4.2 Threads, Prozessorvergabe

Literatur

Kleiman, S.: Shah, D.; Smaalders, B.: Programming with Threads.SunSoft Press, 1996.

❑ 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 einer allgemeinen 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.

  - 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.

- Implementierung von leichtgewichtigen Fäden (lightweight processes) im Betriebssystem
  - Beispiel: SOLARIS lwp-Schnittstelle
    - Erzeugung und Tilgung
      - _lwp_makecontext(...)
      - _lwp_create(...)
      - _lwp_continue(...)
      - _lwp_exit(...)
    - Koordination
      - _lwp_wait(...)

Beispiel: Erzeugung eines ‘kernel thread’.

```
#include <sys/types.h>
#include <sys/lwp.h>
#include <ucontext.h>
#include <signal.h>
#include <iostream.h>
#include <malloc.h>

extern "C" void sleep(int);

void func(char *s)
{ cout << “Ich bin da\n” << flush;
}
```

```
int main()
{ ucontext_t *contextp;
  caddr_t stackbase;
  size_t stacksize = 1000000;
  char arg[1000], Private[1000];
  int error;
  lwpid_t new_lwp;

  cout << “Beginn des Hauptprogramms\n” << flush;
  contextp
    = (ucontext_t *)malloc(sizeof(ucontext_t));
  stackbase = (caddr_t)malloc(stacksize);
  sigprocmask(SIG_SETMASK, NULL,
              &contextp->uc_sigmask);
  _lwp_makecontext(contextp, func,
                   arg, Private,
                   stackbase, stacksize);
  error = _lwp_create(contextp, NULL, &new_lwp);
  cout << “error = “ << error << “\n” << flush;
  sleep(20);
  cout << “Ende des Hauptprogramms\n” << flush;
}
```

```
faui45q: 15:42 ~ [73] >a.out
Beginn des Hauptprogramms
error = 0
Ich bin da
Ende des Hauptprogramms
faui45q: 15:50 ~ [74] >
```


- Implementierung von leichtgewichtigen Fäden (lightweight processes) durch Multiplexen schwer- oder mittelgewichtiger Fäden
  - Beispiel: POSIX pthreads
    - Erzeugung und Tilgung
      - pthread_create(...)
      - pthread_exit(...)
      - pthread_cancel(...)
    - Koordination
      - pthread_join(...)
    - Attribute
      - pthread_attr_init(...)

Beispiel: Es werden fünf Fäden erzeugt und das Hauptprogramm wartet auf die Beeendigung aller Fäden.

```
#include <pthread.h>
#include <iostream.h>

extern "C" void sleep(int);

#define NUM_THREADS 5
#define SLEEP_TIME 10

void *sleeping(void *);   // thread routine
void test_argv();         // optional
int i;
pthread_t tid[NUM_THREADS];
                          // array of thread IDs
int sleep_time = SLEEP_TIME;

void *sleeping(void *sleep_time)
{ cout << "thread " << pthread_self()
       << " sleeping "
       << SLEEP_TIME << " seconds ...\n";
  sleep(*((int *)sleep_time));
  cout << "thread " << pthread_self()
       << " awakening\n";
}
```



```
int main(int argc, char **argv)
{ for ( i = 0; i < NUM_THREADS; i++)
    pthread_create(&tid[i], NULL, &sleeping,
                   &sleep_time);

  for ( i = 0; i < NUM_THREADS; i++)
    pthread_join(tid[i], NULL);

  cout << "main() reporting that all " << i
       << " threads have terminated\n";
}
```

```
faui45q: 15:25 96/programme [45] > time a.out
thread 4 sleeping 10 seconds ...
thread 5 sleeping 10 seconds ...
thread 6 sleeping 10 seconds ...
thread 7 sleeping 10 seconds ...
thread 8 sleeping 10 seconds ...
thread 5 awakening
thread 6 awakening
thread 7 awakening
thread 4 awakening
thread 8 awakening
main() reporting that all 5 threads have termi-
nated
0.04u 0.11s 0:10.09 1.4%
faui45q: 15:25 96/programme [46] >
```

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 weitergeschoben, 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 Kommunikationsstruktur; 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

