# Hierarchical Schedulers in the PM System-Architecture

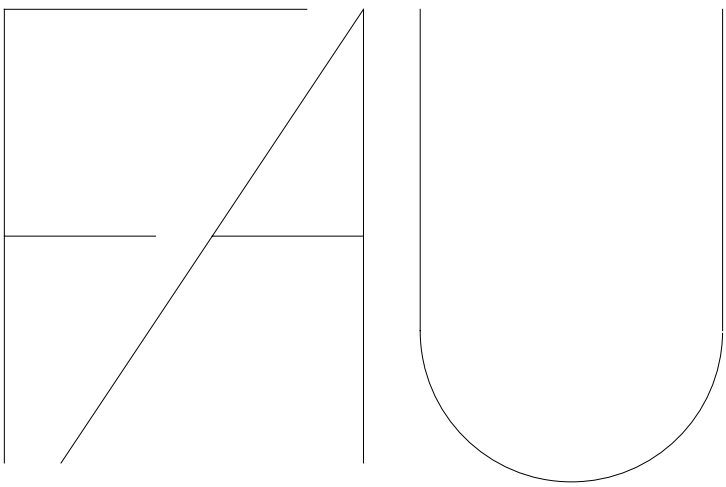
Jürgen Kleinöder, Thomas Riechmann

June 1994 TR-I4-94-16

Technical Report

Institut für
Mathematische Maschinen
und Datenverarbeitung
der
Friedrich-Alexander-Universität
Erlangen-Nürnberg

Lehrstuhl für Informatik IV
(Betriebssysteme)



# Hierarchical Schedulers in the PM System Architecture

— Extended Abstract —

Jürgen Kleinöder, Thomas Riechmann

University of Erlangen-Nürnberg
Department of Computer Science IV
Martensstr. 1, D-91058 Erlangen, Germany
{kleinoeder, riechmann}@informatik.uni-erlangen.de

## Abstract

Today's computer and network architectures provide means for parallel execution of computing intensive applications. The primary operating system abstraction for supporting parallelization of applications is the thread. However, current implementations have severe deficiencies: kernel threads have too much overhead, and user-level threads are not integrated well enough into the operating system. Furthermore, an application may gain an unfair advantage from the scheduler by forking many threads. In this paper, we propose a hierarchically structured scheduling system. With a homogeneous integration into an open operating system architecture, the disadvantages of other thread concepts are avoided. In addition, a distribution of computing time among all applications and application subsystems is easily possible, independent of the number of threads.

## 1 Introduction

With the introduction of distributed systems and multiprocessors, more requirements need to be met by operating systems these days. First of all, such modern hardware architectures are much more complex to manage as the older monoprocessor systems. With the greater amount of processing power available, new and more complex types of applications are used, which in turn expect more features from the operating system. This created a need for operating systems that are adaptable to different hardware and different types of applications. Examples of components that need to be flexible, depending on the application, include security mechanisms and memory and thread management, especially in parallel architectures.

The possibility of adapting or extending these features of an operating system greatly depends on how well it is modularized and how the interfaces to these modules are defined. For example, traditional monolithic kernels provide little, if any, mechanisms that the user can alter, whereas a microkernel with external servers already offers more flexibility. However this does not apply to the microkernel itself but only to external mechanisms. As these are structured into coarse-grain servers the adaption to specific applications usually requires the substitution or addition of a whole server.

A typical example of the lack of flexibility in a microkernel system is the thread management. There are two different strategies: Using so-called kernel threads, the microkernel does the management. Creating threads and switching between them is expensive, and the application cannot control any aspect of the scheduling strategy, since it is included in the microkernel. Alternatively, the thread management and scheduling strategies can be implemented in the application, which makes thread switching fast. However, the missing integration of such user-level threads into the operating system also causes problems (see sec. 3.2).

There is an additional problem that arises when an application creates a large number of threads. The computing time is allocated globally, or at least within the application, not only according to priorities, but also according to the number of threads. Applications or parts of applications that create more threads also get a larger time allocated for them, which bypasses the scheduling strategy.

