

→ „Erinnerung“ an LV „Betriebssysteme“
+ Besonderheiten bei BS zur Parallelverarbeitung

- 8.1 typ. Architekturen
- 8.2 Betriebsarten
- 8.3 Aktivitätsträger
- 8.4 Koordinierung
- 8.5 Speicherverwaltung
- 8.6 Scheduling
- 8.7 verteilte Filesysteme

8. Betriebssystemunterstützung

- Hauptaufgabe: Parallelität der HW auf verschiedenen Ebenen für Anwendungen verfügbar machen,
- d.h. effektive Beherrschung und Ausnutzung der verfügbaren Prozessoren
- i.allg.: BS für PR = erweiterte/modifizierte Einprozessor-BS
Erweiterungen/Modifikationen z.B. bzgl.
 - Scheduling/Zuordnung auf n Prozessoren
 - Speicherverwaltung
 - verteilte Dateisysteme
 - Lastverteilung

8.1 Architekturen von PR-Betriebssystemen (I)

Anforderungen an PR-BS:

- Effizienzverlust durch das BS ist bei PV viel kritischer als in „gewöhnlichen Umgebungen“, in denen der Rechner oft gar nicht voll ausgelastet ist
- nebenläufige Ausführung von BS-Mechanismen nötig, damit BS nicht selbst zum Engpass wird, daher auch gute BS-Struktur nötig

Typische BS-Architekturen für PR:

- Monolith. Systeme
- Mikrokern-basierte Systeme
- Beachtung objektorientierter Prinzipien

Spezielle BS-Architekturen wurden früher oft verwendet, sind heute aber nicht mehr typisch

8.1 Architekturen von PR-Betriebssystemen (II)

Monolithische Systeme

- Alle BS-Fkt. sind im monolith. Kern zusammengefasst
- Vorteile:
 - alle Kernfunktionen laufen im privilegierten Systemmodus und alle BS-Daten liegen im selben Adressraum, daher effiziente Implementierung, schneller Zugriff auf BS-interne Daten (schnelle interne Kommunikation) und Abschottung des Kerns gegenüber Anwendungen
- Nachteile:
 - nicht sehr flexibel: BS-Mechanismen können nicht einfach ausgetauscht oder weggelassen werden, da sie im Kern festgelegt sind
 - Seiteneffekte im Kern möglich
 - schwierige Erweiterbarkeit/Wartbarkeit/Portierung

8.1 Architekturen von PR-Betriebssystemen (III)

Mikrokern-basierte BS

- Kern mit Basisfunktionen zuzügl. ausgelagerte Systemprozesse
- Vorteile:
 - klare Struktur, Kapselung von BS-Fkt.-gruppen mit klaren Schnittstellen
 - dadurch können viele Teile aus dem Kern herausgenommen werden, diese Teile werden wie gewöhl. Anwenderprozesse bearbeitet
 - einfachere Anpassbarkeit an spez. Forderungen bei PR
- Nachteile:
 - Aufrufe zwischen ausgelagerten Systemprozessen sind zeitaufwändiger wegen Adressraumwechsel
 - daher oft doch „Rückverlagerung“ von Fkt. in den BS-Kern

Typischer Vertreter: MACH-Kernel als Basis viele prakt. BS

8.1 Architekturen von PR-Betriebssystemen (IV)

Prinzipien der Objektorientierung

- stärkere Beachtung der Prinzipien der Objektorientierung
- z.B. durch: feinere Modularisierung der BS-Komponenten und einheitl. Kommunikationsmechanismen
- BS-Module als Objekte in einem gemeins. Adressraum, dadurch effiziente Kommunikation (ähnl. wie bei monolith. Systemen), aber trotzdem Kapselung der Module gegeben
- durch klare Schnittstellen einfache Modifikation von Modulen sowie Integration neuer Module möglich

8.1 Architekturen von PR-Betriebssystemen (V)

Beispiele für PR-Betriebssysteme

- UNIX inkl. verschiedener Derivate und Ableitungen (z.B. IRIX, Solaris, PARIX)
- Linux (heute vor allem für Multicomputer/Cluster!)
- Mikrokern basierte BS
 - z.B. MACH: KSR-OS (KSR), OSF/1, SPP-UX (Convex)
 - z.B. Chorus: UNICOS-MAX (Cray)
- Windows (für größere PR-Systeme spez. Server-Versionen)

8.1 Architekturen von PR-Betriebssystemen (VI)

Möglichkeiten zur Integration der Kommunikation in das BS

a) Verteilte Umgebung

„gewöhnl.“ lokales BS + zusätzliche Kommunikation + Middleware

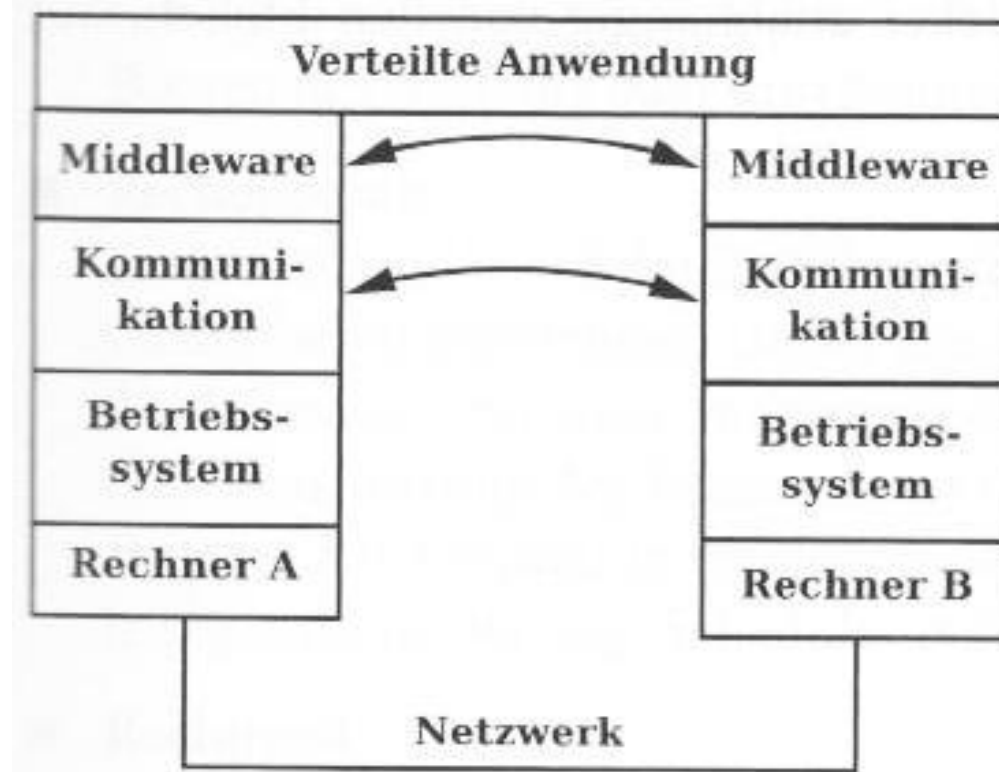


Bild-Quelle: Peschel: Nebenläufige und verteilte Systeme. Bonn: mitp, 2006

8.1 Architekturen von PR-Betriebssystemen (VII)

Möglichkeiten zur Integration der Kommunikation in das BS

b) Netzwerk-BS

„gewöhnl.“ lokales BS mit integrierter Kommunikation + Middleware

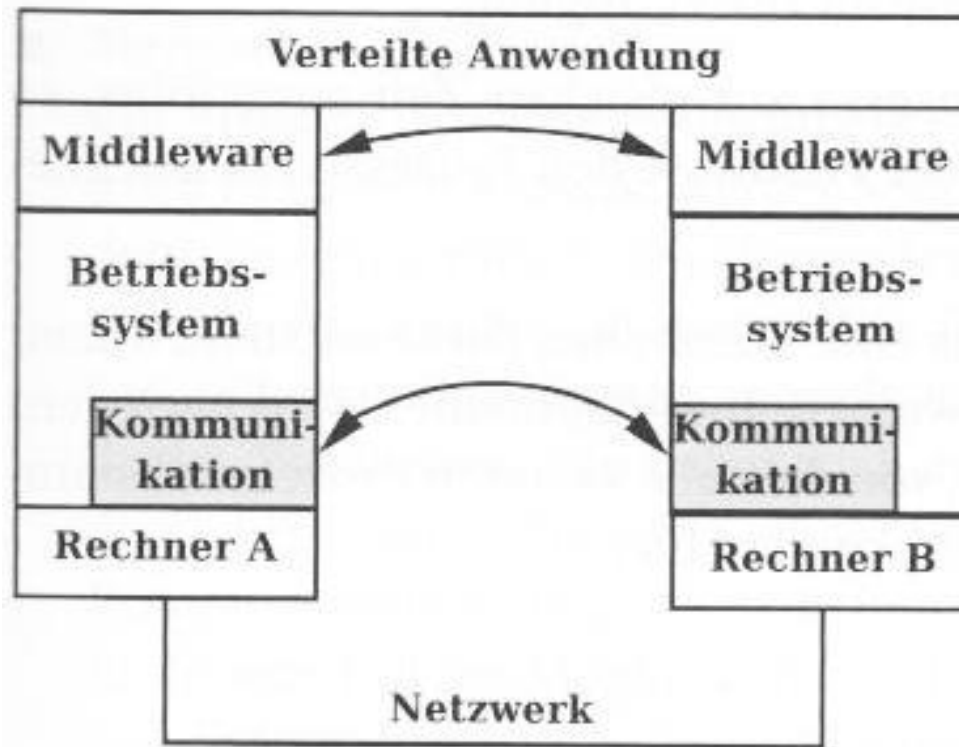


Bild-Quelle: Peschel: Nebenläufige und verteilte Systeme. Bonn: mitp, 2006

8.1 Architekturen von PR-Betriebssystemen (VIII)

Möglichkeiten zur Integration der Kommunikation in das BS

c) Verteiltes BS

BS mit integrierter Kommunikation und integrierter Verteilung

Nutzung der Verteiltheit ist transparent

Ressourcenzuteilung ist unabhängig vom Ort der Aktivität

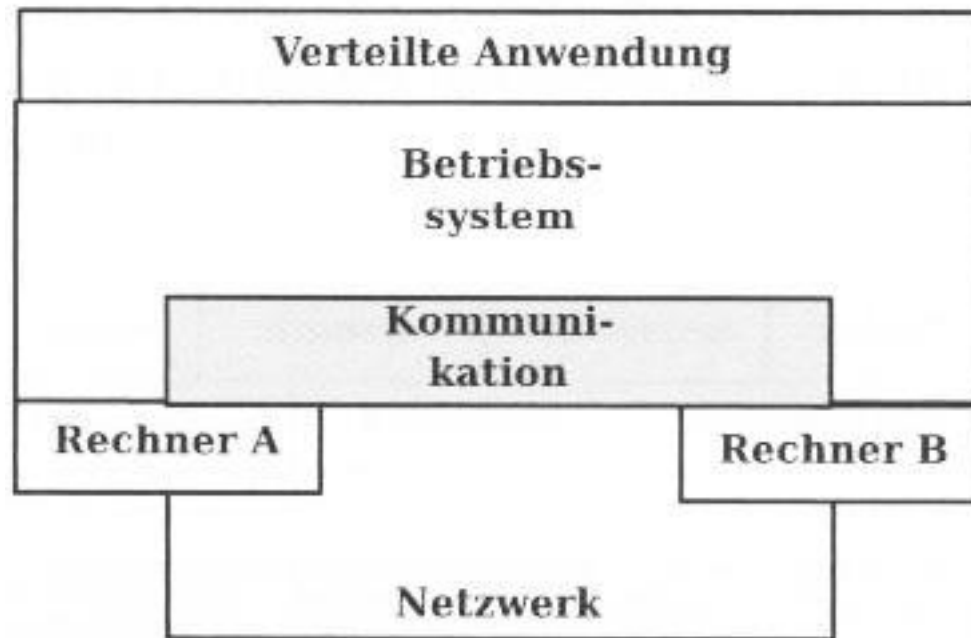


Bild-Quelle: Peschel: Nebenläufige und verteilte Systeme. Bonn: mitp, 2006

8.2 Betriebsarten

unterschiedlich,

je nach Aufbau des Gesamtsystems und Lokalisation des BS:

- je ein BS auf jedem Prozessor/Rechner
 - bei massiv-parallelen Systemen: Singletasking
 - bei Cluster u.ä.: optimiertes oder auch vollständiges BS (mit Multitasking bzw. Multithreading, Dialogbetrieb, ggf. Multi-User-Betrieb usw.)
- ein BS für alle Prozessoren (1 BS verwaltet sämtl. CPUs)
 - Multitasking/Multithreading
 - ggf. Kopplung mit einem Frontend (Batch-System)
- BS auf ggf. eigenem Frontend-Rechner
 - „Zugangs-/Verwaltungssystem“ für das parallele Gesamtsystem
 - Multitasking/Multithreading, meist Multi-User-Betrieb
 - Dialog- und Batch-Betrieb

8.2.1 Systeme mit gemeinsamem Speicher (I)

(zur Erinnerung: z.B. Multiprozessorsysteme, Multicores)

a) Asymmetrisches Multiprocessing (AMP)

- auf jedem Prozessor (bzw. Kern) läuft ein separates BS
 - homogen: auf jedem Prozessor dasselbe BS
 - heterogen: evtl. auf jedem (einigen) Prozessor ein anderes BS
- Ausführungsumgebung hier ähnlich wie bei Einprozessor-BS
daher einfache Portierung von (meist nicht-parallelem) Alt-Code
und Nutzung üblicher Tools (z.B. Debugger) möglich
- Aufteilung von Ressourcen für die parallelen Anwendungen durch
Entwickler vorab (mögl. unter Beachtung ausgewogener Last!),
Bereitstellung der Ressourcen (z.B. Speicher) statisch beim Booten
- Prozesse laufen immer auf denselben Prozessoren, ggf. bleiben
einige Prozessoren frei (trotz Überlast bei anderen)
d.h. Lastverteilung durch Prozess-Verlagerung schwierig/unmöglich
dafür aber kaum Cache-Invalidierungen, d.h. wenig Leistungsverlust

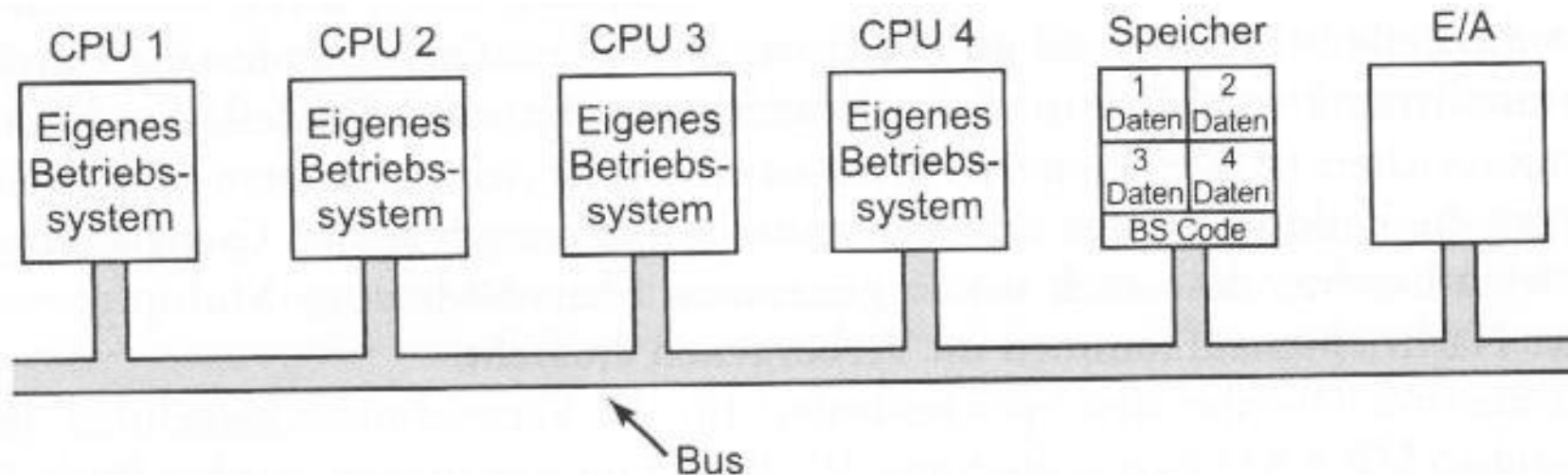
8.2.1 Systeme mit gemeinsamem Speicher (II)

a) Asymmetrisches Multiprocessing (AMP, Forts.)

