucleus implementing at least network integrity (refer to Section 2.4).

Exporting a memory segment means directly transferring data out of the (virtual) source address space. For this purpose, the sequel tank of the `holder` part of the `niceExport` object is set up to refer to that segment.[29] Segment import basically means to instruct the peer nucleus to export a specific memory region. This is expressed by deriving `niceImport` from `niceExport`. Class `niceImport` adds the destination address for the exported segment. This address refers to the (virtual) address space of the importing thread. Having received an import request, NICE first uses the `sector` part of `niceImport` to set up the sequel tank of the enclosed `holder` object, then copies the remote segment base address into the `sector` instance, and finally blasts the segment described by the sequel tank to the importing thread. All these functions

[29]Note that every incoming or outgoing NICE request is stored in a kernel-level message buffer described by the `holder` abstraction. As can be seen from Figure 6.31, `holder` is part of the `nice` inheritance path. This particularly makes the `holder` object implicitly part of a `niceImport`/`niceExport` instance—`niceImport`/`niceExport` objects are compatible with `holder` objects.

are performed by the `blast()` method provided by the `niceRegion` abstraction. Thus, the upcall handler of `niceImport` solely verifies the validity of that transfer and then invokes `blast()`. The result of this call is the transfer of a `niceRegion` packet followed by the contents of the memory segment. In fact, `blast()` is implemented with `stream()` (see Section 4.3.4). At the importing site a `niceRegion` upcall takes place, accepting the incoming segment data and transferring it directly into the memory location specified by the `niceRegion` object.

A slightly different communication model is supported by the `niceStore` and `niceFetch` abstractions. Concerning the specification of the address space segments to be transferred, `niceStore` corresponds to `niceExport`, while `niceFetch` corresponds to `niceImport`. The difference is in the specification of the transfer capability. In the `niceExport/niceImport` case, the transfer capability is represented by the identifier of the server that accepted a rendezvous with the client. Thus, only threads staying in a rendezvous relationship with the client thread are able to access the client address space. In the `niceStore/niceFetch` case, the transfer capability is a `unique` identifier. This identifier serves as a general *transfer ticket* and allows even unrelated threads to access the client address space. These threads must possess a transfer ticket, which is then encoded in `niceStore` and `niceFetch` messages to enable the necessary checks at the client site. The model assumes that transfer tickets have been exchanged earlier by either carrying out a rendezvous or utilizing the name service.

Besides import or export of memory segments on a message-passing basis, NICE supports a relaxed model of *synchronization send* [52]. This model blocks a sender thread only as long as a segment transfer request has not been finished locally. Thus, the sender thread is not blocked until the receiver thread has provided a buffer in order to take over the transmitted segment. Rather, the model assumes the network itself to be the segment buffer. Sender thread processing is continued as soon as the segment is copied into the "network buffer." This type of message passing is implemented by the `niceCharge` abstraction. The segment to which the contents of the "network buffer" is to be transferred is again specified by `niceRegion`. The segment to be copied into that buffer is described by the `holder` part of a `niceCharge` instance.

6.5.3.5 Packet Transfer

In contrast to segment transfer, message passing on a packet basis follows the pattern of *request/response* models of interprocess communication. A *client* invokes a `request()` primitive. This primitive causes the transfer of a *rendezvous request* to the *server* and the blocking of the client. The server invokes `receive()` in order to accept the next request. Request processing is finished by a call on `respond()`, sending a reply message packet back to the client. Reply delivery then implicitly unblocks the client.

This communication pattern is provided by the `parcel` abstraction and implemented by exploiting specifically designed NICE classes. The names of these classes (Figure 6.32) correspond to the names of the primitives mentioned above. Class `niceRequest` implements the networking variant of `request()`. This class adds to `niceLiaise` the `packet` (i.e., `container`) object submitted by `request()`. As shown in Figure 6.32, `respond()` is implemented by a class that is a specialization of `niceRequest`. More specifically, `niceRespond` exploits `niceRequest` to encode the origin and target of the packet transfer and to be able to actually carry a packet to the client. In addition to this, `niceRespond` adds an `opaque` instance variable that identifies the original server to which the client has previously issued a `niceRequest` message. This entry is used at the client site to verify whether the responding thread is granted the right to terminate the client rendezvous. The server identifier serves the same purpose as discussed above in the context of exporting/importing memory regions.

Thus, in the request/response model implemented, `receive()` does not cause NICE activities. Rather, this `parcel` primitive is only of local significance. It solely depends on the actual COSY configuration whether the receive of a `niceRequest` packet entails additional network traffic, namely, when COSY packets carrying that NICE request are to be acknowledged. From the NICE point of view, only two network operations take place in order to execute a complete rendezvous: the issuing of a `niceRequest` and a `niceRespond` packet.

The last NICE abstraction to be mentioned is `niceForward`. This class implements the `forward()` primitive provided by `parcel`. The `forward()` primitive is used to relay a received `parcel` instance to the same, or a different server process. Forwarding a packet in a network context means to instruct the client site to re-issue the formerly stated request. Both the server address and the packet to be forwarded are explicitly specified by the server site. Thus, from the nucleus point of view, forwarding is the means by which the client is instructed to stay in a rendezvous with a possibly different server. This rendezvous may be established by sending a quite different data packet to that server. Hence, whether the same data is to be forwarded depends on the thread invoking the `forward()` primitive. The thread may or may not decide to forward the same packet that was previously delivered by the `receive()` call.

6.6 Program Families Revisited

The fundamental design principle behind the implementation structure of the kernel entity discussed in this chapter relies on the program family concept [67]. This concept served designing a family of parallel operating systems and a family of microkernels. An object-oriented implementation of the minimal basis for building the microkernel family was presented in the previous sections. The fundamentals of object orientation (i.e., inheritance and polymorphism) help implement program families in a very direct way. As was shown in the introduction (Section 1.4), the minimal subset of system functions in the program family concept can be adequately implemented by properly designed superclasses. Similarly, minimal system extensions are then implemented by subclassing and dynamic binding.

Program families can be implemented in many ways and are not only object-oriented. Nevertheless, the analogy between program families and object-oriented software construction makes an object-oriented implementation inescapable. This analogy does not imply, however, that an object-oriented design and implementation automatically yields a program family. The question is the complexity of the superclass acting as the minimal basis for a particular system component.

In the object-oriented design process, the system architect is faced with, basically, two considerations of the purpose of a superclass. The first aspect is establishing with the superclass the superset of system functions as well. In this case, a default implementation is specified, providing a general-purpose solution. Subclassing is then employed to develop customized implementations, providing special-purpose solutions. Thus, inheritance leads to a functional emaciation of a possibly complex superclass implementation.

The alternative is to design the superclass as a subset of system functions. On this basis, subclassing leads to a functional enrichment of a minimal superclass implementation. That is, special-purpose solutions are transformed into general-purpose solutions. Starting from a subset and elaborating a functionally enriched class hierarchy conforms to the family idea. This is because the implementation, as seen by derived abstractions, is always minimal. Selecting an arbitrary class from the hierarchy, the user will be offered maximum flexibility and burdened with only the minimal possible overhead.

Each of the class hierarchies described in this chapter was designed strongly bottom-up. That is, the most fundamental, common implementation is represented by a superclass. In the following, the modeling of a PCB (Figure 6.26) will be used to exemplify the object-oriented development of a microkernel family. Roughly speaking, this example defines a *thread family* and is shown in Figure 6.33.

The example sketched in Figure 6.33 exhibits a thread family consisting of four members. Thus, the original class hierarchy shown in Figure 6.26 serves more than just the composition of a PCB. Class `cpuSpirit` also provides the fundamental abstraction for implementing *compiler-intrinsic threads*. This class was introduced for implementing a placeholder of the nonvolatile register set of a thread. It also implements the proper abstraction for use by a compiler for *multithreaded architectures*. The compiler automatically creates threads and also knows about the actual processing state of these threads. Highly optimized thread-switching code can be generated. This code may be different for different switching points, since it depends on the actual use of processor registers of each individual thread pair. The `cpuSpirit` abstraction provides space for the entire processor register set. However, the entire register set need not be saved in every case. In particular, for a given thread the number of saved registers may be different for the different switching points. This situation is reflected by `cpuSpirit` in implementing only a register store but no context-switching method.

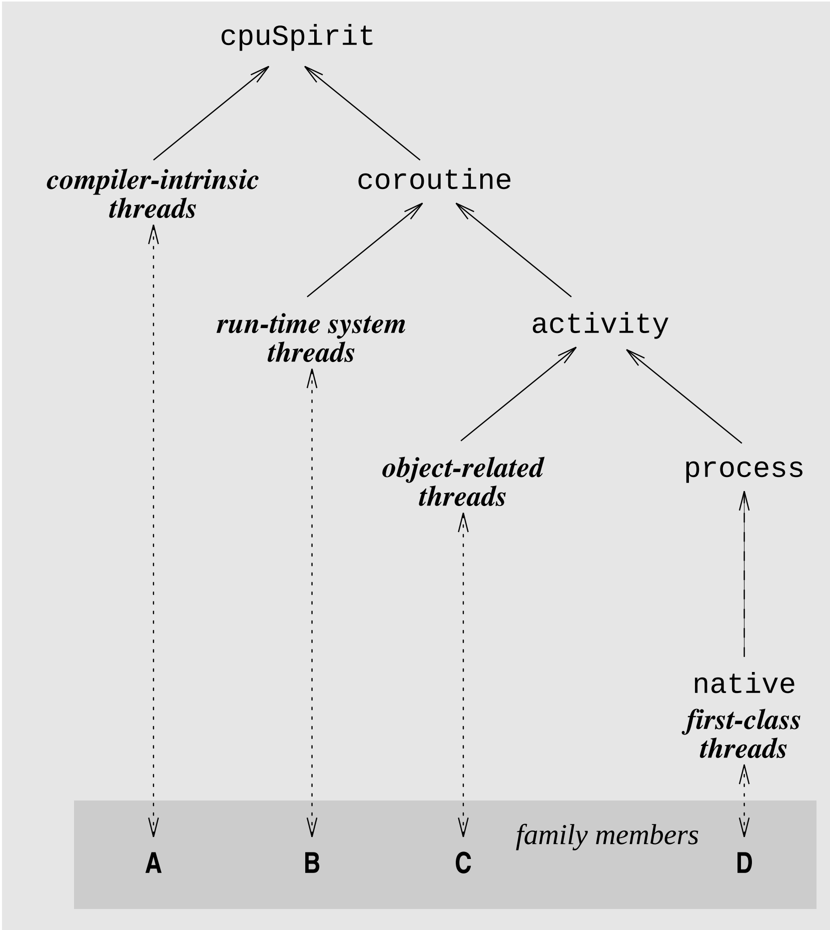

Figure 6.33: A thread family

A minimal extension to `cpuSpirit` is introduced by the `coroutine` abstraction. As mentioned in Section 6.4.4, `coroutine` solely introduces a method for switching the processor register set described by `cpuSpirit`. This measure introduces a platform for developing customizable *threads packages* (i.e., *runtime system threads*). An example of such a threads package is represented by the original class hierarchy shown in Figure 6.26. This package implements (kernel-level) active objects. The `coroutine` abstraction was introduced as a minimal subset of system functions for implementing various thread family members.

The next category of thread family members is established by the `activity` class. This level of abstraction introduces *object-related threads*, that is, threads encapsulated by an object. The `activity` abstraction provides measures for developing active objects. It does not prescribe the shape of an active object in terms of instance variables. Rather, `activity` solely sets up the fundamentals to be able to create application-oriented active objects. This level of abstraction provides a processor register store, a routine for switching processor registers and means for dynamic binding (i.e., the two virtual functions) for establishing arbitrary active objects. Again, the original modeling of a kernel-level active object as described in Section 6.4.4 introduces only one possible thread family member. Other members are quite possible, since the design decision of the properties of active objects has been postponed.

Finally, Figure 6.33 shows a fourth thread family member. This member stands for the kind of active object described in Section 6.4.4. It implements *first-class threads*, that is, threads that are known to the kernel or operating system. However, the `native` path defines only one possible way of implementing kernel-level active objects. There may be many other approaches.

This brief discussion, along with Figure 6.33, should reveal that, by carefully applying subclassing, a truly open system implementation may be the outcome. This holds for any software system, and in particular for operating systems or operating-system kernels. The class hierarchies introduced in this chapter were all designed with this aspect as the ulterior motive. In order to implement program families in general and operating-system families in particular, inheritance provides the means of introducing new fundamental system functions combined with a possibly extended object data space manipulated by the added functions. This is the strength of object orientation in the design and implementation process of operating-system families.

6.7 Summary

This chapter explains fundamental classes used to construct the minimal subset of system functions of the PEACE nucleus. Simple abstractions such as queues, stacks and chains are discussed, as well as more complex abstractions. The complex class structures described are used for trap/interrupt handling, object identification and hashing, and the modeling of processes and messages. The discussion is conducted on a very technical basis, trying to illustrate the actual object-oriented implementation of most of the PEACE nucleus.

Although the discussion is quite substantial, it is far from complete. Only in a very few cases are actual class implementations presented. Rather, a compromise is made. The goal is to provide a description that is detailed enough to get insight into how to employ object orientation in the implementation process of a real operating-system kernel. On the other hand, the description should also be suited to serve as a documentation of the source code of the PEACE kernel. The class ensemble presented can also be considered as an experiment in employing *pure object orientation* in the implementation process of a message-passing kernel family. In PEACE, almost every kernel feature is implemented by a class. This decision was made even though it was clear quite early in the design that there would be at most only one instance of the particular class. Classes are used not only to specify the common behavior of a possibly large number of objects, but also to serve as an abstraction mechanism similar to the *abstract data type* paradigm. A typical example is presented with the modeling of a process control block. Each nucleus/kernel layer introduces a new abstract data type. Subclassing is employed to compose abstract data types for higher levels. Another example is given with interfacing higher-level classes to low-level processor functions dealing with traps and interrupts (i.e., the processing of hardware exceptions). As the description shows, the outcome is a highly modular kernel implementation.

C++ is used as the programming language, but most of the class hierarchies presented are independent of a particular object-oriented programming language. Logical relationships between classes are described. Nevertheless, C++ had an impact on the design of a number of class hierarchies. In other words, it was less C++ than the concept of *multiple inclusion* that influenced the design. The essential implementation assumption was that the default modeling of class hierarchies will always end up in a flat representation of an instance of some subclass. That is, an object such as the process control block will never be represented internally as a dynamic data structure. When required, exceptions from this would have been explicitly stated by exploiting specific C++ features (deriving classes prefixed with `virtual`). In the current PEACE kernel implementation, no such exception must be made.

Chapter 7

System Abstractions

The previous chapter described the fundamental class hierarchies that implement the minimal basis of a family of parallel operating systems. An excerpt from the actual implementation of the two PEACE building blocks, nucleus and kernel, was presented. Because of the pure object-oriented representation, PEACE takes on the role of a *class library*. That is, the user sees a number of classes and is able to customize them in a problem-oriented manner. Customization is strongly supported by the polymorphic appearance of classes and methods.

Accessing system services by means of class inheritance has advantages and disadvantages. The advantage is that the user is able to tailor a given (default) implementation to his own specific needs. An application-oriented system is statically composed, since inheritance implicitly provides a global view of all classes (and thus functions) used by a given program. However, at the same time, this second aspect may be a disadvantage for the user. It makes applications (language and compiler) dependent on the underlying system implementation. A modification of the class hierarchy, or of only one of the classes, requires the recompilation of the application program or parts of it.

Object orientation provides the user with almost unlimited freedom to build his own special-purpose version. At the same time, it burdens the user with a possibly complete view of the internals of the system—the user generally sees to many details. This situation may be somewhat annoying for a user if openness of the underlying system is not needed. Furthermore, compelling the user to be concerned with all the details of the underlying object-oriented system would violate the family idea: in order to access system services, the user has no other choice but to exploit inheritance. Therefore, a compromise must be made, which is to establish two main branches of the operating-system family. One branch leaves the system open and maintains a strongly object-oriented view of all components. The other approach hides system internals (i.e., abstracts from the internal inheritance tree). As a consequence of this splitting, the entire family provides the user with two system views: one fully and one partly object-oriented view.

Hiding the internals of the system is accomplished by a "thin" mapping layer consisting of several problem-oriented classes. The classes of the mapping layer may be composed through inheritance. However, the inheritance tree does not include classes that implement the nucleus or kernel. Rather, the tree includes classes that hide the internals of the nucleus and kernel from higher layers. Each of these nucleus/kernel interface classes implements an *abstract data type* (ADT). For completeness, it must be mentioned that all the considerations just made with respect to the alternate nucleus/kernel ADT interface also hold for the POSE complex.

The purpose of the following sections is to give a brief impression of the complexity introduced by the mapping layer and demonstrate the modeling of problem-oriented user abstractions by exploiting the class ensemble that implements the nucleus. A case study of selected features provided by three different layers is made. From the bottom up, these layers are the nucleus and the kernel, the runtime system and the dual object subsystem. Before the description is given, a presentation of the fundamental execution model implemented by the nucleus seems appropriate.

7.1 Execution Model

The discussion on the minimal subset of system functions (Section 2.1) emphasized the need for two fundamental abstractions, namely *processes* and *messages*. It was also mentioned that both abstractions must be interlocked with each other, not only for performance reasons but also because of the nature of distributed-memory architectures. This leads to an *execution model*, which is characterized by a specific process concept and communication paradigm.

This section describes the "default" execution model that the minimal subset of system functions will provide. The model supports the execution of and communication between user as well as system processes. It is suited to enable cooperation between any sorts of active objects.

7.1.1 Processes

Figure 7.1 shows the PEACE process model. This model distinguishes between four concepts: (1) unit of execution, (2) unit of preemption, (3) unit of comradeship and (4) unit of distribution. The first three concepts are provided by the `thread` abstraction, and the fourth concept is provided by the `entity` abstraction. Both abstractions are implemented by exploiting kernel functions. The `thread` abstraction is thoroughly discussed in Section 6.4.2.

The *unit of execution* is represented by a *thread* (curled lines in Figure 7.1). A thread is a *lightweight process*. Several threads typically share the same address space. This address space encapsulates a single *unit of distribution*, which is referred to as an *entity*. In this model, a *task* is implemented as an entity. An entity may comprise only a single thread, which lets the entity appear as a traditional *heavyweight process*.

In order to support the parallel or pseudo-parallel execution of a single (user) task, an entity can be further subdivided into one or more groups of functionally related threads. One such structuring measure is the *bundle* (shaded boxes in Figure 7.1) and defines the *unit of preemption*. A bundle consists of one or more threads, with nonpreemptive scheduling of threads of the same bundle. Preemptive scheduling only happens between bundles. Thus, bundles define autonomous scheduling domains and are given a limited time quantum during which execution proceeds without scheduler intervention.

With the scheduling of bundles of threads that reside within the same entity (i.e., address space), in a shared-memory multiprocessor environment several processors may be available to execute different bundles. In this model, an application task consists of several bundles

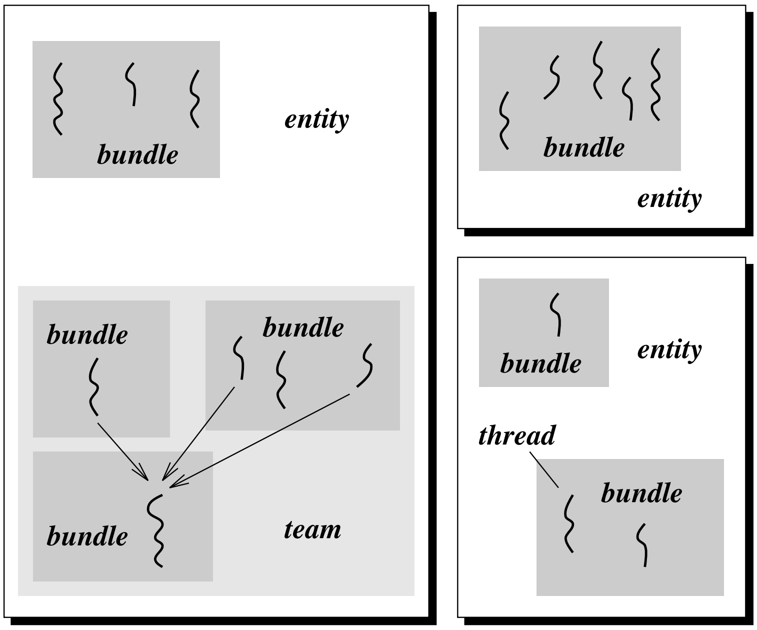

Figure 7.1: Process model

to take advantage of preemption and the shared-memory processor architecture.

The second entity structuring measure is the *team*. A team establishes the *unit of comradeship* [20] and primarily groups threads that are ready to receive messages. Threads are the targets of message transactions, and thread identifiers serve as communication endpoint addresses. Normally, a received message is delivered to the thread addressed by the message header. However, if this thread is not ready to receive the delivered message, the nucleus tries to dispatch a *comrade thread* instead. The dispatched comrade thread then receives and processes the message on behalf of the originally addressed receiver thread.

Comrade threads jointly build a team. They volunteer the message acceptance service of some other thread of the same team (expressed by the arrows in Figure 7.1). This implies that messages are sent to a team, although only a single thread serves as the target, and concurrent message processing is directly supported by the nucleus. In particular, which thread is in charge of message processing is transparent to the message sender.

As illustrated in Figure 7.1, a team may span several thread bundles. Thus, teams may be subjected to preemptive scheduling. They may also be executed in parallel on a shared-memory multiprocessor system. As with entities, the processing of teams is then nondeterministic. Only the threads of a bundle are processed deterministically.

7.1.2 Leagues

Communication between threads belonging to the same communication domain is unlimited. A communication domain is defined by a *league*. The league is a loose construction of threads that are in possession of the same *communication capability*. Usually, a league is built by a number of related threads whose entities are distributed over the parallel machine. By exchanging communication capabilities, the threads of these entities form one or more leagues. Thus, the league is a logical structuring measure employed by the threads themselves to establish "communication firewalls."

A communication capability is implemented by the `ticket` abstraction of the nucleus and defines a communication endpoint address of some thread. The communication endpoint address is created from the *communication gate* associated with a thread. As described in Section 6.3.3, communication gates are implemented by the `gate` abstraction interfacing to the kernel. Initially, threads and communication gates are allocated in a one-to-one correspondence. In this case, a communication endpoint address also stands for a systemwide unique thread identifier. During execution, threads may request allocation/deallocation of additional gates. By that means, threads are not only enabled to provide a number of communication endpoint addresses; the model also allows the migration of communication gates to different threads, and thus supports dynamic system reconfiguration.

Gate migration does not change the communication endpoint address of a thread; rather, it changes the association between a communication endpoint address and the corresponding thread. Since threads implement active objects, and active objects encapsulate operating system services, the capability to transparently change the gate/thread association is fundamental for replacing service implementations "on the fly"—the gate is then bound to an active object encapsulating the alternative service implementation.

7.1.3 Synchronization

Introducing the model of lightweight processes sharing the same address space calls for *thread synchronization* functions. Threads are "first-class citizens" of the nucleus, which implies that the nucleus must take care of thread synchronization. The thread synchronization model of the nucleus relies on event counter objects and is implemented by the `notice` abstraction. A notice event is a *unique bit pattern* and is technically represented by the C++ `this` pointer of the `notice` instance used for synchronization. That is, the bit pattern is a memory address and is assumed to be unique only within the (logical) address space defined by the entity of the threads. Figure 7.2 shows the fundamental synchronization patterns.

