Betriebssysteme

Robert Kaiser

(HTTP: http://www.cs.hs-rm.de/~kaiser
EMail: robert.kaiser@hs-rm.de)

Wintersemester 2020/2021

https://www.animierte-gifs.net/data/media/1798/animiertes-richter-bild-0001.gif

Prozesse und Threads

Hochschule **RheinMain**

Prozessmodell

Hochschule RheinMain

Definition: Prozess (engl. *process*)

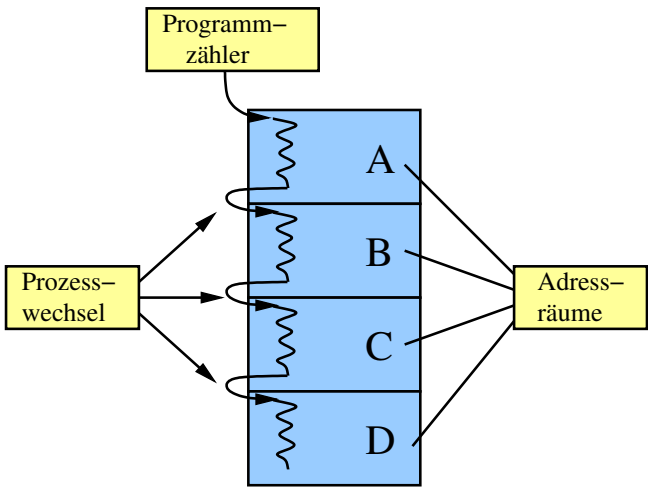
Ein **Prozess** ist ein sich in Ausführung befindliches Programm (einschließlich seiner aktuellen Werte des Programmzählers, der Register, Speichervariablen, Stack).

- Ein Prozess besitzt einen privaten Adressraum
  - Menge von (virtuellen) Adressen, von Prozess zugreifbar
  - Programm und Daten in Adressraum sichtbar.
- Verhältnis Prozess - Prozessor
  - Prozess besitzt konzeptionell einen eigenen virtuellen Prozessor
  - Reale(r) Prozessor(en) wird / werden zwischen den virtuellen Prozessoren umgeschaltet (Mehrprogrammbetrieb).
  - Die Umschaltungseinheit heißt **Scheduler** oder **Dispatcher**, der **Scheduling-Algorithmus** legt die Regeln der Umschaltung fest.
  - Der Umschaltvorgang heißt **Prozesswechsel** oder **Kontextwechsel** (*Context Switch*).
  - Multicore-Prozessoren enthalten mehrere Prozessoren auf einem Chip

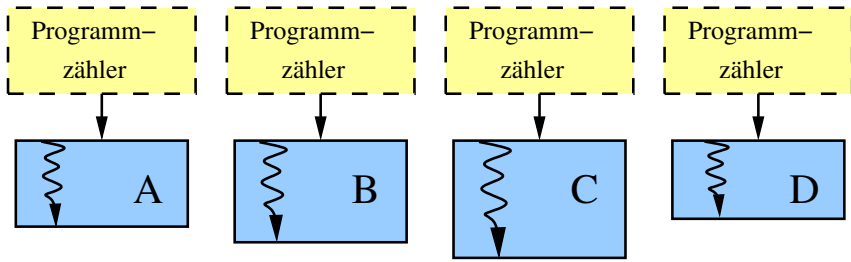
Zur Verdeutlichung

Hochschule RheinMain

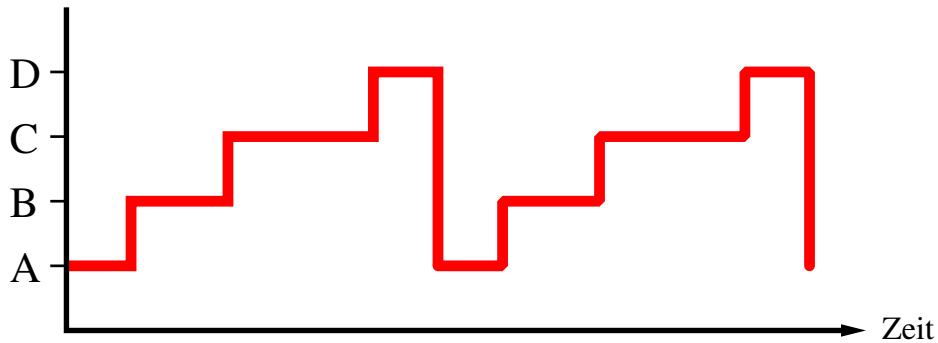
(a) Mehrprogrammbetrieb mit 4 Programmen