Beispiel mit 4 identischen Betriebssystemen auf 4 CPUs

Die 4 BS teilen sich ihren Code im Speicher (homogenes AMP)
und ebenso die E/A-Geräte

Jedes BS hält seine eigenen Daten im gemeinsamen Speicher



8.2.1 Systeme mit gemeinsamem Speicher (III)

b) Symmetrisches Multiprocessing (SMP)

- alle Prozessoren/Kerne sind funktional identisch
- ein einziges BS (eine einzige Installation) betreibt alle Prozessoren/Kerne des gesamten Systems gleichermaßen
- BS kann auf jedem beliebigen Prozessor/Kern laufen
- jede Applikation kann auf jedem beliebigen Prozessor/Kern laufen
- BS kann dynamisch Last balancieren (kann den Applikationen Ressourcen/Prozessoren ohne Vorgabe durch Entwickler bereitstellen)
- BS stellt (bekannte) Mittel und API-Fkt. für Prozesse/Threads und zur Synchronisation bereit (z.B. pthreads, Locks, Semaphore)

8.2.1 Systeme mit gemeinsamem Speicher (IV)

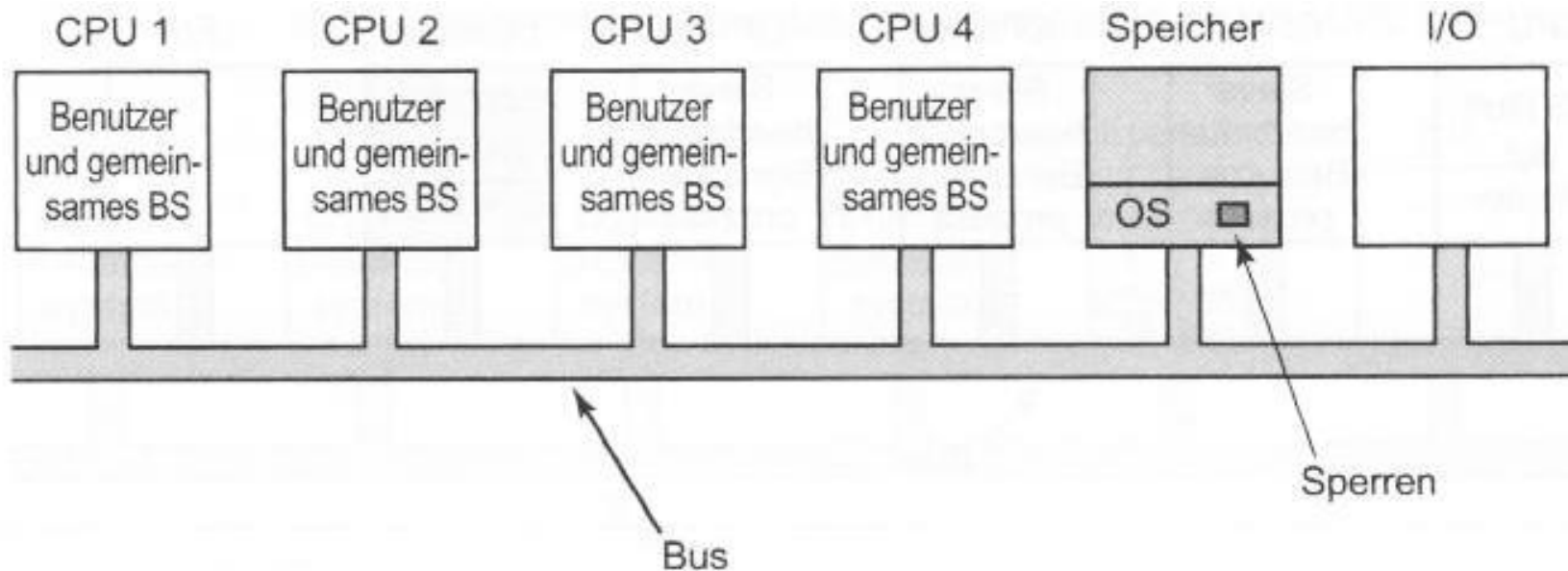
b) Symmetrisches Multiprocessing (SMP, Forts.)

- aber: mehrere Anwendungen könnten auf verschiedenen Prozessoren gleichzeitig dieselbe BS-Fkt. ausführen wollen! Daher Synchronisation bzgl. BS-Zugriffen nötig!
Lösungen:
 - gesamtes BS ist ein einziger krit. Abschnitt
„Floating Master“, d.h. BS „wandert“ zwischen den Prozessoren, jede Anwendung kann es nutzen, aber immer max. eine gleichzeitig
evtl. lange Wartezeiten bzgl. BS-Nutzung!
 - verschiedene Teile des BS (Code, aber auch Tabellen!) werden durch verschiedene (unabhängige!) krit. Abschnitte geschützt
Leistungsverbesserung, mehr Parallelität möglich
allerdings Aufwand bei BS-Implementierung und Deadlock-Gefahr für einzelne Komponenten des BS

8.2.1 Systeme mit gemeinsamem Speicher (V)

Beispiel für SMP:

Bild-Quelle: Tanenbaum: Moderne Betriebssysteme. München: Pearson Studium, 2002



8.2.1 Systeme mit gemeinsamem Speicher (VI)

c) Master-Slave-Systeme, „gebündeltes“ Multiprocessing (BMP)

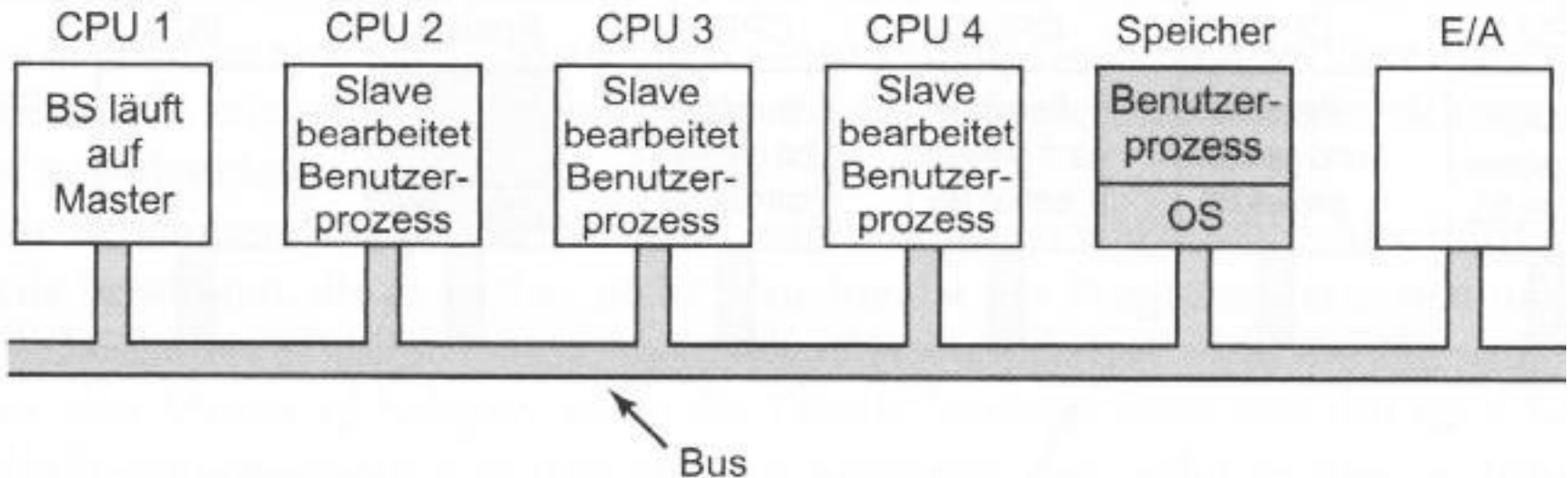
- Kombination aus Vorteilen von AMP und SMP
- alle Prozessoren/Kerne sind funktional identisch
- ein einziges BS (eine einzige Installation) betreibt alle Prozessoren/Kerne des gesamten Systems gleichermaßen, aber es läuft nur auf genau einem Prozessor/Kern = Master-Prozessor
- jede Applikation kann auf jedem beliebigen Slave-Prozessor/Kern laufen (notfalls sogar auch auf dem Master)
- alle BS-Rufe via API-Fkt. werden an den Master-Prozessor umgeleitet und dort behandelt
- gute Auslastung möglich, da BS freie Prozessoren sofort mit Anwendungen belegen kann
- auch Speicher wird dynamisch zugeteilt,
- keine Cache-Inkonsistenzen möglich, da nur ein Cache verwaltet
- Aber: Master ist der Engpass (insbes. bei größeren Systemen)

8.2.1 Systeme mit gemeinsamem Speicher (VI)

c) Master-Slave-Systeme, „gebündeltes“ Multiprocessing (BMP, Forts.)

Beispiel:

Bild-Quelle: Tanenbaum: Moderne Betriebssysteme. München: Pearson Studium, 2002



Einige BMP-Systeme erlauben dem Entwickler, Eingrenzungen bzgl. der Prozessoren vorzugeben (vgl. AMP)

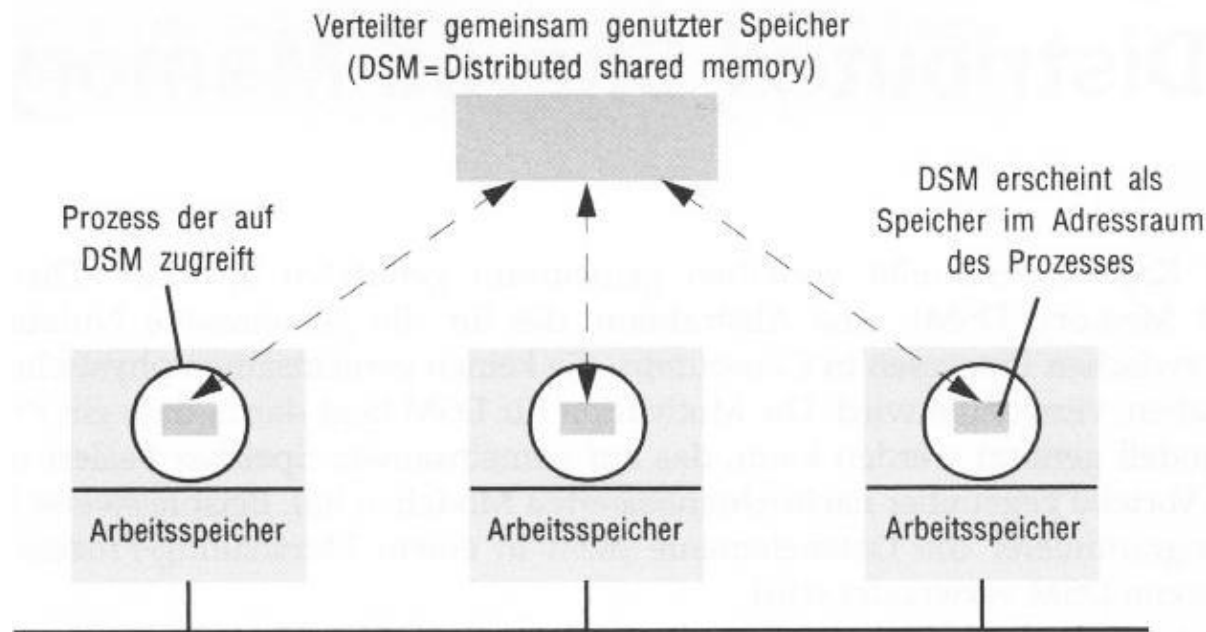
Generell bei Multiprozessorsystemen: Abgleich der lokalen Caches nötig! HW-Lösungen, nicht durch BS.

8.2.2 Systeme mit verteiltem gemeinsamen Speicher DSM (I)

(Zur Erinnerung: physikalisch kein gemeinsamer Speicher, aber virtuell gemeinsamer Adressraum für die verschiedenen Prozessoren)

→ DSM ist eine Abstraktion!

gemeinsamer Adressraum aus Sicht der Prozesse muss durch Systemsoftware „künstlich“ erzeugt werden, jeder Prozess muss Daten-Änderungen anderer Prozesse auf anderen Prozessoren „sehen“ können



Quelle: Colouris , u.a.:
Vereilte Systeme.
München: Pearson
Studium, 2002

8.2.2 Systeme mit verteiltem gemeinsamen Speicher DSM (II)

- Vergleich: a) echter gemeinsamer Speicher
b) verteilter gemeinsamer Speicher (DSM durch BS implementiert)
c) verteilter gemeinsamer Speicher (DSM durch höhere SW implementiert)

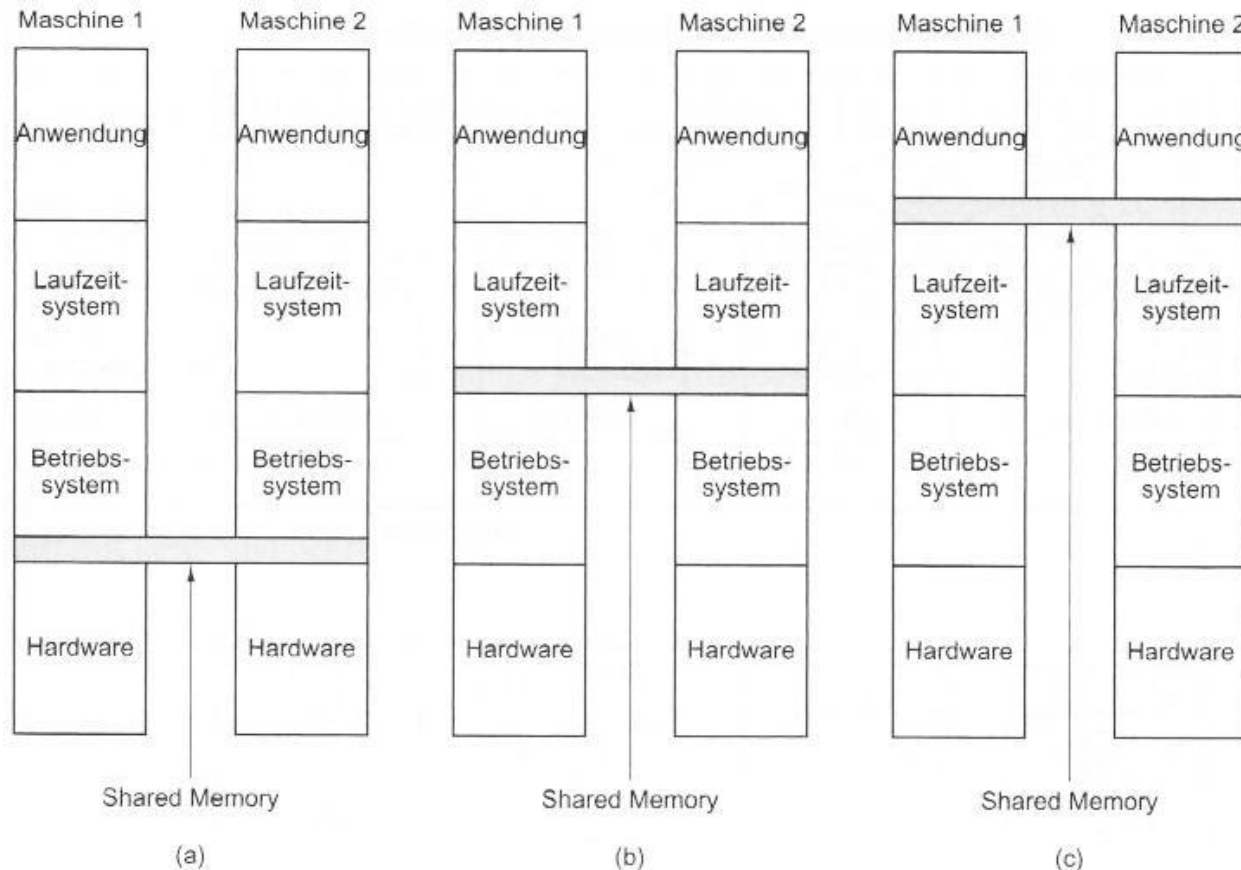


Bild-Quelle:
Tanenbaum:
Moderne
Betriebssysteme.
München:
Pearson Studium,
2002

8.2.2 Systeme mit verteiltem gemeinsamen Speicher DSM (III)

Das BS erspart den Programmierern den expliziten Nachrichtenaustausch zwischen ihren parallelen Anwendungen.

