5.2 Threads, Prozessorvergabe

❏ Vom Betriebsystem zu unterstützende Ziele

- Hoher Wirkungsgrad (performance)
- Ausbaufähigkeit (scalability)
- Hohe Verfügbarkeit (availability)und Zuverlässigkeit (reliability), sanfter Leistungsabfall bei Teilausfällen (graceful degradation)

❏ Spezielle Probleme, die aus dem Verlangen nach hohem Wirkungsgrad und Ausbaufähigkeit resultieren:

- Zugriffsschutz in sehr großen Adreßräumen (64 Bit für Adressen)
- Vermeidung von Verklemmungen (deadlock prevention)
- Ausnahmebehandlung bei sehr vielen Prozessoren
- effizient bearbeitbare Darstellung von asynchron arbeitenden oder bearbeitbaren Einheiten
- Bereitstellung angepaßter Interaktionsmechanismen
- Betriebsmittelverwaltung (Prozessorzuordnung, Speicherverwaltung)
- Parallelisierung des Betriebssystems selbst

❏ Detaillierte Betrachtung der Prozessorvergabe

- Unix-Prozeß bestehend aus Adreßraum und Aktivitätsträger (heavyweight processes)
  - Parallelisierung einer Aufgabe nur durch Benutzung mehrerer Prozesse; disjunkte Adreßräume unnatürlich bei Datenpartitionierung
  - Erzeugung, Zuordnung und Tilgung von Prozessen aufwendig; dementsprechend zeitraubend ist der Wechsel zwischen seriellen und parallelen Phasen
  - Prozeßwechsel sind aufwendig wegen des damit verbundenen Kontextwechsels und den daraus resultierenden Cache- und TLB-Invalidierungen

- Entkopplung von Adreßraum und Aktivitätsträger (middleweight processes, kernel-level processes)
  - Prozessorvergabe muß nicht mit Kontextwechsel verbunden sein; bei ‘single task’- Betrieb keine Kontextwechsel
  - Bieten eine allgemeine Benutzerschnittstelle

    Beispiel: Mach, Windows NT
  - Betriebssystem macht Zuordnung und kann dabei seine Kenntnis des Systemzustandes nutzen, um gute Betriebsmittelauslastung zu erzielen
  - Für feingranulare Parallelität immer noch zu schwerfällig (z. B. ist bei manchen Rechensystemen Sicherung und Wiedereinsetzung der Register für Gleitkommabearbeitung aufwendig)
  - Verwaltungsoperationen für Fäden (z. B. im Rahmen von Interaktionen) erfordern Systemaufruf
  - Es ist unwahrscheinlich, daß eine (auch parametrisierte) Zuordnungskonzeption für alle Anwendungen effizient ist.

- Implementierung von leichtgewichtigen Fäden (lightweight processes) durch Multiplexen schwer- oder mittelgewichtiger Fäden
  - Realisierung und Verwaltung durch Bibliotheken

    Beispiel: Sun LWP

    Thread Creation

    lwp_self(tid)

    lwp_getstate(tid, statvec)

    lwp_setregs(tid, machstate)

    lwp_getregs(tid, machstate)

    lwp_ping(tid)

    lwp_create(tid, pc, prio, flags, stack,
    nargs, arg1, ..., argn)

    lwp_destroy(tid)

    lwp_enumerate(vec, maxsize)

    pod_setexit(status)

    pod_getexit()

    pod_exit(status)

    SAMETHREAD(t1, t2)

Thread Scheduling

pod_setmaxpri(maxprio)

pod_getmaxpri()

pod_getmaxsize()

lwp_resched(prio)

lwp_setpri(tid, prio)

lwp_sleep(timeout)

lwp_suspend(tid)

lwp_resume(tid)

lwp_yield(tid)

lwp_join(tid)

Error Handling

lwp_geterr()

lwp_perror(s)

lwp_errstr()

Messages

msg_send(tid, argbuf, argsize,
resbuf, ressize)

msg_recv(tid, argbuf, argsize,
resbuf, ressize, timeout)

MSG_RECVALL(tid, argbuf, argsize,
resbuf, ressize, timeout)

msg_reply(tid)

msg_enumsend(vec, maxsize)

msg_enumrecv(vec, maxsize)

Event Mapping (Agents)

agt_create(agt, event, memory)

agt_enumerate(vec, maxsize)

agt_trap(event)

Thread Synchronization: Monitors

mon_create(mid)

mon_destroy(mid)

mon_enter(mid)

mon_exit(mid)

mon_enumerate(vec, maxsize)

mon_waiters (mid, owner, vec, maxsize)

mon_cond_enter(mid)

mon_break(mid)

MONITOR(mid)

SAMEMON(m1, m2)

Thread Synchronization: Condition Variables

cv_create(cv, mid)
cv_destroy(cv)
cv_wait(cv)
cv_notify(cv)
cv_send(cv, tid)
cv_broadcast(cv)
cv_enumerate(vec, maxsize)
cv_waiters(cv, vec, maxsize)
SAMECV(c1, c2)

Exception Handling

exc_handle(pattern, func, arg)
exc_unhandle()
(*exc_bound(pattern, arg))()
exc_notify(pattern)
exc_raise(pattern)
exc_on_exit(func, arg)
exc_uniqpatt()

Special Context Handling

lwp_ctxinit(tid, cookie)
lwp_ctxremove(tid, cookie)
lwp_ctxset(save, restore, ctxsize, optimise)
lwp_ctxmemget(mem, tid, ctx)
lwp_ctxmemset(mem, tid, ctx)
lwp_fpset(tid)
lwp_libcset(tid)

Stack Management

CHECK(location, result)
lwp_setstkcache(minsize, numstks)
lwp_newstk()
lwp_datastk(data, size, addr)
lwp_stkcswset(tid, limit)
lwp_checkstkset(tid, limit)
STKTOP(s)