This paper presents a concept for a homogeneous integration of an user-level thread management into an operating system. The flexibility in the application level is not compromised in any way. The basis for this system is a fine-granular modularization of the appropriate components of the operating system and application interfaces provided by those components. Using these interfaces, the thread management can be adapted to each application but remains completely integrated into the operating system. Creating a scheduler hierarchy can also avoid the problem of unequal distribution of computing time among applications that use different degrees of parallelization.

The work described in this paper is part of the PM research project[1]. Chapter 2 gives a brief overview of the structure and goals of this project, which serve as a basis for a fine-granular operating system with open interfaces. Chapter 3 then describes the structure of the thread management in the PM architecture and compares it to other thread management concepts. Chapter 4 presents the current work in progress and application support.

# 2 PM: A Distributed Object-oriented Operating System

Goal of the PM project is the implementation of a distributed object-oriented operating system. Objects are the basic units that are supported by this operating system, which is object-oriented as well. The main goal is to make the adaption to requirements of different classes in regular applications possible.

The PM project consists of two major parts: the *object model* and the *system architecture*. The PM object model defines the distributed, object-oriented language that contains abstractions such as distribution, concurrency and protection. The PM system architecture (*PM/SA*) uses the object model to implement the components of the operating and run-time system. As the objects that implement the operating system services and those objects that use these features are implemented in the same object model, the interfaces are compatible. Given sufficient rights, it is possible that every application object interacts with any kernel object. Obviously, the application

1. The PM project is funded by the Deutsche Forschungsgesellschaft (DFG) as part of the grant SFB 182/ TP B2.

has to have some sort of reference to the operating system objects. Similar mechanisms for an object-oriented interaction between the operating system and applications can be found in Choices ([CaI93], [CIR+93]).

Since the operating system is not a monolithic block with a fixed interface, an application can build operating system mechanisms on top of existing abstractions. This is done by implementing objects that implement operating system functionality for other objects by communicating with operating system objects. This creates relationships at the implementation level, beyond the normal client/server relationship of caller and callee. This corresponds to the object/meta-object relationship in Apertos [Yok92].

The assignment of operating system objects to virtual address spaces is not set beforehand by a division into the microkernel, operating system server, and applications, but can be done on each node according to the security and efficiency requirements.

Virtual address spaces are called object spaces. Objects can be mapped into several object spaces. Objects from the operating systems can be placed in the application object space, and application objects can be mapped into an operating system object space providing supervisor state. Method calls across object borders are more expensive than local method calls. However, since the association of objects to object spaces is very flexible, the costs can be reduced to a minimum by an appropriate placement of the objects [Kle93].

# 3 Thread Management in the PM System Architecture

## 3.1 Motivation

Many applications require a mechanism that allows the programming of concurrent execution of several threads within one application. Aside from the better structure in the program, one can use a multiprocessor system to execute these threads in parallel. In order to optimally use the available processors, one obviously needs at least as many threads as processors. Since it may not always be possible to divide an algorithm into any number of equal parts (e.g. because the programming would be too complicated), it seems reasonable to just create any number of threads and leave the task of parallelization up to the operating system. Important is a very high efficiency in the thread creation and deletion, as well as thread scheduling.

Thread support can be found in many operating systems these days. There are two basic concepts: kernel threads (the thread management structures are in the kernel) and user-level threads (thread management in the user space). Both alternatives are widespread, but none are completely satisfactory.

A concept that uses the advantages of both methods is presented in the next sections.

## 3.2 Overview of the Current Thread Concepts

Operating systems today support either user-level or kernel threads. Both concepts and their advantages and disadvantages are described below.

| | Kernel-Threads | User-Level-Threads |
|---|---|---|
| Inclusion into the operating system | + easy | – hard |
| Scheduling strategies implemented by the user | – no | + yes |
| Efficiency | – slow | + fast |

**Tab. 3.1 Advantages and disadvantages of kernel and user-level threads**

Obviously, when kernel threads are used, the kernel manages the threads. Switching between threads is not as expensive as switching between traditional UNIX processes, as long as the threads are in the same address space. They are not suited for an application that needs to be massively parallelized, since the creation and deletion of threads and the switching between threads is still too expensive. Because of the complete inclusion into the operating system — every thread can be blocked and unblocked by the kernel — this technique is used in many new operating systems (e.g. Mach [Bla90] or Windows-NT [Cus93]).