A synchronization event is indicated by executing the nonblocking `signal()` primitive. The event is propagated to all threads (of the entity of the signaling thread) expecting its occurrence. Event propagation means incrementing a counter and readying a possibly blocked thread. Therefore, the event is not lost if the expecting thread is unable to immediately react to the propagation.

The `pickup()` method is used to consume signaled events. In this case, thread blocking happens only if pending events are not encountered. In order to avoid deadlocking on an event never signaled, `pickup()` allows the calling thread to specify a *timeout* for the blocking period. An aborted `pickup()` call returns with an error code.

The announcement of readiness to receive a particular event is indicated by calling either `pickup()` or `define()`. A synchronization event is stored as a bit pattern in the PCB of a thread. With `pickup()`, the calling thread defines the PCB with the event bit pattern and consumes the specified number of events. Event definition takes place only if a new event shall be handled. Event consumption may cause thread blocking. In contrast to `pickup()`, `define()` will consume all events related to the currently defined event bit pattern and create a new bit pattern.

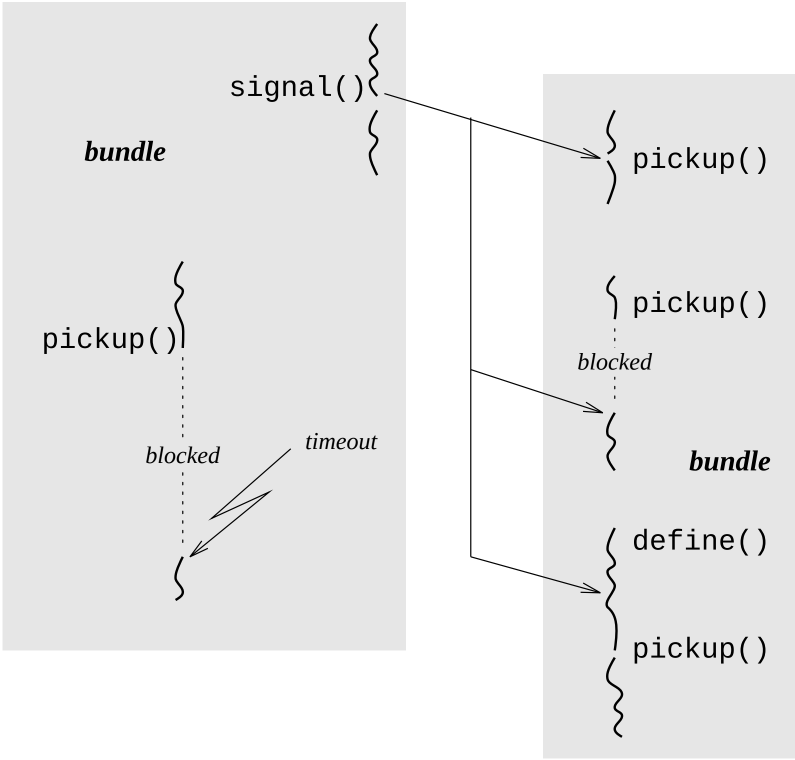

Figure 7.2: Thread synchronization

The thread may block when executing `pickup()`, but will never block in `define()`. The `define()` primitive allows the receive of signals even if `signal()` precedes its corresponding `pickup()` (see the lower right part of Figure 7.2).

Thread synchronization follows a *one-to-many* communication pattern. The same event may be signaled to several threads of the same entity. Also shown in Figure 7.2, event propagation may cross thread bundles—signaling is always entity-wide. When an event is propagated to a blocked thread of a bundle other than that of the signaling thread, the blocked thread is readied and immediately started. This corresponds to the strategy of preemptive scheduling on the basis of bundles instead of threads.

7.1.4 Communication

Two categories of message transactions are provided, based on packets and segments, respectively. Packet communication is synchronous, uses fixed-size data sets and follows a *request/response* pattern of interaction. In the case of networkwide transfers, a relaxed synchronous communication pattern is supported. Segment communication is asynchronous and uses variable-size data sets.

Packets are motivated by the communication protocol used to carry out networkwide message passing. Communication protocols are typically driven by message headers, also called *protocol data units*. These headers are set up by the sending and, once received, interpreted by the receiving site in order to determine how to continue message processing. Commands and communication endpoint addresses must be encoded and decoded, for example. For this purpose, at least, message headers must be temporarily buffered. Each of these buffers is dimensioned such that a limited amount of user-level information (namely, the packet) can be additionally stored. This allows the user to transfer a packet by executing a single protocol primitive only.

7.1.4.1 Packet Transfer

Packet communication is used primarily to model *client/server* configurations, whereas segment communication supports the transfer of arbitrarily sized data and application-oriented message passing. The model defines three main communication patterns. These patterns are described in the following paragraphs.

Rendezvous. By executing the *request/response protocol*, a packet is delivered when a *rendezvous* between the client and server takes place. For the client, the rendezvous lasts as long as the reply sent by the server has not yet been received. For the server, the rendezvous with a particular client lasts as long as the reply has not been sent. The server may have several rendezvous with different clients at the same time. Figure 7.3 illustrates the two possible execution sequences for carrying out a rendezvous between client and server thread.

Packet communication is implemented by the `parcel` abstraction

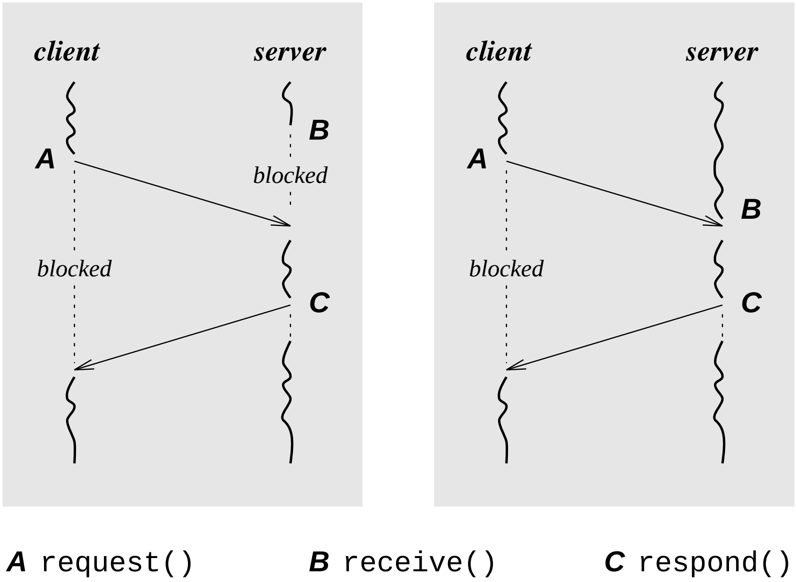

Figure 7.3: Rendezvous with a thread

of the nucleus. The client applies `request()` on a `parcel` object and blocks until a reply sent by the server has been received. At the receiving site of the request packet, the server typically invokes `receive()` to accept the rendezvous request issued by a client. A *nonspecific receive* is performed. The rendezvous is finished by executing `respond()`, which sends the reply and unblocks the client. Sending a reply to the client is a nonblocking operation. Server processing is delayed only for the time it takes for `respond()` to set up the packet transfer. At the receiving site of the reply packet, the client is set ready and, once restarted by the scheduler, returns from `request()`.

Every thread has a mailbox of delivered but not yet received packets. The mailbox is filled by the communication system of the nucleus and emptied by executing the `receive()` method. In this course, `receive()` blocks only if the mailbox of the calling thread is empty. This execution pattern results in the two possible server situations indicated in Figure 7.3. The left part corresponds to a *blocking receive* (i.e., `receive()` detects an empty mailbox). In contrast, the right part of Figure 7.3

shows a *nonblocking receive* (i.e., `receive()` detects a non-empty mailbox). Thus, the nonblocking receive occurs as a side effect of executing a specific interaction pattern between client and server. In other words, `receive()` will be a nonblocking operation only if the client/server protocol ensures a non-empty mailbox of the server. Either way, whether the server is ready to immediately receive a message packet is transparent to the client.

Blocking of either the client or server thread causes the invocation of the scheduler. The scheduler:

1. Tries to start a thread of the same bundle

2. If no other thread is ready to run, tries to start a thread of a different bundle

3. If no other bundle is ready to run, tries to start a thread (of a bundle) of a different entity

4. If no other entity is ready to run, enters the idle loop

Only the third step implies a complete, heavyweight context switch by activating a different address space. Idle loop processing usually does not imply a thread switch: the currently running and logically blocked thread executes the idle loop. Consequently, the client or server thread is idling (i.e., "busy waiting") only if no other thread is waiting for processor allocation.

Logically, clients issue a rendezvous request to a server team, not only to a single server thread. As described in the previous subsection, the receiving entity may contain comrade threads. A comrade thread will be instructed to receive the request packet if the originally addressed server is currently busy. In addition, once a rendezvous request has been accepted by a server thread, any thread of the same entity implicitly receives the capability to terminate the rendezvous by sending a reply to the client. That is, the execution of `respond()` may be performed by a thread other than the original server thread that had previously called `receive()` to accept the next rendezvous. This is illustrated in Figure 7.4.

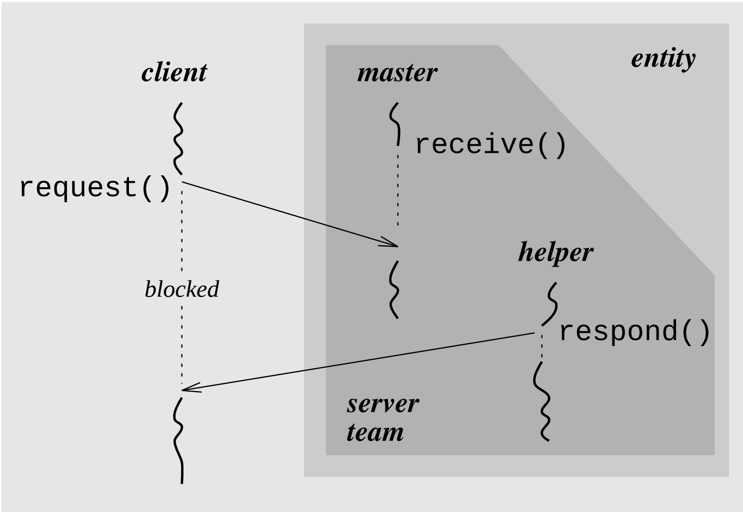

Figure 7.4: Rendezvous with a team

Figure 7.4 is the typical case for a *multithreaded server*. One *master* (possibly a comrade thread) is responsible for opening the rendezvous with the client. Further assume that finishing the rendezvous is possible only after some other external event has been processed. That is, the server implements a *state machine* that makes the termination of a rendezvous dependent on the actual server state. If the state does not allow for replying to the client, the master will call `receive()` to accept another rendezvous with a different client. This may cause the blocking of the master, and thus the inability to process the external event needed to terminate the pending rendezvous. In order to resolve blockings in this situation, the master is supported by one or more *helper* threads. The helper is able to process external events independent of the execution state of the master thread. Since the helper shares the same entity (i.e., address space) with the master, observing and manipulating the state of the server team is a straightforward undertaking. In particular, the helper may detect that finishing the pending rendezvous becomes possible and will thus be able to call `respond()` to send the outstanding replies. Note that this server protocol may cause several pending (i.e., open) rendezvous.

An example of such a configuration is a server also playing the role of a client, such as a file server issuing I/O requests to a disk server. The master thread then controls the (higher-level) user interface, while the helper thread controls the (lower-level) I/O interface. This allows the master to process incoming user requests independent of the execution state of pending I/O requests. Vice versa, the helper becomes able to finish I/O requests independent of the master execution state. That is, the file server acts as mediator between a number of user threads and one or more I/O servers.

Delegation. Under certain circumstances, a server may detect that a received rendezvous request is to be delegated, or forwarded, to a different server. For example, the object specified in the request message sent by the client is no longer managed by the addressed server. Rather, the object has been migrated to a different server without noticing the client(s). Object migration is a load-balancing issue and should be as transparent as possible to the client(s). Transparency is maintained if the addressed server forwards the rendezvous request without client intervention. This approach is illustrated in Figure 7.5.

In the figure, the server receiving the original rendezvous request issued by the client plays the role of a *router*. The router thread calls `receive()` to accept the rendezvous. Afterwards, it decides to delegate the rendezvous to the right *server* by calling `forward()`. The execution of `forward()` is nonblocking and causes the client site to re-issue the rendezvous request to the server specified in the forward message. Re-issuing the rendezvous request is not done by the still-blocked client, but by the nucleus at that site.

The reason for transferring a forward message to the client site is to update the client rendezvous state maintained there. In order to be able to verify the right to terminate a rendezvous, the client PCB stores the communication endpoint address submitted to the `request()` primitive to address the server. Usually, this address is the (unique) server identifier and will be checked against the identifier received in the reply packet. The identifier of the alternate server must be registered in the PCB to let a later `respond()` by that server succeed.

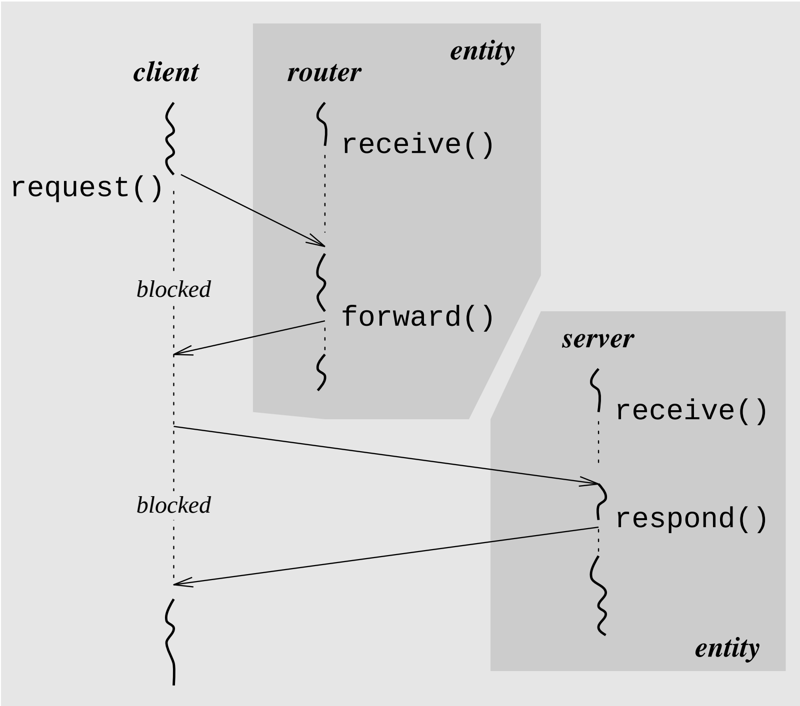

Figure 7.5: Delegating a rendezvous request

Requesting the client site to re-issue the rendezvous request basically means executing the `request()` protocol remotely. In particular, forwarding a rendezvous request causes an update of the existing logical connection between client and server. It does not necessarily mean transferring the same request message to the alternate server. In the case of Figure 7.5, by applying `forward()` to a `parcel` instance other than the one delivered by `receive()`, the router may decide to instruct the client site to transfer a different request packet to the server. Normally, the router forwards the request packet received from the client.

It is worth noting that *message forwarding* is a function that is also performed by the nucleus implicitly under certain conditions. Message forwarding will take place whenever the nucleus detects that the communication endpoint address of a thread refers to a *forwarding identifier* (see Section 5.3.5 for a detailed description). Forwarding identifiers are maintained by communication gates and can be exploited to implement location transparency of objects managed by server threads. Location transparency can be achieved by allocating, for every object, a communication gate, and associating this gate with the object server. Gate allocation and gate-to-thread binding will be the task of the server. By allocating the gate, an alternate communication endpoint address is introduced for the server. The unique gate identifier becomes the *alias* of a server managing a particular object. This identifier will be exported by the server to make the object globally accessible to the clients. The alias is used by the ROI layer and submitted to the `request()` primitive when operations are performed on the (remote-residing) object. Figure 7.6 shows a fictitious configuration in which location transparency of a *mobile object* is demonstrated by exploiting communication gates.

The upper half of Figure 7.6 (case a) depicts the starting point. A server thread manages two objects. From the viewpoint of the nucleus, the thread is represented by a PCB, and each of the two objects is represented by a *communication gate* (COG). Both COG instances are bound to the PCB. Note that the PCB also includes a COG. The COG is implemented by the `guide` abstraction inherited by the classes implementing a PCB. Refer to Sections 6.3.4.1 and 6.4.4 for a description of the class hierarchy used to model COG and PCB objects. In the example shown, the server thread has three communication endpoint addresses (i.e., unique identifiers).

Furnishing `request()` with the communication endpoint identifier of either of the two COG instances causes the nucleus to automatically forward the request.[1] The same pattern applies when a message is received from the network and delivered to the destination thread. Message forwarding does not take place when the communication endpoint identifier represented by the COG part of the PCB is used for server addressing. The lower half of Figure 7.6 (case b) shows the configuration in which one object has been migrated to a different server (from right to left). Location transparency is maintained by updating

[1] Forwarding applies to any message-passing primitive provided by the nucleus, not only to `request()`. Of the primitives discussed so far, message forwarding is also performed with `forward()` and `respond()`.

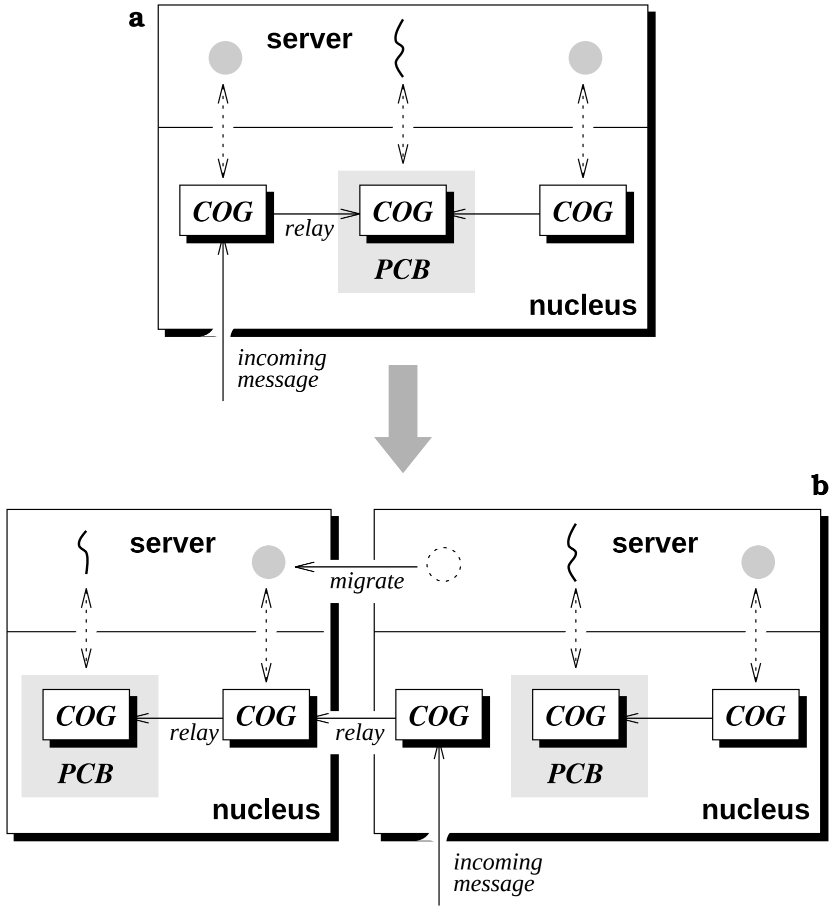

Figure 7.6: Location transparency of mobile objects

the gate-to-thread binding. This is accomplished by overwriting the forwarding identifier stored in the old COG with the unique identifier of the new COG. Updating forwarding identifiers is one of the (`loophole`) service functions provided by the kernel. The (right) server will have to call this service function after object migration has been confirmed (by the left server).

Relaxed Rendezvous. From the client point of view, the rendezvous model discussed so far defines a strongly synchronous control flow transfer. This model is extremely well suited for the implementation of RPC or ROI patterns. The `request()` primitive implements the calling sequence typically required to invoke procedures or objects in a distributed environment. It takes care of only the control flow transfer from client to server, and does not take care of the marshaling/unmarshaling procedure. Marshaling and unmarshaling of function parameters, for example, is the responsibility of the RPC/ROI stubs. The client/server interaction protocol is then driven by the stubs through exploiting `request()`, `receive()`, and `respond()`.

As was mentioned above in Section 7.1.4, packet communication is primarily motivated by the communication system. Protocol state information must be set up and transmitted by the sender site, and received and buffered at the receiver site. Buffering must be able to execute the communication protocol by interpreting the received protocol data unit. Taking the network into account, buffer space is not only limited to the node memories; the network itself may constitute a very large virtual buffer space. Furthermore, buffer space can be found in the network hardware interfaces relying on FIFO registers or caches. All these buffering capabilities can be utilized to implement a relaxed form of synchronous communication (Figure 7.7).

The `deliver()` method sends a packet to the network and returns when the packet has been accepted for transmission by the underlying communication system. Thus, `deliver()` performs only a setup operation and terminates. Under certain circumstances, the communication system will delay the acceptance of the packet to be transferred. This is the case when the available buffer space is exhausted. The communication system performs a flow-controlled data transfer. If a temporary shortage of buffer space is indicated, *back-pressure flow control* (see also Section 2.5.1) causes `deliver()` to be delayed.

Every thread has a private message buffer used for issuing rendezvous requests. This buffer is utilized by `deliver()` and forwarded to the communication system. Every invocation of `deliver()` first checks the processing state of the per-thread message buffer. Only when the buffer is inactive (i.e., not in use by the communication system) does `deliver()` execute the message setup procedure and hand the

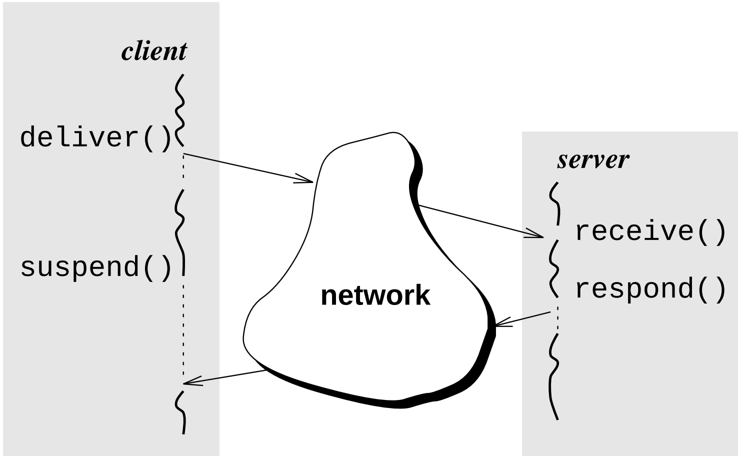

Figure 7.7: Relaxed rendezvous

next packet over to the communication system. This strategy implements *single-buffered asynchronous communication* and thus allows user threads to proceed concurrently with the physical transfer of their message packets. In the case of a *dual processor node* architecture (see also Section 3.3), the threads may even proceed in parallel to the transfer.