Benutzerfäden mit einem Round-Robin-Scheduler

```
#include <lwp/lwp.h>
#include <lwp/lwpmachdep.h>
#include <lwp/stackdep.h>
#include <lwp/interfaces.h>

#include <iostream.h>

#define MAXPRIO      10

int main(int argc, char **argv)
{ int i;
  void task(int);
  void scheduler();
  thread_t tid;

  cout << "main here\n" << flush;
  (void)pod_setmaxpri(MAXPRIO);
  lwp_setstkcache(1000, 5);
  (void) lwp_create((thread_t *)0, scheduler,
                    MAXPRIO, 0, lwp_newstk(),
                    0);
  for (i = 0; i < 3; i++)
  lwp_create((thread_t *)0, task,
             MINPRIO, 0, lwp_newstk(), 1, i);
  return 0;
}
```



```
void scheduler()
{ struct timeval quantum;
  quantum.tv_sec = 0;
  quantum.tv_usec = 10000;
  for (;;) {
    lwp_sleep(&quantum);
    lwp_resched(MINPRIO);
  }
}

void task(int arg)
{ for(;;)
    cout << arg << flush;
}
```

001111111111111111111111111111111111111112222222222222222222222222222222
2222222222222222222222000000000000000000000000000000000011111111111111111
111111111111111111111111111111111111112222222222222222222222222222222222
222222222222222222000000000000000000000000000000000000000000000000000011
111111111111111111111111111111111111111111111222222222222222222222222222
222222222222222222222222200000000000000000000000000000000000000000000000
000001111111111111111111111111111111111111111111111111112222222222222222
222222222222222222222222222222222220011122200000000000000000000000000011
111111111111111111111111111111111111111111111111111122222222222222222222
222222222222222222222222222222220000000000000000000000000000000000000000
000000000000011111111111111111111111111111111111111111111111111111122001
122000112222222222222222222222222222222222222000000000000000000000000000
000000000000000000000000011111111111111111111111111111111111111111111111
122222222222222222222222222222222222222222222222222222000000000000000000
000000000000000000000000000000000001111111111111111111111111111111111111
111111111111111222222222222222222222222200011220011220011111111111111111
111111111111111111111111112222222222222222222222222222222222222222222222
222222200000000000000000000000000000000000000000000000000000111111111111
111111111111111111111111111111111111111112222222222222222222222222200000
000000000000000000000000000000000000000111111111111111111111111111111111
111111111111111111112222222222222222222222222222222222222222222222222222
000111222222222222222222222222222222222222222220000000000000000001111111
111111111111122222222222222001111111111111111111111111112222222222222222
222222222222222222222222222222222220000000000000000000000000000000000000
000000000000111111111111111111111111112200112200000000001111111111111111
111111111111112222222222222222222222222222220000000000000000000000000000
000000000000000000000011111111111111111111111111111111111122222222222222
222222222222200000000000000000000000000000001111111111111111111111111122
222222222222222222222222222222222200112200112200000000000000001111122222
222222222222222222222220000000000000000000000000000000000000000000000000
000111111111111111111111111111122222222222222222222222222220000000000000
000000000000000000000000000000111122222222222222222222222222222222222222
222222222222200111112222222222222222222222222222222000000000000000000000
000000000000000000001111111111111111111111111111111111112222222222222222
222222222222222222222222200000000000000000000000000000000000000000000001
111111111111111111111111111111111111111111112222222222220000000000000000
000000000000000000000000111111111111111111111111111111111111111112222222
222222222222222222111111111111111111111111111122222222222222222222222222
200000000000000000000000000000000000000000000000011220000000000111111111
111111111111222222222222222222000000000000000000000000000111111111111111
111111111111112222222222222222222222000000000000000000000000000011111111
111111111111111111111111111111111111222222222222222222222000000000000000
000000111111111111111111111111111111111111112222222222222222222222222222
200000000000000000000000000000000011222001111111111111111111111111111111
111111111111222222222222220000000000000000000000000000000001111111111111
111111111111111111111111111111111122222222222222222222222222222222222222
222222222222222000000000000000000000000000000000000000000000000011111111
111111111111111111111111111111111111111111112222222222222222222222222222
200000000000000000000000000000000000000000000000000001111111111111111111
111111111111111111111111111111111220011222222222222222222222222222200000
000000000000000000000000000000000000000000000000111111111111111111111111

09.07.97 Threads, Prozessorvergabe

- Erfordert Prozessorvergabe auf wenigstens zwei Ebenen:
  1. im Betriebssystem
  2. im schwer- oder mittelgewichtigen Faden

  Nachteile:

  Die Verwaltung leichtgewichtiger Prozesse hat keine Kenntnisse über betriebssysteminterne Ereignisse

  Schedulingentscheidungen des Betriebssystems wirken sich in gleicher Weise auf alle leichtgewichtigen Fäden aus, die durch Multiplexen aus einem schwer- oder mittelgewichtigen hervorgehen (z. B. Blockierung infolge E/A eines leichtgewichtigen Fadens).

  Kern hat bei seinen Entscheidungen keinerlei Kenntnis von den leichtgewichtigen Fäden.

• Einführung mehrstufiger (vor allem zweistufiger) Scheduler
  - Benachrichtigung des Benutzerschedulers über bestimmte Kernereignisse durch ‘upcalls’
  - Benachrichtigung des BS-Kerns über alle Ereignisse der Benutzerebene, die für das BS-Scheduling bedeutsam sind.

