Echtzeitbetriebssysteme

Oliver Jack

Ernst-Abbe-Hochschule Jena
Fachbereich Elektrotechnik und Informationstechnik

Sommersemester 2025

Ernst-Abbe-Hochschule Jena
University of Applied Sciences

Lerneinheit 2. Laufzeitmodelle

Lerneinheit 2. Laufzeitmodelle

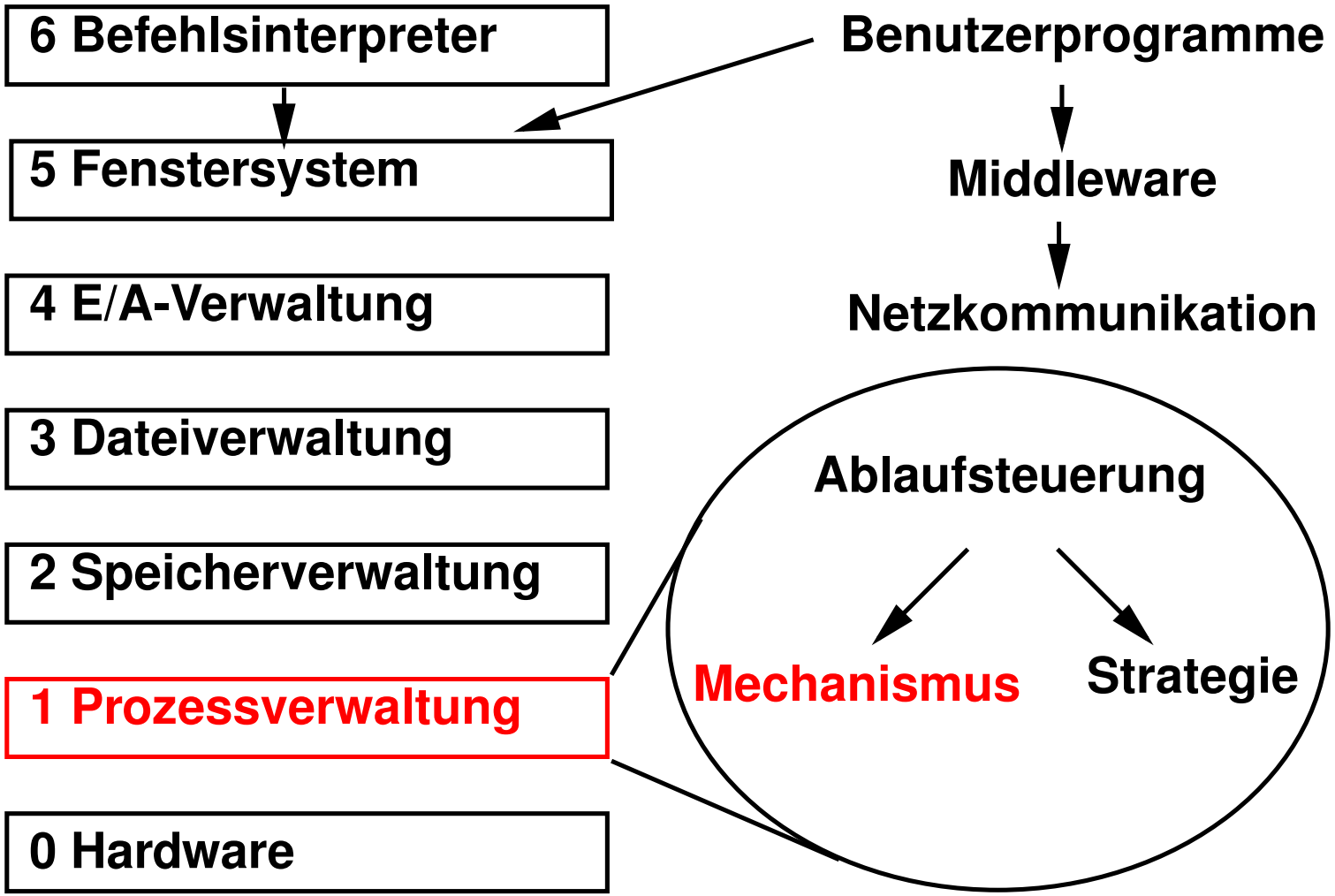
Lernziele

- Kenntnis von Threads und Tasks
- Kenntnis von Prozesszuständen
- Kenntnis von Prozesskontrollmechanismen
- Kenntnis von Kernel-Level und User-Level-Threads

Lerneinheit 2. Laufzeitmodelle

Lerneinheit 2. Laufzeitmodelle

Funktionale Hierarchie moderner Betriebssysteme



Prozessverwaltung (process management)

- Ein Prozess ist ein Objekt der Verwaltung mit Operationen zur geeigneten Manipulation.
- Prozessverwaltung besteht aus Prozessumschaltung (Mechanismus) und Scheduling (Strategie).
- Prozessverwaltung ist die zeitliche Zuordnung eines physischen Prozessors zu Prozessen, das heißt Multiplexing.
- Die primäre Unterbrechungsbehandlung wird durchgeführt.
- Eng verzahnt mit der Prozessverwaltung sind
  - Speichermanagement (Adressraum = Abstraktion des physischen Speichers),
  - Prozessinteraktion und -synchronisation (Konkurrenz, Kooperation).

Thread und Task

Thread

- Threads sind virtuelle Prozessoren.
- Abstraktionen eines realen Prozessors
- Stellen Einheiten dar, die von einem Scheduler (Planer) bezüglich der Ressourcennutzung einzuplanen sind

Task

- Ein Task ist ein Rechenprozess.
- Ein Task besteht aus
  - ausführbarem Programmcode
  - eigenem Speicher(-bereich)
  - eigenen Variablen

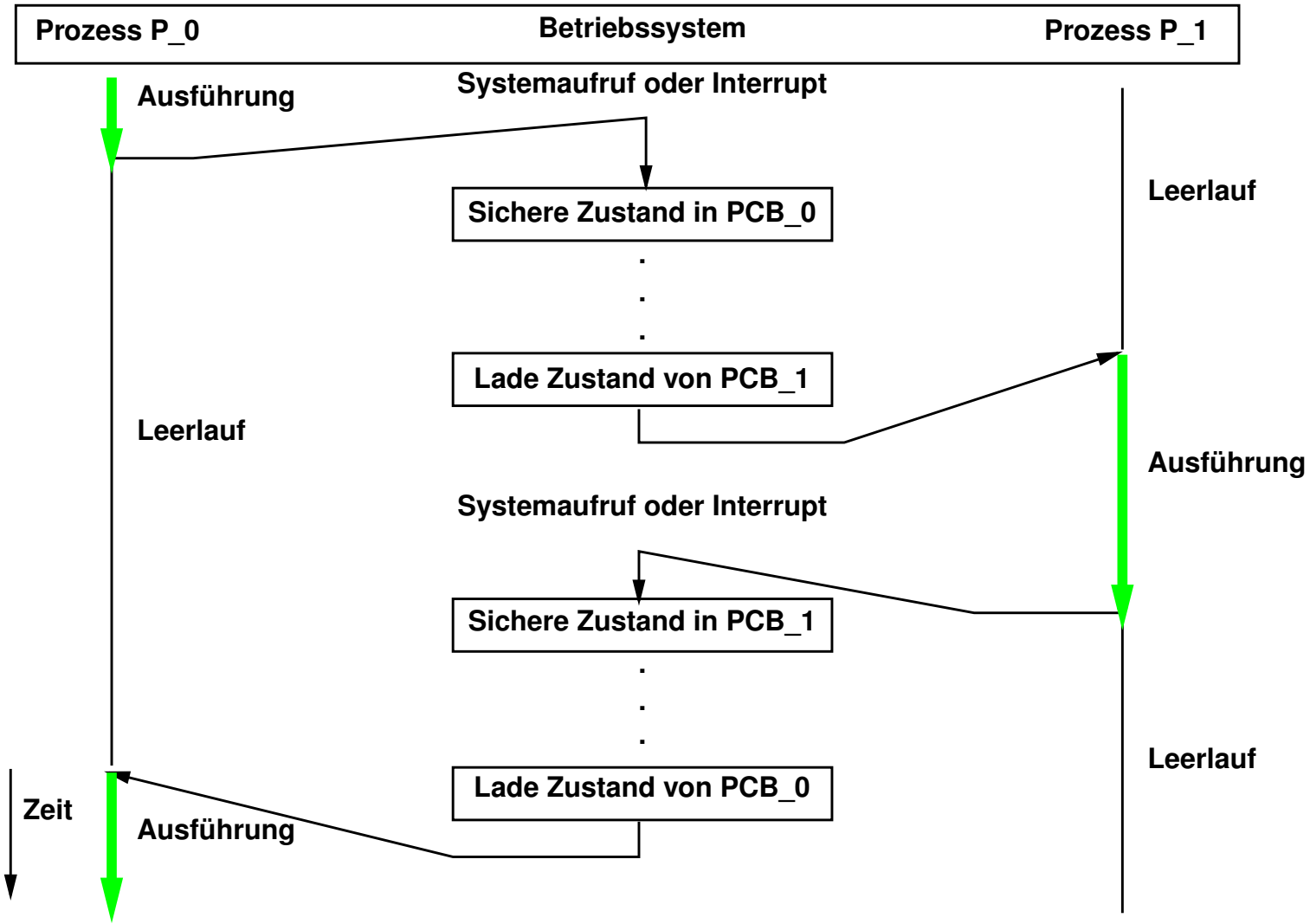
  Ein Task besistzt i.A. eine Priorität. Ein Task besitzt einen Zustand.

Threads und Tasks können dynamisch erzeugt und terminiert werden.

