Vorlesung

Programmierung Paralleler Systeme

Wintersemester 1998/99 (10317)

Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

I Inhaltsverzeichnis

A Organisatorisches ____ 1

A.1 Inhalt 2

A.2 Vorlesung, Übung, Schein, Folien, URL 3

B Einführung ____ 1

B.1 Programmierung paralleler Systeme, warum? 1

B.2 Ebenen der Parallelität 2

1 Prozessorebene 3
2 Speicherebene 10
3 Systemebene 13
4 Netzwerkebene 28

B.3 Klassifikation 29

B.4 Programmiermodelle 31

B.5 Lokaler vs. Globaler Adreßraum 38

B.6 Leistungsausbeute 41

1 Grundlagen für Prozessoreffizienz 42
2 Leistungsausbeute durch Parallelisierung 45

© Frank Bellosa, Univ. Erlangen-Nürnberg, IMMD IV, 1999

Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

I Inhaltsverzeichnis

C Optimierung des Speicherzugriffs ______ 49

C.1 Virtuelle Adressierung 49

1 Grundlagen 49
2 Translation Look-Aside Buffer TLB 55

C.2 Caches 58

1 Grundlagen 58
2 Zugriffskonflikte 65

C.3 Ein/Ausgabe. 69

1 "Klassisches" Lesen/Schreiben in Dateien 69
2 Speicherabgebildete Dateien 70

D SUN Enterprise X000 ______ 76

D.1 Systemarchitektur. 77

1 Gigaplane Bus 78
2 CPU/Memory Modul 79
3 I/O Board 81

© Frank Bellosa, Univ. Erlangen-Nürnberg, IMMD IV, 1999

Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

I Inhaltsverzeichnis

E Pthreads ______ 83

E.1 Grundlagen 83

1 Prozeßmodell 84

2 Threadmodell 88

E.2 Basic Pthread Management 94

1 Erzeugung 94

2 Thread-Attribute 95

3 Terminierung 96

4 Rückgabewerte 97

5 Beispiel zur Thread-Erzeugung 98

6 Identifikation 101

7 Initialisierungen 102

8 Threadlokale globale Variablen 104

E.3 Synchronisation. 106

1 Mechanismen zur Thread-Koordination 108

2 Mutex 109

3 Condition-Variablen 113

Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

I Inhaltsverzeichnis

E.4 Parallelisierungsmodelle 121

1 Master-Slave Modell 121
2 Pipeline Modell 122
3 Boss-Worker Modell 123

E.5 Attribute 124

E.6 Thread Scheduling 126

1 Festlegung des Thread-Modells (1 x 1, M x 1, M x N) 126
2 Scheduling Strategie 129
3 Threadpriorität 130
4 Scheduling Parameter 131
5 Vererbung von Scheduling Parametern 132
6 Priority Inversion 133
7 Entkopplung von Threads (detached threads) 137

E.7 UNIX Interaktion 139

1 Bibliotheksaufrufe 140
2 Signale 143
3 Abbruch eines Threads 149
4 Der Cleanup Stack 154
5 Fork/Exec/Exit 160
6 Interprozeß-Synchronisation 166

Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

Inhaltsverzeichnis

E.8 Zusammenfassung 167

F Übung 1 ____ 168

Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

A Organisatorisches

- Dozenten
  - Dr. F. Bellosa, IMMD IV, Frank.Bellosa@informatik.uni-erlangen.de
  - Dr. G. Wellein, RRZE, Gerhard.Wellein@rrze.uni-erlangen.de

- Vorlesung und Übungen
  - für Studierende der Fachrichtung Informatik im Hauptstudium,
    für Studierende der Ingenieur- und Naturwissenschaften im Hauptstudium
  - 4 anrechenbare Semesterwochenstunden
    2 SWS Vorlesung, 2 SWS Übungen

Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

A.1 Inhalt

- Vorlesung
  - Architekturprinzipien und Programmiermodelle von Parallelrechnern
  - Effiziente Nutzung von Monoprozessoren (Cache, Speicher, E/A)
  - Threads und Koordinierung in Multiprozessoren mit gemeinsamem Speicher
  - Architektur und Programmierung von Rechnern mit verteiltem Speicher
  - Architektur und Programmierung von Vektorrechnern

- Übungen
  - Einsatz von Cache-, TLB- und Speicher-angepassten Datenstrukturen
  - Speicherabgebildete Dateien
  - **P**osix Threads (Pthreads)
  - **M**essage **P**assing **I**nterface(MPI)
  - Unterstützung der Vektorisierung durch Programmgestaltung und Direktiven

Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

A.2 Vorlesung, Übung, Schein, Folien, URL

- Vorlesung: 15.3. - 19.3. und 22.3. - 26.3. von 9:00 bis 12:15 im 0.031
- Übungen: 15.3. - 19.3. und 22.3. - 25.3. von 13:00 bis 16:00 im 01.153
- Schein: nach einer mündliche Prüfung am 26.3.
- Folien: werden im WWW zur Verfügung gestellt.
- URL zur Vorlesung:
  - http://www4.informatik.uni-erlangen.de/Lehre/WS98/V_PPS/
  - hier findet man Folien zum Ausdrucken und Zusatzinformationen

Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

B Einführung

B.1 Programmierung paralleler Systeme, warum?

- Nutzung von *mehr* Hardware zur Lösung eines Problems
  - Nutzung brachliegender Ressourcen
  - Anschaffung von spezieller Hardware für das parallele Rechnen
- Potentielle Erweiterbarkeit der Hardware bei wachsender Problemgröße
- Abbildung der nebenläufigen Problemstellung auf eine parallele Lösung (z.B. verteilte Anlagensteuerung, Telefonnetz)
- Verknüpfen bestehender sequentieller Komponenten zu einem parallelen System
- Zuverlässigkeit durch Replikation

Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

B.2 Ebenen der Parallelität

- Prozessorebene
- Speicherebene
- Systemebene, Verbindungsnetz
- Netzwerkebene
- E/A-Ebene (nicht Teil dieser Vorlesung)

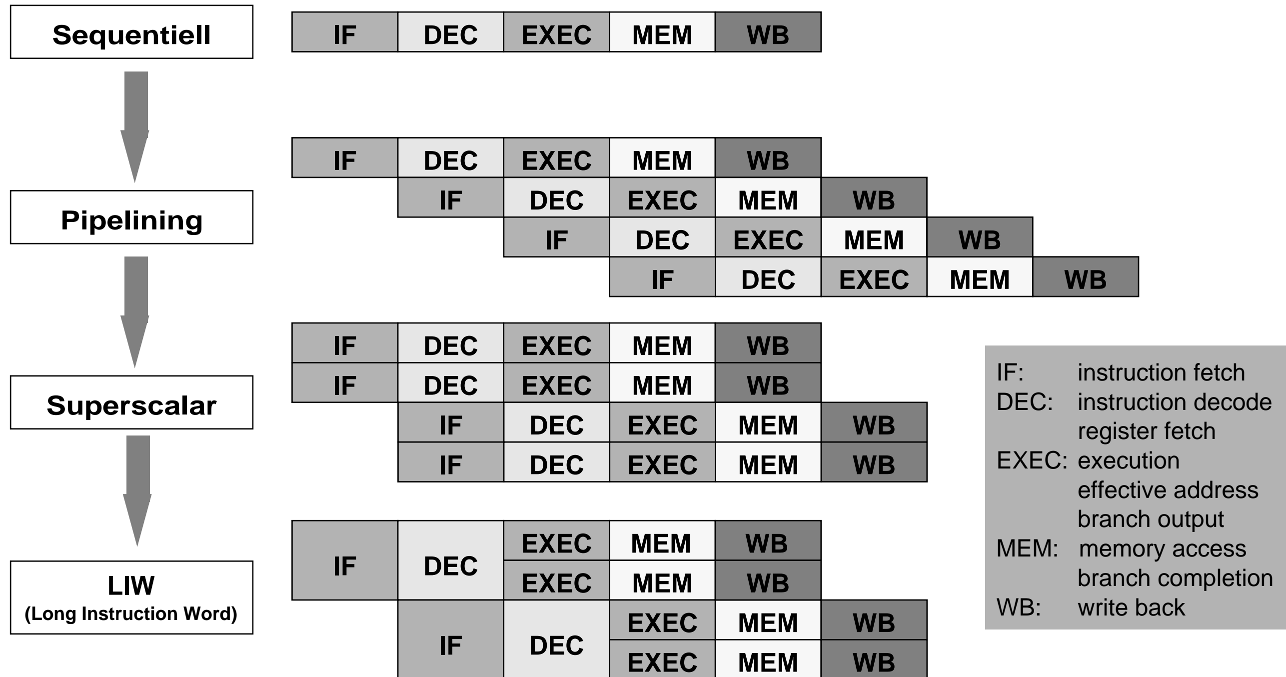
Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

1 Prozessorebene

- Parallelisierung des Befehlsstroms
  - Sequentiell:
    Die einzelnen Phasen einer Befehlsausführung werden sequentiell abgearbeitet. Ein Instruktionsstrom.
  - Pipelining:
    verschiedene Phasen verschiedener Instruktionen werden parallel verarbeitet. Ein Instruktionsstrom.
  - Superscalar:
    die Möglichkeit mehrere Funktionen parallel zu verarbeiten
    z. B. Load / Store : Add / Multiply : Branch
    Paralleler Instruktionsstrom ausgewählter Instruktionstypen.
  - VLIW (Very Large Instruction Word):
    “Breite” Instruktions-Verarbeitung:
    Mehrere Befehle eine parallelen Instruktionsstroms werden bearbeitet.

Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

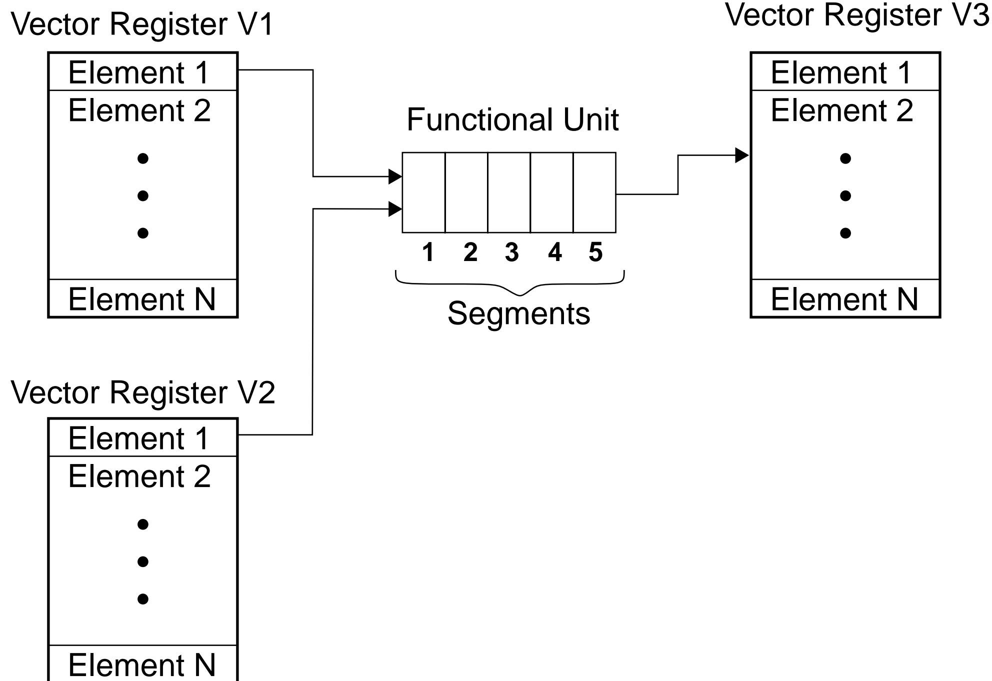
1 Prozessorebene (2)



Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

1 Prozessorebene (3)

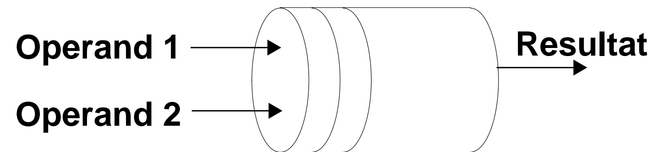
- Parallelisierung des Datenstroms
  - Vektoreinheiten



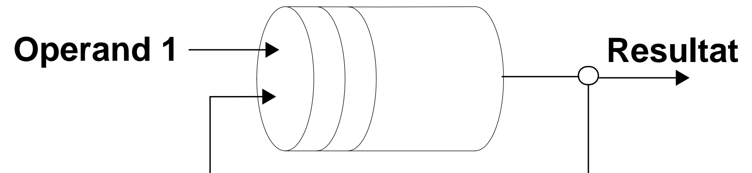
Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

1 Prozessorebene (4)

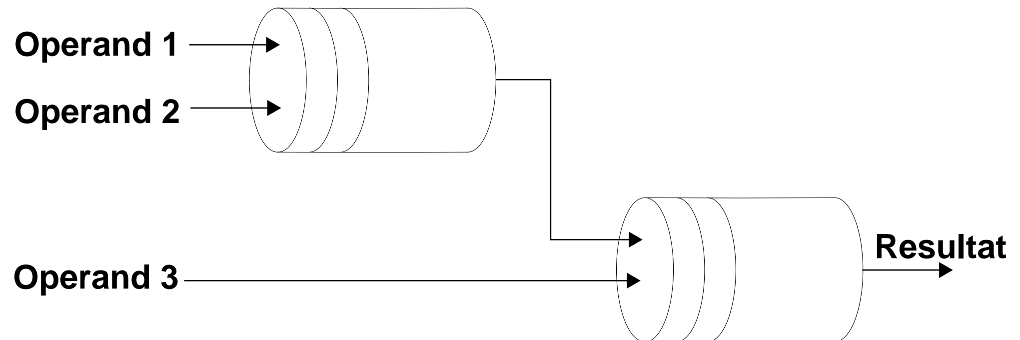
- Vektorverarbeitung in Kombination mit “Instruktionsparallelität”
  - normale Betriebsart



  - Rückkopplung



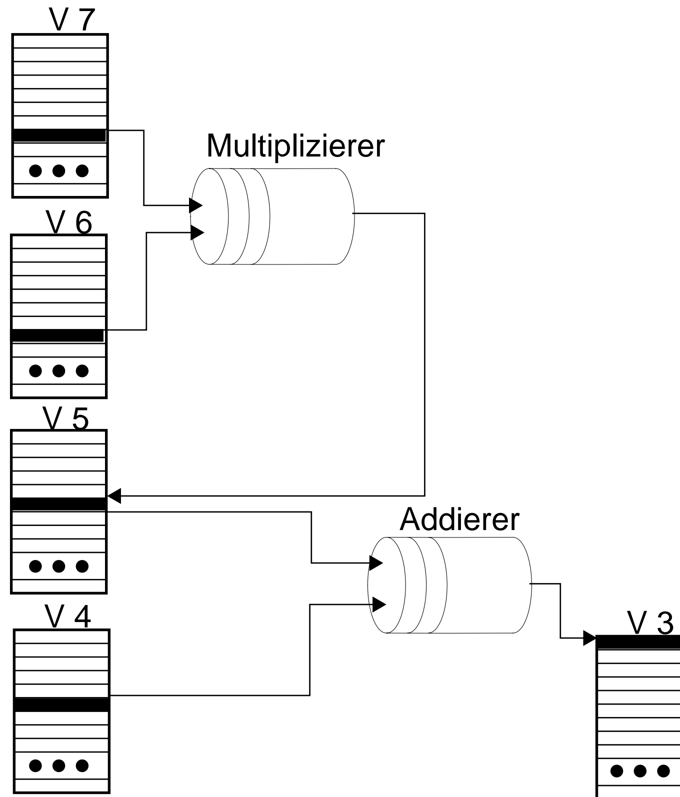
  - Verkettung (Chaining)



Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

1 Prozessorebene (5)

- Chaining zwischen Registern

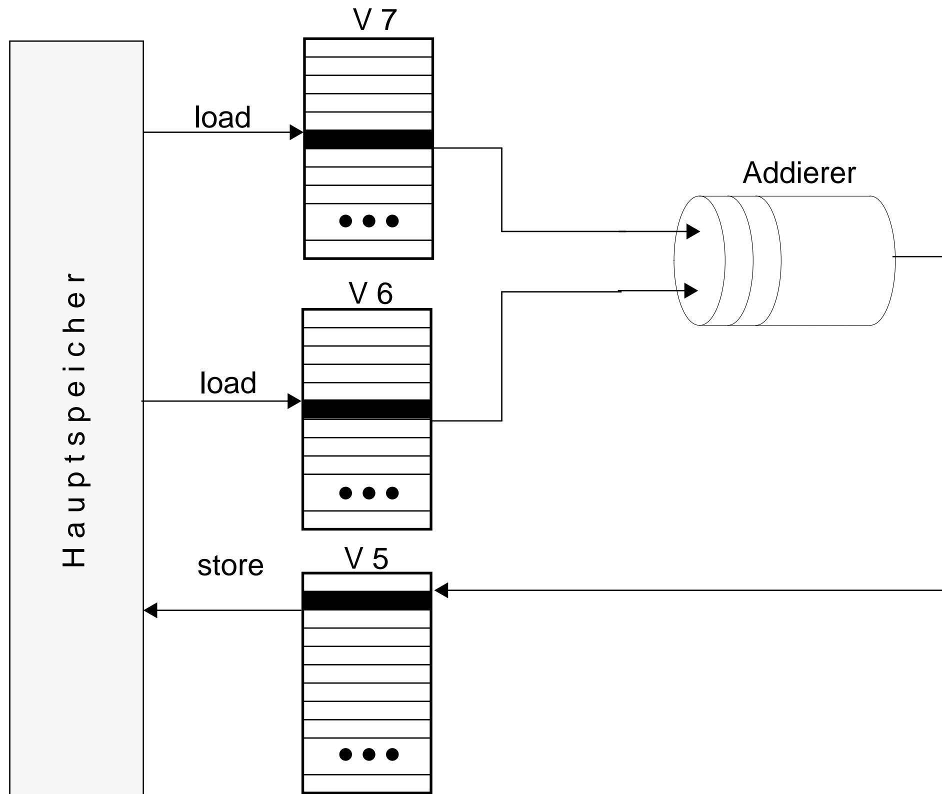


© Frank Bellosa, Univ. Erlangen-Nürnberg, IMMD IV, 1999

Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

1 Prozessorebene (6)

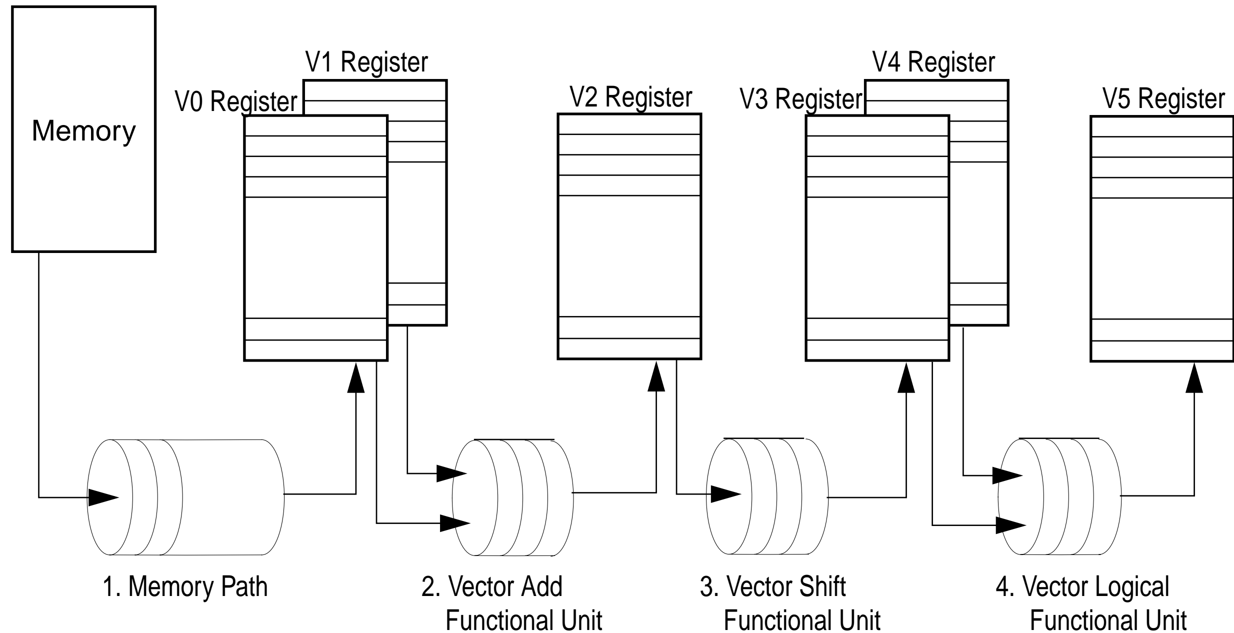
- Chaining zum Speicher



Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

1 Prozessorebene (7)

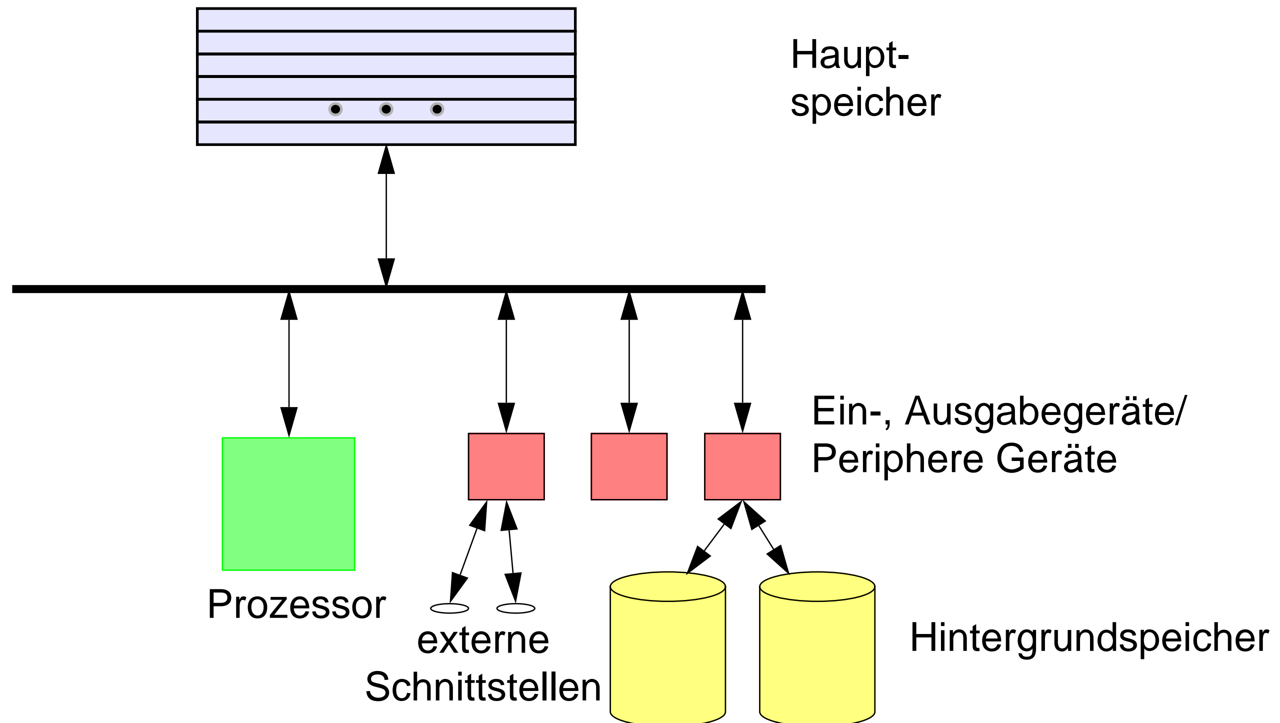
◆ Beispiel:



Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

2 Speicherebene

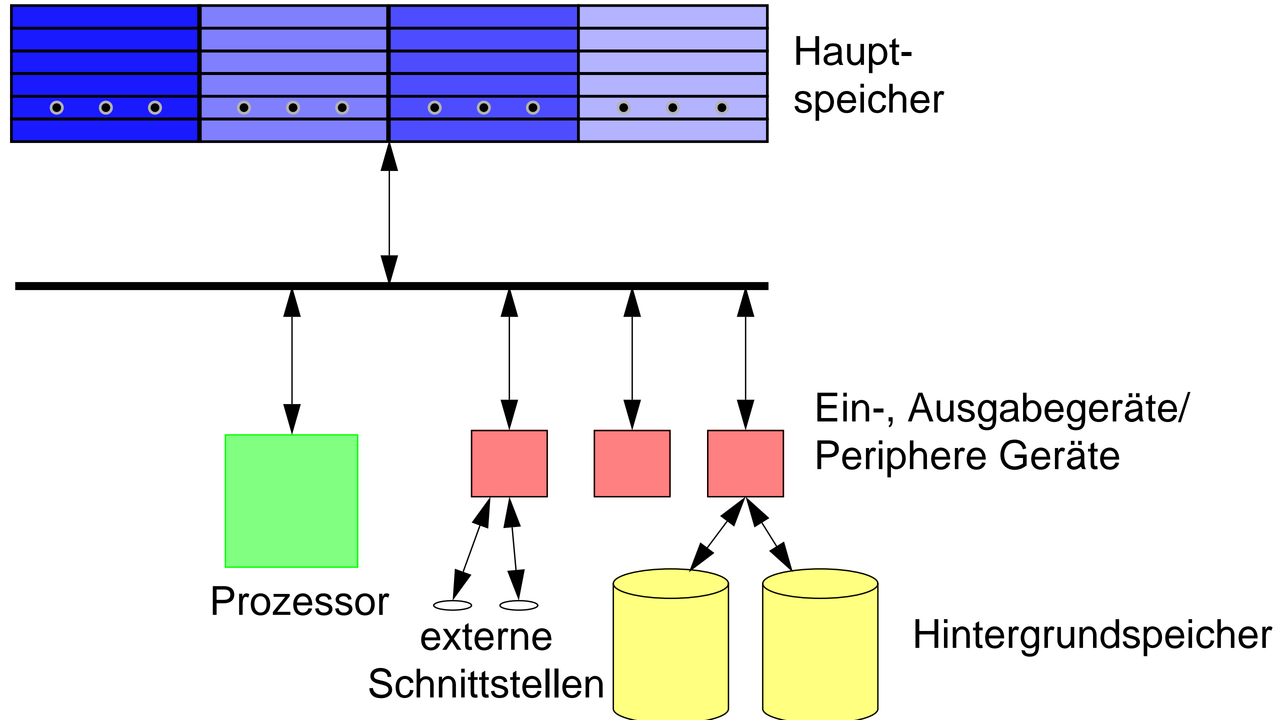
■ Hauptspeicher mit einer Speicherbank



Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

2 Speicherebene (2)

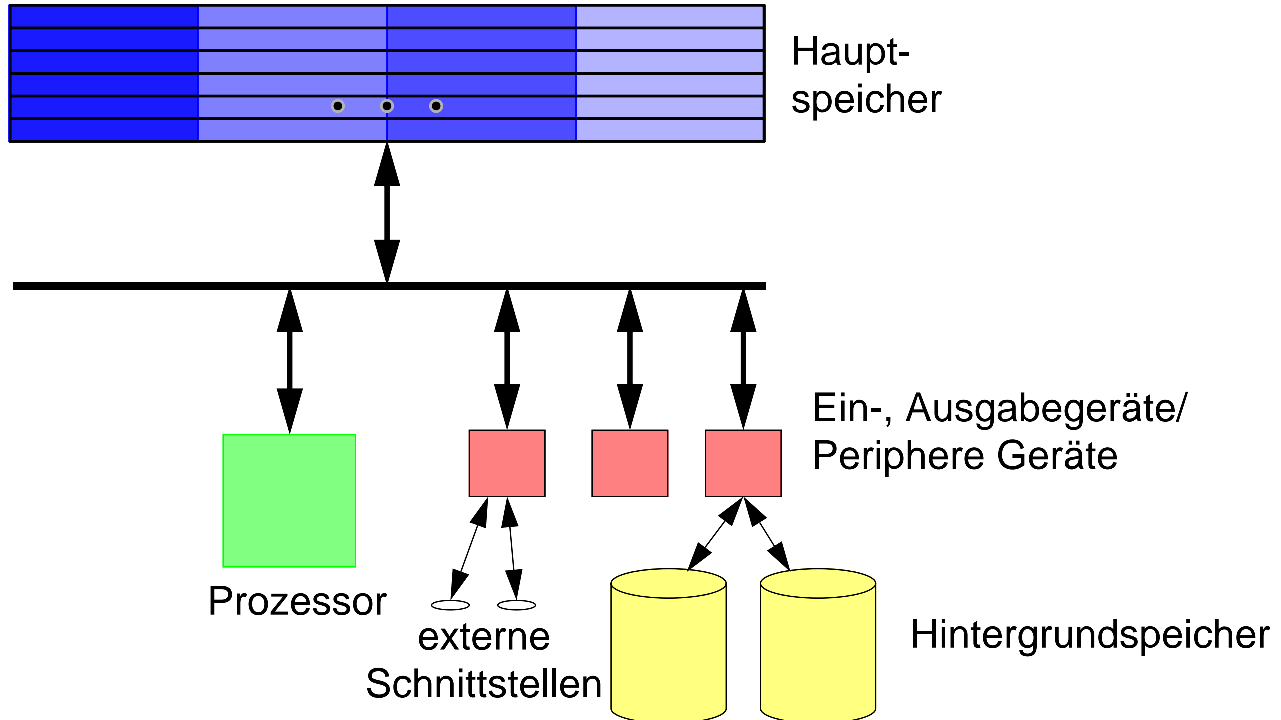
- Parallelisierung des Zugriffs durch Speicherbänke



Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

2 Speicherebene (3)

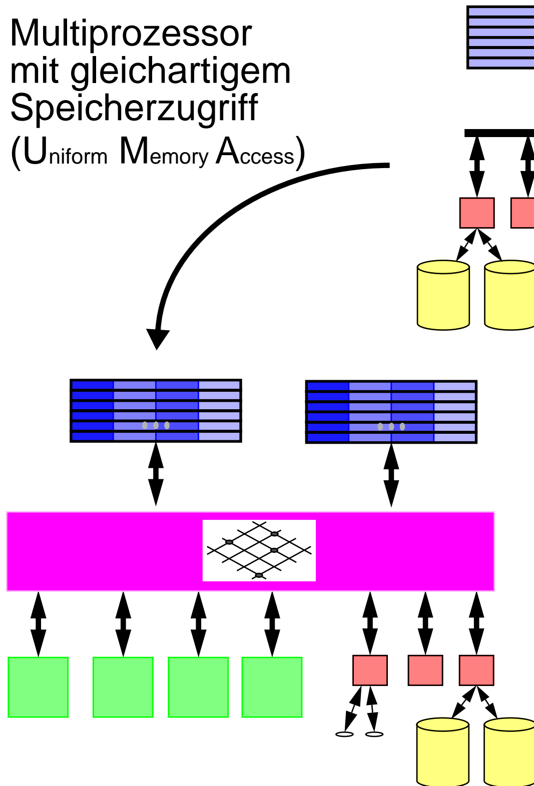
- Parallelisierung des sequentiellen Zugriffs durch Interleaving



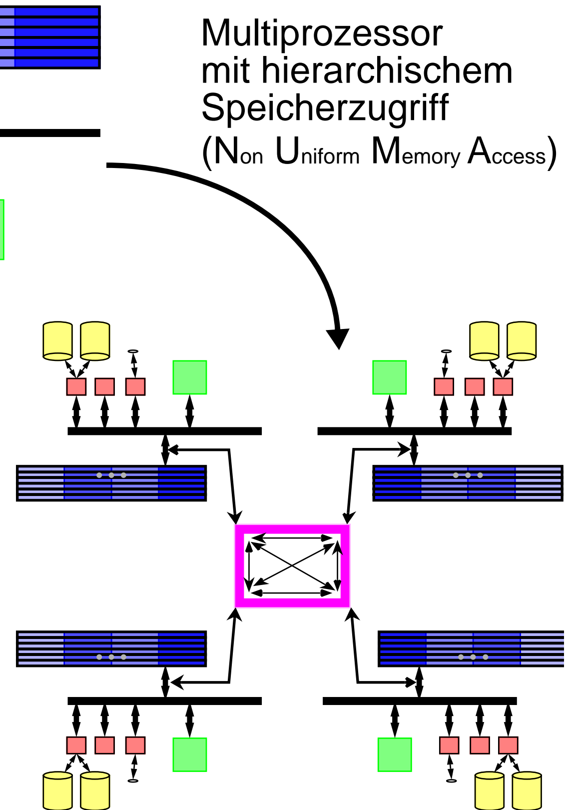
© Frank Bellosa, Univ. Erlangen-Nürnberg, IMMD IV, 1999
Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