(b) Konzeptionelles Modell: 4 unabhängige, sequenzielle Prozesse



(c) Gantt-Diagramm: (1 Prozessor)



zu jedem Zeitpunkt nur 1 Programm aktiv

Programm ≠ Prozess!

Hochschule RheinMain

Programm:

- **feststehende** Beschreibung eines sequenziellen Algorithmus
  (→ „Rezept“)

Windbeutel aus Brandteig

ΦArbeitszeit ca. 20 Minuten ΦRuhezeit ca. 10 Minuten ΦKoch-/Backzeit ca. 20 Minuten
ΦGesamtzeit ca. 50 Minuten

Zubereitung

Mit einem Spritzbeutel mit großer Sterntülle 14 Teighäufchen auf das Backblech setzen und im vorgeheizten Backofen bei 200°C Ober-/Unterhitze ca. 20 Minuten backen. Während der Backzeit die Backofentür nicht öffnen, da das Gebäck sonst zusammenfällt.

Sofort nach dem Backen von jedem Windbeutel einen Deckel abschneiden und das Gebäck auf einem Kuchenrost erkalten lassen. Die Sahne mit Sahnesteif, Puderzucker und Vanillezucker schlagen und damit die kalten Windbeutel füllen.

Prozess:

- **Ausführung** der Beschreibung
  (→ „Vorgang“)
- Ein und dasselbe Programm kann mehrmals (auch gleichzeitig!) innerhalb verschiedener Prozesse ausgeführt werden.

Beachte

- Ein Prozess besitzt einen privaten **Adressraum**,
- hat einen **Prozesszustand**, beschrieben durch
  - Programm
  - Daten
  - Momentane Arbeitsposition,
- kann i.d.R. jederzeit unterbrochen . . .

  → Prozesszustand speichern
- . . . und später fortgeführt werden

  → Prozesszustand wiederherstellen
- Die Ausführungsgeschwindigkeit eines Prozesses ist daher **nicht gleichmäßig** und **nicht reproduzierbar**.

  → Bei der Programmierung sind keine a-priori-Annahmen über den zeitlichen Verlauf zulässig.

  → Bei zeitkritischen Anforderungen, z.B. Echtzeit-System, sind besondere Vorkehrungen im Scheduling-Algorithmus notwendig.

Prozesserzeugung

Hochschule RheinMain

Einfachster Fall: Feste Menge von Prozessen wird beim Systemstart erzeugt

- Einfache, meist *eingebettete* Systeme
- Beispiele
  - Autoelektronik (Motorsteuerung, etc.):
    Betriebssystem „OSEK/VDX“[1]: bietet **keine** Funktionen zur dynamischen Prozesserzeugung, stattdessen offline-Konfiguration in spezieller Sprache („OIL“)
  - Videorekorder , settop-Box, Handy (einfache), Weltraumsonde ...
- Vorteil: geringer Verwaltungsaufwand, deterministisches Zeitverhalten
- Nachteil: unflexibel

[1] heute: „AUTOSAR Classic“

Prozesserzeugung (2)

Bei komplexeren Systemen werden neue Prozesse im Laufe der Zeit dynamisch erzeugt, z.B.