Prozesse und Adressräume

- Ein (logischer) Adressraum eines Prozesses ist die Gesamtheit seiner gültigen Adressen, auf die er zugreifen kann.
- Moderne Prozessoren ermöglichen eine relative Adressierung durch Basisregister und stellen eine Speicherabbildungsfunktion (Memory Management Unit, MMU) zur Verfügung.
- Damit können beliebig viele *logische Adressräume* automatisch und gestreut auf den physikalischen Speicher abgebildet werden.
- Die Adressräume sind dadurch gegenseitig geschützt.
- Adressräume sind unabhängig von Prozessen zu sehen.

Prozessumschaltung



Prozessumschaltung

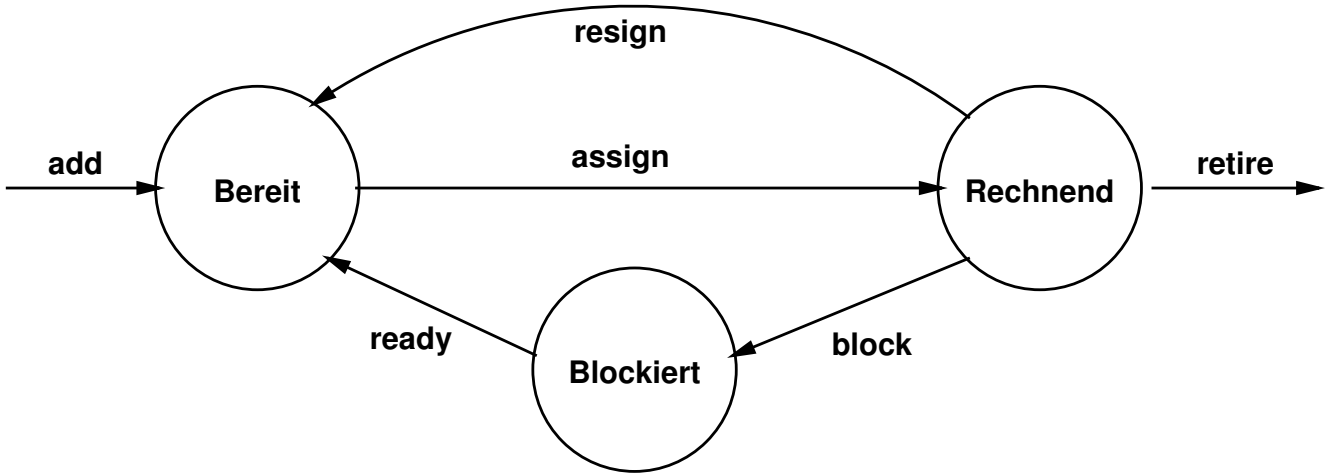
- Systemsoftware erzwingt durch Zeitscheibenverfahren implizit eine zeitlich versetzte Ausführung der Threads.
- Es erfolgt ein Kontextwechsel: der Zustand des aktuell ausgeführten Threads wird gespeichert, der vorher gespeicherte Zustand eines anderen Threads wird vom Prozessor übernommen (Dispatcher).
- Auswahl des Kandidaten für den Kontextwechsel übernimmt der Scheduler: die Wahl hängt vom Optimierungsziel der Prozessverwaltung ab.
- Teil des Kontextwechsels kann ein Adressraumwechsel sein.
- Umschaltung kann erfolgen bei
  - Aufruf einer blockierenden E/A-Operation,
  - freiwilliger Abgabe des Prozessors (Yielding),
  - Eintreffen eines asynchronen Hardware-Interrupts, z. B. Timer.

Verdrängung (preemption)

- In vielen Einsatzgebieten sind nicht alle Prozesse/Threads von gleicher Wichtigkeit oder Dringlichkeit, was zur Verwendung von Prioritäten führt.
- Ziel ist es, zu jedem Zeitpunkt einen Prozess/Thread mit höchster Priorität auszuführen.
- Prozessumschaltung erfolgt sobald ein Prozess/Thread mit höherer Priorität als der des aktuell rechnenden bereit ist.
- Der rechnende Prozess/Thread wird durch einen höher priorisierten verdrängt.

Lerneinheit 2. Laufzeitmodelle

Zustandsmodell



Rechnend Thread ist im Besitz eines physischen Prozessors.

Blockiert Thread wartet auf die Beendigung einer E/A-Operation oder Eintritt einer Synchronisationsbedingung.

Bereit Thread ist potentiell ausführbar, aber nicht im Besitz eines physischen Prozessors.

Zustandsübergänge

add Ein neuer Thread wird dynamisch in die Menge der bereiten Threads eingefügt.

assign Durch Kontextwechsel wird einem bereiten Thread der Prozessor zugeordnet. Der ausgewählte Thread wird ausgeführt.

block Bei Aufruf einer blockierenden E/A- oder Synchronisationsoperation wird dem rechnenden Thread der Prozessor entzogen.

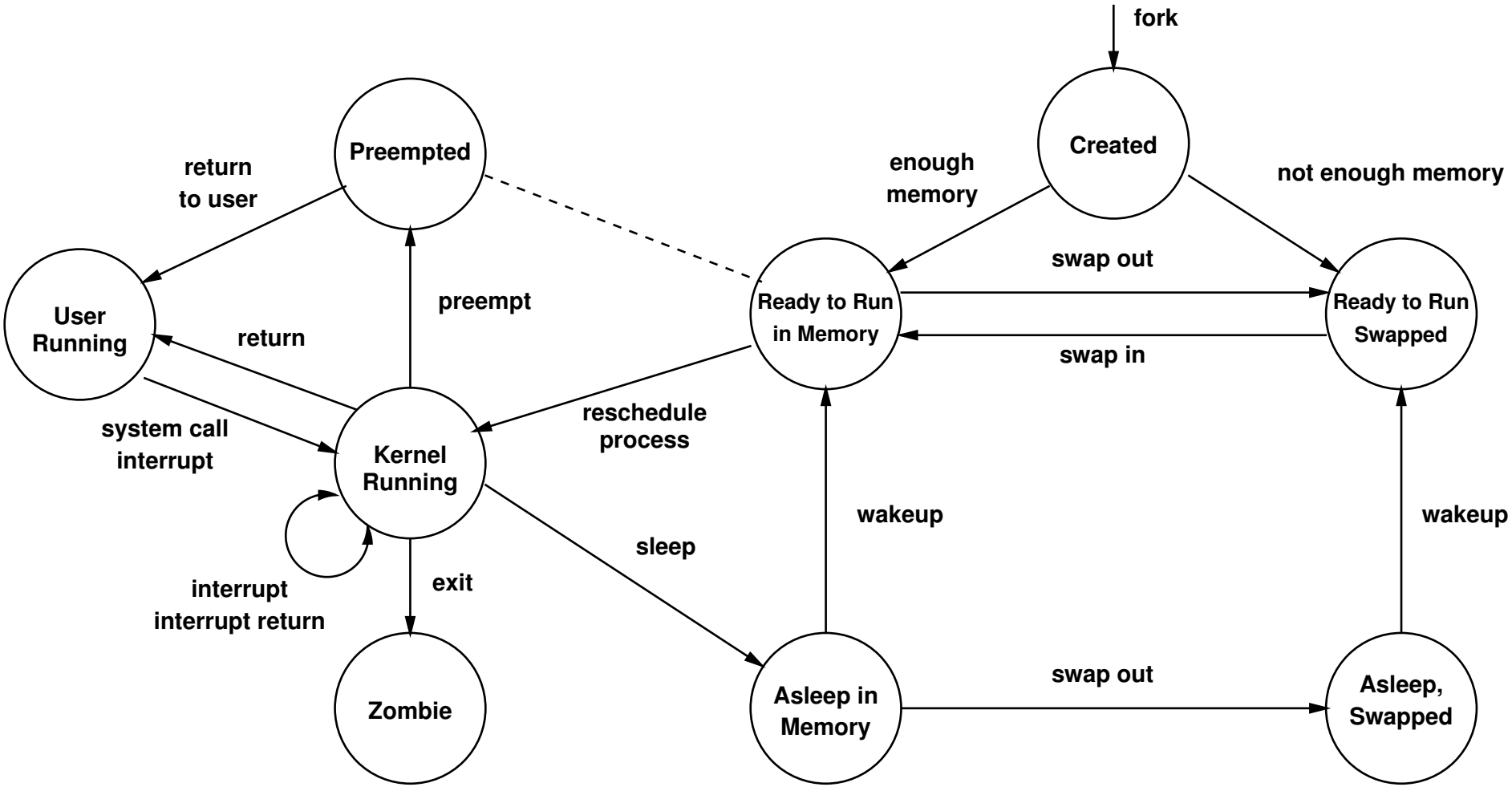
Zustandsübergänge

ready Ein blockierter Thread wechselt nach Beendigung einer angestoßenen Operation in den Zustand “Bereit” und bewirbt sich um einen physischen Prozessor

resign Einem rechnenden Thread wird der Prozessor, z. B. auf Grund eines Timer-Interrupts, vorzeitig entzogen. Er bewirbt sich erneut um einen Prozessor.

retire Ein aktuell rechnender Thread terminiert und gibt den Prozessor ab. Belegte Hard- und Software-Ressourcen werden freigegeben.

Prozesszustände in UNIX



Umsetzung des Zustandsmodells

Thread-Zustand wird durch einen Prozesskontrollblock (PCB) beschrieben, der alle für die Prozessverwaltung nötigen Informationen enthält:

- eindeutige Identifikation PID,
- Speicherplatz zur Sicherung des Prozesszustands bei Kontextwechsel,
- Informationen über den Wartegrund im Fall eines blockierenden Threads,
- Adressrauminformationen, Verweis auf Adressraumbeschreibung,
- Zustandsinformationen und Statistiken für das Scheduling,
- Scheduling-Priorität.