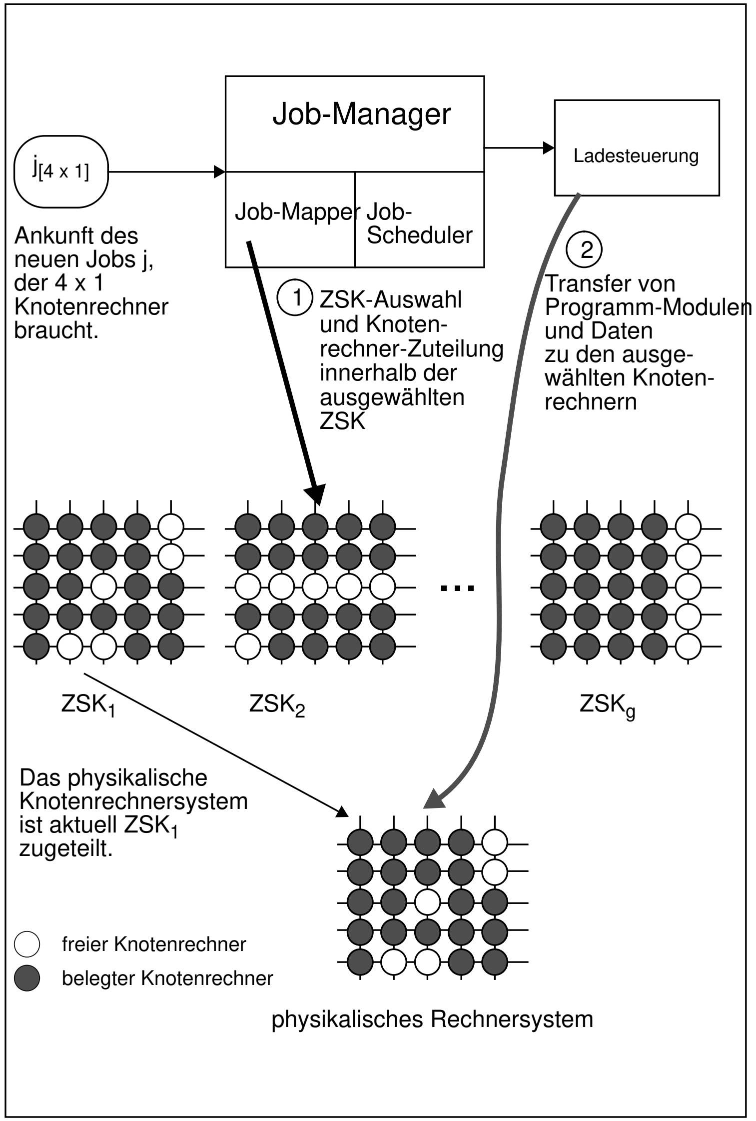
Eine *Zeitscheibenklasse (ZSK)* eines Parallelrechnersystems PS ist ein virtuelles Knotenrechnersystem, das folgende Eigenschaften hat:

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

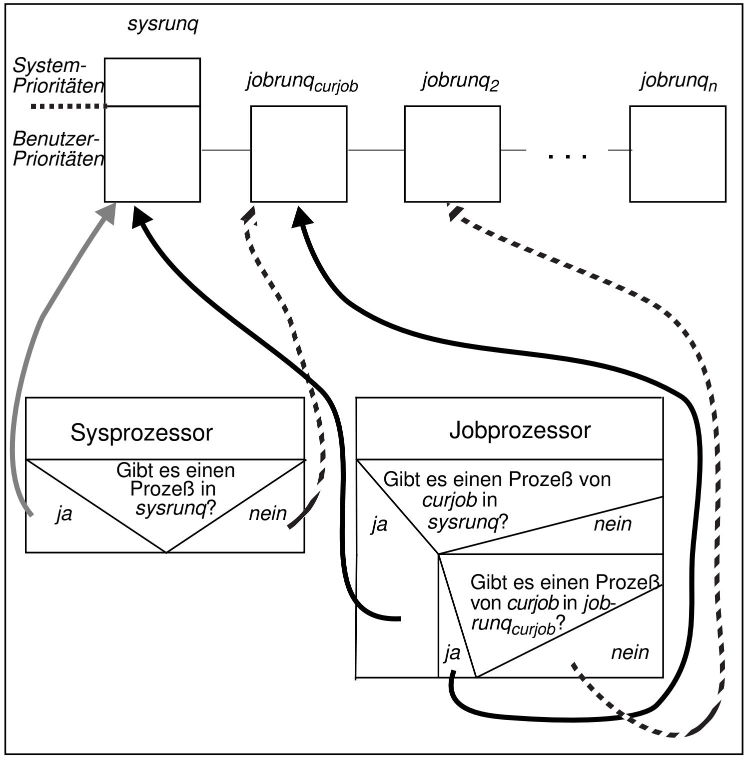
(b) ZSK hat die gleiche Verbindungstopologie wie PS.

(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.