- Beim **Systemstart**
  - z.B. UNIX-„Daemons“: Hintergrundprozesse zum Annehmen von E-Mail, Druckjobs, Web-Anfragen, ...
- durch **andere Prozesse** per Systemfunktion (z.B. „Hilfsprozess“ erzeugen)
- durch **Benutzende** veranlasst
  - z.B. Programmstart: „Prozesserzeugung per Doppelklick“ oder zur Abarbeitung von **Batch-Jobs** (auf Großrechnern)
- Impliziert die Bereitstellung geeigneter Systemaufrufe durch das Betriebssystem

Prozessende

- **Freiwilliges Prozessende** (Prozess beendet *sich selbst*)
- Normale Beendigung
  - Prozess ist „normal“ durchgelaufen
  - z.B.: Aufruf von `exit(EXIT_SUCCESS)`
- Beendigung aufgrund eines Fehlers
  - z.B. angegebene Datei kann nicht geöffnet werden, Programm sieht Ausgabe einer Fehlermeldung und Prozessende vor
  - `exit(EXIT_FAILURE)`
- **Unfreiwilliges Prozessende** (Prozess *wird* beendet)
  - Beendigung aufgrund eines schweren Fehlers, z.B.
    - Zugriff auf unzulässige Speicheradresse
    - Division durch Null
- Beendigung durch anderen Prozess
  - ein anderer Prozess hat mit Hilfe einer Systemfunktion das Betriebssystem überzeugt, den Prozess abzubrechen.

Prozesshierarchie

- Manche Systeme merken sich Zusammenhang zwischen erzeugendem Prozess (Elternprozess) und von diesem erzeugtem Prozess (Kindprozess)
- **Prozessfamilie**: Prozess und alle seine Nachkommen
- Prozesshierarchie: Baum-strukturierte Prozess-Menge (z.B. UNIX)
- Gegenbeispiel Windows:
  - keine Hierarchie,
  - alle Prozesse sind gleichwertig,
  - erzeugender Prozess erhält Verweis („Handle“) auf erzeugten Prozess,
  - dieses Handle kann er jedoch beliebig weitergeben ($\rightarrow$ nicht notwendig Baumstruktur)

Beispiel: UNIX Systemstart

- Beim UNIX-Systemstart wird der Prozess `init` (Prozess Nr. 1) erzeugt (Parent aller nachfolg. Prozesse)
- `init` führt eine Reihe von Programmen (i.d.R. Shell-Scripts) aus dem Verzeichnis `/etc/init.d` aus.
- Dabei werden unter anderem auch Anmelde-Prozesse gestartet
- Melden sich Benutzende an, wird jeweils ein Shell-Prozess erzeugt, der wiederum bei Kommandoeingaben entsprechende Unterprozesse erzeugt usw.
- UNIX-Kommandos zur Ausgabe der Prozessliste:
  - `ps` Standard-Kommando
  - `pstree` baum-formatierte Ausgabe (nicht überall verfügbar)

Prozesszustände

- Prozesse, obwohl unabhängige Einheiten, können aufgrund des Algorithmus logisch voneinander abhängig sein.
- Beispiel (UNIX Shell):`cat datei1 datei2 datei3 | grep hugo`
- In Abhängigkeit von den relativen Ausführungsgeschwindigkeite kann ein Prozess warten müssen, bis eine Eingabe vorliegt.
- Allgemeiner sagt man: Er **blockiert** und wartet auf ein (für ihn externes) **Ereignis**.
- Prozessor wird dann unmittelbar einem anderen Prozess zugeordnet. Entzug des Prozessors (**Suspendierung**) in diesem Fall problembegründet.
- Auch möglich: Scheduler entscheidet auf Prozesswechsel, obwohl der erste Prozess weiter ausgeführt werden könnte (**Preemption**).

Prozesszustände (2)

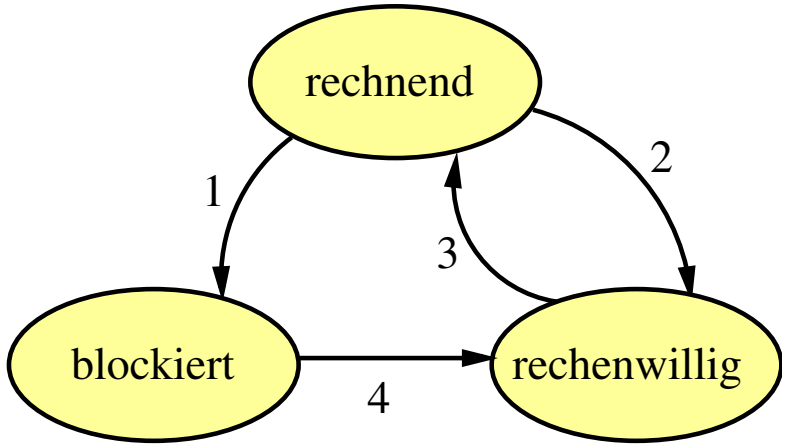
Damit sinnvolle Prozesszustände:

- **rechnend** (oder **aktiv**):dem Prozess ist ein Prozessor zugeordnet, der das Programm vorantreibt.
- **rechenwillig** (oder **bereit**):Prozess ist ausführbar, aber alle verfügbaren Prozessoren sind anderen Prozessen zugeordnet.
- **blockiert** (oder **schlafend**): Prozess wartet auf Ereignis. Er kann solange nicht ausgeführt werden, bis das Ereignis eintritt.

Gelegentlich noch folgende Zustände:

- **initiiert**: in Vorbereitung (Anfangszustand).
- **terminiert**: Prozess ist beendet (Endzustand).

Zustandsübergangsdiagramm

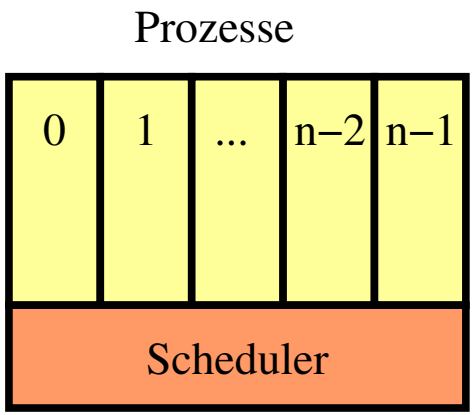


Zustandsübergänge:

1	rechnend → blockiert:	Versetzung in den Wartezustand (Suspension, warten auf Ereignis)
2a	rechnend → rechenwillig:	Scheduler entzieht Prozessor. (Preemption)
2b	rechnend → rechenwillig:	Prozess gibt Prozessor freiwillig ab. (Yield)
3	rechenwillig → rechnend:	Scheduler teilt Prozessor zu.
4	blockiert → rechenwillig:	Ereignis tritt ein.

Zusammenfassung

- Das Prozessmodell vereinfacht die Beschreibung der Aktivität des Rechensystems.
- Die ineinander verwobene Aktivität des Systems wird durch eine Menge von sequentiellen Prozessen beschrieben.
- Die unterste Schicht eines Betriebssystems behandelt die Unterbrechungen und ist für das Scheduling verantwortlich. Der Rest des Systems besteht aus sequentiellen Prozessen.