Implementierungsvarianten für DSM:

- HW-basiert
- seitenbasiert
- objektbasiert

a) **HW-basiertes DSM**

- Zugriffe auf entfernte Speicherzellen werden durch eine MMU abgefangen
- Kommunikationshardware leitet den Zugriff dann (über ein schnelles Verbindungsnetz) auf die entfernte Speicherzelle um
- insbes. günstig für kleine Speichereinheiten
- Bsp: NUMA-Systeme

8.2.2 Systeme mit verteiltem gemeinsamen Speicher DSM (IV)

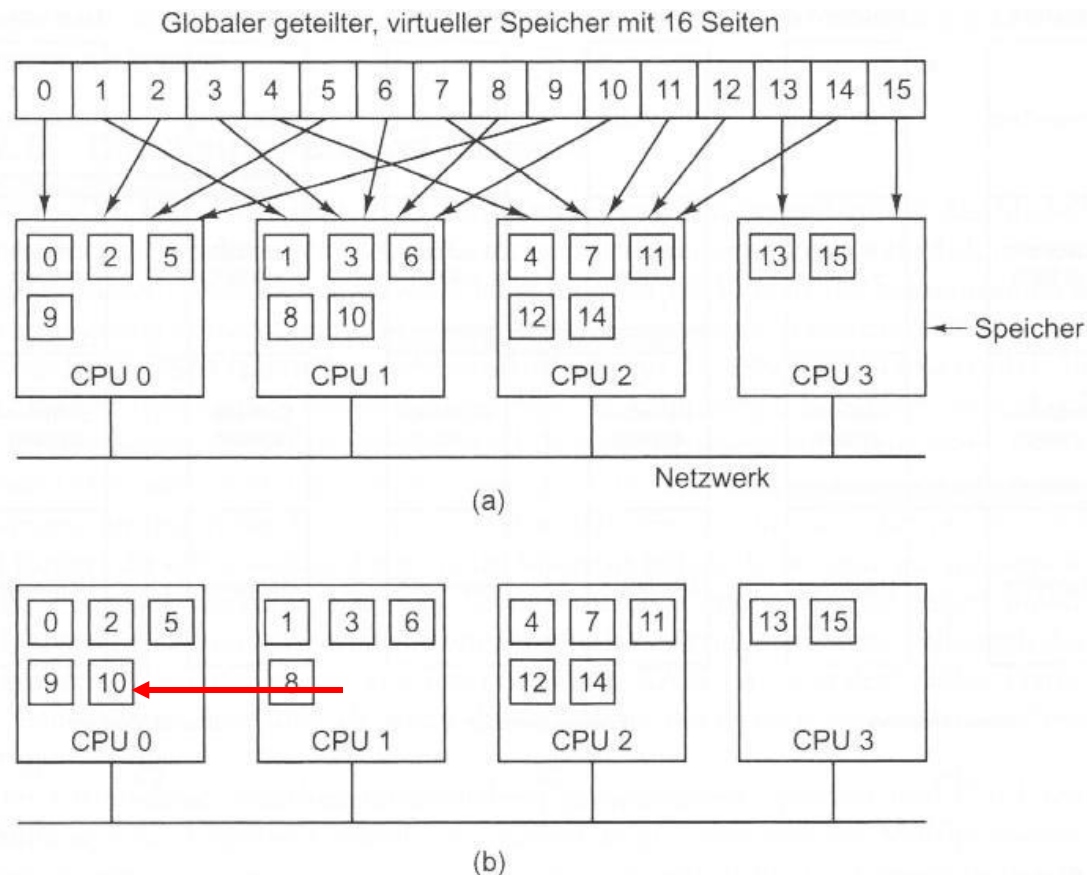
b) Seitenbasiertes DSM

- DSM wird als Bereich im virtuellen Speicher implementiert, der im Adressraum jedes Prozesses den selben Bereich belegt
- Basis der Verteilung sind also Seiten (pages)
- Zugriffe auf entfernte Seiten werden wie Seitenzugriffsfehler des Virt. Speichers behandelt
d.h. Speicherkohärenzproblem wird durch Behandlung eines Seitenzugriffsfehlers gelöst!
- Realisierung durch BS, Middleware oder auch mit HW-Unterstütz.
- setzt i.allg. einheitl. Daten- und Pagingformate auf allen Knoten voraus
- Bsp.: Ivy, Munin, Mirage, Clouds, Choices, COOL

8.2.2 Systeme mit verteiltem gemeinsamen Speicher DSM (V)

b) Seitenbasiertes DSM (Forts.)

Bsp.: Bild-Quelle: Tanenbaum: Moderne Betriebssysteme. München: Pearson Studium, 2002



a) Ausgangslage: 4 CPUs mit insges. 16 gemeinsamen DSM-Seiten und akt. lokaler Verfügbarkeit

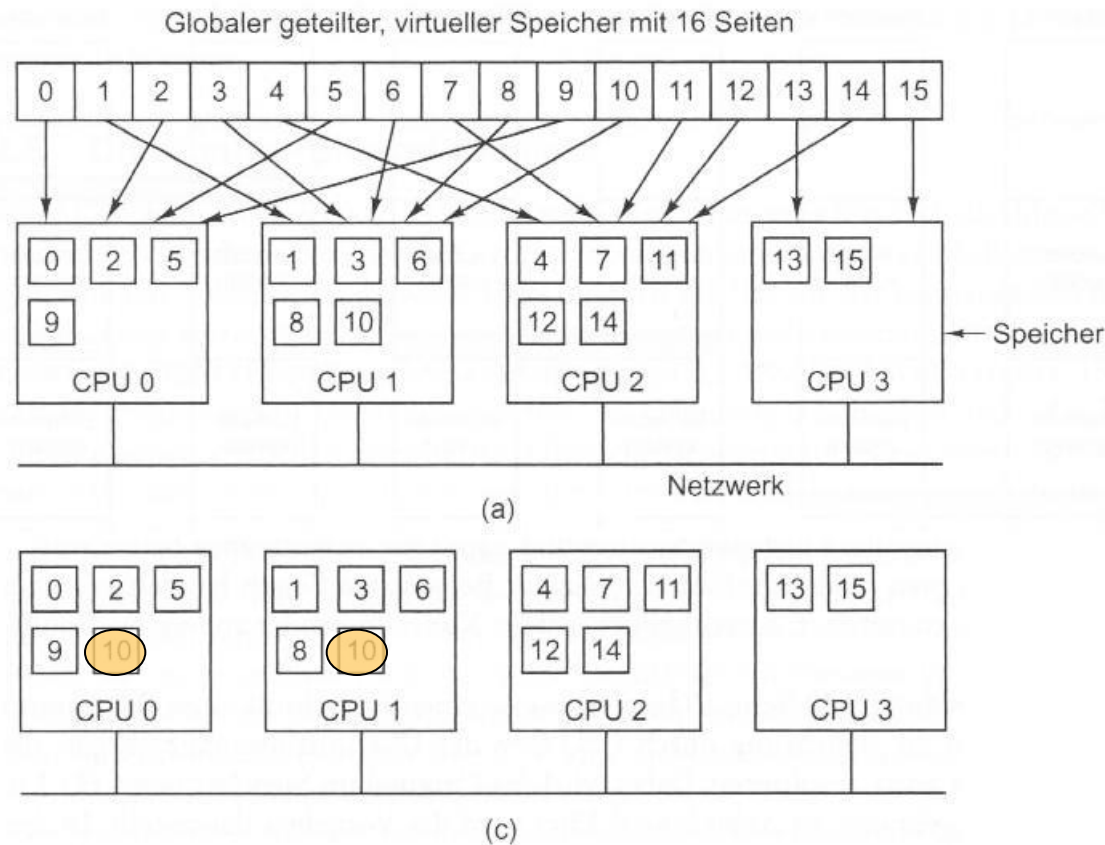
Lokale Seiten von CPU0 sind 0,2,5,9.

b) Falls CPU0 auf die entfernte Seite 10 zugreifen will, wird nach dem Seitenzugriffsfehler die DSM-Software die Seite 10 von CPU1 zu CPU0 verschieben

8.2.2 Systeme mit verteiltem gemeinsamen Speicher DSM (VI)

b) Seitenbasiertes DSM (Forts.)

Bsp.: Bild-Quelle: Tanenbaum: Moderne Betriebssysteme. München: Pearson Studium, 2002 (angepasst)



a) Ausgangslage: 4 CPUs mit insges. 16 gemeinsamen DSM-Seiten und akt. lokaler Verfügbarkeit

Lokale Seiten von CPU0 sind 0,2,5,9.

Annahme: Seite 10 ist *read-only* (z.B. Code)!

b) Falls CPU0 auf die entfernte Seite 10 zugreifen will, wird diese zu CPU0 kopiert (repliziert)!

8.2.2 Systeme mit verteiltem gemeinsamen Speicher DSM (VII)

b) Seitenbasiertes DSM (Forts.)

Größe der Verteilungseinheit (= Seiten bzw. Vielfache davon)

- Vorteil großer Einheiten: weniger Datentransfers nötig,
- Nachteil großer Einheiten: längere Netzwerkbelastung beim Transfer, längere Blockade von Seitenzugriffsfehlern anderer Prozesse!
- Mögliches Problem hier: „False Sharing“
 - in einer Seite liegen z.B. (zufällig!) zwei unabhängige Variablen a,b
 - CPU1 benutzt oft a, CPU2 benutzt oft b
 - Folge: die Seite wird oft zwischen CPU1 und CPU2 hin- und her geschoben!
 - manchmal kann ein kluger Compiler diesen Effekt vermeiden ...

8.2.2 Systeme mit verteiltem gemeinsamen Speicher DSM (VIII)

c) Objektbasiertes DSM

- Zugriffe auf Variablen-Inhalte möglich
- Variablenzugriffe auf verteilte Objekte werden nicht vom BS, sondern von Compilern oder Laufzeitsystemen abgefangen und durch entfernte Zugriffe ersetzt
- dazu werden i.allg. spez. Bibliotheksroutinen für DSM zur Anwendung dazu gelinkt
- Einheit der Verteilung sind Objekte (vgl. ooP) oder Tupel
- hier keine Forderung mehr nach Homogenität der HW oder des BS auf den Knoten, daher für heterogene Systeme geeignet.
- semantische Informationen aus den Programmen können zur Optimierung der Verteilung entfernter Zugriffe benutzt werden
- allerdings müssen die Anwendungen an diese DSM-Form angepasst werden
- Bsp.: Orca (in Amoeba), Linda, JavaSpaces, TSpaces

8.2.2 Systeme mit verteiltem gemeinsamen Speicher DSM (IX)

Grundsätzlich:

Sicherung der Speicherkonsistenz nötig
dazu diverse Verfahren ...

Replikation (Replizierung)

= *mehrfache Speicherung* von Daten an verschiedenen Orten
Aktualität der Replikate kritisch (Abgleich nötig, z.B. via atomarer Transaktionen)

Migration

= *Verschieben* eines Prozesses von einem Ort (Prozessor) an einen anderen (keine Replikation!)

Dazu muss die Prozessbeschreibungsstruktur und der Adressraum verschoben werden und die Verbindungen des Prozesses mit Ressourcen und anderen Prozessen sind zu aktualisieren

8.2.3 Systeme mit verteiltem Speicher (I)

(zur Erinnerung: z.B. Multicomputer, Cluster, ...)

Unterschiede innerhalb dieser Klasse:

Element	Multicomputer	Verteiltes System
Knotenkonfiguration	CPU, Speicher, Netzwerk	Vollständiger Rechner
Knotenperipherie	Geteilt, evtl. selbe Platte	Vollständiger Knoten
Ort	Selber Raum	Evtl. weltweit
Kommunikation der Knoten	Spezielle Verbindung	Herkömmliches Netz
Betriebssysteme	Viele gleiche	Evtl. alle verschieden
Dateisysteme	Eins, gemeinsam	Jeder Knoten eigenes
Administration	Eine Organisation	Viele Organisationen

Bild-Quelle: Tanenbaum: Moderne Betriebssysteme. München: Pearson Studium, 2002 (angepasst)

8.2.3 Systeme mit verteiltem Speicher (II)

Aus BS-Sicht:

im Prinzip Verwendung üblicher Einprozessor-BS auf den Knoten inkl. leistungsfähiger Kommunikations- (Netzwerk-) Software + Middleware! (insbes. bei verteilten Systemen)

Typische Verfahren zur IPC im BS hierfür:

- Nachrichtenaustausch
 - entfernte Prozedur- (Methodenaufrufe) wie RPC/RMI
- sowie Middleware wie verteilte Dateisysteme + Verzeichnisdienste, CORBA, Jini usw.

erforderliche besondere Funktionen (Bsp. Cluster):

- einheitliches Systemerscheinungsbild (single image system)
- Lastverteilung (später...) durch Prozessmigration
- Jobverwaltung (ohne genaues Zielsystem angeben zu müssen)
- Checkpointing (Rücksetzpunkte speichern)

...

8.3 Aktivitätsträger (I)

- zur parallelen Ausführung auf mehreren CPUs sind mehrere Aktivitätsträger (in der Regel Threads) nötig
- Anwendungen können sich nicht oder nur mit großem Programmieraufwand auf die aktuelle vorliegende CPU-Zahl einstellen
- aber BS kann helfen...

Mögliche Thread-Implementierungen:

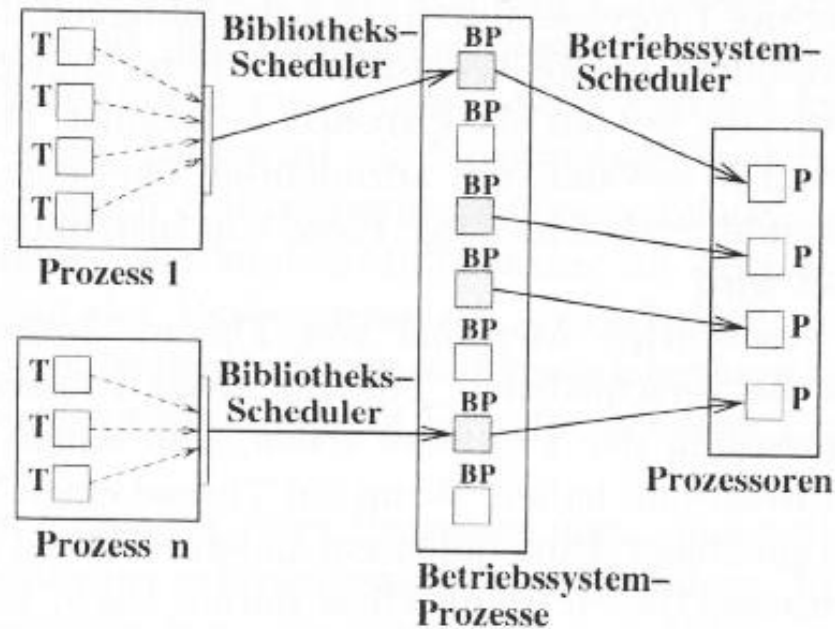
- Threads auf Benutzerebene (user level threads) mittels Thread-Bibliotheken (z.B. POSIX pthreads)
- Threads im BS auf Kernel-Ebene (kernel-level threads)

Typische Zuordnungen:

- n user-level threads zu 1 Prozess (bei BS ohne eig. Multithreading)
- 1 user-level threads zu 1 kernel thread (bei BS mit eig. Multithreading)
- n user-level threads zu m kernel threads (- // -)

8.3 Aktivitätsträger (II)

Threadzuordnung
n : 1



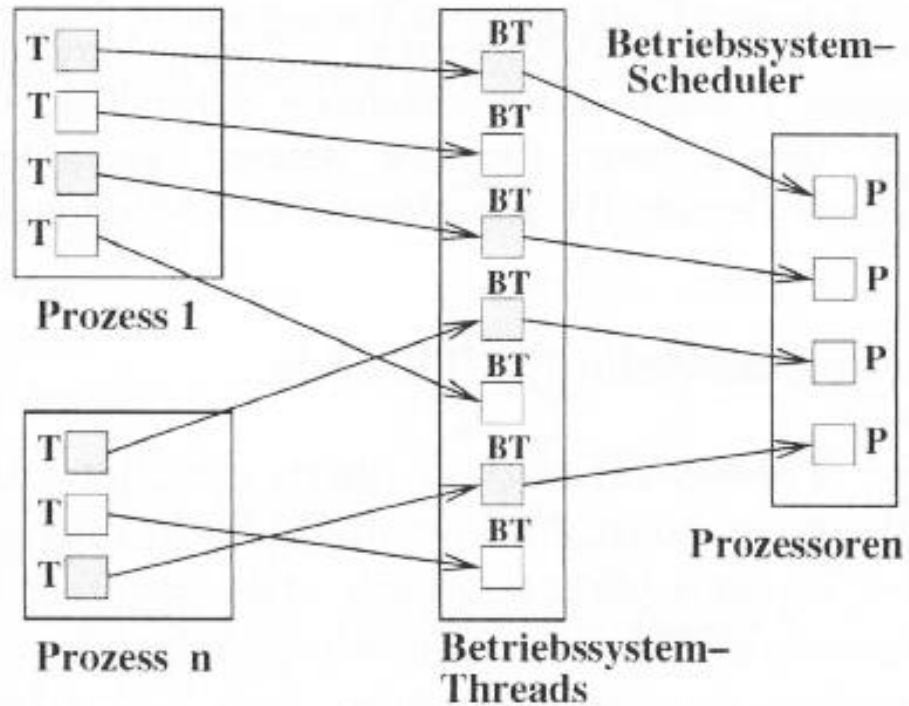
N:1-Abbildung – Thread-Verwaltung *ohne* Betriebssystem-Threads. Der Scheduler der Thread-Bibliothek wählt den auszuführenden Thread T des Benutzerprozesses aus. Jedem Benutzerprozess ist *ein* Betriebssystemprozess BP zugeordnet. Der Betriebssystem-Scheduler wählt die zu einem bestimmten Zeitpunkt auszuführenden Betriebssystemprozesse aus und bildet diese auf die Prozessoren P ab.