■ Multiprozessoren mit globalem Adreßraum

Multiprozessor
mit gleichartigem
Speicherzugriff
(Uniform Memory Access)



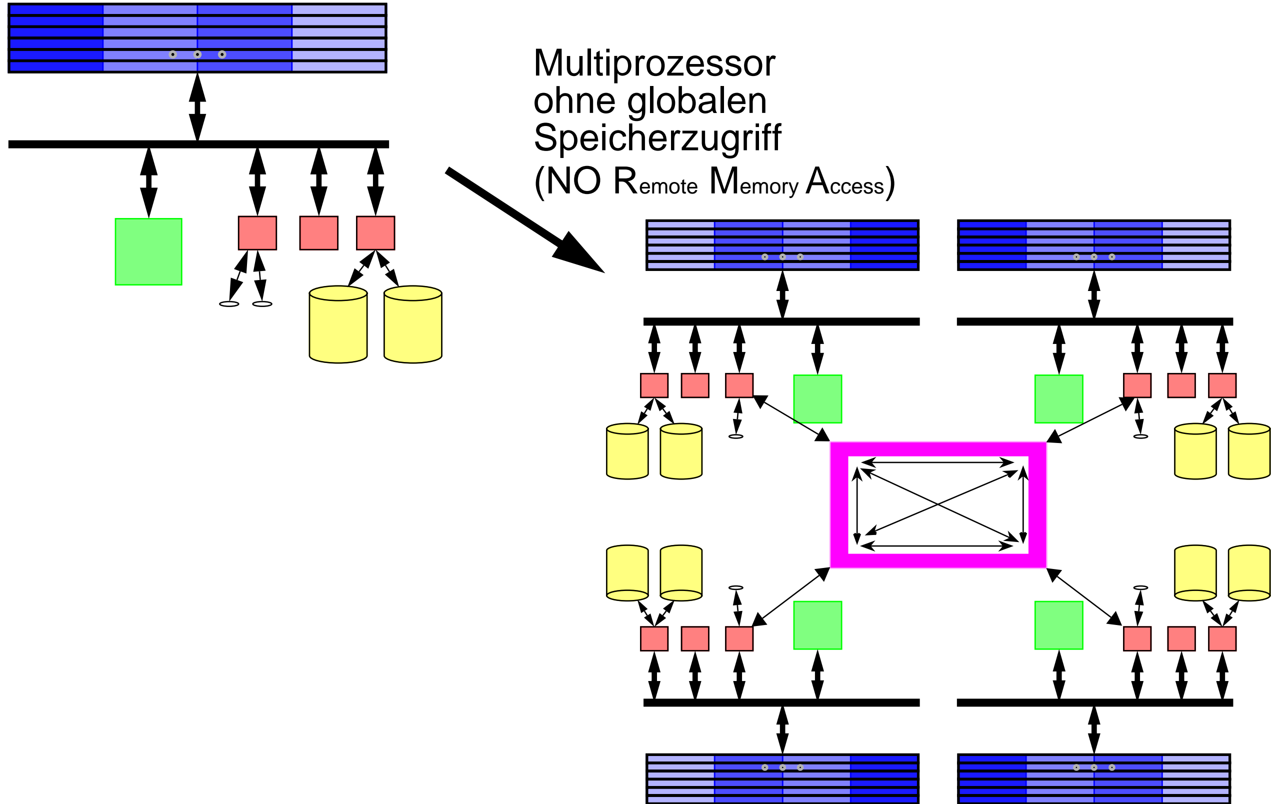
Multiprozessor
mit hierarchischem
Speicherzugriff
(Non Uniform Memory Access)



© Frank Bellosa, Univ. Erlangen-Nürnberg, IMMD IV, 1999
Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

3 Systemebene (2)

- Multiprozessor mit lokalem Adreßraum



Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

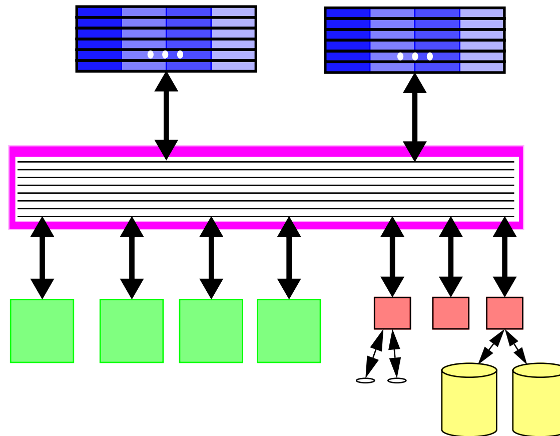
3 Systemebene (3)

- Verbindungsnetzwerke für Multiprozessoren
  - Wunsch: Vollständiges Verbindungsnetzwerk; Komplexität steigt mit K!
  - Abhilfe:
    - Einschränkung der Nachbarschaftsbeziehung
    - globale Kommunikation wird mehrstufig mit - möglichst autarken, nicht die Rechenleistung einzelner Knoten belastenden - Routern realisiert.
  - Kriterien der Bewertung von statischen Verbindungsnetzwerken
    - Anzahl der direkt erreichbaren Nachbarknoten (Links)
    - Anzahl der Zwischenstufen die erforderlich sind um den am weitesten entfernten Knoten zu erreichen (Durchmesser).
    - Anzahl der Knoten in der nächsten Ausbaustufe (Ausbauinkrement)
    - Anzahl der Verbindungssegmente (Komplexität)

Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

3 Systemebene (4)

- Bussysteme (meist in UMA Multiprozessoren).

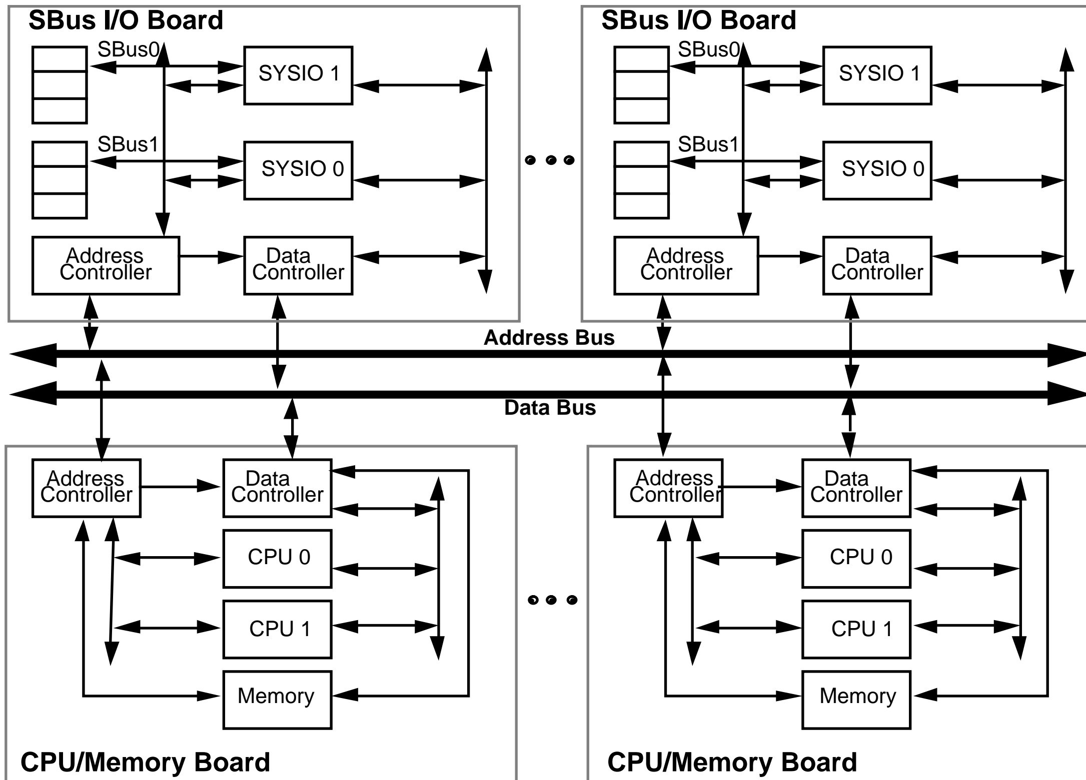


- Maximal erreichbare Prozessorzahl liegt heute bei 24 Prozessoren.
- Parallelisierung durch Aufspaltung in Address- und Datenbus
- Latenzverbergung durch Transaktionskonzept
- Bustakt kann nur durch Verkürzung des Busses gesteigert werden (-> Centerplane Ansatz, maximaler Bustakt derzeit bei ca. 100 Mhz).
- Beispiel: SUN Enterprise 3000/4000/6000 Server

Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

3 Systemebene (5)

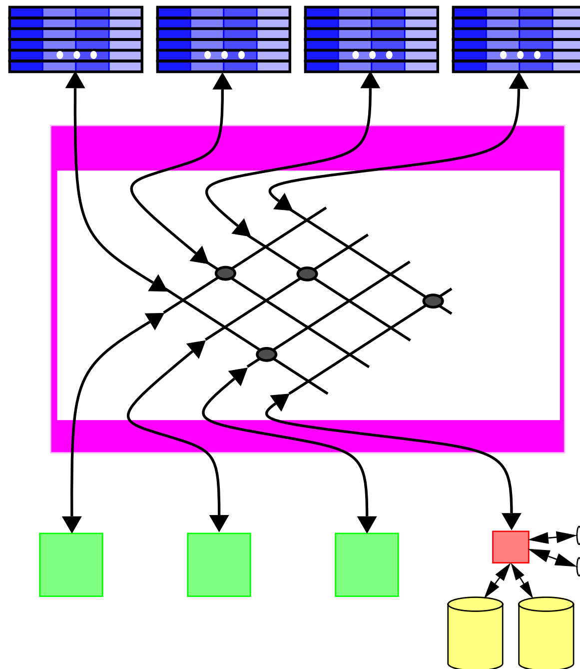
- Architektur SUN Enterprise X000



Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

3 Systemebene (6)

- Kreuzschiene=Crossbar in UMA Multiprozessoren

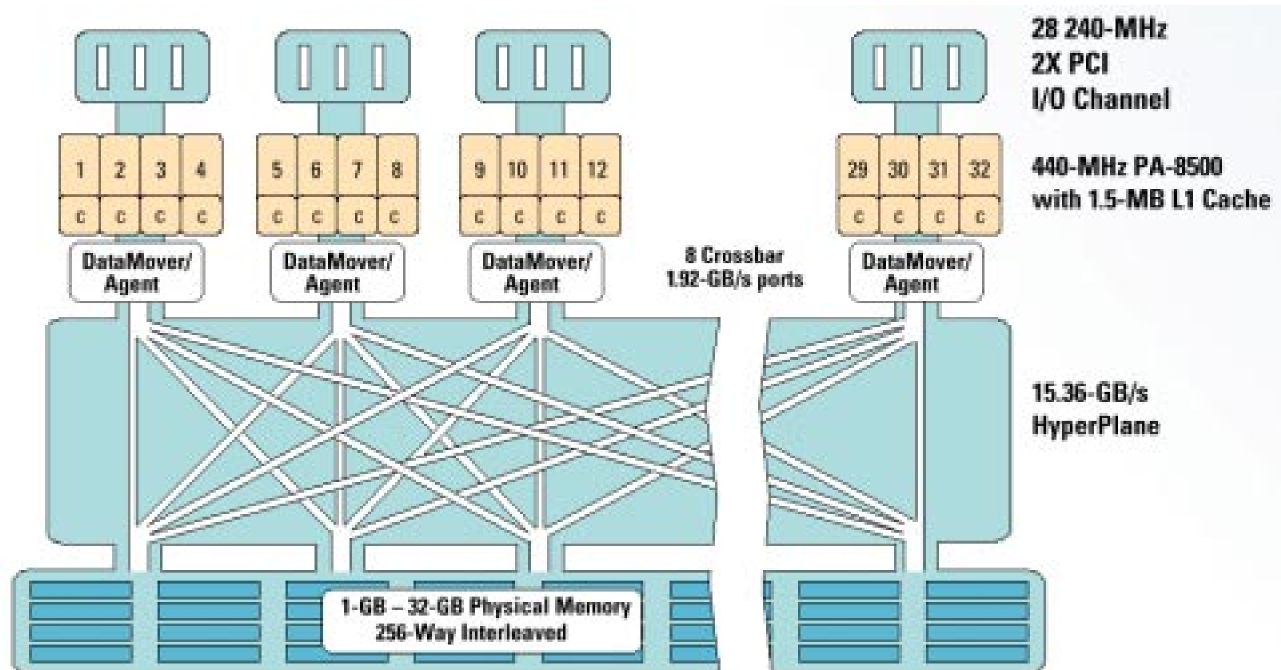


- Nichtblockierende Verbindung zum Hauptspeicher
- Hohe Bandbreite
- Hardware-Komplexität steigt quadratisch
- Maximal erreichbare Prozessorzahl liegt heute bei 32 Prozessoren
- Beispiel: HP V2500 Klasse

Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

3 Systemebene (7)

- Architektur HP V2500 Class

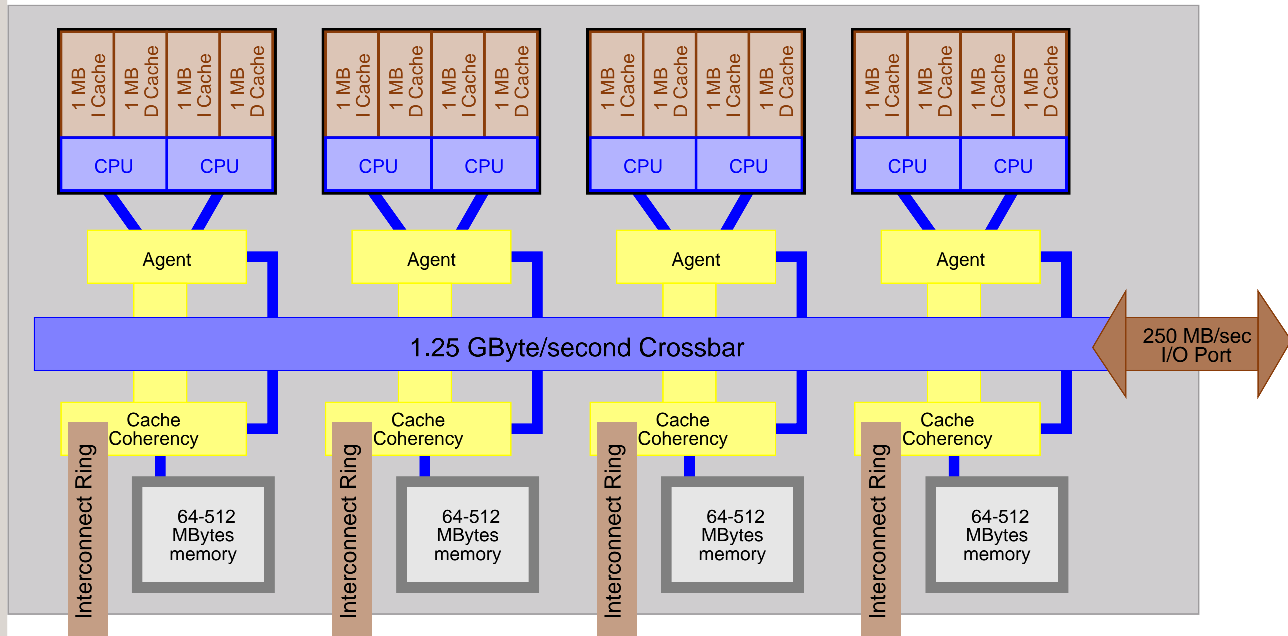


Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

Systemebene (8)

- Multiprozessoren mit hierarchischem Speicherzugriff (NUMA)

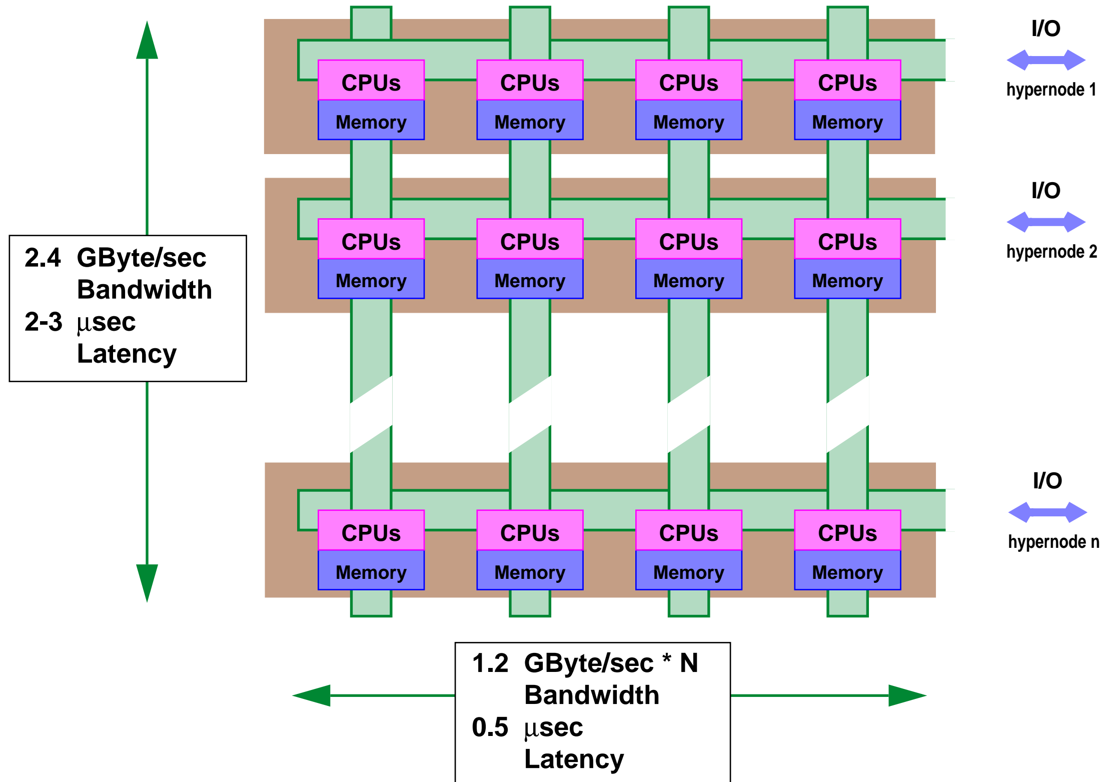
Convex SPP1600 - Hypernode Architektur (RRZE)



Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

Systemebene (9)

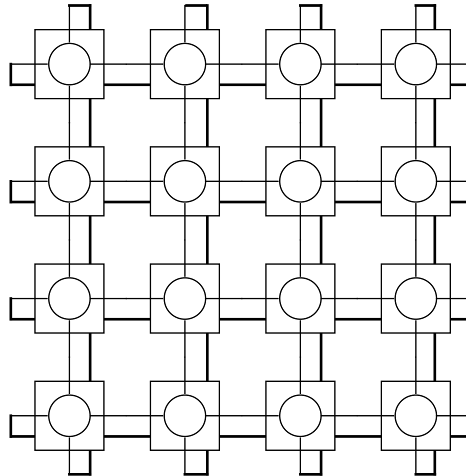
- Convex SPP1 - Speicherhierachie



Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

3 Systemebene (10)

■ 2D *(3D)* Torus



◆ Links pro Knoten: 4 *(6)*

◆ Anzahl der Knoten: $K = q^{2(3)}$; q = Kantenlänge ; Komplexität: $A = 2*q^2$ $(3*q^3)$

◆ Ausbauinkrement: I = 2(q + 1) - 1 $(3q^2 + 3q + 1)$

◆ Max. Routingstufen: $D = 2 \cdot \lfloor q/2 \rfloor$; $(3 \cdot \lfloor q/2 \rfloor)$

◆ Typische Vertreter: Intel ASCI Red (9152 PPro), *Cray T3E (2048 Alpha)*

Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

- Beispiel NORMA MP: Intel ASCI (**A**ccelerated **S**trategic **C**omputing **I**nitiative) Red

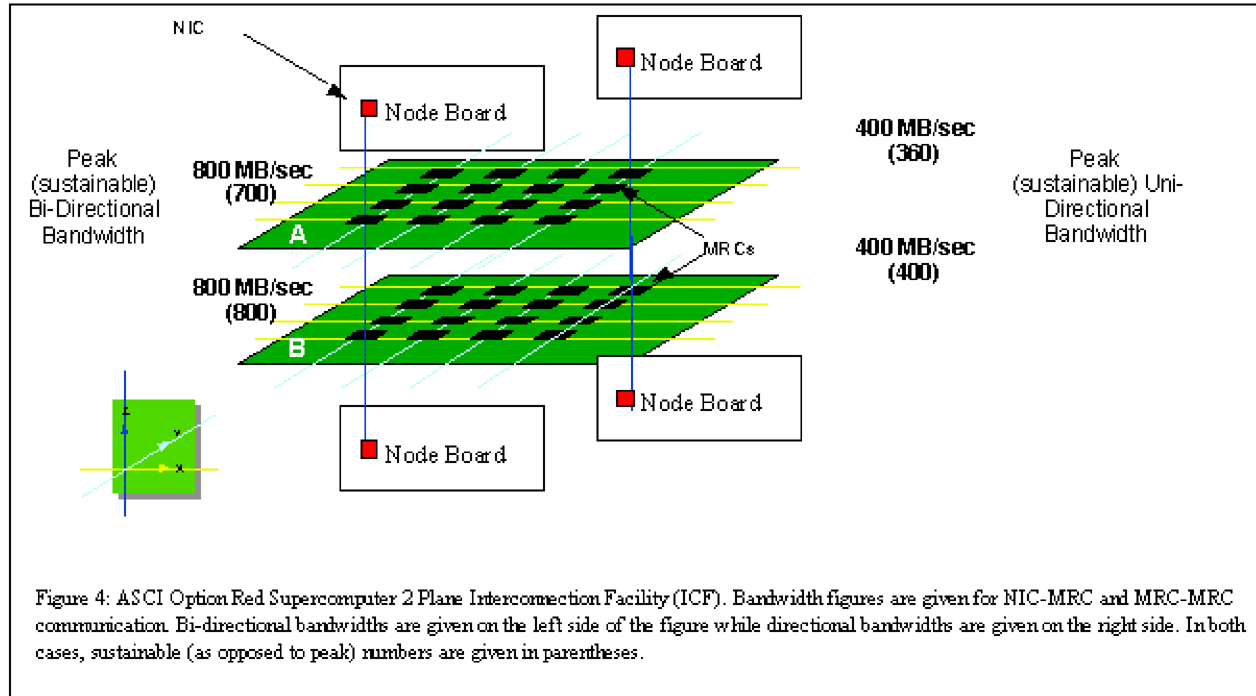
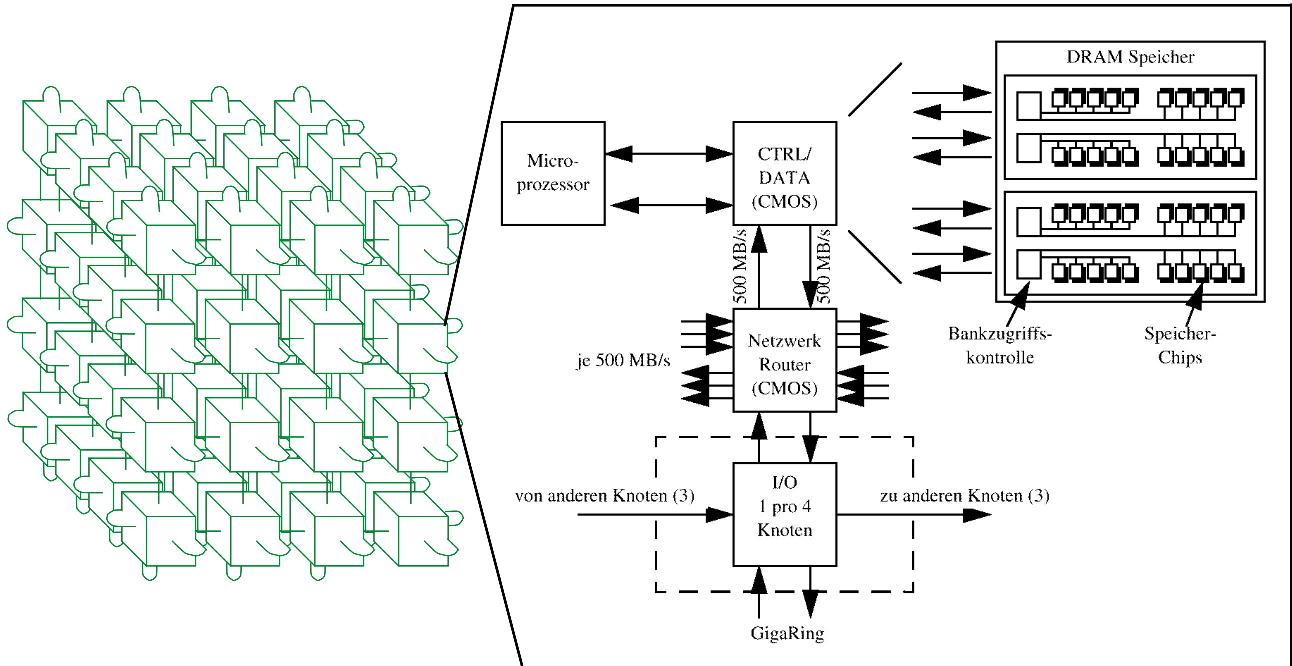


Figure 4: ASCI Option Red Supercomputer 2 Plane Interconnection Facility (ICF). Bandwidth figures are given for NIC-MRC and MRC-MRC communication. Bi-directional bandwidths are given on the left side of the figure while directional bandwidths are given on the right side. In both cases, sustainable (as opposed to peak) numbers are given in parentheses.

Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

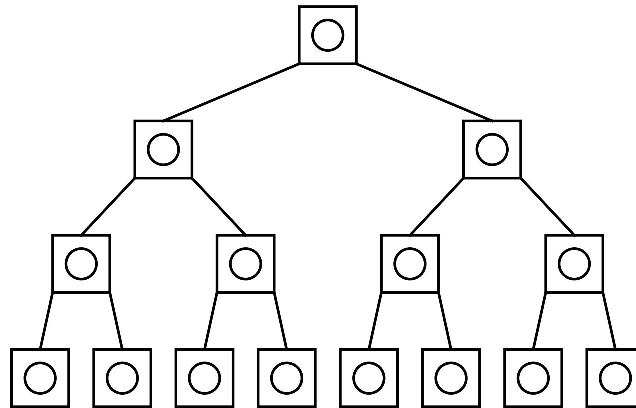
3 Systemebene (12)

- Beispiel NUMA MP: Cray T3E mit DEC Alpha 21164 CPU (600 MHz)



Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

Binärbaum

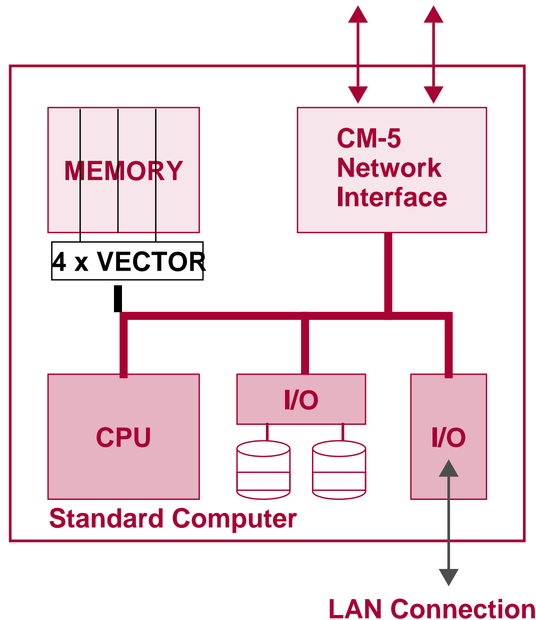


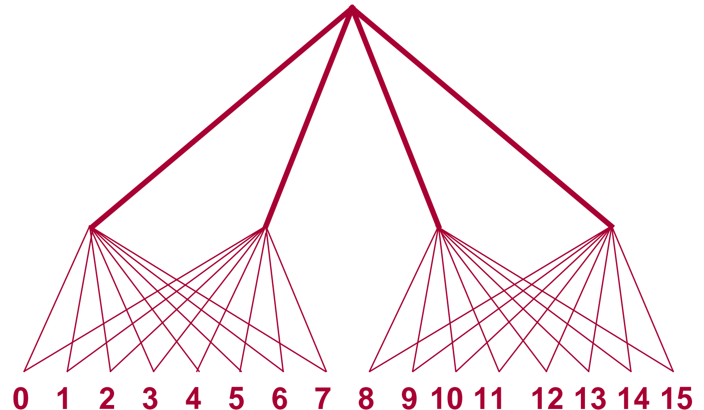
- Links pro Knoten: 3;
- Anzahl der Knoten: $K = 2^d - 1$; d = Ebenenzahl
- Ausbauinkrement: I = K + 1;
- Max. Routingstufen: D = 2(ld (K+1)-1) = 2(d - 1)
- Vermeidung des Kommunikationsengpasses zur Wurzel hin: Fat-Tree
- Typischer Vertreter: Thinking Maschines CM5

Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

3 Systemebene (14)

- Beispiel NORMA MP: Thinking Machines CM-5 am IMMD

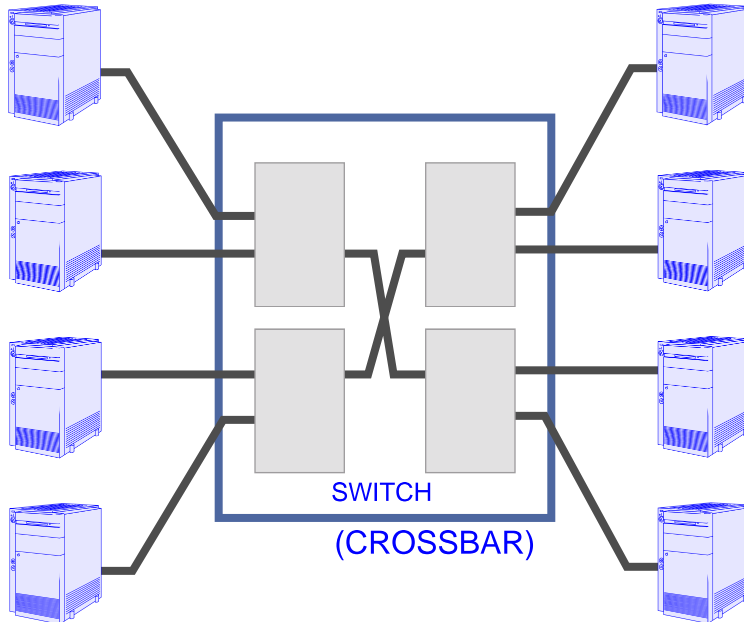




Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

3 Systemebene (15)

- Verbindungnetz mit zentraler Paketvermittlung
  - ◆ Beispiel NORMA MP: IBM SP-2 (max. 128 Prozessoren) am LRZ München



- 70 Knoten IBM 9000/590 (66.7 MHz, 267 MFlop/s)
- 4 dicke, 256MByte
- 56 dünne, 128MB, geringere Speicherbandbreite, kleinerer E/A-Ausbau, gleiche Peak-Leistung
- 300 µsec Latenz (IP)
- 40 µsec Latenz(IBM Prot.)
- 12 MB/s Pt. - Pt. Bandbr. (IP)
- 35 MB/s Pt. - Pt. Bandbr.

Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

4 Netzwerkebene

- Verbindung von mehreren Rechnern mit IP-Technologie zu einem Cluster
- Keine Spezialhardware erforderlich
- Die LAN-Technologie nähert sich immer mehr dem Spezialnetzwerk an
  - Myri-Net ca. 300 MBit/s
  - ATM 622 MBit/s
  - Ethernet 1 Gbit/s

Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

B.3 Klassifikation

- nach Flynn

Data-Stream / Instruction-Stream	SINGLE	MULTIPLE
SINGLE	**SISD** Klass. von Neumann-Architektur (Monoprozessor)	**SIMD** Vektor - Array - Prozessoren
MULTIPLE	**MISD** nicht sinnvoll	**MIMD** Parallelrechner - aus SISD - Prozessoren - aus SIMD - Prozessoren

Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

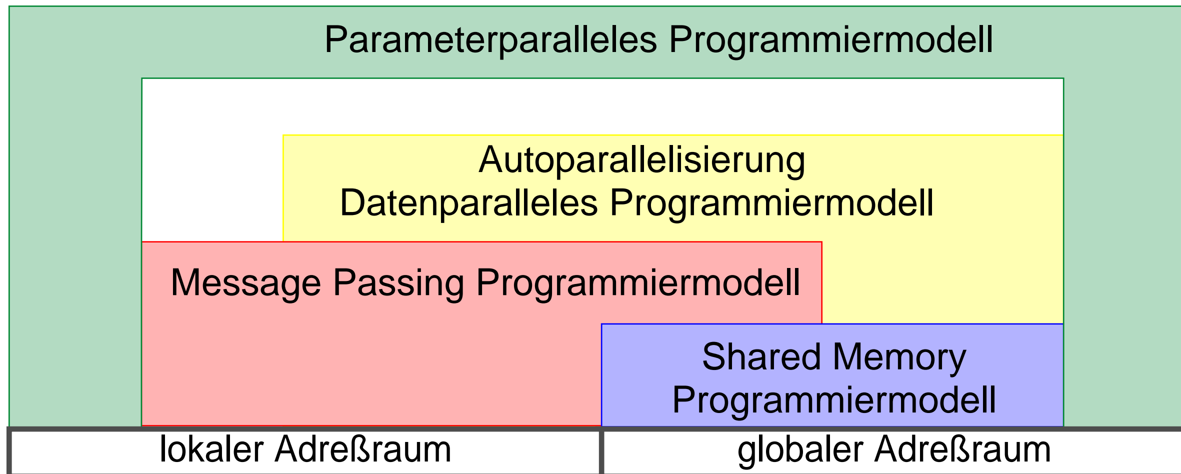
B.3 Klassifikation (2)

Typische MIMD Vertreter

	lokaler Adreßraum	globaler Adreßraum
verteilte Speicherorganisation	NORMA: • Workstation-/Server-Cluster • IBM SP2 • Intel ASCI Red • Fujitsu VPP (?)	ccNUMA: • Convex SPP • SGI Origin • Sequent NumaQ NUMA: • CRAY T3E
zentrale Speicherorganisation		UMA: • SUN Enterprise X000 • HP V-Class • DEC AlphaServer • IBM RS/6000 S70 • NEC SX/4 / CRAY T90

Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

B.4 Programmiermodelle



- Unabhängig vom Programmiermodell muß man beim Programmieren paralleler Systeme ein Optimum folgender Eigenschaften erreichen:
  - Minimierungen der Kommunikation
  - Maximierung des "lokalen" Verhaltens
  - Verteilung der Rechenlast auf alle Prozessoren (loadbalancing)

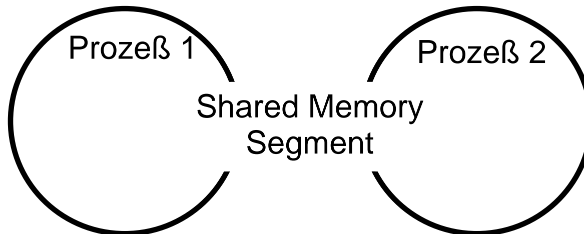
Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

B.4 Programmiermodelle

Shared Memory Programmiermodell

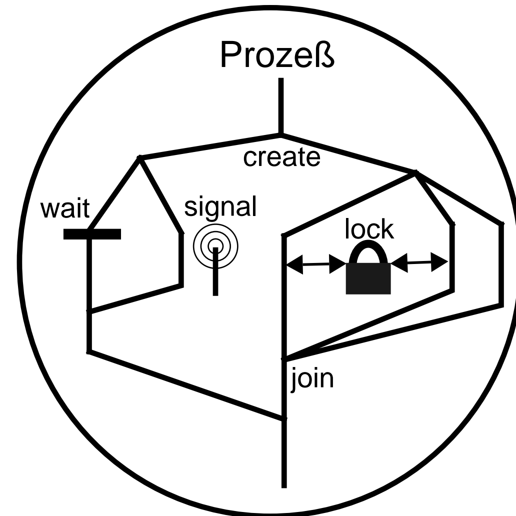
- Geeignet für UMA- und NUMA-Architekturen
- Kommunikation über gemeinsame Speicherbereiche
- Explizite Koordinierung
- Aktivitätsträger sind Prozesse und/oder Threads

Parallele Prozesse



shm_id = shm_get(Schlüssel,Größe,Zugriffsrechte)	
ptr = shm_at(shm_id)	shm_dt(prt)
semaphore = sema_create(Schlüssel,Zugriffsrechte)	
wait(semaphore)	signal(semaphore)

1 Prozeß + Parallele Threads



Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

B.4 Programmiermodelle

Shared Memory Programmiermodell (2)

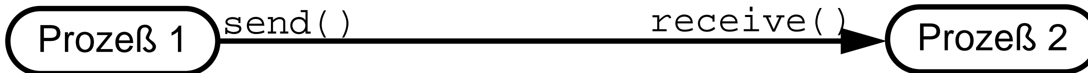
- Keine Unterstützung durch spezielle Sprachkonstrukte
- Keine “automatische” Parallelisierung
- Unterstützung durch Thread-Libraries (Pthreads-Library)
- Prozessorzuordnungen ist meist beinflußbar
- Betriebssystem/Thread-Library sorgt für Lastverteilung
- Feingranulare Parallelität möglich
- Programmiersprachen: C; C++
- Geeignet, hocheffiziente Programme zu schreiben

Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

B.4 Programmiermodelle

Message Passing Programmiermodell

- Geeignet für alle MIMD Architekturen
- Kommunikation über *sent/receive*-Mechanismen



- Explizite Koordinierung; implizit realisiert über *sent/receive*-Mechanismen
- Aktivitätsträger sind Prozesse (lokal sind Threads möglich)
- Keine Unterstützung durch spezielle Sprachkonstrukte
- Keine “automatische” Parallelisierung
- Unterstützung durch Message Passing-Libraries: (PVM; MPI)
- NORMA: Feste Zuordnung Prozeß/Prozessor; Prozeßmigration unmöglich
- Geeignet für grobgranulare Parallelität; Grad ist abhängig von verwendeter Rechnerarchitektur (*Latenz/Bandbreite* des Verbindungsnetzwerks).
- Programmiersprachen: C; C++; Fortran
- Geeignet portable Programme für alle denkbaren Architekturen zu schreiben

Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

B.4 Programmiermodelle

Shared Memory - Autoparallelisierung/Datenparallele Programmierung

- Geeignet für alle UMA/NUMA Architekturen
- Implizite Kommunikation, realisiert über gemeinsame Speicherbereiche
- Implizite Koordinierung
- Aktivitätsträger sind Prozesse/Threads mit identischer Kontrollflußstruktur; dynamisch verschiedener Kontrollfluß möglich.
- Unterstützung der parallelen Programmierung durch spezielle Sprachkonstrukte; typischer Weise "Feldverarbeitung"
- Automatische Parallelisierung potentiell nebenläufiger Programmteile (z.B. Schleifen); Unterstützungsmöglichkeit durch Compilerdirektiven
- Feingranulare Parallelität möglich
- Nur ein Teil der Nebenläufigkeit des Lösungsverfahrens ist nutzbar.
- Programmiersprachen: Fortran90/95, C, C++
- Einfachere Fehlersuche, da Koordinierungsfehler nicht möglich sind.

Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

B.4 Programmiermodelle

- Message Passing - Autoparallelisierung/Datenparallele Programmierung
  - Geeignet für alle MIMD Architekturen
  - Implizite Kommunikation/Koordination über *sent/receive* Mechanismen
  - Aktivitätsträger sind Prozesse mit identischer Kontrollflußstruktur; dynamisch verschiedener Kontrollfluß möglich
  - Unterstützung der parallelen Programmierung durch spezielle Sprachkonstrukte; typischer Weise “Feldverarbeitung”.
  - Automatische Parallelisierung potentiell nebenläufiger Programmteile (z.B. Schleifen); Unterstützungsmöglichkeit durch Compilerdirektiven
  - Geeignet für grobgranulare Parallelität; Grad ist abhängig von verwendeter Rechnerarchitektur (*Latenz/Bandbreite* des Verbindungsnetzwerks).
  - Nur ein Teil der Nebenläufigkeit des Lösungsverfahrens ist nutzbar.
  - Programmiersprachen: High Performance Fortran - HPF
  - Einfachere Fehlersuche durch implizite Kommunikation/Koordination, jedoch komplexer Debugger zur Überwachung einer Vielzahl von Rechnern

Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

B.4 Programmiermodelle

- Parameterparalleles Programmiermodell
  - Keine dynamische Kommunikation zwischen den Komponenten des parallelen Systems
  - Master/Slave - Konfiguration, wobei der Masterprozeß an die sonst unabhängigen Slaveprozesse Parametersätze verteilt und ggf. die Ergebnisse wieder einsammelt und auswertet.
  - Im einfachsten Fall ist der Masterprozeß eine Shellprozedur.

Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

B.5 Lokaler vs. Globaler Adreßraum

- NORMA Architekturen
  - ◆ Basis-Programmiermodel ist 'Message Passing'.
  - ◆ Das Verhältnis Befehlsausführungszeit/Latenzzeit zwischen Aktivitätsträgern(Threads, Prozesse) auf verschiedene Prozessoren liegt bestenfalls bei 1 : 10.000 (IBM SP-2);
    Hinzu kommt ein erheblicher Protokoll - Overhead
  - ◆ Eine 'shared memory'-Programmiermodell kann als 'shared distributed memory' vom Betriebssystem über die Seitenadressierung simuliert werden, ist aber extrem ineffizient. ('paging' über 'message passing'); Beispiel: Intel Paragon

© Frank Bellosa, Univ. Erlangen-Nürnberg, IMMD IV, 1999
Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

B.5 Lokaler vs. Globaler Adreßraum

- UMA/NUMA Architekturen

  ◆ Basis-Programmiermodel ist 'Shared Memory'.

  ◆ Das Verhältnis Befehlsausführungszeit/Latenzzeit zwischen Aktivitätsträgern(Threads, Prozesse) auf verschiedene Prozessoren reduziert sich auf Speicherzugriffe.
  Bei NUMA-Architekturen hängt die Zeit von der "Entfernung" ab.
  Beispiel Convex SPP1600:

Beispiel Convex SPP1600:	1 : 1	Register/Cache
	1 : 40	Nodespeicher (UMA)
	1 : 200	Internodespeicher (NUMA)

  ◆ Ein 'message passing'-Programmiermodell läßt sich effizient auf einem 'shared memory'-System simulieren.
  ('message passing' über 'shared data'-Segmente)

- Schlußfolgerung:

  ◆ Grobgranulare Parallelität ist geeignet für beide Architekturklassen

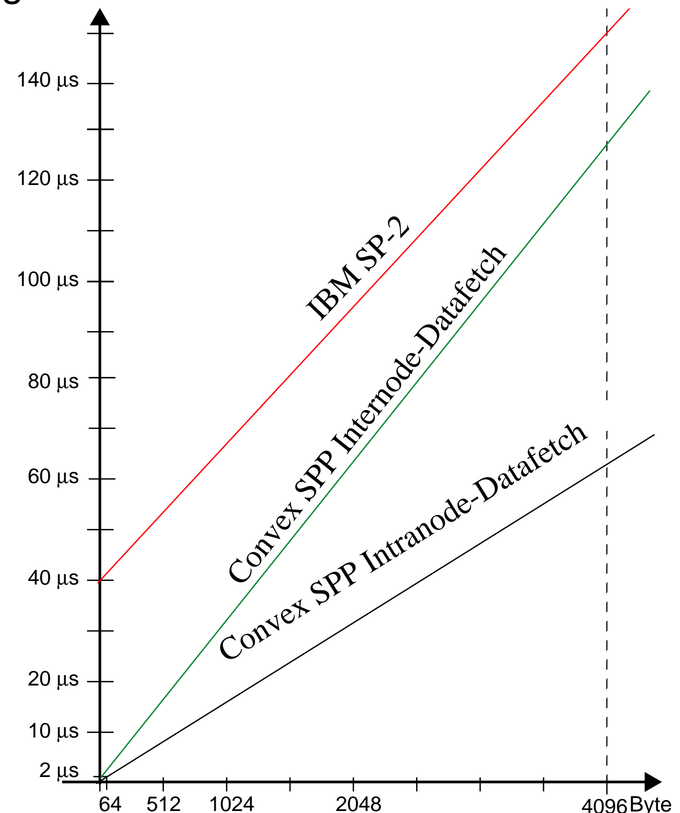
  ◆ Feingranulare Parallelität ist nur geeignet für 'shared memory'-Systeme

Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

B.5 Lokaler vs. Globaler Adreßraum

Wann ist ein Programm grob-/feingranular?

- IBM SP-2
  - 40 µs Latenz
  - 35MB/s
  - Transfereinheit “beliebig”
- Convex SPP1600
  - 0.5 µs Latenz (Intranode)
    1.2 GB/s Bandbreite
    Transferblock 32 Byte
  - 2 µs Latenz (Internode)
    2.4 GB/s Bandbreite
    Transferblock 64 Byte



Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

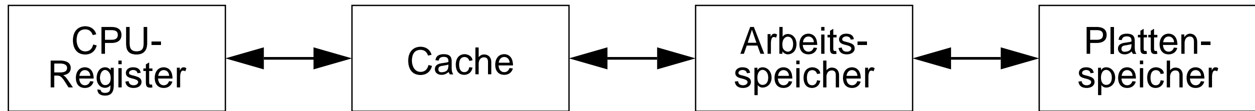
B.6 Leistungsausbeute

- Peak-Leistung ist die theoretisch obere Schranke
- Viele Leistungsmaße geben jedoch immer nur spezifische Teilaspekte wieder. Beispiele:
  - SPECmarks
  - LINPACK
  - LAPACK
  - TPC
  - Beliebt sind ’rankinglists’ (TOP 500) egal welchen Aussagewert sie haben.
- Sequentielle Leistung:
  - Wie gut wird die Leistung des Prozessors im sequentiellen Fall ausgenutzt?
- Parallele Leistung
  - Welche Leistung erhält man durch die Parallelisierung?

Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

1 Grundlagen für Prozessoreffizienz

- Schneller Speicherzugriff durch Ausnutzung der Speicherhierarchie



niedrige Latenz
hohe Bandbreite

hohe Latenz
niedrige Bandbreite

◆ Zeitliche Lokalität:
Wenn ein Speicherort angesprochen wird, dann wird er innerhalb eines kleinen Zeitintervalls mehrfach angesprochen

◆ Örtliche Lokalität:
Wenn ein Speicherort angesprochen wird, dann werden innerhalb eines kleinen Zeitintervalls überwiegend Speicherorte mit geringem Adreßabstand angesprochen

Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

1 Grundlagen für Prozessoreffizienz (2)

- Beispiel

```
int M = 10000;
int main()
{
 int i, j, k;
 int a[M][M], b[M][M], c[M][M];
 for (i = 0; i < M; i++)
     for (j = 0; j < M; j++) {
       a[i][j] = i; b[i][j] = j; c[i][j] = 0;
     }
 for (i = 0; i < M; i++)
     for (j = 0; j < M; j++)
         for (k = 0; k < M; k++)
             c[i][j] = c[i][j] + a[i][k] * b[k][j];
}
```

- `c[i][j]` weist zeitliche Lokalität auf
- `a[i][k]` weist örtliche Lokalität auf
- `b[k][j]` weist keines von beidem auf

© Frank Bellosa, Univ. Erlangen-Nürnberg, IMMD IV, 1999 B-Intro.fm 1999-03-15 17.40
Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

1 Grundlagen für Prozessoreffizienz (3)

- Ausnutzung der Prozessorparallelität
  - architekturspezifische Codegenerierung (Prozessortyp, Cachegröße)
  - geringe Datenabhängigkeit im Code
    (➜instruction reordering, ➜outstanding loads, ➜instruction scheduling)
  - verhersagbare Sprünge (➜branch prediction, ➜loop unrolling)
  - vorhersagbare Speicherzugriffe (➜prefetch operations)

- Ausnutzung optimierter Bibiotheken und Algorithmen
  - libc (allgemeine Bibliotheken z.B. **`memcpy`**(), **`bsort()`**)
  - NAG, BLAS, LAPACK (numerische Bibliotheken)
  - *Numerical Recipies*
  - ...

Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

2 Leistungsausbeute durch Parallelisierung

- *Speedup* (im Vergleich zum Ablauf des gleichen Programms auf einem Prozessor)

$$s(p) = \frac{t(1)}{t(p)}$$

- Effizienz (im Vergleich zum Ablauf des gleichen Programms auf einem Prozessor)

$$e(p) = \frac{t(1) \cdot 100}{t(p) \cdot p} \%$$

- Eine andere Variante der Betrachtung wäre den *Speedup* bzw. die *Effizienz* durch “Vergrößerung” des Problems zu bestimmen.

Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

2 Leistungsausbeute durch Parallelisierung (2)

Amdahls Gesetz

- $(1) = t(1) \cdot (f_s + (1 - f_s)) = t_s + t_p$

  f_s = sequentieller Programmanteil
  f_p = parallelisierbarer Programmanteil

- $t(p) = t_s + \frac{t_p}{p}$

- $(p) = \frac{1}{f_s + \frac{1 - f_s}{p}} \leq \frac{1}{f}$

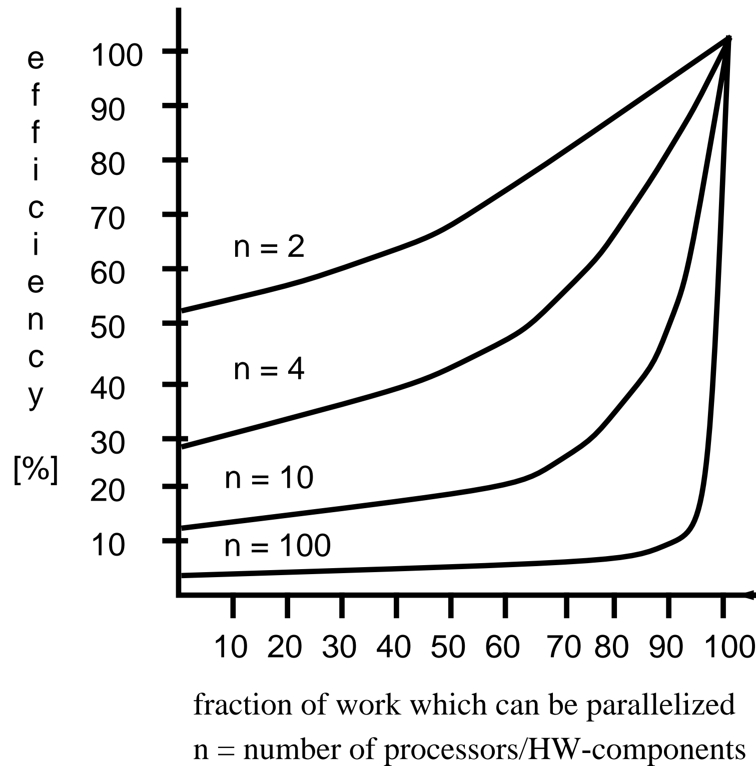
  d.h. der maximale Speedup bzw. die max. Effizienz sind durch den sequentiellen Anteil (wesentlich) beschränkt.

- Der algorithmische Mehr-/Minderaufwand der durch die Parallelisierung des Problems selbst entsteht, wird bei diesen einfachen Abschätzungen nicht berücksichtigt, kann aber erheblich sein!

- Bei der Parallelisierung kann die effektive Speicherbandbreite der Prozessoren durch die Ausnutzung der Caches ansteigen (Parallelität auf Speicherebene). Damit wird ein superlinearer Speedup möglich.

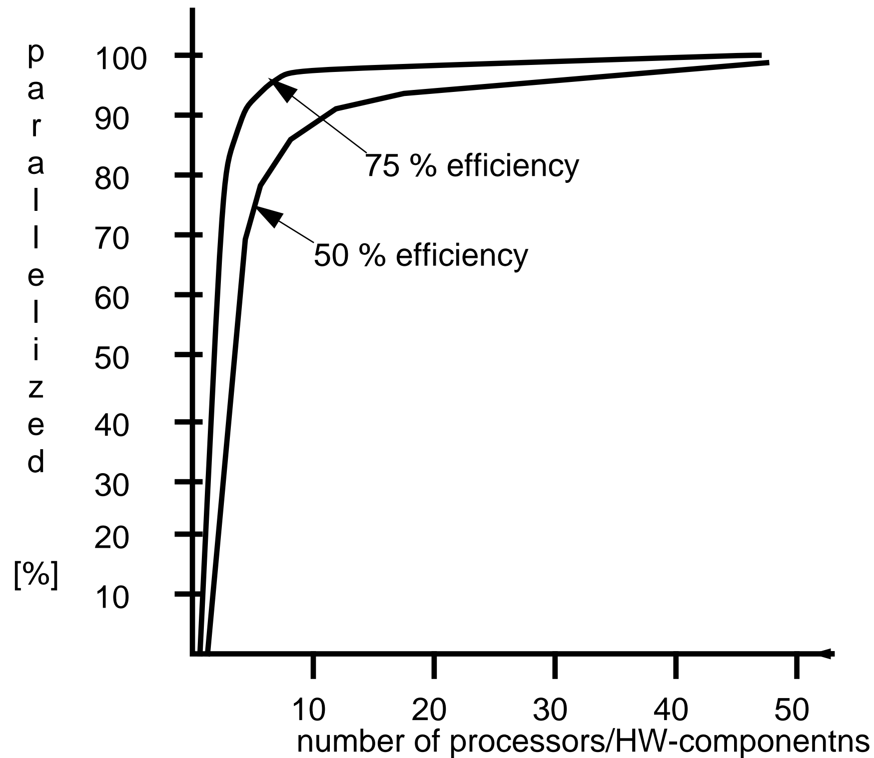
Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

2 Leistungsausbeute durch Parallelisierung (3)



© Frank Bellosa, Univ. Erlangen-Nürnberg, IMMD IV, 1999
Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

2 Leistungsausbeute durch Parallelisierung (4)



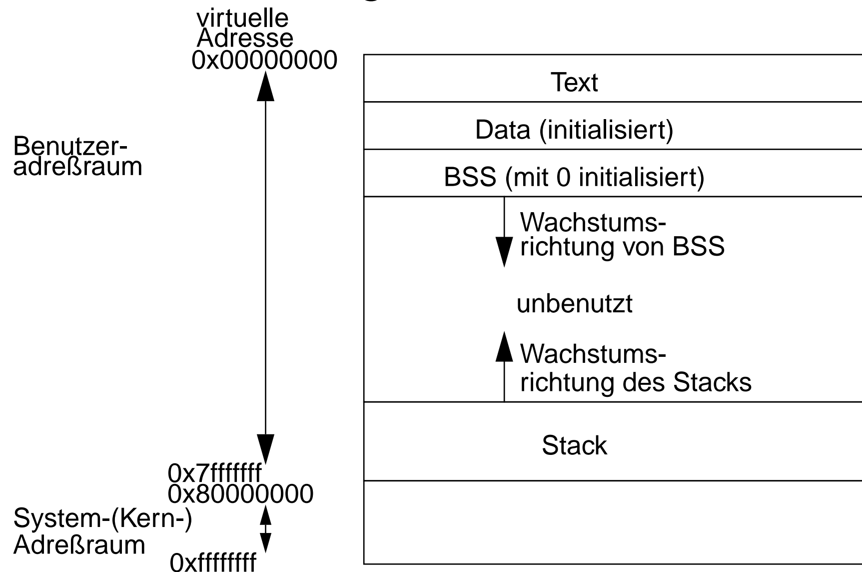
Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

C Optimierung des Speicherzugriffs

C.1 Virtuelle Adressierung

1 Grundlagen

- Jeder Prozeß besitzt eigenen virtuellen Adreßraum



© Frank Bellosa, Univ. Erlangen-Nürnberg, IMMD IV, 1999
Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

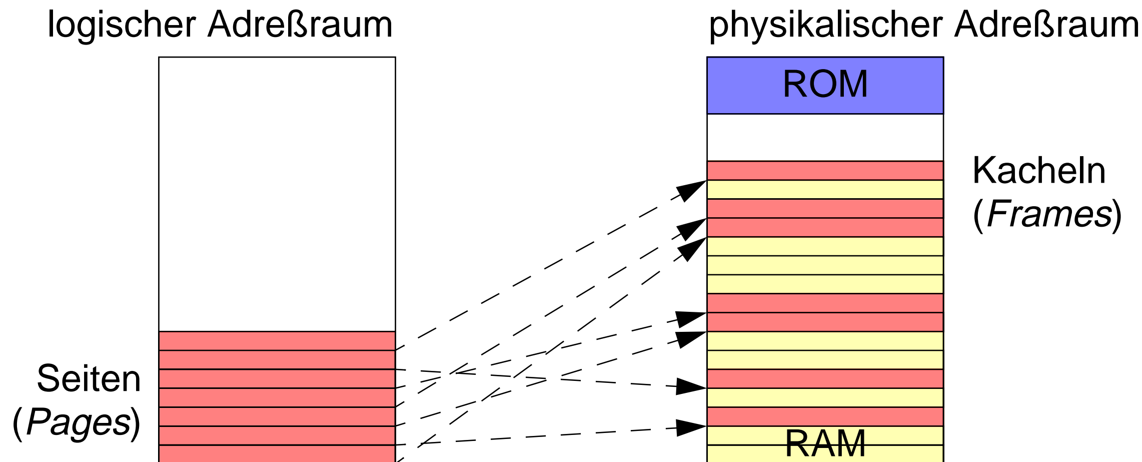
1 Grundlagen der virtuellen Adressierung (2)

- Ziel: Entkoppelung des Speicherbedarfs vom verfügbaren Hauptspeicher
  - Prozesse benötigen nicht alle Speicherstellen gleich häufig
    - bestimmte Befehle werden selten oder gar nicht benutzt (z.B. Fehlerbehandlungen)
    - bestimmte Datenstrukturen werden nicht voll belegt
  - Prozesse benötigen evtl. mehr Speicher als Hauptspeicher vorhanden
- Idee:
  - Vortäuschen eines großen Hauptspeichers
  - Einblenden benötiger Speicherbereiche
  - Abfangen von Zugriffen auf nicht-eingeblendete Bereiche
  - Bereitstellen der benötigen Bereiche auf Anforderung
  - Auslagern nicht-benötigter Bereiche

Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

1 Grundlagen der virtuellen Adressierung (3)

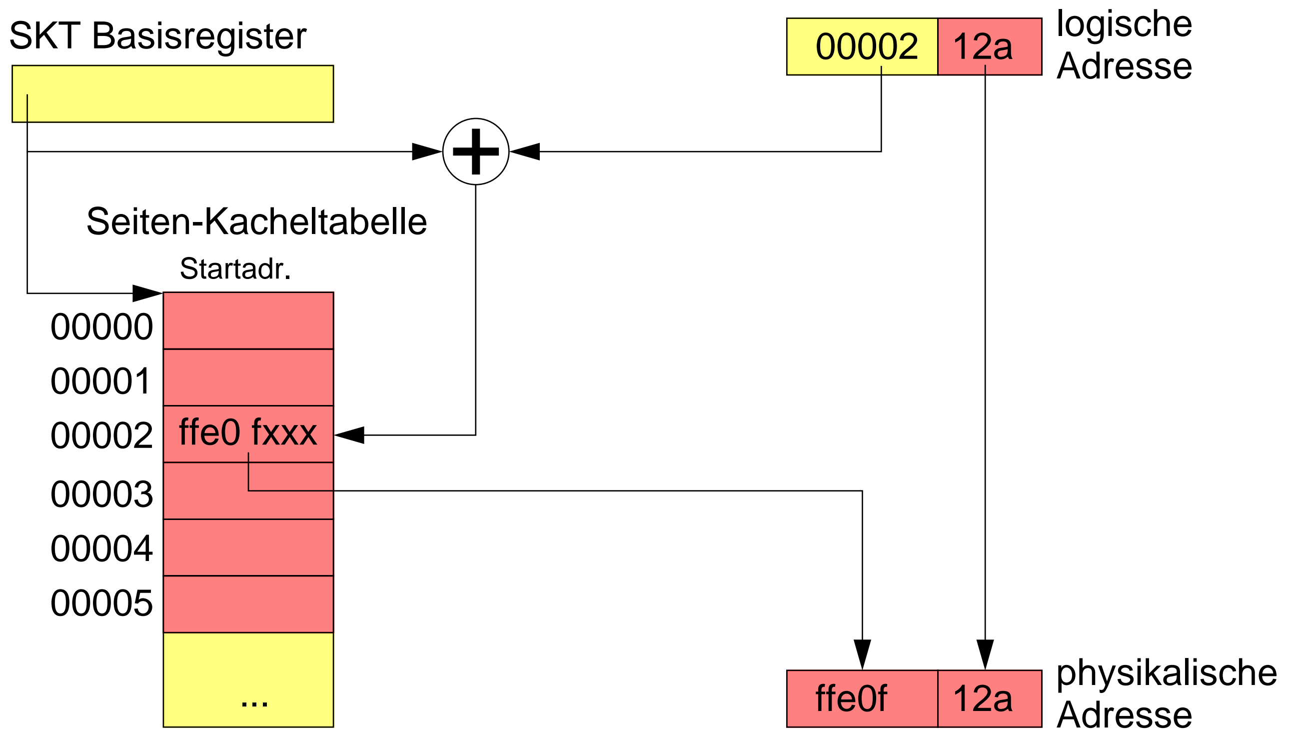
- Einteilung des logischen Adreßraums in gleichgroße Seiten, die an beliebigen Stellen im physikalischen Adreßraum liegen können
  - ◆ Lösung des Fragmentierungsproblem
  - ◆ keine Kompaktifizierung mehr nötig
  - ◆ Vereinfacht Speicherbelegung und Ein-, Auslagerungen



Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

1 Grundlagen der virtuellen Adressierung (4)

- Tabelle setzt Seiten in Kacheln um

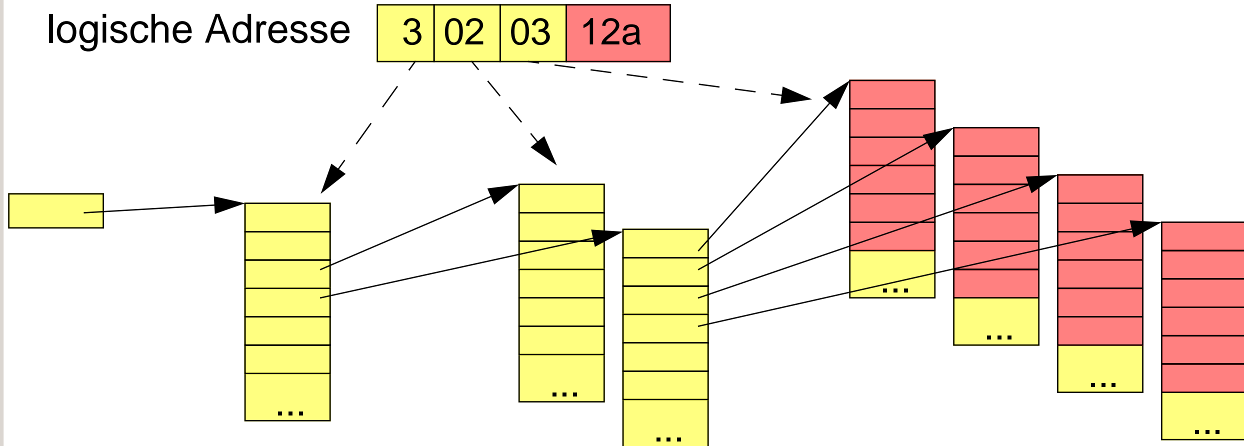


© Frank Bellosa, Univ. Erlangen-Nürnberg, IMMD IV, 1999

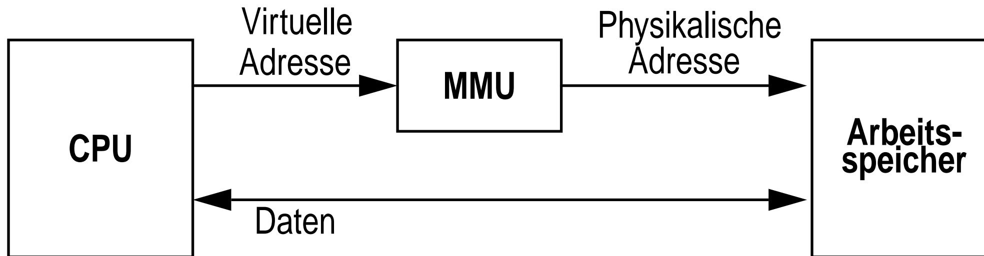
Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