In principle, `deliver()` is an asynchronously operating message-passing primitive. Only back-pressure flow control performed by the network communication system may cause a temporary delay of the calling thread. A *relaxed rendezvous* then means that issuing the rendezvous request does not block the client. This relaxed communication pattern is readily supported by `deliver()` and shown in Figure 7.7. The client executes `suspend()` if it decides to block on the server (i.e., to accept the reply packet). Execution of `suspend()` entails no communication activities, but rather creates a logical client/server connection and provides a `parcel` object into which the reply message is transferred. The logical connection defines the rendezvous state.

In the relaxed rendezvous model, `request()` is split into two primitives: `deliver()`, to issue the rendezvous request, and `suspend()`, to await and accept the reply. In particular, the model assumes that a reply will be rejected when `suspend()` is not called before the reply packet transmitted by `respond()` arrives at the client site. Rejection happens because processing the reply packet received by the client site will detect an invalid client rendezvous state (i.e., a nonexisting client/server connection). Packets sent by `respond()` are not buffered at the client (receiver) site. If the receive of the reply has been rejected, the COSY transport protocol will take care of retransmitting the corresponding message packet. Independent of the capabilities of the underlying network system, COSY will be demanded by applications which cannot ensure that `suspend()` may be called before the reply to the corresponding `deliver()` has been received.[2] The key point here is that a COSY layer (offering retransmission functions) is not physically present in all cases. Sacrificing COSY will cause the loss of a reply when the `suspend()` comes too late.

It is worth noting that the relaxed rendezvous approach is applicable only in the case of a nucleus that supports remote communication. As indicated in Figure 7.7, the network must be involved in the message-passing activities. In the case of local communication, because of the lack of buffer management a relaxation of the strongly synchronous rendezvous concept is not feasible without introducing additional overhead. The relaxed rendezvous model is free only with networkwide message passing. In other words, applying `deliver()` to a local thread employs the network driver subsystem: a message buffer is taken from the free list, filled with the request packet and queued in the (local) receiver mailbox. The same pattern is executed by the network driver in reply to an interrupt indicating a receive-pending message packet.

7.1.4.2 Segment Transfer

Packet-based communication is mainly synchronous and relies on an end-to-end protocol between client and server. Only in the case of networkwide communication is a relaxed rendezvous protocol supported by utilizing the buffering capabilities of the network, the network hardware interface and the network device driver. The `deliver()` method of the `parcel` abstraction passes these capabilities directly through to the clients.

[2]See also Section 2.5.3.2 for COSY transport protocol and the user and/or network constraints imposed by it.

The `region` abstraction of the nucleus provides general asynchronous communication patterns. Methods of this class support reading and/or writing of arbitrarily sized segments from and/or to remote address spaces. For the issuing thread, reading is slightly less asynchronous than writing. This is because the read operation terminates only when the segment has been received from the source and delivered to the destination address space. In contrast, the write operation terminates when the segment has been accepted for transmission by the local communication system. Thus, returning from a write operation does not guarantee that the segment was entirely transferred into the destination address space. The `region` primitives are asynchronous with respect to thread-to-thread communication. Regarding the (local) thread-to-nucleus interaction pattern, these primitives operate synchronously.

Two variants of segment transfer are supported. These variants are shown in Figure 7.8. They basically distinguish between segment transfers carried out during a rendezvous (upper half of Figure 7.8) and those carried out within a league of threads (lower half of Figure 7.8). Both variants conform to the same transfer strategy, but differ with respect to verifying the authority for a thread to perform the transfer.

Rendezvous-Based Segment Transfer. As the name indicates, the *rendezvous-based segment transfer* allows reading/writing of memory segments only during an existing rendezvous between client and server. In order to be able to transfer segments from/into the client address space, the server must have performed a `receive()` but not yet a `respond()`. Also note that the thread issuing the segment transfer and the thread opening the rendezvous are not required to be identical. Once the rendezvous has been established, all threads of the same (server) entity implicitly possess the capability to read/write memory segments from/into the client address space and terminate the rendezvous. Thus, the pattern discussed earlier along with Figure 7.4 also applies to rendezvous-based segment transfers.

Segment reading is performed by `fetch()`, and segment writing is performed by `store()`. As indicated in Figure 7.8, fetching a remote memory segment causes the remote nucleus to transfer the requested

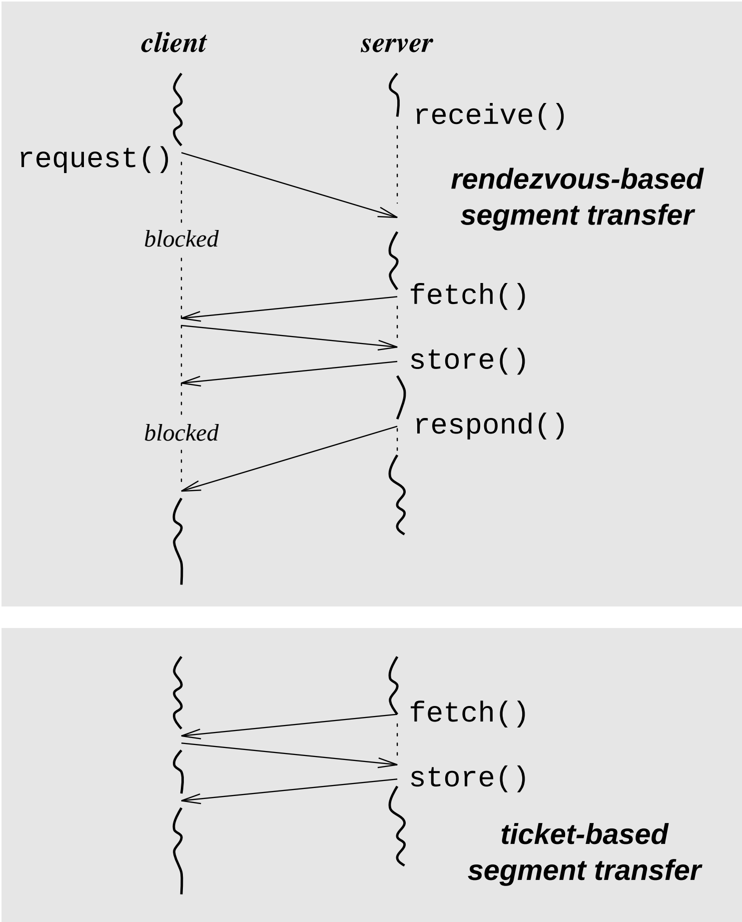

Figure 7.8: Segment transfer models

segment out of the client address space directly (across the network) into the server address space. Roughly speaking, the remote nucleus is instructed to execute a `store()` directed to the server. Both primitives (`fetch()` and `store()`) specify base address and length of the local and remote segments involved in the transfer procedure. Since the data transfer happens directly between the peer address spaces, the `region` primitives correspond to low-level memory read/write CPU instructions operating in a distributed environment.

Ticket-Based Segment Transfer. The *ticket-based segment transfer* sacrifices the "synchronization bracket" implied by the rendezvous-based approach. Instead of using an existing rendezvous as a transfer capability implicitly passed to the server (entity), a *transfer ticket* is required to perform the systemwide access of address spaces. The transfer ticket is a *unique identifier* (refer to Section 5.3.4) created by a thread and exported to grant address-space access rights to other threads (possibly executed on different nodes). Thus, being in possession of a transfer ticket allows unlimited access to the address space associated with that ticket. All threads holding the same transfer ticket build a league.

Figure 7.8 indicates that reading/writing a client address space, using the ticket-based approach, may entirely overlap the ongoing computation carried out by the client or other threads executing in the same address space. This communication pattern is especially interesting for applications of the *single-program/multiple-data* (SPMD) type. In these applications, the computation and communication bursts virtually happen at the same time. Thus, there is an implicit synchronization point, defined by the algorithm of the parallel program, at which all threads will start communication. The threads typically send to and then receive data from neighboring threads. This suggests exploiting `store()` to send the data and local memory tests to verify the successful receive of data. There is no need for the threads to explicitly receive messages from the network. The sender thread only has to know about the buffer location in the receiver address space. From the viewpoint of the algorithm implemented by the parallel program, there is no need to have the segment transfer performed by the system in a synchronized manner.

Transitive Segment Transfers. In order to fetch/store message segments, the rendezvous and the ticket approach work similar. In both cases, a *logical connection* is established. The connection is represented by a *unique identifier*. In the rendezvous case, connection establishment happens implicitly with the execution of the `request()` primitive. The (unique) identifier, submitted to `request()` and used to address the server thread, also specifies the client/server connection. In the ticket case, the connection identifier must be explicitly created. For this purpose, the `ticket` class provides the `keyword()` method, which transfers a unique identifier created by the nucleus (or the thread) to the PCB of the calling thread. It is entirely the responsibility of the ticket client to make the transfer ticket known to other threads.

The rendezvous-based segment transfer is primarily employed in simple, two-stage client/server configurations, but it fails to efficiently support multistage client/server complexes. Imagine a multistage hierarchy of active objects implementing the user task (top), a file server (middle) and a disk server (bottom). Further assume that the user task issues an I/O request to the file server, and the file server, in order to fulfill the user request, must itself issue an I/O request to the disk server. The disk server shall perform I/O directly between the user task address space and the device (i.e., disk buffer cache), bypassing the file server.

All the requests issued are of the RPC-type; more precisely, they are carried out on an ROI basis. In particular, the disk server request requires some finishing when the file server resumes operation. For example, the (stateful) file server has to advance the per-file read/write pointer after the I/O operation performed by the disk server succeeds. In such a configuration, relying on a rendezvous-based segment transfer, the disk server will be unable to access the user task address space. The reason is that when the data transfer takes place, the disk server stays in a rendezvous with the file server but not with the user task.

A solution to this problem is to let the user task request the rendezvous with the disk server. First, a file server request is issued to determine the parameters (e.g., the disk and the disk block number) needed by the disk server to carry out the I/O operation. Then, the disk server is instructed to perform I/O. Finally, a second file server request is issued to update the file state. Due to the more complicated user task/file server protocol, this solution is not only inefficient (at least three RPC/ROI requests to carry out a single I/O operation) but also critical with respect to the integrity of the file server state. The file server state depends on the correctness of the user task. Only a stateless file server overcomes this integrity problem—but may raise other problems. Another drawback of this solution is that the specific file server and disk server organization is not transparent to the client. The protocol executed by the client must be changed when the system designer (or administrator) decides to aggregate the file and disk servers into a single server complex. That is, exploiting the I/O protocol just mentioned seems to be a straightforward solution, but limits system configurability.

If a stateless file server is unfeasible, the adequate solution is to make segment transfers *transitive*. The *transitive segment transfer* requires some preparation by the user task, namely, the creation of a transfer ticket. As part of the RPC (or ROI), the transfer ticket is then passed down to the file server, which hands it over to the disk server. Basically, the transfer ticket is part of the segment address submitted to the file server. By applying the ticket-based segment transfer, the disk server is enabled to directly access the address space of the user task. Due to the RPC (or ROI) model, finishing the I/O request when the file server resumes operation becomes possible without further protocol activities—the disk server just has to complete the I/O operation and return from the RPC (i.e., ROI).

Monitored Segment Transfer. As mentioned above, segment communication overlaps ongoing computations in the address space serving as the source or destination for the transfer. More specifically, the address space is unaware of the segment transfer. However, *monitoring* segment transfers becomes possible by exploiting the thread synchronization functions described earlier in Section 7.1.3. This approach is shown in Figure 7.9. The ticket-based transfer model is used as an example; however, the technique also applies to the rendezvous-based model.

Monitoring segment transfers is accomplished by having a thread in the remote address space in charge of listening for a specific synchronization event. Every `fetch()` or `store()` reaching the remote site causes the propagation of a *transfer event* to the thread addressed by the two primitives. Thus, the (client) thread address submitted to `fetch()`/`store()` basically serves two purposes: specifying the remote address space and the thread to be checked for its readiness to accept a transfer event. If this thread has a private (i.e., non-shared) bundle, as shown in Figure 7.9, the completion of the segment transfer will cause the thread to start execution immediately. In this model, the unique bit pattern required for event propagation is represented by the (logical)

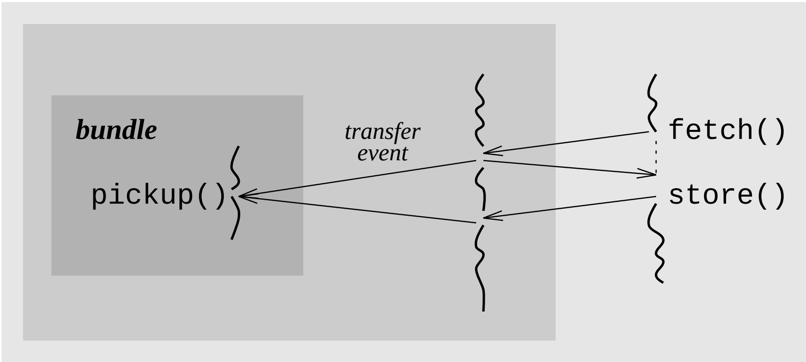

Figure 7.9: Monitored segment transfer

memory address of the segment that has been read or overwritten.

7.2 Software Backplane

Defining an *open system architecture* is one of the main goals in the design and development of operating-system families. These goals were particularly reflected in the design of the PEACE nucleus. The nucleus implements the performance-critical message-passing operations. Communication performance greatly depends on the semantics of the message-passing primitive provided. For example, depending on the given application, a nonblocking send primitive with copy semantics may be less efficient than a blocking send primitive without copy semantics. Deciding on only one category will handicap applications demanding the other category. Implementing a nonblocking send using threads and a blocking send may be disadvantageous due to the thread synchronization and scheduling overhead. Vice versa, implementing a blocking send by exploiting the nonblocking send may be disadvantageous because of the copy semantics and the implied buffering and memory management overhead.

Trying to give a "speech for the defense" of synchronous communication carries the risk of only opening an endless discussion. The same would hold for the attempt to promote asynchronous communication.

In many cases, a “religious crusade” must be fought. For the system designer, the fundamental communication principle offered to the applications should never be a matter of faith or, more drastically, a dogma. Rather, a compromise must be reached. The compromise is to design a *software backplane* on top of which any of the problem-oriented communication models may be implemented. The software-technical framework for building the backplane is the *program family concept* [66, 67].

To avoid misunderstandings, the compromise to be reached concerns the primitives actually provided by the “standard” nucleus interface. The compromise does not concern the design principle behind the nucleus. That is, the classes implementing the nucleus interface are replaceable and may be different (or will have different implementations) in the context of different applications. Providing a standard nucleus interface does not mean that the user is compelled to implement new features only by exploiting the given interface functions. Introducing minimal extensions to the nucleus, thus implementing new interface functions, may be the better approach towards a more efficient solution.

The nucleus takes the role of a software backplane for multithreaded message-passing applications. This backplane is constituted by three main building blocks (Figure 7.10), supporting thread *localization*, *communication* and *scheduling*. The building blocks are object-orientedly implemented, and thus can be customized to specific user demands. Customization is achieved by designing classes that implement application-oriented functions. The new classes may be inherited from or layered onto existing nucleus classes. By applying class inheritance, the nucleus is equipped with an object-oriented interface. When the new classes are layered onto existing nucleus classes, a more or less contemporary nucleus interface emerges. Figure 7.10 illustrates the layered approach.

The following subsections describe the nucleus interface and present exemplary implementations of selected interface functions. An implementation case study is made to demonstrate what is meant by minimal nucleus extensions adding an interface library. As will be seen, the nucleus extensions are implemented by a “handful” of C++ statements. Before the case study is presented, the class hierarchy building the “standard” nucleus and kernel interface is discussed.

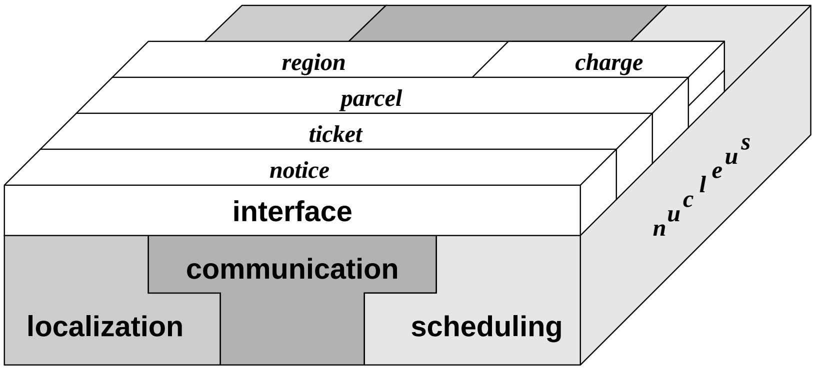

Figure 7.10: Nucleus as a software backplane

7.2.1 Class Hierarchy

The standard nucleus interface is built by four main classes. These classes are (1) `notice` to support thread synchronization, (2) `ticket` to implement communication endpoint addresses, (3) `region` to provide segment-based communication between address spaces and (4) `parcel` to provide packet-based inter-thread communication. The `charge` abstraction is a (minimal) `region` extension introducing segment-based inter-thread communication. In addition to these nucleus interface classes, the "standard" kernel interface is implemented by the `loophole` abstraction and allows the creation/destruction of communication and process gates (see also Section 6.3.4.2). Figure 7.11 gives the big, object-oriented picture of the nucleus and kernel interface.

Many of the classes shown in Figure 7.11 have been discussed earlier.[3] The focus here is on the relationship between these classes and on how these classes map higher-level user (i.e., application and POSE) abstractions onto low-level nucleus/kernel abstractions (i.e., the PCB). As mentioned at the beginning of this chapter, the sole purpose of introducing classes representing the nucleus and kernel interface is hiding implementation details from higher levels.

The PCB is the central nucleus/kernel data structure. It stores in-

[3]Refer to Sections 5.3.4, 6.3.4.2, 6.4.2, and 6.4.4.

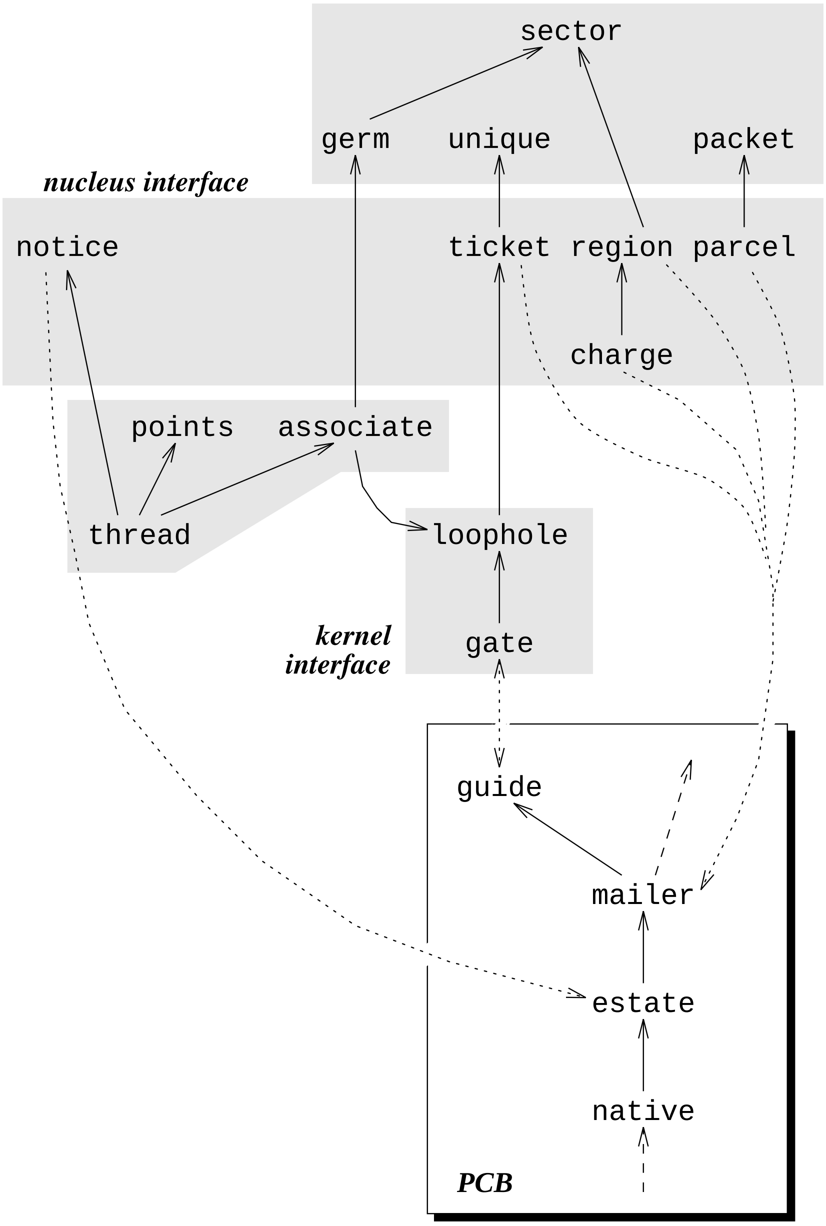

Figure 7.11: Information hiding using interface classes

formation required for scheduling, communication, synchronization and basic resource management. Every thread is implemented by a PCB, and every thread-related operation will cause the manipulation of one or more PCB objects. For example, performing a rendezvous with a local thread means storing a `parcel` object in the server mailbox, ensuring the ready-to-run state of the server, and blocking the requesting client. This effects PCB objects related to both the client and server threads. In the example mentioned, the server PCB is taken from the hash table by using the server thread identifier (i.e., communication endpoint address) as hash key. In contrast, the client PCB is determined by reading the global `life` pointer variable maintained by the nucleus.[4]

In Figure 7.11, the reference relationship between the nucleus interface classes and PCB objects is indicated by the curved, dotted arrows. As described in Section 6.4.4, the PCB structure defines a hierarchically composed complex in which classes implement a certain nucleus layer. The `mailer` abstraction (Figure 7.11) is the layer that supports the implementation of communication classes building the nucleus interface. In contrast, the `estate` abstraction is the layer that serves the implementation of thread synchronization interface classes. For example, in the case of `notice`, signal propagation requires knowledge about all the threads sharing the same entity (see also Section 7.1.3). This (resource management) information is maintained by `estate`.

The `gate` abstraction, which implements the basic kernel interface, provides the means by which threads can create, manipulate, observe and destroy PCB objects or forwarding identifiers (i.e., process gates and communication gates, respectively). This abstraction introduces dynamics: it extends the nucleus by the fundamental features required for dynamic process management. The creation of a `gate` object may entail the creation of a PCB object and thus a thread. Vice versa, the destruction of a `gate` object may cause PCB deletion and thus thread destruction.

Allocated PCB objects are kept in a hash table. The hash key and hash item necessary for PCB retrieval is implemented by the `guide`

[4]The `life` pointer is required to always be able to address the currently executing thread. In a multiprocessor environment, a `life` pointer array is maintained. This array has as many entries as there are "on-board" processors.

part of the class hierarchy used for PCB composition. Class `guide` also implements the forwarding identifier to a thread. In other words, the PCB is nothing but a specialized forwarding identifier denoting the endpoint of a forwarding chain. The `gate` abstraction of the kernel interface is responsible for adding PCB objects to the hash table and removing them upon request. This is indicated in Figure 7.11 by the dotted arrow between `gate` and `guide`.

7.2.2 Interface Classes

The rule of thumb in designing new nucleus interface classes is to add only minimal functionality. That is, the semantic gap closed by the new interface class should be very small. The ideal case is to come up with one or more classes providing mechanisms suitable for implementing the demanded functions at a higher, more application-oriented layer. A major question is which function to exclude from the design.