Bild-Quelle:
Rauber/Rünger: Muticore:
Parallele Programmierung.
Springer, 2008

8.3 Aktivitätsträger (III)

Threadzuordnung

1 : 1



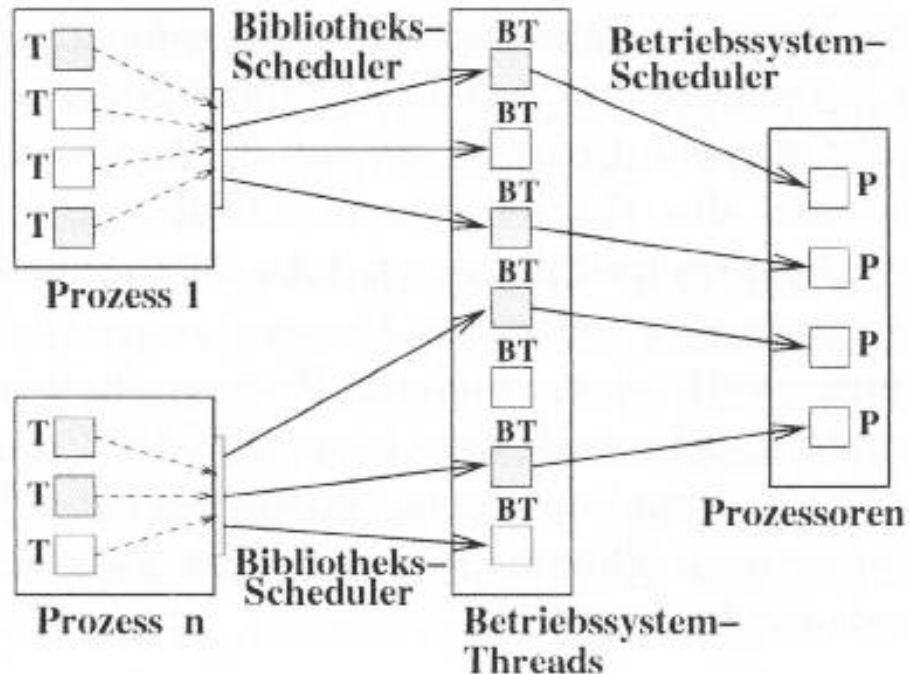
1:1-Abbildung – Thread-Verwaltung mit Betriebssystem-Threads. Jeder Benutzer-Thread T wird eindeutig einem Betriebssystem-Thread BT zugeordnet.

Bild-Quelle:
Rauber/Rünger: Muticore:
Parallele Programmierung.
Springer, 2008

8.3 Aktivitätsträger (IV)

Threadzuordnung

n: m



N:M-Abbildung – Thread-Verwaltung mit Betriebssystem-Threads und zweistufigem Scheduling. Benutzer-Threads T verschiedener Prozesse werden einer Menge von Betriebssystem-Threads BT zugeordnet (N:M-Abbildung).

Bild-Quelle:
Rauber/Rünger: Muticore:
Parallele Programmierung.
Springer, 2008

8.4 Koordination

Konkurrenz- und Reihenfolge-Probleme existieren auch bei parallelen Prozessen (sowohl zwischen Anwendungen als auch versch. BS-Prozessen!)

gemeinsamer Speicher verfügbar?

- Ja: bekannte Verfahren aus Einprozessor-BS anwendbar (z.B. Semaphore)
- Nein: komplizierte Verfahren nötig, denn:
im verteilten System gibt es i.allg. keine zentrale Instanz bzw.
keine Garantie für „gleichen Status/Zeit“ auf allen Systemen!
daher „Ersatzlösungen“ unter Nutzung der Kommunikation

8.4.1 Sperrsynchronisation (I)

Zur Lösung des wechselseitigen Ausschlusses

Bei PV: „*Lieber Daten sperren als Code sperren*“, weil dadurch andere Prozesse, die nicht dieselben Daten brauchen, parallel laufen können.

Primitive Synchronisationsverfahren (z.B. INT-Sperre) sind bei PR wirkungslos, weil sie nur für den lokalen Prozessor wirken

Typische Mittel zur Sperrsynchronisation:

- **Sperran (Locks)** mit den Funktionen lock()/unlock()
- **Transaktionsverfahren**

8.4.1 Sperrsynchronisation (II)

Sperren (Locks)

- **Spin Locks:**
 - Sperre mit aktivem Warten („spinning“)
 - Implementierung z.B. mittels atomarer Befehle wie TS oder XCHG
 - `spin_lock()`, `spin_unlock()`
 - kann aktives Warten akzeptiert werden?
oft ja, weil Sperrdauer oft gering und alternativer Prozesswechsel zu zeitaufwändig sowie Gefahr der Cacheveränderung
 - Sonderform: Reader-/Writer-Locks: erlauben parallele Lesezugriffe aber nur exklusiven Schreibzugriff (z.B. in Linux)
- **Semaphore („Queued Locks“):**
 - passives Warten, wobei Zugriff auf Warteliste mittels Spin Lock

8.4.1 Sperrsynchronisation (III)

Sperren (Locks)

Nachteile von Locks:

- aktives Warten beschränkt die parallele Ausführung
- Fehlende Synchronisation (fehlende Sperren) kann zu Fehlern führen
- Verklemmungsgefahr
- Gefahr der Prioritätenumkehr
- bei Abbruch eines Prozesses, der eine Sperre gesetzt hat, wird die Sperre nicht wieder freigegeben und blockiert damit ggf. andere Prozesse

8.4.1 Sperrsynchronisation (IV)

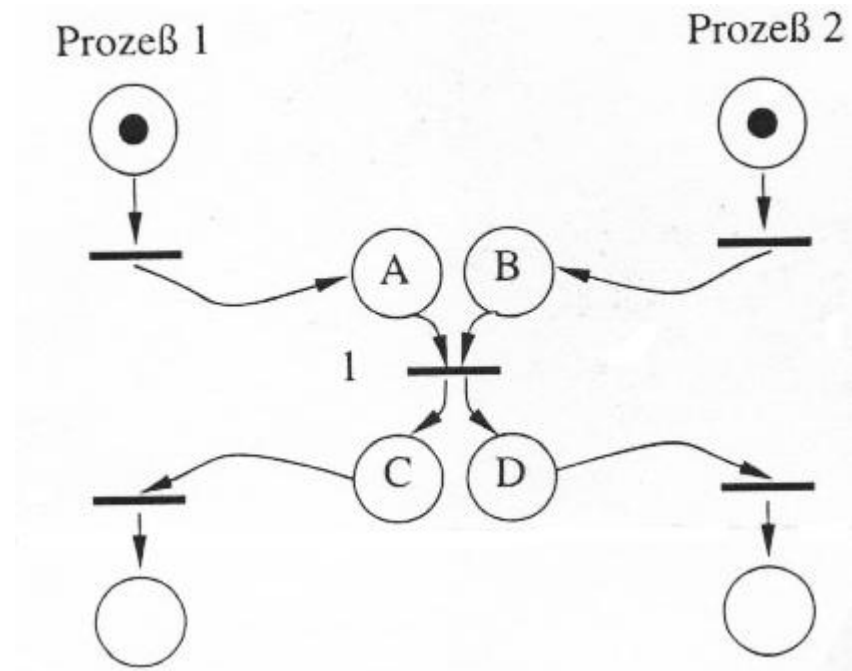
Transactional Memory (TM)

- ist ein Konkurrenz-Kontroll-Konstrukt auf Basis von Transaktionen
- „lock-free“, vermeidet damit Nachteile von Locks
- „wait-free“ (jeder Thread kommt weiter, egal, was die anderen tun!)
- **Memory Transaction** = Folge von Operationen auf gemeinsamem Sp., die entweder komplett ausgeführt und festgeschrieben (commit) oder abgebrochen (abort) und ohne Veränderung zurückgesetzt wird
Transaktion → atomare Operation in Bezug auf konkurr. Threads
kein anderer Thread kann eine dazu in Konflikt stehende Op. tun.
- Implementierung von Transaktionen:
 - in SW: Software Transactional Memory (STM) mittels API-Fkt.
 - direkt in der HW: Hardware Transactional Memory (HTM)
z.B. mittels Caches und Cachekohärenzprotokollen
 - hybride Formen

8.4.2 Ereignissynchronisation

Barrieren (barrier, Schranke, Hürde)

- Schranke für mehrere Threads: kein Thread kann hinter der Schranke fortfahren, bevor nicht alle anderen Threads an der Schranke angekommen sind
- „Rendezvous“ von n Threads
- Anwendung z.B. bei Iterationen (zur Sicherung, dass jeweils alle Threads den selben Iterations-Schritt ausführen)



8.4.3 Kommunikation

Shared Memory

- nur bei Systemen mit physikal. gemeinsamem Speicher möglich!

Nachrichtenaustausch (message passing)

- für alle Systeme möglich
- in Systemen ohne gemeins. Speicher einziges Mittel zur Synchronis.!
- zahlreiche Varianten in BS für PR
 - mit/ohne Synchronisation beim Sender und/oder Empfänger
 - Adressierung Sender/Empfänger direkt/indirekt
 - Art der Verbindung: 1:1, M:1, 1:N, M:N, multicast/broadcast
 - mit/ohne Zwischenspeicherung
- diverse Implementierungen
 - Mailbox, Message Queue, Pipes ,...
 - Prozedur-Fernaufruf (RPC), Methoden-Fernaufruf (RMI),...

8.4.4 Synchronisation ohne gemeinsamen Speicher (I)

Ausgangslage: auf jedem System gibt es jemanden, der (lokal) den wechselseitigen Ausschluss sichert

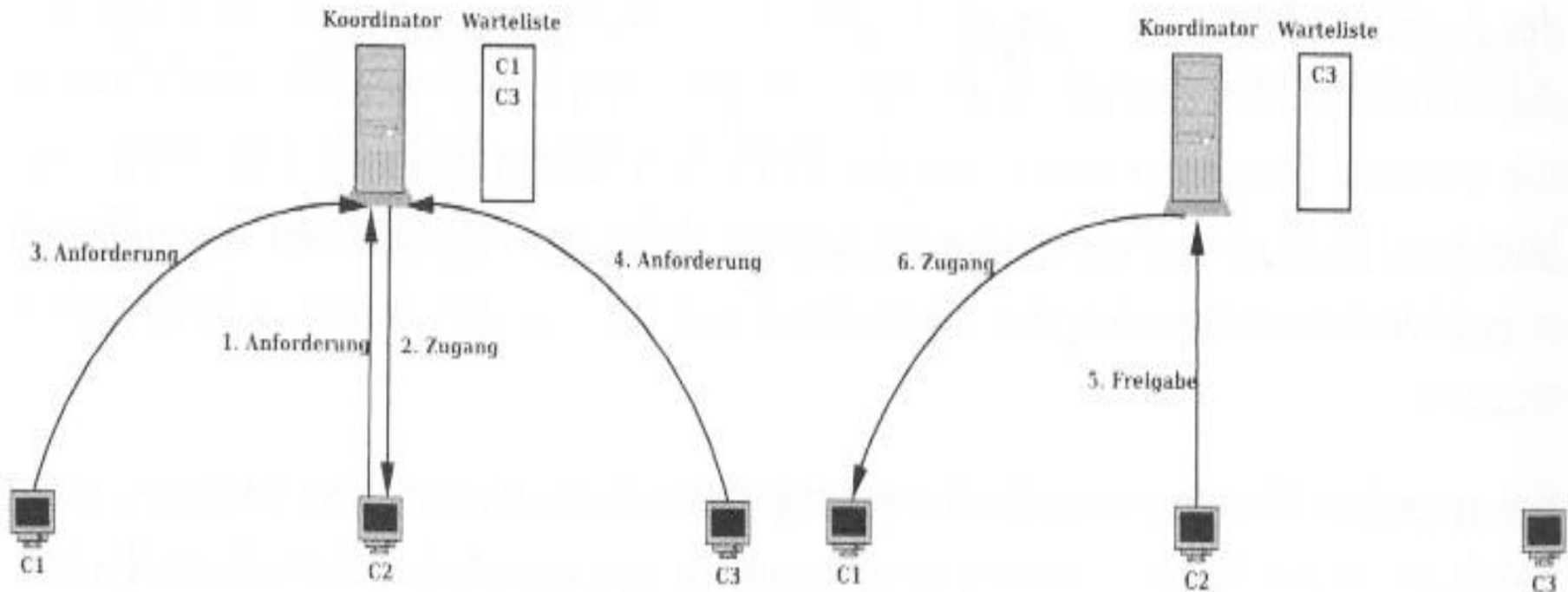
Problem: „Zusammenarbeit“ dieser Einrichtungen auf allen Rechnern nötig zwecks Erreichung eines verteilten WA („Abgleich“ nötig)

Lösungen:

- **zentrale Lösung: ein Knoten ist das alleinige Kontrollsystem**
 - nur dieses kontrolliert Zugriff auf alle exklusiven Objekte
 - nur dieses hat sämtl. Infos über die zu verwaltenden Objekte
 - Prozess mit Nutzungswunsch muss sich via Nachricht an seinen lokalen Verwalter wenden
 - dieser sendet Anforderung an zentrale Kontrollinstanz und erwartet Erlaubnis-Antwort bzgl. Zuteilung/Berechtigung
 - Freigabe via Nachricht vom Prozess an Kontrollknoten
 - Nachteile: Kontrollknoten ist Engpass (Ausfall?)
hohe Kommunikationslast für Anford./Freigabe

8.4.4 Synchronisation ohne gemeinsamen Speicher (II)

Bsp. für zentrale Lösung



8.4.4 Synchronisation ohne gemeinsamen Speicher (III)

- **verteilte Lösung: ein System ist das alleinige Kontrollsystem**
 - alle Systeme (Knoten) verfügen über dieselbe Menge an Inform.
 - jeder Knoten kennt nur Teilbild des gesamten Systems und muss seine Entscheidungen auf dieser Basis treffen
 - alle Knoten tragen dieselbe Verantwortung bzgl. Entscheidung
 - Ausfall eines Knotens führt i.allg. nicht zum Versagen des Gesamtsystems
 - es gibt keinen systemweiten gemeins. Taktgeber für gleiche Zeitbasis

Hauptproblem sind also die Verzögerungen bei der Kommunikation und das Fehlen einer einheitlichen Zeitbasis!