User-level threads (e.g. FastThreads [And89] or QuickThreads [Kep93]) are implemented with library functions that are linked to the application. No interaction with the operating system is therefore necessary for thread management. Virtual processors (e.g. in the form of traditional processes or kernel threads) are allocated by the application, which can then be distributed among the user-level threads. The structures used for the management are completely within the application's address space. Creation and deletion of threads, as well as switching between them, is very fast and the scheduling strategy can be implemented by the user.

Since the management of user-level threads is not included in the operating system, the kernel cannot get any information on those threads. If the user-level thread executes some action that will cause it to block (e.g. a blocking system call or a page fault), the kernel will block the whole process (or kernel thread) that included the user-level thread. The application loses the control over the scheduling and suddenly owns less virtual processors than requested. The same problem arises when the kernel preempts a process as part of its scheduling.

There are several proposals for solutions that provide specific interfaces to the kernel for the management of these user-level threads ([And92], [MLSM91]). Communication between kernel and thread management or direct access from the kernel to the user-level thread management is used to get the information in all relevant situations (e.g. the kernel changes the number of virtual processors that are allocated for the application, or the number of virtual processors needed by the application changes). However, the communication and the actions executed by the kernel are almost as expensive as the management of kernel threads. Shared use of the man-

agement structures through the kernel also restricts the flexibility of the application. Finally, all these solutions have the disadvantage that there are still two different classes of threads with different properties.

For massively parallel applications, there is no alternative to the management of threads within the application. Since there is no real border between the kernel and the applications in the PM system, it is well suited for a homogeneous integration of threads into applications as well as the operating system.

## 3.3 Threads and Scheduling in PM

The main disadvantage of user-level threads lies in the fact that the kernel knows nothing about them, just about the virtual processors that it allocated for the application. Within the application there is a second “mini” operating system that assigns the threads to the virtual processors.

We chose to not use a “mini” operating system but make the thread management objects in the application available to the kernel and let the kernel directly interact with them.

Furthermore, the term “kernel” is being substituted by “system object space”, since there is no kernel in PM/SA in the actual sense. The system object space contains objects that are responsible for the management of applications and their object spaces.

The management structures of the application-threads are completely in the application object space. As opposed to the user-level threads, though, there is only one type of thread in the entire system — there are no virtual processors. Threads are bound to their object space. Method calls across objects spaces results in the creation of a new thread in the target object space. System calls are regular method calls to objects in the system object space.

Since PM is an object-oriented operating system, threads and schedulers are also objects (like the meta-objects in Apertos [Yok92]). Again threads are used for execution of the methods of these objects (the recursion ends with threads and a scheduler that are not implemented in the PM object model -- similar to the *Apertos Meta Core*).

In current operating systems, the thread objects for kernel threads are in the kernel (fig. 3.1a). During a thread switch, the executing thread moves from the user address space ① into the kernel space ②, calls a scheduling function ③, and switches to another thread ④. This thread will then move back to its user address space ⑤.

In PM, the switching is done by a separate scheduler thread (a thread that executes methods in a scheduler object) — see fig. 3.1b. The switch from the application thread to the scheduler thread is done either by an explicit transfer from the application thread (e.g. there is a command coded into the method) or is forced by some system objects (e.g. as part of a time interrupt). After switching the object space, the processor context is stored in the application thread object and the context of the system scheduler thread can be restored. If the scheduling in the system space decides to give control to another thread in the application object space, the reverse is done.