Betriebsprogrammierung II
© 1997, Universität Erlangen-Nürnberg, IMMD IV, F. Hofmann

Reproduktion jeder Art oder Verwendung dieser Unterlage zu Lehrzwecken außerhalb der Universität Erlangen-Nürnberg ist ohne Genehmigung des Autors nicht zulässig

Scheduling

❑ Scheduler-Struktur

- Struktur der Warteschlangen
- Parallele Bearbeitung von Warteschlangen

❑ Strategien

- Statische oder dynamische Zuordnung von Prozessor und Faden bei Mehrbenutzerbetrieb;
  Ergebnisse der Untersuchungen von Zahorjan und McCann
  (J. Zahorjan and C. McCann. Processor scheduling in shared memory multiprocessors. In *Proceedings of the 1990 ACM SIGMETRICS Conference on Measurement and Modeling of Computer systems*, pages 214-225, May 1990.):
  - Unabhängig von der Einzel- und Gesamtlast ist dynamisches Scheduling am besten, wenn der Zusatzaufwand für Kontextumschaltungen gering ist.
  - Die Vorteile dynamischer Zuordnung werden um so größer, je häufiger der ‘Parallelitätsgrad’ sich ändert.
  - Die Vorteile dynamischer Zuordnung werden mit zunehmender Systemlast größer.
  - Bezüglich der mittleren Antwortzeit ist dynamisches Zuordnen fast immer besser.

- Gemeinsame Bereitschlange, aus der sich freie Prozessoren selbst bedienen:
  - Vorteil:
    Automatische Lastverteilung
  - Nachteile:
    Keine Rücksichtnahme auf häufig interagierende Fäden
    Keine Rücksichtnahme auf Reihenfolgeforderungen (z. B. beschrieben durch Präzedenzgraphen) und damit keine Gesamtoptimierung bezüglich der Aufträge (jobs)
- interaktionsorientiertes Scheduling (meist ‘coscheduling’ oder ‘gang scheduling’ genannt)

  Ousterhout entwickelte drei Methoden:

  Matrix-Scheduling (matrix)

  fortlaufendes Scheduling (continuous)

  ungeteiltes Scheduling (undivided)

- Matrix-Scheduling
  alle Fäden eines Auftrags in einer Zeile
  Fäden einer Zeile werden gleichzeitig zugeordnet

Prozessor

Zeitscheibe	1	2	3	4	5	6	7	8
1								
2								
3								
4								
5								
6								

Löcher führen zu Effizienzverlusten

- fortlaufend

  Matrix linearisiert durch Aneinanderreihung der Zeilen

  zu jedem Zeitpunkt Fäden eines Fensters der Länge p zugeordnet

  Neuankömmling wird in aktives Fenster aufgenommen, wenn dies möglich ist,

  andernfalls wird Fenster solange nach rechts geschoben, bis zum erstenmal das linke Feld leer ist. Dies wird wiederholt, bis Auftrag eingefüllt werden kann.

  Nach jedem Zeitquant wird Zuordnungsfenster eitergeschoben, bis der linke Eintrag zu einem Auftrag gehört, der im vorherigen Zeitschritt nicht zugeordnet war.

  Große, unzusammenhängende Aufträge können gegenüber kleinen zusammenhängenden benachteiligt sein.

- ungeteilt

  analog fortlaufend, aber nur zusammenhängende Einordnung

Es handelt sich um zentralisierte Algorithmen

Anderer Ansatz: baumartig angeordnete Controller verwalten Prozessoren ähnlich einem Buddy-Verfahren bei Speicherplatzvergabe.

- Präzedenzgraphorientierte Strategien
  - Die meisten Untersuchungen betreffen einstufige Präzedenzgraphen, d. h. ein Hauptfaden erzeugt eine Reihe von Unterfäden und beendet sich dann. Meist wird noch angenommen, daß die Unterfäden nicht interagieren.
  - Bekannte Strategie für diesen Fall ist RR

    Erste Version führt Schlange von bereiten Fäden und bearbeitet sie nach RR, d. h. jeder freigewordene Prozessor bearbeitet den nächsten Auftrag für Q Zeiteinheiten und reiht ihn dann am Ende der Bereitschlange wieder ein.

    Zweite Version führt in der WS Aufträge. Falls ein Auftrag mehr Fäden besitzt als es Prozessoren gibt, wird innerhalb des auftrags wiederum nach RR zugeteilt.
  - Probleme:

    Verdrängung während eines Spinlock oder in kritischem Abschnitt

    Häufige Kontextumschaltungen

- Statische oder dynamische Partitionierung
  - Ziel ist die Minimierung der Kontextumschaltungen
  - Hypothese: Aufträge erreichen die beste Effizienz, wenn die Zahl der Fäden gleich der Zahl der verwendeten Prozessoren ist.
  - In NUMA-Architekturen existiert häufig eine von der Anwendung vorteilhaft nutzbare Kommunikations-struktur; wegen ihrer guten Skalierbarkeit spielen Gitterarchitekturen eine besondere Rolle.
  - Als Strategien bieten sich Weiterentwicklungen der Matrixmethode von Ousterhout an.

Beispiel einer solchen Zuteilungsstrategie für ein torusartiges System, dessen Knoten Multiprozessoren sind (MEMSY):