Daher sind Lösungen sehr aufwändig! Bsp.:

- verteilte Warteschlangen, Multicast, Zeitstempel, Wahlverfahren
- Ringverfahren (Tokenweitergabe)

8.5 Speicherverwaltung (I)

zumeist ähnlich wie in konventionellen/universellen BS

Typische Beispiele:

- NORMA mit eigenem Bediensystem (z.B. bei MPP-Systemen)
 - distributed memory!
 - auf den Knoten nur statische Speicherverwaltung
 - quasi Stapelbetrieb (für SPMD-Modell, [später]), Annahme und Zuteilung von Jobs an die CPUs durch Bediensystem
- NORMA mit (verteilter) BS auf jedem Knoten
 - distributed memory!
 - spezif. Speicherverwaltung entsprechend der HW des Knotens
 - Swapping/Paging bei Bedarf, aber Zeitverlust
- UMA-Systeme
 - shared memory!
 - echt paralleler Zugriff auf gemeins. Speicher möglich
 - meist virt. Speicher mit Paging

8.5 Speicherverwaltung (II)

- NUMA-Systeme
 - distributed (virtual) shared memory!
 - d.h. physikalisch verteilter Speicher muss zu logisch gemeinsamem Adressraum zusammen gefasst werden:
 - via Software: sehr aufwändig wegen eines hohen Kommunikationsbedarfs
 - HW-Unterstützung zur Cache-Kohärenz bietet bessere Lösung
 - noch besser: Möglichkeit zur gezielten Platzierung von Speichersegmenten anhand ihrer Eigenschaften, z.B.:
 - „Prozess-privat“: Segm. in lokalen Speicher des Prozesses legen
 - „Knoten-privat“: Segm. an lokalen Speicher des Knotens binden
 - „nah-gemeinsam“: Zugriff für alle Prozesse möglich, Platzierung im lokalen Speicher des Erzeugers
 - „entfernt-gemeinsam“: Zugriff für alle Prozesse möglich, Segm. wird in Seiten verteilt über den globalen Speicher platziert
 - ...

8.6 Scheduling

Architektur des PR-Systems hat großen Einfluss auf geeignetes Scheduling

- Gemeinsamer Speicher??
- Gleichartige Prozessoren??
- ...

(Aktuell) hauptsächlich interessante Fälle:

- Multiprozessorsysteme
- Multicomputer/Cluster

8.6.1 Scheduling in Multiprozessorsystemen (I)

Vgl. Scheduling in **Uniprozessorsystemen** vs. **Multiprozessorsysteme:**
Scheduling „eindimensional“ „mehrdimensional“
Welchen Prozess als Welchen Prozess als
nächsten ausführen? nächsten ausführen
und auf welcher CPU?

Außerdem zu klären:

Sind alle Prozesse unabhängig voneinander oder gibt es „Gruppen“ von untereinander abhängigen Prozessen?

In der Praxis:

oft Kombination und BS-spezif. Abwandlung der folg. Verfahren

8.6.1 Scheduling in Multiprozessorsystemen (II)

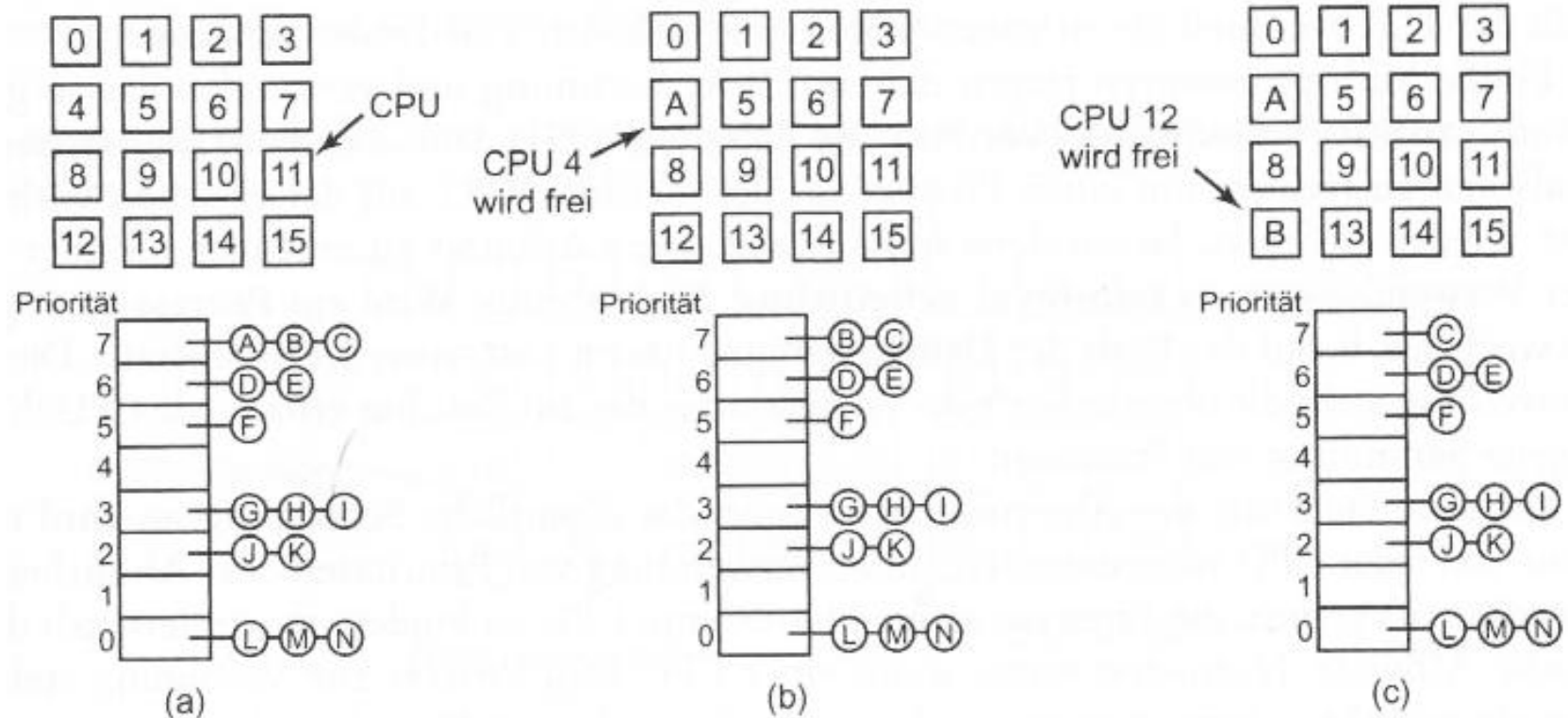
Beispiel: **Load Sharing** („geteilte Last“, list scheduling)

- für unabhängige Prozesse
(z.B. durch Aufträge unabh. Benutzer beim Time-Sharing-Betrieb)
- eine einheitl. Datenstruktur zur Verwaltung laufbereiter Prozesse
Realisierungsbeispiele:
 - eine einzige ready-Liste
→ sinnvolle Strategien: **RR, FCFS, SRT, ...**
 - eine Menge von ready-Listen
→ sinnvolle Strategien: versch. **Priorität** je Liste
- wird eine CPU frei, so wird ihr ein laufbereiter Prozess zugeteilt
- Vorteil: quasi automatischer Lastausgleich
- Nachteile: Zugriff auf globale Datenstruktur ist Engpass
Bei Zuweisung einer anderen CPU für einen unterbrochenen Prozess: Cache neu aufbauen!

8.6.1 Scheduling in Multiprozessorsystemen (III)

Beispiel: Load Sharing

Bild-Quelle: Tanenbaum: Moderne Betriebssysteme. München: Pearson Studium, 2002



16 (beschäftigte) CPUs, 14 laufbereite Prozesse in 8 Listen mit versch. Priorität

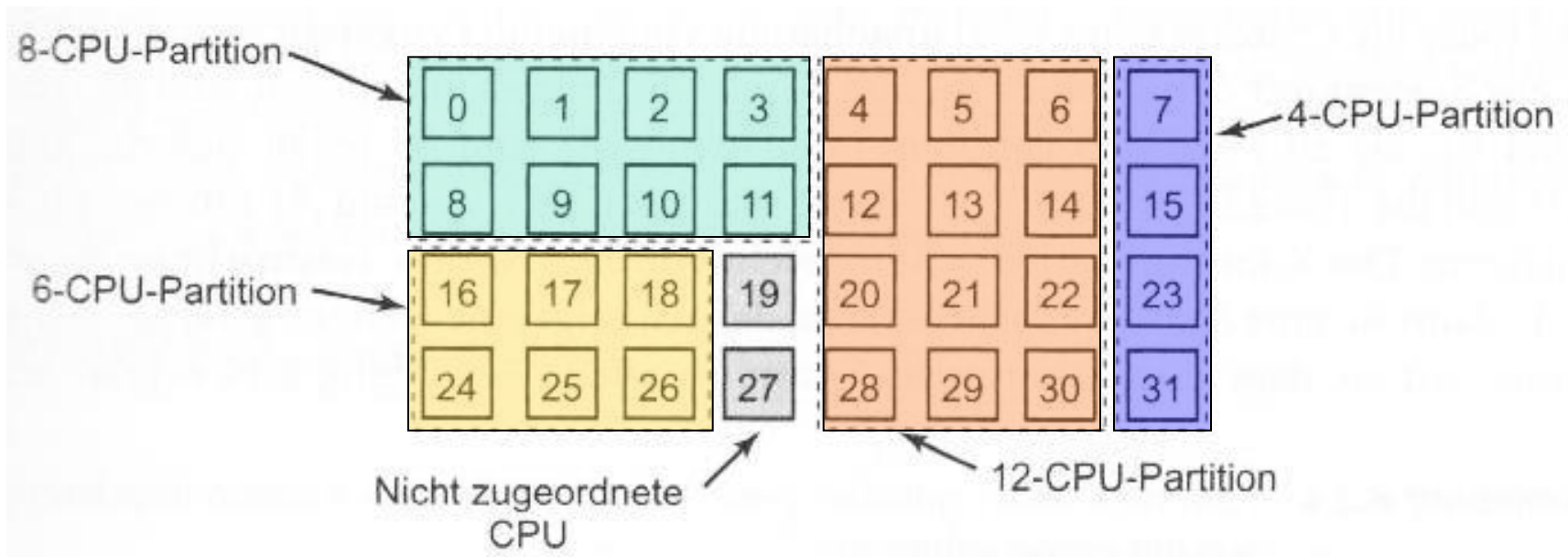
8.6.1 Scheduling in Multiprozessorsystemen (IV)

Beispiel: **Space Sharing**

- **Partitionierung** der Prozessormenge in Teilmengen: jede Partition führt mehrere Prozesse/Threads aus
- bei Erzeugung einer Prozess- oder Threadmenge wird eine ausreichend große Partition von freien Prozessoren gesucht
- bei Erfolg: Zuteilung/Belegung durch die Prozesse/Threads
- gelingt die Zuordnung nicht, müssen die Prozesse warten, bis eine ausreichend große Partition frei ist
- blockierte Proz./Thr. bleiben ihrer CPU zugeordnet
- beendete Proz./Thr. geben CPU frei
- Scheduling erneut, wenn eine neue Gruppe von Proz./Thr. kommt
- Nachteil: ggf. bleiben CPUs in einer Partition frei
- Anzahl und Größe der Partitionen oft fest, manchmal dynamisch

8.6.1 Scheduling in Multiprozessorsystemen (V)

Beispiel: Space Sharing



32 Prozessoren

4 unterschiedlich große CPU- Partitionen

zur Zeit 2 freie CPUs, Aufträge mit Bedarf > 2 CPUs müssen warten

8.6.1 Scheduling in Multiprozessorsystemen (VI)

Beispiel: Space Sharing

Hauptproblem: geeignete dynamische Partitionierung finden

- einfacher Ansatz: Prozess fordert n Prozessoren und erhält alle (oder keine!)
- Alternative: zentraler Server mit „Wissen“ über die Prozesse, ihren min. und max. CPU-Bedarf usw.
Jede CPU fragt periodisch beim Server an, welche anderen CPUs sie zur weiteren Erfüllung ihrer Aufgabe noch hinzuziehen darf
Ggf. wird Anzahl der Proz/Thr. (+/-) angepasst,
Bsp.: nur noch 2 freie CPUs und Anforderung einer 4er Partition:
der Server wird z.B. den nächsten 2 anfragenden CPUs den Befehl geben, in den Leerlauf zu wechseln, damit dadurch eine freie 4er Partition gebildet und zugeteilt werden kann
- weitere „ausgefeilte“ Algorithmen zur „optimalen“ Zuordnung

8.6.1 Scheduling in Multiprozessorsystemen (VII)

Beispiel: Space Sharing

Bsp. für weitere „ausgefeilte“ Algorithmen zur „optimalen“ Zuordnung:

- **Largest Size Scheduling (LS)**
 - Größe = Parallelitätsgrad des Jobs
 - größere Jobs = höhere Prio
 - nicht zuteilbare Jobs bekommen in jeder Scheduling-Runde eine Prio-Erhöhung
 - Nach Terminierung eines Jobs versucht der Scheduler mögl. viele Jobs aus der Ready-Liste Prozessoren zuzuteilen. Reichen die freien CPUs für einen lafbereiten Job mit max. Prio nicht aus, so wird der Scheduler vorübergehend inaktiv
- **Smallest Number of Processors First (SNPF)**
 - Gegenteil von LS



8.6.1 Scheduling in Multiprozessorsystemen (VIII)

Beispiel: Gang Scheduling (Gruppen-Scheduling)

- Gang = zusammenhängende Prozesse (z.B. alle Kindprozesse mit ident. Elternprozess) bzw. Menge aller Threads eines Prozesses
- Idee: alle Prozesse der Gruppe (bzw. alle Threads eines Prozesses) werden zeitgleich für eine best. Dauer (Quantum, Δt) einer Menge von Prozessoren zugeteilt, also dynamische CPU-Zuteilung
- bei blockierten oder beendeten Gruppenmitgliedern bleibt die betreffende CPU für den Rest dieses Quantums einfach frei d.h. kein Scheduling „zwischendurch“
- Vorteil: geringerer Verwaltungsaufwand (weg. Gruppenzuteilung)
 weniger Prozesswechsel
 geringerer Zeitverlust bei Kommunik. zw. Gruppenmitglied.
- Nachteil: ggf. Vergeudung von Rechenzeit auf (teilw.) freien CPUs

8.6.1 Scheduling in Multiprozessorsystemen (IX)

Beispiel: Gang Scheduling (Gruppen-Scheduling)

Zeit Prozessoren 


A1	A2	A3	A4	A5	B1	B2	B3
C1	C2	C3	C4	C5	C6	D1	E1
E2	E3	E4	E5	E6	E7	E8	E9
A1	A2	A3	A4	A5	B1	B2	B3
C1	C2	C3	C4	C5	C6	D1	E1
E2	E3	E4	E5	E6	E7	E8	E9

Beispiel:

8 Prozessoren

5 Anwendungen mit
Threads:A(5), B(3), C(6),
D(1), E(9)In 3 Zeitscheiben
können alle Threads
aller 5 Anwendungen
einmal für ein
Quantum laufen

8.6.1 Scheduling in Multiprozessorsystemen (X)

Beispiel: Affinity Scheduling

- Idee: Affinität eines Prozesses zu einer bestimmten CPU beachten also Versuch, einen Prozess (immer oder möglichst oft) auf der CPU laufen zu lassen, auf der er auch vorher lief
- Vorteil: sein Cache (und ggf. lokaler Speicher) ist noch „ok“, damit Geschwindigkeitsgewinn,
- Nachteil: Aufwand zur Bestimmung der Affinität und deren Sicherung
- hard processor affinity: Prozess bleibt immer auf „seiner“ CPU ggf. CPU-Verschwendung, aber kein Prozesswechselaufwand!
- soft processor affinity: abgeschwächt, besserer Lastausgleich

8.6.2 Scheduling in Multicomputern/Clustern (I)

→ andere HW-Situation als bei Multiprozessoren:

- jeder Knoten hat eigenen Speicher und eigene Prozessmenge
- zentrale Liste lafbereiter Prozesse klappt nicht
- nach Zuweisung eines Prozesses an einen Knoten erfolgt Scheduling nach jeweiliger Strategie des lokalen BS (vgl. UNIX/Linux o.a.)
- umso mehr Gewicht kommt der vorherigen Zuweisung des Prozesses zu einem Knoten zu!
 - Prozessor-Allokation, -Zuteilung
 - entsch. Rolle des Batch-Systems (auf dem Frontend/Master)
- was sollte (außer lokalem BS) noch auf jedem Knoten laufen?
 - Client des Batch-Systems
 - Client zum Monitoring

8.6.2 Scheduling in Multicomputern/Clustern (II)

Batch-System, Jobverwaltungssystem

- Aufgabe:
 - Nutzeraufträge für rechenintensive Anwendungen (Jobs) annehmen
 - Starten der Jobs auf einer Menge von Knoten/Prozessoren, sobald genügend frei sind
- Ziel: bessere Auslastung, Fairness unter den Benutzern sichern

dazu sind Nutzerangaben zum Job hilfreich bzw. nötig, z.B.:

- Anzahl benötigter CPUs
- (geschätzter) Rechenzeitbedarf
- Kommunikationsbedarf
- Speicherbedarf
- ggf. weitere Eigenschaften wie HW/SW-Besonderheiten von Knoten

8.6.2 Scheduling in Multicomputern/Clustern (III)

Batch-System

- Nutzer gibt Auftrag an Batchsystem inkl. Anforderungen/Bedarf
- Batchsystem registriert den Job inkl. Zeitstempel
- Jobsystem berechnet Prio des Jobs anhand div. Merkmale
- sobald Knoten/Prozessoren frei sind, wird ihnen der höchstpriorre Job übergeben, dessen Anforderungen von den Knoten erfüllbar sind
- Verwaltung der Jobs in Queues
 - Queues können ggf. auch einzelnen Knoten oder Knotengruppen zugeordnet sein
- sollte *Checkpointing* unterstützen: erlaubt Unterbrechung von Jobs, Sicherung ihres Status, ihrer Daten usw.
 - ist hilfreich zum Wiederanlauf nach Abstürzen
 - erlaubt ggf. auch Job-Verlagerung auf anderen Knoten (Job-Migration)

8.6.2 Scheduling in Multicomputern/Clustern (IV)

Möglichkeiten des Job-Scheduling auf Batch Systemen:

- **First Come First Served FCFS**

- einfachste Strategie
- aber evtl. schlechte CPU-Auslastung
(Bsp.: ein früher eingetroffener Job mit Bedarf von 16 CPUs
blockiert trotz 15 freier CPUs evtl. nachfolgende kürzere
Jobs)

8.6.2 Scheduling in Multicomputern/Clustern (V)

- **Backfilling**

- bessere Gesamtauslastung gegenüber FCFS, weil ggf. kürzere bzw. niederprioritäre Jobs vorgezogen werden, sofern einige CPUs frei sind
- Der Job mit der höchsten Priorität wird gestartet, sobald die von ihm angeforderten Ressourcen verfügbar sind.
- Falls nicht genügend CPUs oder Hauptspeicher verfügbar sind, muss der Job mit der höchsten Priorität warten, bis andere Jobs enden und ihre Ressourcen freigeben. In diesem Fall wird ein Job mit niedrigerer Priorität gestartet, wenn er folgende Bedingungen erfüllt:
 - Er fordert weniger Ressourcen an, als noch verfügbar sind.
 - Seine angeforderte Verweilzeit ist kürzer als die aus den angeforderten Verweilzeiten der laufenden Jobs abzusehende Wartezeit für den höchstprioritären Job.