Lerneinheit 2. Laufzeitmodelle

Dispatcher (Monoprozessorsystem)

- Arbeitet auf Listen von PCBs, hier vereinfacht als abstrakter Datentyp dargestellt.

```
List(PCB) running = 0;
List(PCB) ready = 0;
List(PCB) blocked = 0;
```

- Neuer Thread

```
Dispatcher.Add (NewPCB) {
  ready.Put(NewPCB);
}
```

- Thread terminieren

```
Dispatcher.Retire (pid) {
  return running.Get(pid);
}
```

Dispatcher (Monoprozessorsystem)

- Bei Kerneintritt: Sicherung des Prozessorzustands durch Funktion `M0.ContextSave(reg_save)` in einem besonderen Speicherbereich `reg_save`.
- Thread einem Prozessor zuordnen

```
Dispatcher.Assign () {
  PCB next;
  next = ready.Get();
  running.Put(next);
  M0.ContextRestore(next->r); /* r = Register */
}
```

- Thread den Prozessor entziehen

```
Dispatcher.Resign () {
  PCB current;
  current = running.Get();
  current->r = reg_save; /* r = Register */
  ready.Put(current);
}
```

Dispatcher (Monoprozessorsystem)

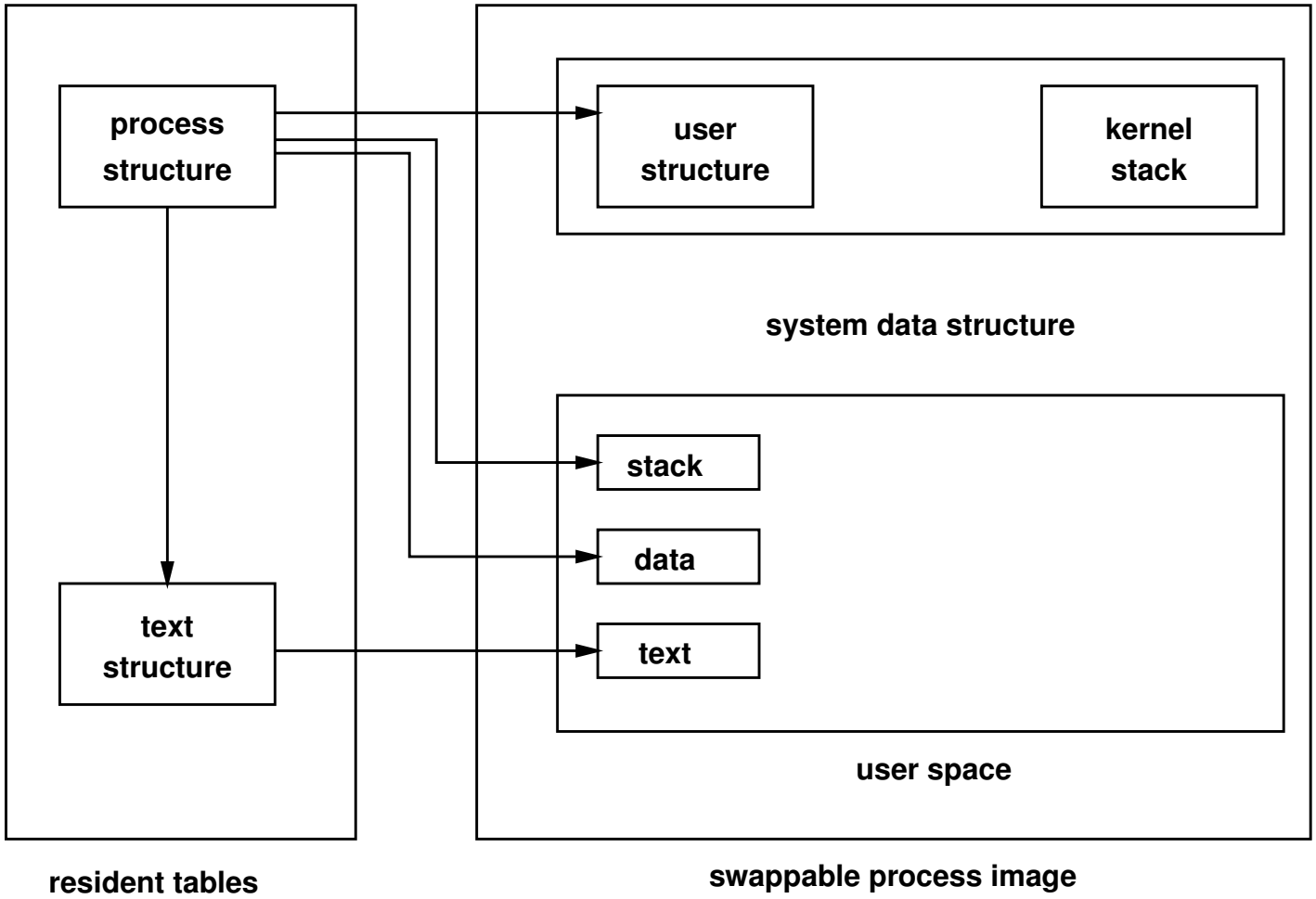
- Thread in den Zustand “Bereit” setzen

```
Dispatcher.Ready (pid) {
  PCB p;
  p = blocked.Get(pid);
  ready.Put(p);
}
```

- Thread blockieren

```
Dispatcher.Block () {
  PCB self;
  self = running.Get();
  self->r = reg_save;
  blocked.Put(self);
}
```

PCB in UNIX


process structure
user structure
kernel stack
system data structure
stack
data
text
text structure
user space
resident tables
swappable process image

Prozessverwaltung in UNIX

- Bei einem Systemaufruf läuft der Thread im Systemmodus, andernfalls im Benutzermodus.
- Ein Thread ist niemals simultan im Benutzer- und Systemmodus.
- Im Systemmodus wird der Kernel-Stack benutzt, andernfalls der Benutzer-Stack.
- Kernel-Stack und Benutzerstruktur formen das System-Daten-Segment.
- Prozessumschaltung erfolgt gemäß Zeitscheibenverfahren (round-robin), Takt 0,1 s mit 1 s Prioritätsanpassung.
- Jeder Thread besitzt eine Scheduling-Priorität, die dynamisch vom System geändert wird.
- Kernel-Threads haben höhere Priorität als Benutzer-Threads.
- Es findet keine Verdrängung von Kernel-Threads durch andere Kernel-Threads statt.

Prozessverwaltung in UNIX

- Blockieren erfolgt durch die Kernel-Operation `sleep`
- Argument von `sleep` ist die Adresse einer Kernel-Datenstruktur bezüglich des erwarteten Ereignisses.
- Bei Eintreten des Ereignisses ruft der entsprechende System-Prozess ein `wakeup` mit der zugehörigen Adresse auf.
- Alle Threads, die ein `sleep` auf dieses Ereignis haben, werden in den "Bereit"-Status gesetzt.

Prozesskontrolle auf Benutzerebene

- UNIX-Systemaufrufe `fork`, `execve`, `wait` und `exit`.
- `fork` erzeugt eine Kopie des originalen Prozesses mit dem selben Adressraum, Programm und Daten. Beide Prozesse, Eltern- und Kindprozess werden ausgeführt, bzw. sind bereit.
  - Der aufrufende Thread erhält als Rückgabewert die Prozessidentifikation des neu erzeugten Thread.
  - Der neu erzeugte Thread erhält als Rückgabewert 0.
- `execve` ersetzt in dem Prozess, in dem es aufgerufen wird, Adressraum, Programm und Daten und startet das Programm.
- `exit` terminiert den Prozess.
- Ein Elternprozess kann mit `wait` auf das `exit`-Ereignis seines/seiner Kindprozesse warten. Er ist dann blockiert.

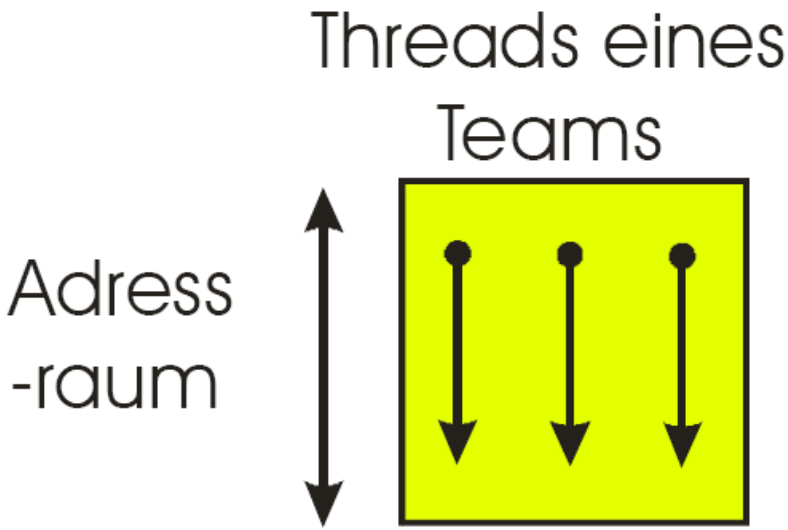
Prozesskontrolle auf Benutzerebene

```
cpid = fork();
if (cpid == 0) { /* Programmcode Kindprozess */ }
else { /* Programmcode Elternprozess */ }

cpid = fork();
if (cpid == 0) { execve ( Pfadname, Argumente, Environment ); }
else { /* ... */ wait(); /* ... */ }

execve ( Pfadname, Argumente, Environment );

/* POSIX: neuer Thread im selben Adressraum */
int pthread_create (
  pthread_t *tid, /* Thread-Id */
  pthread_attr_t *attr, /* UL- oder KL-Thread, Stackposition und -größe, */
                        /* Schedulingsinfo */
  void * (*start) (void *), /* Startfunktion */
  void *arg) /* Argument */

/* POSIX: Terminierung eines Threads */
int pthread_cancel ( pthread_t t )
```

Definition

Team

- Speichergekoppelte Tasks heißen ein Team
- Anderer Begriff: Leichgewichtige Tasks (Lightweight Tasks)
- Mehrere Threads benutzen
  - denselben physikalischen Prozessor und
  - einen gemeinsamen Speicheradressraum.