In the following subsections, some of the nucleus extensions are presented. The implementation of three typical communication functions will be discussed. These functions deal with packet delivery and acceptance, and the transmission of segments into (remote) address spaces. The examples shall give an impression of the complexity and "thickness" of the layer introduced to hide the internals of the nucleus from (user/system) applications. At the end of this section, an assessment of the overhead of this layer is made.

7.2.2.1 Thread Localization

The central nucleus data structure is the PCB. Thread localization is the procedure to find the PCB object that models (or implements) a particular thread. Every nucleus interface function accesses at least the PCB of the currently running (i.e., calling) thread. Depending on the function called, the PCB of a different thread must be accessed as well. For example, in the case of a local message-passing activity, the PCB of the running (i.e., sending) thread and the PCB of the receiving thread must be manipulated in order to carry out the communication function.

As mentioned in Section 7.2.1, the PCB of the currently executing thread is accessed by the global `life` pointer. The `life` pointer is updated by the dispatcher whenever a thread switch takes place. PCB objects related to threads acting as communication partners must be retrieved from the hash table. In this case, the thread identifier is used as a hash key. Figure 7.12 shows a code fragment of the retrieval of a PCB given an arbitrary thread identifier. This code is part of almost every function exported by the nucleus.

```
 1: #include "mailer.h"
 2: #include "opaque.h"
 3: #include "secure.h"
 4: #include "hashOpaque.h"
 5:
 6: extern hashOpaque pool;
 7:
 8: type class::method (...const opaque* seek...) {
 9:    register mailer* self = (mailer*)(activity*)life;
10:    ...
11:    secure nucleus;
12:    ...
13:    register guide* gate = (guide*)pool.lookup(seek);
14:    while (gate) {
15:       if (gate->bypass() <= 0) {
16:          ...
17:          return ...;
18:       }
19:       self->pause();
20:       seek = (opaque*)(unique*)gate;
21:       gate = (guide*)pool.hashUnique::lookup(gate->bypass());
22:    }
23:    ...
24:    return ...;
25: }
```

Figure 7.12: Code template performing thread localization

Thread localization starts with retrieving the communication gate bound to the `seek` parameter from the hash table (line 13). The hash table is implemented by the `pool` object. The `seek` parameter specifies a communication endpoint address and, depending on the actual nucleus configuration, refers to a weak or strong unique identifier (see

Section 5.3.4.3 for more details). Correspondingly, hash table lookup means checking for the presence of a PCB stamped with a hash key of weak or strong uniqueness. Class `opaque` implements the appropriate lookup functions and distinguishes between the two possible main nucleus configurations. In case of a hit, the retrieved gate is checked for the presence of a forwarding identifier (line 15). A forwarding identifier is a systemwide unique identifier. The `bypass()` method returns the forwarding identifier of the applied `guide` instance.

If a forwarding identifier has been detected, `guide` object retrieval continues with using the forwarding identifier as a hash key (line 21). Thread localization keeps on as long as a communication gate (i.e., `guide` object) could be retrieved from the hash table. However, before the next element of the forwarding chain is checked, the current thread relinquishes control of the processor (line 19). The `pause()` method marks a *preemption point* and requests the check for pending interrupt epilogues and attempts to allocate the processor to a different thread of the active bundle.

Thread localization terminates if a hash table miss is encountered or the retrieved `guide` object constitutes a PCB fraction. The hash table miss is indicated by an invalid (i.e., zero) `guide*` value. A `guide` object constituting a PCB is indicated by a `bypass()` return value less than or equal to zero. In the case of Figure 7.12, the code fragment starting from line 16 is executed only if the addressed thread shares the same node with the currently running thread. Refer to Section 6.4.4 for the role of class `guide` in building a PCB object.

7.2.2.2 Packet Delivery

Figure 7.7 showed the *relaxed rendezvous* protocol implemented by the `parcel` abstraction or, more specifically, by exploiting `deliver()` and `suspend()` instead of `request()`. However, whether or not a rendezvous is really performed depends on the client/server protocol. The `deliver()` primitive is a general nonblocking message-passing operation, implementing single-buffered asynchronous communication. If agreed on by client and server, `deliver()` need not necessarily be followed by a `suspend()`. That is, `deliver()` can be exploited to only set up a one-way communication activity, allowing the issuing thread to "immediately" continue with its computation. The implementation of `deliver()` is shown in Figure 7.13.

The C++ code listed in Figure 7.13 is typical for almost every message-passing primitive implemented by the nucleus. Three main code parts can be distinguished: thread localization, local communication, and remote communication. Lines 16–18 and lines 27–29 show the thread localization code and have been explained in the previous section. The code for local communication is listed in lines 19–25 and for remote communication in lines 31–36.

In the relaxed rendezvous protocol, delivering a `parcel` object to a local thread (lines 19–25) employs the network driver subsystem. Via `pilfer()` (line 21), a free message buffer is claimed and filled (line 22) with data from the `parcel` instance applied to `deliver()`. Afterwards, the message buffer is stored in the receiver mailbox (line 24), and the receiver thread is set ready to run (line 24). The weak unique part of the receiver thread identifier is passed back as return code to the calling thread.

Remote packet delivery starts with checking the actual processing state of the message buffer of the currently running thread (line 31). The thread temporarily suspends further execution while a previously issued remote delivery request is still active. A *busy waiting* scheme is performed, based on the `pause()` primitive used to mark a preemption point (see the previous subsection). When the inactive state of the message buffer has been indicated by the network driver subsystem, the `niceRequest` PDU is set up (lines 32–34). Finally, as shown in line 35, the PDU is handed over to the network layer and transmitted to the node specified by the actual parameter of method `issue()`.

7.2.2.3 Packet Acceptance

The acceptance of a message packet (i.e., a `parcel` object) may be blocking or nonblocking for the receiving thread. When `receive()` is applied to a non-empty mailbox, the calling thread continues execution with the next available message packet. In addition to taking over the packet, the (weak/strong) unique identifier as well as the communication endpoint address of the sender thread are returned to the receiver. This packet receive function is implemented by the code example illus-

```
#include "wagon.h"
#include "assert.h"
#include "mailer.h"
#include "parcel.h"
#include "secure.h"
#include "vessel.h"
#include "hashOpaque.h"

extern hashOpaque pool;

int parcel::deliver (register const opaque* seek) {
   register mailer* self = (mailer*)(activity*)life;
   secure nucleus;
   register niceRespond* data;
   data = (niceRespond*)self->depot(); assert(data);
   register guide *gate = (guide*)pool.lookup(seek);
   while (gate) {
      if (gate->bypass() <= 0) {
         register niceRespond* free;
         if (!(free=(niceRespond*)data->pilfer()))
            return -1;
         self->space()->fetch(*this, *free);
         register mailer* peer= (mailer*)gate;
         peer->store(free); peer->knock();
         return *(lui*)gate;
      }
      self->pause();
      seek = (opaque*)(unique*)gate;
      gate = (guide*)pool.hashUnique::lookup(gate->bypass());
   }
   while (data->active()) self->pause();
   data->sequel((volume*)(packet*)this,sizeof(packet));
   *(opaque*)data = (opaque*)seek;
   ((niceRequest*)data)->setup(*self);
   ((niceRequest*)data)->issue(*(lui*)data);
   return *(lui*)data;
}
```

Figure 7.13: Single-buffer asynchronous communication

trated in Figure 7.14.

```
 1: #include "wagon.h"
 2: #include "mailer.h"
 3: #include "parcel.h"
 4: #include "secure.h"
 5: #include "vessel.h"
 6:
 7: const opaque* parcel::receive (unique& from) {
 8:    register mailer* self = (mailer*)(activity*)life;
 9:    secure nucleus;
10:    register wagon* data = (wagon*)self->store();
11:    while (!data) {
12:       self->await();
13:       if (self->error() < 0) return 0;
14:       data = (wagon*)self->store();
15:    }
16:    self->strip();
17:    self->space()->store(*data, *this);
18:    if (!data->remote()) {
19:       self->space()->store(data->alien(), from);
20:    } else {
21:       if ((int)data->alien() >= 0) {
22:          self->space()->store(data->alien(), from);
23:          data->donate();
24:       } else {
25:          data->alien(-(int)data->alien());
26:          self->space()->store(data->alien(), from);
27:          *(opaque*)data = data->alien();
28:          ((niceLiaise*)data)->setup(*self);
29:          ((niceLiaise*)data)->issue(*(lui*)data);
30:       }
31:    }
32:    return *(opaque*)&from;
33: }
```

Figure 7.14: Receive of a message packet

As shown in line 10, the receiver thread first attempts retrieving a message packet from the mailbox. If the mailbox is empty, the receiver is suspended (line 12) until a message has been sent by some other thread. Thread reactivation may be caused by an abort signal or the receive of a message. In the first case, `receive()` terminates with an error condition (line 13). The error condition is indicated to the receiver thread by an invalid communication endpoint address delivered as return value. In the second case, the thread selects the next available message packet from the mailbox (line 14) and attempts to leave the receive loop.

Having successfully passed the receive loop, the thread removes the message packet from the mailbox (line 16) and copies the contents into the `parcel` object applied to `receive()` (line 17). Afterwards, a check for the origin of the message is made. If a local transaction has been performed, the sender identifier must merely be copied into the buffer supplied by the caller (line 19).

Note that an `opaque` identifier is copied into a buffer large enough to hold a `unique` identifier. Depending on the actual nucleus configuration, either a weak or strong unique identifier value is copied. As described in Section 5.3.4.3, a weak unique identifier is an `lui` instance and is the size of a plain `int` object. In contrast, a strong unique identifier is a `unique` instance and may have a fairly large size, depending on the size of the `sparse` component added to the `lui` part. Class `opaque` provides the necessary type transfer operations.

In case of a remote transaction, the consumed message buffer must be given back to the network driver subsystem. Two cases must be distinguished. The first, simple, case is that the message was received directly (i.e., without intermediate forwarding). In this case, the sender identifier is passed to the receiver, and the message buffer is put back on the free list maintained by the network driver (lines 22 and 23, respectively). The second, more complicated, case is the receive of a forwarded message. This case requires notifying the sender site about the actual receiver of the message. A side effect of message forwarding is that the identifier of the ultimate receiver thread will be different from the identifier submitted to the `request()` primitive and stored in the client PCB to specify the rendezvous. The client rendezvous state must be updated to let a later `respond()`, `forward()` or `fetch()/store()` succeed. This update request is set up and issued in lines 27–29. The buffer is put back on the free list if the network driver finished the update-request transfer.

7.2.2.4 Segment Transmission

The third code example deals with the *ticked-based transfer* of arbitrarily sized memory segments. Along general lines, although the semantics of this communication function is different from the function of `deliver()` described before, the actual code illustrated in Figure 7.15 is largely identical to the code of Figure 7.13.

The transfer capability is represented by the `key` parameter, which is a `unique` object. In the local case, `store()` differs from `deliver()` with respect to lines 19–21. The transfer ticket is verified (line 19), and `store()` fails if the current thread does not belong to the league of the addressed thread (i.e., does not hold the right transfer capability). Line 20 performs the local memory-to-memory transfer, and line 21 propagates a transfer signal to the addressed thread.

In the remote case, the main difference from `deliver()` is the PDU setup. In the `store()` case, PDU setup is concerned with requesting from the network layer a segment transfer directly out of the sender address space. Line 31 defines the sequel tank of the `holder` abstraction (see Section 4.3.2.2 for more detail) with the descriptor of the segment to be transferred. Roughly speaking, this statement sets up a DMA descriptor. The statement in line 32 encodes the sender identifier and the transfer ticket. Finally, line 33 serves to encode the destination segment of the receiver address space. This address space is specified by the (`opaque`) receiver thread identifier set up in line 30. Segment transfer is requested by the `issue()` method of the `niceStore` PDU (line 34). As with `deliver()`, before the setup code is executed, `store()` makes sure that a previously issued communication request has been finished (line 29).

7.2.2.5 Assessment

The code examples discussed in the previous subsections were developed with the goal of achieving the utmost efficient implementation. As the following discussion reveals, both the message startup time and message latency are not dominated by the implementation of the nucleus interface functions, but by the network driver subsystem.

Viewing it top-down, the network layer of the nucleus is repre-

```
#include "wagon.h"
#include "assert.h"
#include "region.h"
#include "mailer.h"
#include "secure.h"
#include "vessel.h"
#include "hashOpaque.h"

extern hashOpaque pool;

int region::store(const opaque* seek,
                  const unique& key, volume* sink) const {
   register mailer* self = (mailer*)(activity*)life;
   secure nucleus;
   register guide* gate = (guide*)pool.lookup(seek);
   while (gate) {
      if (gate->bypass() <= 0) {
         register mailer* peer = (mailer*)gate;
         if (peer->track() != key) return -1;
         self->space()->carry(base, size, sink, peer->space());
         peer->knock(sink, 1);
         return *(lui*)peer;
      }
      self->pause(); seek = (opaque*)(unique*)gate;
      gate = (guide*)pool.hashUnique::lookup(gate->bypass());
   }
   niceStore* item = (niceStore*)self->depot();
   assert(item);
   while (item->active()) self->pause();
   *(opaque*)item = (opaque*)seek;
   item->sequel(*this);
   item->setup(*self,key);
   item->setup(sector(sink,size));
   item->issue(*(lui*)item);
   return *(lui*)item;
}
```

Figure 7.15: Ticked-based segment transmission

sented by NICE (see also Section 2.5.3). The `issue()` primitive of class `niceRequest` (Figure 7.13) and `niceStore` (Figure 7.15) instructs

COSY—the next lower (transport protocol) layer—to transfer a message to the node whose logical address is taken from the `site` field of the weak unique identifier submitted as parameter. The message length corresponds to the actual size of the PDU. As mentioned in Section 6.5.3, a class hierarchy is used to compose the various PDU types of the NICE layer. Every class in this hierarchy provides the `issue()` method. This measure ensures that every `issue()` method knows exactly the size of its PDU. Thus, it is able to transfer only the minimally required amount of data across the network.

The decision for the inline (thus redundant) implementation of the thread localization code results from the hard performance requirements that must be fulfilled by the communication primitives of the nucleus. Separating the code by an out-of-line implementation would introduce a performance handicap. The resulting external function would then exhibit two output parameters (corresponding to `gate` and `seek`). More specifically, the performance handicap is not the output parameter, in general, but the use pattern of a function exhibiting output parameters. For the actual `seek` parameter, the majority of the nucleus interface functions assume the storage class "register" at the user and nucleus levels. Thus, in a single-address-space configuration, an optimizing compiler receives a hint as to which parameter should be passed by register and which can be passed via the stack. The register storage class can no longer be maintained when `peer` becomes a reference (i.e., pointer) parameter of the new function. In this case, the compiler would generate a temporary (`const opaque*`) object, pass the object as an actual parameter and assign the object to the allocated processor register when the function returns.

Three objects build the cornerstone for implementing nucleus interface functions: the hash table, the PCB and the message buffer. As shown in the code examples, these objects are accessible only via methods. More generally speaking, instance variables of all (nucleus/kernel) classes are never made public; only methods may be made public. The majority of the methods used implement a *fast path* to access the object state in a controlled manner. A fast path means the exploitation of (C++) inline methods. In many cases, the methods are split into two parts: a small inline fraction, and a possibly large and more sophisticated out-of-line fraction (i.e., externally implemented method). The inline part corresponds to the fast path implementation and supports the most frequently occurring use patterns.

All in all, this implementation strategy guarantees a very efficient message startup without sacrificing the ADT principle—or without being dependent on a compiler that performs code optimization by automatically inlining functions consisting of a handful of statements only. For example, in the case of `deliver()` and `store()`, the overhead for reaching the NICE/COSY interface (by calling `issue()`) and returning to the caller amounts to about 4 μs on a 50-MHz i860XP. The entire `deliver()/store()` delay is 23 μs on a MANNA node (see also Section 3.3) when exploiting a single CPU of the above-mentioned sort. Here, the network subsystem introduces a delay of about 20 μs. This delay can be hidden from the application layer by using a second CPU in charge of executing network communication. Instructing the second CPU to carry out the message transfer, the delay is 4 μs. In the two-CPU node configuration, the NICE/COSY interface is the software equivalent of the AP/CP hardware interface defined by the MANNA node architecture.

As pointed out in [36], "*it is the system design which is hierarchical, not its implementation.*" The fast-path implementation of the methods of the three main nucleus abstractions (PCB, message buffer and hash table) is a typical example of this statement. When taking a look at the compiler-generated code sequences, all nucleus primitives exported by the interface classes exhibit a flat structure. Assume the normal case of executing a remote message-passing operation such as `deliver()` or `store()`, namely, that a hash table miss is encountered when the input parameter specifying the receiver thread is used as a hash key (see line 13 of Figure 7.12). In this situation, fast-path implementation also means that reaching the NICE/COSY (i.e., AP/CP) interface happens without any intermediate procedure call. The C++ code, however, is strewn with method calls. Using the terminology of structured computer organization [86], the code appearing at the *problem-oriented programming-language level* (e.g., implemented by a C++ compiler) is structured and virtually slow, while the equivalent code appearing at the *conventional machine level* (e.g., implemented by an i860XP) is entirely unstructured but fast.

7.3 Message Streams

Message passing is not always the appropriate communication paradigm, especially for applications exhibiting (fine-grain) massive parallelism. Executing a parallel program by several thousands of threads and relying on explicit communication and synchronization measures may result in highly error-prone implementations. These sorts of parallel applications demand new programming paradigms. As opposed to *function parallelism*, the generally accepted opinion is that *data parallelism* will be the only way of achieving massive parallelism. It is extremely difficult to design a program that exhibits only a few tens of different, parallel executable functions which, in addition, jointly perform a meaningful task. This difficulty does not arise when focusing on the data that may be processable in parallel.

Data parallelism means that a (possibly very) large number of threads execute a *single program* distributed over a (possibly large) number of nodes. Each of these threads executes largely autonomously and will work on private data objects, thus resulting in the parallel processing of *multiple data*. This model is known as the SPMD principle. Although the threads are largely autonomous as they execute the same program, a more or less synchronous processing pattern can often be observed. That is, all the threads will execute their computation and communication bursts virtually at the same time.

The granularity of the data processed by and communicated between the threads is strongly application-dependent. Typically, (numerical) parallel applications work on extremely large arrays consisting of thousands or even millions of floating point elements. The art of programming these types of applications relies on parallel algorithms that define a very efficient partitioning of the monolithic array structure. Likewise, they result in a very efficient distribution of the array fragments over a distributed-memory machine. In many cases, the communication topology defined by these algorithms is of regular, but not necessarily static structure.

Especially in the case of topologies that manifest a nearest-neighbor configuration, a *virtual channel* concept provides the appropriate means of communication. The channels are established before a certain processing step begins and are destroyed after this step has been finished. A natural means of communication is to set up *message streams* which are generated and processed on the fly by the sender and receiver threads, respectively.

7.3.1 The Ideal Model

Virtual channels are not only useful in specifying the communication topology of an SPMD program; by explicitly establishing channels between the communicating threads, the operating system is enabled to schedule communication resources in advance. Thus, the operating system receives important hints from the application as to how many communication resources should be available at a certain point in time. In order to achieve good user acceptance, the main problem to be solved is guaranteeing a "*featherweight*" channel establishment and elimination procedure.

Figure 7.16 shows the virtual channel model dealt with in the following subsections. The model is implemented as a *user-level class library*, introducing a minimal extension to the nucleus interface classes discussed in the previous section. In addition, the model supports fine-grain and typed communication between the threads and maintains *network transparency*. Here, network transparency does not only mean hiding the specific multinode architecture from the application, but also making applications independent from the particular properties of the underlying network data interface. That is, the user is presented the illusion of an ideal network interface capable of receiving and delivering the basic data items typically processed by an application program. Instead of defining a message-passing interface, the virtual channel will support reading/writing of typed objects (i.e., simple-typed variables as defined by C++) and will be capable of transferring message streams.

The actual software implementation of the model shown in Figure 7.16 and described in the following relies on *data aggregation* (i.e., intermediate buffering) to compose a message on the fly. Similarly, at the receiving end of the channel, the data is assigned to program variables by reading from the buffer. Thus, the implementation is suited to close the semantic gap between an application requiring fine-grain, typed communication and a hardware platform supporting only coarse-grain, buffered, untyped communication.

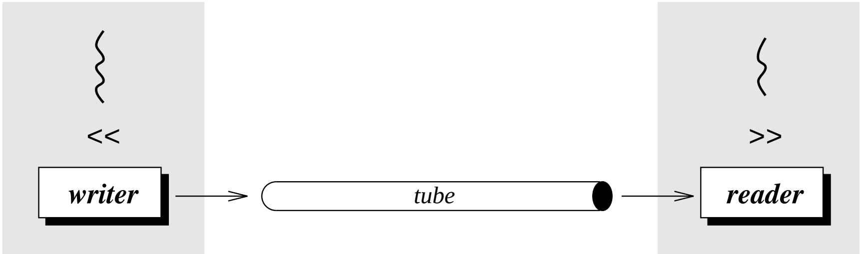

Figure 7.16: Virtual channel communication

Data gathering and scattering will be done by exploiting the `parcel` abstraction to send and receive the contents of the buffer. In particular, communication will be carried out by using the `deliver()` and `receive()` methods. Buffer transmission is nonblocking, while buffer acceptance may be blocking for the issuing thread. The buffer is managed by the `tube` object connecting the two communicating sites (refer to Figure 7.16). Communication is *one-way*, from a `writer` to a `reader` object. The external interface of these two objects is made of methods ensuring typed data transmission. In the case of C++, these methods are the *binary operators* `<<` and `>>`, respectively. Thus, the classes from which the `reader` and `writer` objects have been created conform to the standard C++ I/O stream concept.

In order to achieve an efficient implementation, the buffering of data elements must pay attention to alignment restrictions possibly present in the underlying CPU. For example, the memory access model of an i860 or SPARC processor assumes that N-bit data will be stored N-bit-aligned. A very common feature of RISC processors is to signal a trap when the thread performs a memory access with a misaligned data address. This misalignment trap must then be handled by the kernel, by simulating an ideal (abstract) processor capable of accessing aligned as well as misaligned data elements from memory without raising an exception. Needless to say, in this situation performance is bad. For this reason, `tube` provides methods by means of which aligned data access can be enforced. These methods are used by the `reader/writer` operators. Since every C++ (plain) type is handled by a dedicated operator, alignment enforcement becomes straightforward.[5]

7.3.2 Typed Data Streams

As shown in Figure 7.16, the `writer` instance defines the beginning, and the `reader` instance the end, of a virtual channel. In order to establish a *writer channel*, the receiving site of the data must be specified. For this purpose, the `writer` class contains a constructor to identify the receiving end of and qualify the connection. The first aspect means introducing a *connection identifier*, that is, associating the virtual channel with the systemwide unique identifier of the thread acting as receiver. In contrast, the second aspect introduces a *virtual channel number* relative to a particular connection. That is, channel 4711 may have different meanings when considering a different connection. The `writer` class specification is shown in Figure 7.17.

The writer channel constructor is listed in line 5. Lines 7–18 list the shift operators used for typed data transmission over the virtual channel. Every shift-out advances the per-buffer write pointer by the number of bytes specified by the size of the formal parameter type of each operator. If the remaining space is not large enough to store the next data item, the `tube` buffer is sent out by calling `deliver()`, and the write pointer is set to the first location available for data aggregation. In addition to the user objects, the buffer also stores the `tube` channel number and the actual size (in bytes) of the aggregated data stream. These two variables constitute the PDU used to control the data transfer via a virtual channel. The SDU available for transmitting user objects is adjacent to the PDU. Thus, the initial write pointer value is the base address of the SDU in the buffer.