8.6.2 Scheduling in Multicomputern/Clustern (VI)

Fairshare

- versucht faire Aufteilung bzgl. verschiedener Nutzer
- bisheriger Ressourcenverbrauch von Anwendungen jedes Nutzers werden bei Prio-berechnung berücksichtigt, d.h. Jobs von Nutzern, die noch nicht viel gerechnet haben, werden gegenüber solchen von Nutzern, die schon viel gerechnet haben, bevorzugt.

• Exclusive

- pro CPU wird nur ein (oder nur ein rechenintensiver) Job angenommen
- Minimierung von Umschaltzeiten, Verbesserung der Durchlaufzeit

8.6.2 Scheduling in Multicomputern/Clustern (VII)

Beispiele von Batch-Systemen für Cluster:

- *OpenPBS*
 - Vorläufer PBS (Portable Batch System) war Grundlage für POSIX1003.2d, an dem sich auch weitere Batch Systeme orient.
 - für UNIX/Linux
 - Scheduler kann modifiziert werden
 - unterstützt Checkpointing
 - kommerzielle Version PBS Pro
 - ähnliches neueres System TORQUE
- *Sun Grid Engine*
 - basiert auf *Codine* von Gridware
 - als Open Source und als erweiterbare kommerzielle Lösung verfügbar
 - graf. Konfigurationswerkzeuge
 - hostorientierte Zuordnung der Queues
 - Scheduler anpassbar
 - unterstützt Checkpointing

8.6.2 Scheduling in Multicomputern/Clustern (VIII)

Maui („the most advanced scheduler in the world“ ...)

- für Cluster und Supercomputer, sehr leistungsfähig
- Community-Projekt, Quellen verfügbar/anpassbar
- besitzt „Meta-Scheduler“, der verschiedene anderer Scheduler (z.B. von OpenPBS) benutzt und erweitert
 - zahlreiche versch. Verfahren zur Job-Prio-Berechnung
 - Vorab-Ressourcenreservierung
 - bestimmte Ressourcen können bestimmten Jobs zugesichert werden (QoS)
 - Fairness-Regeln definierbar
 - beherrscht Backfilling

8.6.2 Scheduling in Multicomputern/Clustern (IX)

Weiteres nützliches Tool für Cluster: **Cluster-Management-System**

- Administration weg. verteilter Struktur meist schwierig
- Cluster-Management-Systeme verfügbar
 - vom Cluster-Hersteller
 - Open Source
- Hauptziele: leichtere Cluster-Administration, Automatisierung von Routineaufgaben
- Nachteil: erfordern viel Detailwissen
- Beispiele:
 - OSCAR (Open Source Cluster Application Resources)
Sammlung von Tools zur Linux-Cluster-Administration inkl. „Cluster Command & Control“, Bibliotheken für MPI/PVM, Batch-Systeme OpenPBS und Maui usw.
 - Rocks
Eine für Cluster optimierte Linux-Distribution basierend auf RedHat, inkl. Bibliotheken für MPI, Batch-Systemen und Überwachungsprogr., berücksichtigt verschiedene Knotentypen im Cluster

8.6.3 Scheduling in Grids (I)

Besonderheiten:

- Grid-Ressourcen sind über versch. administrative Domänen verteilt!
- hohe Dynamik im Grid:
 - Anwendungen mit völlig unterschiedl. Anforderungen
 - Ressourcen-Pool im Grid ändert sich oft bzgl. Verfügbarkeit u.ä.
 - Anwendungen brauchen oft verschied. Ressourcen gleichzeitig
- wer „beherrscht“ die jeweils lokalen Ressourcen in einer Domäne?
→ der lokale Scheduler!
- was kann dann noch der Grid-Scheduler tun?
 - agiert im Auftrag eines Grid-Nutzers
 - bezieht Infos aus einem Grid-Informationssystem
 - führt 3 Schritte aus:
 - Ressourcensuche
 - Systemauswahl
 - Jobausführung

8.6.3 Scheduling in Grids (II)

Schritt 1: Ressourcensuche

ermittelt Menge von nutzbaren Ressourcen mit best. Anforderungen anhand von Vor-Wissen über die Anwendung (Job)

1. Herausfiltern von Grid-Ressourcen, für die kein Zugriffsrecht besteht
2. Definition von nutzerspezif. Anforderungen des Jobs an Ressourcen (z.B. best. BS, best. Prozessor, Speicherbedarf, Antwortzeitforderungen, Kosten,...)
3. Filtern der noch verfügbaren Ressourcen bzgl. Anforderungen

8.6.3 Scheduling in Grids (III)

Schritt 2: Systemauswahl

Auswahl eines benutzbaren und geeigneten Rechnersystems aus dem Grid:

1. Dynamische Informationsbeschaffung mittels Grid-Informationssystem
2. Systemauswahl und damit dynam. Lastverteilung im Grid (soweit der Grid-Scheduler dazu berechtigt ist, ansonsten weiter durch den lokalen Scheduler) und ggf. unter Hilfe entspr. Lastverteilungssoftware (z.B. *Condor*)

8.6.3 Scheduling in Grids (IV)

Schritt 3: Jobausführung

1. optional: Reservierung von Ressourcen
2. Jobübergabe an das ausführende System inkl. Transfer benötigter Daten usw.
3. Jobüberwachung zur Laufzeit
4. Beendigung des Jobs inkl. „Aufräumen“

8.7 verteilte Dateisysteme

8.7.1 Überblick (I)

Filesystem

- zur persistenten Speicherung praktisch unverzichtbar
- enthält die Dateien als auch Verwaltungsinformationen (Attribute der Dateien inkl. Angaben zur Lokalisierung, Verzeichnisse usw.)
- *Lokales* Filesystem:
 - lokale Dateiverwaltung des BS auf dem jeweiligen Knoten (sofern dort verfügbar), Zugriff zu lokalen ext. Speichern
 - Bsp.: ext3fs/ext4fs (Linux), NTFS (Windows)
- *verteiltes (Netzwerk-)Dateisystem*:
 - Dateisystem zur Verwaltung von Dateien, die sich auf unterschiedl. Knoten eines verteilten Systems befinden
 - für Anwendungen erscheint der Zugriff zu entfernten Dateien ähnlich wie zu lokalen Dateien
 - Bsp.: NFS, CFS/SMB, AFS, DFS, xFS, ...

8.7.1 Überblick (II)

Anforderungen an verteilte Dateisysteme aus Benutzersicht:

- *Ortstransparenz:*
Pfadname einer Datei sollte den konkreten Speicherort nicht sichtbar machen
- *Ortsunabhängigkeit:*
Verschieben einer Datei von Knoten A nach B sollte keine Änderung des Pfadnamens bewirken

Besonderes Problem für verteilte Dateisysteme:

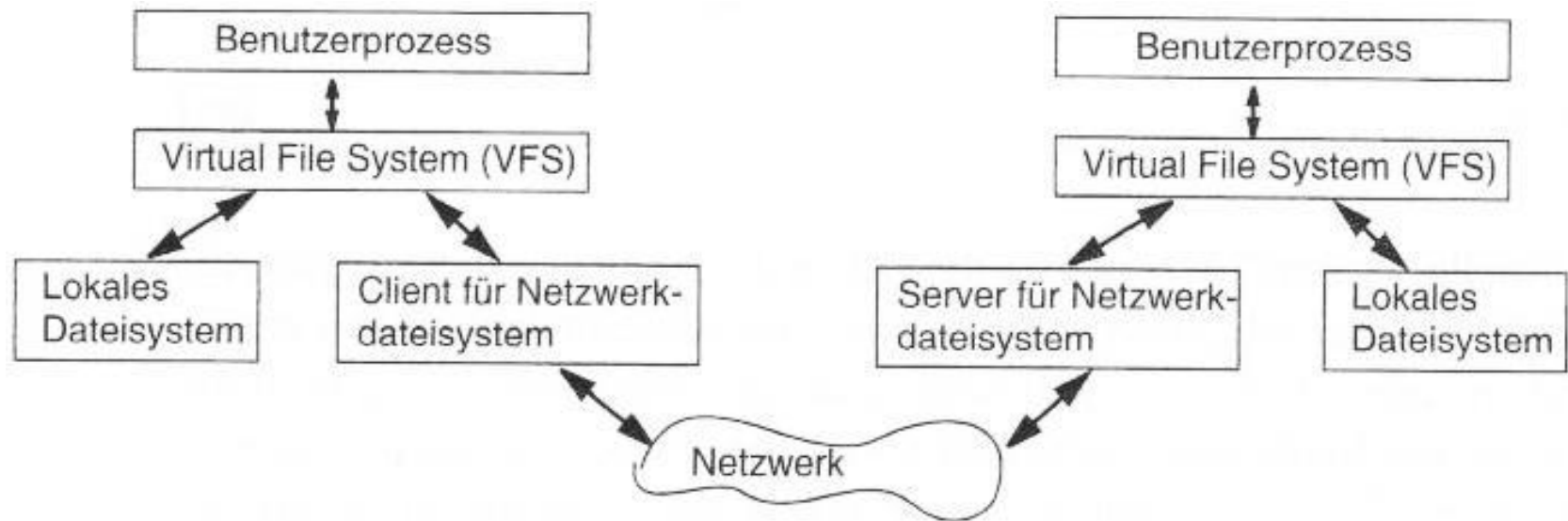
→ Konsistenzsicherung bei mehrfachen/verteilten Zugriffen!

8.7.1 Überblick (III)

Realisierung verteilter Dateisysteme:

→ **Client – Server – Modell**

- Zwischenschicht (Bsp.: VFS) bietet Anwendungen eine einheitliche Schnittstelle zu lokalen und verteilten/entfernten Dateisystemen
- jeder Knoten kann Server als auch Client sein



8.7.1 Überblick (III)

Modelle für den Zugriff auf entfernte Dateien bei Client-Server-System:

a) *Upload/Download*

- Datei wird beim Öffnen vom Server geholt und als lokale Kopie auf dem Client gespeichert (Download)
- Alle weiteren Dateizugriffe erfolgen lokal auf dem Client
- Nach dem Schließen der Datei wird sie auf den Server zurück gebracht (Upload)
- Vorteil:
 - einfach implementierbar,
 - effizienter Zugriff
- Nachteil:
 - Aufwand für Dateitransport,
 - Problem der Reihenfolge-Einhaltung bei parallelen Schreibzugriffen auf verschiedenen Clients

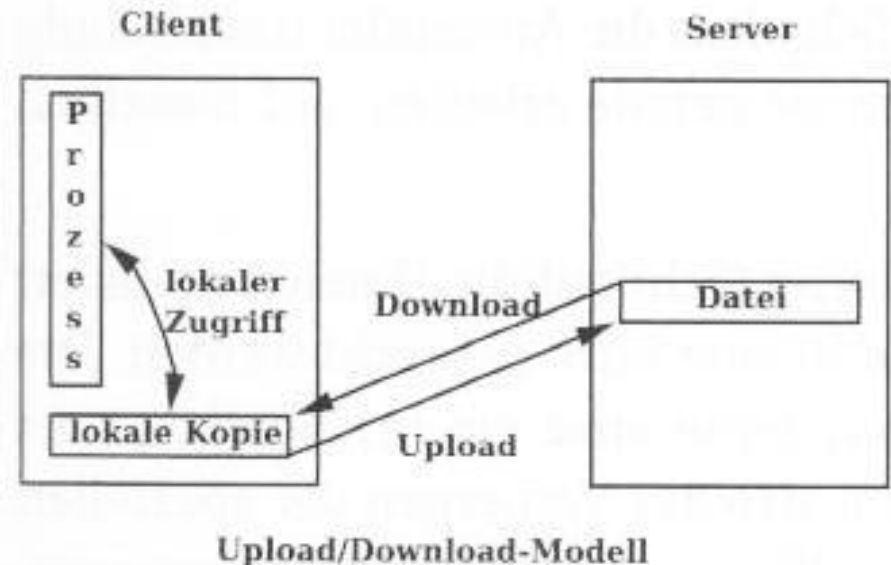
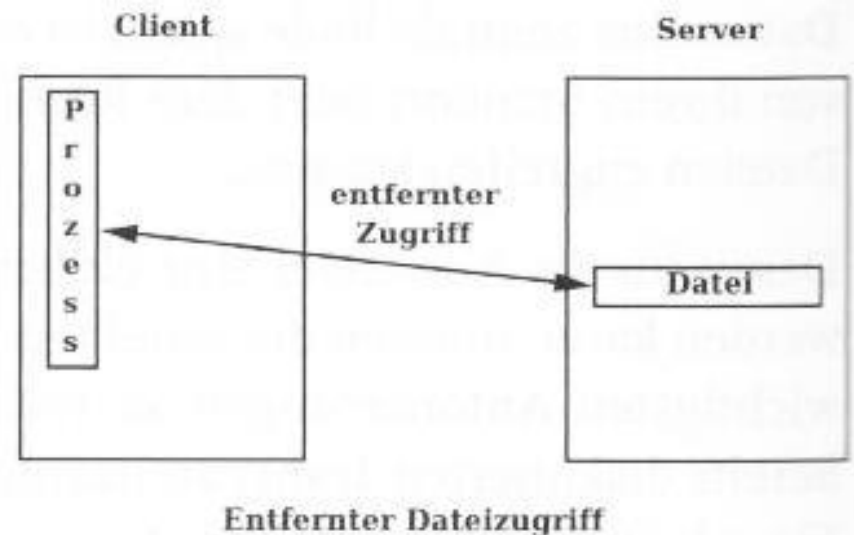


Bild-Quelle: Peschel: Nebenläufige und verteilte Systeme. Bonn: mitp, 2006

8.7.1 Überblick (IV)

b) *Entfernter Dateizugriff*

- Datei existiert nur auf dem Server
- Client hat keine Information über die Datei
- alle Dateizugriffe werden nur auf dem Server ausgeführt
- Vorteile: schlanker Client (keine lokale Kopie nötig),
kein Dateitransfer zwischen Client und Server nötig,
einfache Konsistenzsicherung zentral auf dem Server möglich
- Nachteil: Effizienzverlust,
da jeder Dateizugriff entfernt
ausgeführt werden muss



8.7.1 Überblick (V)

Nutzungseigenschaften von Dateien sind besonders wichtig für die Realisierung verteilter Dateisysteme:

- Dateien werden häufiger gelesen als geschrieben
- viele Dateien sind temporär (kurze Lebensdauer)
- ein Prozess verwendet nur rel. wenige Dateien
- die meisten Dateien sind eher klein
- parallele Dateizugriffe sind eher selten

→ Klassifikation von Dateien möglich, z.B.:

- temporäre Dateien, die nur von einem Prozess (und damit nie parallel) benutzt werden
- ausführbare Dateien, die nie oder nur selten verändert werden
- private Dateien, die nur jeweils einem Benutzer gehören
- usw.