Implementierung des ganzen Teams auf einem einzigen Rechner

Definition

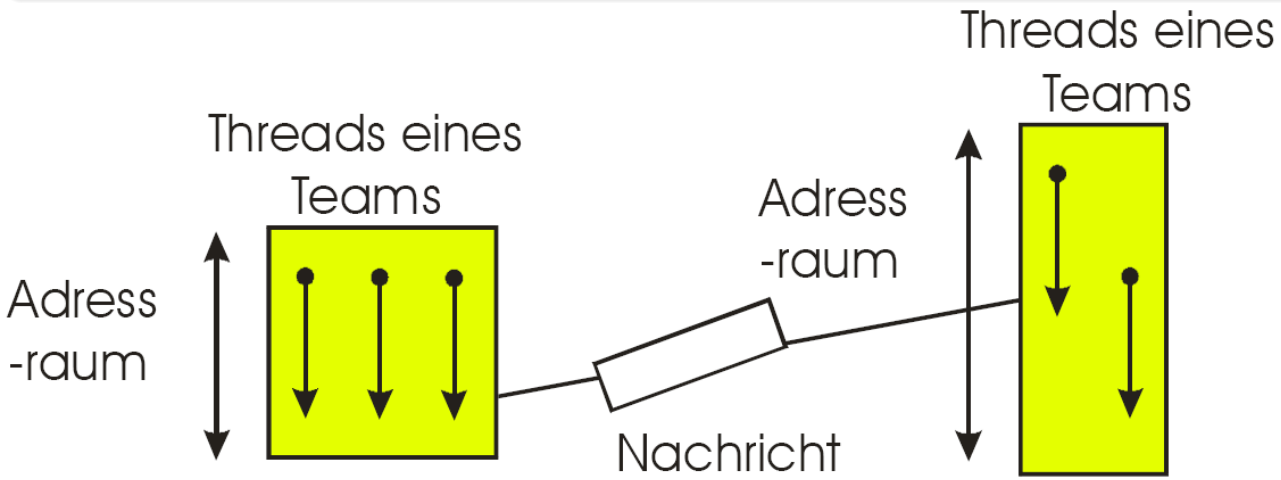
Nachrichtengekoppelte Tasks

- Jeder Task besteht aus einem einzigen Thread,
- Tasks
  - haben keinen gemeinsamen Adressraum und
  - kommunizieren über Nachrichten miteinander.
- Implementierung auf Mehrrechnersystemen bietet sich an

Definition

Nachrichtengekoppelte Teams

- Kombination der beiden Laufzeitmodelle
- Nachrichtengekoppelte Tasks und speichergekoppelte Tasks (Teams)
- Innerhalb eines Teams: Threads interagieren über den gemeinsamen Speicher
- Threads aus verschiedenen Teams: Kommunikation über Nachrichten



Lerneinheit 2. Laufzeitmodelle

Zeitaspekte

- Kreieren eines neuen Threads innerhalb eines Teams dauert nicht so lange wie kreieren eines neuen Teams (Messungen an UNIX-ähnlichen Systemen: Faktor 10)

Zeitaspekte

- Kreieren eines neuen Threads innerhalb eines Teams dauert nicht so lange wie kreieren eines neuen Teams (Messungen an UNIX-ähnlichen Systemen: Faktor 10)
- Beenden eines Threads ist kürzer als Beenden eines Teams

Zeitaspekte

- Kreieren eines neuen Threads innerhalb eines Teams dauert nicht so lange wie kreieren eines neuen Teams (Messungen an UNIX-ähnlichen Systemen: Faktor 10)
- Beenden eines Threads ist kürzer als Beenden eines Teams
- Umschalten zwischen Threads ist kürzer als zwischen Teams

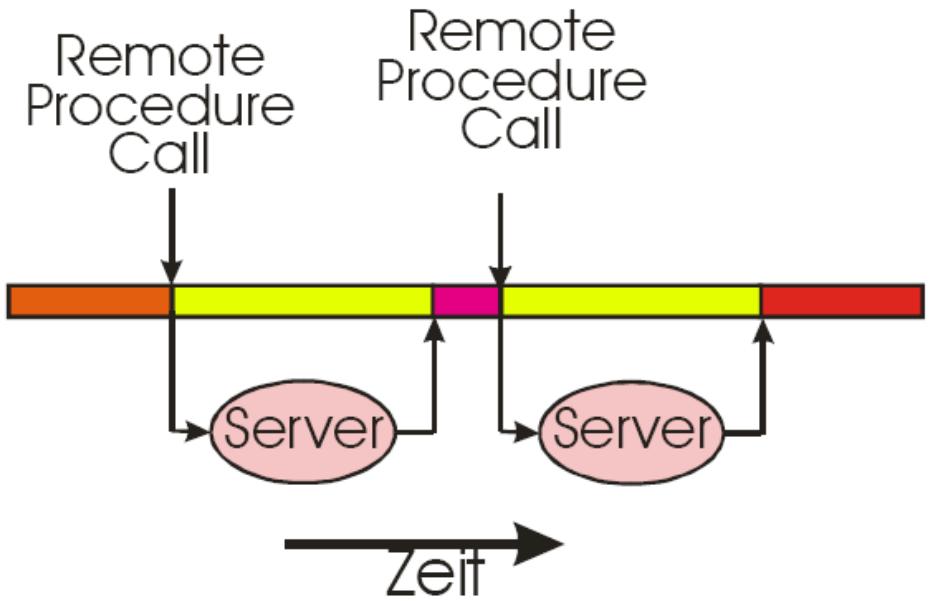
Zeitaspekte

- Kreieren eines neuen Threads innerhalb eines Teams dauert nicht so lange wie kreieren eines neuen Teams (Messungen an UNIX-ähnlichen Systemen: Faktor 10)
- Beenden eines Threads ist kürzer als Beenden eines Teams
- Umschalten zwischen Threads ist kürzer als zwischen Teams
- Kommunikation zwischen Threads braucht keinen Kernel-Eingriff