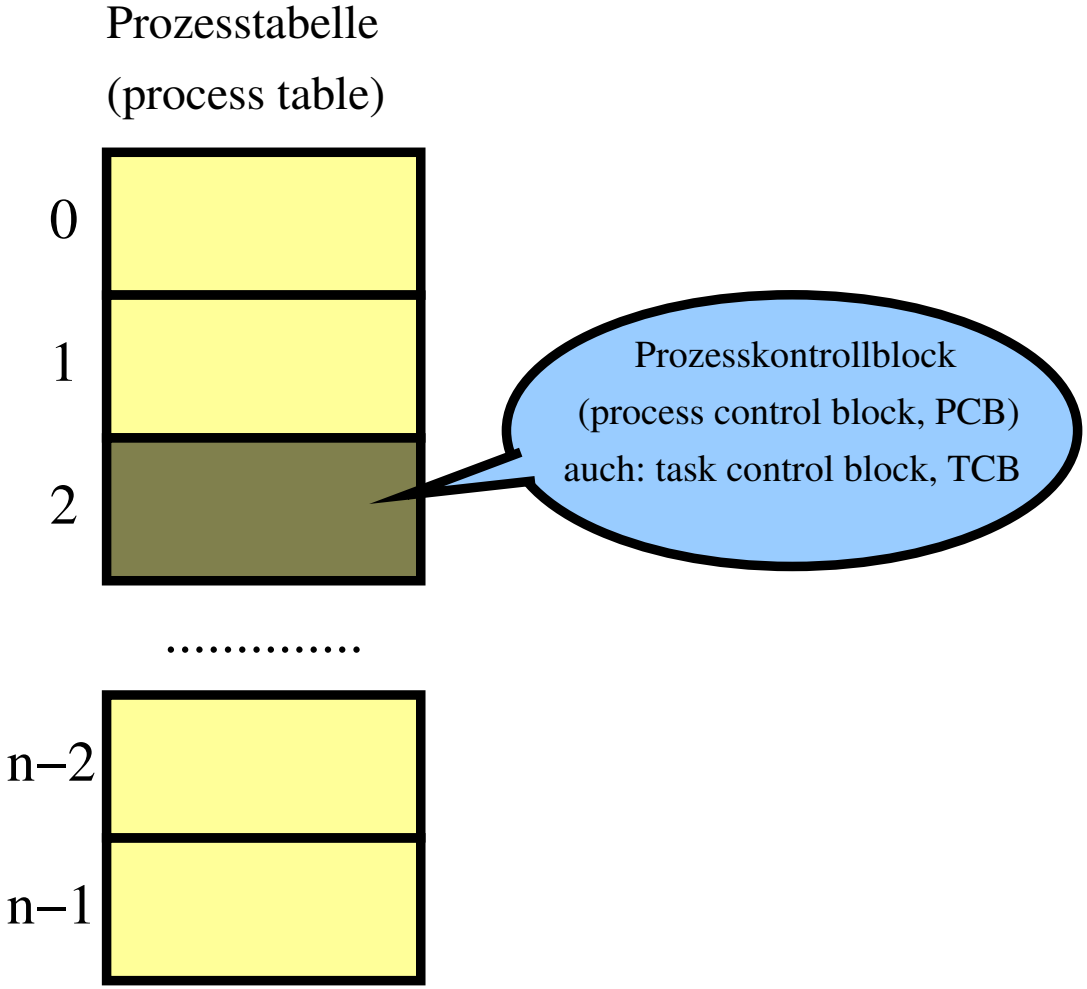
Anmerkungen

- Mechanismen zur Synchronisation und Kommunikation von Prozessen sind notwendig → Kap. 5, 6
- Programmierung von Anwendungen aus mehreren nebenläufigen Prozessen heißt Concurrent Programming.
- Das klassische Prozessmodell wird verfeinert durch Einführung sogenannter **Leichtgewichtsprozesse** (light weight processes, oder **Threads** = Fäden), die mehrere Aktivitätsträger in einem einzigen Adressraum darstellen → 3.3.

Implementierung von Prozessen

Hochschule RheinMain

Datenstrukturen im BS-Kern zur Prozessverwaltung:



PCB dient zum Speichern des gesamten Zustandes eines Prozesses

Prozesskontrollblock

Typische Inhalte eines Prozesskontrollblocks:

Prozessverwaltung
Register
Programmzähler
Programmstatuswort
Stack-Zeiger
Prozesszustand
Prozessnummer
Elternprozessnummer
Prozesserzeugungszeitpunkt
Terminierungsstatus
verbrauchte Prozessorzeit
Prozessorzeit der Kinder
Alarm-Zeitpunkt
Signalstatus
Signalmaske
unbearbeitete Signale
Zeiger auf Nachrichten
verschiedene Flags