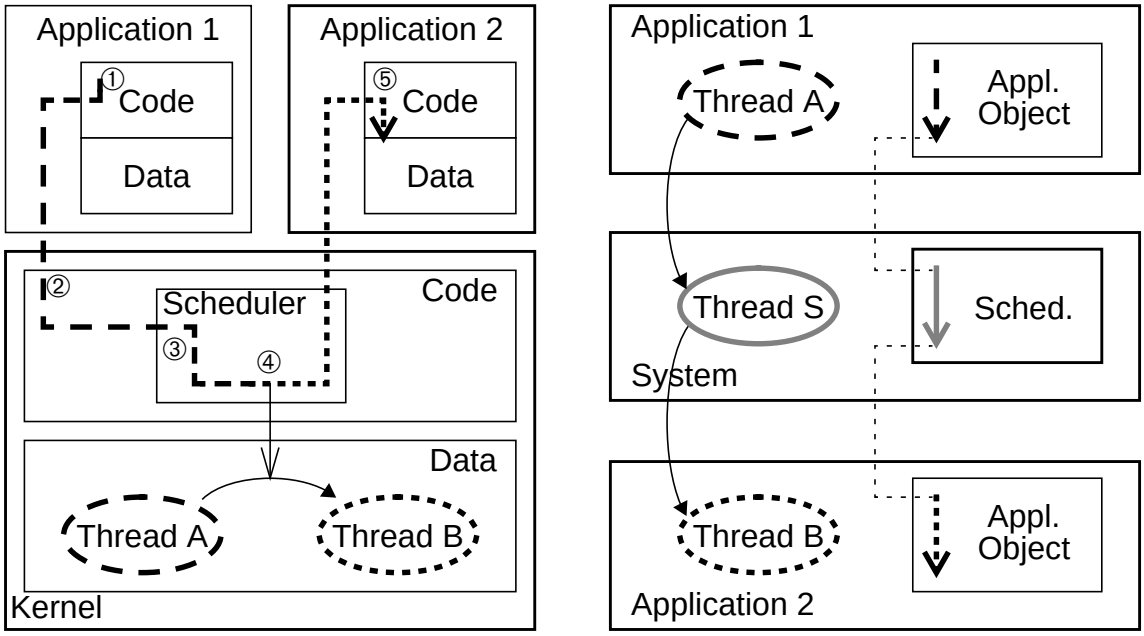



**Fig. 3.1 Thread switching in kernel thread systems (a) and in PM (b)**

The thread state is therefore always completely within the application object space. In order to block a thread, for example, the state of the thread object is altered, regardless if the blocking is caused by operating system objects or the application itself. The difference to current concepts also lies in the mutual interaction between system and application object space.

This method provides the advantages of user-level threads (user-defined scheduling, fast creation and deletion of threads, easy thread switching). In addition, the advantages of kernel threads remain (complete integration into the operating system).

## 3.4 Scheduler Hierarchies

There are several reasons for creating a scheduler hierarchy. First, a scheduler inside an application is needed for user-level scheduling, independent of the system scheduler. Second, it can be very useful to have a separate scheduling strategy for a group of threads inside an application — another scheduler is therefore needed below the application scheduler. Finally, the problem of uneven distribution of computing time can be avoided for applications that create a large number of threads.

Application schedulers (that distribute cpu time to threads within the application) lie in the application object space. Since they are independent objects, there must be a thread that executes the scheduling methods. The central scheduler for the system assigns a processor to this thread like it does to all other threads.

The scheduler object then determines which application thread will get the processor next, saves its current state and activates that thread. In a multiprocessor system, the system scheduler could assign another processor to the scheduler object of the application, which in turn could pass it on to another thread. A parallel execution of threads in an application is therefore also supported. After the control over the processor has been passed from the system scheduler, every action is done completely within the application object space.