Definition

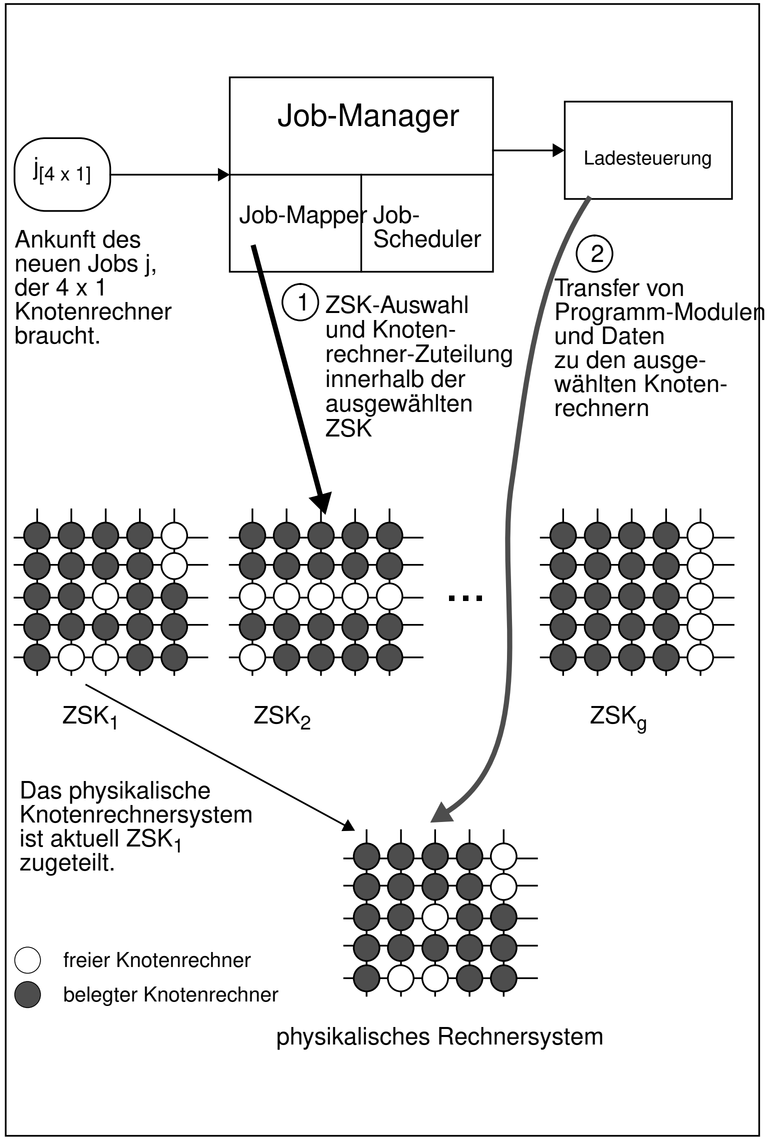
Eine *Zeitscheibenklasse (ZSK)* eines Parallelrechnersystems PS ist ein virtuelles Knotenrechnersystem, das folgende Eigenschaften hat:

(a) Die Anzahl der virtuellen Knotenrechner in ZSK ist gleich der Anzahl der physikalischen Knotenrechner in PS.

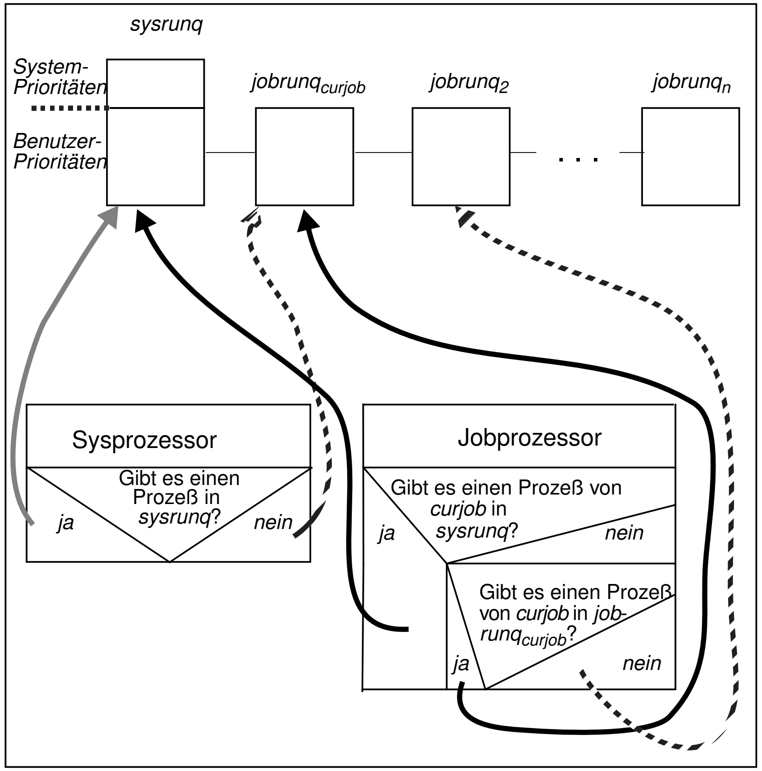
(b) ZSK hat die gleiche Verbindungstopologie wie PS.

(c) Es gibt eine bijektive Abbildung vps_{ZSK} der Knotenrechner von ZSK auf die Knotenrechner von PS. Gilt (für ein $V \in ZSK$ und ein $R \in PS$) $R = vps_{ZSK}(V)$, so sind V und R *topologisch äquivalent*.
Dabei wird unter dem Begriff der *topologischen Äquivalenz* hier folgendes verstanden: wenn PS und ZSK jeweils das gleiche Koordinatensystem, das die Knotenrechner bijektiv auf Koordinaten abbildet, zugrundegelegt würde, so wären ein Knotenrechner $K_v \in ZSK$ und ein Knotenrechner $K_r \in PS$ topologisch äquivalent, wenn ihnen die gleichen Koordinaten zugeordnet wären.