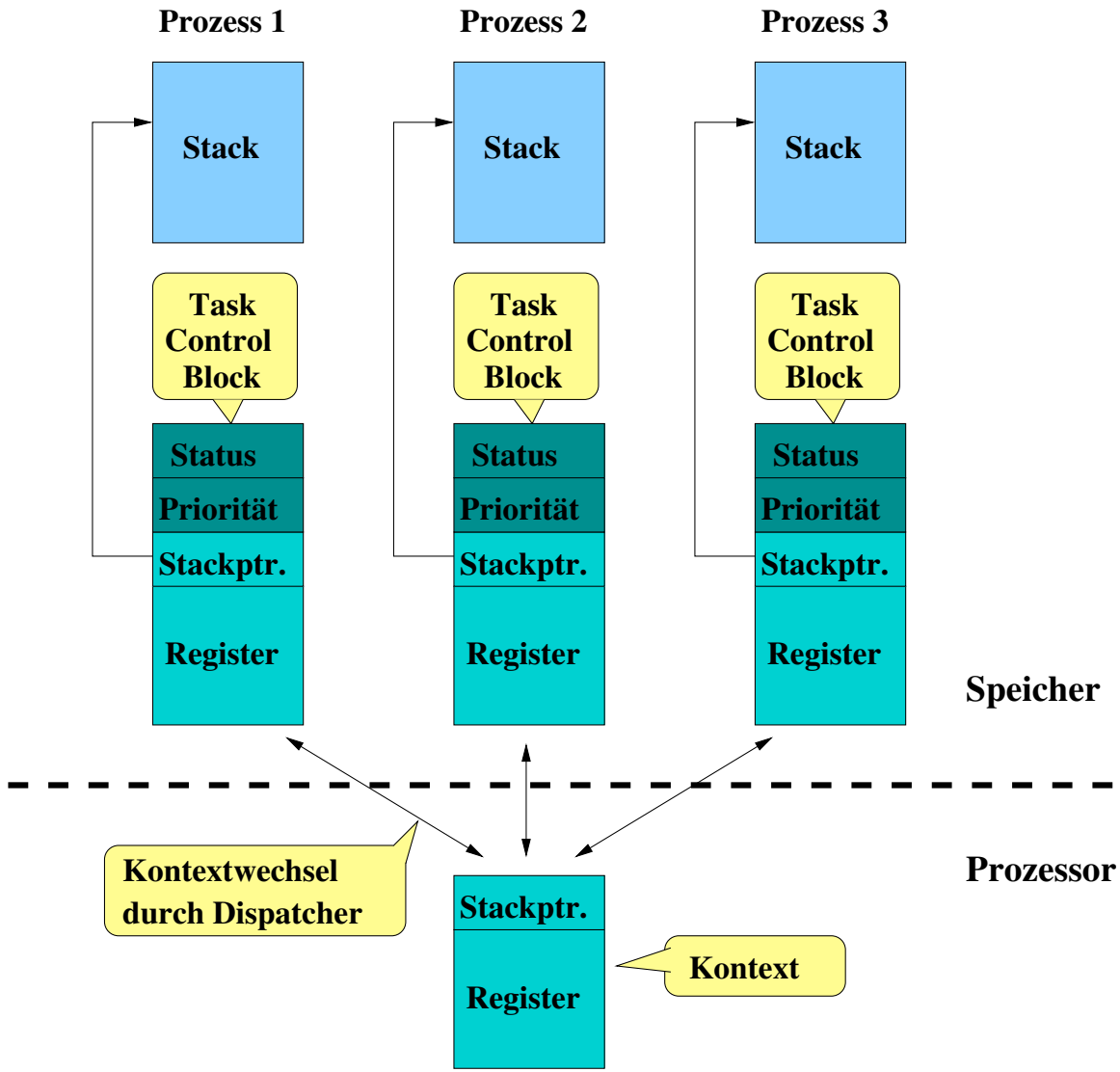
Speicherverwaltung
Zeiger auf Textsegment
Zeiger auf Datensegment
Zeiger auf BSS-Segment
Prozessgruppe
reale UID
effektive UID
reale GID
effektive GID
verschiedene Flags

Dateisystem
Wurzelverzeichnis
aktuelles Verzeichnis
UMASK-Maske
offene Dateideskriptoren
effektive UID
effektive GID
Aufrufparameter
verschiedene Flags