Zeitaspekte

Beispiel: Remote Porcedure Call

Ein Thread



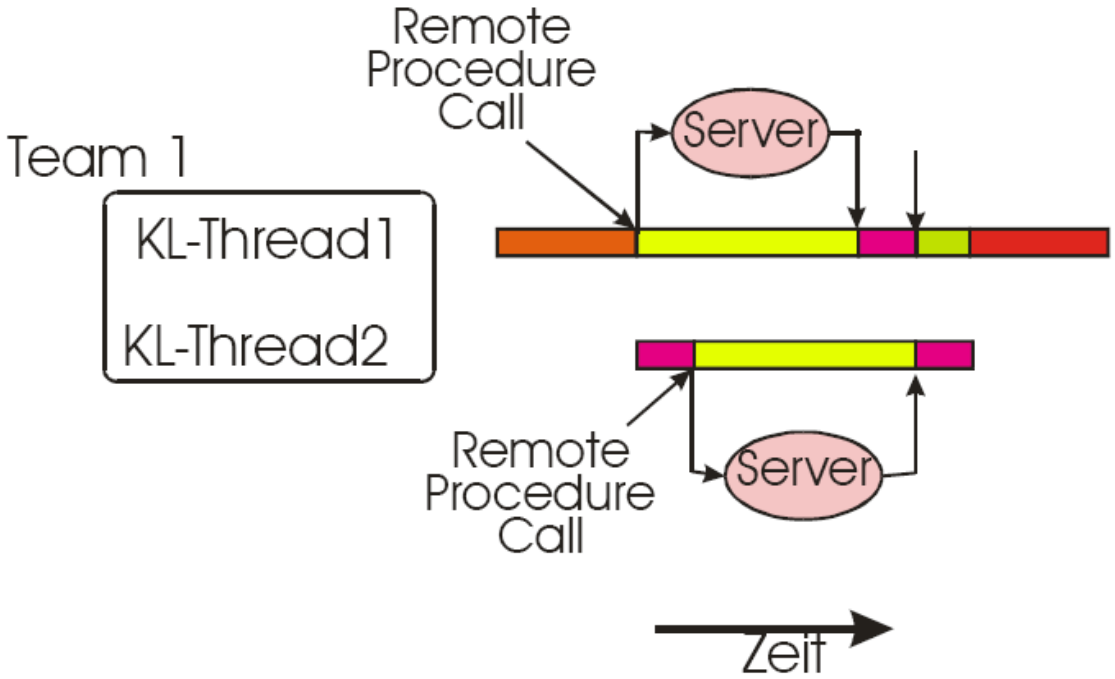
Thread belegt Prozessor

Team blockiert, wartet auf Antwort auf RPC

Zeitaspekte

Beispiel: Remote Porcedure Call

Zwei Threads



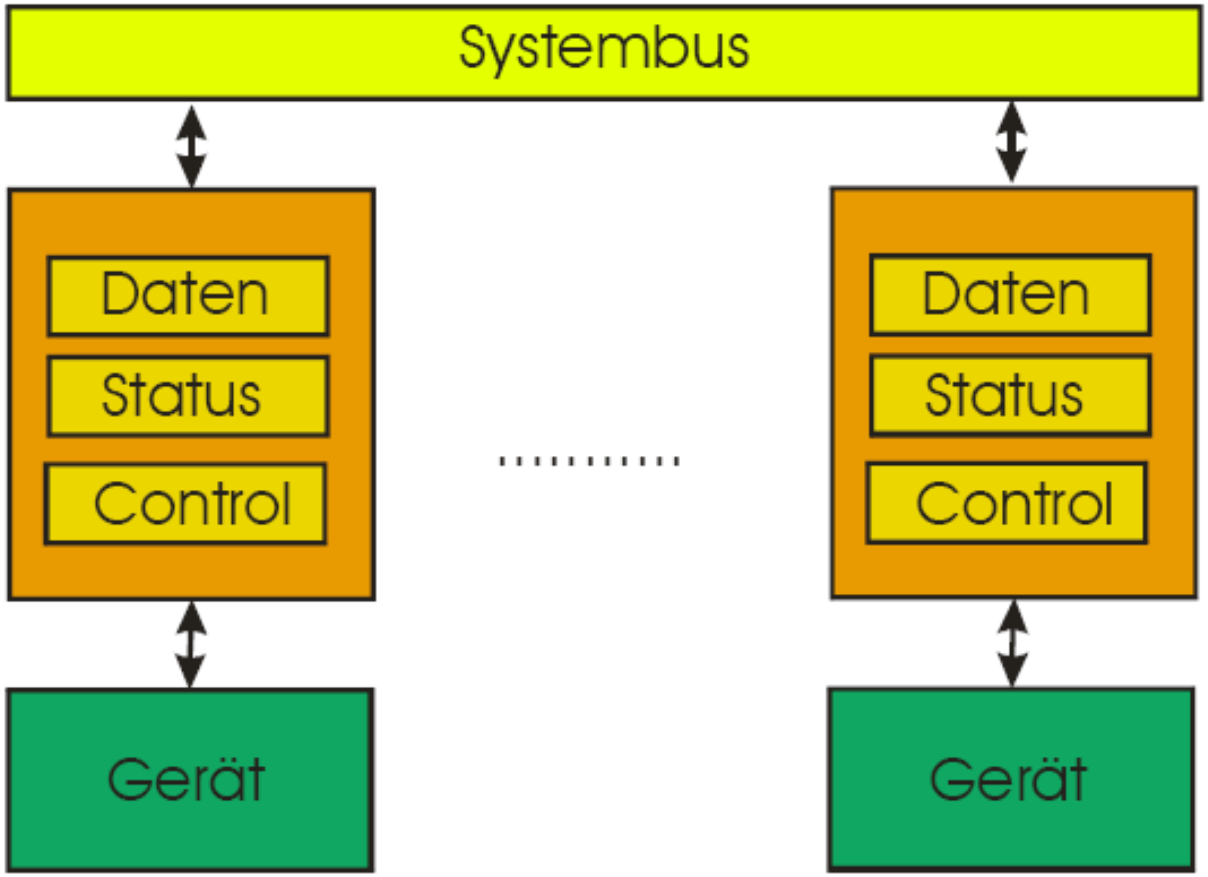
Thread belegt Prozessor

Thread blockiert, wartet auf Antwort auf RPC

Thread blockiert, wartet auf Prozessor

Erweiterung der elementaren Laufzeitmodelle

- Kontrollierter Gerätezugriff durch die Laufzeitumgebung notwendig
- Typischer Anschluss von Geräten (memory mapped)



Wichtig für Echtzeitsysteme

- Für harte Echtzeitanforderungen ist der vollständige Entzug der Gerätekontrolle problematisch

Wichtig für Echtzeitsysteme

- Für harte Echtzeitanforderungen ist der vollständige Entzug der Gerätekontrolle problematisch
- Einflussmöglichkeiten auf die Zuteilungsstrategien für Prozessor und Speicher durch Anwendung notwendig.

Wichtig für Echtzeitsysteme

- Für harte Echtzeitanforderungen ist der vollständige Entzug der Gerätekontrolle problematisch
- Einflussmöglichkeiten auf die Zuteilungsstrategien für Prozessor und Speicher durch Anwendung notwendig.
- Teilweise Kontrolle über E/A-Aktivitäten

Wichtig für Echtzeitsysteme

- Für harte Echtzeitanforderungen ist der vollständige Entzug der Gerätekontrolle problematisch
- Einflussmöglichkeiten auf die Zuteilungsstrategien für Prozessor und Speicher durch Anwendung notwendig.
- Teilweise Kontrolle über E/A-Aktivitäten
- Erweiterung des Thread-Konzeptes um Mechanismen zur anwendungsspezifischen Zuteilung von Prozessoren zu Threads

Wichtig für Echtzeitsysteme

- Für harte Echtzeitanforderungen ist der vollständige Entzug der Gerätekontrolle problematisch
- Einflussmöglichkeiten auf die Zuteilungsstrategien für Prozessor und Speicher durch Anwendung notwendig.
- Teilweise Kontrolle über E/A-Aktivitäten
- Erweiterung des Thread-Konzeptes um Mechanismen zur anwendungsspezifischen Zuteilung von Prozessoren zu Threads
- Einführung von Prioritäten

Wichtig für Echtzeitsysteme

- Für harte Echtzeitanforderungen ist der vollständige Entzug der Gerätekontrolle problematisch
- Einflussmöglichkeiten auf die Zuteilungsstrategien für Prozessor und Speicher durch Anwendung notwendig.
- Teilweise Kontrolle über E/A-Aktivitäten
- Erweiterung des Thread-Konzeptes um Mechanismen zur anwendungsspezifischen Zuteilung von Prozessoren zu Threads
- Einführung von Prioritäten
- Seitenaustauschverfahren einschränken

Ausschluss von Bereichen vom Seitenaustauschverfahren

- Bei Ein-Ausgabe unbedingt erforderlich (I/O-Locking)

Ausschluss von Bereichen vom Seitenaustauschverfahren

- Bei Ein-Ausgabe unbedingt erforderlich (I/O-Locking)
- Seite, auf die bei I/O zugegriffen werden soll, darf während der I/O-Operation nicht ausgelagert werden

Ausschluss von Bereichen vom Seitenaustauschverfahren

- Bei Ein-Ausgabe unbedingt erforderlich (I/O-Locking)
- Seite, auf die bei I/O zugegriffen werden soll, darf während der I/O-Operation nicht ausgelagert werden
- Echtzeitbetrieb: Harte Zeitbedingungen können bei unkontrolliertem Ein- und Auslagern nicht eingehalten werden: Zugriffszeit auf Speicherhierarchie nicht vorhersagbar

Ausschluss von Bereichen vom Seitenaustauschverfahren

- Bei Ein-Ausgabe unbedingt erforderlich (I/O-Locking)
- Seite, auf die bei I/O zugegriffen werden soll, darf während der I/O-Operation nicht ausgelagert werden
- Echtzeitbetrieb: Harte Zeitbedingungen können bei unkontrolliertem Ein- und Auslagern nicht eingehalten werden: Zugriffszeit auf Speicherhierarchie nicht vorhersagbar
  - Anwendungen können virtuelle Adressbereiche vom Seitenaustauschverfahren ausschließen.

Ausschluss von Bereichen vom Seitenaustauschverfahren

- Bei Ein-Ausgabe unbedingt erforderlich (I/O-Locking)
- Seite, auf die bei I/O zugegriffen werden soll, darf während der I/O-Operation nicht ausgelagert werden
- Echtzeitbetrieb: Harte Zeitbedingungen können bei unkontrolliertem Ein- und Auslagern nicht eingehalten werden: Zugriffszeit auf Speicherhierarchie nicht vorhersagbar
  - Anwendungen können virtuelle Adressbereiche vom Seitenaustauschverfahren ausschließen.
  - Ressource Arbeitsspeicher wird für die anderen Programmteile knapper

Ausschluss von Bereichen vom Seitenaustauschverfahren

- Bei Ein-Ausgabe unbedingt erforderlich (I/O-Locking)
- Seite, auf die bei I/O zugegriffen werden soll, darf während der I/O-Operation nicht ausgelagert werden
- Echtzeitbetrieb: Harte Zeitbedingungen können bei unkontrolliertem Ein- und Auslagern nicht eingehalten werden: Zugriffszeit auf Speicherhierarchie nicht vorhersagbar
  - Anwendungen können virtuelle Adressbereiche vom Seitenaustauschverfahren ausschließen.
  - Ressource Arbeitsspeicher wird für die anderen Programmteile knapper
  - Zulässigkeit der Anwendung dieser Methode nur bei entsprechender Privilegstufe

Wichtige Aspekte

Explizites Sperren durch die Anwendung

Detaillierte Kenntnisse über den Aufbau des virtuellen Adressraums und der gegenseitigen Abhängigkeiten von Teiladressräumen notwendig

Wichtige Aspekte

Explizites Sperren durch die Anwendung

Detaillierte Kenntnisse über den Aufbau des virtuellen Adressraums und der gegenseitigen Abhängigkeiten von Teiladressräumen notwendig

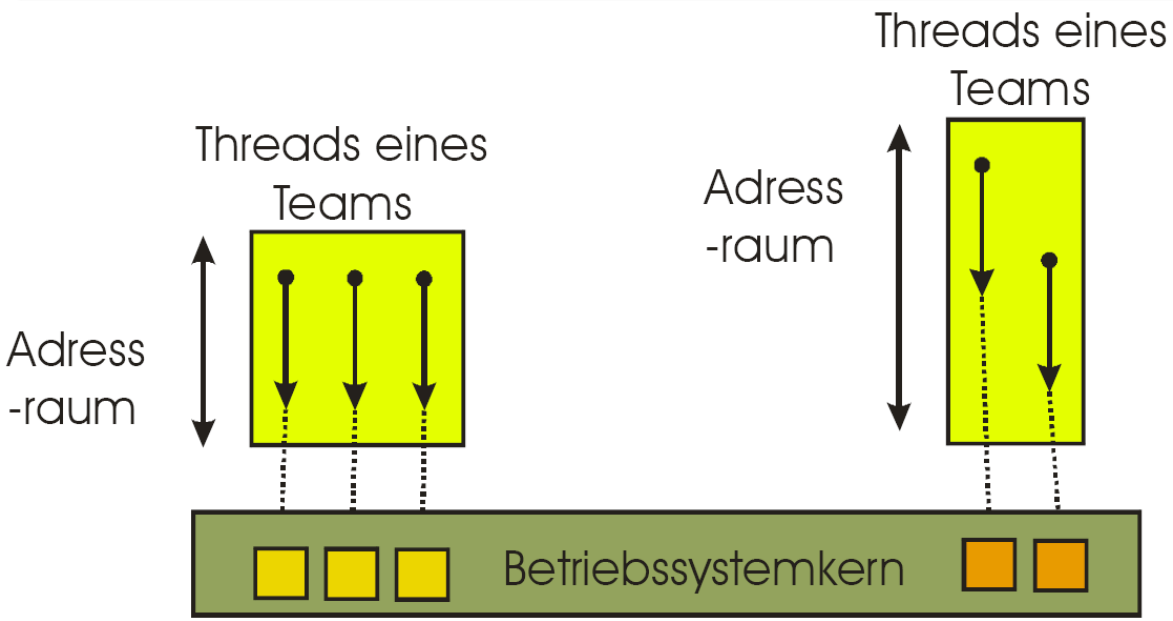
Beispiele

- Programmbereiche und die entsprechenden Datenbereiche gemeinsam sperren
- Bibliotheksfunktionen vor dem Verdrängen bewahren, wenn sie aus einem im Speicher festgesetzten Programmbereich aufgerufen werden.