... sollte durch das vert. Filesystem beachtet werden!

8.7.1 Überblick (VI)

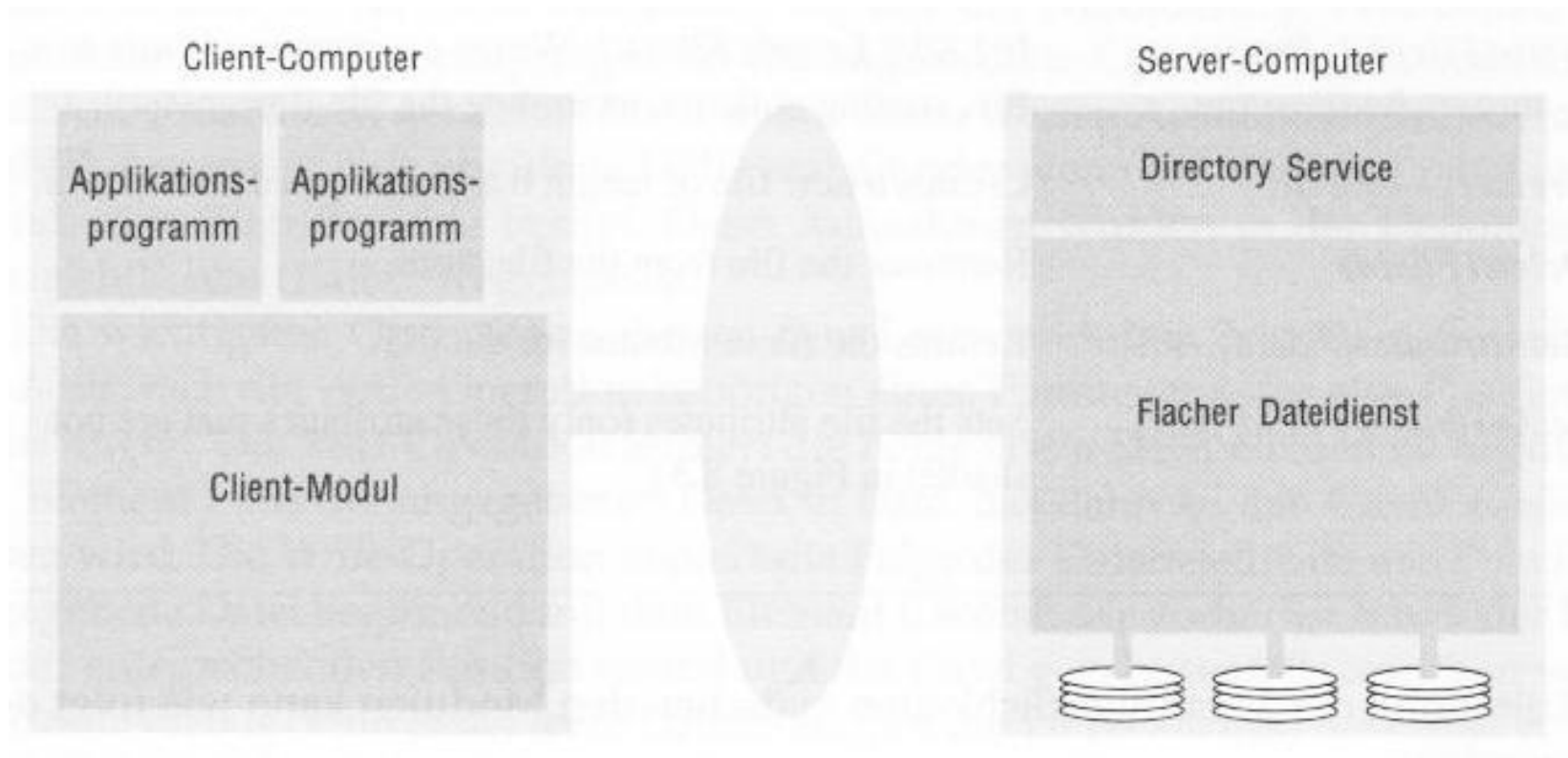
Struktur eines verteilten Dateisystems

Typisch 3 Schichten:

- Client-Schnittstelle
 - zur Ausführung der Dateioperationen durch Clients
 - Sicherung der Transparenz
 - Realisierung „Client-nah“
- Verzeichnisdienst
 - Abbildung der Dateinamen aus Sicht des Clients auf die Dateien im verteilten Filesystem (entspr. quasi Namensdienst)
 - verwaltet Attribute der Dateien und bietet Funktionen zu deren Veränderung an
- Dateidienst
 - stellt die eigentlichen Dateizugriffsoperationen (Öffnen, Schließen, Lesen, Schreiben) bereit
 - Realisierung „Server-nah“

8.7.1 Überblick (VII)

Struktur eines verteilten Dateisystems



8.7.1 Überblick (VIII)

Realisierung der Transparenzanforderungen aus Sicht der Clients wird durch Aufbau der Verzeichnisstrukturen und der Dateinamen bestimmt:

a) *Explizite Serverangabe*

- z.B.: `fileserver:/home/meier/dat.c`
- jeder Server hat eigene Namensregeln
- nicht transparent, bei Verschiebung der Datei muss ihr Name aktualisiert werden

b) *Entferntes Mounten*

- Einbindung ähnlich wie bei UNIX („mount“)
- nicht völlig transparent, da eine Datei auf verschiedenen Clients auch verschiedene Namen haben kann
- insbes. Verknüpfungen/Links sind problematisch

c) *Einheitlicher (uniformer) Namensraum*

- identische Namensgebung auf allen Clients, unabhängig von Platzierung
- z.B. durch übergeordnete Wurzel über alle lokalen Dateisysteme

8.7.1 Überblick (IX)

Konsistenzsicherung in verteilten Dateisystemen

ist insbesondere schwierig, wenn Replikation oder Caching benutzt wird

Verfahren dazu:

- „*UNIX-Semantik*“
 - Jede Schreiboperation ist sofort für alle Clients sichtbar, d.h. Schreibdaten müssen sofort dorthin transportiert werden, wo sich die Datei befindet
 - ggf. hohe Netzwerkbelastung
 - bei mehreren Schreiboperationen auf verschiedenen Clients ist für den Server die Einhaltung der Reihenfolge schwierig
 - wird selten benutzt

8.7.1 Überblick (X)

- „*Sitzungssemantik*“
 - beim Öffnen einer Datei erhält der Client eine Kopie
 - er arbeitet nur mit dieser Kopie
 - beim Schließen der Datei erfolgt Zurückschreiben durch Server
 - die Änderungen werden erst danach für andere Prozesse sichtbar (falls sie diese Datei danach öffnen), alle anderen sehen alten Inhalt
 - „Sitzung“ beginnt beim Datei-Öffnen und endet mit Datei-Schließen
 - Problem: falls mehrere Clients parallel dieselbe Datei ändern, muss der Server danach die verschiedenen Varianten zusammenführen:
 - entsprechend der Reihenfolge beim Schließen („last writer wins“, nur die letzte Version ist gültig)
 - nach (evtl. aufwändiger) Abstimmung über die gültige Version
 - mittels Versionierung (Server führt jede geschlossene Datei eigenständig mit speziellen Attributen oder Namen weiter). Die Versionen können später vom Nutzer/Admin zusammengeführt werden.

8.7.1 Überblick (XI)

- „*Read-Only-Semantik*“
 - Dateien können nur erstellt und gelesen werden, jedoch nicht verändert!
 - Veränderung ist nur durch Erstellen einer neuen Datei möglich
 - Problem ist ähnlich wie bei Sitzungssemantik auf eine Versionsverwaltung verlagert
- „*Transaktionssemantik*“
 - jede Operation mit einer Datei wird als Transaktion behandelt
 - hoher Aufwand, aber sicher: nach Transaktionsabschluss sind Änderungen für alle Clients sichtbar

8.7.1 Überblick (XII)

Caching (Pufferung) in verteilten Filesystemen

- sinnvoll, weil es beim Zugriff auf entfernte Dateien Verzögerungen gibt:
 - beim Plattenzugriff
 - bei der Netzwerkübertragung
- Pufferung von ganzen Dateien oder nur von Dateiabschnitten möglich
- *Caching beim Server:*
 - Einsparung beim Plattenzugriff
- *Caching beim Client:*
 - Einsparung beim Plattenzugriff und beim Netzwerkverkehr,
 - aber evtl. Konsistenzprobleme, daher müssen Änderungen
 1. an den Server übermittelt werden (write-through, delayed write)
 2. und allen Clients mitgeteilt werden
 - evtl. gesonderter Cache-Manager, der einen zentralen Cache verwaltet, Dateizugriffe laufen dann via IPC mit dem Manager ab
- *Caching durch die Anwendung selbst* (ohne BS-Beteiligung)
 - günstig für Dateien, die nur von 1 Prozess benutzt werden

8.7.1 Überblick (XIII)

Verwaltung der Zustände aller geöffneten Dateien (z.B. aktuelle Schreib-/Leseposition, Dateisperren, ID des benutzenden Clients usw.):

- *zustandsbehafteter (stateful) Server*
 - Verwaltung der Zustandsinformation durch den Server
 - daher geringere Netzbelastung
 - aber fehleranfälliger insbes. bzgl. Serverausfall
- *zustandsloser (stateless) Server*
 - Server speichert keine Infos über geöffnete Dateien
 - Info muss bei jedem Dateizugriff durch den Client an den Server übermittelt werden, daher höhere Netzbelastung
 - Server spart Speicherplatz
 - ist fehlertolerant, weil trotz Serverausfall die Informationen bei den Clients weiter verfügbar sind

8.7.1 Überblick (XIV)

Dateireplikation

durch Speicherung mehrere Kopien einer Datei auf versch. Servern

- Leistungssteigerung möglich durch geschickte Wahl des am besten geeigneten (z.B. wenig belasteten) Servers
- höhere Verfügbarkeit/Zuverlässigkeit bzgl. Ausfall eines Servers; inkonsistente Dateien können mittels Replikaten wieder hergestellt werden
- Skalierbarkeit: Bei Überlastung eines Servers kann er durch weitere Server unterstützt werden

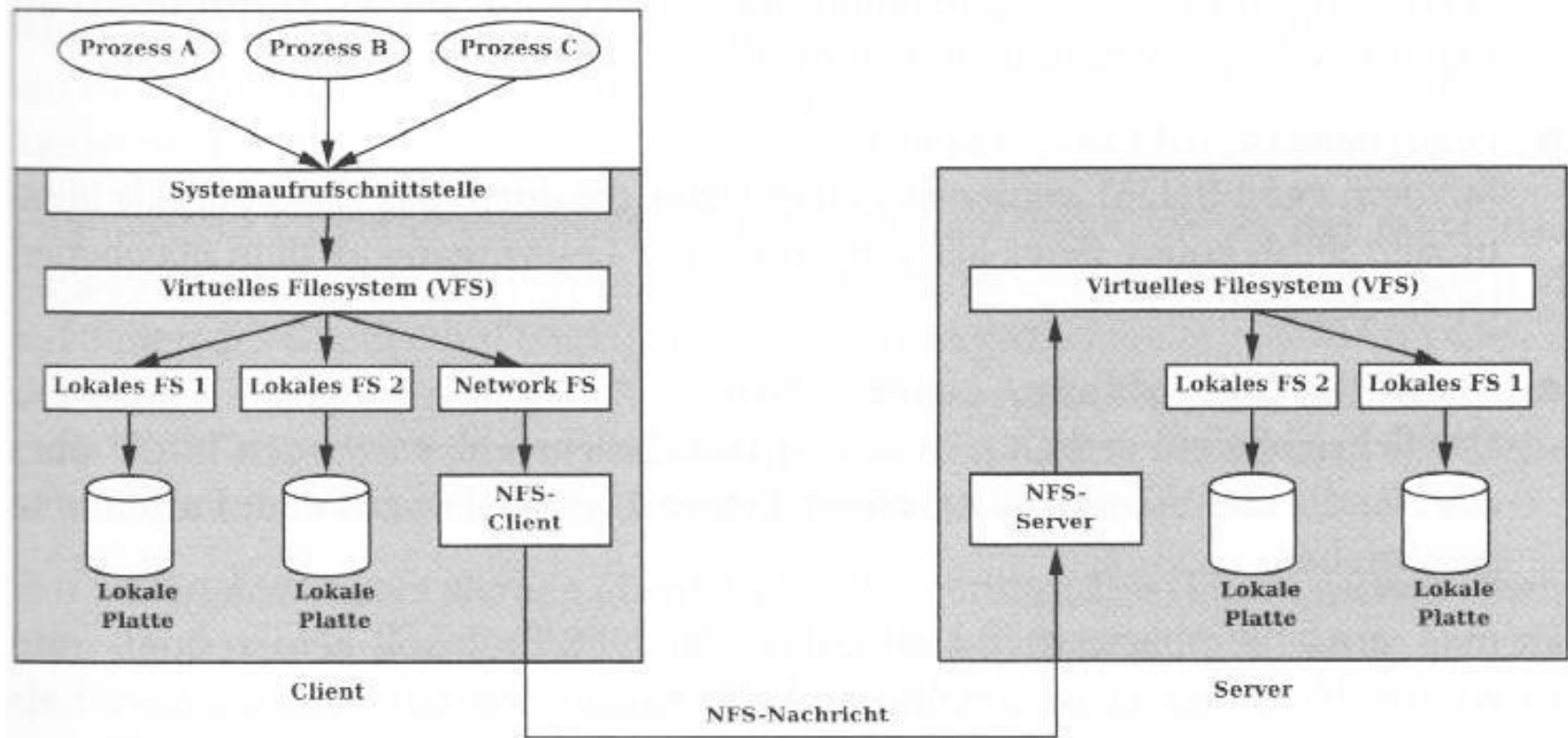
Verfahren:

- beim Lesezugriff wird irgendeines der Replikate geliefert und beim Schreiben werden alle Replikate aktualisiert (mitunter schwierig)
- Prozess muss best. Mindestzahl („Quorum“) von (insges. N) Replikaten lesen/schreiben: beim Lesen: $R + W > N$ und beim Schreiben: $W > N/2$

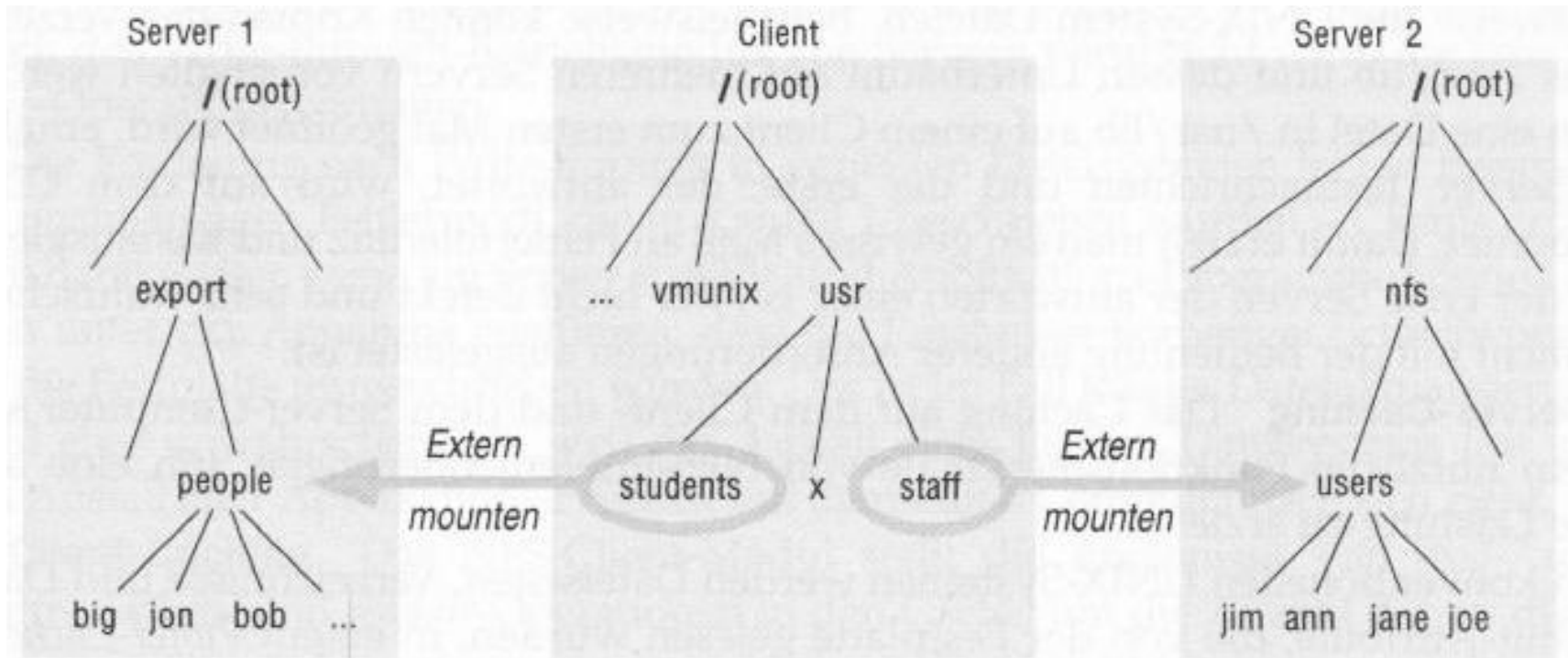
8.7.2 Beispiel: [Sun Network Filesystem \(NFS\)](#) (I)