zusätzlich: Zeiger zur Verkettung des PCB in (verschiedenen) Warteschlangen

Multitasking

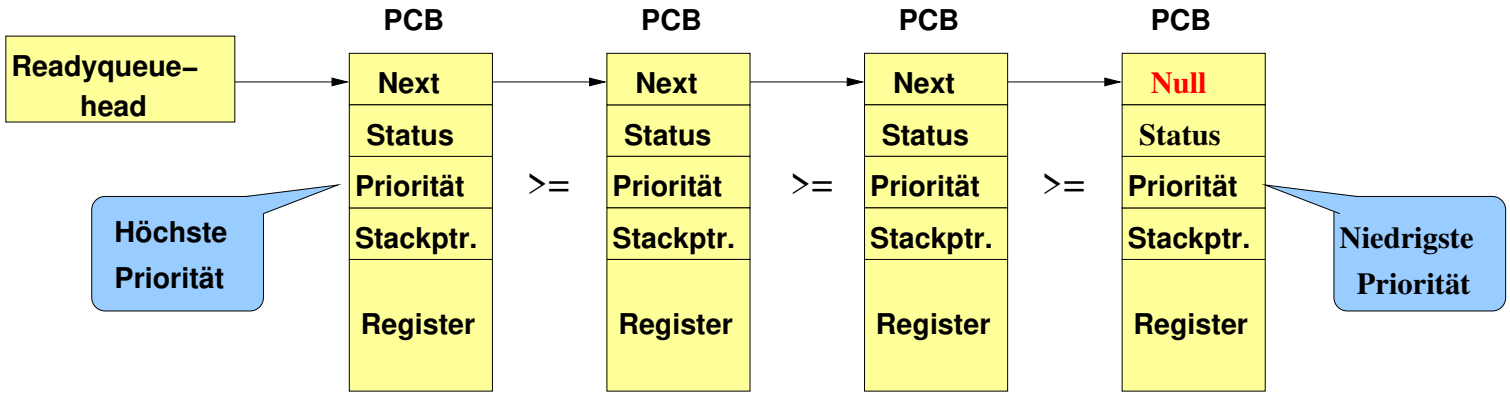
Hochschule RheinMain



Warteschlangenstruktur

Hochschule RheinMain

Einfache Struktur der Liste der rechenwilligen Prozesse (Bereit-Liste oder Ready Queue):

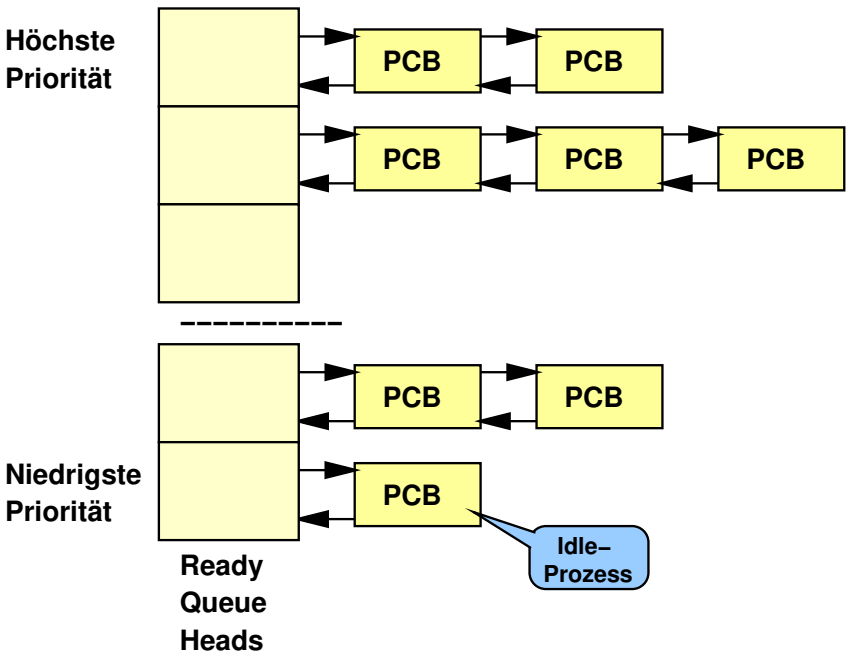


Bei gleicher Priorität: Einreihen nach „first in / first out“
Nachteil: Laufzeit abhängig von Prozessanzahl

Warteschlangenstruktur

Hochschule RheinMain

Typische Struktur der Liste der rechenwilligen Prozesse (Bereit-Liste oder Ready Queue):