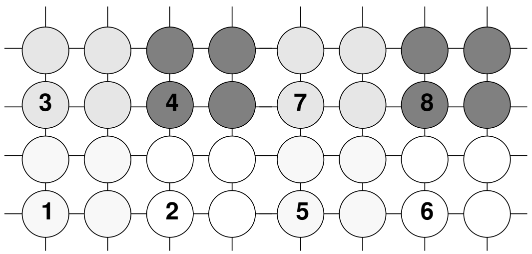
Einordnung neuer Aufträge



Knotenscheduling

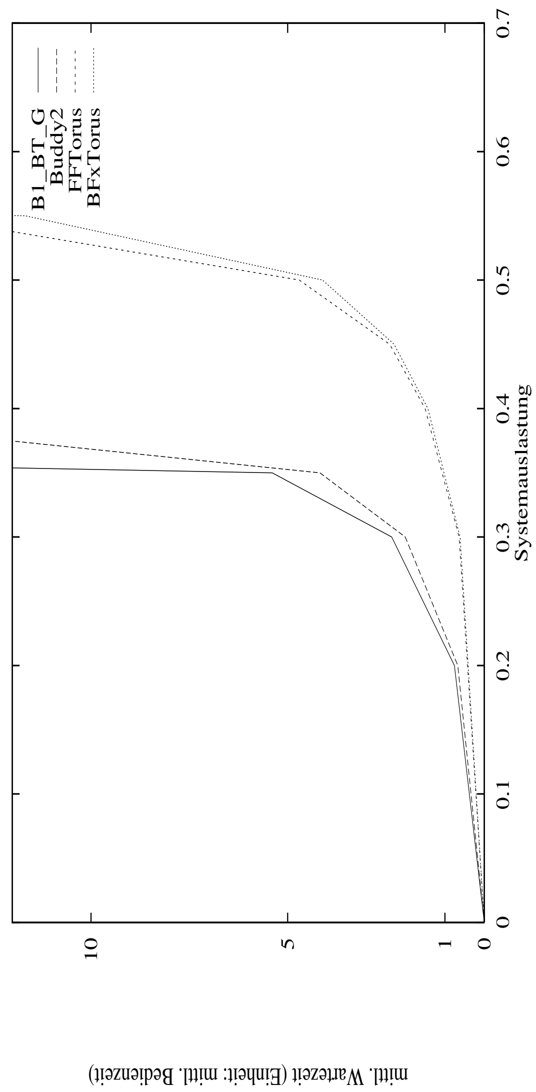


Anpassung der eindimensionalen Buddy-Strategie



Ein Torus(8 x 4)-System, das aus Torus(2 x 2)-System-Einheiten basierend auf einer zweidimensionalen Buddy-Strategie aufgebaut wird.
Die Zahlen in den Knoten notieren die Reihenfolge, in der die entsprechenden (2 x 2)-Systeme hinzugefügt werden.

Simulationsergebnisse für verschiedene Plazierungsstrategien, exponentielle Verteilung der Zwischenankunftszeiten und Bedienzeiten, Höhen und Längen der angeforderten Teilbereiche unabhängig gleichverteilt



- Hand-off Scheduling

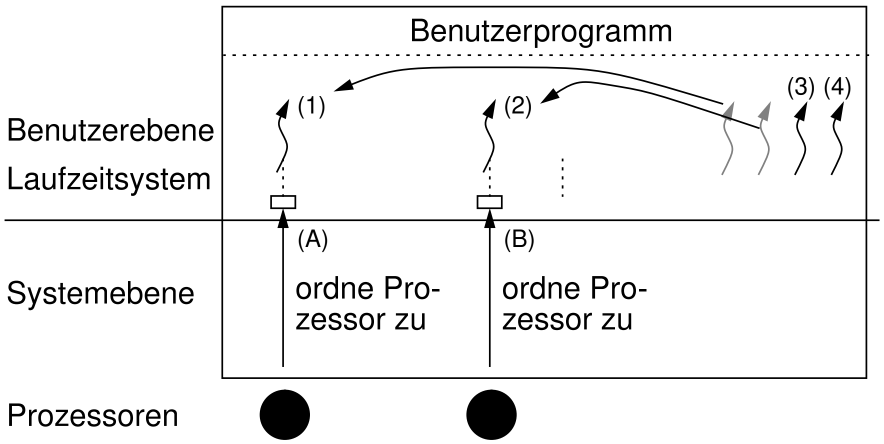
  Benutzer gibt Hinweise

  - Hinweise auf Unzweckmäßigkeit des Zuordnens
  - Hinweise auf zuzuordnende Prozesse

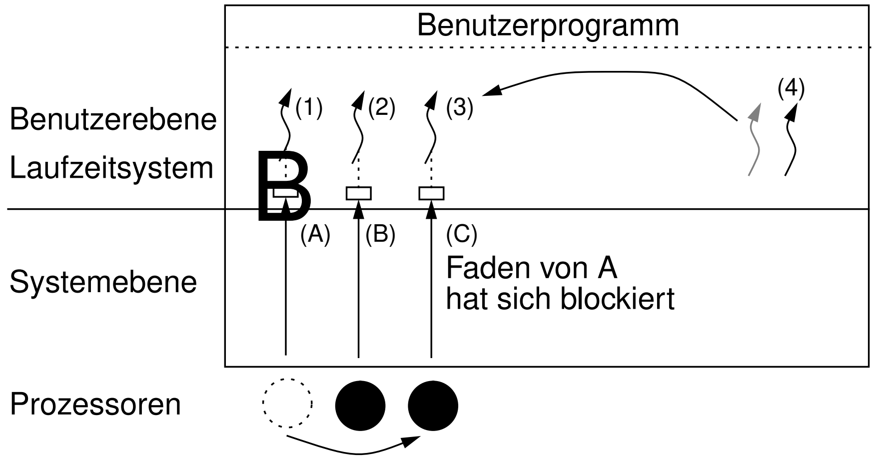
  Letzteres ist insbesondere im Zusammenhang mit Kooperation von Interesse.