The shift operator capable of processing an entire `writer` object (line 20) causes the flushing of a "dirty" `tube` buffer. If the buffer has been written into since the last `deliver()`, a transfer to the thread at the other end of the channel will be initiated. Line 21 shows the

[5] Usually, objects of different types are of different size, and thus will have differently aligned memory addresses. For example, using 32-bit processor technology, a `short` object is a 16-bit quantity and requires 16-bit alignment. A `long` object is a 32-bit quantity and requires 32-bit alignment. In contrast, a `char` instance has a size of 8 bits and requires no alignment at all.

```
#include "canal.h"

class writer : private canal {
public:
   writer (const unique&, unsigned = 0);

   writer& operator<< (char);
   writer& operator<< (unsigned char);
   writer& operator<< (short);
   writer& operator<< (unsigned short);
   writer& operator<< (int);
   writer& operator<< (unsigned int);
   writer& operator<< (long);
   writer& operator<< (unsigned long);
   writer& operator<< (const char*);
   writer& operator<< (void*);
   writer& operator<< (float);
   writer& operator<< (double);

   writer& operator<< (const writer&);
   writer& operator<< (const morsel&);
};
```

Figure 7.17: Writer stream abstraction

operator that copies a complete SDU into the `tube` buffer. This operator neither manipulates the write pointer nor causes the invocation of `deliver()` to transfer the buffer contents. The `morsel` class implements the abstraction from the `tube` SDU.

At the other end of the `tube`, a `reader` object serves the purpose of clearing the associated channel. The external interface of this object is basically the same as that of a `writer` object. Typed data transfer (i.e., reception) is made feasible by overloaded C++ shift operators of class `reader`, as shown in Figure 7.18.

Instead of providing only one constructor as in the `writer` case, class `reader` specifies four constructors (lines 5–8). Each of these constructors stands for a different shift-in strategy. Constructor `reader()` establishes a *reader channel* accepting data sent by any thread and via any writer channel. A reader channel that accepts data sent by

```
#include "canal.h"

class reader : private canal {
public:
   reader ();
   reader (unsigned);
   reader (const unique&);
   reader (const unique&, unsigned);

   reader& operator>> (char&);
   reader& operator>> (unsigned char&);
   reader& operator>> (short&);
   reader& operator>> (unsigned short&);
   reader& operator>> (int&);
   reader& operator>> (unsigned int&);
   reader& operator>> (long&);
   reader& operator>> (unsigned long&);
   reader& operator>> (char*);
   reader& operator>> (void*&);
   reader& operator>> (float&);
   reader& operator>> (double&);

   reader& operator>> (const reader&);
   reader& operator>> (morsel*);
};
```

Figure 7.18: Reader stream abstraction

any thread, but via a specific writer channel, is established by executing `reader(unsigned)`. The formal parameter of this constructor specifies a (writer) virtual channel number. Constructor `reader(const unique&)` establishes a reader channel that accepts data sent only by the specified thread; however, the data may have been sent via any writer channel. This constructor expects the sender thread identifier as parameter. Finally, constructor `reader(const unique&, unsigned)` is used to establish a reader channel associated with a specific sender thread and a specific writer channel.

Upon `reader` object construction, the chosen receive strategy is indicated to the `tube` object. The receive function of a `tube` manages a cache of accepted but not yet consumed message packets, each one held in a buffer. The *receiver buffer cache* is implemented by a *hash table* and stores messages containing a `tube` PDU and SDU, as set up by the data injecting site of a virtual channel. Filling up the cache is accomplished by applying `receive()` to a `parcel` object. Clearing the cache happens through the various shift operators of the `reader` abstraction.

When a `reader` shift-in operation is attempted on an empty `tube` buffer, the `tube` receive function is activated. This function performs a hash table lookup to check for pending messages according to the receive strategy associated with the "calling" `reader` instance. The `tube` will invoke `receive()` if a properly qualified message is not stored in the hash table. Similarly, once returned from `receive()`, the accepted message may fail to exhibit the expected reader channel qualification (i.e., sender thread identifier and/or channel number). In this case, the received message is made pending and will be kept in the buffer cache. Since the buffer cache is implemented as a hash table, the virtual channel number contained in the accepted message serves as the hash key. That is, attempting to receive a message via a `reader` object bound to reader channel 4711 may cause the buffer cache to be filled up with messages not containing the expected channel number, 4711.

The low-level `tube` receive function may cause any thread to accept data possibly related to other threads of the receiving site (i.e., the same entity). Most significantly, `tube` assists the nucleus in preventing a run-out of network buffers and the discarding of incoming receive requests. Messages received from the network are temporarily buffered by the nucleus, by entering the per-message buffer descriptors into the mailbox of the destination thread. From the nucleus viewpoint, a network buffer is returned to the network driver only after the receiver thread has accepted the delivery of the contained message by a call on `receive()`. The `receive()` primitive transfers the buffer contents into the applied `parcel` object and frees the buffer descriptor (refer also to the `receive()` implementation depicted in Figure 7.14). Consequently, the more often a thread calls `receive()`, the more unlikely a run-out of network buffers. Note that the network buffer pool is global to the nucleus and cannot be maintained on a per-thread basis, for example.[6]

---

[6]In order to allocate a free buffer, PCB location must have been finished if

As shown in Figures 7.17 and 7.18, classes `reader` and `writer` are an extension to the `canal` abstraction. Class `canal` implements the common access to the `tube` object. The complete class hierarchy designed to implement virtual channels is shown in Figure 7.19.

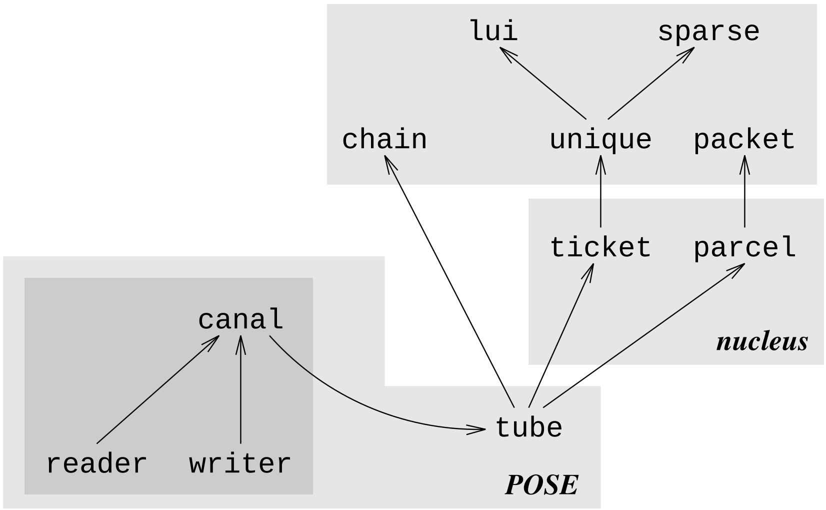

Figure 7.19: A class hierarchy implementing virtual channels

Class `chain` makes `tube` objects maintainable in free lists and hash lists. The connection identifier currently associated with a `tube` is implemented by the `ticket` branch. By means of `ticket`, the `unique` thread identifier used to specify the connection endpoint will be translated to a communication endpoint address needed by `deliver()` to locate the receiver thread. Finally, the `tube` buffering and message-passing capability is inherited from the `parcel` abstraction. Class `parcel` not only manifests the size of the message `packet`, but also

---

the network buffer pool is maintained on a per-thread basis. A PCB is addressed remotely via a search key set up by the sending, and interpreted by the receiving, site. The search key is the systemwide unique thread identifier and is transmitted as part of the PDU carrying the user message (i.e., SDU). Search key interpretation is possible only after the enclosing message has been transferred to memory. Thus, before a per-thread network buffer pool could be utilized, a global buffer is required to temporarily store at least the network PDU.

introduces the methods `deliver()` and `receive()` used to transfer data through the `tube`.

7.4 Distributed Processes

Section 7.1.1 described the fundamental process model implemented by the nucleus and kernel. Two main concepts were distinguished, a *heavyweight process* and a *lightweight process*. The former is implemented by the `entity` abstraction and results from establishing a new, initially single-threaded address space on some node. Additional threads are to be created on demand by exploiting the `thread` abstraction. Creating a new `thread` instance introduces a new lightweight process sharing the same address space with other threads. A `thread` instance implements an *active object*.

The initial thread of a newly created address space is represented by the `primer` instance (see also Section 6.4.2). A `primer` is an active object with `argc`, `argv` and `envp` as instance variables. That is, the `primer` is the object-oriented variant of the standard C++ `main()` procedure. The autonomous flow of control of an active object is implemented by a customizable `action()` method. This method is automatically called by the system when a new active object has been created.

Creating a lightweight process does not necessarily mean adding a new `thread` instance to the address space of the creator. The notion of a "lightweight process" stands only for a number of threads executing within the same address space, thus making context switching between these threads lightweight. Usually, new threads are created locally, that is, by some other thread of the same address space. By exploiting *dual objects* (see Section 5.3.2), threads may also be created remotely. However, in the local as well as the remote case, the outcome will be the same: a thread will be added to an existing address space. Figure 7.20 shows the example of an `actor` active object implemented as a dual object.

The `actor` is specified by a *dual class*, which is indicated by the `dual` annotation in line 14. This annotation instructs the dual object generator to create not only `actor` *likeness* and *prototype* classes but also a class specifying an `actor` *clerk*. As discussed in Section 5.3.2, the

```
#include "reader.h"
#include "writer.h"
#include "thread.h"

class actor : private thread {
   reader  in;
   writer  out;
   ...
   void action ();
public:
    actor (const unique& peer) : out(peer) {...}
   ~actor ()                             {...}

   operator unique () const    { return *this; }
   ...
}/*!dual!*/;
```

Figure 7.20: A thread implemented as a dual object

likeness class implements the client view of an `actor` instance, while the prototype class implements the server view. The generated clerk class serves the purpose of creating one or more `actor` servers. Each of the `actor` clerks accepts and executes ROI requests to create, access and destroy `actor` prototypes. Roughly speaking, an `actor` prototype is specified by the original, non-annotated C++ class definition and is thus a "normal" C++ object. In order to deal with a dual `actor` object, the client receives a likeness instance from the clerk as the result of prototype installation. The likeness is a networkwide `actor` prototype capability and provides the network-transparent access to a particular `actor` prototype.

An `actor` "is a" `thread`. Thus, `actor` creation entails the creation of an active object. Exactly the same happens when the `actor` clerk receives a request to create an `actor` prototype: the `actor` clerk creates a new active object on behalf of the client. This semantic-preserving property of the dual object model enables the programmer to design and implement a multithreaded distributed program in a straightforward way. The resulting program defines a *distributed process*. Figure 7.21 gives a sketchy outline of the main part of such a (fictitious) process.

```
 1: #include "actor.h"
 2: #include "primer.h"
 4:
 5: void primer::action () {
 6:    actor   actN(*this, argv[1]);
 7:    actor   actE(*this, argv[2]);
 8:    actor   actS(*this, argv[3]);
 9:    actor   actW(*this, argv[4]);
10:
11:    reader  fanin;
12:
13:    writer  north(actN);
14:    writer  east(actE);
15:    writer  south(actS);
16:    writer  west(actW);
17:
18:    unsigned profile, data[4];
19:
20:    for (;;) {
21:       fanin >> data[0] >> data[1] >> data[2] >> data[3];
22:
23:       profile = (data[0] + data[1] + data[2] + data[3]) / 4;
24:
25:       north << profile;
26:       east  << profile;
27:       south << profile;
28:       west  << profile;
29:    }
30: }
```

Figure 7.21: The primer thread

The `primer::action()` implementation shown in Figure 7.21 creates four `actor` threads remotely (lines 6–9). Each of the `actor` constructors is furnished with two parameters. The first parameter is demanded by the `actor` constructor specification depicted in Figure 7.20. It becomes the unique identifier of the `primer` thread and is used by the `actor` to establish a writer channel to the `primer`. The second parameter of the `actor` constructor specifies the location of the clerk selected for prototype installation. Note that the `actor` constructor calls listed in Figure 7.21 correspond to the likeness class. Being fed with the dual `actor` class (Figure 7.20), the class generator creates likeness constructors whose last parameter is always a clerk address. This parameter has a default assignment, and thus can be omitted when creating a dual object. In order to support a problem-oriented distribution, the default assignment may be overwritten by some other, configuration-dependent address. In the example shown, the clerk addresses are taken from the `argv` array, which has been set up with user-defined values at `primer` creation time.

The rest of the `primer::action()` implementation works as follows. To each of the remotely created `actor` threads, a writer channel is established. Note that the receiver thread is addressed by the unique identifier bound to an `actor` object. This identifier is delivered by the implicit execution of the type transfer operator specified by class `actor` (see line 14 of Figure 7.20) when an `actor` object is used where a `unique` object is expected. A single reader channel serves for receiving data from any of the four remote threads. The task of the `primer` thread (lines 21–28) is to read a load index from the threads, build the cross section and write the resulting profile to the threads. A code example for the corresponding `actor` thread is shown in Figure 7.22.

```
 1: #include "actor.h"
 2:
 3: void actor::action () {
 4:    unsigned profile, load = 0;
 5:    for (;;) {
 6:       out << load;
 7:       ...
 8:       in >> profile;
10:       ...
11:       load = profile + runtime;
12:    }
13: }
```

Figure 7.22: The actor thread

As indicated in the figure, `actor::action()` sends its current load index to the `primer` thread (line 6) and reads the actual cross section of the four load indices computed by the `primer` (line 8). The new load index is the sum of the received profile value and the cumulative runtime of the `actor` thread.

The code examples of Figures 7.21 and 7.22 do not stand for a sophisticated algorithm implementing distributed load index computation. Rather, both examples serve the purpose of underpinning the advantages of object-oriented software construction. The abstractions discussed in this and the foregoing sections give an almost complete overview of the implementation of a distributed process. Messages streams, which were used for communication between the spatially arranged `actor` objects and their `primer` instance, are implemented by:

1. The `reader` and `writer` classes (Figures 7.18 and 7.17)

2. Which rely on virtual channels (Figure 7.19)

3. Which are implemented by the `parcel` interface class, particularly the methods `deliver()` and `receive()` (Figures 7.13 and 7.14, respectively)

All of these examples confirm the statement made earlier; namely, that interface customization of the nucleus can be achieved by only a "handful" of lines of C++ code. The guiding rule was implementing basic mechanisms by introducing merely minimal nucleus extensions. Similar considerations hold for the implementation of the distributed process. The outstanding aspect here was the exploitation of the dual object model (see also Section 5.3.2) to create a distributed, object-oriented program. In the case of the `primer` example, the dual object runtime system depends only on the following abstractions:

1. Class `thread`: to model clerks

2. Class `ticket`: to determine communication endpoint addresses

3. Class `parcel`: to execute the ROI request-response protocol

For the system designer, a very important aspect—if not the most important—is to be able to rely on proper programming-language support helping to cope with distribution. This aspect becomes significant especially in the light of an object-oriented and dynamically alterable operating system. Without adequate language support, the designer would be lost trying to get around obstacles that may have very little in common with his true problems. The risk is extremely high that, in the resulting implementation, different levels of abstraction get unfavorably interwoven with each other. An example of adequate language-level support for object-oriented (system) programming has been demonstrated by exploiting dual classes (refer to Section 5.3.2) to model homogeneous but physically fragmented objects. The expressiveness of dual objects (created from dual classes) and of the classes provided by the nucleus and kernel allows the system designer to take a fairly abstract view of the underlying architecture. At the same time, performance need not be sacrificed.

7.5 Summary

Inheritance is a powerful instrument to construct an application-oriented operating system and customize lower-level system abstractions to specific user demands. Since user applications may influence the organization and function of the underlying object-oriented software platform, an object-oriented operating system seems to provide the best support. However, the other side of the coin is that in a strongly object-oriented driven approach, higher-level user/system components often become interlocked with lower-level system components. That is, a user application may be compelled to see a lot of system internals. Changes made in the class hierarchy deep inside the system may require a recompilation of user programs.

The openness of an object-oriented operating system may negatively affect user acceptance. This aspect is a fact, although object orientation helps to fend off any prejudice brought forward by the parallel programming community when broaching the operating system. A compromise must be made. Applications that are not afraid of, and may benefit from, an interlocked user/system representation are faced with the object-oriented view of the underlying system. In contrast, applications that may have trouble in taking the object-oriented view are faced with a contemporary user/system interface. The only admission assumed is that even in the latter case classes implement the interface of the operating system. On the one hand, the operating system interface will be *class-based*; on the other hand, it will be *object-oriented*.

These two views underlie the two main branches of a family of parallel operating systems. More specifically, the class-based view is introduced as an add-on to the object-oriented platform. This add-on is technically represented by a "thin" library package. The platform is constituted by the nucleus and the kernel. The nucleus and kernel define an *abstract machine* on top of which more elaborate functions can be layered to introduce application-oriented and yet highly efficient abstractions. The nucleus is responsible for systemwide communication and thread processing, and the kernel introduces dynamic behavior into the system. In conjunction with an object model suited for distributed (system) programming and a message stream concept implemented as a runtime library package, a *process execution and communication environment* (PEACE) for developing parallel user and system programs emerges.

PEACE supports the construction of distributed programs, without depending on a central process management subsystem. In PEACE terminology, a process management subsystem is nothing but a distributed system program. The presented `actor` subsystem is a typical example of this position and is transferable to every other system service. At an abstract level, operating-system services are implemented by passive objects and will technically appear as *dual objects* if distribution comes into play. Dual objects, however, are managed by active objects. By the ROI layer, the passive object model is transformed into an active object model. This transformation is transparent to the clients of the passive objects.

In this sense, the fundamental abstractions provided by an object-oriented parallel operating system must be complex enough to relieve the user from the technical details of systemwide message passing and CPU multiplexing between a couple of (related) threads, simple enough to guarantee high (communication) performance and suitable enough to support the modeling and implementation of almost arbitrary user/system applications. These abstractions must define a *framework* for parallel distributed system programming—and nothing else.

Chapter 8

Discussion

The foregoing chapters presented a *logical design* of a *family* of *parallel operating systems*. By starting on an abstract level, the "big picture" of a parallel operating system has been stepwise refined. The refinement was carried out top-down to the implementation level, to give an impression of the message startup time overhead implied by a hierarchically composed, family-based system. A functional decomposition took place, having elaborated three main building blocks: the nucleus, kernel and *parallel operating system extensions* (POSE). Section 2.2.1 gave a detailed description of these building blocks.

This coarse-grain structuring of the system was motivated by two fundamental ideas. First, the nodes of a distributed-memory parallel machine should be mainly allocated to user application tasks. If possible, the node resources should be made exclusively available to the user of the machine. The node resources are the CPU, communication devices *and* physical memory. Virtual memory is an operating-system feature that is not in all cases considered a "lucky dog," since it may cause a serious performance loss due to the (often unpredictable) paging overhead.

Virtual shared memory (VSM), however, is a different story. This approach aims at abstracting from the distributed-memory architecture by providing a common view of a global address space spanning the node memories of the parallel machine. The primary goal is not to tackle the problem of memory overallocation on a node; rather, paging serves to make remote data objects accessible, and memory protection enforces the controlled access of shared data sections. Page faults generally signal the read or write access to a particular (global) memory region, not the need to transfer a (readable and writable) memory region from secondary to primary storage.

In order to prevent thrashing effects and costly memory-to-memory paging, an efficient VSM system must harmonize with the compiler. The task of the compiler will be automatic data partitioning and establishing data localities, thus reducing the number of pages that constitute the shared memory space to an absolute minimum. On this basis, the VSM system will be responsible for controlled access only on the shared-memory fraction of the entire distributed address space. The designer must strive for a symbiosis between the compiler and VSM system. In other words, VSM is less an autonomous function of the parallel operating system than a function of the runtime environment supported by specific operating-system features.

Parallel machines are thought to accelerate applications such as weather forecasting, (winter/summer) smog prediction, volume rendering, protein synthesis, crash simulation, aerodynamics and so on. They are not thought to accelerate operating systems in general. Global (computing) resource sharing may become an issue only if the application tolerates the exploitation of the parallel machine to speed up (distributed) operating-system services. This aspect calls for a highly modular system structure. Whether operating-system functions are distributed over the entire parallel machine or moved closer to the application task should be a configuration decision. A parallel operating system must be organized in such a manner that it can, but not must, run on every node.

The second idea of designing the triumvirate of nucleus, kernel and POSE was to decouple performance-critical and performance-uncritical system components. That is, the design project should end up in a system which exhibits a single and, beyond that, small component guaranteeing a short message startup time and low message latency. This component was the nucleus. Design decisions that would have blown up the complexity of the nucleus were postponed. Complex software systems are extremely overhead-prone. Taking off located performance bottlenecks becomes difficult, if not impossible. An example of a postponed design decision has been multitasking. The minimal subset of nucleus functions implements only a single-user, single-tasking mode of operation. Multitasking, and thus address space protection, is introduced as a minimal nucleus extension. The same happens with multi-user support. In other words, the nucleus is implemented as a family of closely related modules dealing with thread communication, scheduling and synchronization, task scheduling and protection, kernel protection and security (refer to Section 2.4).

The nucleus is a *software backplane* and is the media by means of which plugged-in (user/system) entities can be executed and are able to communicate. Nodes *must* be equipped with the nucleus. The kernel acts as the *docking station* of a node, adding node-bound functions demanded by the application or by the underlying hardware architecture. Multitasking support is an example of the application case, and device driver support is an example of the hardware case. Nodes *can* be equipped with the kernel. POSE acts as a *warehouse* for the entire system, providing the user with high-level system services. Service examples are naming, process and memory management, file handling, host coupling and so on. Depending on whether the parallel machine is a hosted or self-hosted system, POSE services are executed mainly by the host or jointly by the nodes. Nodes *may* be equipped with POSE entities. See Section 3.1.2 for a discussion of the mapping of the three PEACE building blocks on the nodes of a distributed-memory parallel machine.

8.1 Microkernels

Reflecting the considerations made in the previous paragraphs, the question may come up of whether *microkernel technology* is appropriate or inappropriate to cope with the complexity and performance aspects mentioned. The introduction (Chapter 1) argued against *state-of-the-art* microkernel technology, but not against microkernel technology in general. This aspect will be revisited here to make clear why specifically designed parallel operating systems are required, and a new era of microkernel technology has to thrive.

During the past two decades, operating-system research was largely dominated by, or (more or less) put on a level with, UNIX research. The same applies to microkernels. Large efforts have been made developing a new technology through which UNIX functions appeared in a different shape. The era of monolithic systems came to an end. Mechanisms were developed aimed at migrating traditional operating system functions out of the monolithic kernel into user-level processes. So-called microkernels emerged. The appendage "micro" suggests that the kernel provides only a minimal function that allows user-level system processes to perform operating-system services efficiently. This function typically encompasses process management, inter-process communication, address space management and hardware (i.e., device) abstractions. However, it would be a wrong conclusion that microkernels are necessarily small in terms of code complexity [30]. The two dominant representatives of this new generation of operating-system technology, *Mach* [96] and *Chorus* [71], set a bad example when considering the software complexity of the kernel. Microkernels appeared on the (research/commercial) market the complexity of which is significantly greater than the complexity of the old-fashioned, monolithic and breakthrough bringing UNIX Version 6 [50, 51]. The latest example of this trend is WINDOWS-NT [24].