1 Grundlagen der virtuellen Adressierung (5)

- Mehrstufige Seitenadressierung



- MMU führt Adreßabbildung in Hardware durch



Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

1 Grundlagen der virtuellen Adressierung (6)

- Seitenadressierung erzeugt internen Verschnitt
  - letzte Seite eventuell nicht vollständig genutzt

- Seitengröße
  - kleine Seiten verringern internen Verschnitt, vergrößern aber die Seiten-Kacheltabelle (und umgekehrt)
  - übliche Größen: 512 Bytes — 8192 Bytes (8192 Bytes bei UltraSPARC)
  - Systemaufruf: `getpagesize()`

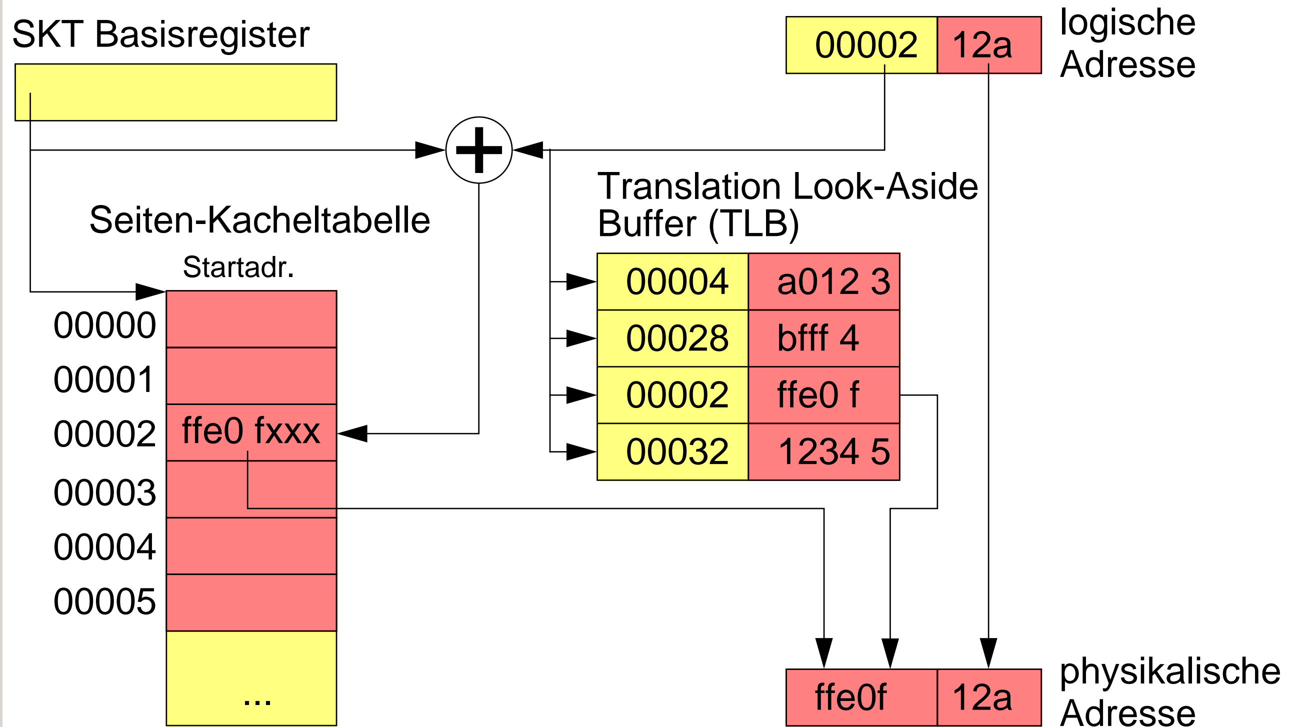
- große Tabelle, die im Speicher gehalten werden muß

- viele implizite Speicherzugriffe nötig

Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

2 Translation Look-Aside Buffer TLB

- Schneller Registersatz wird konsultiert bevor auf die SKT zugegriffen wird



© Frank Bellosa, Univ. Erlangen-Nürnberg, IMMD IV, 1999
Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

2 Translation Look-Aside Buffer (2)

- Schneller Zugriff (< 1 Taktzyklus) auf Seitenabbildung, falls Information im voll-assoziativen Speicher des TLB
  - keine impliziten Speicherzugriffe nötig
- Bei Zugriffen auf eine nicht im TLB enthaltene Seite wird die entsprechende Zugriffsinformation in den TLB eingetragen
  - Ein alter Eintrag muß zur Ersetzung ausgesucht werden
    - Random
    - Round Robin
- TLB Größe
  - Pentium: Daten TLB = 64, Code TLB = 32, Seitengröße 4K
  - Sparc V9: Daten TLB = 64, Code TLB = 64, Seitengröße 8K
  - Größere TLBs bei den üblichen Taktraten zur Zeit nicht möglich

Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

2 Translation Look-Aside Buffer (3)

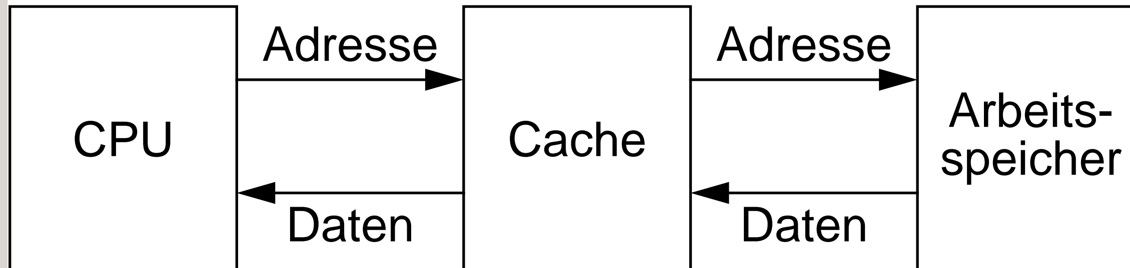
- Der TLB kann nur einen kleinen Teil des Adreßraums abbilden. (z.B. UltraSPARC IIi 64 Einträge x 8 KB = 512 KB)
- Ein TLB-Zugriffsfehler wird in der Regel durch Software aufgelöst. (Durchsuchen der SKT und Einlagerung im TLB; < 32 Instruktionen).
  - ◆ Vorteil: Flexibles Design der SKT möglich
  - ◆ Nachteil: 1 Trap ist nötig (Overhead of precise exceptions in pipelined/superscalar CPUs)
  - ◆ Ausnahmen: Intel x86, teilw. Motorola/IBM PowerPC, HP PA-RISC)
- TLB-Zugriffsfehler erscheinen nicht in der OS-Statistik.
- 1 TLB-Zugriffsfehler = 2 Speicherzugriffe (bis zu 1000 Zyklen!)
- Schon eine Arbeitsmenge von 8KB kann den TLB überfordern.
  ➜ siehe Übung

Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

C.2 Caches

1 Grundlagen

■ Funktion



- ◆ Pufferspeicher zur Ausnutzung der zeitlichen und örtlichen Lokalität
- ◆ Zugriff auf gespeicherte Daten mit geringer Latenz und hoher Bandbreite
- ◆ Reduktion der Zugriffe auf den Arbeitsspeicher (wichtig in Multiprozessen)
- ◆ Puffer für asynchrone Prefetch-Operationen

Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

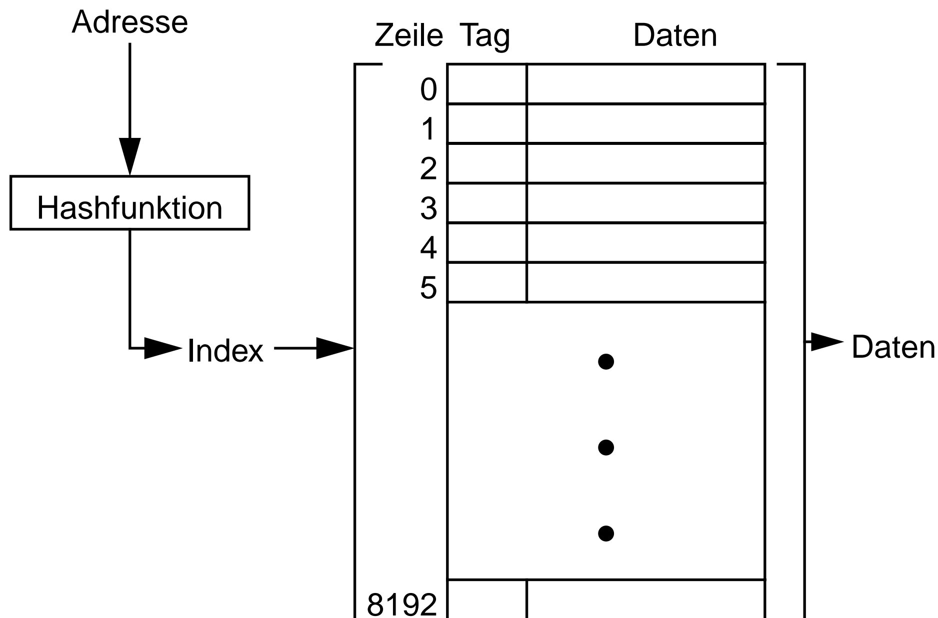
1 Grundlagen (2)

■ Organisation

◆ Aufbau aus Cache-Zeilen fester Länge (typischerweise 32/64 Bytes)

◆ Abbildung der Addresse auf die Zeile (1):

Beispiel: Direkt abgebildeter Cache (direct mapped cache)

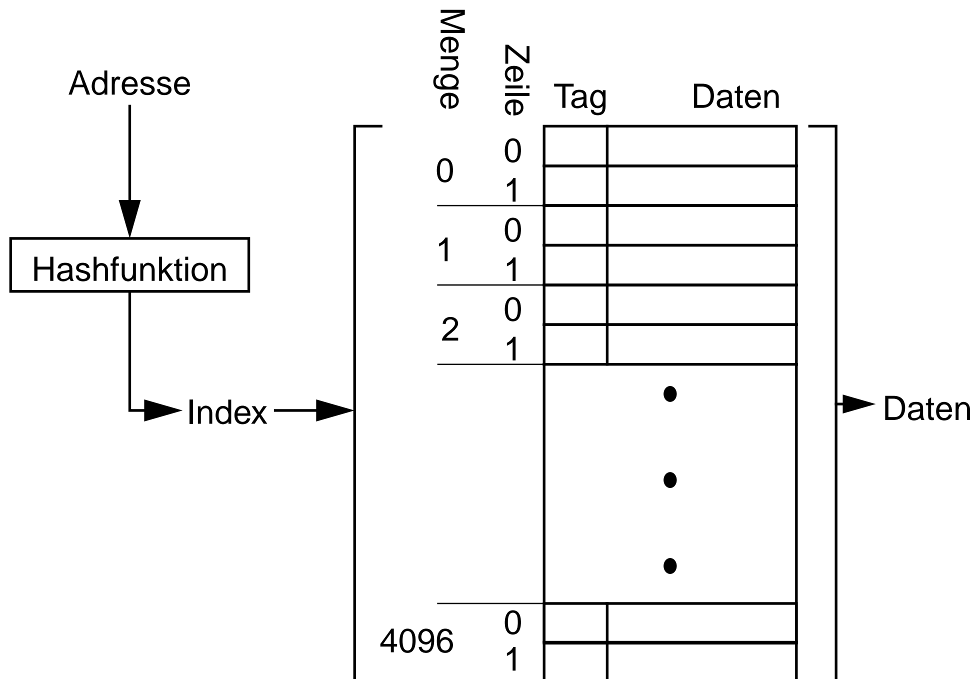


Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

1 Grundlagen (3)

◆ Abbildung der Addresse auf die Zeile (2):

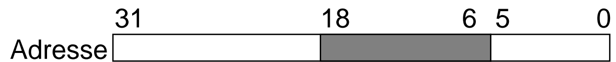
Beispiel: Zweifach assoziativer Cache (two-way set associative cache)

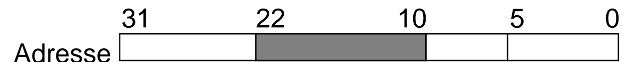


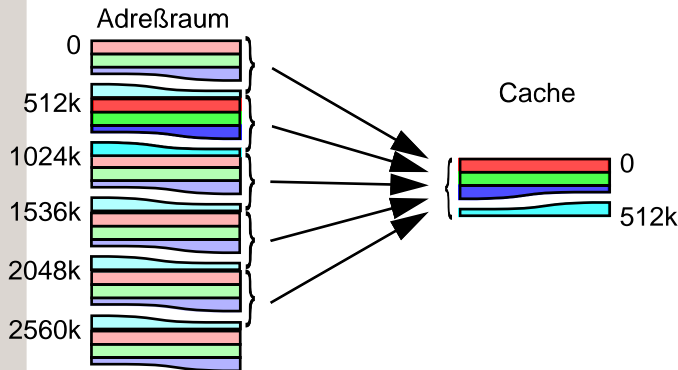
Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

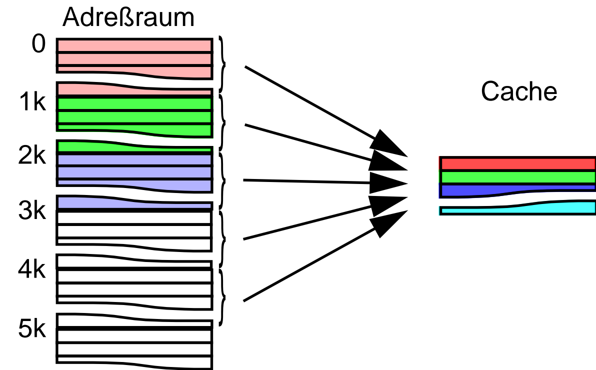
1 Grundlagen (4)

- Hash-Algorithmen
  - Modulo-Hashing, z. B. Bits 0 - 5 zur Auswahl des Bytes innerhalb einer Zeile, Bits 6 - 18 zur Auswahl der Zeile, Bits 19 - 31 sind Tag
  - Hashing durch Ausschnitt aus der Adresse, z. B. Bits 10 - 22 als Zeilenadresse. Dieses Verfahren wird bei Caches jedoch nicht eingesetzt.







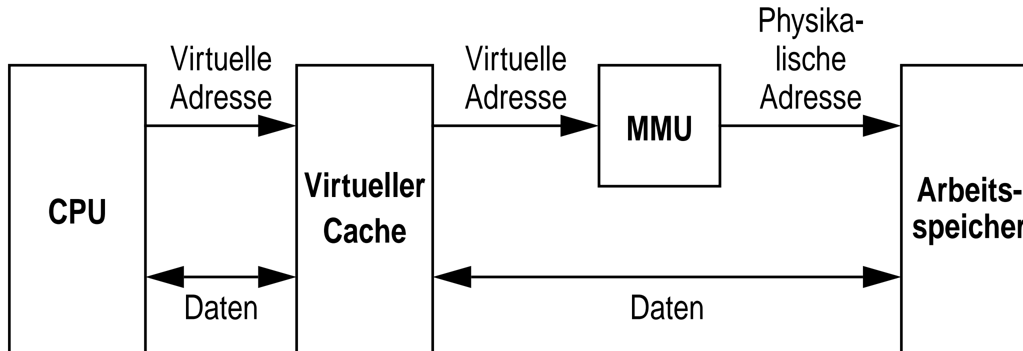


© Frank Bellosa, Univ. Erlangen-Nürnberg, IMMD IV, 1999

Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

1 Grundlagen (5)

■ Virtuelle Caches



◆ Probleme:

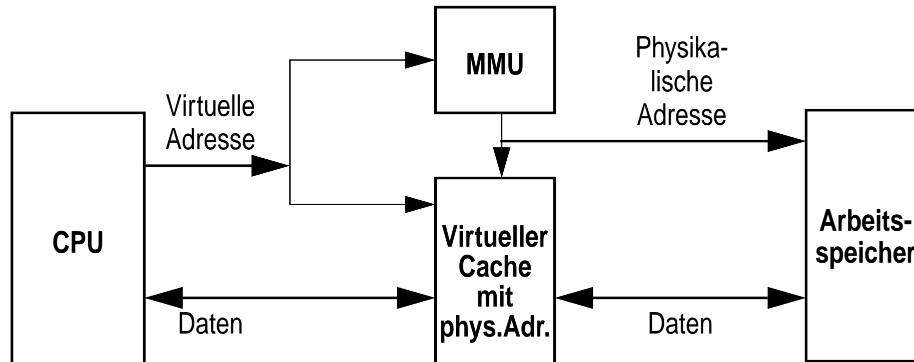
- Mehrdeutigkeit (ambiguity):
  Zwei identische virtuelle Adressen zeigen auf verschiedene Speicherbereiche
- Bedeutungsgleichheit (alias):
  Verschiedene virtuelle Adressen zeigen auf den gleichen Speicherbereich

◆ Lösung: Systemsoftware kann Probleme mit viel Aufwand verhindern

Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

1 Grundlagen (6)

- Virtuelle Caches mit physikalischen Adressen als Tags

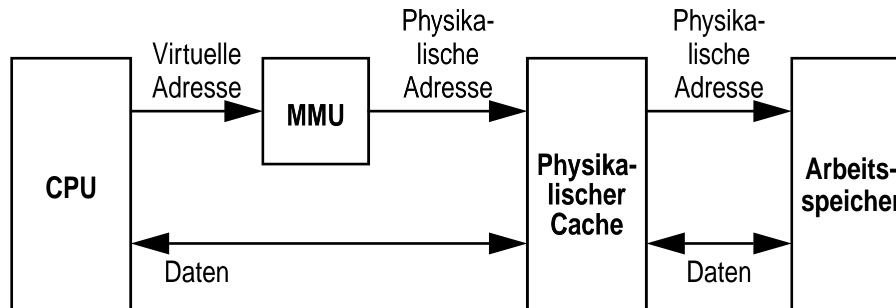


- ◆ Vorteil:
  - Keine Mehrdeutigkeiten mehr
- ◆ Nachteile:
  - Bedeutungsgleichheiten noch möglich (➜OS Unterstützung nötig)
  - Suchgeschwindigkeit durch Umsetzgeschwindigkeit der MMU begrenzt
  - Problematischer Einsatz in Multiprozessoren mit gemeinsamem Speicher (Welche Zeile soll invalidiert werden?)

Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

1 Grundlagen (7)

Physikalische Caches



◆ Vorteile

- Absolute Transparenz aus Sicht des Prozessors
- Keine performancekritische Systemunterstützung nötig
- Multiprozessoren mit gemeinsamem Adreßraum können mit einem Cache-Kohärenzprotokoll in Hardware aufgebaut werden.

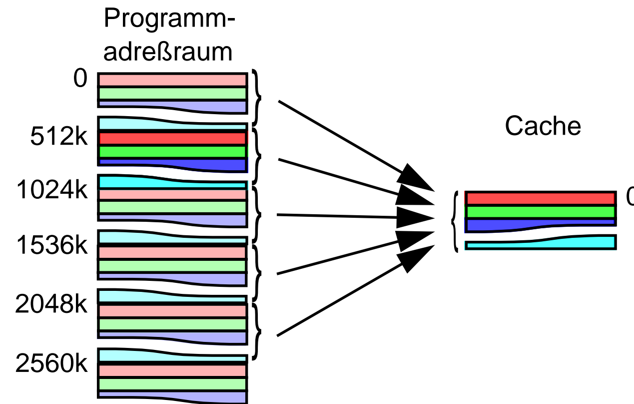
◆ Nachteil:

- Bei jedem Zugriff ist Adreßumsetzung durch MMU erforderlich

© Frank Bellosa, Univ. Erlangen-Nürnberg, IMMD IV, 1999
Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

2 Zugriffskonflikte

- Virtueller Cache (z.B. PA-RISC 1.1 CPUs in Convex SPP 1600)



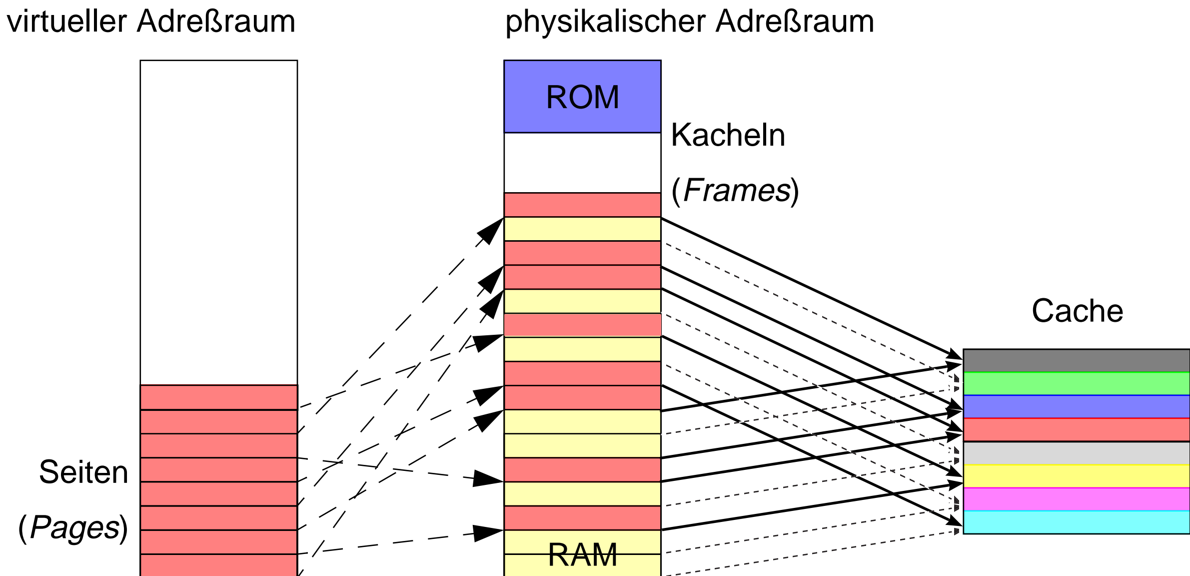
◆ Datenstrukturen, bei denen die Differenz der Adreßbereiche einem Vielfachen der Cachegröße enspricht, werden auf die selbe Cachezeile abgebildet.

◆ Virtuelle Caches mit physikalischen Tags werden häufig als first-level Caches eingesetzt (z.B. UltraSPARC II 16 KB direct mapped).
➜ siehe Übung

Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

2 Zugriffskonflikte (2)

- Physikalische Caches
  - ◆ Seitenkonflikte durch zufällige Anforderung von physikalischem Speicher:



Zusammenhängender virtueller Speicher wird in der Regel auf beliebige freie physikalische Seiten abgebildet

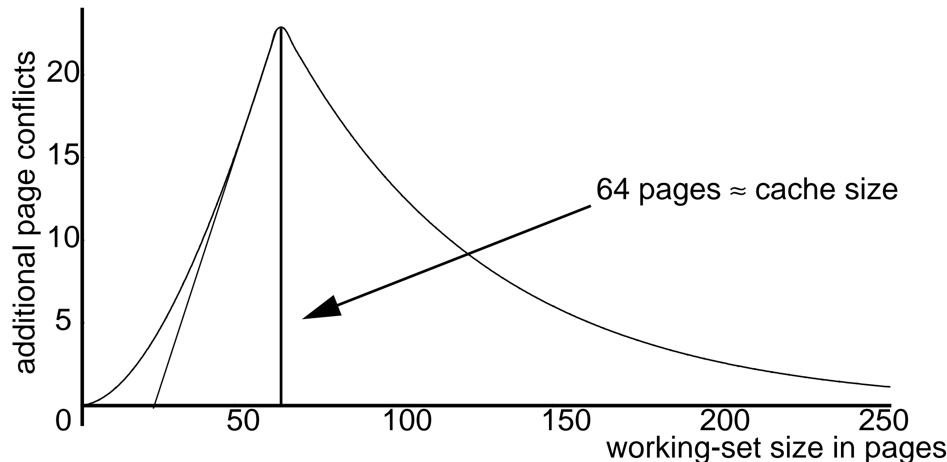
Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

2 Zugriffskonflikte (3)

Physikalische Caches (2)

- Auswirkungen des "*random page coloring*":
  - ➜ Es kommt zu Konflikten beim Cache-Zugriff
  - ➜ Nur ein Teil des Caches wird ausgenutzt
  - ➜ Erhebliche Laufzeitschwankungen!

$$X(\text{p in bin}) = \binom{P}{p}\left(\frac{1}{C}\right)^{p}\left(1-\frac{1}{C}\right)^{P-p}$$

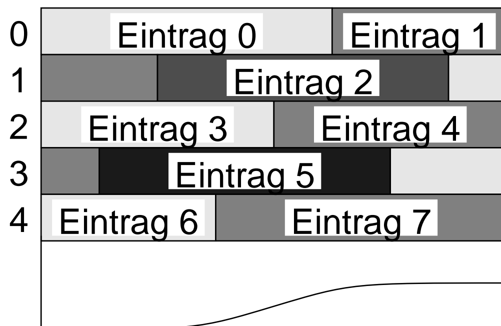


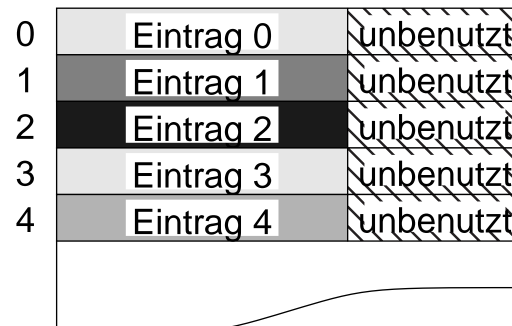
- Abhilfe: "Frischen" Speicher anfordern und im Speicher festhalten (mlock)

Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

2 Zugriffskonflikte (4)

- Einfluß der Zeilenlänge des Caches auf die Datenstrukturen
  - ◆ Problem: Mehrfache Fehlzugriffe durch zeilenübergreifende ’non aligned’ Datenstrukturen
  - ◆ Lösung: Auffüllen der Strukturen auf Vielfache der Zeilenlänge (Padding)





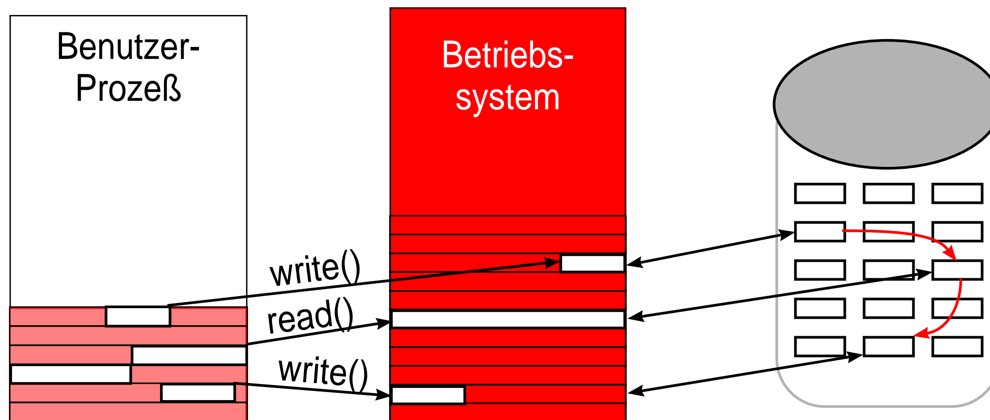
© Frank Bellosa, Univ. Erlangen-Nürnberg, IMMD IV, 1999

Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

C.3 Ein/Ausgabe

1 "Klassisches" Lesen/Schreiben in Dateien

- open/seek/read/write Operationen
- Kopieren der Daten in den Kern
- Blockpufferung der Plattenblöcke im Betriebssystem

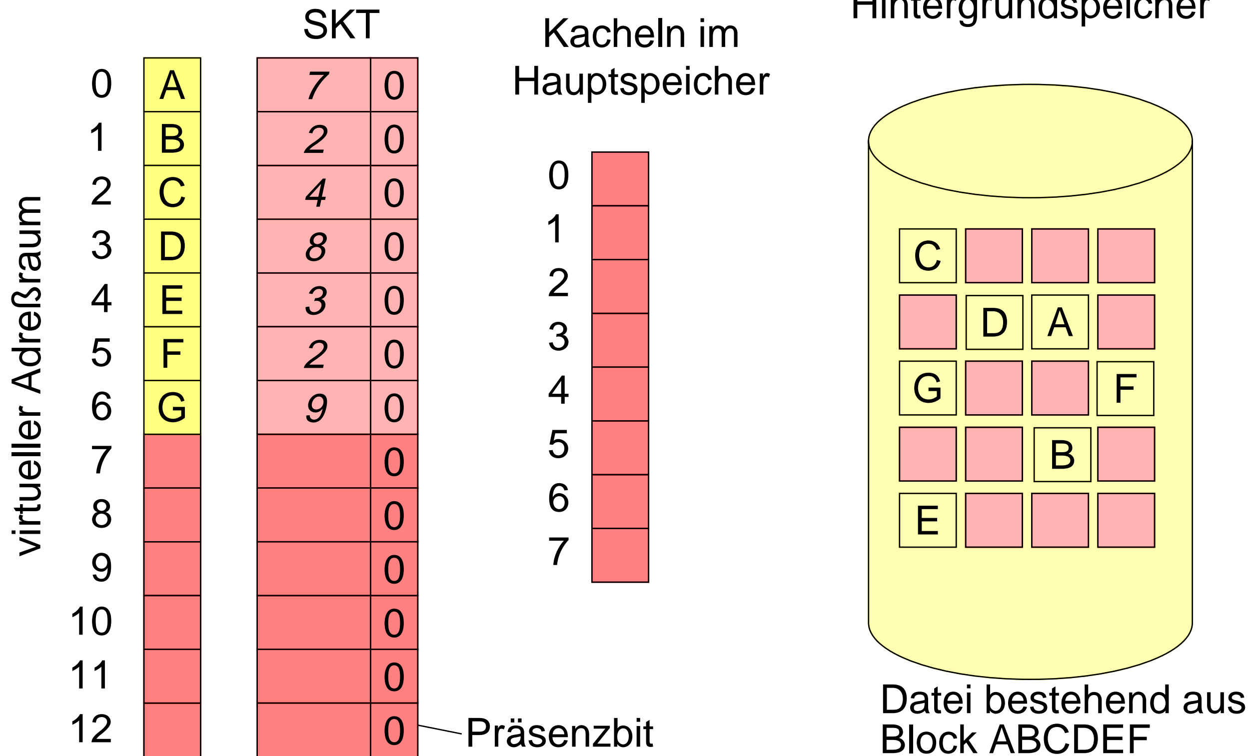


- Interessenkonflikt zwischen Speicherverwaltung und Buffercache

Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

2 Speicherabgebildete Dateien

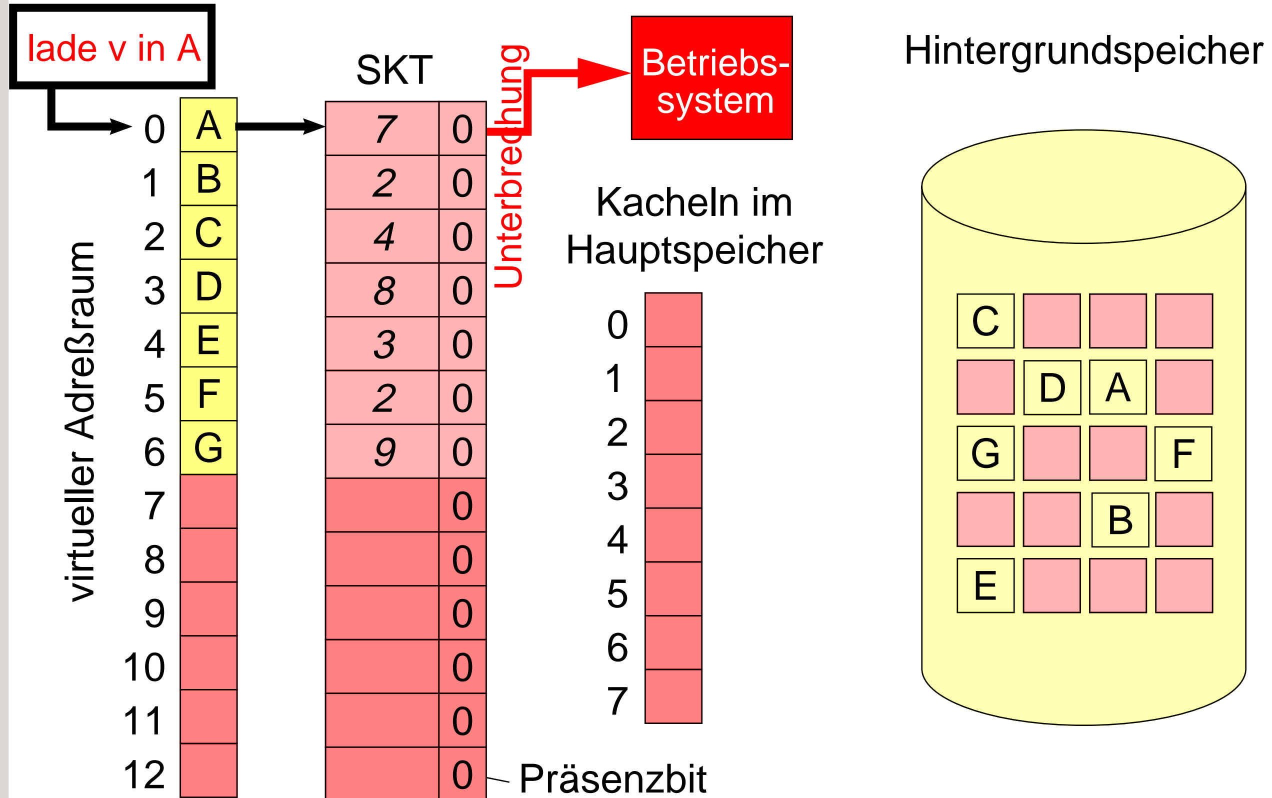
- Abbilden einer Datei auf virtuelle Seiten



Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

2 Speicherabgebildete Dateien (2)

■ Reaktion auf Seitenfehler



Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

2 Speicherabgebildete Dateien (3)

- Einlagerung

lade v in A

virtueller Adreßraum

Seite	Inhalt
0	A
1	B
2	C
3	D
4	E
5	F
6	G
7	
8	
9	
10	
11	
12	

SKT

Kachel	Präsenzbit
7	0
2	0
4	0
8	0
3	0
2	0
9	0
	0
	0
	0
	0
	0
	0

Präsenzbit

Betriebs-system

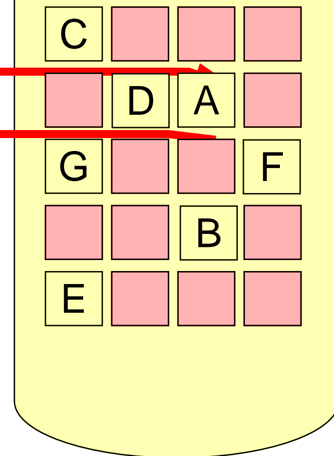
Kacheln im Hauptspeicher

Kachel	Inhalt
0	
1	
2	A
3	
4	
5	
6	
7	

Einlagern der Seite

Hintergrundspeicher

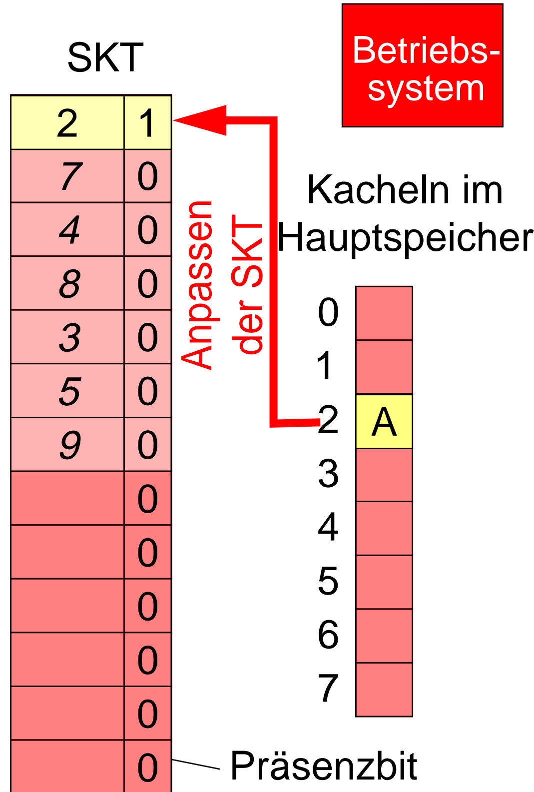
Ermitteln der ausgelagerten Seite



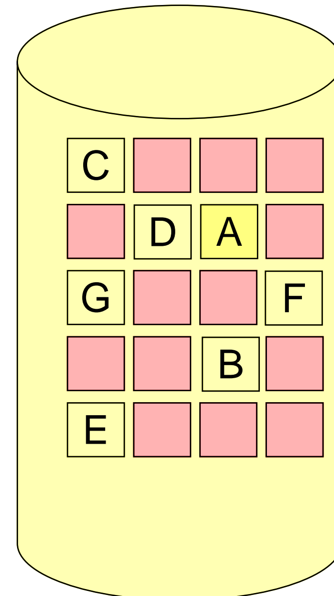
Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

2 Speicherabgebildete Dateien (4)

- Einblenden der physikalischen Seite

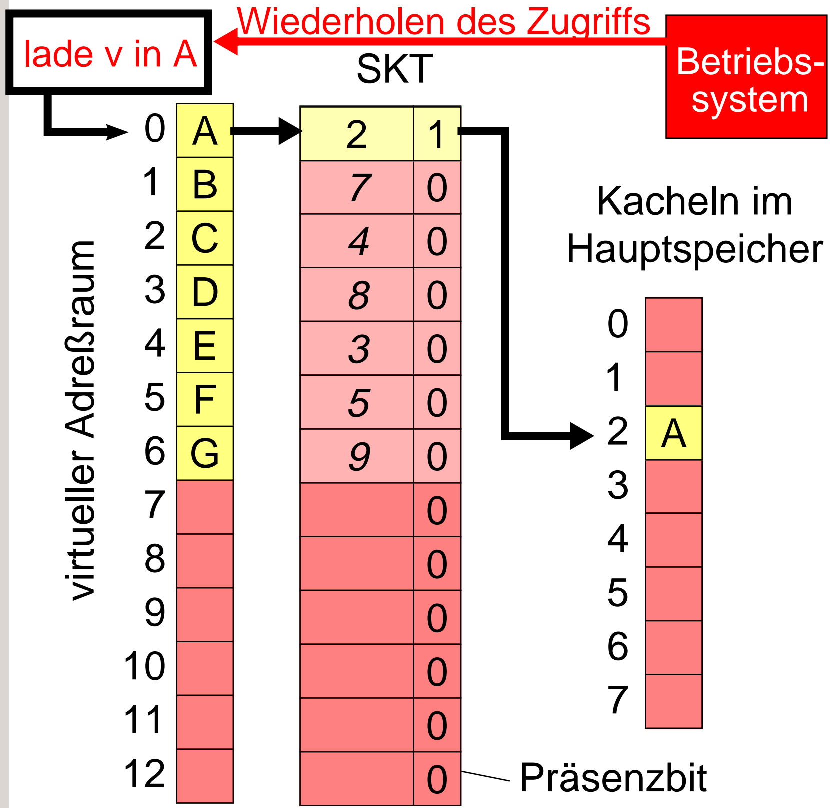


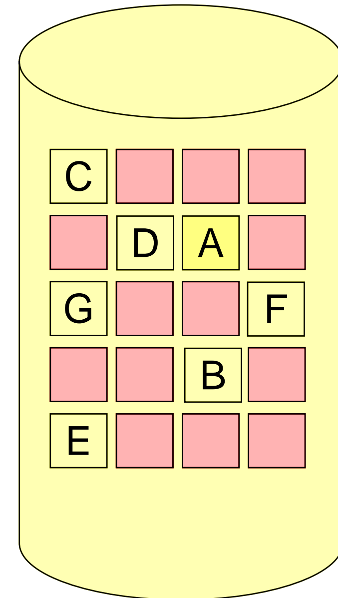
Hintergrundspeicher



Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

- Zugriff auf Seite





Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

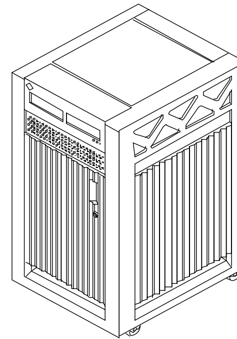
2 Speicherabgebildete Dateien (6)

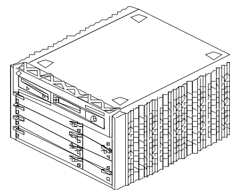
- Aufrufe: **`mmap(), munmap(), madvise(), msync()`**
- Vorteile:
  - ◆ Kein Kopieren der Daten vom Benutzer in den Kernadreßraum
  - ◆ Kein Interessenkonflik zwischen Seitenadressierung und E/A Puffern
  - ◆ Wesentliche Reduktion der Systemaufrufe (kein read/write)
  - ◆ Angabe von Zugriffsstrategien (NORMAL, RANDOM, SEQUENTIAL, WILLNEED, DONTNEED)
  - ◆ Einfaches Adressierung der Daten innerhalb der Datei über Adreßpointer
  - ◆ Mehrere Prozesse können (ohne Kohärenzproblem) auf der gleichen Datei abeiten.

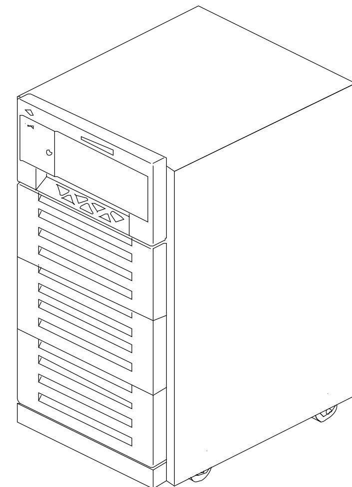
Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

D SUN Enterprise X000

- Einsatz am RRZE/IMMD
  - cssun: Enterprise 4000
    8 CPUs, 8 GByte
  - faui40: Enterprise 4000
    4 CPUs, 1 GByte
  - faui09: Enterprise 3000
    4 CPUs, 1 GByte
  - faui01: Enterprise 3000
    2 CPU,512 MByte

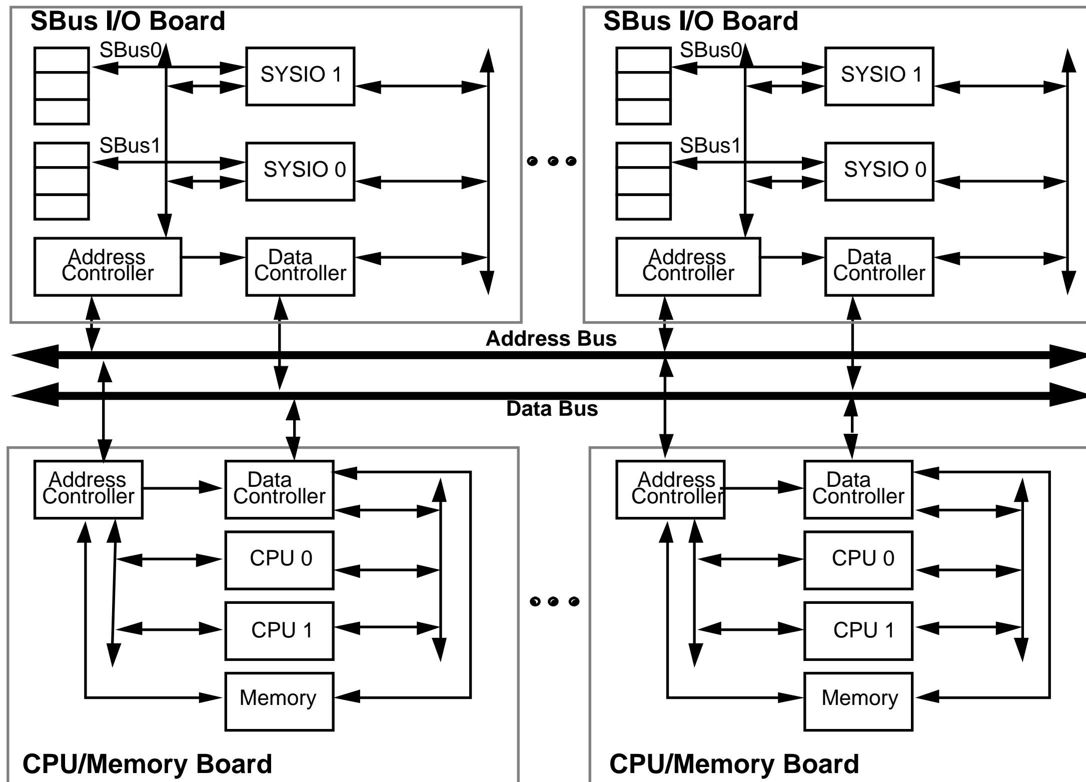






Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

D.1 Systemarchitektur



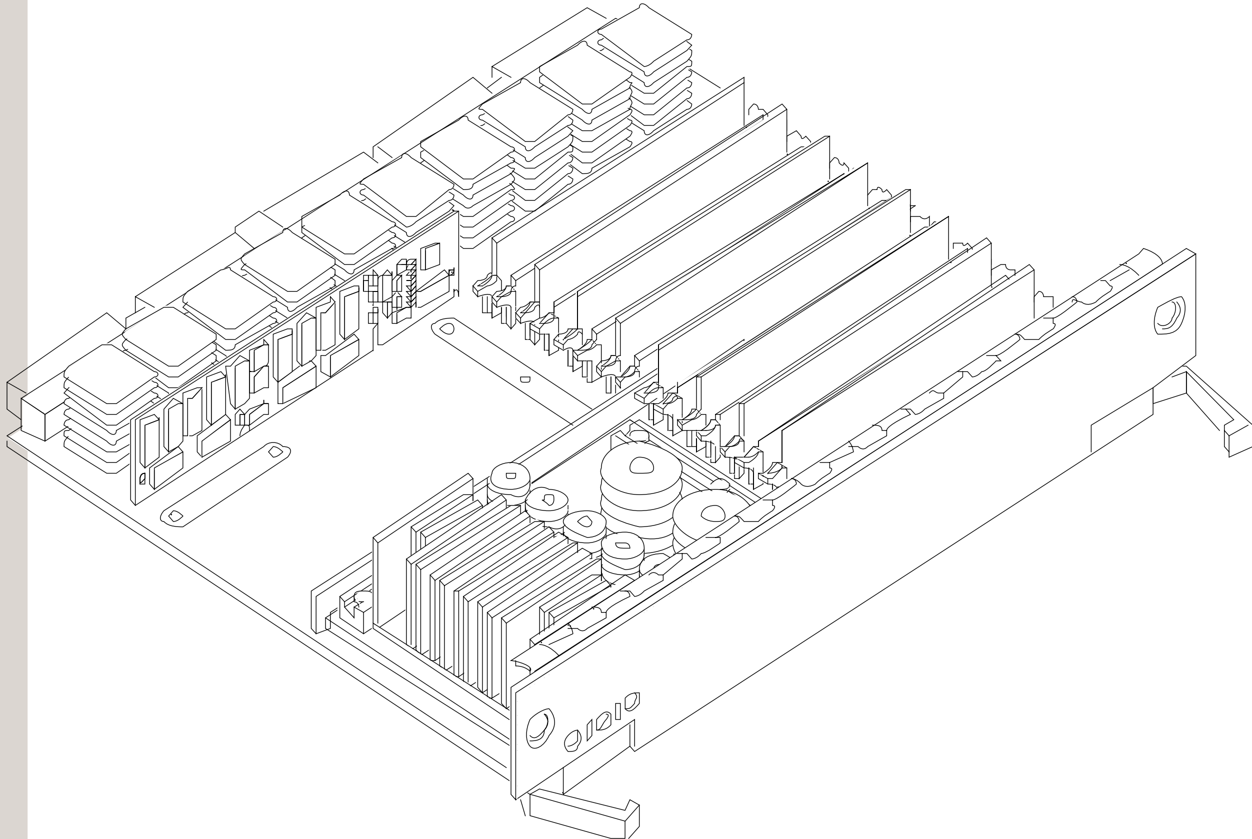
© Frank Bellosa, Univ. Erlangen-Nürnberg, IMMD IV, 1999
Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

1 Gigaplane Bus

- Sehr schneller (mit 83.3/100 MHz getakteter) und breiter (256 Bits) Bus auf der *Centerplane*.
- Kann eine vollständige Cacheline (64 Bytes) in 2 Zyklen transferieren, das ist eine Datentransferrate von 2.672 GB/s.
- Kein Multiplexen von Adressen und Daten, kein Arbitrierungsoverhead, daher extrem niedrige Latenzzeit.
- “Packet-switched” Architektur ermöglicht hohen Durchsatz.
- Die elektrischen Eigenschaften limitieren den Bus auf 16 Anschlüsse. Deshalb wird innerhalb der einzelnen Boards über den standard UPA (*Ultra Port Architecture*) Bus weiterverzweigt.

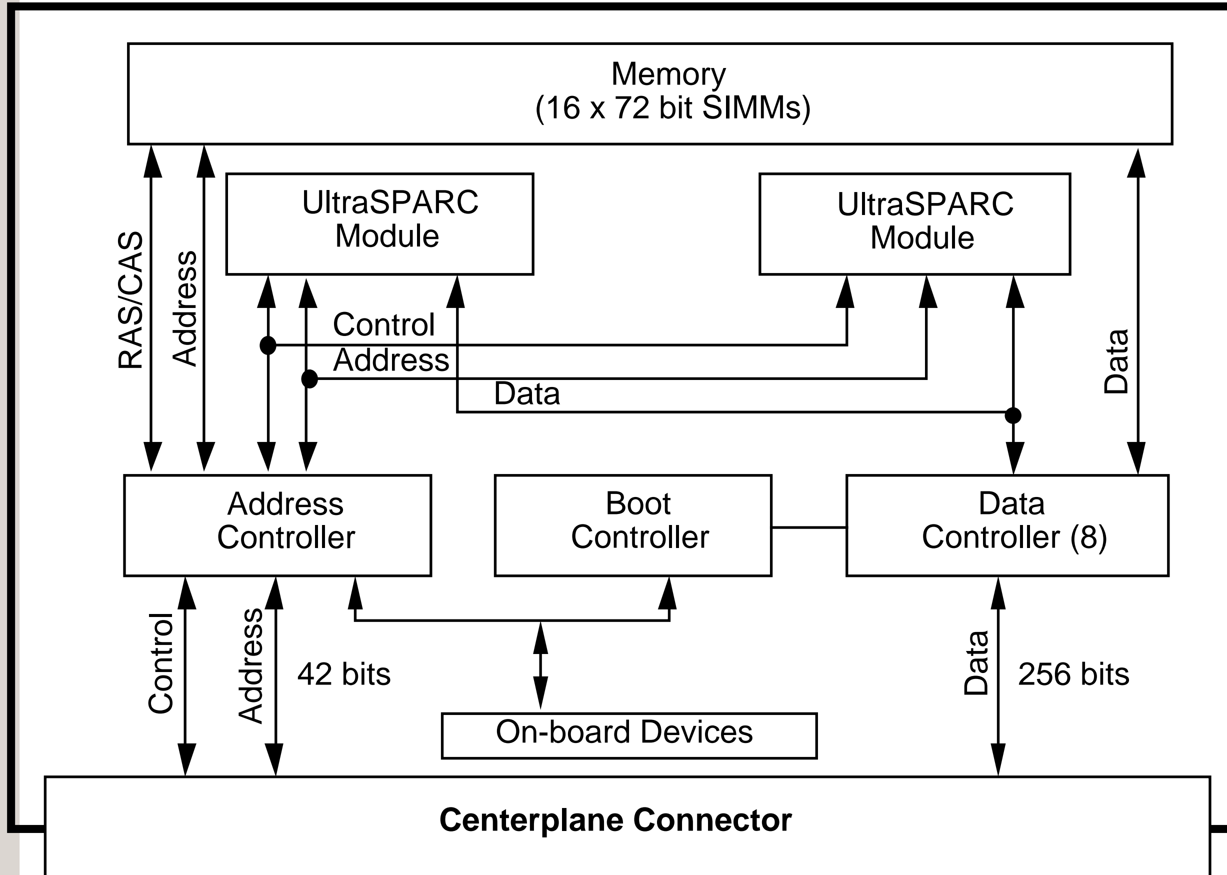
Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

2 CPU/Memory Modul



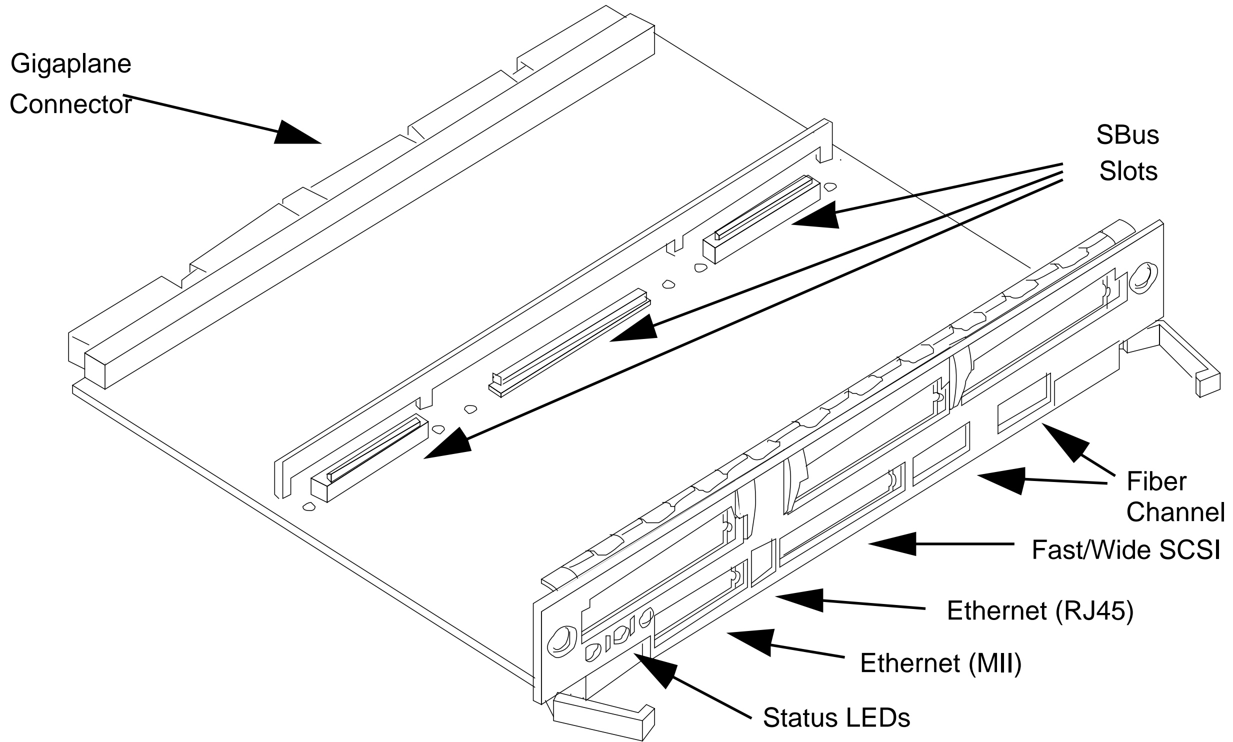
Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

2 CPU/Memory Module (2)



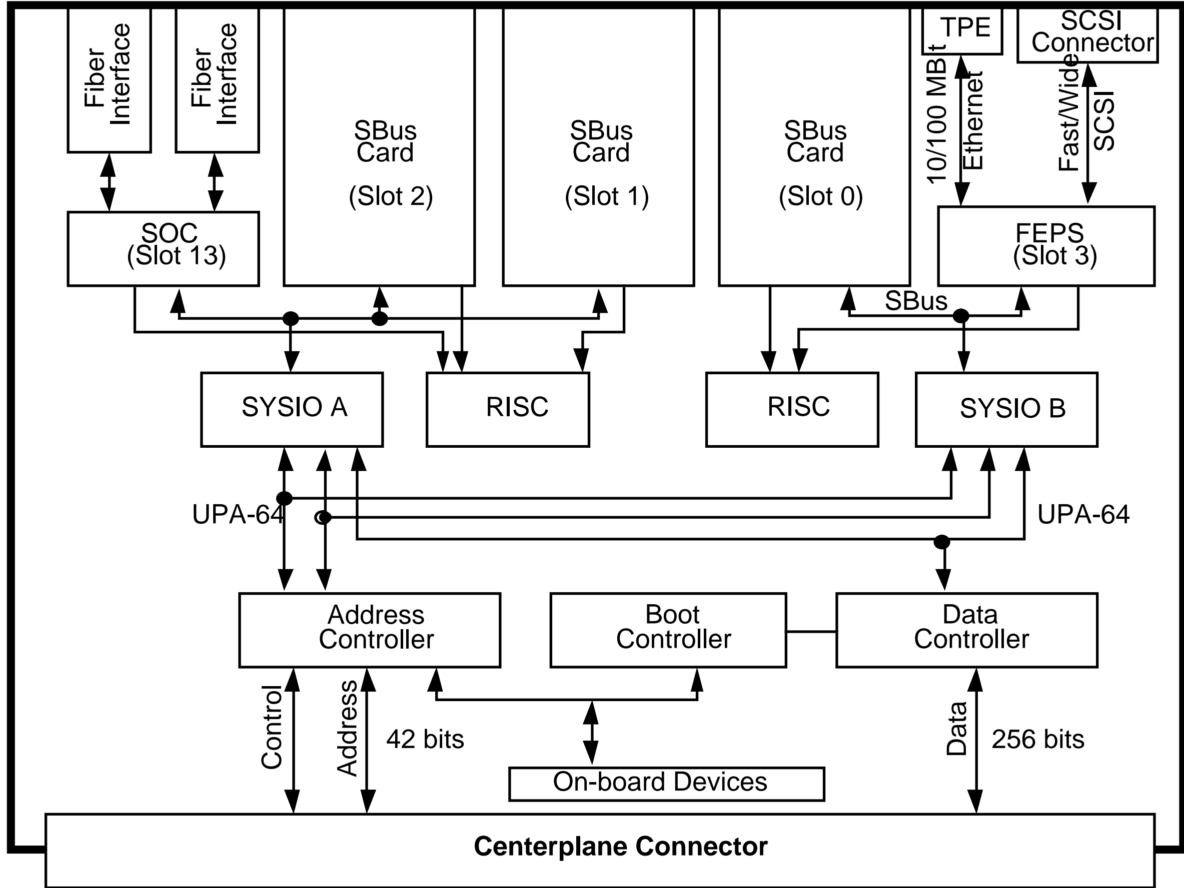
Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

3 I/O Board



Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

3 I/OBoard (2)



Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

E Pthreads

E.1 Grundlagen

- Der Name steht für POSIX Threads.
  POSIX = Portable Operating System Interface)
- Verabschiedet im Juni 1995 als POSIX Standard 1003.1c
  - Leider ist der Standard nicht im Netz erreichbar, sondern muß für viel Geld bei IEEE erworben werden.Pthreads sind standardisiert und laufen deshalb unter fast jedem OS.
  - Pthreads sind unter fast jedem OS verfügbar.
- Bücher zur Vorlesung
  - “Pthreads Programming”. O’REILLY *Nutshell* Buch
  - “Threadtime, The Multithreaded Programming Guide”, Prentice Hall

Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

1 Prozeßmodell

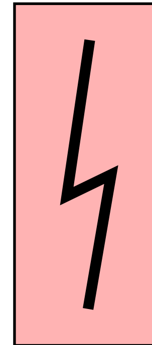
■ Der einfädige (single-threaded) Prozeß

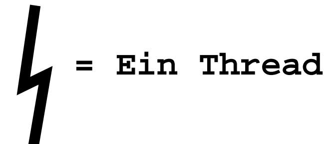
Program "hello.c"

```
#include <stdio.h>
main()
{
 printf("hello PPS\n");
}

$ cc -o hello hello.c
$ ./hello
```

Process



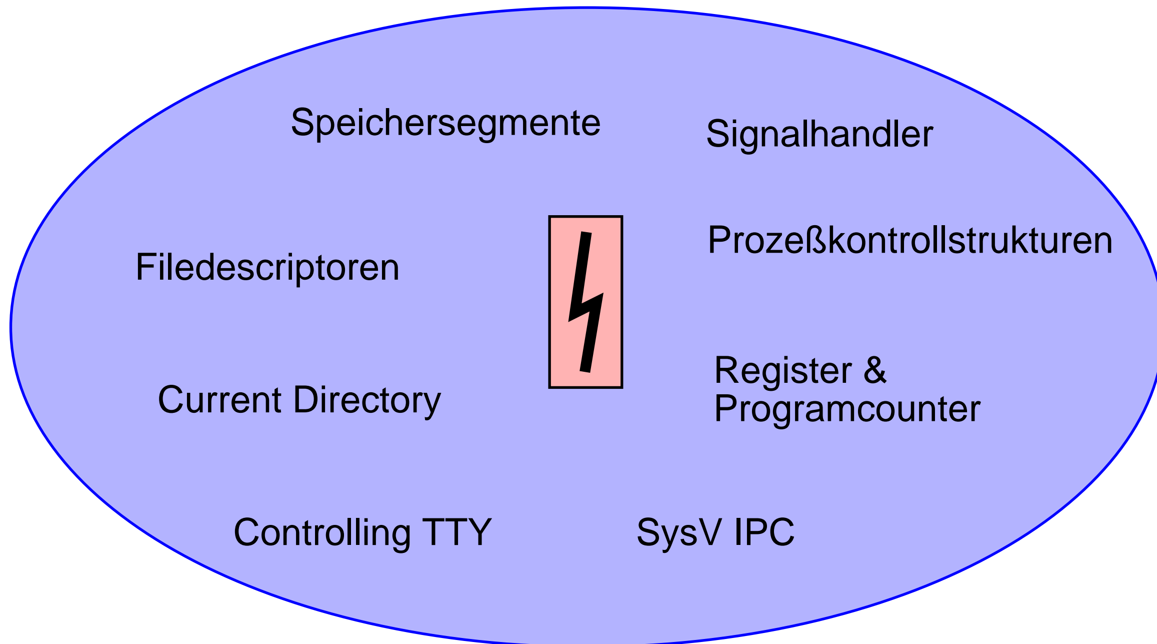


◆ Ein Prozeß ist die Instanz eines Programmes in Ausführung.

Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

1 Prozeßmodell (2)

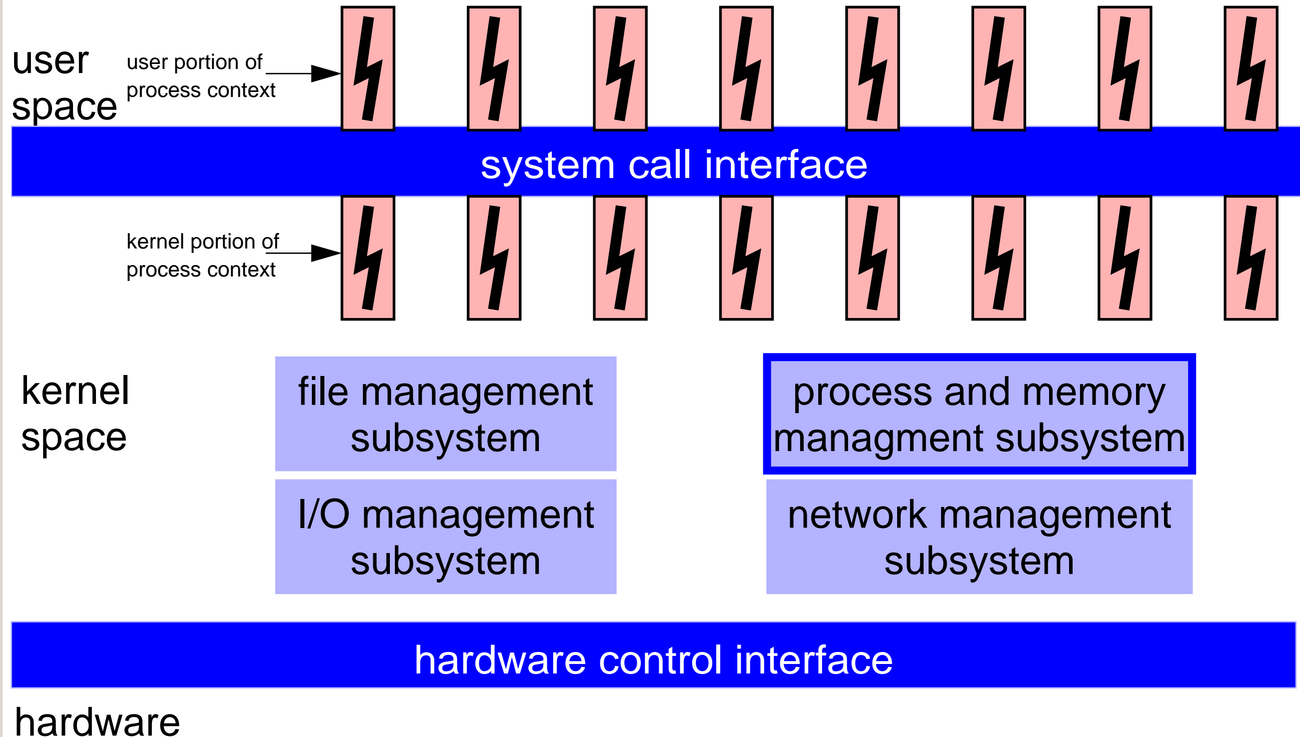
- Aufbau eines UNIX Prozesses:



Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

1 Prozeßmodell (3)

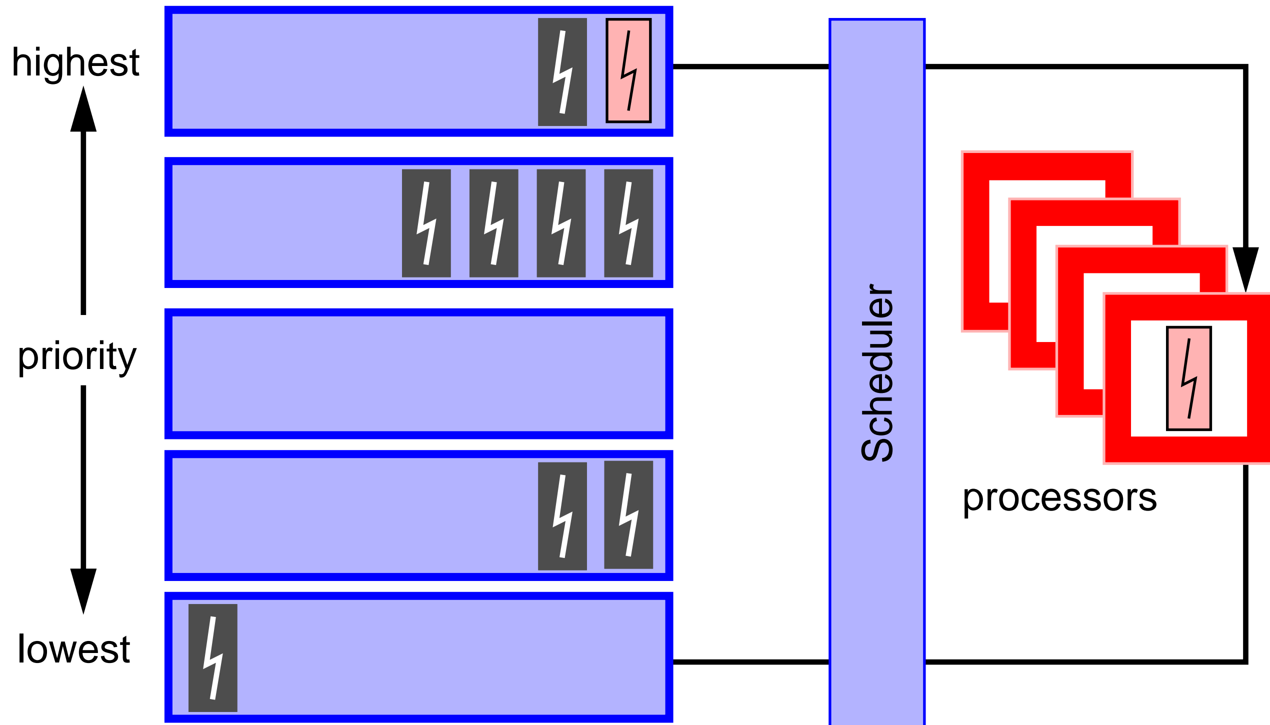
- Betriebssystem-Architektur für das Prozeßmodell



Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

1 Prozeßmodell

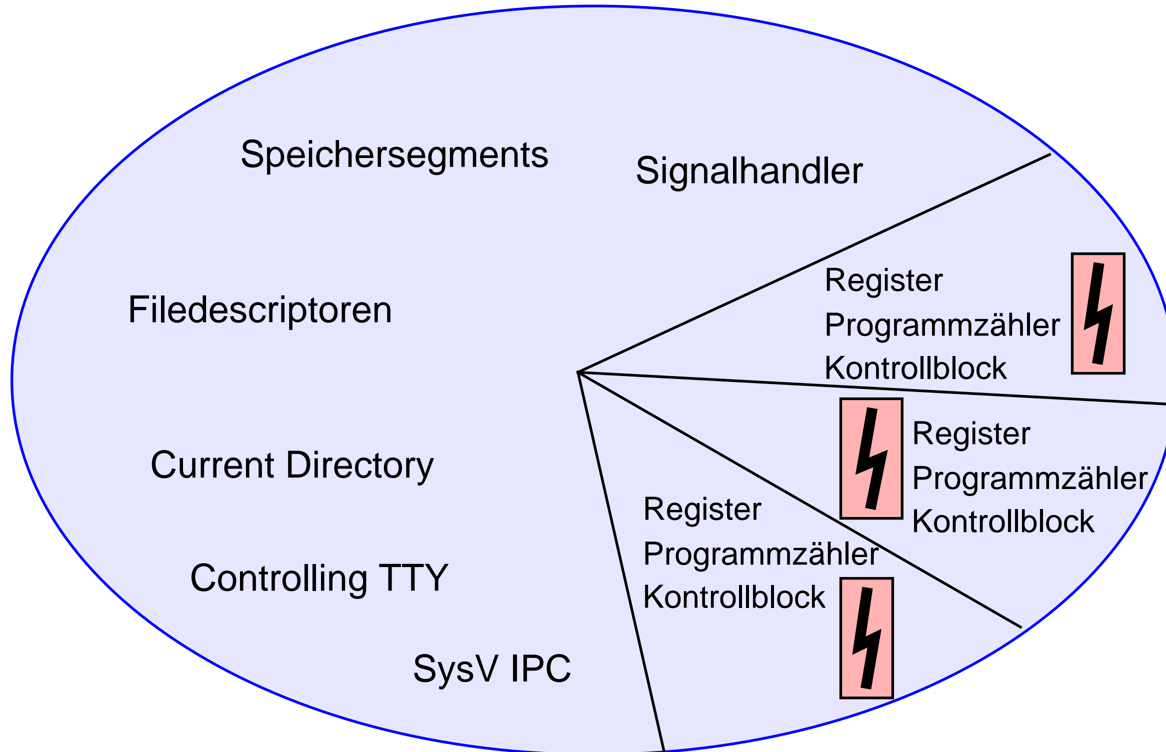
■ Scheduling



Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

2 Threadmodell

- Aufspaltung eines Prozesses in mehrere Threads



Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

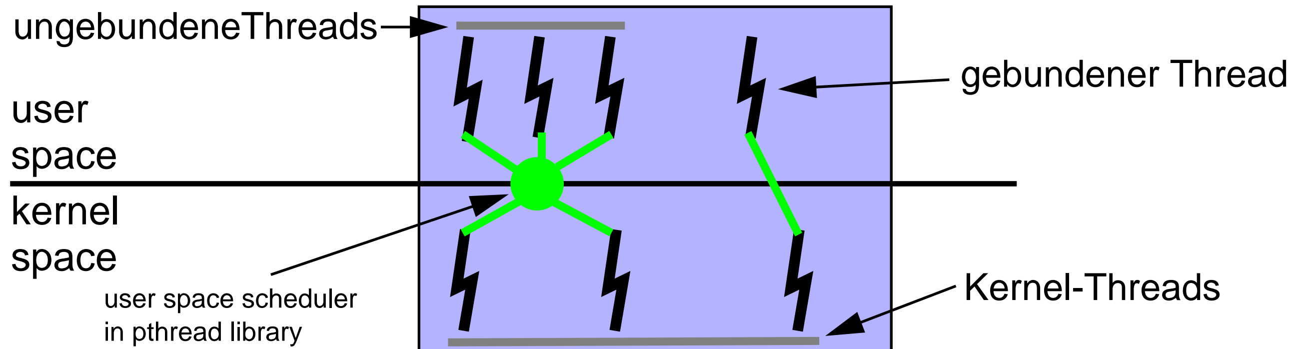
2 Threadmodell (2)

- Der Einsatz von Threads anstelle von Prozessen ist aus folgenden Gründen sinnvoll:
  - ◆ Die Prozeßerzeugung durch `fork()` ist sehr zeitaufwendig, da ein neuer Prozeßkontext angelegt werden muß.
  - ◆ Ein neuer Thread verbraucht weniger Systemressourcen als ein neuer Prozeß.
  - ◆ Der Scheduler kann wesentlich schneller zwischen zwei Threads eines Prozesses umschalten, da die gleichen Prozeßresourcen (insbesondere der gleiche Adreßraum) verwendet werden.
    ➜ Bessere Antwortzeiten (Beispiel: Apache vs. Netscape Web-Server)
  - ◆ SysV-IPC Mechanismen sind kompliziert, nicht an Prozesse gebunden (sie bleiben nach Prozeßende bestehen) und langsam (da ein Kernelaufruf getätigt werden muß).

Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

2 Threadmodell (3)

- Gebundene und Ungebundene User-Threads

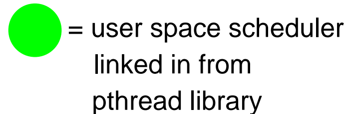


- ◆ Gebundene User-Threads werden direkt an Kernel-Threads gebunden. Aus Sicht der gebundenen User-Threads stellen Kernel-Threads virtuelle Prozessoren dar. Die Kernel-Threads konkurrieren mit allen Kernel-Threads des Systems um die verfügbaren physikalischen Prozessoren.
- ◆ Ein ungebundener User-Thread kann von jedem Kernel-Thread eines Prozesses ausgefüht werden. Ungebundene User-Threads konkurrieren um die Kernel-Threads des Prozesses. Die Zuweisung erfolgt durch die Thread-Bibliothek (Pthreads Library).

Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

2 Threadmodell (4)

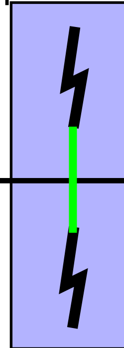
■ Many-to-One (M x 1)



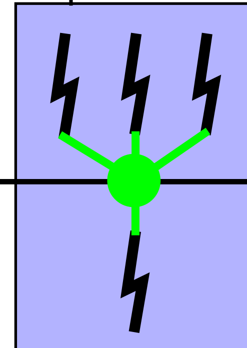
user space

kernel space

single-threaded process



multithreaded process



M x 1

Vorteile:

- Schnelle Thread Management Operationen
- Sparsamer Gebrauch von Systemressourcen
- Nebenläufiges Programmiermodell, das ohne Modifikation auch auf mehreren Prozessoren ablaufen kann (1 x 1, M x n Modell)

Nachteile:

- Keine echte Paralellität
- Verklemmung in Systemaufrufen möglich!
- Signalbehandlung ist kritisch
- Profiling einzelner Threads nicht möglich
- Debugging einzelner Threads nicht möglich

© Frank Bellosa, Univ. Erlangen-Nürnberg, IMMD IV, 1999

Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

2 Threadmodell (5)

■ One-to-One (1 x 1)

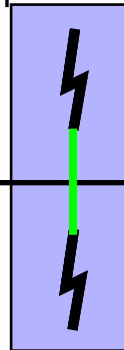
= user space scheduler linked in from pthread library

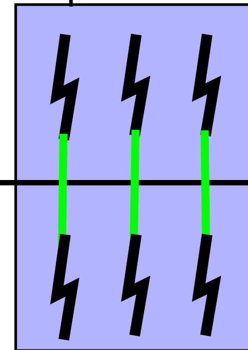
single-threaded process

multithreaded process

user space

kernel space





1 x 1

Vorteile:

- Echte parallele Ausführung der Threads im Multiprozessor möglich
- Profiling einzelner Threads möglich
- Debugging einzelner Threads möglich

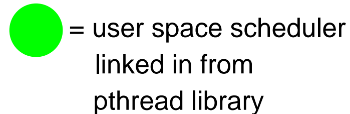
Nachteile:

- Aufwendige Threadverwaltung im OS-Kern
- Hoher Verbrauch an Systemressourcen

Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

2 Threadmodell (6)

Many-to-Many (M x N)

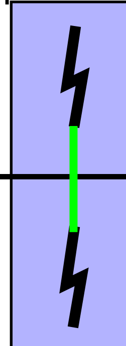


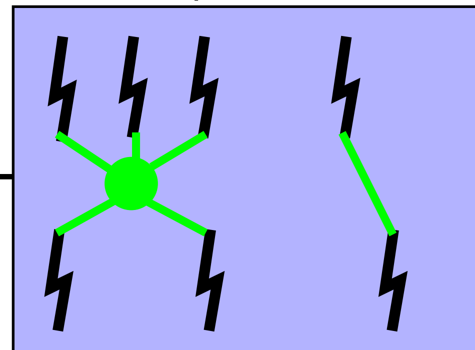
single-threaded process

multithreaded process

user space

kernel space





M x N

Vorteile:

- Flexibles Modell
- Effiziente parallele Ausführung
- Verklemmung an Systemaufrufen kann durch Nachstarten von Kernel-Threads aufgelöst werden.

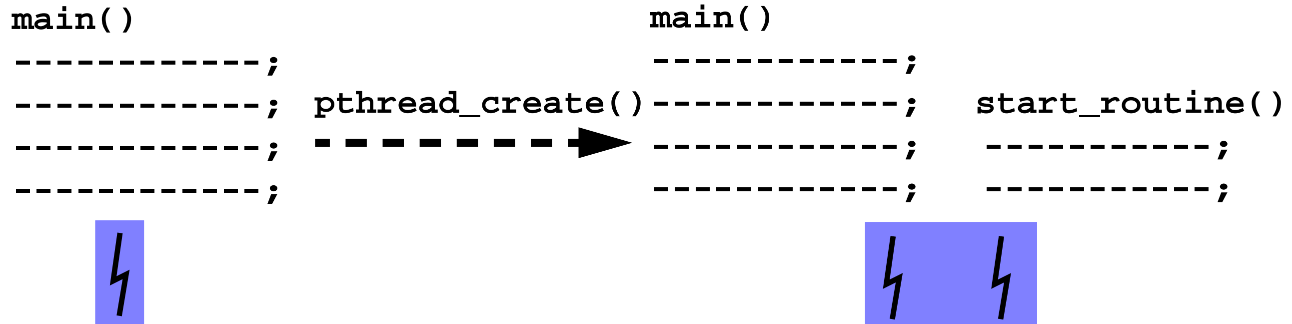
Nachteile:

- Scheduling auf zwei Ebenen (User- und Kernel-Ebene) steigert die Komplexität.
- Profiling und Debugging ungebundener Threads ist kritisch.

Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

E.2 Basic Pthread Management

1 Erzeugung



◆ Ein neuer Thread kann durch

```
pthread_create(pthread_t *thread,
               const pthread_attr_t *attr,
               void *(*start_routine)(void *),
               void *arg)
```

erzeugt werden. Der erzeugte Thread führt als erstes die Funktion `start_routine(arg)` auf. `thread` kann vom Vaterthread als *Handle* auf den erzeugten Thread verwendet werden.

Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

2 Thread-Attribute

◆ Über das `attr` Argument können die Attribute des erzeugten Threads modifiziert werden. Ist `attr == NULL`, werden die Standardattribute gesetzt.

- Initialisieren des Attribut-Objekts:

```
ret_val = pthread_attr_init (&attr);
```

- Setzen der Stack-Größe

```
ret_val = pthread_attr_setstacksize(&attr,stacksize);
```

- Lesen der Stack-Größe

```
ret_val = pthread_attr_getstacksize(&attr,&stacksize)
```

◆ Es gibt noch weiter Parameter (Stackadresse, Joinable/Detached, ...).
➜ **`man pthread_create`**

Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

3 Terminierung

- Selbst-Terminierung:
  - Ein Thread kann sich durch `void pthread_exit(void *retval)` beenden. Dies geschieht automatisch, wenn der Thread aus der Funktion `start_routine` zurückspringt.

- Warten auf die Terminierung eines Thread
  - Mit der Funktion
    `pthread_join(pthread_t thread, void **retvalp)`
    kann der Vaterthread auf die Beendigung eines erzeugten Threads warten.
    `thread` ist dabei der durch den Aufruf von `pthread_create` erzeugte Handle.
    Über `retvalp` wird der Exitstatus des Threads zurückgeliefert. Interessiert der Status nicht, kann als `retvalp` auch `NULL` übergeben werden.

Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

4 Rückgabewerte

◆ Die Pthread Funktionen weichen von ihrer Aufrufsemantik von den Standard Kernaufrufen ab. Zwar wird bei einem erfolgreichen Aufruf auch `0` zurückgeliefert, im Fehlerfall wird aber auf die Verwendung von `errno` verzichtet und der Fehlercode direkt zurückgeliefert.

◆ Die anderen Systemaufrufe setzen immer noch die `errno` Variable. Die Fehlerwerte der Pthreads Funktionen lassen sich über `perror()` oder `strerror()` als Fehlermeldung ausgeben.

◆ Da `errno` für jeden Thread verschieden sein muß, ist es normalerweise keine Variable mehr, sondern wird über ein `#define` auf einen Funktionsaufruf abgebildet.

Setzen des Parallelitätsgrades in SUN Solaris 2.5/6

◆ Unter Solaris muß der Parallelitätsgrad (Anzahl der Kernelthreads im M x N Modell) über `thr_setconcurrency(n)` konfiguriert werden.

◆ Das Defaultverhalten ist, daß nur **ein** Kernelthread läuft.

Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

5 Beispiel zur Thread-Erzeugung

- Beispiel 1: Multiplikation Matrix mit Vektor

```
#include <stdio.h>
#include <pthread.h>
double a[100][100], b[100], c[100];

main() {
    pthread_t tids[100];
...
    for (i = 0; i < 100; i++)
        pthread_create(tids + i, NULL, mult,(void *)(c + i));
    for (i = 0; i < 100; i++)
        pthread_join(tids[i], NULL);
...
}
```


Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

5 Beispiel zur Thread-Erzeugung (2)

◆ Multiplikationsfunktion **`mult()`**

```
void *mult(void *cp){
    int j, i = (double *)cp - c;
    double sum = 0;

    for (j = 0; j < 100; j++)
    sum += a[i][j] * b[j];
    c[i] = sum;
    return 0;
}
```


Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

5 Beispiel zur Thread-Erzeugung (3)

- Ergebnisse: 1000x1000 Matrix, `mult` *k*-fach wiederholt

k	Anzahl der Prozessoren (*nproc*)				
	–	1	2	3	4
1	0.6	1.1	0.8	0.8	0.8
2	0.9	1.3	0.9	0.8	0.8
10	2.9	3.4	2.0	1.4	1.2
100	26.3	26.7	13.3	9.2	7.1

- => Laufzeit setzt sich ungefähr folgendermaßen zusammen:
  - 0.3s Initialisierung,
  - 0.4s Starten und Beenden von 1000 Threads,
  - (k x 0.26s) /*nproc* reine Rechenzeit.

© Frank Bellosa, Univ. Erlangen-Nürnberg, IMMD IV, 1999
Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

6 Identifikation

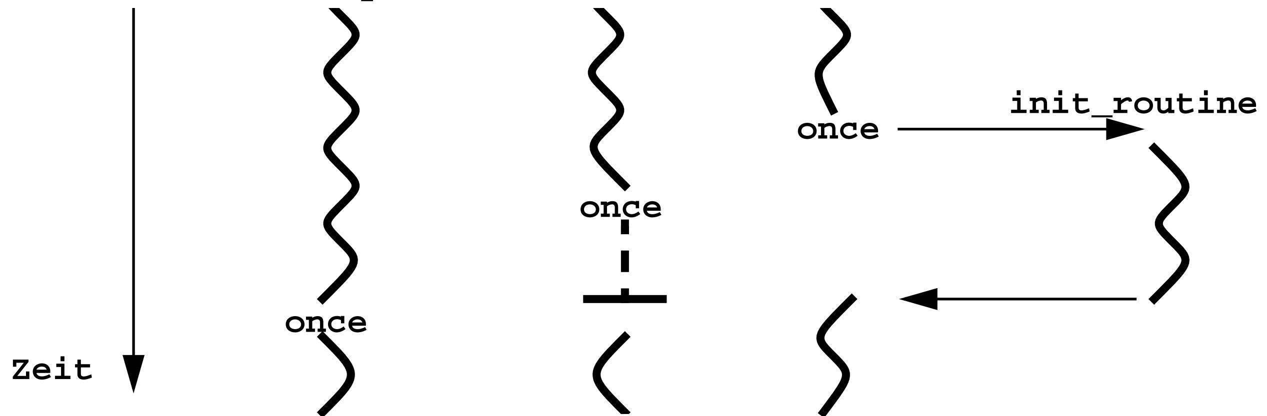
- Ein Thread kann seinen Thread-Handle über `pthread_t pthread_self(void)` erfragen.

- Es ist nicht garantiert, daß man Thread-Handles mit `==` auf Gleichheit überprüfen kann.
  ➜ Die Funktion `pthread_equal(pthread_t t1, pthread_t t2)` liefert `TRUE` zurück, wenn die beiden Threads identisch sind.

Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

7 Initialisierungen

- Initialisierungen dürfen häufig nur genau einmal durchgeführt werden.
- Die Pthread Bibliothek erleichtert diese Aufgabenstellung mit der `pthread_once()` Funktion.
- Die übergebene Funktion wird genau einmal vom dem Thread ausgeführt, der das erste Mal `pthread_once` aufruft.
- Arbeitet gerade ein Thread die Initialisierungsfunktion ab, blockieren alle anderen Threads im `pthread_once` Aufruf.



© Frank Bellosa, Univ. Erlangen-Nürnberg, IMMD IV, 1999
Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

7 Initialisierungen (2)

- Syntax:

```
pthread_once(pthread_once_t *once_block,
             void (*init_routine)(void))
```

- Der `once_block` wird von der Pthreads Bibliothek zur Speicherung von Statusinformation benötigt. Er muß *statisch* mit `PTHREAD_ONCE_INIT` initialisiert werden.

- Leider kann der Initialisierungsfunktion kein Argument mitgegeben werden.

Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

8 Threadlokale globale Variablen

- Threadlokale Variablen sind normalerweise nur Variablen, die auf dem Programmstack abgelegt wurden.

- Es ist manchmal wünschenswert, threadspezifische Variablen zu verwenden, die unabhängig von der Aufrufhierarchie (Stack) sind.

- Threadlokale Variablen werden durch globale *Keys* implementiert. Jedem Key kann ein Wert zugewiesen werden, der threadlokal gespeichert wird.
  - ◆ Ein Key kann mit der Funktion

    ```
    pthread_key_create(pthread_key_t *key,
                               void (*destructor)(void *))
    ```

    angelegt werden. Dieser Aufruf bewirkt folgendes:

    - Das System legt Speicherplatz für den Wert des Keys an.
    - Außerdem kann dem System eine Funktion mitgeteilt werden, die beim Überschreiben und Löschen eines Wertes aufgerufen werden soll. Hierdurch lassen sich auch komplizierte Strukturen als Wert speichern.

Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

8 Threadlokale globale Variablen (2)

- Soll ein Wert an einen Key zugewiesen werden, ist

  ```
  pthread_setspecific(pthread_key_t key,
                      void *value)
  ```

  aufzurufen. Hierbei wird der Wert `value` unter dem Key `key` gespeichert.

- Der gespeicherte Wert kann mit

  ```
  void *pthread_getspecific(pthread_key_t key)
  ```

  abgefragt werden.

- Wird ein globaler Key nicht mehr benötigt, sollte

  ```
  pthread_key_delete(pthread_t key)
  ```

  aufgerufen werden.

Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

E.3 Synchronisation

- Da die einzelnen Threads gleichzeitig ablaufen, kann es zu ungewollten Effekten beim Zugriff auf den Speichern kommen (sogenannte *Race Conditions*).

Beispiel: Erhöhung eines Zählers

```
int var;
void *incr(void *arg) {
  int i;
  for (i = 0; i < 10000; i++)
    var++;
  return NULL;
}
.......
pthread_create(&son1, NULL, incr, NULL);
pthread_create(&son2, NULL, incr, NULL);
pthread_join(son1, NULL);
pthread_join(son2, NULL);
printf("Result: %d\n", var);
```

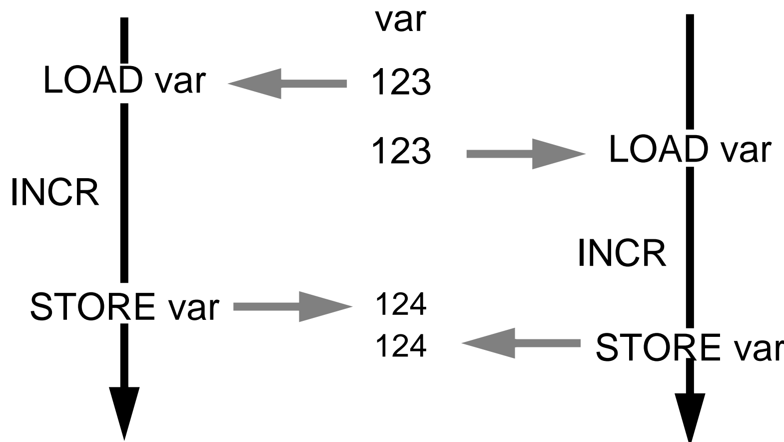

Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

E.3 Synchronisation (2)

- Dreimaliges Aufrufen des Programms auf einer SUN ergab folgende Werte:

```
Result: 13950
Result: 13881
Result: 13638
```

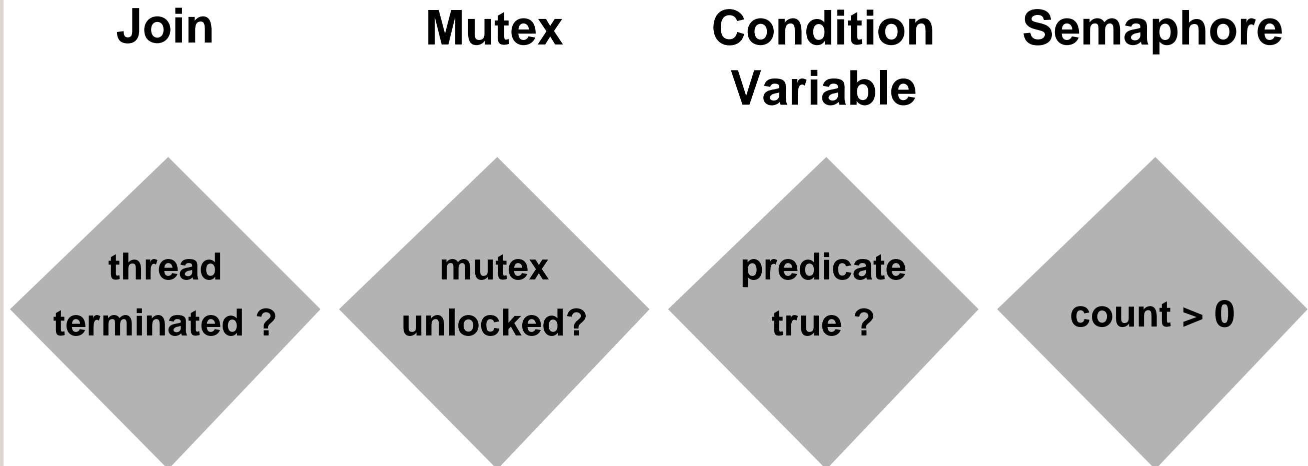
Ursache für dieses Verhalten: Die `var++` Anweisung besteht aus einer Lese- und einer Schreiboperation. Versucht der zweite Thread zwischen diesen beiden Operationen, den Zähler zu erhöhen, geht ein `var++` verloren:



Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

E.3 Synchronisation (3)

1 Mechanismen zur Thread-Koordination



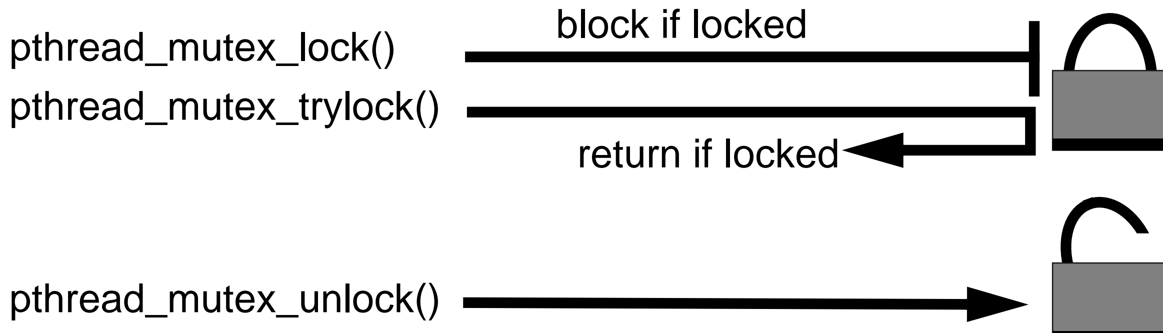
Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

2 Mutex

- Programmabschnitte, in denen sich zu einem Zeitpunkt nur ein Thread befinden darf, werden als *critical Section* bezeichnet.
  Beispiel: Erhöhung eines Zählers

- Der Pthread Standard bietet die Möglichkeit, kritsche Abschnitte durch einen *Mutex* zu sperren.
  In einem gesperrten Bereich kann sich höchstens ein Thread aufhalten.
  Will zusätzlich ein zweiter Thread den Bereich betreten, muß dieser solange warten, bis der erste Thread den Bereich "entriegelt" hat.

- Ein Mutex ist also mit einem Türschloß vergleichbar.



© Frank Bellosa, Univ. Erlangen-Nürnberg, IMMD IV, 1999

Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

2 Mutex (2)

- Erzeugen eines Mutexes:
  - Beim Kompilieren (statisch), durch
    ```
    pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
    ```
  - Beim Programmlauf (dynamisch), durch
    ```
    pthread_mutex_init(pthread_mutex_t *mutex,
                       pthread_mutexattr_t *attr)
    ```
    `mutex` muß auf einen gültigen Speicherbereich zeigen.
  - Beim dynamischen Aufruf können wie bei der Erzeugung von Threads Attribute gesetzt werden. Für einen Standardmutex kann dieses Argument `NULL` sein.

- Vernichten eines Mutexes:
  - Wird ein Mutex nicht mehr benötigt, kann dies dem System mit `pthread_mutex_destroy()` mitgeteilt werden.
    Die Pthread Bibliothek kann dann die zugeordneten Resourcen freigeben.

Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

2 Mutex (3)

- Ein Mutex kann durch
`pthread_mutex_lock(pthread_mutex_t *mutex)`
verriegelt werden. Nach diesem Aufruf ist sichergestellt, daß der aktuelle Thread als einziges den Mutex belegt hat.

- Nach dem kritischem Codeabschnitt kann mit
`pthread_mutex_unlock(pthread_mutex_t *mutex)`
der Mutex entriegelt werden.