❑ Mischung aus Kern- und Benutzerfäden

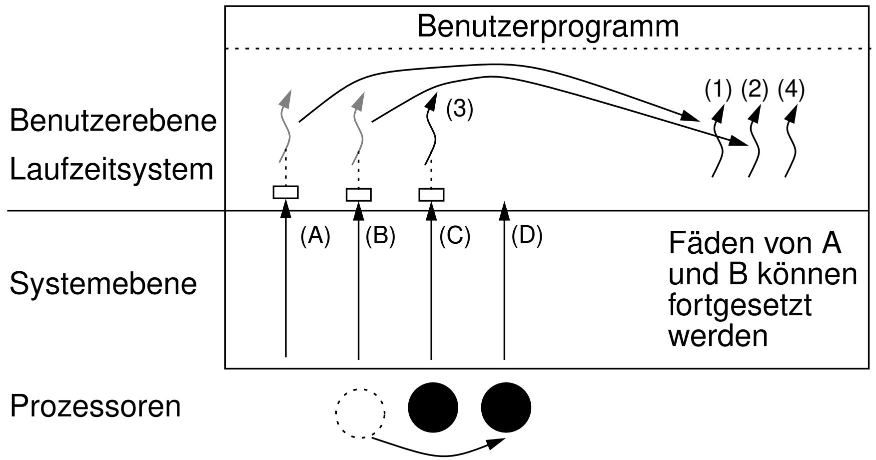
- Kern ordnet der Anwendung zwei Prozessoren zu



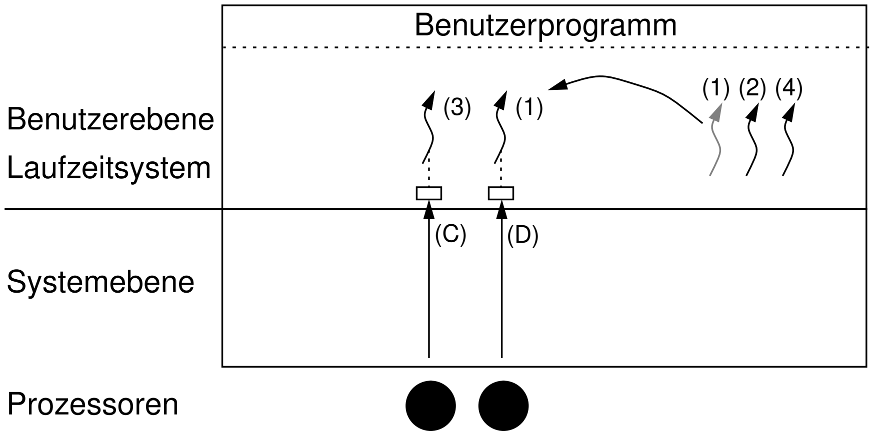
- Ein Benutzerfaden blockiert sich im Kern wegen E/A



- E/A abgeschlossen



• ‘upcall’ wegen freien Prozessors



❑ Forschung:

Berücksichtigung von Echtzeitanforderungen wie sie z. B. in Multimedia-Anwendungen auftreten

5.3 Arbeitsspeicherverwaltung

Arbeitsspeicherverwaltung in Mach

❑ flexible Verwaltung basierend auf Seitenadressierung

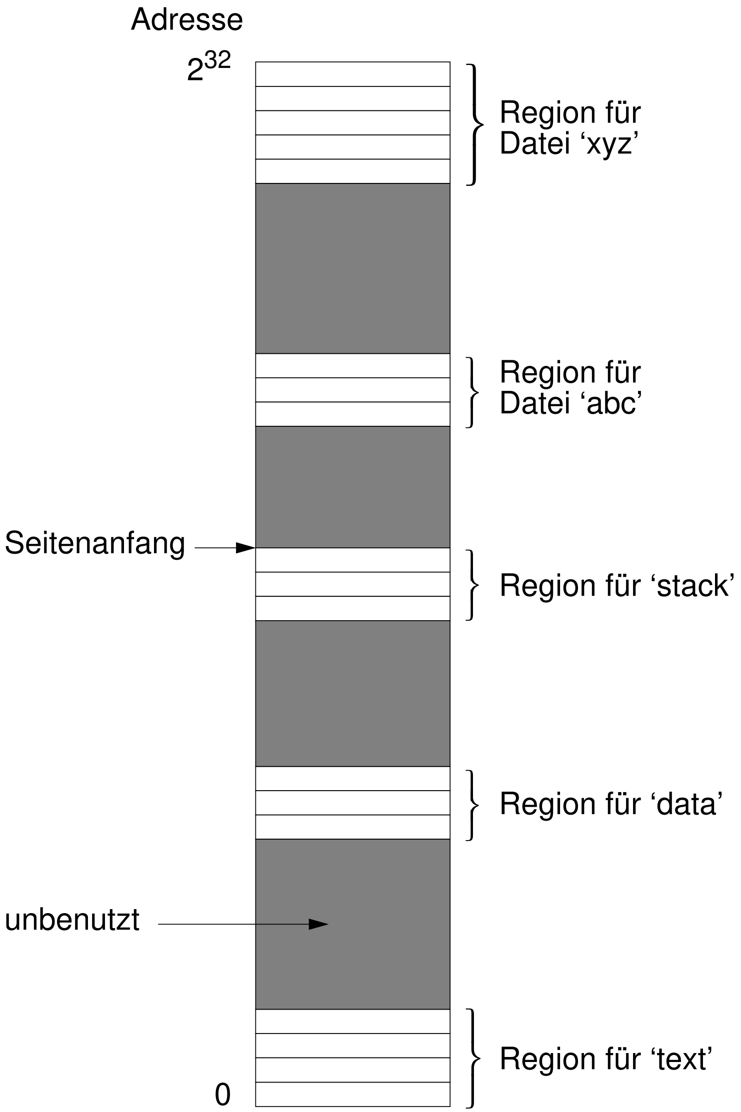
❑ Trennung in maschinenabhängigen und maschinenunabhängigen Teil auf eine sehr innovative Weise

❑ Drei Module mit klar definierten Schnittstellen:

- Das erste Modul läuft im privilegierten Modus und verwaltet die MMU, ist daher hochgradig maschinenabhängig.
- Das zweite Modul läuft ebenfalls im privilegierten Modus, ist aber maschinenunabhängig. Es behandelt Seitenfehler, Adreßräume und Seitenersetzung.
- Das dritte Modul läuft als Benutzerprozeß. Es behandelt hauptsächlich den logischen Teil der Speicherverwaltung insbesondere die Hintergrundspeicher (memory manager, external pager).
- Kommunikation zwischen Modulen durch Nachrichten an ‘ports’, die an Module gebunden sind

❑ Virtueller Speicher

- Adreßräume sind linear
- Effiziente Handhabung dünn besetzter Adreßräume durch das Konzept der Regionen (regions)
- Regionen können zugeteilt und freigegeben werden, Anfangsadresse ist bei Zuteilung angebbar.