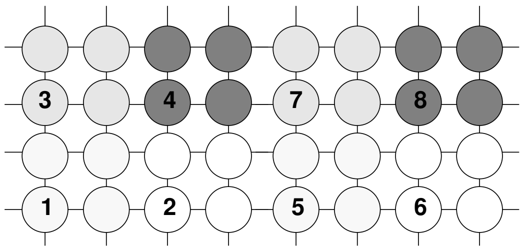
Einordnung neuer Aufträge



Knotenscheduling

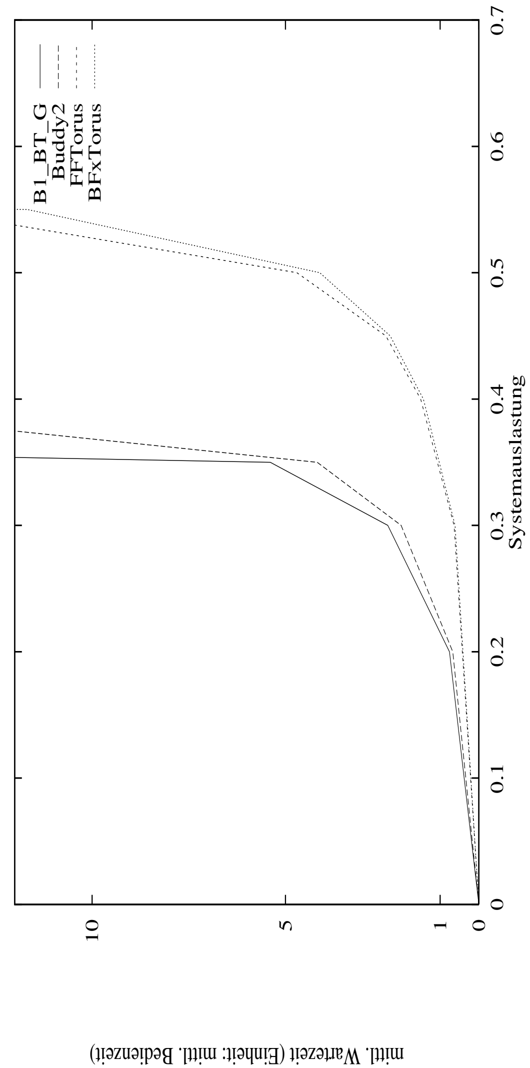


Anpassung der eindimensionalen Buddy-Strategie



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

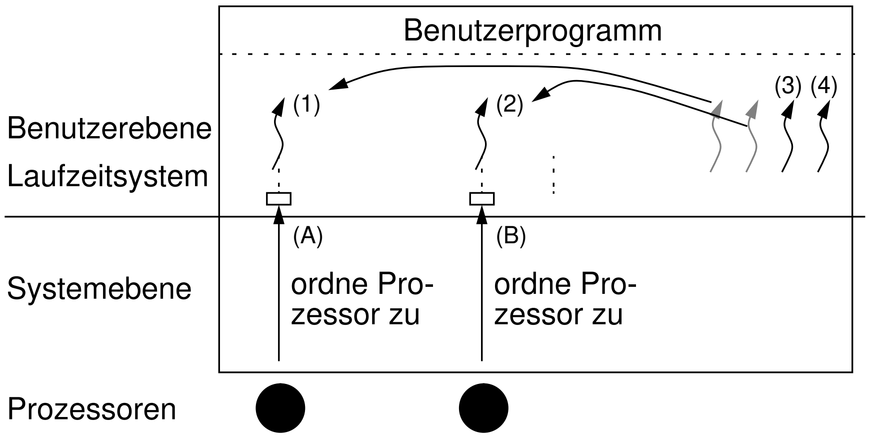
  Benutzer gibt Hinweise

  - Hinweise auf Unzweckmäßigkeit des Zuordnens
  - Hinweise auf zuzuordnende Prozesse

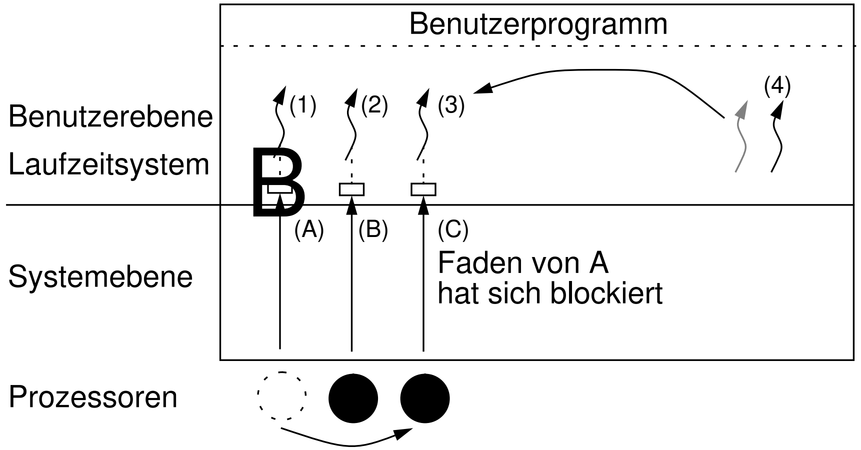
  Letzteres ist insbesondere im Zusammenhang mit Kooperation von Interesse.