When comparing microkernel-based systems to monolithic systems, a shifting of functionality took place but complexity remained almost identical. The complexity problem given with monolithic systems will not be solved by providing a (complex) platform that allows the designer to migrate functions from kernel to user space. Monolithic systems are not "clumsy" just because all of their functions are executed within the kernel address space. Rather, these systems are often inherently inflexible and difficult to maintain because of a poor software structure. UNIX Version 6 could have been an outstanding operating system had its software structure not been so bad. The same opinion applies to all its successors, including the systems of the BSD[1] branch. UNIX Version 6 is understandable not because of its software structure but because of its relatively small number of C code lines.[2]

The poor software structure and the incidental shortcomings (not

[1] *Berkeley Software Distribution.*

[2] The entire Version 6 kernel consists of less than 10,000 lines of C code, including device drivers. Almost any computer science student is able to understand within one semester what happens inside the kernel. Only this aspect made UNIX a success story.

only) of monolithic operating systems led, in the '70s, to the idea of *operating-system families* [36, 66, 67]. It seems that this software engineering idea has been largely ignored by the operating-system community. Operating-system families and UNIX were established during the same period. This fact may give an answer to the question of why UNIX has not been subjected to a family-based design. The family concept did not proceed beyond the early stages, while UNIX was already on the way to becoming one of the most commercially successful operating systems ever built. Since the late '70s, every new timesharing operating system has had to compete with UNIX. The success and failure of the new systems depended on the achieved enhancements and UNIX compatibility. Microkernel development has been (and still is) confronted with exactly this situation. Current microkernels can therefore be regarded as the best possible compromise to introduce new operating-system technology and, at the same time, maintain UNIX compatibility. However, they cannot be regarded as the ultimate approach to developing parallel operating systems for any target market.

For example, custom-embedded or parallel systems play a slightly different game. All along, these systems make great demands on performance. Embedded systems operating in safety-critical environments also expect great things from the design methodology applied. The size of the entire system (in terms of memory space) is another important aspect. Operating systems in embedded systems usually reside in a PROM. UNIX compatibility is of minor importance, or incompatibility tolerated by the user, if performance, safety and size are all right. The rule of thumb was (and still is) keeping complexity as low as possible. A complex software system runs a very high risk of breeding inefficiencies and failing to meet safety checks. The concept of *program families* [65] was a meaningful contribution to avoiding these risks, and hence is worth experiencing a revival in the realm of massively parallel systems.

There are only a very few operating-system examples, not to say "exceptional cases," that are family-based designs. Modern representatives are *Choices* [14] and PEACE [77, 21, 80]. The family concept is a means to an end but not an end in itself. It requires that the designer be very disciplined and precise, if not pedantic, in the construction of operating-system software. Typically, an operating-system family exhibits a highly modular and hierarchical software structure. A system with tens of levels is not uncommon. The hierarchy is built by postponing design decisions that would restrict the already designed family [36]. A major concern is to decide which functions to exclude from a design. In other words, functions are introduced only on a per-application basis (i.e., if demanded by a higher layer). The new functions define an off-spring of the existing family.

Always forcing oneself to reason about the necessity of a certain function at a certain level is often deemed too academic, less pragmatic, causing an overstructuring of the resulting system—but it is worth it. The self-assessment during the design (and implementation) process helps to reduce the complexity of lower-level software components to an absolute minimum. Functions are not simply added to an existing compilation unit when they are not used by other functions of that unit. This rule holds even when new functions share the same knowledge (e.g., data structure) with the functions contained in the compilation unit. Similarly, the implementation of an *abstract data type* (ADT) must not necessarily mean ending up in a single compilation unit. In order to maintain the family idea, a restructuring of the existing ADT may be the consequence. The outcome of restructuring would then be that the new and old functions share the same knowledge but are not forced to coexist. This brief discussion hints that the implementation of a program family entails a large number of ADTs, compilation units or files.

Applied to the microkernel concept, a family-based design would imply providing the above-mentioned functions, such as process management, inter-process communication, address space management and hardware (i.e., device) abstractions, only on a per-application basis. In particular, the multi-user/multitasking capability of microkernels would have to be provided only by a dedicated family member, not all of them. Current microkernels are not designed this way. The resulting inflexibility of microkernels makes it extremely difficult to meet the specific demands of (massively) parallel distributed-memory architectures and (massively) parallel applications (see Chapter 1). Specific hardware developments become necessary to help microkernels execute applications more efficiently and, more precisely, to allow applications to bypass the microkernel. It is mandatory for an operating-system designer to first seek *lightweight software structures* before calling for faster hardware to fulfill the performance requirements.

Current microkernels are not lightweight enough to support a broad range of parallel applications. This situation induced diverse groups to invent the notion of the "*nano-kernel.*" An example from the parallel operating system area is *PARIX* [68]. However, even this attempt to express the light- or even featherweightness of microkernels is somewhat misleading. It does not really overcome the original problem, providing a computing platform that scales well not only with the underlying hardware but also with the given application. In many cases, the nano-kernel is nothing but a threads package running on the bare hardware and providing dedicated communication primitives. Multiuser/multitasking is not supported. Even the nano-kernel presents only a single solution to a problem space that requires a number of solutions. It does not scale up, while the microkernel does not scale down. A *microkernel family* would include both.

8.2 Programming Environments

Choosing the right programming language to implement operating systems is as important as choosing the right design methodology. In exceptional cases, the programming language used to implement the operating system practically arises, or is refined, with the operating-system implementation. This happened with C and UNIX; neither of them is imaginable without the other. In the meantime, compilers and operating systems have become too large to be developed in the same project within an acceptable time frame.

The previous section mentioned that the implementation of an ADT does not necessarily mean cramming everything into a single compilation unit. Whether or not this "fine-grain monolithic" implementation can be avoided depends on the programming language and/or programming environment. For example, if an ADT is implemented as an Oberon-2 [57] module, all functions exported by that ADT must be contained in the same compilation unit. Unless the compiler performs "dead code elimination," the one-to-one correspondence between the ADT and the compilation unit may result in a large program due to existing but unused functions. A solution to this problem is to implement an ADT in several modules. The drawback of this approach may be an increased runtime overhead when operations exported by the ADT are called and the compiler is unable to perform "function inlining." The resulting needlessly less efficient, large program (i.e., operating system) does not meet the original goals of a family-based design very well. Both examples show that a successful implementation of program families, in consideration of the family idea itself, requires a compiler offering specific optimization capabilities or a programming language allowing a decentralized implementation of an ADT.

A family-based implementation of an operating system can be achieved by making closely related functions a single compilation unit, building an archive of object modules generated from the compilation units and using a linker to statically assemble those functions from the archive that are really used by the given application program. The first aspect corresponds to the decentralized implementation of an ADT. It should be mentioned that the linking procedure may be supported by *dynamic binding* to allow for the generation of an incomplete system. Anyway, the rule of thumb is to make a family-based operating system appear as a collection of object modules kept in archives and not in directories. That is, a family-based operating system has to take the role of a library that can be more or less directly linked to application programs.

C++ [84] is an example of an object-oriented programming language that supports the decentralized implementation of an ADT in the above-mentioned sense. The ADT is implemented as a class that specifies the object state common to every method of that class. In addition, a complete class-based ADT implementation may be carried out by including the code body of every method in the class specification. Depending on the C++ compiler used and on code complexity of the methods, this approach may result in a highly efficient implementation. Method implementations contained in the class specification are automatically subjected to "function inlining." The other side of the coin may be code explosion, if the compiler is unable to inline the methods. In this case, every compilation unit that includes the class specification (i.e., C++ header file) contains statically declared methods implicitly generated by the compiler. The compiler enforces an out-of-line implementation if the method appears to be too complex for inlining.

Consequently, if an operating-system family consists of a large number of compilation units sharing the same class specification, code explosion may be tremendous.[3]

Reflecting the last paragraphs, the question is up for discussion as to whether a family-based design of an operating system can be implemented in a manner satisfying the family idea. Acceptance of a family member depends on the quality of its implementation. The best theory is of little value if working examples fail to demonstrate the feasibility of the proclaimed concepts. In many cases, there is no way to get around making a compromise concerning the programming language and specific language features not adequately supported by the compiler. Only a fully integrated design, with language, compiler, runtime and operating system checked against one another, will help to overcome the second aspect mentioned. The programming-language aspect still holds unless the integrated design starts from a standard, commonly accepted language definition. Even if, for example, C++ could be regarded as such a starting point, the size of the entire project should not be underestimated. Attempting to develop compiler, runtime and operating system in their entirety for a massively parallel (distributed-memory) machine may easily be beyond any scope.

Taking user applications into account, especially in the case of parallel systems, the language issue becomes predominant. It is safe to say that implementing an operating system, whether family-based or not, say, in Fortran would mean a setback to the Middle Ages. This aspect makes an integrated design basically impossible—at least, it extends the problem space by another component (e.g. a Fortran-to-C++ translator). The compromise is therefore unavoidable.

A justifiable compromise is C++. The language has a broad *ac-*

[3]The C++ implementation of the PEACE kernel encompasses about 120 classes, depending on the actual family member. Using the `cfront` to compile the kernel for a SPARC (i.e., RISC) processor, the code size is about 280 KB. In contrast, using the GNU `g++`, the code size of the same kernel is about 80 KB. This difference is mostly due to the restricted function inlining capabilities of the `cfront`. In addition, this difference is indicative of the large number of object modules, not the complexity of the PEACE kernel. Most of these object modules contain the statically declared methods that the `cfront` is unable to inline. Every method, whether used or not, too complex for inlining is statically declared. The `cfront` generates C code, while `g++` is a native code compiler.

ceptance in not only the application but also the system programming area. It offers high *expressiveness*, which makes the language specifically suited for system programming. Whatever a programmer wants to express, C++ will not prevent it. Note that this freedom may easily result in unreadable and hard-to-maintain software programs; therefore, a high programming discipline must be exercised to make C++ programs manageable. If leaving the object-oriented features of the language unexploited, C++ programs run with the same *efficiency* as equivalent C programs. Even object-oriented C++ programs exhibit a very good performance, which is not at all inferior to comparable C programs. Last but not least, C++ compilers have high *availability*. Almost every computer manufacturer has C++ in the product palette. The programmer can fall back on a set of existing high-quality production compilers.

The object-oriented side of C++ proves to be extremely *lightweight*. Using C++ to implement an object-oriented operating-system family has many advantages, although the language itself is not in all cases the best choice. Because of the static nature of C++, almost no "inheritance tax" must be paid for having built class hierarchies. By default, the inheritance model relies on multiple inclusion. Since dynamic binding is not a default case, "tax return" is possible. By supporting *class-based object orientation* [10], methods can be represented as *inline* functions. This way, "tax abatement" can be asserted. All these aspects enable the implementation of featherweight system components and disprove the "myth" that object-oriented operating systems can only offer poor performance in the area of distributed/parallel computing. The PEACE development proves that object-oriented operating systems can be highly efficient. In certain cases, the object-oriented implementation made its high performance possible after all.

8.3 Trends

A particular design has been presented in the preceding chapters. An in-depth presentation was given rather than a broad overview. Instead of discussing several designs more or less superficially, a single, elaborate case study was made on the basis of PEACE. The PEACE evolution can be described as follows:

1. Originally developed for the 256-node (MC68020-based) SUPRENUM supercomputer [31][4]

2. Ported to 8-, 16- and 40-node (T805-based) Transputer architectures

3. Redesigned and re-implemented in C++ for the (dual processor, i860-based) 2-, 16- and 64-node MANNA supercomputer [33]

4. Ported to (SPARC-based) SunOS 4.1 and adapted for use in workstation clustering

5. Currently ported to (i386-, 486-based) personal computers running MS-DOS 5.0 and WINDOWS-NT

6. Ported to *PowerPC*-based massively parallel computers

7. Extended by a *PARIX* [68] family member.

With the exception of the guest-level variants (Section 3.2) for MS-DOS, WINDOWS-NT and SunOS, the target architecture of the native system (Section 3.1) was always of distributed-memory type. In view of the main development phase from 1986 to 1993, PEACE has been approximately a 100-man-year project. Such a dimension is not atypical for operating-system projects in general. With this "biogenetical" background, PEACE turns out to be a qualified candidate to demonstrate fundamental design principles suitable for the development of highly *portable*, *efficient*, *adaptable*, *configurable* and *extensible* (i.e., "PEACE-ful") parallel operating systems.

Operating-system research would show weaknesses if there had been only a single design concept developed in the past years. The hottest research topic currently on the horizon, microkernel technology, has been already discussed in Section 8.1. Concerning the area of parallel operating systems, a trend is to also port microkernels to parallel computers. As mentioned in the introduction (Chapter 1), the attempt to promote microkernels as a high-performance parallel computing platform has

[4] Taking into account the four system nodes contained in each of the 16 clusters, SUPRENUM was a 320-node machine.

its limits. Table 1.2 illustrates with examples where these limits are presently set.

The main performance problem encountered with microkernels is the artificial boundary between user and kernel space. Unless specific hardware support is given, this boundary has a negative effect on the message startup time and message latency. From the point of view of the majority of parallel applications, the user/kernel space boundary is nothing but overhead, since in many cases a one-to-one task/node correspondence is given. That is, taking into account the organization of many parallel (SPMD) programs, there is no need to safely provide means for user task protection and thus isolate the kernel, because there is only a single user task per node.

The boundary is required by the microkernel architecture, not by the parallel application. Traditional kernel services are executed as system processes in user space. In the case of node-resident services, a multitasking mode of operation is the consequence, with only one user task and possibly several system tasks sharing the same node. The system task(s) must be protected from the user task, and the microkernel must be protected from the user task. In the light of these types of parallel applications, if performance is the predominant issue, specifically developed hardware becomes necessary only to bypass the microkernel. This hardware would be absolutely superfluous when sacrificing the microkernel and instead relying on a family-based operating-system design.

This observation should not lead to the false conclusion that multitasking support on a node is a feature never required by applications. There are applications whose number of tasks exceeds the number of nodes and is not known in advance but can be only determined at runtime. Concerning multitasking functions, these applications will be appropriately supported by a microkernel on those nodes which must be multiplexed between several user tasks. A *microkernel family* will help avoid the predicament. The family will provide special-purpose single-tasking members and general-purpose multitasking members. Furthermore, the members may be designed so that they are able to coexist on the same parallel machine. All members of the microkernel family will have to share the same low-level communication functions (i.e., protocol).

It was mentioned that the protection boundary between user and kernel space is one of the main performance-limiting factors. A modern approach to overcome the performance bottleneck is to design and implement the microkernel as a *single-address-space architecture* and bet on 64-bit processor technology. Memory addresses are 64 bits wide, and thus can be used as a *sparse capability* to uniquely identify objects. The system guarantees that the same address will "never" be allocated twice. A 64-bit address space ensures enough free play so that it is extremely unlikely that an address value is recreated during the lifetime of the entire system.

Single-address-space architectures have been pioneered by MONADS [41]. Further representatives of this approach are *Opal* [42], *Angel* [59] and PANDA [3, 4]. The exploitation of these architectures for parallel processing fails at the moment because a cost-effective 64-bit processor technology is not available. In addition, programming environment research must be advanced to solve the problems coming up, for example, with immortal objects and debugging. Even if these teething troubles were corrected, it seems as if single-address-space architectures are somewhat overdimensioned for typical parallel applications. The arguments given above, concerning the one-to-one node/task correspondence, still hold even in the light of 64-bit processor technology. In order to bring single-address-space architectures to the user, a rethinking of programming (massively) parallel systems has to take place, which calls for new programming models and sacrificing Fortran.

The *pico-kernel* used in the PANDA approach shows that microkernels and even nano-kernels are not the last resort. With operating-system kernels getting smaller and smaller—the *femto-kernel* will be not long in coming—, programming language and compiler support are of steadily increasing importance. As PANDA demonstrates, designing an extremely scaled-down and efficient kernel does not necessarily result in a user-friendly interface. However, the question is what user-friendliness means. From the viewpoint of the kernel designer, user-friendliness means that applications are not hampered by inappropriate kernel abstractions. From the viewpoint of an application designer, user-friendliness often means easy-to-use kernel functions. In order not to lose all the advantages (in terms of efficiency and flexibility) and hide the peculiarities of extremely scaled-down kernels, compiler and/or programming-language support is required.

Besides architectural issues, current trends in parallel operating-system design concentrate on (static/dynamic) *load balancing*, *fault tolerance*, *parallel I/O*, *monitoring* and *debugging*. It is still an open question how the activities of thousands of autonomous, interconnected nodes can be supervised, controlled and visualized in a user-friendly manner. The design principles used to develop a family of parallel operating systems are, of course, suited to develop these (POSE) services. In this sense, the investigations made with respect to an overall operating-system structure for massively parallel computer architectures allows one to draw conclusions on how to design specific service functions for these architectures.

VSM research (see also Section 8.1) is very popular to support not only parallel but also distributed computing. The view of a common, global address space (distributed over the nodes) helps to overcome difficulties in programming distributed-memory architectures. However, in order to efficiently employ this global address space, the user is often implicitly forced to program in a distributed-memory fashion. Data localities are to be built and access to non-local data minimized. This way of programming in a virtual shared memory context is not very far away from programming in a message-passing context—and is not intended. It has been outlined above that the ultimate goal of VSM is to support automatic parallelizing compilers. This way, contemporary shared-memory programs are made executable on distributed-memory machines. VSM is less a programming model than the means to an end; namely, to aid the compiler in its attempt to automatically parallelize a given program and to perform data partitioning.

8.4 Summary

Parallel operating systems have their own raison d'être. They are designed to specifically support the execution of parallel programs on parallel machines. Message-passing architectures place special importance upon and, in consideration of the specific demands of parallel applications, make extremely difficult successfully adopting existing operating systems. This is especially true for microkernel-based operating systems.

Operating-system construction is primarily a design and structuring problem, which does not cease to exist in the countenance of parallel operating systems. The contrary is the case. In the realm of massively parallel systems consisting of several thousands of autonomous nodes interconnected by a high-speed network, the operating-system design is crucial to achieving high performance. A lightweight system structure is required, meaning to advance into traditional frontier districts of operating systems. The boundary where a parallel operating system begins and a runtime environment ends becomes indistinct. Furthermore, the boundary between application and operating system becomes indistinct. Massively parallel machines make the amalgamation of hardware, operating systems, runtime environments, compiler, programming language and application more important than ever.

The main design goal must be that the user will never have to worry about how to bypass the operating system in order to squeeze the utmost performance out of the parallel machine. First, the software must be in such shape that further optimization is not necessary. This statement is very idealistic and reminds the software engineer of his duty to exhaust all possibilities to succeed in getting a highly satisfactory operating-system implementation. If required, the call for faster or supporting hardware should only be the second step.

A reminiscence of merits, which might have fallen into oblivion, is not always bad. Pioneering innovations are extremely hard to find and, indeed, will not come out of the blue. In this sense, the art of operating-system design is to combine well-established principles with new or different application *and* hardware paradigms.

Glossary

Active Object An object possessing an autonomous thread of control. The thread of control is implemented by a lightweight process, and the object state serves as *own variables* of that process.

Address Space An ensemble of logically correlated segments. The address space stores a program in execution and may be distributed segment-wise over the local memories of the nodes of the parallel machine.

Application A possibly distributed user or system program. The program may consist of several entities.

Bundle The unit of preemption. A bundle consists of one or more threads. Threads of the same bundle are sequentially executed. Several bundles are executed concurrent to each other. In shared-memory multiprocessor systems, different bundles may be executed in parallel by different processors.

CISC *Complex Instruction Set Computer.*

Clerk A lightweight process handling ROI requests sent to the prototype instance of a dual object. Usually, the ROI requests were issued by the likeness instance belonging to the prototype.

Entity The unit of distribution. Used as the software-technical equivalent of "being" or "creature." An entity defines a common address space for one or more bundles of threads. Entities represent user or system programs.

Dual Objects A distributed object consisting of two closely related but physically separated fragments. The prototype fragment has a private and a public state. The likeness fragment consists of the (extracted) public prototype state and possesses a private state. Operations on a likeness are performed via LOI. Operations on a prototype by means of a likeness are performed on an ROI basis. Operations on a prototype without applying a likeness instance are performed via LOI. Likeness creation entails prototype creation, but not vice versa. A likeness always knows its prototype and provides network-transparent access to the private prototype state.

FCFS *First Come First Serve* scheduling discipline. Threads, or bundles of threads, are scheduled in the order of their insertion in the ready list.

FIFO *First In First Out* queuing discipline. Also referring to a hardware queue implemented by a high-speed register bank.

GFLOPS *Giga FLOating Point operations per Seconds.* Specifies the vector performance of a processor.

Heavyweight Process A process possessing a private address space. The heavyweight process consists of at least a single thread of control (i.e., lightweight process). Switching between heavyweight processes means switching between address spaces. Address space switching involves MMU programming and possibly the flushing of caches used for fast translation of logical to physical addresses (unless the cache entries are tagged with process identifiers). Flushing the MMU cache entails costly memory accesses to cache MMU descriptors when the activated heavyweight process starts/continues execution.

Horizontal Isolation MMU-based protection of the address spaces of competing (user/system) tasks. More specifically, prohibiting user-mode entities from accessing each other's address space. The (supervisor-mode) kernel is granted complete access rights.

Incremental Bootstrapping On-demand bootstrapping of the nodes of the parallel machine; part of the incremental loading procedure.

Incremental Loading On-demand loading of system services. The services are implemented as transient objects. Their loading is initiated through a naming exception, that is, after the (client) ROI layer has detected that a particular clerk does not exist.

Kernel The subsystem constituted by site-dependent active objects. It introduces minimal system extensions to the nucleus, provides a dynamic process and address space model, and implements hardware abstractions.

Lightweight Process A process sharing with other processes the same address space. Logically, the lightweight process has a private stack segment and implicitly shares the code and data segment with other lightweight processes. Context switching between lightweight processes does not involve address-space switching, but merely the switching of stack segments and processor registers.

LOI *Local Object Invocation.* Denotes an object-bound local procedure call. The calling thread contains execution within the same address space (i.e., protection domain).

MFLOPS *Mega FLOating Point operations per Seconds.* Specifies the vector performance of a processor.

MIPS *Million Instructions Per Second.* Specifies the scalar performance of a processor.

MMU *Memory Management Unit.* Hardware support required to protect address spaces and implement a virtual (shared) memory system.

NOI *Nearby Object Invocation.* Object-bound *lightweight remote procedure call.* The calling thread continues execution in a different address space (i.e. protection domain) on the same node.

Nucleus The subsystem constituted by site-dependent passive objects. It serves as a minimal subset of system functions that support the processing of parallel programs in a distributed memory environment. Provides a static process model and takes care of systemwide communication.

Own Variables The private data space of a (lightweight) process. This data space remains consistent across process switches.

Passive Object An object without autonomous control flow. The object is activated only by the explicit invocation of one of its exported methods.

PEACE *Process Execution And Communication Environment.* A synonym for the nucleus and kernel, which together build an abstract machine for the processing of active objects in a parallel/distributed environment.

POSE *Parallel Operating System Extension.* The subsystem constituted by site-independent active objects. Provides application-oriented services. Services are implemented as transient objects.

Process Used as a synonym for either *heavyweight process* or *lightweight process* when the different context-switching properties appear to be insignificant.

RISC *Reduced Instruction Set Computer.*

ROI *Remote Object Invocation.* Object-bound RPC. The calling thread (client) suspends execution, and a different thread (server) is in charge of executing the requested function. The server may execute in a different address space (i.e., protection domain) than the client and also on a different node. ROI implies a synchronous control flow transfer between threads, but not necessarily a change of protection domains implemented by address spaces and/or nodes.