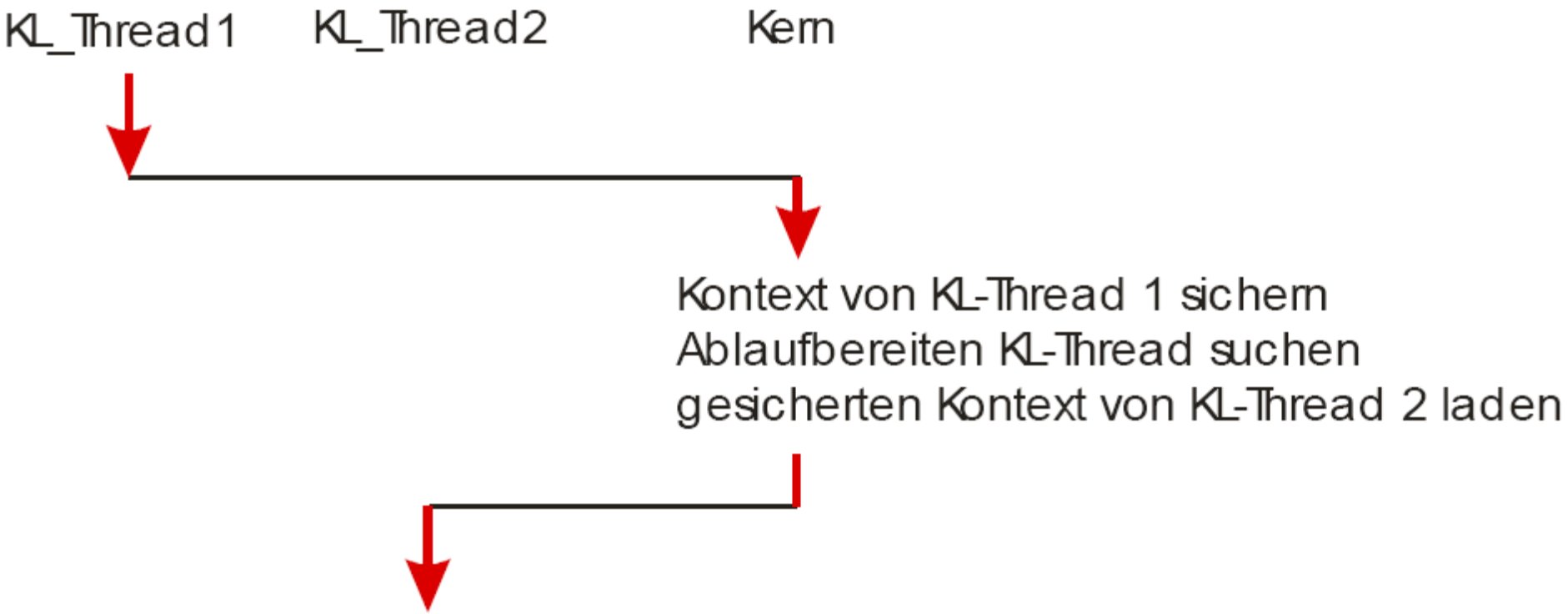
Lerneinheit 2. Laufzeitmodelle

Kernel-Level-Threads

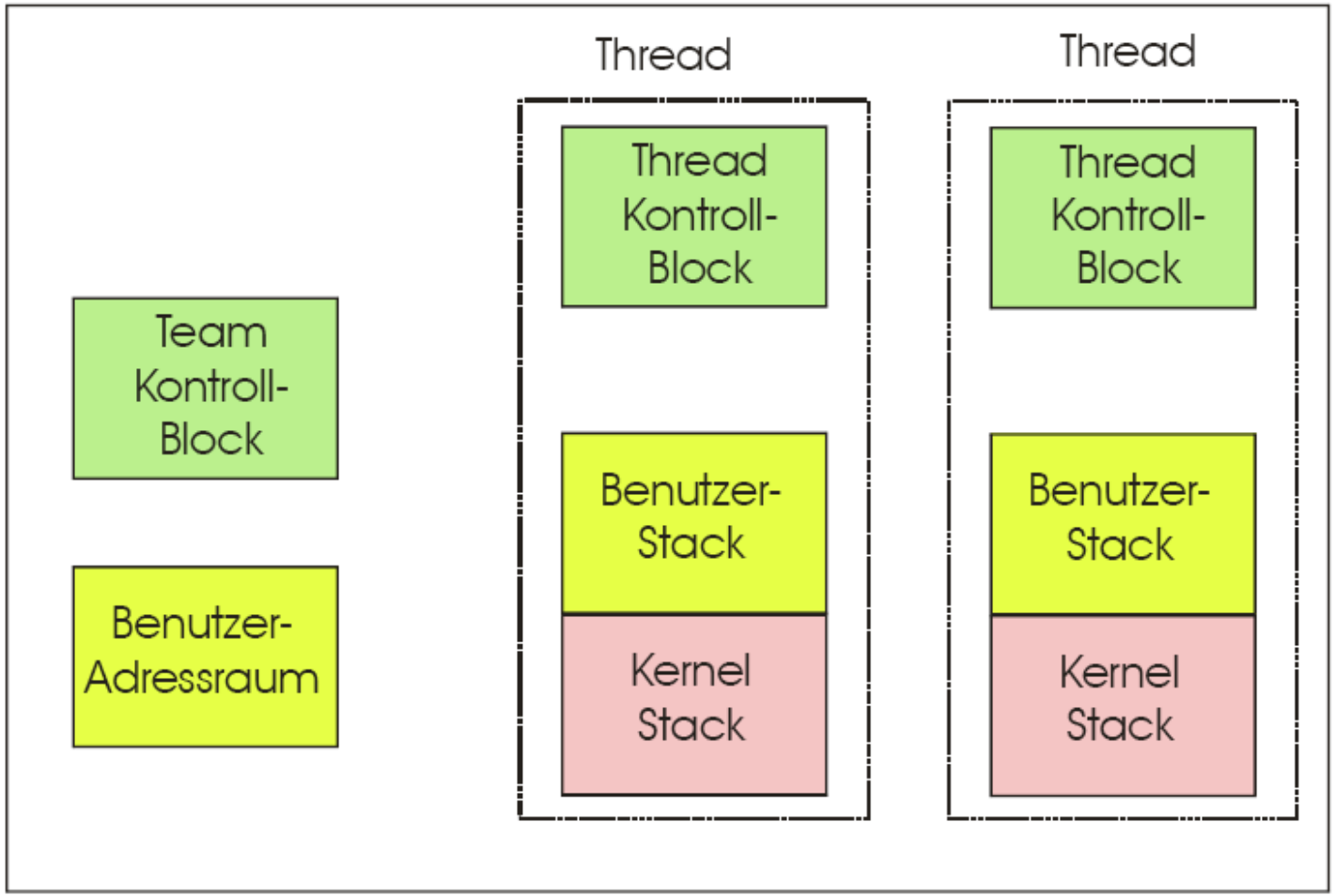
- Einheiten, die vom Scheduler ausgewählt werden, um ihnen nacheinander bzw. verzahnt den Prozessor zuzuteilen.
- Umschaltung zwischen zwei solchen Threads erfordert immer die Übergabe der Kontrolle an den Betriebssystemkern



Thread-Wechsel



Multithreaded Teammodell



Kernel-Level-Threads

Fall 1: Beide Threads gehören zu unterschiedlichen Teams

- Kontextwechsel bedeutet Adressraumwechsel
  - Ändern des Segment- und des Seitentabellenbasisregisters
  - Ungültigsetzen der Einträge der Translation-Lookaside-Tabelle (TLB).
- Unmittelbare Zusatzzeiten sind gering (Umspeicherung einiger Register)
- Mittelbare Folgekosten: Neuaufbau des TLB und des Cache-Inhalts führen zunächst zu längeren Zugriffszeiten auf die Speicherhierarchie (schwergewichtig)

Kernel-Level-Threads

Fall 2: Beide Threads im selben Team

- Adressraum muss nicht gewechselt werden.
- Trotzdem Verzögerung: Lokalitätsmenge ändert sich.

Kernel-Level-Threads

Fall 2: Beide Threads im selben Team

- Adressraum muss nicht gewechselt werden.
- Trotzdem Verzögerung: Lokalitätsmenge ändert sich.

Strategie

Betriebssystem wird bevorzugt jeweils einen ablaufbereiten Thread im selben Team auswählen

Nachteile von Kernel-Level-Threads

- Einschalten des Betriebssystems zur Durchführung des Threadwechsels
- Aus Schutzgründen: Trap-Aufruf (System-Aufruf)
- Teuer im Vergleich zu einem Prozeduraufruf (dauern z.B. doppelt so lang).