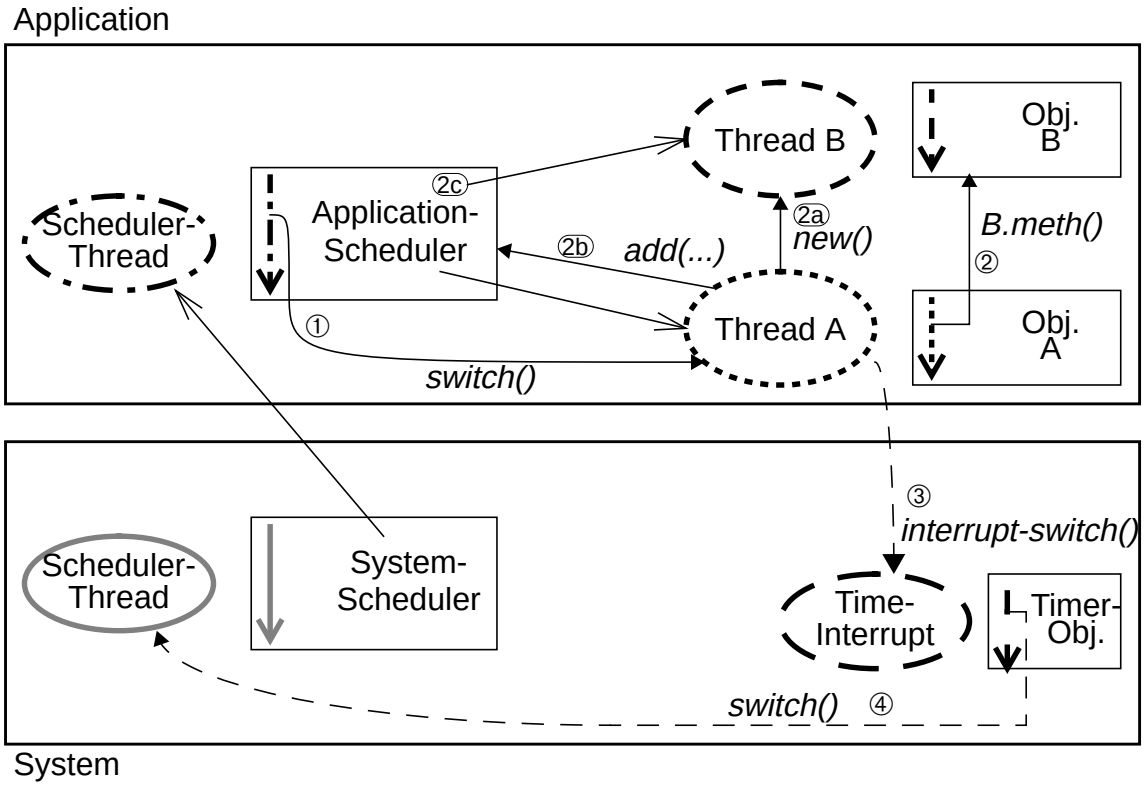



**Fig. 3.2 Scheduler hierarchy with system and application scheduler**

Figure 3.2 shows such a configuration. Every scheduler object has a scheduler thread that executes the methods in that object. The instance variables of the scheduler contain references to the threads to be managed ( ⟶ ). The application scheduler thread is not an abstraction of a virtual processor, but a regular thread. When it activates thread A, it loses the processor ①. New threads are created (e.g. in the context of an asynchronous method call ②) within the application object space ⓶ and references are passed to the scheduler ⓶, ⓶. If the system scheduler wants the processor back after the time slice has expired (in the context of a time-interrupt ③), the processor state is saved into the current thread (e.g. thread A) by the time-interrupt object and control is passed on to the system scheduler thread ④. Notice that thread A has absolutely no choice — just like in normal preemption.

A reactivation by the time-interrupt object can be requested not only by the system scheduler, but by any application scheduler. It is possible that an application wants to use shorter time slices than provided by the system. However, a scheduler can never prevent a time-interrupt activation from a scheduler that lies above it in the hierarchy.

An application scheduler can manage threads that call methods of normal application objects, as well as threads that execute another scheduler. Therefore, the hierarchy can be extended at will. Every scheduler can use its own strategy and its own time-interrupts (similar to the software-interrupts in Psyche [MLS+91]).

Aside from the different strategies, this also provides a method for distributing computing time evenly among different parts of an application. If one part that creates a few number of threads should not be put at a disadvantage to another part with a large number of threads, then a different subscheduler can be instantiated for each. If there are too few processors available, then both parts of the application are served equally and both can pass the computing time onto their threads at will. If there are enough processors, every part will receive a processor for every thread and the highest degree of parallelity is reached.