- Soll nicht die erfolgreiche Verriegelung abgegewartet werden, kann
`pthread_mutex_trylock(pthread_mutex_t *mutex)`
verwendet werden. Diese Funktion liefert einen Fehler (**`EBUSY`**) zurück, wenn der Mutex durch einen anderen Thread belegt ist.

Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

2 Mutex (4)

- Beispiel:

```
int var;
pthread_mutex_t varlock = PTHREAD_MUTEX_INITIALIZER;
...
    for (i = 0; i < 10000; i++) {
        pthread_mutex_lock(&varlock);
        var++;
        pthread_mutex_unlock(&varlock);
    }
```


Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

3 Condition-Variablen

- Häufig muß ein Programm auf das Eintreffen bestimmter Ereignisse warten. Dieses Warten sollte nicht aktiv geschehen (*polling*), da sonst wertvolle Rechenzeit ungenutzt bleibt.

- Stattdessen sollte es eine Möglichkeit geben, daß sich Threads "schlafen legen" (*sleep*). Wenn ein anderer Thread feststellt, daß das gewünschte Ereignis eingetroffen ist, kann der schlafende Thread geweckt werden (*wakeup*).

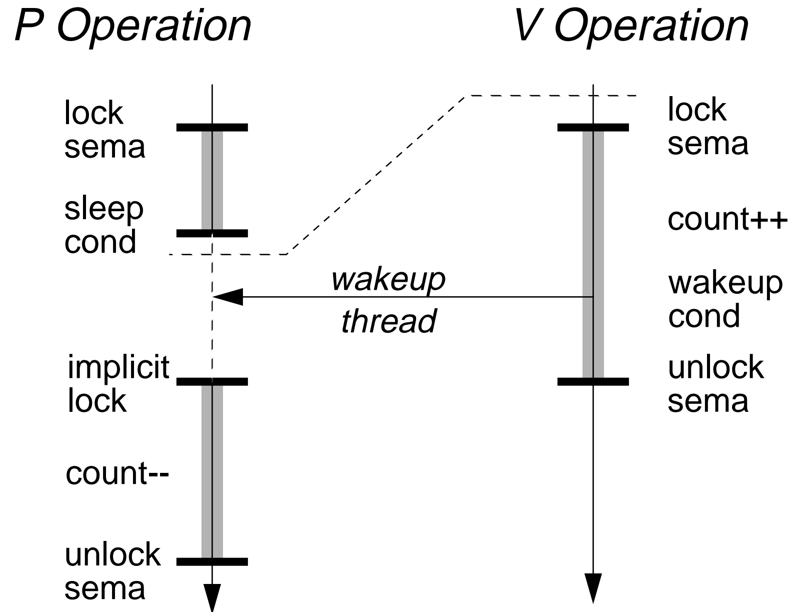
- Normalerweise bezieht sich das Ereignis auf eine Struktur, die durch ein Mutex gelockt werden muß. Deshalb muß der Sleep-Aufruf den Mutex automatisch freigeben. Nach dem Wakeup muß dann ein implizites Lock durchgeführt werden.

- Im Prinzip ist die Idee der Condition-Variablen identisch mit dem `sleep()/wakeup()` Mechanismus, der im UNIX Kern verwendet wird. Allerdings wird dort anstelle der Mutexe ein `splhi()` Aufruf verwendet, der die Interrupts sperrt.

Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

3 Condition-Variablen (2)

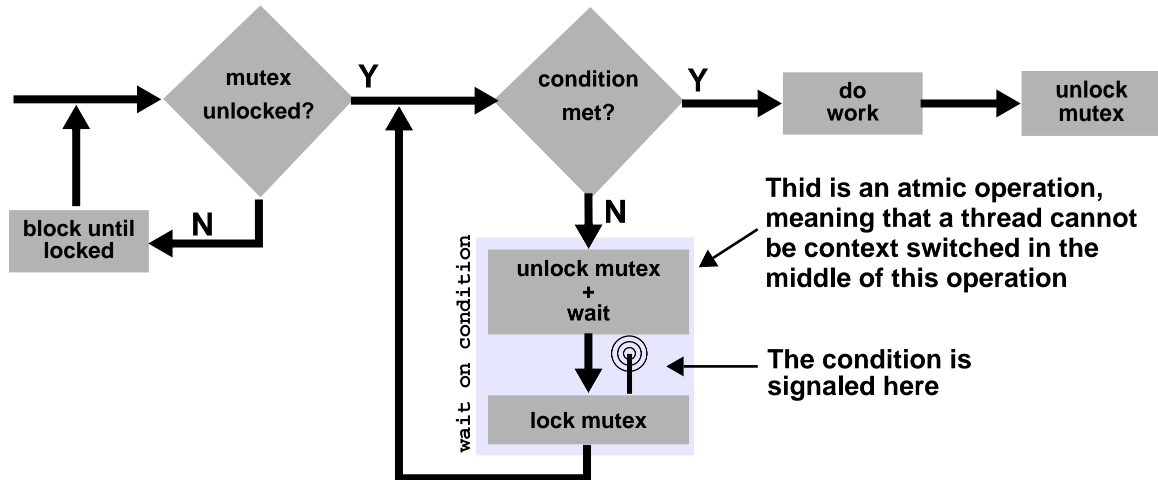
- Beispiel: P/V Semaphore.
  Die kritische Struktur ist der Zähler. Das Ereignis, auf das gewartet werden muß, ist “count > 0”.



Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

3 Condition-Variablen (3)

- Wenn mehr als ein Thread eine P-Operation durchführen kann, muß nach jedem *Wakeup* die Bedingung neu getestet werden und gegebenenfalls wieder ein *Sleep* durchgeführt werden.




```
lock associated mutex
while (predicate is not true)
        wait on condition variable
do work
unlock associated mutex
```


Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

3 Condition-Variablen (4)

Erzeugen einer Condition-Variablen

- Eine Condition-Variable wird ähnlich wie ein Mutex erzeugt:
- Statisch durch

```
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
```

- Dynamisch mit

```
pthread_cond_init(pthread_cond_t *cond,
                  const pthread_condattr_t *attr)
```

- Wie bei der Mutexerzeugung können wieder Attribute angegeben werden. Sollen die Standardattribute verwendet werden, kann `NULL` verwendet werden.

Vernichten einer Condition-Variablen

- Mit der Funktion

```
pthread_cond_destroy(pthread_cond_t *cond)
```

kann dem System mitgeteilt werden, daß diese Condition-Variable nicht mehr verwendet wird.

Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

3 Condition-Variablen (5)

Synchronisation (1):

- Mit

```
pthread_cond_wait(pthread_cond_t *cond,
                  pthread_mutex_t *mutex)
```

  kann sich ein Thread auf der Condition-Variablen `cond` blockieren. `mutex` gibt hierbei den Mutex an, der implizit freigegeben werden soll.

- Von diesem Aufruf gibt es auch eine Version mit Zeitbegrenzung: Mit

```
pthread_cond_timedwait(pthread_cond_t *cond,
              pthread_mutex_t *mutex,
              const struct timespec *abstime)
```

  kann ein Zeitpunkt angegeben werden, zu dem der Aufruf mit einem Fehler (**`ETIMEDOUT`**) abgebrochen wird.

Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

3 Condition-Variablen (6)

- Synchronisation (2):
  - Das Eintreffen eines Ereignisses kann durch
    `pthread_cond_signal(pthread_cond_t *cond)`
    signalisiert werden. Das System weckt hierbei höchstens einen Thread auf (abhängig von der Schedulingpriorität).
  - Sollen alle Threads aufgeweckt werden, die sich an der Condition-Variablen blockiert haben, ist dies mit
    `pthread_cond_broadcast(pthread_cond_t *cond)`
    möglich.
  - Beispiel in Pseudo-Code:

```
lock associated mutex
set predicate to true
signale condition variable (wake-up one or all)
unlock associated mutex
```


Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

3 Condition-Variablen (7)

- Beispiel
  - s soll eine P/V Semaphore implementiert werden.

```
typedef struct {
    pthread_mutex_t mutex;
    pthread_cond_t cond;
    int count;
} mysem_t;
```

  - Initialisierung:

```
void mysem_init(mysem_t *ms, int init) {
    pthread_mutex_init(&ms->mutex, NULL);
    pthread_cond_init(&ms->cond, NULL);
    ms->count = init;
}
```

```
Resourcenfreigabe
    void mysem_destroy(mysem_t *ms) {
        pthread_mutex_destroy(&ms->mutex);
        pthread_cond_destroy(&ms->cond);
    }
```


Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

3 Condition-Variablen (8)

◆ P Operation:

```
void mysem_p(mysem_t *ms) {
    pthread_mutex_lock(&ms->mutex);
    while (ms->count <= 0) {
    pthread_cond_wait(&ms->cond,
                      &ms->mutex);
    }
    ms->count--;
    pthread_mutex_unlock(&ms->mutex);
}
```

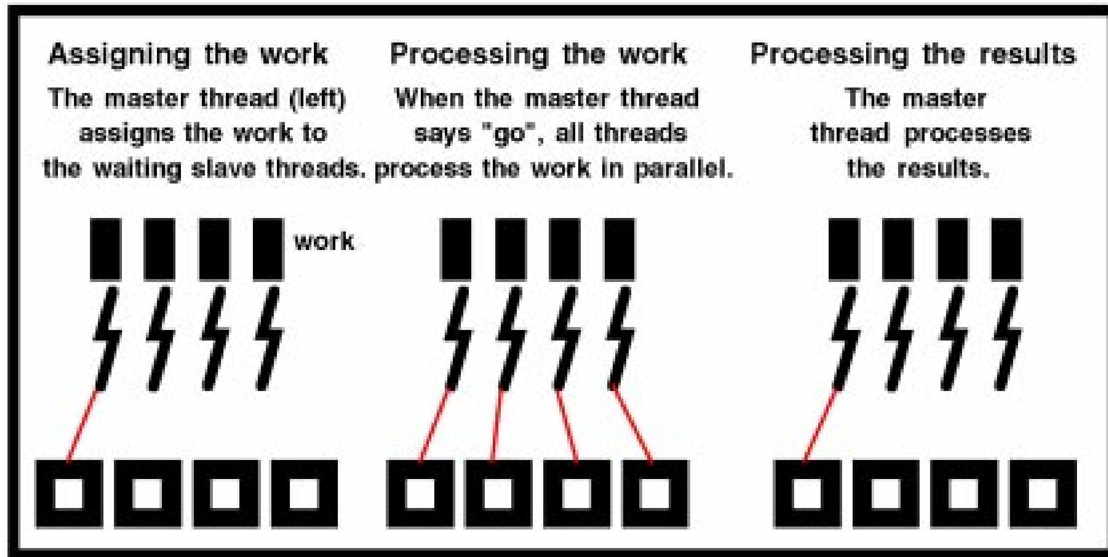
◆ V Operation:

```
void mysem_v(mysem_t *ms) {
    pthread_mutex_lock(&ms->mutex);
    ms->count++;
    pthread_cond_signal(&ms->cond);
    pthread_mutex_unlock(&ms->mutex);
}
```


Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

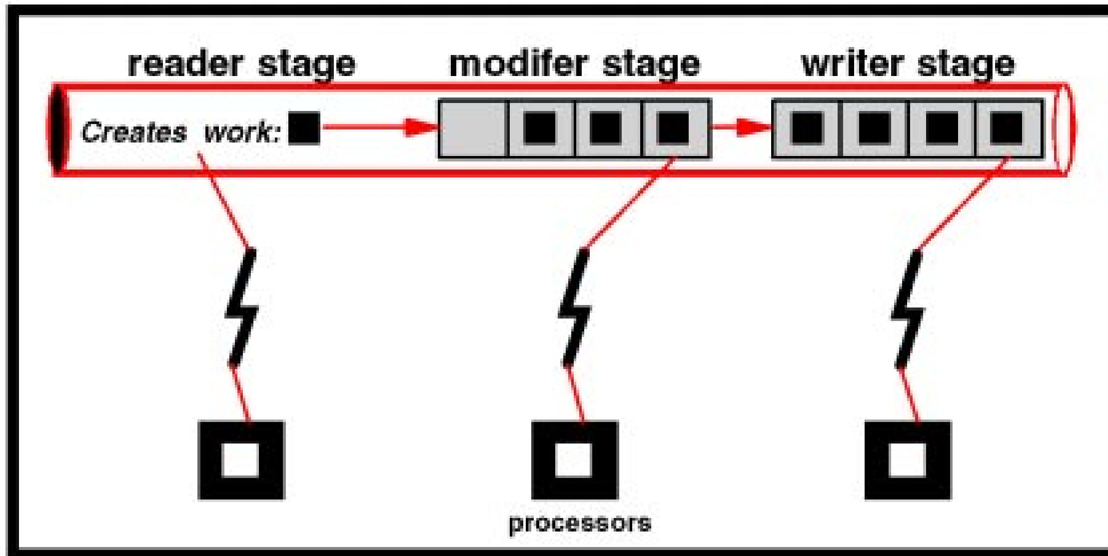
E.4 Parallelisierungsmodelle

1 Master-Slave Modell



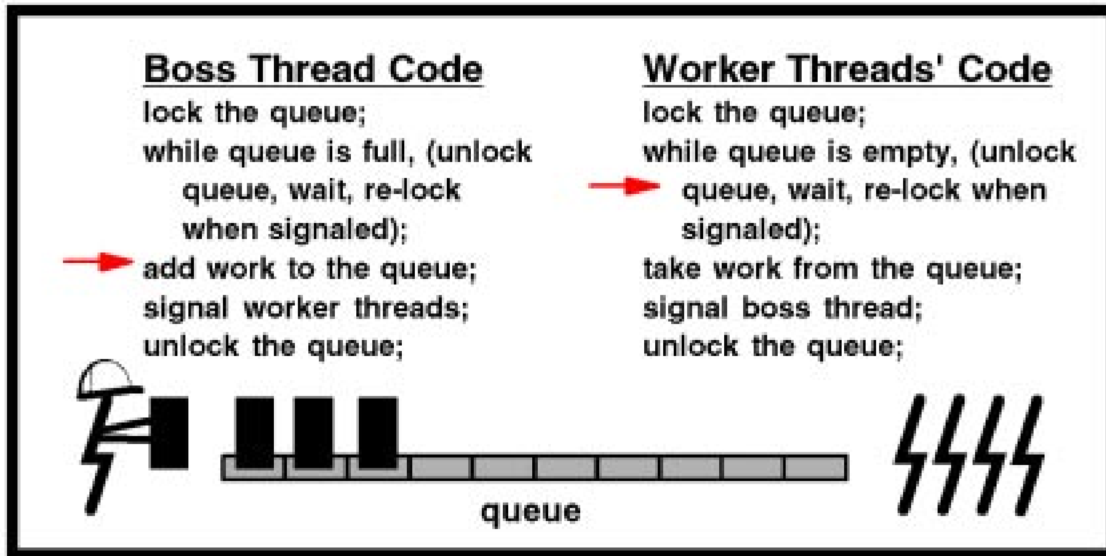
Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

2 Pipeline Modell



Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

3 Boss-Worker Modell



© Frank Bellosa, Univ. Erlangen-Nürnberg, IMMD IV, 1999

Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

E.5 Attribute

- Bei einigen Pthread Aufrufen (`pthread_create`, `pthread_mutex_init`, `pthread_cond_init`) kann eine Attributstruktur als Parameter spezifiziert werden, mit dem das neue Objekt konfiguriert wird.

- Das Erzeugen/Setzen/Abfragen von Attributen geschieht immer nach dem gleichen Schema:
  - Initialisierung der Attributstruktur.
  - Abfragen oder Setzen der gewünschten Werte.
  - Erzeugung des neuen Objektes unter Benutzung eines Zeiger auf die Attributstruktur als Parameter.
  - Gegebenenfalls Freigabe der Attributstruktur.

Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

E.5 Attribute (2)

- Initialisieren der Attributstruktur:

```
pthread_XXXattr_init(pthread_XXXattr_t *attr)
```

- Auslesen eines einzelnes Attributs:

```
pthread_XXXattr_getATTR(pthread_XXXattr_t *attr,
                        ATTRTYP *pointer)
```

- Setzen eines einzelnes Attributs:

```
pthread_XXXattr_setATTR(pthread_XXXattr_t *attr,
                        ATTRTYP value)
```

- Freigeben eines einzelnes Attributs:

```
pthread_XXXattr_destroy(pthread_XXXattr_t *attr)
```

- Attributierbare Objekte: `XXX`:
  - Threads: `XXX` = “” (Leerstring)
  - Mutex: `XXX` = “`mutex`”
  - Conditional: `XXX` = “`cond`”

Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

E.6 Thread Scheduling

1 Festlegung des Thread-Modells (1 x 1, M x 1, M x N)

- Das Modell für die Bindung an die Kernel-Threads wird durch den Scheduling-Scope festegelegt. Der gewünschte Scope für einen Thread kann durch ein Threadattribut der Funktion `pthread_create` mitgeteilt werden:

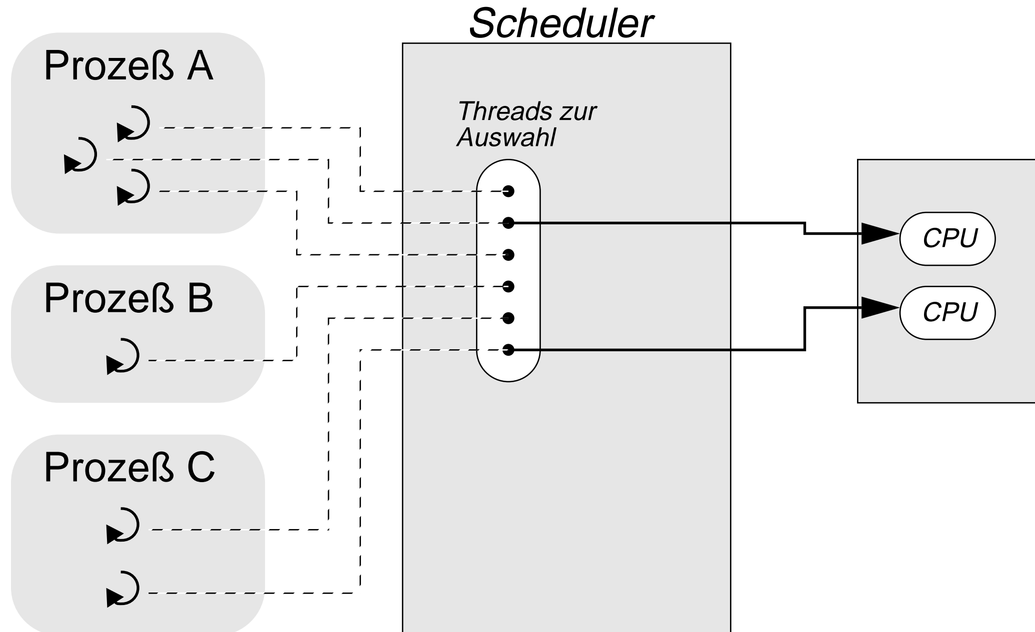
  ATTR=`scope`, ATTRTYP=`int`

  - `PTHREAD_SCOPE_SYSTEM`: Die Schedulingstrategie soll für alle laufenden Threads des Systems (also aus allen Prozessen) gelten.
    Es handelt sich hier um einen gebundenen Thread (1 x 1 Modell).
  - `PTHREAD_SCOPE_PROCESS`: Die Schedulingstrategie soll nur für die Threads eines Prozesses gelten. Zwischen den Prozessen wird mit dem standard Systemscheduler umgeschaltet.
    Es handelt sich hier um einen ungebundenen Thread (M x 1, M x N Modell)

Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

1 Scheduling Scope (2)

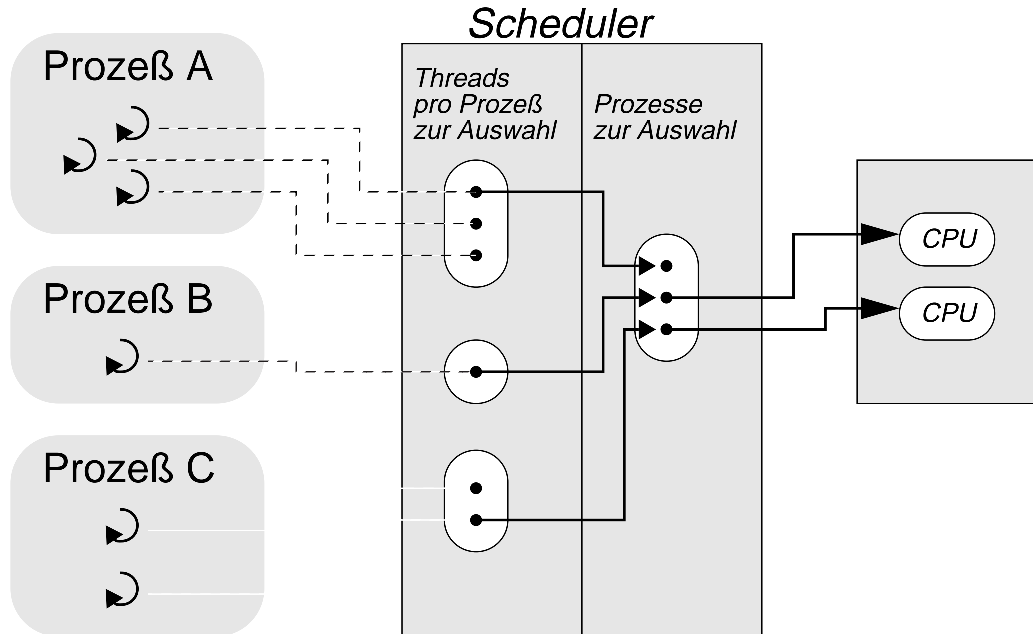
■ Beispiel für System-Scope



Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

1 Scheduling Scope (3)

- Beispiel für Prozeß-Scope (M x 1)



Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

2 Scheduling Strategie

- Die gewünschte Scheduling Strategie für einen Thread kann durch ein Threadattribut der Funktion `pthread_create` mitgeteilt werden:

  ATTR=`schedpolicy`, ATTRTYP=`int`

  - `SCHED_FIFO`: Ein Thread läuft bis er stirbt oder sich blockiert. Wenn er, nachdem er sich blockiert hat, weiterlaufen kann, wird er am Ende der Queue für seine Priorität eingehängt.
    Dieser Attributwert ist nur für gebundene Threads anwendbar und für Echtzeitanwendungen vorgesehen (In Solaris nicht implementiert).
  - `SCHED_RR`: Ähnlich wie `SCHED_FIFO`, allerdings wird der Thread auf jeden Fall nach Ablauf eines festen Quantums unterbrochen.
    Dieser Attributwert ist nur für gebundene Threads anwendbar und für Echtzeitanwendungen vorgesehen (In Solaris nicht implementiert).
  - `SCHED_OTHER`: Eine beliebige Strategie. Dies ist die Standardeinstellung!
    Dieser Attributwert ist nur für Threads im Time-Sharing Betrieb anwendbar.

Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

3 Threadpriorität

- Die gewünschte Priorität eines **ungebundenen Threads** kann durch ein Attribut der Funktion `pthread_create` mitgeteilt werden:
  ATTR=`schedparam`, ATTRTYP=`struct sched_param`
  Die Struktur wird auch beim `set` Aufruf als Pointer übergeben!
  Die verwendeten Parameter sind abhängig von der eingestellten Schedulingstrategie. POSIX.1b erzwingt genau ein Attribut für `sched_param`:

```
    struct sched_param {
        int sched_priority;
    };
```

  Um den Bereich abzufragen, in dem `sched_priority` gültig ist, können die POSIX.1b Aufrufe
  `sched_get_priority_max(int schedpolicy)`
  `sched_get_priority_min(int schedpolicy)`
  verwendet werden.

- Die Priorität von **gebundenen Threads** kann durch einen Systemaufruf `priocntl()` geändert werden.

Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

4 Scheduling Parameter

- Die Schedulingparameter von laufenden ungebundenen Threads können nachträglich angepaßt werden. Dies geschieht über

```
pthread_getschedparam(pthread_t thread,
                      int *policy,
                      struct sched_param *param)
```

  und

```
pthread_setschedparam(pthread_t thread,
                      int policy,
                      struct sched_param *param)
```

- `policy` und `param` entsprechen den Parametern, wie sie bei den Threadattributen werden (policy: `SCHED_FIFO`, `SCHED_RR`, `SCHED_OTHER`, param: `sched_priority`).

© Frank Bellosa, Univ. Erlangen-Nürnberg, IMMD IV, 1999
Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

5 Vererbung von Scheduling Parametern

- Ein Attribut kann festlegen, ob die Schedulingparameter aus der Attributstruktur übernommen werden sollen, oder ob diese ignoriert werden und die Parameter vom Vaterthread kopiert werden sollen.
  ATTR=`inheritsched`, ATTRTYP=`int`
  - `PTHREAD_INHERIT_SCHED`: Kopiere die Parameter vom Vaterthread
  - `PTHREAD_EXPLICIT_SCHED`: Übernehme die Parameter aus dem Prozeßattributobjekt.

Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

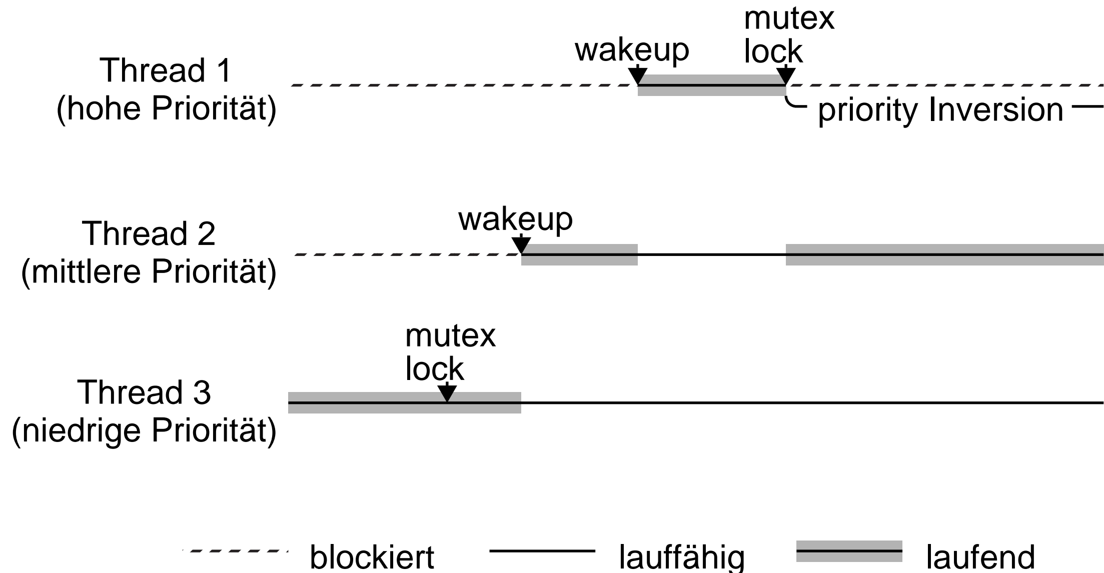
6 Priority Inversion

- Über die Mutexattribute lassen sich Strategien zur Vermeidung von *Priority Inversion* konfigurieren. Außerdem ist einstellbar, daß Mutexe auch über Prozeßgrenzen hinweg funktionieren sollen.
- Priority Inversion bedeutet, daß ein Thread mit hoher Priorität nicht arbeiten kann, weil ein anderer mit niedriger Priorität eine Resource hält.
- Dies kann in Realtime-Anwendungen zu drastischen Problemen führen.

Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

6 Priority Inversion (2)

- Beispiel:



Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

6 Priority Inversion (2)

- Strategie 1: Priority Ceiling
  - Hier wird jedem Mutex eine eigene Priorität zugeordnet. Wenn ein Thread einen kritischen Abschnitt betritt und einen Mutex lockt, bekommt er automatisch die Priorität zugeteilt. Beim Unlock des Mutex wird seine alte Priorität restauriert.
  - Diese Strategie ist verwendbar, wenn die Konstante `_POSIX_THREAD_PRIO_PROTECT` definiert ist.
  - Priority Ceiling wird gewählt, indem das Attribut ATTR=`protocol`, ATTRTYPE=`int` in den Mutexattributen auf `PTHREAD_PRIO_PROTECT` gesetzt wird.
    Der Attributwert `PTHREAD_PRIO_NONE` schaltet Priority Ceiling wieder ab.
  - Die Priorität des Mutexes sollte so hoch gewählt werden, wie das Maximum der Prioritäten aller Threads, die den Mutex belegen können.
  - Die Priorität des Mutexes ist mit dem Attribut ATTR=`prioceiling`, ATTRTYPE=`int` einstellbar.

Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

6 Priority Inversion (3)

- Strategie 2: Priority Inheritance
  - Hier wird die Priorität des Threads, der den Mutex belegt hat, dynamisch so verändert, daß sie immer so groß ist wie das Maximum der Prioritäten aller Threads, die auf den Mutex warten.
  - Diese Strategie ist verwendbar, wenn die Konstante `_POSIX_THREAD_PRIO_INHERIT` definiert ist.
  - Priority Inheritance wird gewählt, indem das Attribut ATTR=`protocol`, ATTRTYPE=`int` in den Mutexattributen auf `PTHREAD_PRIO_INHERIT` gesetzt wird.

Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

7 Entkopplung von Threads (detached threads)

- Mit `pthread_detach(pthread_t thread)` kann der Pthread-Bibliothek mitteilen werden, daß der Exitstatus des Threads `thread` nicht gespeichert werden soll.
  ➜ Automatische Freigabe alle Thread-Ressourcen.
  ➜ Ein `pthread_join` ist nicht mehr möglich.
  ➜Die ID eines Thread kann nach seiner Terminierung für einen neuen Thread wiederverwendet werden! Falls unklar ist,ob ein entkoppelter Thread schon terminiert ist, darf seine Handle nicht mehr verwendet werden.

- Die Entkopplung kann auch direkt über ein Attribut beim Aufruf von `pthreat_create` geschehen.

  - ATTR=`detachstate`, ATTRTYP=`int`
    - `PTHREAD_CREATE_JOINABLE`: Auf diesen Thread kann ein `join()` durchgeführt werden, d.h., das System muß sich den “Exitstatus” merken.
    - `PTHREAD_CREATE_DETACHED`: Dieser Thread soll völlig unabhängig vom Vaterthread laufen.

Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

7 Entkopplung von Threads (2)

- Soll ein Server für jede Anfrage einen neuen Thread starten (ala *inetd*), ist es nicht sinnvoll, daß der erzeugt Thread noch Verbindung zu seinem Vaterthread hat.

- Beispiel – Server Applikation:

```
main() {
    ...
    for (;;) {
        pthread_t worker;
        req = malloc(sizeof *req);
        wait_for_request(&req);
        pthread_create(&worker, NULL,
                       process, req));
        pthread_detach(worker);
    }
}
```


Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

E.7 UNIX Interaktion

- UNIX kennt traditionell nur Prozesse als Aktivitätsträger, die alle einen eigenen Adreßraum besitzen. Daraus resultieren eine Reihe von Problemen bezüglich Threads:
  - Können sich mehrere Threads gleichzeitig in einer Bibliotheksfunktion befinden?
  - Was passiert, wenn eine Cancellation geschieht, während der Thread eine Bibliotheksfunktion abarbeitet?
  - UNIX Signale: Wenn ein Prozeß ein Signal empfängt, welche Folgen entstehen für die Threads?
  - Was passiert bei *fork()*, *exec()* und *exit()*?
  - Gibt es eine Synchronisation über Prozeßgrenzen hinweg?
  - Was passiert, wenn eine Funktion einen Systemaufruf tätigt, der sich blockiert?

Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

1 Bibliotheksaufrufe

- Durch die Verwendung von Threads kann es passieren, daß sich mehrere Aktivitätsträger gleichzeitig in einer Funktion befinden. Wenn die Funktion Datenstrukturen manipuliert, kann es zu Fehlern kommen, wenn diese nicht mit Mutexen oder anderen Locks versehen wurden.

- Funktionen, die so gesichert wurden, nennt man (*multi-*) *thread-safe* (*MT-safe*) oder *reentrant*.

- Alle POSIX-Funktionen sind MT-safe, bis auf zwei Klassen von Ausnahmen:
  - Funktionen, deren Spezifikation dies nicht zuläßt.
  - Funktionen, die aus Geschwindigkeitsgründen nicht mit Locks versehen wurden.

Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

1 Bibliotheksaufrufe (2)

- Funktionen, die von ihrer Spezifikation nicht threadsafe gemacht werden können, sind solche, die Daten in einen statischen Puffer hinterlegen und häufig einen Zeiger auf sie zurückliefern. Dies sind zum Beispiel `rand()`, `getpw*()`, `ctime()`, `strtok()`...

- Für viele dieser Funktionen spezifiziert der Pthread Standard ein zweites Interface, bei dem der Puffer über zusätzliche Parameter übergeben werden kann. Diese Funktionen enden alle auf `_r` (wie reentrant).

- Beispiel:

```
char *ctime_r(const time_t *clock,
              char *buf, int buflen)
```


© Frank Bellosa, Univ. Erlangen-Nürnberg, IMMD IV, 1999

Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

1 Bibliotheksaufrufe (3)

- Die Standard-IO Funktionen `getc/getchar/putc/putchar` wurden zwar threadsafe gemacht, es existiert aber noch eine Variante, bei der auf das Lock verzichtet wird.

- Diese Funktionen enden auf `_unlocked`. Sie eignen sich vor allem dann, wenn sie für eine Zeitlang häufig hintereinander aufgerufen werden. In diesem Fall ist es sinnvoll, vor dem ersten Aufruf das `FILE` zu locken, die Aufrufe durchzuführen, und danach ein Unlock aufzurufen.

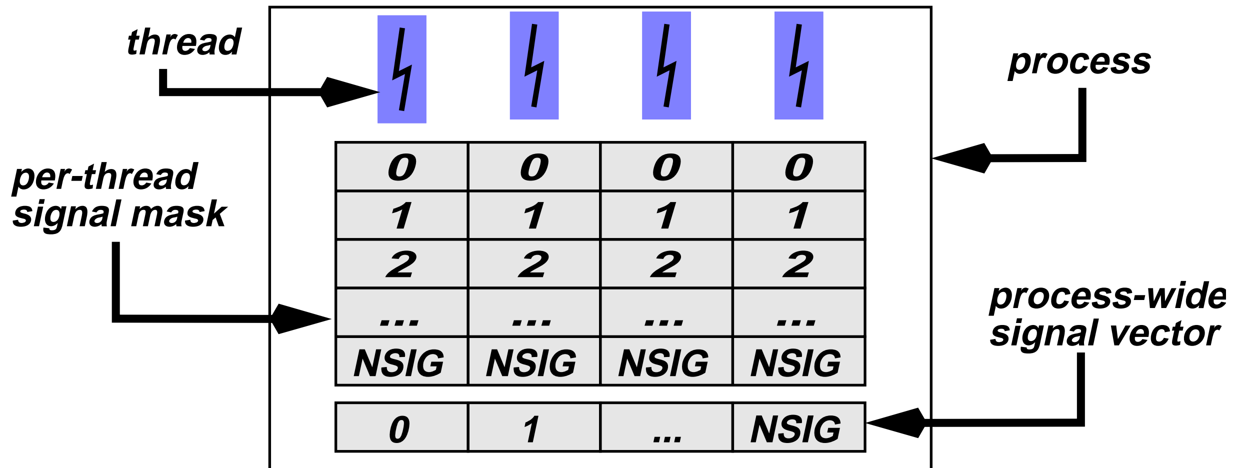
- Die `FILE`-Struktur kann mit `flockfile(FILE *stream)`, `funlockfile(FILE *stream)` und (nicht blockierend) mit `ftrylockfile(FILE *stream)` gesperrt und freigegeben werden.

Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

2 Signale

- Um kompatibel zu UNIX zu bleiben, spezifizierten die Pthread Entwickler, daß Signale sich auf ganze Prozesse beziehen. Ein Signalhandler gilt also für alle Threads, es kann der standard POSIX Aufruf *sigaction()* verwendet werden.

- Allerdings erlaubt der Pthread Standard jedem Thread, Signale auszumaskieren. Dies erlaubt den Benutzer, die Aufgaben des Signalhandlings an einzelne Threads zu verteilen.



Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

2 Signale (2)

- Die Art, Signale zu maskieren, entspricht der Technik, die POSIX für Prozesse spezifiziert hat:
  - Es gibt eine Signalmaske die alle Signale enthält.
  - Diese kann gelesen/verändert/gesetzt werden.
- Setzen der Threadsignalmaske (analog zu `sigprocmask`):

```
int pthread_sigmask(int how, const sigset_t *set,
                              sigset_t *oset)
```

- Wenn `oset` gesetzt ist, wird hier die alte Maske hinterlegt.
- Ist `set` gesetzt, wird die aktuelle Maske verändert. `how` gibt an, in welcher Weise dies geschehen soll:
  - `SIG_BLOCK`: die Maske wird zu der aktuellen Maske hinzugefügt.
  - `SIG_UNBLOCK`: die Maske wird von der aktuellen entfernt.
  - `SIG_SETMASK`: die Maske wird übernommen.

Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

2 Signale (3)

- Beim Eintreffen eines Signals wählt das System einen Thread aus, der den Signalhandler aufruft.
  - Wenn das Signal eindeutig einem Thread zugeordnet werden kann (z.B. bei Exceptions (`SEGV`...)), wird dieser Thread verwendet.
  - Andernfalls wird ein beliebiger Thread ausgewählt, der das Signal nicht blockiert hat.

- Es ist auch möglich, ein Signal direkt an einen Thread des gleichen Prozesses zu schicken:

  ```
  pthread_kill(pthread_t thread, int sig)
  ```

  `thread` ist durch `pthread_create` oder `pthread_self` erhaltene Threadkontrollblock.

Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

2 Signale (4)

- Für einen Signalhandler gelten einige Einschränkungen: Er darf nur Funktionen aufrufen, die *asynchronous signal-safe* sind. Das bedeutet, daß die Funktion gefahrlos unterbrochen werden kann, ohne daß Inkonsistenzen oder Deadlocks eintreten. Async signal-safe ist somit eine Einschränkung von MT-safe.

- Der Standard führt eine Reihe von Funktionen auf, die signal-safe sein müssen. Dies sind praktisch nur Kernaufrufe.

- *Keine* der Pthread Funktionen muß signal-safe sein! Das bedeutet, daß in einem Signalhandler keine Mutexe oder ähnliches verwendet werden dürfen.

© Frank Bellosa, Univ. Erlangen-Nürnberg, IMMD IV, 1999

Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

2 Signale (5)

- Diese Einschränkung ist nicht ganz so schlimm, wie sie sich zuerst anhört. Man löst das Problem dadurch, indem man einen Thread startet, der nur für die Annahme des gewünschten Signals zuständig ist. Alle Threads maskieren dieses Signal aus.

- Dieser Thread ruft die POSIX Standardfunktion `sigwait()` auf, die solange wartet, bis ein Signal eintrifft, und dann zurückkehrt:

  ```
  int sigwait(const sigset_t *set, int *sig)
  ```

  `set` gibt an, auf welche Signale gewartet werden soll. In `sig` wird bei Eintreffen die Signalnummer gespeichert.

© Frank Bellosa, Univ. Erlangen-Nürnberg, IMMD IV, 1999

Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

2 Signale (6)

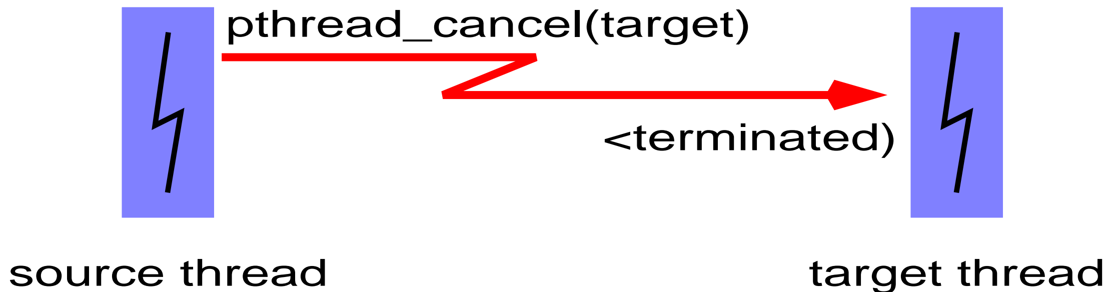
- Beispiel: `SIGCHLD` Handler

```
sigset_t chldset;
pthread_t chldthr;
...
void *chldhand(void *arg) {
    int sig;
    for (;;) {
    sigwait(&chldset, &sig);
    waitpid()
    ...
}
...
    sigemptyset(&chldset);
    sigaddset(&chldset, SIGCHLD);
    pthread_sigmask(SIG_BLOCK, &chldset, NULL);
    pthread_create(&chldthr, NULL, chldhand, NULL);
```


Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

3 Abbruch eines Threads

- Manche Probleme erfordern die Möglichkeit, laufende Threads abzubrechen (sogenannte *Thread Cancellation*).
  Beispiel: Falls ein Thread in einem Suchalgorithmus erfolgreich war, ist es nicht mehr nötig, daß alle andern Threads weitersuchen.



- Ein Thread kann mit
  `pthread_cancel(pthread_t thread)`
  versuchen, den durch `thread` beschriebenen Thread abzubrechen.

Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

3 Abbruch eines Threads (2)

- Der Pthread Standard ermöglicht drei Arten, mit denen ein Thread auf eine Abbruchaufforderung reagieren kann:
  - Er kann sie ignorieren (`PTHREAD_CANCEL_DISABLE`)
  - Er kann sofort abbrechen.
    (`PTHREAD_CANCEL_ENABLE + PTHREAD_CANCEL_ASYNCHRONOUS`)
  - Er kann sich die Aufforderung merken und beim nächsten Cancellation-Point abbrechen.
    (`PTHREAD_CANCEL_ENABLE + PTHREAD_CANCEL_DEFERRED`)
    Dies ist standardmäßig eingestellt.

Cancelstate	Canceltype
PTHREAD_CANCEL_ENABLE	PTHREAD_CANCEL_ASYNCHRONOUS oder PTHREAD_CANCEL_DEFERRED
PTHREAD_CANCEL_DISABLE	--

Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

3 Abbruch eines Threads (3)

- Der Empfang von Cancelversuchen kann mit

```
pthread_setcancelstate(int state,
                       int *oldstate)
```

  ein- und ausgeschaltet werden. (ähnlich wie die Signalmaske)
  Gültige Werte für `state` sind die vordefinierten Werte:

  - `PTHREAD_CANCEL_ENABLE`
  - `PTHREAD_CANCEL_DISABLE`

- Ob ein Cancelversuch sofort wirksam wird oder erst an einem Cancellation-Point, ist mit

```
pthread_setcanceltype(int type,
                      int *oldtype)
```

  konfigurierbar.
  Auch hier gibt es zwei vordefinierte Werte:

  - `PTHREAD_CANCEL_ASYNCHRONOUS`
  - `PTHREAD_CANCEL_DEFERRED`

Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

3 Abbruch eines Threads (4)

- Da ein Thread wichtige Resourcen belegt haben kann (z.B. Mutexe), ist es aber meist sinnvoll, ihn nur an bestimmten Stellen im Programm abbrechen zu lassen. Diese Stellen werden *Cancellation-Points* genannt.

- Ein Cancellation-Point kann durch den Aufruf von
  `void pthread_testcancel(void)`
  markiert werden. Zusätzlich sind folgende Aufrufe automatisch gültige Cancellation-Points:
  - pthread_cond_[timed]wait
  - pthread_join
  - Systemfunktionen, die längere Zeit warten können, dürfen (und müssen teilweise) auch Cancellation-Points sein.

Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

3 Abbruch eines Threads (5)

- Beispiel:

```
/* make shute we cannot be canceled asynchonously */
type = PTHREAD_CANCEL_DEFERRED;
(void) pthread_setcanceltype(type, &oldtype);

/* do some buffer processing */
(void) pthread_mutex_lock(&write_mtx);
...
(void) pthread_mutex_unlock(&write_mtx);

/* act on cancellation requests before writing database */
pthread_testcancel();

/* don't process cancellation requests */
state = PTHREAD_CANCEL_DISABLE;
(void) pthread_setcancelstate(state, &ostate);
...
write_into_database(buffer);
...
/* restore cancelability state and type */
(void) pthread_setcancelstate(oldstate, &state);
(void) pthread_setcanceltype(oldtype, &type);
```


Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

4 Der Cleanup Stack

- Problem: Verklemmung durch belegte Locks nach einer Cancellation

```
/* make shute we cannot be canceled asynchonously */
type = PTHREAD_CANCEL_DEFERRED;
(void) pthread_setcanceltype(type, &oldtype);

/* do some buffer processing                        */
(void) pthread_mutex_lock(&write_mtx);
if (global_data_ptr != NULL) {
   write(fd, global_data_ptr, sizeof(global_data_ptr));
   free(global_data_ptr);
   global_data_ptr = NULL;
(void) pthread_mutex_unlock(&write_mtx);
```

cancel

◆ Der Mutex wird durch den abgebrochenen Thread nicht freigegeben. Ein anderer Thread, der diesen Code-Bereich durchlaufen wird, wird für immer auf die Freigabe des Mutex warten.
➜ Die Anwendung verklemmt sich!

Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

4 Der Cleanup Stack (2)

- Problem: Inkonsistente Daten nach einer Cancellation

```
/* make shute we cannot be canceled asynchonously */
type = PTHREAD_CANCEL_ASYNCHRONOUS;
(void) pthread_setcanceltype(type, &oldtype);

/* do some buffer processing                       */
(void) pthread_mutex_lock(&write_mtx);
if (global_data_ptr != NULL) {
   write(fd, global_data_ptr, sizeof(global_data_ptr));
   free(global_data_ptr);
   global_data_ptr = NULL;
(void) pthread_mutex_unlock(&write_mtx);
```

cancel

◆ Die Schreiboperation **write** wurde beendet. Durch den Abbruch des Threads wird aber der globale Zustand nicht mehr richtig gesetzt. Die Anwendung denkt, die Daten seien noch nicht geschrieben worden.
➜ Die Daten der Anwendung werden inkonsistent!

Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

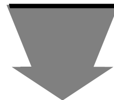
4 Der Cleanup Stack (3)

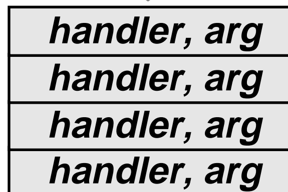
- Um die Speicherfreigabe und andere Aufräumarbeiten zu vereinfachen, hält die Pthread Bibliothek pro Thread einen sogenannten *Cleanup Stack*.

- Wenn ein Thread verstirbt (durch `pthread_exit`, Rücksprung aus der Startfunktion oder einer erfolgreichen Cancellation), werden alle Routinen aufgerufen, die auf den Stack geschoben worden sind (natürlich in LIFO Reihenfolge).

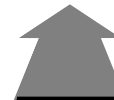
- Zu jedem Push-Aufruf muß es genau ein Pop-Aufruf geben, und zwar auf gleicher Programmblockebene. Dies erlaubt es, die beiden Funktionen als Macros zu definieren, die neue Variablen auf dem Stack anlegen.

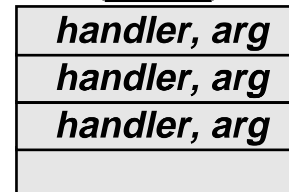
pthread_cleanup_push(handler,arg)





pthread_cleanup_pop(execute)





© Frank Bellosa, Univ. Erlangen-Nürnberg, IMMD IV, 1999
Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

4 Der Cleanup Stack (4)

- Push auf den Cleanup Stack:

```
void pthread_cleanup_push(
                         void (*routine)(void *),
                         void *arg)
```

  Die Funktion muß vom Typ `void` sein. Das Argument `arg` wird beim Aufruf der Cleanupfunktion als Parameter übergeben.

- Pop der letzten Funktion:

```
void pthread_cleanup_pop(int execute)
```

  Falls `execute` gesetzt ist, wird die gespeicherte Funktion aufgerufen.

Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

4 Der Cleanup Stack (5)

```
void cleanup_func (char *ptr) {
 free(ptr);
}
...

void start_routine() {
 char *ptr
 ptr = (char *) malloc(1024);
 pthread_cleanup_push((void (*)())cleanup_func, (void *)ptr)

 /* Code to do sometinhg with pointer ptr       */
 ...
 /* Finish with ptr, free() it up an remove the */
 /* Cancellation cleanup handl                  */
 pthread_cleanup_pop(0);
 free(ptr);

 pthread_exit(0);
}
```


Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

4 Der Cleanup Stack (6)

```
void cleanup_mutex (pthread_mutex_t *lock) {
 pthread_mutex_unlock(lock);
}

void start_routine() {
struct job_task *task;
for ( ; ; ) {
  (void) pthread_mutex_lock(task_lock);
  pthread_cleanup_push((void (*)())cleanup_mutex,
                       (void *)task_lock)
  while (task_queue_empty())
        pthread_cond_wait(task_cv, task_lock);
  pthread_cleanup_pop(0);
  task = task_dequeue();
  pthread_mutex_unlock(task_lock);
  /* Process the Task */
  ...
  free((void *)task);
 }
}
```


© Frank Bellosa, Univ. Erlangen-Nürnberg, IMMD IV, 1999

Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

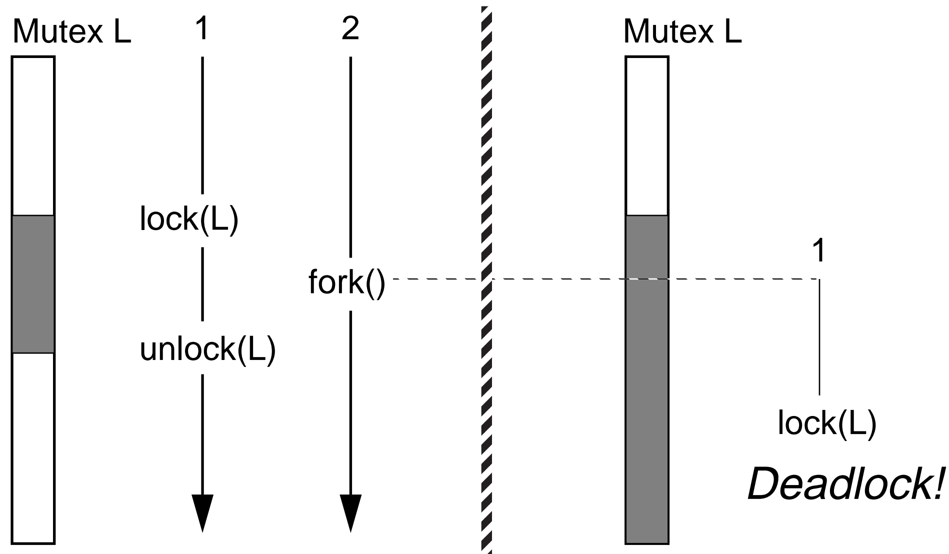
5 Fork/Exec/Exit

- Die Pthread Designer haben versucht, sich möglichst wenig von der Semantik der UNIX Prozesse zu entfernen.
- Ruft ein Thread in einem Programm, in dem mehrere Threads laufen, `fork()` auf, wird ein neuer Prozeß mit einem Abbild das Adreßraumes und genau einem Thread gestartet.
- Dieses führt zu zwei schwerwiegenden Problemen:
  - ◆ Der Kindprozeß erbt den Zustand aller Locks des Vaterprozesses, ohne über deren Status genaueres zu wissen.
  - ◆ Der Kindprozeß erbt den Datenbereich, also auch den von anderen Threads alloziierten Speicher. Ein sauberes Freigeben der unbenötigten Bereiche ist schwierig.
- Diese Probleme sind natürlich irrelevant, wenn nach dem `fork()` direkt ein `exec()` oder `exit()` aufgerufen wird.

Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

5 Fork/Exec/Exit (2)

- Beispiel für einen Deadlock:



Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

5 Fork/Exec/Exit (3)

- Die Pthread Bibliothek bietet zur Lösung dieses Problems sogenannte *fork-handling Stacks* an. Diese werden bei einem `fork()` Aufruf automatisch abgearbeitet.

- Es gibt drei unterschiedliche Stacks:
  - Der *prepare-Stack*: Die Funktionen auf diesem Stack werden automatisch vor dem eigentlichen fork() aufgerufen.
  - Der *parent-Stack*: Der Vater des erzeugten Prozesses ruft nach dem fork() automatisch alle Funktionen auf diesem Stack auf.
  - Der *child-Stack*: Der erzeugte Prozeß ruft alle auf diesen Stack gelegten Funktionen auf.

Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

5 Fork/Exec/Exit (4)

- Die Syntax:

```
pthread_atfork(void (*prepare)(void),
               void (*parent)(void),
               void (*child)(void))
```

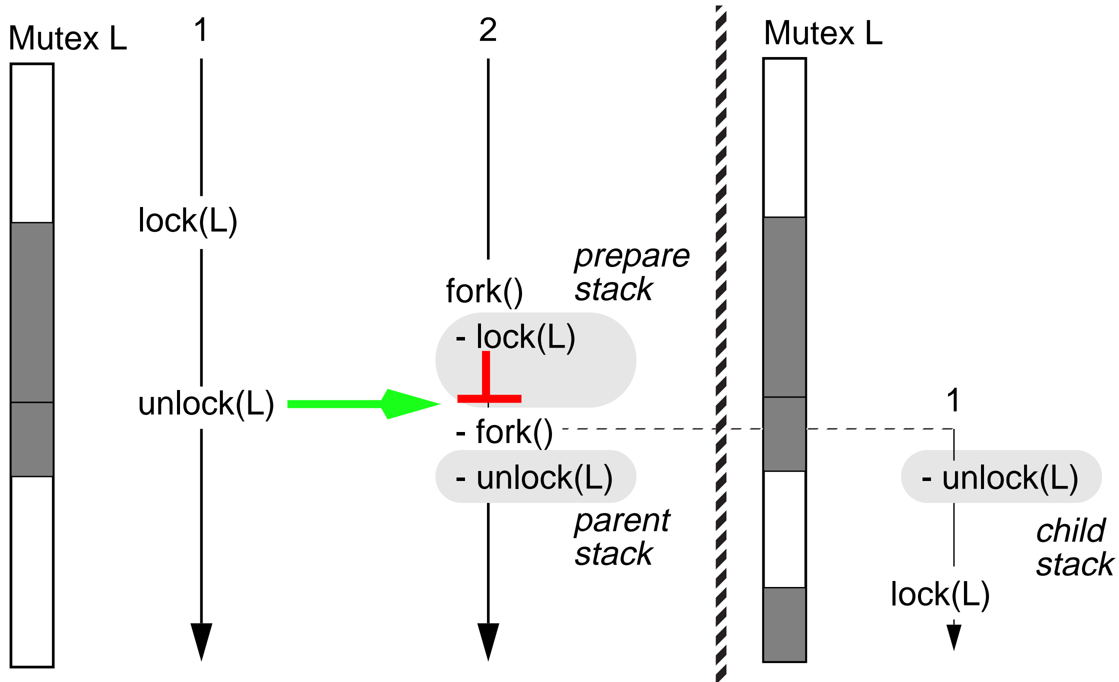
Ein Stack kann durch Angabe von `NULL` unverändert bleiben.

- Mit dieser Funktion ist das fork-Problem leicht lösbar:
  - Die Routinen auf dem prepare-Stack belegen alle Mutexe (in der richtigen Reihenfolge), die der erzeugte Prozeß benötigt.
  - Die Funktionen auf dem parent- und dem child-Stack geben die Mutexe wieder frei. Zusätzlich kann die child-Funktion noch weitere Resourcen freigeben.

Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

5 Fork/Exec/Exit (5)

- Vermeidung des Deadlocks:



Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

5 Fork/Exec/Exit (6)

- Da der Pthread Standard Implementierungen zuläßt, die auf einem Monoprozessor ohne Kernunterstützung laufen, muß ein Aufruf von `exec()` und `exit()` eine automatischen Terminierung aller Threads bewirken.

- Ein Aufruf von `exit()` bewirkt noch einige Aufräumaktionen (Flush aller geöffneten Files).

- Soll dies vermieden werden, kann `_exit()` aufgerufen werden. Dies ist der direkte Kernaufruf.

Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

6 Interprozeß-Synchronisation

- Durch das Setzen von Attributen können Mutexe und Conditionals über Prozeßgrenzen hinweg verwenden werden.

- Der Mutex/Conditional muß hierzu in einem Shared-Memory-Bereich liegen, auf den die Prozesse zugreifen können (über `mmap()` oder SysV-IPC). Weiterhin ist diese Option nur gestattet, wenn die Konstante `_POSIX_THREAD_PROCESS_SHARED` definiert ist.

- Das benötigte Attribut heißt ATTR=`pshared`, ATTRTYPE=`int`.
  - `PTHREAD_PROCESS_SHARED`: ermöglicht diese Option,
  - `PTHREAD_PROCESS_PRIVATE`: verbietet sie.

Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

E.8 Zusammenfassung

- Threads sind ein mächtiges Hilfmittel zur Parallelisierung eines Programmes.
- Der Vorteil des schnellen und uneingeschränkten Zugangs auf alle Prozeßresourcen (Speicher, Files) ist allerdings gleichzeitig das Hauptrisiko, da es leicht zu sehr schwer zu findende *Raceconditions* kommen kann.
- Man sollte sich immer genau überlegen, ob man auf Synchronisations- und Ausnahmebehandlungsroutinen wirklich verzichten kann.

Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

F Übung 1

- Unter ~iwi425/PPS/streams finden Sie die Sourcen des Streams-Benchmark zur Bestimmung der Speicherbandbreite und der Rechenleistung in Abhängigkeit von der Arbeitsmenge.
  Arbeiten Sie auf einer unbelasteten Workstation vom Typ SUN Ultra-10 (300 MHz Ultra-SPARC IIi, 128 MB Speicher, 512 KB physical Cache, 64 entry Data-TLB)
  - Ändern Sie den Benchmark (C-Variante) derart ab, daß Sie bei einem Workingset von 32 KB (Summe aller beteiligten Cache-Zeilen) bei jedem Speicherzugriff einen TLB-Zugriffsfehler erzeugen.
    Vergleichen Sie Ihre Ergebnisse mit der einer TLB-freundlichen Variante.
  - Weisen Sie den Effekt des zufälligen Page-Coloring nach.
    Allokieren Sie im Streams Benchmark 0.5 MB Speicher und lassen die Messung durchlaufen, beenden den Prozeß aber nicht. Jetzt starten Sie mehrfach den gleichen Prozeß mit 100 MB Speicheranforderung. Heftige Auslagerung von Seiten ist die Folge. Lassen Sie jetzt den zuerst gestarteten Benchmark weiterlaufen und messen Sie den Druchsatz.

Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

F Übung 2

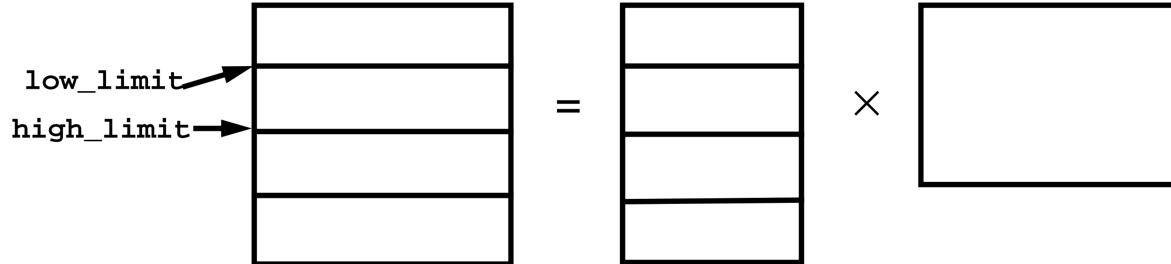
- Arbeiten Sie auf einer unbelasteten Workstation vom Typ SUN Ultra-10 (300 MHz Ultra-SPARC IIi, 128 MB Speicher, 512 KB physical Cache, 64 entry Data-TLB)
  - ◆ Schreiben Sie ein 2-dimensionales Feld mit Integerzahlen (Feldgröße 110 MB) in eine Datei unter /usr/tmp.
    - Schreiben Sie mit dem `write()` call und analysieren Sie unterschiedliche lange Puffergrößen (8190 Bytes, 8192 Bytes).
    - Allokieren Sie eine Datei von 110 MB (`open()`, `lseek()`), mappen Sie die Datei in den Addressraum (mmap), initialisieren Sie das Feld (z.B. alle Feldelemente auf 1) und machen sie die Änderungen persistent.
    - Summieren sie alle Element des in die Datei gespeicherten Feldes auf. Wählen Sie einen Ansatz mit `read()` und einen mit `mmap()`.

Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

F Übung 3

- Implementieren Sie analog zur Matrix-Vektoroperation eine parallele Matrixmultiplikation mit Pthreads

$$C[i,j] = \sum_{k=1}^{s} A[i,k] \cdot B[k,j]; \quad 1 \le i \le n, 1 \le j \le m$$



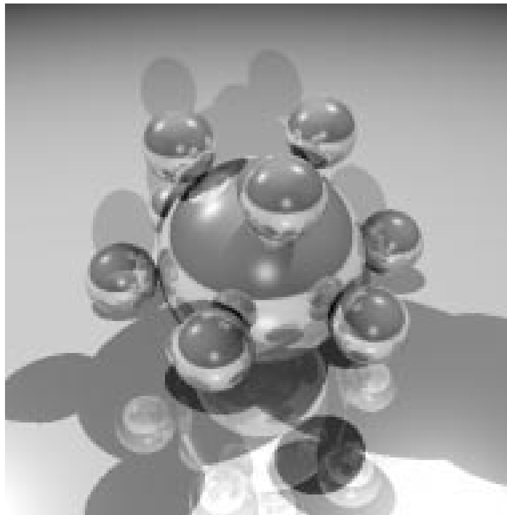
- Vereinfachung: Quadratische Matrizen der Größe n×n ; n = dimension
- Compilieren Sie mit /opt/SUNWspro/bin/cc -mt ..... -lpthread
- Variieren Sie die Zahl der Kernel-Threads.
- Messen Sie den Speedup auf dem Rechner cssun mit 1-8 Kernel-Threads in Abhängigkeit von der Matrixgröße.

© Frank Bellosa, Univ. Erlangen-Nürnberg, IMMD IV, 1999

Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

F Übung 4

- Raytracer eignen sich hervorragend zur Parallelisierung, da jeder Bildpunkt unabhängig von den anderen berechnet werden kann.
- In `~iwi425/PPS/ray/` findet sich ein kleiner Raytracer, der das Bild einer einfachen Szene erzeugt.



- Start des raytracers mit **`tr | /local/bin/xv -`**

Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

F Übung 4 (2)

- Parallelisieren Sie den Raytracer mit Hilfe von Pthreads.
  - Unterteilen Sie hierzu das Bild in einzelne Streifen und lassen sie die Streifen von eigenen Threads berechnen.
  - Vergessen sie nicht, in Ihr Program den Aufruf von `thr_setconcurrency(8)` einzubauen, um auf der `cssun` mit allen 8 Prozessoren rechnen zu können!
  - Warum ist die einfache Aufteilung in 8 Streifen nicht optimal? Die Rechenzeit verkürzt sich, wenn mehr Streifen und damit (Userlevel-)Threads eingesetzt werden. Was ist die optimale Anzahl von Threads?
  - Vergrößern Sie den Bildbereich, so daß ein starkes Lastungleichgewicht bei 8 Threads entsteht.
    - Implementieren Sie jetzt mit Hilfe von Thread-Cancellation eine Repartitionierung der noch nicht berechneten Streifen.
    - Lösen Sie diese Aufgabe auch durch den Einsatz von Signalen (SIGUSR1).

© Frank Bellosa, Univ. Erlangen-Nürnberg, IMMD IV, 1999

Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

F Übung 5

- In dieser Aufgabe soll ein HTTP Server parallelisiert werden. Der serielle Source findet sich in `~iwi425/PPS/tinyhttp/`
- Zum Testen kann der WWW Baum des RRZE verwendet werden. Ein Aufruf wäre:
  `./tinyhttp 2345` /home/rzhome/iwi4/iwi425/PPS/wwwcip
  Danach könnte zum Beispiel über die URL
  `http://cssun.rrze:12345/docs/RRZE/`
  auf den Server zugegriffen werden können.
- Da der Server zu langsam arbeitet, sollen die einzelnen Requests parallel von Threads abgearbeitet werden.
- Implementieren Sie den Server mit dem Boss-Worker Modell.
- Setzen Sie einen Signalhandler auf, der die Anwendung geordnet mit Cancellation terminiert.

Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.