Gründe für Mehraufwand

- Instruktionen zum Systemaufruf sind aufwendig

Gründe für Mehraufwand

- Instruktionen zum Systemaufruf sind aufwendig
- Durch Wechsel des Adressraums oder zumindest der Lokalitätsmenge entstehen zusätzliche Verzögerungszeiten

Gründe für Mehraufwand

- Instruktionen zum Systemaufruf sind aufwendig
- Durch Wechsel des Adressraums oder zumindest der Lokalitätsmenge entstehen zusätzliche Verzögerungszeiten
- Die Ausführung allgemeiner Verwaltungsfunktionen des Betriebssystems erzeugt zusätzliche Verarbeitungszeiten

Gründe für Mehraufwand

- Instruktionen zum Systemaufruf sind aufwendig
- Durch Wechsel des Adressraums oder zumindest der Lokalitätsmenge entstehen zusätzliche Verzögerungszeiten
- Die Ausführung allgemeiner Verwaltungsfunktionen des Betriebssystems erzeugt zusätzliche Verarbeitungszeiten
- Beim Aufruf der Betriebssystemfunktion aus dem Kern heraus, kann ein Wechsel zu einem anderen Team (= anderen Adressraum) stattfinden.

Lerneinheit 2. Laufzeitmodelle

Benutzerthreads (User-Level-Threads)

Anwendung organisiert, gestützt auf Bibliotheksfunktionen (Thread package), die Threadverwaltung.

- Kreieren eines Threads, d.h. Aufruf der entsprechenden Bibliotheksfunktion (Prozeduraufruf)

Benutzerthreads (User-Level-Threads)

Anwendung organisiert, gestützt auf Bibliotheksfunktionen (Thread package), die Threadverwaltung.

- Kreieren eines Threads, d.h. Aufruf der entsprechenden Bibliotheksfunktion (Prozeduraufruf)
- Kreieren: Zuweisen einer entsprechenden Datenstruktur

Benutzerthreads (User-Level-Threads)

Anwendung organisiert, gestützt auf Bibliotheksfunktionen (Thread package), die Threadverwaltung.

- Kreieren eines Threads, d.h. Aufruf der entsprechenden Bibliotheksfunktion (Prozeduraufruf)
- Kreieren: Zuweisen einer entsprechenden Datenstruktur
- Kreieren: Einweisen in eine Liste bereiter Threads

Benutzerthreads (User-Level-Threads)

Anwendung organisiert, gestützt auf Bibliotheksfunktionen (Thread package), die Threadverwaltung.

- Kreieren eines Threads, d.h. Aufruf der entsprechenden Bibliotheksfunktion (Prozeduraufruf)
- Kreieren: Zuweisen einer entsprechenden Datenstruktur
- Kreieren: Einweisen in eine Liste bereiter Threads
- Verwalten: Scheduling-Algorithmus

Benutzerthreads (User-Level-Threads)

- Arbeiten ausschließlich im Benutzeradressraum (Kontextwechsel entspricht Prozeduraufruf)

Benutzerthreads (User-Level-Threads)

- Arbeiten ausschließlich im Benutzeradressraum (Kontextwechsel entspricht Prozeduraufruf)
- Threadverwaltung des Kerns weiß nichts von der Existenz solcher Threads.

Benutzerthreads (User-Level-Threads)

- Arbeiten ausschließlich im Benutzeradressraum (Kontextwechsel entspricht Prozeduraufruf)
- Threadverwaltung des Kerns weiß nichts von der Existenz solcher Threads.
- Kern betrachtet das Team als einen einzigen Thread und weist ihm einen der Zustände *bereit*, *ablaufend*, *blockiert* zu.

Benutzerthreads (User-Level-Threads)

- Arbeiten ausschließlich im Benutzeradressraum (Kontextwechsel entspricht Prozeduraufruf)
- Threadverwaltung des Kerns weiß nichts von der Existenz solcher Threads.
- Kern betrachtet das Team als einen einzigen Thread und weist ihm einen der Zustände *bereit*, *ablaufend*, *blockiert* zu.
- Die Bibliotheksfunktionen weisen den Threads entsprechende Zustände zu

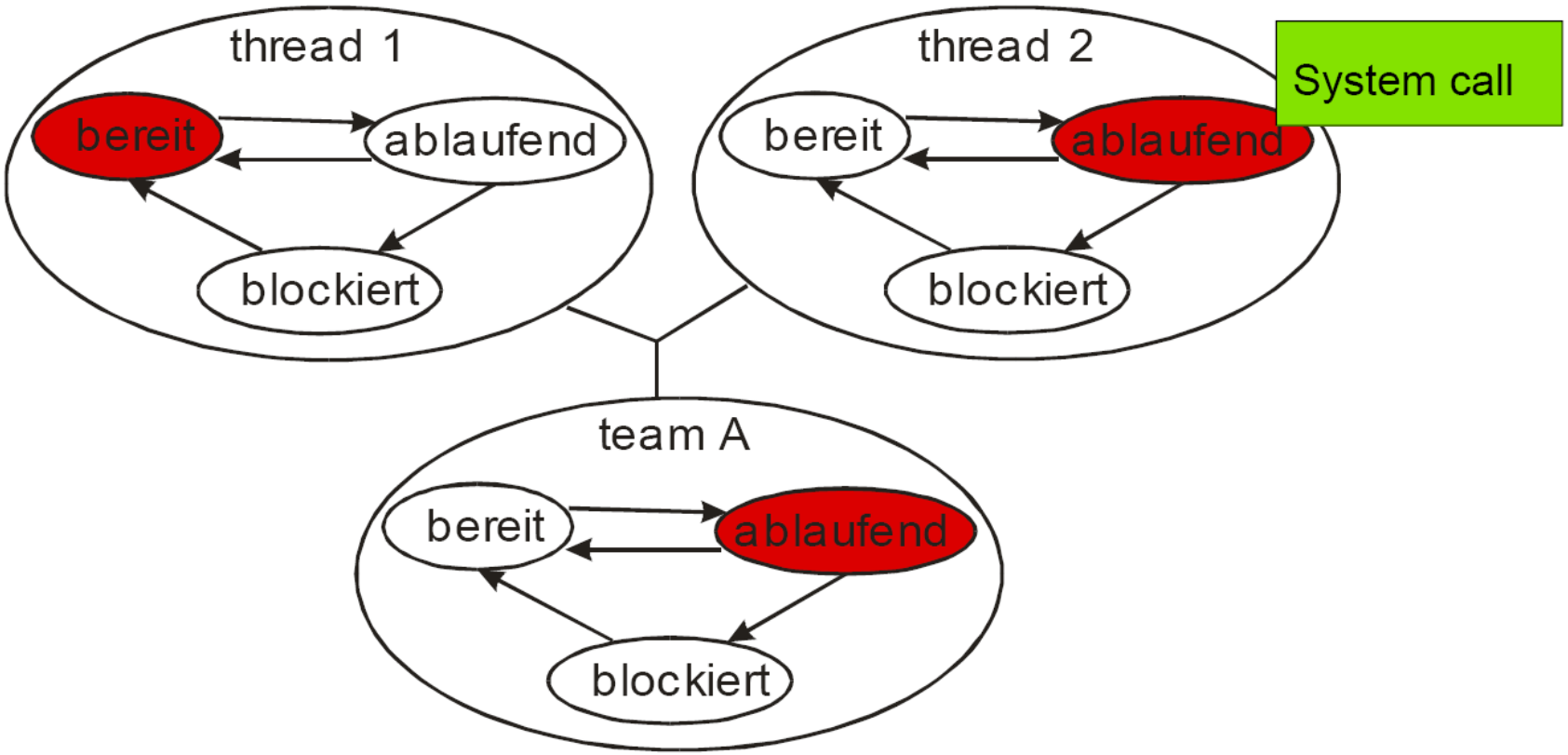
Benutzerthreads (User-Level-Threads)

- Arbeiten ausschließlich im Benutzeradressraum (Kontextwechsel entspricht Prozeduraufruf)
- Threadverwaltung des Kerns weiß nichts von der Existenz solcher Threads.
- Kern betrachtet das Team als einen einzigen Thread und weist ihm einen der Zustände *bereit*, *ablaufend*, *blockiert* zu.
- Die Bibliotheksfunktionen weisen den Threads entsprechende Zustände zu
- Zustände haben nicht unbedingt die Bedeutung, die ihrem Namen entspricht

Benutzerthreads (User-Level-Threads)