Das Interface zwischen Kern und ‘external pager’ besteht aus folgenden Methoden

- Methoden des Kerns für Benutzerfäden (vm_)

Methode	Beschreibung
allocate	Eine Region virtueller Adressen zugänglich machen
deallocate	Eine Region von Adressen unzugänglich machen
map	Ein Speicherobjekt in den virtuellen Adreßraum abbilden
copy	Kopie einer Region an einer anderen Stelle des virtuellen Adreßraums anlegen
vm_inherit	Vererbungsattribut für eine Region setzen
read	Einen Bereich aus dem virtuellen Adreßraum eines anderen Prozesses lesen
write	Einen Bereich in den virtuellen Adreßraum eines anderen Prozesses kopieren

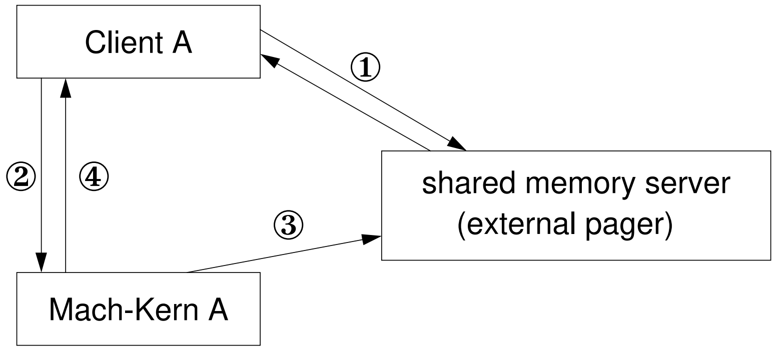
- Nachrichten des Kerns an den ‘external pager’ (memory_object_)

Nachricht	Beschreibung
init	Initialisieren eines neu abgebildeten Speicherobjektes
data_request	Dem Kern ist die wegen eines Seitenfehlers benötigte Seite zu übergeben
data_write	Rückgabe einer modifizierten Seite
data_unlock	Anforderung von Zugriffsattributen durch den Kern
lock_completed	Eine frühere Aufforderung zum Rückschreiben modifizierter Seiten ist vollzogen
terminate	Anzeige, daß das Objekt zurückgegeben ist und nicht mehr benutzt wird

Nachrichten des ‘external pager’ an den Kern (memory_object_)

Nachricht	Beschreibung
set_attributes	Übergabe von Attributen an den Kern z. B. zur Steuerung der Ersetzungsstrategie
data_provided	Übergabe der angeforderten Seite (Antwort auf data_request)
data_unavailable	Angeforderte Seite ist nicht verfügbar (Antwort auf data_request)
lock_request	Aufforderung an den Kern modifizierte Seiten zurückzuschreiben, zu invalidieren oder Zugriffsattribute zu ändern (wird beantwortet mit lock_completed)
destroy	Aufforderung an den Kern, das Objekt zurückzugeben (wird beantwortet mit terminate)

Beispiel: Initialisierung



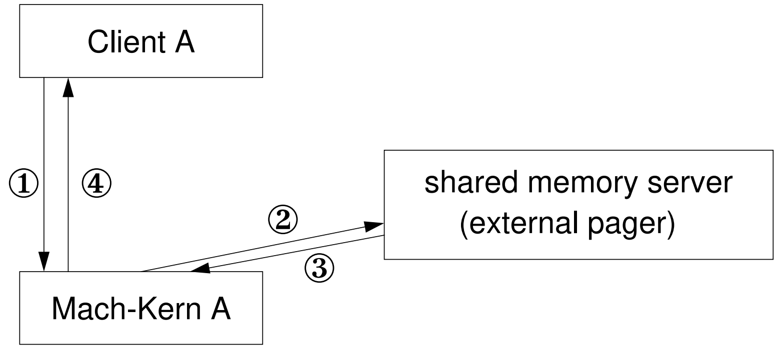
① get_object

② allocate + map

③ init

④ client resumed

Seitenfehler



① Zugriff mit Seitenfehler

② data_request

③ data_provided

④ client resumed

Verteilter, gemeinsam genutzter Speicher

❏ Anforderungen

- Beispiel: parallele Bearbeitung von Satellitendaten mit unterschiedlichen Programmen
- Differenzierte Behandlung der Regionen bei fork: ‘text’-Segmente können gemeinsam genutzt werden
- manche Regionen (z. B. solche die Dateien beherbergen) können für den erzeugten Prozeß irrelevant sein
- bei manchen Regionen kann der Anfangszustand relevant sein, aber es sollen erzeugender und erzeugter Prozeß eigene Kopien besitzen

❏ Antwort in Mach: Vererbungsattribute

- Region wird vom erzeugten Prozeß nicht benötigt
- Region wird von erzeugendem und erzeugtem Prozeß kooperativ benutzt
- erzeugter Prozeß erhält Kopie der Region

❏ Organisation

- externer Speicherverwalter, läuft auf Benutzerebene
- drei ports
  - ‘object port’ beim Speicherverwalter
  - ‘control port’ im Kern als Adressat für Nachrichten eines Speicherverwalters
  - ‘name port’ wird als eine Art Name für das Speicherobjekt benutzt (z. B. kann zu einer vorgebbaren virtuellen Adresse erfragt werden, zu welchem Speicherobjekt sie gehört. Die Antwort besteht aus einem Zeiger auf den ‘name port’.)