❏ Mischung aus Kern- und Benutzerfäden

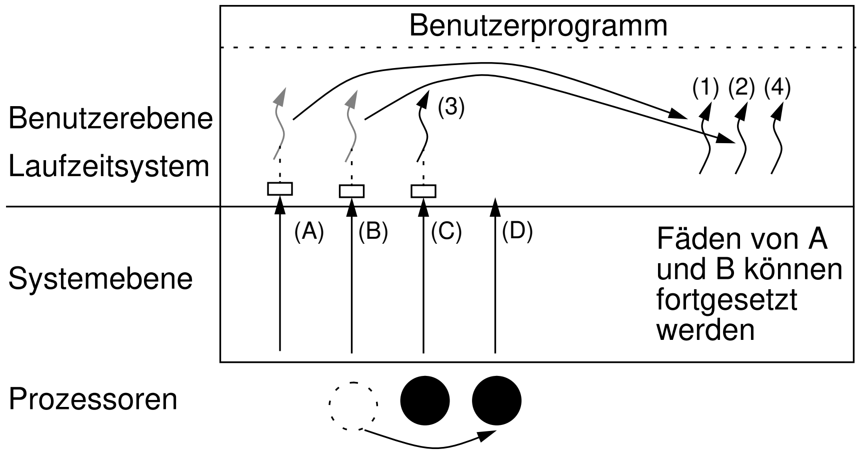
- Kern ordnet der Anwendung zwei Prozessoren zu



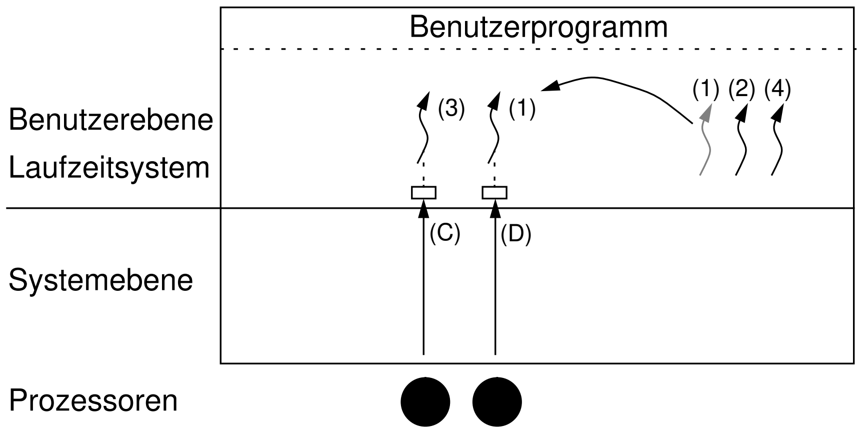
- Ein Benutzerfaden blockiert sich im Kern wegen E/A



- E/A abgeschlossen



• ‘upcall’ wegen freien Prozessors



❏ Forschung:

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

POSIX-Standard: IEEE Portable Operating System Interface
standard P1003.1c

auch: Pthreads, ISO/IEC 9945-1:990

Benutzerschnittstelle:

- pthread_create()
- pthread_exit()
- pthread_join()
- pthread_yield()
- pthread_self()
- pthread_kill()
- pthread_sigmask()
- pthread_setschedparam()
- pthread_getschedparam()
- pthread_key_create()
- pthread_setspecific()
- pthread_getspecific()
- pthread_once()
- pthread_equal()
- pthread_cancel()
- pthread_testcancel()
- pthread_cleanup_push()
- pthread_cleanup_pop()

pthread_setcanceltype()

pthread_setcancelstate()

pthread_mutex_lock()

pthread_mutex_unlock()

pthread_mutex_trylock()

pthread_mutex_init()

pthread_mutex_destroy()

pthread_cond_wait()

pthread_cond_timedwait()

pthread_cond_signal()

pthread_cond_broadcast()

pthread_cond_init()

pthread_cond_destroy()