Team A besteht aus Thread 1 und Thread 2

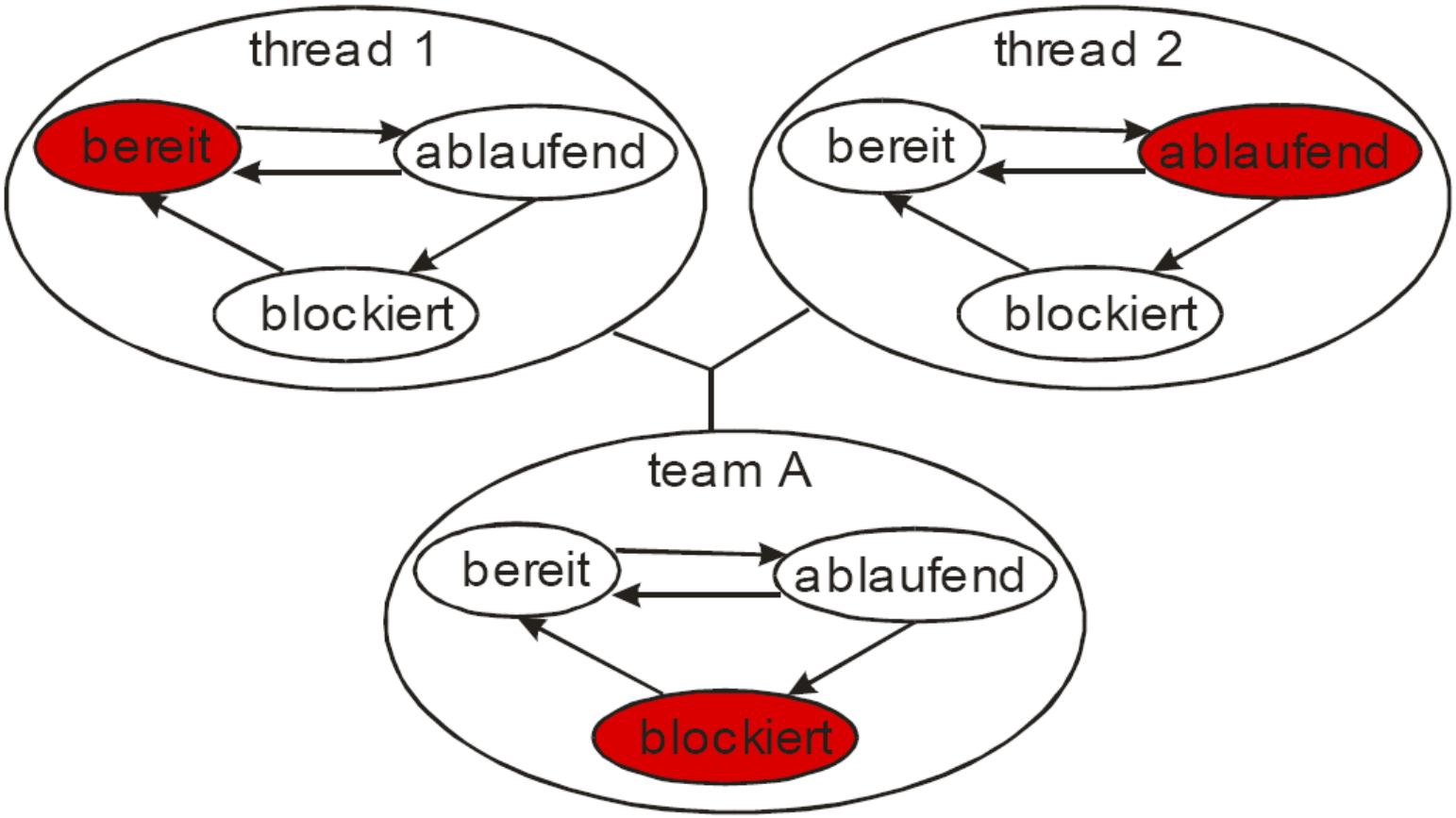
- Betriebssystem hat dem Team A den Prozessor zugeteilt
- Thread-Bibliothek führt Thread 1 im Zustand *bereit* und Thread 2 im Zustand *ablaufend*



Benutzerthreads (User-Level-Threads)

Annahmen: Thread 2 fordert Dienst vom Betriebssystem zur Durchführung einer Ein- Ausgabe

- Kern blockiert Team A
- Bibliotheksfunktionen führen Thread 2 als *ablaufend*

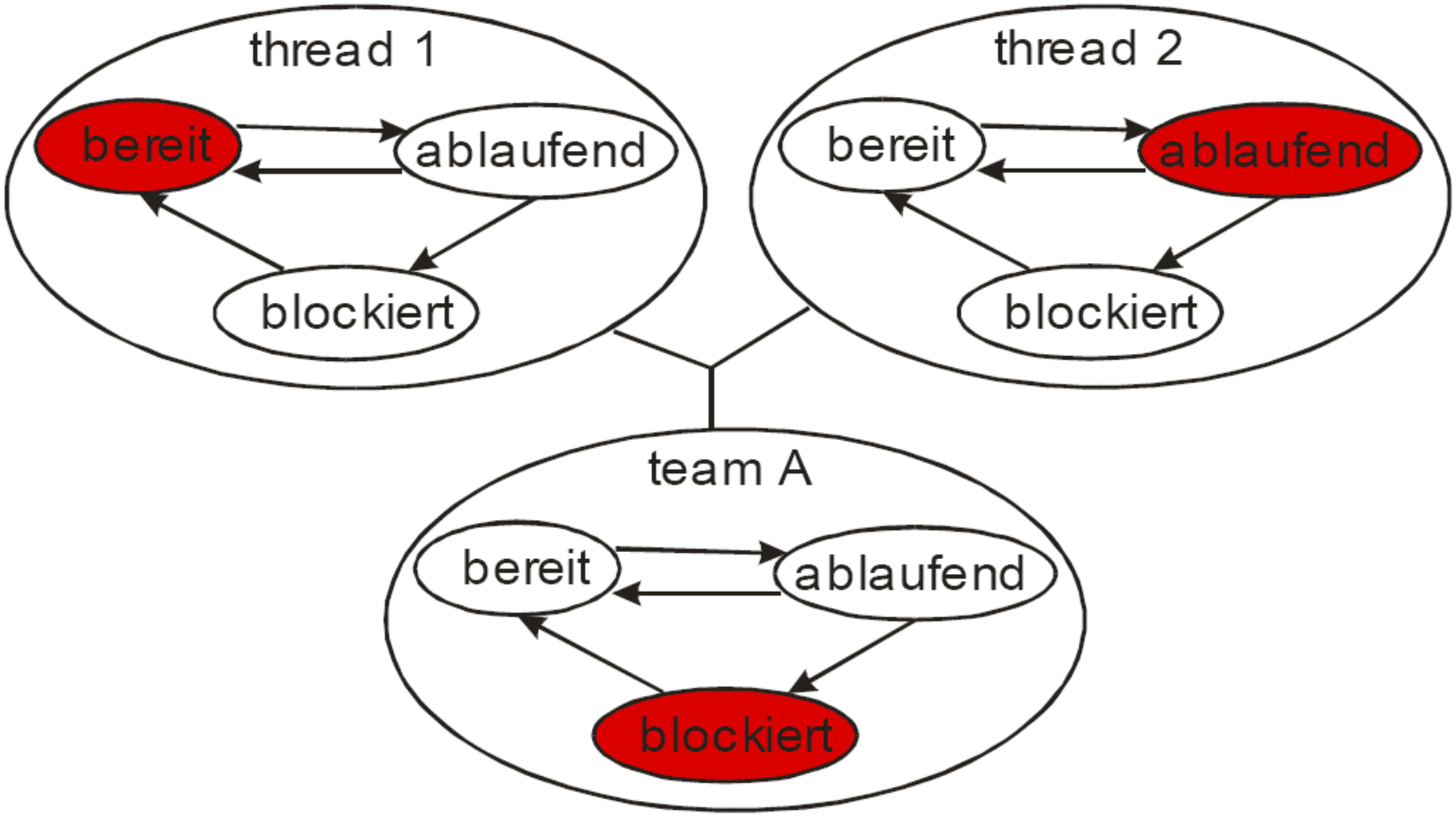


Vorteile von Benutzerthreads

- Reduktion des Overheads für Threadwechsel
- Scheduling der Threads kann anwendungsspezifisch vorgenommen werden
- Unabhängigkeit vom Betriebssystem

Problem bei Benutzerthreads

Kern kann den Prozessor in der gezeigten Situation nicht an Thread 1 vergeben.



Nachteile von Benutzerthreads

- Da viele Dienstanforderungen an den Kern blockierend sind, werden häufig alle Threads eines Teams gemeinsam blockiert

Nachteile von Benutzerthreads

- Da viele Dienstanforderungen an den Kern blockierend sind, werden häufig alle Threads eines Teams gemeinsam blockiert
- In einer Multiprozessoranwendung können die Benutzerthreads nicht verschiedenen Prozessoren zugeordnet werden.

Nachteile von Benutzerthreads

- Da viele Dienstanforderungen an den Kern blockierend sind, werden häufig alle Threads eines Teams gemeinsam blockiert
- In einer Multiprozessoranwendung können die Benutzerthreads nicht verschiedenen Prozessoren zugeordnet werden.
- Gegen die Intention: Anwendungsprogrammierer sollen sich nicht um Threadverwaltung kümmern müssen.

Nachteile von Benutzerthreads

- Da viele Dienstanforderungen an den Kern blockierend sind, werden häufig alle Threads eines Teams gemeinsam blockiert
- In einer Multiprozessoranwendung können die Benutzerthreads nicht verschiedenen Prozessoren zugeordnet werden.
- Gegen die Intention: Anwendungsprogrammierer sollen sich nicht um Threadverwaltung kümmern müssen.

Die meisten Betriebssysteme (z.B. Windows, Solaris, Linux, OS/2), bieten alle beschriebenen Möglichkeiten der Thread-Realisierung.

Varianten von Echtzeitbetriebssystemen

Merkmale	Interruptsteuerung	Zeitscheiben-steuerung	Einfacher Scheduler	Prioritätssteuerung
Betriebssystemkern	nein	nein	zum Teil	ja
Unterscheidung Rechenprozesse und Interrupts	nein	nein	ja	ja
Verarbeitungs-geschwindigkeit	maximal	schnell	mittel	designabhängig
Mittlerer Speicher-bedarf	$<$ 100 Byte	$>$ 500 Byte	$<$ 5 KByte	$<$ 50 KByte
Programmkontext	Interrupt Service Routine (ISR)	ISR und IDLE Loop	ISR und Prozes-skontext	ISR und Prozes-skontext
Kontextwechsel (Mehraufwand)	nicht vorhanden	wenig Mehrauf-wand Prozessorre-gistersicherung	erhöhter Mehrauf-wand Rechenpro-zesswechsel und Interruptwechsel; Prozessorregister-variablen zu sichern	hoher Mehrauf-wand Rechenpro-zesswechsel und Interruptwechsel; Prozessorregister-variablen zu sichern
Portabilität (Pro-zessorwechsel)	nicht möglich	bedingt möglich	mit Anpassung möglich	sehr einfach mög-lich

Quelle: Michael P. Witzak: Echtzeitbetriebssysteme. 2000. Franzis Verlag. Poing.

Lerneinheit 2. Laufzeitmodelle

Laufzeitmodelle

- beziehen sich auf Threads und Tasks
- basieren auf Prozesszuständen
- definieren Prozesskontrollmechanismen als Zustandsübergänge
- Threads können zu Teams zusammengefasst werden
- Es gibt Kernel-Level-Threads und Benutzerthreads (User-Level-Threads)