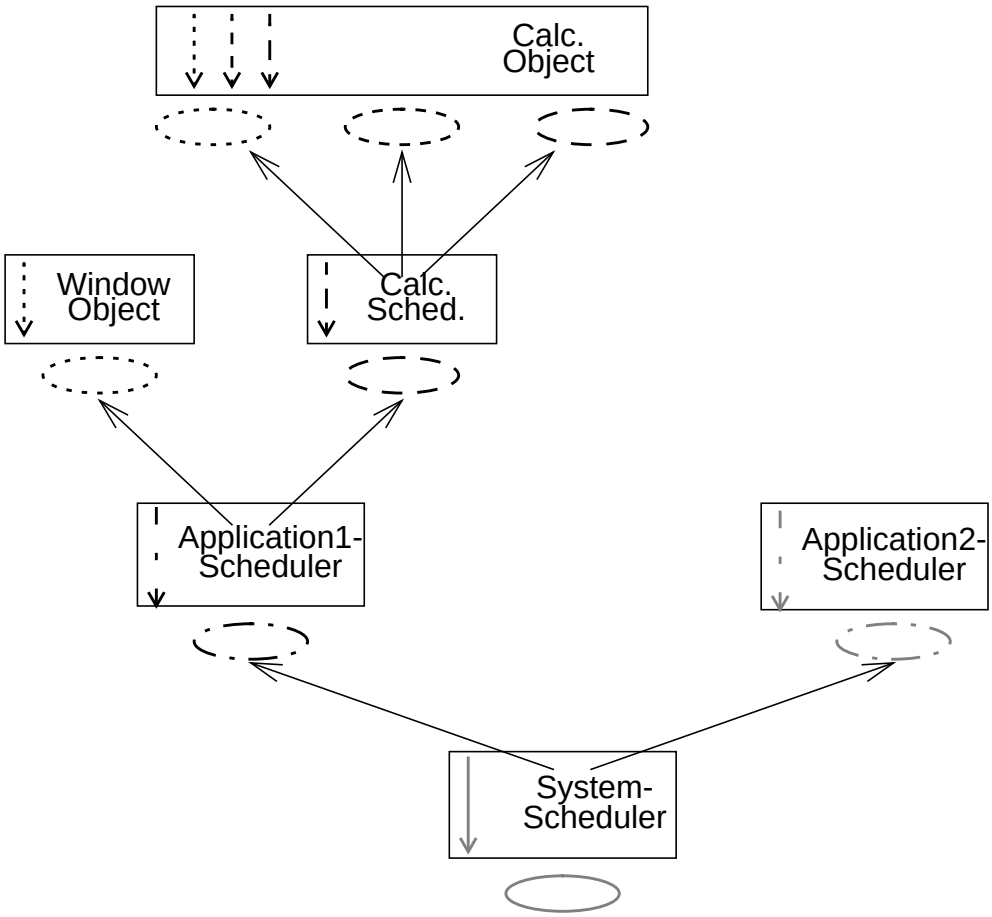



**Fig. 3.3 Scheduler hierarchy within an application**

Fig. 3.3 shows the scheduler hierarchy in an application that displays some intermediate values at a given rate. Every 10 seconds, the application scheduler activates the window object to update the display but allocates the rest of the time for the calculation scheduler. That scheduler, in turn, will distribute the cpu time, using a FIFO or round-robin strategy, to the threads that do the actual calculation.

### 3.5 Method Calls across Object Spaces

In order to use services from operating system or communicate with other applications, PM provides a mechanism to make asynchronous method calls to other objects that lie in different object spaces. It does this by creating a thread in the target address space and adds that thread to the scheduler of that object space.

The final paper will describe this concept more extensively.

## 4 Current Work on PM

A prototype of the design of threads and schedulers were implemented in C++ on a Sequent/ Symmetry multiprocessor system running Dynix/PTX and tested for efficiency. The efficiency of thread creation and deletion, as well as thread switching, were especially noted. A thread switch takes about twice as long as a procedure call. Thread generation and deletion takes about 20 times as long as a procedure call. These factors correspond to the results of fast user-level thread packages [And92]. The costs that arise from the switch between system and application object spaces could not be measured in the current implementation. The overhead for inter-address-space RPC's are published in other articles ([HaK93], [BAL+90]). A good optimization will put the overhead in the range of 10-20 times as long as a procedure call.