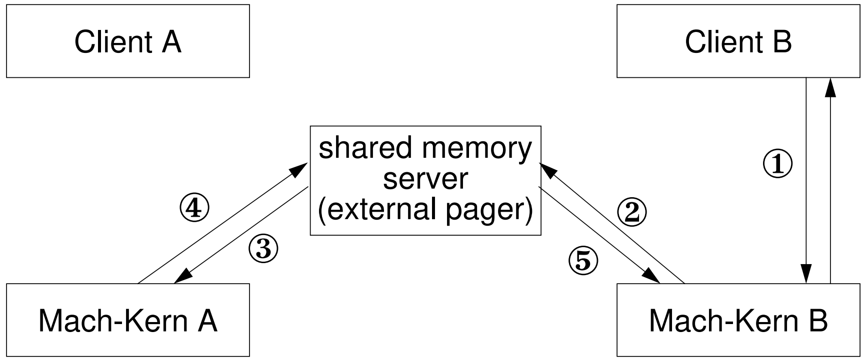
❏ Gemeinsam genutzter Speicher

Sowohl für NORMA- als auch für NUMA-Architekturen von Interesse

Wichtige Gesichtspunkte

- Struktur und Granularität
- Kohärenz-Semantik
- Skalierbarkeit
- Implementierung
  - Datenlokalisierung und Zugriff
  - Kohärenzprotokoll (write-invalidate, write-update)
  - Ersetzungsstrategie
  - Thrashing
  - Interaktionsmechanismen

Gemeinsam genutzter Speicher,von A angesprochene Seite wird von B lesend angesprochen



① Zugriff mit Seitenfehler
② data_request
③ lock_request (read only)
④ lock_completed
⑤ data_provided (read only)

Speicherkohärenz

Strikte Konsistenz

Lesen liefert zuletzt geschriebenen Wert

Sequentielle Konsistenz

Die Ausführung erscheint als zeitliche Durchmischung der Operationen, wie sie bei Mehrprozeßbetrieb auftreten kann.

Prozessorkonsistenz

Schreibaufrufe eines Prozessors werden immer in der Reihenfolge ihrer Anfordrung sichtbar. Über Anforderungen von verschiedenen Prozessoren kann keine Aussage über die Reihenfolge gemacht werden, in der ihre Ausführung sichtbar wird.

Schwache Konsistenz

Koordinierungsoperationen sind sequentiell konsistent.

Vor Zugriff auf Koord.-Variablen müssen alle Schreibvorgänge abgeschlossen sein.

Zugriff auf Daten nur, wenn alle vorherigen Zugriffe auf Koord.-Variablen abgeschlossen sind.

Freigabe (Release) Konsistenz

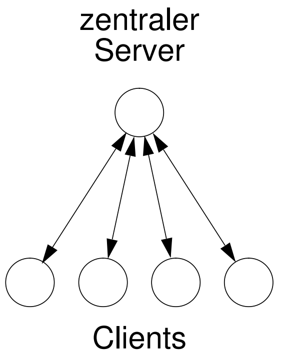
Jede der beiden Koordinierungsoperationen acquire und release ist prozessorkonsistent

Vor Ausführung von release müssen alle vorherigen Schreib- und Lesevorgänge des Prozesses abgeschlossen sein

- Algorithmen für die Plazierung von Seiten

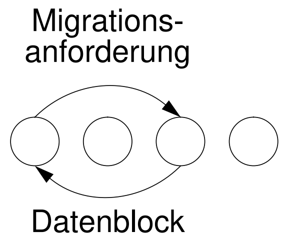
	nicht vervielfachend	vervielfachend
Nicht migrierend	zentralisiert (central server)	allgemein vervielfachend (full-replication)
migrierend	migrierend (migration)	vervielfachend für Lesezugriff (read-replication)

- Zentralisierter Server (Central-server algorithm)



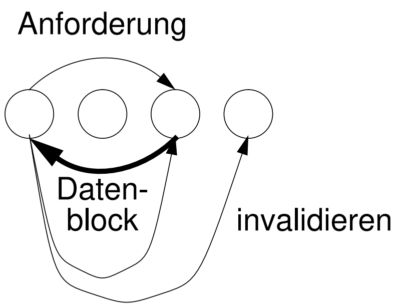
Client	zentraler Server
Datenanforderung senden	
	Anforderung entgegennehmen Datenzugriff durchführen Antwort senden
Antwort entgegennehmen	

- Migrationsalgorithmus



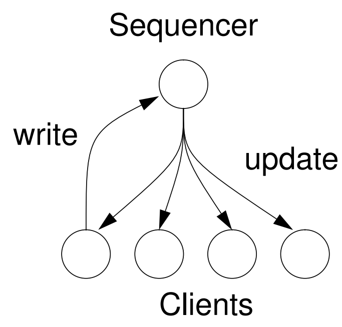
Client	Verwalter
Falls Block nicht lokal, Speicherort ermitteln und Anforderung senden	
	Anforderung entgegennehmen Block senden
Antwort entgegennehmen Datenzugriff durchführen	

- Vervielfachung für lesenden Zugriff



Client	Verwalter
Falls Block nicht lokal, Speicherort ermitteln und Anforderung senden	
	Anforderung entgegennehmen Block senden
Antwort entgegennehmen Invalidierung an alle anderen	
	Invalidierung vollziehen
Zugriff	

- Allgemein vervielfachend



Client	Sequencer	Verwalter
Falls ‘write’ Daten an Sequencer		
	Sequenznummer anfügen Multicast	
		Daten empfangen ‘update’ lokal
Ausführung bestätigt ‘update’ lokal		

- Vergleichende Analyse

  Stumm, M.; Zhou, S.: Algorithms Implementing Distributed Shared Memory. Computer, May 1990, pp. 54-64.