RPC *Remote Procedure Call.* A synchronous control flow transfer via a "narrow channel."

Segment The constituent of an address space. A segment is used for structuring an address space and may be protected by exploiting specific hardware measures such as an MMU. By relying on a paging MMU, a segment may also be physically distributed over the local memories of the nodes of the parallel machine.

SPMD *Single Program, Multiple Data.* A parallel program equally distributed over the nodes of a parallel machine. Every node executes the same program or task. This single but replicated program is fed with different data sets. Data parallel programming is exploited.

Task Synonym for *heavyweight process.*

Team The unit of comradeship. A collection of cooperating threads executing the same nondistributed program (fragment). The threads may belong to different bundles.

Thread Synonym for *lightweight process*.

Transient Objects Mobile entities subjected to incremental loading and garbage collection. These entities are implemented by dual objects.

Vertical Isolation MMU-based protection of the kernel address space. More specifically, denying a user-mode entity access to the kernel address space, but granting the (supervisor-mode) kernel entity access to the user-mode address space.

Bibliography

[1] T. E. Anderson, B. N. Bershad, E. D. Lazowska, and H. M. Levy. Scheduler Activations: Effective Kernel Support for the User-Level Management of Parallelism. In *Proceedings of the Thirteenth ACM Symposium on Operating System Principles*, volume 25 of *Operating Systems Review*, pages 95–109, Pacific Grove, CA, 1991.

[2] Arvind. Dataflow Architectures: From Monsoon to *T. Talk given at GMD FIRST, Berlin, 1992. Research Colloquium.

[3] H. Assenmacher, T. Breitbach, P. Buhler, V. Hübsch, and R. Schwarz. PANDA—Supporting Distributed Programming in C++. In O. Nierstrasz, editor, *Proceedings of the 7th European Conference on Object-Oriented Programming (ECOOP '93)*, volume 707 of *Lecture Notes in Computer Science*, pages 361–383. Springer-Verlag, July 1993.

[4] H. Assenmacher, T. Breitbach, P. Buhler, V. Hübsch, and R. Schwarz. The PANDA System Architecture—A Pico-Kernel Approach. In *Proceedings of the Fourth Workshop on Future Trends of Distributed Computing*, pages 470–476, Lisbon, Portugal, September 1993. IEEE Computer Society Press.

[5] J. Backus. Can Programming Be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs. *Communications of the ACM*, 21(8):613–641, 1978.

[6] M. J. Beckerle. Overview of the START(*T) Multihreaded Computer. In *Digest of Papers of the 38th IEEE Computer Scociety In-*

ternational Conference, Spring COMPCON '93, IEEE, pages 148–156, San Francisco, USA, 1993.

[7] P. M. Behr, W. K. Giloi, and W. K. Mühlenbein. Rationale and Concepts for the SUPRENUM Supercomputer Architecture. In *Proceedings of the IFIP Working Conference on Highly Parallel Computers for Numerical and Signal Processing Applications*, Nice, France, March 24–26, 1986.

[8] P. M. Behr, W. K. Giloi, and W. Schröder. Synchronous vs. Asynchronous Communication in High Performance Multicomputer Systems. In *Proceedings of the 1988 IFIP WG 2.5 Working Conference on Aspects of Computation and Asynchronous Parallel Processors*, pages 239–248, Stanford, CA, August 22–26, 1989. North-Holland, Amsterdam.

[9] B. N. Bershad, T. E. Anderson, E. D. Lazowska, and H. M. Levy. Lightweight Remote Procedure Call. *ACM Transactions on Computer Systems*, 8(1):37–55, 1990.

[10] G. S. Blair, J. J. Gallagher, and J. Malik. Genericity vs Inheritance vs Delegation vs Conformance vs... *Journal of Object-Oriented Programming*, 2(3):11–17, 1989.

[11] S. Borkar, R. Cohn, G. Cox, T. Gross, H. T. Kung, M. Lam, M. Levine, B. Moore, W. Moore, C. Peterson, J. Susman, J. Sutton, J. Urbanski, and J. Webb. Supporting Systolic and Memory Communication in iWarp. Technical Report CMU-CS-90-197, Carnegie Mellon University, 1990.

[12] U. Brüning. Personal communication, 1994.

[13] R. Bryant, H.-Y. Chang, and B. Rosenburg. Experience Developing the RP3 Operating System. *Computing Systems, The Journal of USENIX Association*, 4(3):183–216, 1991.

[14] R. Campbell, G. Johnston, and V. Russo. Choices (Class Hierarchical Open Interface for Custom Embedded Systems). *Operating Systems Review*, 21(3):9–17, 1987.

[15] D. R. Cheriton. *The Thoth System.* Elsevier Science Publishing Co., 1982.

[16] D. R. Cheriton. The V Kernel: A Software Base for Distributed Systems. *IEEE Software*, 1(2):19–43, 1984.

[17] D. R. Cheriton. Request-Response and Multicast Interprocess Communication in the V Kernel. *Lecture Notes of Computer Science*, (248):297–312, 1986.

[18] D. D. Clark. The Structuring of Systems Using Upcalls. *Operating Systems Review*, 19(5):171–180, 1985.

[19] J. Cordsen. Personal communication, 1992.

[20] J. Cordsen, M. Krauser, and P. Preikschat. Personal communication, 1993.

[21] J. Cordsen and W. Schröder-Preikschat. Object-Oriented Operating System Design and the Revival of Program Families. In *Proceedings of the Second International Workshop on Object Orientation in Operating Systems (I-WOOOS '91)*, pages 24–28, Palo Alto, CA, October 17–18, 1991.

[22] J. Cordsen and W. Schröder-Preikschat. Towards a Scalable Kernel Architecture. In *Proceedings of the OpenForum '92 Technical Conference*, Utrecht, The Netherlands, November 25–27, 1992.

[23] Thinking Machines Corp. The Connection Machine CM–5 Technical Summary. System Reference Manual, October 1991.

[24] H. Custer. *Inside WINDOWS-NT.* Microsoft Press, 1993.

[25] E. Dijkstra. The Structure of the THE-Multiprogramming System. *Communications of the ACM*, pages 341–346, May 1968.

[26] R. P. Draves, B. N. Bershad, R. F. Rashid, and R. W. Dean. Using Continuations to Implement Thread Management and Communication in Operating Systems. In *Proceedings of the Thirteenth ACM Symposium on Operating System Principles*, volume 25 of

Operating Systems Review, pages 122–136, Pacific Grove, CA, 1991.

[27] R. Esser and R. Knecht. Intel Paragon XP/S—Architecture and Software Environment. In *Proceedings of Supercomputer '93*, Lecture Notes in Computer Science, Mannheim, Germany, June 24–26 1993. Springer-Verlag.

[28] R. S. Fabry. Capability-Based Addressing. *Communications of the ACM*, 17(7):403–412, July 1974.

[29] D. Ferrari. Real–Time Communication in an Internetwork. Technical Report TR-92-001, International Computer Science Institute, Berkeley, CA, 1992.

[30] M. Gien. Micro-kernel Architecture – Key to Modern Operating System Design. Technical Report CS/TR-90-42.1, Chorus systèmes, Paris, 1990.

[31] W. K. Giloi. The SUPRENUM Architecture. In *Proceedings of CONPAR 88*, pages 10–17, Manchester, UK, September 12–16, 1988. Cambridge University Press.

[32] W. K. Giloi. Personal communication, 1993.

[33] W. K. Giloi and U. Brüning. Architectural Trends in Parallel Supercomputers. In *Proceedings of the Second NEC International Symposium on Systems and Computer Architectures*, Tokyo, August 1991. Nippon Electric Corp.

[34] W. K. Giloi and W. Schröder-Preikschat. Very High-Speed Communication in Large MIMD Supercomputers. In *Proceedings of the International Conference on Supercomputing (ICS '89)*, pages 313–321, Crete, Greece, June 5–9, 1989.

[35] R. Govindan and D. P. Anderson. Scheduling and IPC Mechanisms for Continuous Media. In *Proceedings of the Thirteenth ACM Symposium on Operating System Principles*, volume 25 of *Operating Systems Review*, Pacific Grove, CA, 1991.

[36] A. N. Habermann, L. Flon, and L. Cooprider. Modularization and Hierarchy in a Family of Operating Systems. *Communications of the ACM*, 19(5):266–272, 1976.

[37] P. B. Hansen. *Operating System Principles.* Prentice Hall International, 1973.

[38] J. Hennessy. Scalable Multiprocessors and the DASH Approach. Lunch Video, March 18, 1993.

[39] A. J. Herbert and J. Monk. *ANSA Reference Manual.* Advanced Network Systems Architecture, Cambridge, UK, 1987.

[40] C. Hewitt. Viewing Control Structures as Patterns of Passing Messages. *Artificial Intelligence*, (8):323–364, 1977.

[41] J. F. Keedy. The MONADS-PC System: A Programmers Overview. Technical Report 8/89, Universtät Bremen Informatik, Germany, 1989.

[42] E. J. Koldinger, J. S. Chase, and S. J. Eggers. Architectural Support for Single Address Space Operating Systems. In *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 175–186, 1992.

[43] J. Kramer and J. Magee. Dynamic Configuration for Distributed Systems. *Transactions of Software Engineering*, SE-11(4), 1985.

[44] K. A. Lantz, W. I. Nowicki, and M. M. Theimer. An Empirical Study of Distributed Application Performance. Technical Report STAN-CS-86-1117, Department of Computer Science, Stanford University, 1985. Also available as CSL-85-287.

[45] H. C. Lauer and R. M. Needham. On the Duality of Operating System Structures. *Operating Systems Review*, 13(2):3–19, April 1979.

[46] D. Lenoski, K. Laudon, K. Charachorloo, W.-D. Weber, A. Gupta, J. Hennessy, M. Horowitz, and M. Lam. The Stanford Dash Multiprocessor. *Computer*, 25(3):63–79, March 1992.

[47] K. Li. *Shared Virtual Memory on Loosely Coupled Multiprocessors.* PhD thesis, Yale University, 1986.

[48] K. Li and R. Schaefer. A Hypercube Shared Virtual Memory System. In *Proceedings of the 1989 International Conference on Parallel Processing*, volume 1, pages 125–132, 1989.

[49] INMOS Limited. *Transputer Instruction Set: A Compiler Writer's Guide.* Prentice Hall International, 1988.

[50] J. Lions. A Commentary on the UNIX Operating System. Technical report, Department of Computer Science, The University of New South Wales, 1977. Second Printing.

[51] J. Lions. UNIX Operating System Source Code Level Six. Technical report, Department of Computer Science, The University of New South Wales, 1977. Second Printing.

[52] B. H. Liskov. Primitives for Distributed Computing. In *Proceedings of the Seventh ACM Symposium on Operating System Principles*, volume 13 of *Operating Systems Review*, pages 33–42, 1979.

[53] B. H. Liskov and S. Zilles. Programming with Abstract Data Types. *SIGPLAN Notices*, 9(4), 1974.

[54] B. D. Marsh, M. L. Scott, T. J. LeBlanc, and E. P. Markatos. First-Class User-Level Threads. In *Proceedings of the Thirteenth ACM Symposium on Operating System Principles*, volume 25 of *Operating Systems Review*, pages 110–121, Pacific Grove, CA, 1991.

[55] Meiko Ltd., Bristol, UK. *CS/Tools Programmer's Manual*, 1990.

[56] H. Mierendorff. Bounds on the Startup Time for the GENESIS Node. Technical report, GMD F2.G1, Bonn, Germany, 1989. ESPRIT Project No. 2447.

[57] H. Mössenböck and N. Wirth. The Programming Language Oberon-2. *Structured Programming*, 12(4), 1991.

[58] S. J. Mullender and A. S. Tanenbaum. The Design of a Capability-Based Distributed Operating System. *The Computer Journal*, 29(4):289–299, 1986.

[59] K. Murray, T. Wilkinson, T. Stimerling, and P. Kelly. Angel: Resource Unification in a 64-bit Microkernel. In *Software Technology*, Proceedings of the Twenty-Seventh Annual Hawaii International Conference on System Sciences, Maui, Hawaii, January 4–7 1994. IEEE Computer Society Press.

[60] B. J. Nelson. Remote Procedure Call. Technical Report CMU–81–119, Carnegie-Mellon University, 1982.

[61] J. Nolte. Language Level Support for Remote Object Invocation. Arbeitspapiere der GMD 654, Gesellschaft für Mathematik und Datenverarbeitung, St. Augustin, Germany, June 1992.

[62] J. Nolte and W. Schröder-Preikschat. Modeling Replication and Placement in the PEACE Parallel Operating System—A Case for Dual Objects. In *Proceedings of the Workshop on Dynamic Object Placement and Load Balancing in Parallel and Distributed Systems (ECOOP '92)*, Utrecht, The Netherlands, June 29, 1992.

[63] B. Oestmann. Personal communication, 1992.

[64] E. Organick. *The Multics System: An Examination of its Structure*. MIT Press, 1972.

[65] D. L. Parnas. On the Design and Development of Program Families. Technical Report BS I 75/2, TH Darmstadt, 1975.

[66] D. L. Parnas. Some Hypotheses About the "Uses" Hierarchy for Operating Systems. *Transaction on Software Engineering*, 1976.

[67] D. L. Parnas. Designing Software for Ease of Extension and Contraction. *Transaction on Software Engineering*, SE-5(2), 1979.

[68] Parsytec Computer GmbH, Aachen, Germany. *PARIX Programmer's Manual*, 1991.

[69] J. L. Peterson and A. Silberschatz. *Operating System Concepts*. Addison-Wesley, second edition, 1985.

[70] Quantum Software Systems Ltd., Toronto, Canada. *QNX Operating System User's Manual*, 1984.

[71] M. Rozier, V. Abrossimov, F. Armand, I. Boule, M. Gien, M. Guillemont, F. Herrman, C. Kaiser, S. Langois, P. Leonard, and W. Neuhauser. CHORUS Distributed Operating Systems. *Computing Systems Journal*, 1(4), 1988. Also as Technical Report CS/TR-88-7.9, Chorus systèmes, Paris.

[72] J. H. Saltzer, D. P. Reed, and D. D. Clark. End-To-End Arguments in System Design. *Transactions on Computer Systems*, 2(4):277–288, November 1984.

[73] S. Schindler. Das DAS System. In *GI Jahrestagung*, 1978.

[74] H. Schmidt. Making PEACE a Dynamic Alterable System. In Arndt Bode, editor, *Proceedings of the 2nd European Distributed Memory Computing Conference*, volume 487 of *Lecture Notes in Computer Science*, pages 422–431, Munich, Germany, April 22–24, 1991. Springer Verlag.

[75] W. Schröder. *A Family of UNIX-Like Operating Systems – Use of Processes and the Message-Passing Concept in Structured Operating System Design*. PhD thesis, Technical University of Berlin, 1986. German.

[76] W. Schröder. Concepts of a Distributed Process Execution and Communication Environment (PEACE). Technical report, GMD FIRST, Berlin, FRG, 1987.

[77] W. Schröder. A Distributed Process Execution and Communication Environment for High-Performance Application Systems. In J. Nehmer, editor, *Proceedings of the International Workshop on Experiences with Distributed Systems*, volume 309 of *Lecture Notes in Computer Science*, pages 162–188, Kaiserslautern, Germany, September 28–30, 1987. Springer-Verlag.

[78] W. Schröder-Preikschat. Overcoming the Startup Time Problem in Distributed Memory Architectures. In Veljko Milutiniovic and Bruce D. Shriver, editors, *Architecture and Emerging Technologies Tracks*, volume 1 of *Proceedings of the Twenty-Fourth Annual Hawaii International Conference on System Sciences*, pages 551–559, Kauai, Hawaii, January 8–11, 1991. IEEE Society Press.

[79] W. Schröder-Preikschat. PEACE—The Evolution of a Parallel Operating System. Arbeitspapiere der GMD 646, Gesellschaft für Mathematik und Datenverarbeitung, St. Augustin, Germany, May 1992.

[80] W. Schröder-Preikschat. Object-Orientation in a Family of Parallel Operating Systems. In Ted Lewis Hesham El-Rewini and Bruce D. Shriver, editors, *Software Technology*, volume 2 of *Proceedings of the Twenty-Sixth Annual Hawaii International Conference on System Sciences*, pages 60–69, Maui, Hawaii, January 5–8 1993. IEEE Computer Society Press.

[81] M. L. Scott, T. J. LeBlanc, and B. D. Marsh. Design Rationale for Psyche, a General-Purpose Multiprocessor Operating System. In *Proceedings of the 1988 International Conference on Parallel Processing*, pages 255–262, August 1988.

[82] M. Shapiro. Structure and Encapsulation in Distributed Systems: the Proxy Principle. In *Proceedings of the 6th International Conference on Distributed Computing Systems*, pages 198–204, Cambridge, MA, 1986.

[83] H. J. Siegel. Report on the Purdue Workshop on Grand Challenges in Computer Architecture for the Support of High Performance Computing. *Journal of Parallel and Distributed Computing*, 16(6):199–211, 1992.

[84] B. Stroustrup. *The C++ Programming Language*. Addison-Wesley, 1986.

[85] C. A. Szyperski. *Insight ETHOS: On Object-Orientation in Operating Systems*. Verlag der Fachvereine Zürich, 1992. Informatik-Dissertationen ETH Zürich, Nr. 40.

[86] A. S. Tanenbaum. *Structured Computer Organization.* Prentice Hall, 1985. Second Edition.

[87] A. S. Tanenbaum. *Computer Networks.* Prentice Hall, 1988. Second Edition.

[88] Telmat, France. *UBIK Programmer's Manual*, 1990.

[89] K. Thompson and D. M. Ritchie. The UNIX Timesharing System. *Communications of the ACM*, 17(7):365–375, 1974.

[90] T. von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser. Active Messages: A Mechanism for Integrated Communication and Computation. Technical Report UCB/CSD 92/675, University of California, Berkeley, CA, 1992.

[91] P. Wegner. Classification in Object-Oriented Systems. *SIGPLAN Notices*, 21(10):173–182, 1986.

[92] S. R. Wheat, A. B. Maccabe, R. Riesen, D. W. van Dresser, and T. M. Stallcup. Puma: An Operating System for Massively Parallel Systems. In *Software Technology*, Proceedings of the Twenty-Seventh Annual Hawaii International Conference on System Sciences, Maui, Hawaii, January 4–7 1994. IEEE Computer Society Press.

[93] M. Wilhelmi. Personal communication, 1993.

[94] N. Wirth. Program Development by Stepwise Refinement. *Communications of the ACM*, 14(4):221–227, 1971.

[95] N. Wirth. The growth of software. Invited Talk at the Twenty-Seventh Annual Hawaii International Conference on System Sciences, Maui, Hawaii, January 4–7, 1994.

[96] M. Young, A. Tevanian, R. Rashid, D. Golub, J. Eppinger, J. Chew, W. Bolosky, D. Black, and R. Baron. The Duality of Memory and Communication in the Implementation of a Multiprocessor Operating System. In *Proceedings of the Eleventh ACM Symposium on Operating System Principles*, volume 21 of *Operating Systems Review*, pages 63–76, Austin, TX, 1987.

[97] R. Zajcew, P. Roy, D. Black, C. Peak, P. Guedes, B. Kemp, J. LoVerso, M. Leibensperger, M. Barnett, F. Rabii, and D. Netterwala. An OSF/1 UNIX for Massively Parallel Multicomputers. In *Proceedings of the USENIX Winter Technical Conference*, pages 449–468, San Diego, CA, USA, January 25-29, 1993.

[98] W. Zwaenepoel. Protocols for Large Data Transfers Over Local Networks. In *Proceedings of the 9th Symposium on Data Communications*, pages 22–32, Whistler Mt., BC (Canada), September 1985.

Index

abstract data type, 30, 47, 195, 319, 354, 412
  implementation, 413
abstract machine, 158, 406
abstract processor, 99, 320
active message, 16, 186, 338
  system integrity, 194
active object, 49, 58, 88, 192, 220, 285, 310, 313–314, 400
  activation, 323
  autonomous method, 313
  construction, 331
  initial flow of control, 316
  installation problem of an, 331–332
  instantiation, 332
  interrupt handling, 285
  kernel emigrant, 323
  kernel immigrant, 323
  kernel level, 310, 311, 318–327
  specialized passive object, 285
  user level, 311, 314–316, 323
    activation, 315
adaptive class, 324
address space
  kernel space, 9
  multithreaded, 41
  protection, 8
  user space, 9
address space model, 227, 308, 309
  multiple address spaces, 18, 111, 221, 294
  single address space, 18, 52, 111, 209, 214, 221, 226, 294, 308, 390

busy waiting, 43, 156, 184, 384

C++
  a justifiable compromise, 415
  advantages of, 271, 414, 416
  class initialization, 279
  compiler, 271
    problems with, 414
  constructor invocation, 274
  destructor invocation, 274
  disadvantages of, 150, 277, 289
  hidden management data, 338
  object construction, 331
  object destruction, 331
  problems with, 279
  templates, 331
  virtual function
    overhead, 324
    pure, 284
call
  inter-domain, 53, 54, 69
    carrier object, 54
    trap, 53
  intra-domain, 53
capability, 223
  random number, 237

revocation of a, 237
sparse, 237, 419
class ensemble
basic abstractions
`annex`, 268, 291, 293
`atomicCounter`, 272, 273
`atomicQueue`, 270
`chain`, 267–268, 301, 325, 334, 335, 399
`counter`, 270, 272, 273, 306, 326, 327
`event`, 268, 269, 293
`guard`, 270–271
`hurdle`, 272, 274
`locker`, 275, 276
`queue`, 267, 268, 270, 271, 292, 326
`secure`, 274, 275
`sluice`, 270–272, 274
`stack`, 267, 268
communication abstractions
`carePacket`, 334
`careWagon`, 334
`holder`, 337, 343, 345, 388
`mailbox`, 322, 326, 335
`nice`, 339
`niceCharge`, 345
`niceExport`, 343
`niceFetch`, 344
`niceForward`, 346
`niceImport`, 343, 344
`niceLiaise`, 341–343, 345, 387
`niceOpaque`, 341
`nicePacket`, 335, 337
`niceRegion`, 343–345
`niceRequest`, 345, 384, 389
`niceRespond`, 345
`niceResume`, 342
`niceStore`, 344, 388, 389
`niceToggle`, 342
`niceWagon`, 335
`plug`, 338
`signalbox`, 322
`timesheet`, 322
`wagon`, 334, 335
device abstractions
`clockDevice`, 292, 293
`clockInstance`, 293
`timer`, 293, 322
`timerInstance`, 293
`watch`, 292, 293
hashing abstractions
`hashSimple`, 294
`guide`, 305, 307, 309, 310, 313, 322, 366, 381, 383
`hash`, 302
`hashOpaque`, 304
`hashSimple`, 303
`hashUnique`, 303
`item`, 301, 303, 306
`itemSimple`, 301, 303
`opaque`, 383, 387
kernel abstractions
`gate`, 307, 309, 311, 312, 315, 328, 331, 357, 380
`germ`, 312, 314, 323, 332
`loophole`, 306, 307, 328, 367, 378
kernel extensions
`thing`, 328–330
`thingDomain`, 328
`thingEmigrant`, 328, 330
`thingGate`, 328
`thingLoophole`, 328
nucleus abstractions
`charge`, 332, 378
`container`, 334, 337, 345