- Kommunikation Client-Server via BS-unabhängigen NFS-Protokoll (realisiert als Menge von entfernten Prozeduraufrufen [Sun RPC])
- NFS-Server-Modul ist im BS-Kern des NFS-Server-Knotens
- NFS-Client wandelt Anfragen zu entfernten Dateien in NFS-Protokoll-Operationen an den entspr. NFS-Server um
- Zugriffstransparenz für Anwendungen durch Integration eines VFS in den BS-Kern (z.B. UNIX) des Client-Rechners, Anwendung muss nicht zwischen lokalen und entfernten Dateizugriffen unterscheiden, dies tut das VFS
- Einfügen eines entfernten Dateisystems z.B. entferntes Mounten durch spez. Mount-Dienstprozess, Client erhält dann Datei-Handles
- VFS verwaltet eine eigene VFS-Dateistruktur und eigene v-Nodes pro geöffneter Datei als Verweis auf diese (lokal oder entfernt)
- Caching auf Server und Client
- NFS-Protokoll ist zustandslos
- relativ effizient und weit verbreitet

8.7.2 Beispiel: Sun Network Filesystem (NFS) (II)



8.7.2 Beispiel: Sun Network Filesystem (NFS) (III)



Hinweis: Das in `/usr/students` auf dem Client installierte Dateisystem ist eigentlich der Unterbaum, der auf Server 1 in `/export/people` installiert ist; das in `/usr/staff` auf dem Client installierte Dateisystem ist eigentlich der Unterbaum, der sich auf Server 2 in `/nfs/users` befindet.

8.7.3 Beispiel: [Coda](#) (I)

- Weiterentwicklung des Andrew Filesystems AFS für viele Clients
- hohe Forderungen nach Skalierbarkeit, Fehlertoleranz und Replikation sowie Einbindung mobiler Geräte
- Serverkomponente *Vice*, Clientkomponente *Venus*
- Dateizugriff via Upload/Download-Modell: nach Anforderung wird die Datei vom Server zum Client kopiert
- Caching geöffneter Dateien auf dem Client,
→ „Sitzungssemantik“
- Namensraum für Dateien wird in lokalen und gemeinsamen Teil (unter `/coda`) unterschieden
- Einbindung gemeinsamer Dateien in den lokalen Teil via symbol. Link

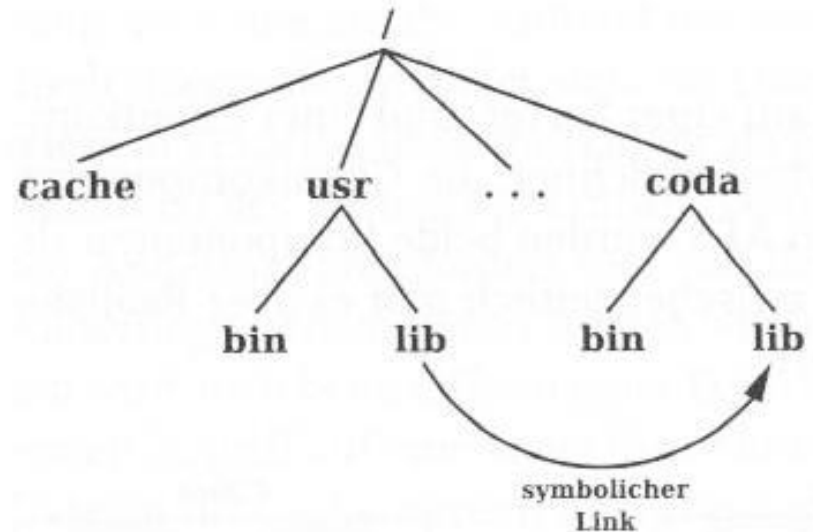


Bild-Quelle: Peschel: Nebenläufige und verteilte Systeme. Bonn: mitp, 2006

8.7.3

Beispiel: Coda (II)

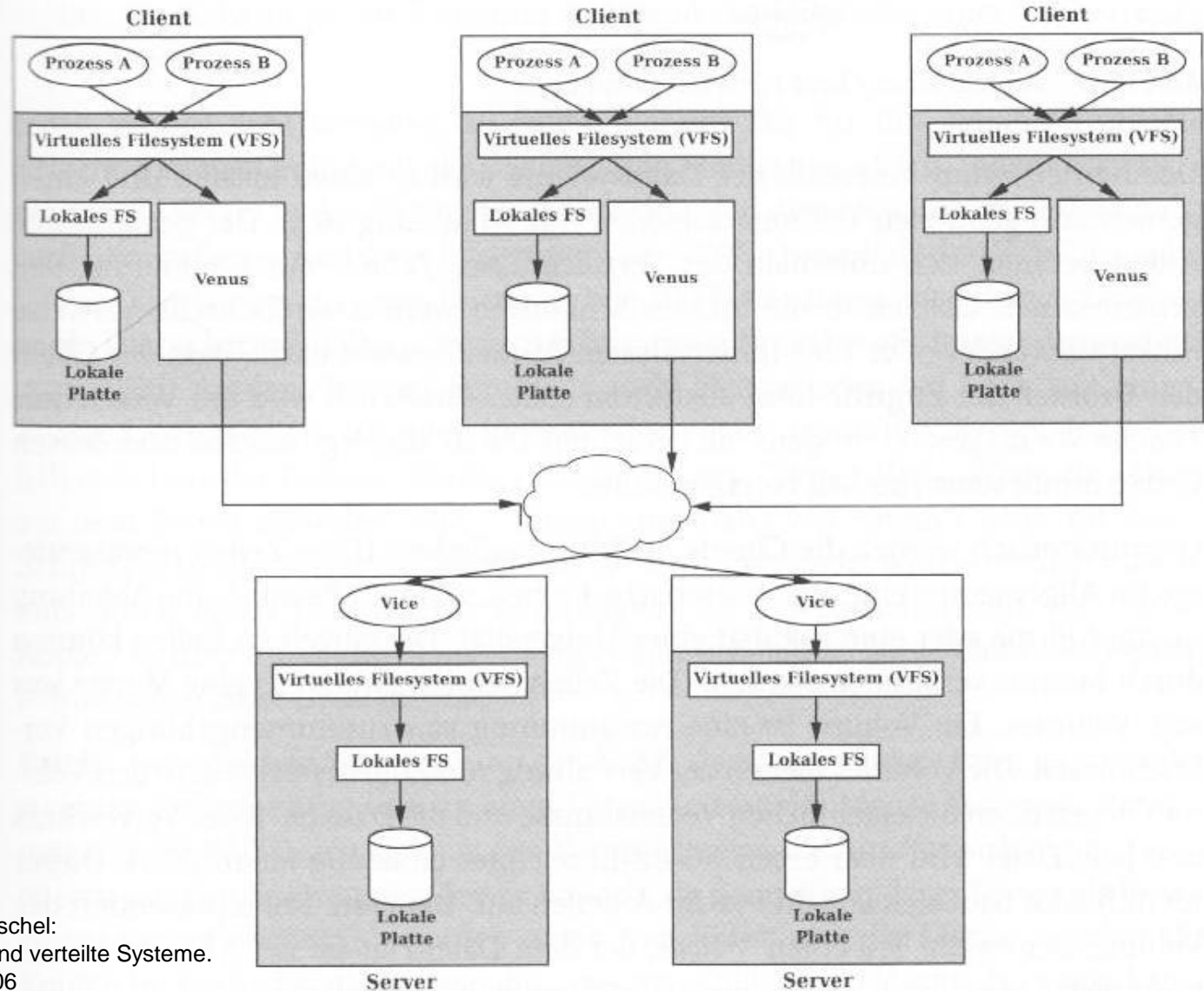


Bild-Quelle: Peschel:
Nebenläufige und verteilte Systeme.
Bonn: mitp, 2006

8.7.3 Beispiel: Coda (III)

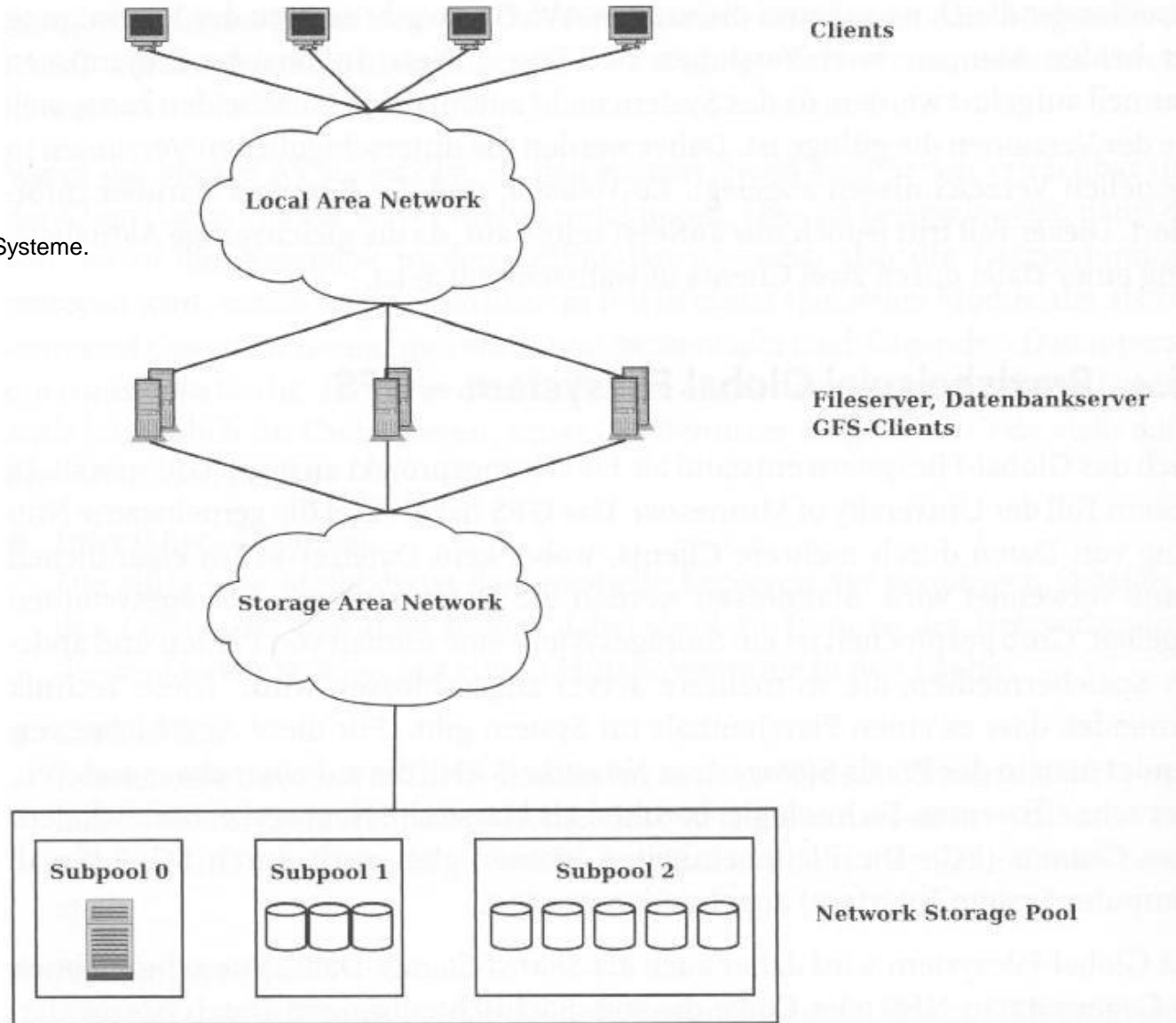
- Clients werden in sog. Zellen organisiert (z.B. Abteilungen einer Fa.)
- Zellen können über Mounts verbunden sein
- jede Zelle enthält Volumes (= Menge von zusammengehörigen Verzeichnissen zur Rechteverwaltung)
- Replikation auf Basis kompletter Volumes
- es ist nicht zwingend, dass immer eine Verbindung zu allen Servern besteht, Coda unterscheidet deshalb zwischen
 - Volume Storage Group VSG (alle Server, die ein Replikat eines Volumes haben)
 - Available VSG (AVSG, alle zur Zeit erreichbaren VSGs)
- Clients verwalten ihre pers. AVSG und aktualisieren sie periodisch durch Nachfrage bei allen Servern der VSG
- Server ist zustandsbehaftet: nach Veränderung einer Datei sendet er eine Info an Clients (Callback), die ihre gecachte Kopie ungültig setzen
- Konsistenzsicherung mittels zusätzl. Versionsvektor für jede Datei

8.7.4 Beispiel: Global Filesystem GFS (I)

- speziell für Cluster entworfen
- Besonderheit: es gibt keinen Datei-Server im engeren Sinne, sondern Nutzung eines Stagesystems (= Sammlung ext. Speicher), z.B. Storage Attached Network, SAN
- Stagesystem ist an mehrere Server angeschlossen, damit kein Flaschenhals
- SAN verwendet z.B. Fibre Channel-Technologie oder SCSI
- GFS ist ein „shared storage“-Filesystem, d.h. anders als bei NFS oder Coda, wird der Dateizugriff hier via Datenaustausch über gemeinsamen Speicher realisiert, unterstützt durch HW für Locking
- Stagesysteme werden in Network Storage Pool (NSP) zus.-gefasst
- Unterteilung in Subpool anhand von Eigenschaften, z.B. Kapazität
- Konsistenzsicherung via Mutexe und Device-Locks mit Timeout

8.7.4Beispiel:GFS (II)

Bild-Quelle: Peschel:
Nebenläufige und verteilte Systeme.
Bonn: mitp, 2006



8.7.5 Beispiel: BeeGFS

- ehemals „FraunhoferFS (FhGFS)“
- speziell für Cluster/HPC entworfen
- seit 2016 open source, Linux-basiert
- Entw.-ziele: leichte Handhabung, hohe Flexibilität, hohe Skalierbarkeit
- Komponenten:
 - Client Services,
 - Metadata Server,
 - Storage Server,
 - Management Service
- Nutzerdaten: in Chunks zerlegt,
- Chunks: verteilt auf Servern gespeichert
- Metaserver: Verwaltung der Daten und Zuordnung einer Datei zu Chunks
- Verbindung der einzelnen Server via Ethernet oder Infiniband, OmniPath, ...



Bild-Quelle: Wikipedia

8.7.6 Beispiel: Lustre

- für Cluster Computing entwickelt (Linux+Cluster)
- verfügbar unter GNU GPL (v2)
- gut skalierbar (viele tausend Clients, Datenmenge im Petabyte-Bereich, Übertragung mit einigen 100 Gbyte/s)
- lustre.org
- es gibt eine Intel Enterprise Edition für Lustre

8.7.7 weitere Beispiele verteilter Filesysteme

- Cryptographic File System (CFS)
- x File system (xFS), Berkeley
- Server Message Block (SMB)
- Common Internet Filesystem (CIFS), Microsoft
- General Parallel File System (GPFS), Cluster-Dateisystem
- Distributed File System der DCE (DCE/DFS), Open Group
- Distributed File System (DFS), Microsoft
- Oracle Cluster File System, Version 2) (OCFS2), Oracle
- ...