The use of different scheduling strategies within one application is made a lot easier by providing the user with a class library of scheduler classes (especially FIFO and round-robin strategies). Obviously, classes can be adapted from any of these library classes when the need arises.

## 5 Conclusion

The performance gain that can be reached by executing parts of applications in parallel is often extremely high. In order to pass this potential to the user, the operating systems must provide adequate abstractions and implement them efficiently. Since the requirements for different applications can be very diverse, it must be easy to modify or add to these mechanisms of the operating system. An architecture for thread management has been presented that avoids the disadvantages of efficiency and lack of integration into the operating system exisiting in current concepts. Application schedulers combine the advantages of kernel threads with the performance of normal user-level threads. Scheduler hierarchies allow an even distribution of computing time to groups of threads, regardless of the number of threads in the group. Different groups can use different scheduling strategies.

The PM kernel concepts are therefore well suited as a basis for fulfilling the requirements for parallelization of different applications on different hardware architectures.

## References


And89 T. Anderson, E. Lazowska, H. Levy: “The Performance Implication of Thread Management Alternatives for Shared-Memory Multiprocessors”, *ACM Transactions on Computers*, Vol. 38, No. 12, pp. 1631 - 1644, Dec. 1989

And92 T. Anderson, B. Bershad, E. Lazowska, H. Levy, “Scheduler Activations: Effective Kernel Support for the User-Level Management of Parallelism”, *ACM Transactions on Computer Systems*, Vol. 10, No. 1, pp. 53 - 79, Feb. 1992.

BAL+90 B. N. Bershad, T. E. Anderson, H. M. Levy, E. D. Lazowska, “Leightweight Remote Procedure Call”, *ACM Transactions on Computer Systems*, Vol. 8, No. 1, pp. 37 - 55, Feb. 1990.

Bla90 David L. Black, “Scheduling Support for Concurrency and Parallelism in the Mach Operating System”, *IEEE Computer*, Vol. 23, No. 5, pp. 35 - 43, May 1990.

CaI93 Roy H. Campbell and Nayeem Islam, "CHOICES: A Parallel Object Orientd Operating System", in Gul Agha, Peter Wegner, and Akinori Yonezawa [Eds.]: *Research Directions in Concurrent Object-Oriented Programming*, MIT Press, Cambridge, Massachusetts, 1993.

CIR+93 Roy H. Campbell, Nayeem Islam, David Raila, and Peter Madany, "Designing and Implementing CHOICES: An Object-Oriented System in C++", *Communications of the ACM*, Vol. 36, No. 9, pp. 117 - 126, Sep. 1993.

Cus93 Helen Custer: *Inside windows NT*, Microsoft Press, Redmond, Washington, 1993.

HaK93 Graham Hamilton, Panos Kougiouris, “The Spring nucleus: A microkernel for objects”, *Proceedings of the USENIX Summer 1993 Conference*, Cincinnati, Jun. 1993.

Kep93 David Keppel: *Tools and Techniques for Building Fast Portable Threads Packages*, Technical Report UWCSE 93-05-06, University of Washington, May 1993.

Kle93 Jürgen Kleinöder, “Object- and memory-management architecture: a concept for open, object-oriented operating systems”; in A. Bode, M. Dal Cin [Eds.]: *Parallel Comp. Architectures: Theory, Hardware, Software, and Appl*; Lecture Notes in Comp. Sci. 732, pp. 150-165; Springer, Berlin et. al., 1993.

MLS+91 Brian D. Marsh, Michael L. Scott, Thomas J. LeBlanc Evangelos P. Markatos, “First-Class User-Level Threads”, *Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles*, pp. 110 - 121, Pacific Grove (CA, USA), published as *ACM Operating Systems Review*, Vol. 25, No. 5, Oct. 1991.

Yok92 Yasuhiko Yokote, “The Apertos Reflective Operating System: The Concepts and Its Implementation”, *OOPSLA ’92 – Conference Proceedings*, pp. 414 - 434, Vancouver (CA), published as *SIGPLAN Notices*, Vol. 27, No. 10, Oct. 1992.