`lui`, 300, 301, 303, 387
`notice`, 316, 358, 378, 380
`opaque`, 300, 304, 307, 334, 336, 341, 345
`packet`, 333, 337, 345, 399
`parcel`, 332, 335, 339, 345, 346, 360, 369, 370, 378, 380, 383, 384, 387, 394, 398
`region`, 298, 332, 335, 371, 372, 378
`sector`, 315, 334, 343, 344
`sparse`, 300, 302, 387
`ticket`, 306, 311, 339, 357, 373, 378, 399
`unique`, 300, 301, 306, 307, 344, 387, 388, 399

POSE abstractions
`associate`, 308, 309, 314, 328, 330, 331
`canal`, 399
`Cthread`, 317
`entity`, 355, 400
`morsel`, 396
`primer`, 316, 400
`reader`, 394–397
`thread`, 313, 314, 317, 318, 329–331, 355, 400
`tube`, 394, 395, 397
`writer`, 394, 395

process abstractions
`activity`, 321, 349
`basket`, 326
`bundle`, 326
`colleague`, 325
`competitor`, 325
`coroutine`, 285, 288, 289, 320, 349
`cpuSpirit`, 320, 347
`domain`, 327, 328
`elevator`, 323
`emigrant`, 323, 327
`estate`, 322, 327, 380
`immigrant`, 323
`kindred`, 325, 327
`mailer`, 322, 326, 380
`motive`, 322, 324, 326, 328, 330
`native`, 285, 288, 291, 313, 314, 318, 322, 327, 330, 331
`process`, 322, 324, 330

processor abstractions
`aplRegion`, 298
`bridge`, 279, 280
`carrier`, 296
`coupling`, 286, 288, 291, 292, 294
`cpuPoints`, 279, 281
`cpuVector`, 279, 281
`escape`, 282, 283, 296
`lifter`, 279
`linkage`, 279, 280, 286, 291
`points`, 279, 281, 284, 286, 312, 314, 321, 323, 330
`regionStore`, 298
`shepherd`, 288–291
`vice`, 294
`victim`, 288, 289

class library, 353

communication
asynchronous, 41, 43, 332, 371
single-buffered, 369
capability, 70, 357
domain, 70
exception, 244
firewall, 70, 357
forwarding identifier, 366

lazy synchronization, 42
logical connection, 373
message forwarding, 365
no-wait-send, 43
packet transfer, 360–370
  motivation, 360
per-thread mailbox, 361
pipelined, 345, 368
preemptive, 133
receive
  blocking, 361
  nonblocking, 362
  nonspecific, 361
remote memory read/write, 372
rendezvous, 360
  delegation, 364
  forwarding, 364
  principle, 333
  relaxed, 369, 370, 383
  segment-based, 371, 374
  with a team, 362
request/response model, 345, 360
segment transfer, 370–381
  monitoring, 375
  ticket-based, 373, 388
  transitivity, 375
semantic gaps, 17
synchronization-send, 43
synchronous, 41
  packet-based, 332
  relaxed, 368
  segment-based, 332
  vs. asynchronous, 376
transfer event, 375
transfer ticket, 373, 375
transport protocol, 370
  packet retransmission, 293
  timeout handling, 293
communication endpoint address, 207, 213, 229, 299, 357
  forwarding identifier, 236, 241
  interpreted by the nucleus, 231
  lifetime, 232
  opaque pointer, 300
    `const opaque*`, 300
  receiver-site interpretation, 341
  simple data type, 231
  structure, 232
  systemwide unique identifier, 231
  technical representation, 306
  thread identifier, 356
  usage, 233
  virtual pointer, 239
communication gate, 241, 299, 305, 309, 357, 366, 383
  allocation, 357
  application of a, 366
  construction, 310
  forwarding identifier, 241–243, 305
    definition, 310
    implementation, 305
    orphanage, 244
    problem with a, 243
    root gate, 260
  migration, 357
communication latency, 21, 180–184
  delivery latency, 182
  header latency, 182
  network latency, 181
  node latency, 181
  packet latency, 182

receiver latency, 182
segment latency, 182
sender latency, 182
trailer latency, 182
communication processor, 44
communication protocol
family, 74, 75, 90
inter-nucleus, 89
object-oriented design, 339–346
problem-oriented, 88
representation-independent, 85
suite, 90
transport protocol, 90
communication system, 15, 72
circuit switching, 78
data security, 73, 75
error correction, 75
error detection, 75
end-to-end arguments, 15
flow control, 73, 77, 79
heterogeneity, 84
heterogeneous network, 85
high-volume data transfer, 174, 343
receiver-initiated, 177, 178
sender-initiated, 175
inter-node communication, 163–185
hardware event, 163
software event, 163
massively parallel systems, 73
message descriptor, 171
header tank, 171
holder abstraction, 171–173
sequel tank, 171
message reassembly, 179
network bandwidth, 24
network data unit, 168
network device driver
functional hierarchy, 164
operating principle, 164
packet switching, 79
performance, 72
piggy-backing, 173
protocol data unit, 168
protocol entity, 171
protocol sequence expressions, 160–163
packet transfer, 161
segment transfer, 162
protocol suite, 17
reassembly, 73, 77
segmentation, 73, 77
sequencing, 73, 81
duplicate suppression, 82
fault tolerance, 82
load balancing, 81
message integrity, 83
service data unit, 168
compute and communicate, 157
communication burst, 78, 157
computation burst, 78, 157
critical section, 146, 166, 274, 289
cross-domain call, 53, 54, 57, 145, 294

data parallelism, 392
deferred class, 268, 269
virtually, 324
design constraints, 20–27
message startup time, 20
never impede performance, 20
design decisions
postponement of, 217
design principles, 27–31
object orientation, 27

program family, 27
dispatcher, 290, 321
distributed-memory systems, 10
dual class, 228, 307, 400
annotated C++ class, 228
class generator, 228, 307
clerk abstraction, 228
client abstraction, 228
server abstraction, 228
software compatibility, 307, 309
constructor
address parameter, 229
signature, 229
inheritance, 228
interface specification, 228
likeness class customization, 231
dual object, 220, 224–228, 307, 308, 400
a programming model, 231
ROI, 231
virtual processor, 231
clerk, 60, 226, 309, 400
an active object, 226
logical location, 229
typed processor, 226, 230
distributed process, 400–405
horizontal consistency, 227
likeness, 224, 400
a prototype capability, 226, 401
creation, 229
customization, 227
deletion, 229
operations on a, 225
proxy object, 224
replication of, 226–227
server reference, 224, 229
systemwide unique identifier, 231
load balancing, 230
operating-system family, 227
placement, 229–231, 403
prototype, 224, 400
collocation, 229
deletion, 229
instantiation request, 229
invocation of, 226
mobile object, 224
operations on a, 225
sharing, 230
runtime system, 404
vertical consistency, 226
dual processor node, 369
dynamic binding, 414
dynamic reconfiguration, 102
active object model, 200–201
bootstrapping, 217–219
boot block, 219
cold startup, 219
first-level, 218
incremental, 217
master node, 217–219
processor bank, 219
root node, 217
slave node, 217, 219
warm startup, 219
coarse-grain restructuring, 202
component replacement, 46
configuration decision, 227
extend-on-use, 201, 215–217
clerk binding, 216
entity fault, 216
proxy call, 216
proxy call repeat, 216
service clerk, 216

service execution, 216
service installation, 216
service proxy, 215
fine-grain restructuring, 202
forwarding identifier, 208
garbage collection, 26, 205
initiating withdrawal, 205
ROI layer, 206
scope of service, 206
third party, 205
incremental bootstrapping, 26, 199, 204
incremental loading, 26, 51, 199, 204
client, 204
required services, 218
user transparency, 208
kernel and POSE, 201, 211–214
administrator, 213–214
client stub, 212
manager, 213
porter, 212
medium-grain restructuring, 203
naming fault, 207
nucleus, 201, 209–211, 298
browser, 211
joint, 209, 211
jump table, 209, 211
jump table index, 209
trap, 209
trap handler, 209
trigger object, 211
plumber, 214
ROI exception, 207
communication exception, 208, 216
naming exception, 208, 216
service access point, 208, 211
interface specification, 214
ROI-capable, 214
service replacement
on the fly, 358
transient object, 27, 199
user orientation, 199

entity, 49, 355

featherweight process, 18, 68, 111
femto-kernel, 419
Fortran, 10
forwarding identifier, 383
function parallelism, 392
functional hierarchy, 49, 56, 158

gang scheduling, 24
generic class, 330
global unique identifier, 238

hardware exception
asynchronous system trap, 282
simulation of an, 282
handler, 277, 286
active object model, 287–291
passive object model, 286–287
interrupt, 276
lifter routine, 278, 280–282
linkage descriptor, 279, 286, 291, 296
generation, 284
page fault handler, 287
points library, 284
propagation, 286
stack frame, 282, 287, 289, 296
trap, 276
vector table, 277, 279, 281, 284, 291

entries, 284
wrapper routine, 277, 280–283, 296
service-function specific, 296
hash fault, 295
hash item, 299, 301, 305
first-level, 301, 303
lookup of a, 303
Nth-level, 301
structure, 301
unique identifier, 301
hash key, 383
hash list, 267, 301–303
hash table, 299, 302, 382, 383
access methods, 303
allocation strategy, 303
ingredients, 302
initialization, 302
instantiation, 305
of VICE, 297
specialization, 294
heavyweight process, 355, 400
horizontal isolation, 59, 96, 99
host interconnection, 120
asymmetrical coupling, 122
symmetrical coupling, 124
host/node programming, 86, 121

idle loop, 157, 184, 362
incremental loading, 51
inheritance, 35
inline functions, 271, 319
code-size explosion, 271
instance class, 293
interrupt, 276
interrupt handling, 142
activation record, 144
activity notification, 270
activity object, 268, 291, 293
activity propagation, 270
asynchronous events, 268
coroutine switch, 288
critical section, 146, 166
device driver
case study, 291–293
programming, 286
epilogue, 266, 268, 293
event serialization, 148
activity notification, 149
activity propagation, 149
activity schedule, 149
epilogue, 166, 191
prologue, 165, 184, 191
event synchronization, 145
epilogue, 151
guard, 153
hardware lock, 146
monitor, 146
prologue, 151
semaphore, 146
software lock, 148, 156
handler continuations, 276
integrating busy waiting, 156
interrupt transparency, 155
locking delay, 154–156
preemption point, 156
nonvolatile processor state, 143, 187, 282, 283
nonvolatile register set, 143, 144, 188, 283, 288, 289
prologue, 266, 268, 291
prologue queue, 292
return from, 289
serving an interrupt, 144
synchronization, 268, 275, 291
locking discipline, 270

volatile register set, 144, 277, 282
wrapper routine, 144, 145
interrupt latency, 194

kernel
entity, 49, 101
family, 47, 52, 66, 101, 283, 299, 340
nucleus members, 302, 324
library, 100
stub, 101
stub library, 101

latency hiding, 23, 43–44
lightweight process, 18, 42, 44, 98, 111, 313, 317, 355, 400
creation/deletion, 313
kernel level, 309
load balancing, 26, 230
local unique identifier (LUI), 232–236, 299
ambiguity of a, 237
decoding, 235
logical nucleus address, 233
rehashing, 233
hash fault, 234
stopgap, 234
structure, 232
structure of, 235–236
uniqueness of a, 236
used as a hash key, 232
lock variable, 272
epilogue lock, 275
processor lock, 275

MANNA
asymmetric multiprocessing, 134–137
application thread, 135
communication thread, 135
dual processor node, 127
application processor, 127
communication processor, 127
message-passing coprocessor, 129–134
symmetric multiprocessing, 137–139
message buffer descriptor, 337
header tank, 337
sequel tank, 343, 344
message delivery, 333
message latency, 21, 88, 142, 388, 418
hiding of, 23
message startup time, 8, 20–25, 35, 88, 388, 418
average number of instructions, 11
impact of multitasking, 21
loss of processor cycles, 20
operating-system level, 11
the challenge, 23
message stream, 393
microkernel, 2, 3, 6, 8, 26, 115, 411
family-based design, 412
functions, 410
performance problem, 418
microkernel architecture, 23, 62, 95
microkernel family, 66, 67, 95, 107, 413, 418
multi-user/multitasking, 72
multi-user/single-tasking, 69–71
kernel isolation, 69
network integrity, 70
object-oriented design, 347

single-user/multitasking, 71–72
task isolation, 72
task scheduling, 71
single-user/single-tasking, 67–69
network communication, 68
nucleus separation, 69
thread scheduling, 68
microkernel technology, 409
drawbacks, 409
minimal basis
execution model, 354
minimal system extension
employing subclassing, 327
kernel level
case study, 327–331
mobile object, 366
monitor, 146
monolith, 9, 25, 115
multiple inclusion, 266
multiple-instruction/multiple-data, 2
multitasking, 3, 9, 21, 46, 59, 96, 111, 134, 183, 408, 418
overhead, 12, 65
multithreaded architecture, 347
multithreaded entities, 49
mutual exclusion, 272

naming, 229, 244–248
access transparency, 246
domain clerk, 258, 259
addressing, 260, 261
forwarding identifier, 260
root gate, 260
domain server, 262
environment variable, 257
location transparency, 246
name domain, 247, 249
construction, 252
inter-domain link, 255
intra-domain link, 250
name mappings, 249
privacy, 252
name plane, 247–249
registration fault, 252
sharing, 252
name porter, 250
name
resolution, 250–251, 256–257
name scope, 253
name server, 246, 249, 258
initialization, 258
per-process binding, 310
shared by processes, 252
name space, 244
avoiding name clashes, 249
construction, 254
entry point, 218
initial system, 218
multi-user environment, 254–257
partitioning, 248
single-user environment, 254
operating-system family, 245
replication transparency, 247
scaling transparency, 248
search space
selection, 257
security, 257
symbolic identifier, 244, 249
significance, 246
systemwide unique identifier, 249, 251, 252
naming exception, 251

nano-kernel, 413
native operating system, 108
network data unit, 168, 335, 338
  design of a, 337
network latency, 21, 44
node latency, 21
nucleus family, 66

object addressing, 299
object invocation layer, 60
object invocation schemes
  local (LOI), 57, 214, 225, 294
  nearby (NOI), 57, 131, 214, 294
    cross-domain call, 57
  remote (ROI), 58, 214, 225, 229, 366
    access transparency, 246
  transitivity, 59, 213
object orientation, 3, 27–31
  active objects, 200
  class hierarchy, 353
  class-based, 271, 416
  deferred class, 279
  dynamic binding, 30, 269, 278, 286
  dynamic typing, 341
  inheritance, 30, 265, 308
  late binding, 30
  multiple inclusion, 266
  multiple inheritance, 30
  passive objects, 200
  polymorphism, 265, 308
  pros and cons, 353
  pure, 266
  single inheritance, 30, 302
  strength of, 330, 350
  subclass, 285
  subclassing, 265
  superclass, 265, 269, 300
object-oriented design, 347
  analogy to program families, 347
  subclassing
    functional emaciation, 347
    functional enrichment, 347
  superclass
    subset of system functions, 347
    superset of system functions, 347
object-oriented operating system, 30
opaque identifier, 239, 299, 301, 334, 341, 343
  strong unique, 240, 301
    overhead, 240
  weak unique, 240, 301
opaque pointer, 287
operating systems
  Amoeba, 34
  Angel, 419
  AX, 32, 33
  Choices, 34, 411
  Chorus, 12, 26, 33, 410
  CMOST, 19
  CS/Tools, 19
  DAS, 33
  FAMOS, 33
  Mach, 7, 8, 23, 26, 33, 34, 410
  MONADS, 419
  MOOSE, 32, 33
  Multics, 101, 217
    trap-on-use, 101, 217
  NX/2, 19
  Opal, 419
  OSF/1 AD, 13, 14, 22, 23
  PANDA, 8, 9, 419

PARIX, 19, 413
PEACE, 4, 23, 31–35, 411
evolution, 416
objective, 34
Psyche, 8
PUMA, 22, 23
QNX, 33
SunOS 4.1, 9
Thoth, 33
Topaz, 8
UBIK, 19
UNIX, 2, 32, 33, 119, 319, 409–411, 413
BSD branch, 410
Version 6, 410
V, 34, 64, 249
WINDOWS-NT, 2, 410
operating-mode variety, 227
operating-system construction
actual structure, 59
building blocks, 107
kernel, 108, 110
nucleus, 108
POSE, 108
design rule, 53
guest level, 117, 122, 124, 125
horizontal isolation, 59, 96
hybrid structure, 60
macrostructure, 37, 47–56
building blocks, 48
design hints, 48
functional hierarchy, 49
maxim, 47
microstructure, 37, 65–72
minimal basis, 39–44
but perfect, 45
minimal system extensions, 45
primary problem, 37
site-dependent components, 108
site-independent components, 108
stepwise refinement, 37
uses relation, 56, 60, 65, 96, 99, 189, 248
meaning of "uses", 14, 46, 319
vertical isolation, 56, 69, 96
operating-system family, 47, 354, 411, 414
abstract machine, 28
hiding system internals, 354
incremental system design, 28
minimal basis, 38, 65
but perfect, 38
minimal but perfect basis, 201
minimal system extensions, 38
role of application, 38
system specializations, 28
virtual machines, 38
own variables, 277, 313

parallel computers
use of, 408
parallel machine
hosted system, 85
parallel operating system
design principle, 408
partial interpretation, 99
passive object, 49, 58, 88, 285
trap handling, 285
PEACE
bootstrapping, 95
building blocks, 407, 408
class library, 353
CLUB, 91, 92, 158, 163, 164, 167, 170, 184, 335, 337,

338
abstract network device, 92
dynamic binding, 93
meaning of, 92, 158
COSY, 90, 93, 95, 124, 158, 159, 164, 169, 173, 291, 335, 337, 339, 346, 370, 390, 391
coexisting entities of, 93
meaning of, 90, 158
transparency, 90
user channel, 93, 94
virtual channel, 91
functional hierarchy, 56
guest level, 88, 117, 122, 124, 125
history of, 31–35
kernel, 49, 108, 110
docking station, 409
functions, 49
hybrid structure, 60, 62
meaning of, 20, 32, 104, 417
multi-address-space configuration, 61
NICE, 90, 91, 95, 129, 130, 132, 158, 160, 163, 169, 173, 176, 178, 335, 337–339, 341, 344, 345, 390, 391
meaning of, 90, 158, 339
nucleus, 49, 88, 108, 126, 408
communication functions, 339
functions, 49, 409
interface, 377
locally operating, 332–334
microstructure, 65–72
NC^2, 89, 123, 126, 129, 132, 138, 172
NICE upcall handler, 339
remotely operating, 335–338
software backplane, 409
OIL, 60
meaning of, 60
`peace()`, 312, 316
POSE, 49, 58, 108, 200
functions, 49, 409, 420
meaning of, 107
parallel service-processing, 204
warehouse, 409
process model, 355
unit of comradeship, 356
unit of distribution, 49, 355
unit of execution, 49, 355
unit of preemption, 355
single-address-space configuration, 60
VICE, 55, 58, 60, 101, 102, 294
meaning of, 55
pico-kernel, 8, 419
functions, 8
preemption point, 156, 184, 276, 383
process control block, 224, 241, 257, 319, 358, 378
process gate, 305, 309, 310, 312
process model
bundle, 68, 312, 355, 362, 375
comrade thread, 356, 362
entity, 49
kernel process, 319
league, 357
multithreaded server, 363
team, 42, 71, 312, 324, 356
built by comrade threads, 356

substitute thread, 312
thread basket, 312
thread bundle, 312
thread, 42
user process, 319
processor bank, 117, 219
program family, 3, 27–28, 268, 411
concept, 346
implementation, 346
minimal basis, 27
minimal system extensions, 27
protection domain, 99
protocol data unit, 130, 168, 337, 360, 395
protocol sequence expression, 161
pseudo process, 65

queuing discipline
FIFO, 267, 268
implementation of, 268
LIFO, 267, 268

reactive object, 186, 192
system integrity, 194
remote object invocation
dynamic reconfiguration, 212, 213
incremental loading, 200
remote procedure call, 45, 216, 368
access transparency, 246
lightweight, 214
lightweight one, 55
marshaling/unmarshaling, 221, 368
object-based, 216
server address, 221–222
stub generator, 222

sandwich, 294
scaling transparency, 18
scheduler, 290, 322
customization of the, 328
invocation, 362
scheduling
domain of, 312
message-driven, 322
of threads/bundles, 293
preemptive, 312, 357
scheduling domain, 68
scheduling unit, 68
semaphore, 146
service
implementation, 220
server, 220
addresses, 221
private state, 223, 224
public state, 223, 224
service access point
interface specification, 214
of nucleus, 294
service data unit, 168, 395
shared-memory multiprocessor systems, 6, 9
single-program/multiple-data, 16, 373, 392
single-tasking, 8, 21, 59, 111, 183, 194
smart pointer, 308
software backplane, 377, 409
cornerstone, 390
fast path, 390
interface classes
design of, 381
thread communication, 377
thread localization, 377, 381
thread scheduling, 377
software compatibility, 221
sparse capability, 237
spin lock, 270, 272

static configuration, 101
stub, 54
  of the nucleus, 54
  of the user, 55
stub generator, 55
stub routine, 54
supervisor mode, 18, 51
symbolic identifier, 244, 249
  significance, 246
synchronization send, 344
system call trap, 294
system configuration
  functional dedication, 117
  generic node, 109, 115–116
    microkernel, 115
    monolith, 115
  guest level, 219
  logical node, 125
  message-passing coprocessor, 129
  system node, 109, 113–115
    device server, 114
  user node, 109–113
    multiple address spaces, 111
    multitasking, 111
    single address space, 111
    single-tasking, 111
systemwide unique identifier, 231, 299, 301, 307, 357
  location-dependent value, 243
  representation, 243
  size of a, 239
  structure, 301
  tradeoff, 232

task, 355
task scheduling, 71
team, 42, 71
team control block, 324
thread, 42, 355
  first-class citizen, 358
  lightweightness, 324
  remote creation, 400
  synchronization, 358
    event consumption, 358
    event counting, 358
    event definition, 358
    event propagation, 358
    one-to-many, 359
    preemption, 359
    timeout, 358
    unique bit pattern, 358
  target of message transactions, 356
threads package, 316–318
timeout handling, 293
timesharing, 9
transfer ticket, 344
transient object, 27, 281
transparency, 3
  access, 110, 246
  interrupt, 155
  location, 246
  replication, 247
  scaling, 40, 111, 248
trap, 8, 53, 98, 276
trap handling
  nucleus trap, 295
    carrier object, 294
    stub routine, 295
  trap handler
    case study, 294–299
trap instruction, 98, 296
trap-on-use, 101, 217

user mode, 18, 51
user partitions, 69
uses relation, 65, 101, 189, 248, 330

sandwich, 55, 60

vertical isolation, 56, 96, 99, 319, 323
virtual channel, 79, 91, 392
connection identifier, 395
data aggregation, 393
hash table, 398
network transparency, 393
number, 395
reader channel, 396
receiver buffer cache, 398
user-level class library, 393
writer channel, 395, 397
virtual function, 269, 270
dynamic binding, 269
pure, 268
virtual machine, 320
virtual memory, 407
user acceptance, 13
virtual processor, 8, 40
virtual shared memory, 10, 13, 133, 407, 420
preventing thrashing, 408
compiler support, 408
data partitioning, 408
von Neumann bottleneck, 10, 13