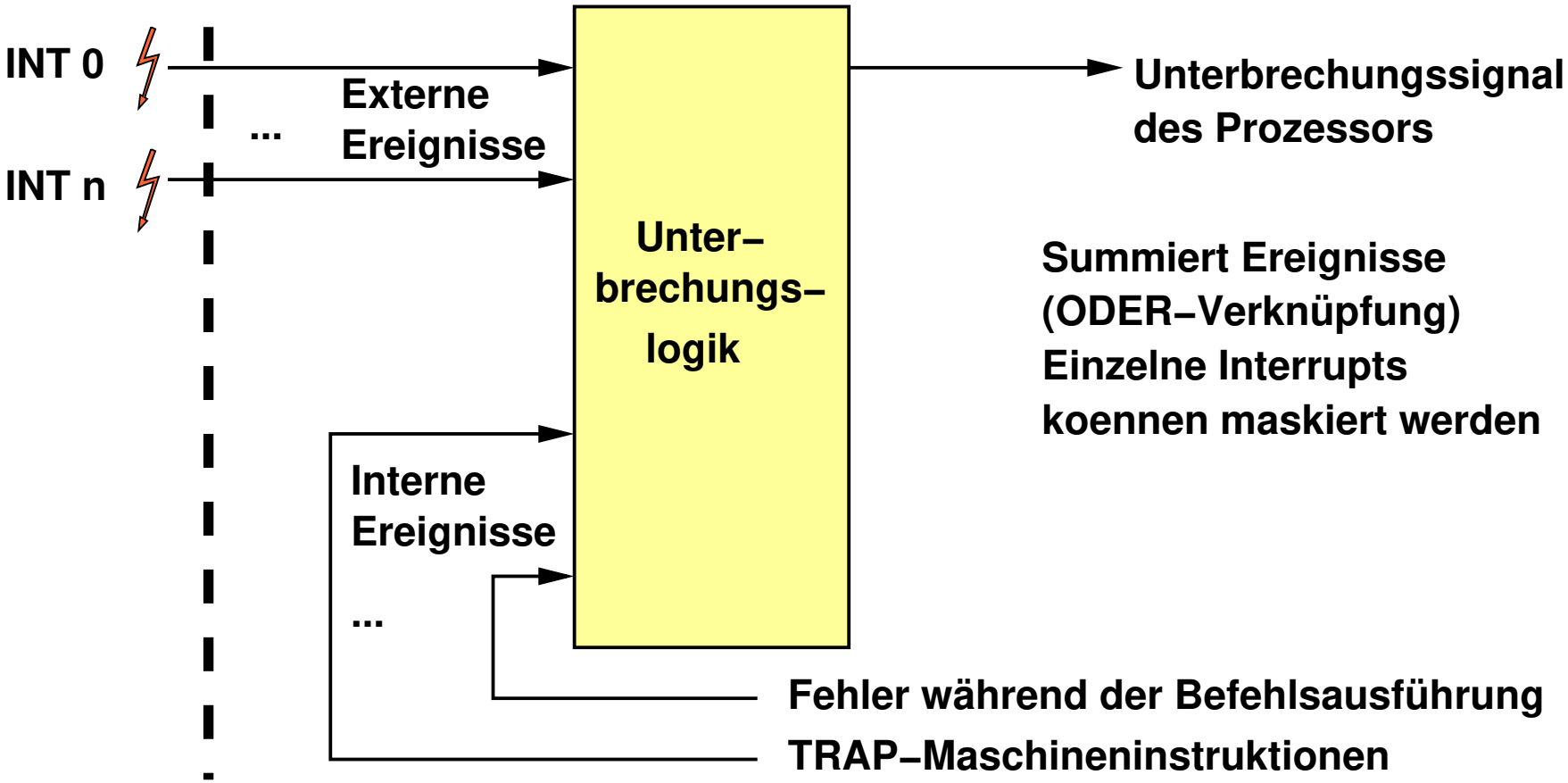
- Eine Warteschlange pro Prioritätsstufe
- Schnelles Einreihen in konstanter Laufzeit („O(1)-Scheduler“)
- (Nachteil: Begrenzte Anzahl möglicher Prioritäten)

Wer (oder was) aktiviert den Scheduler?

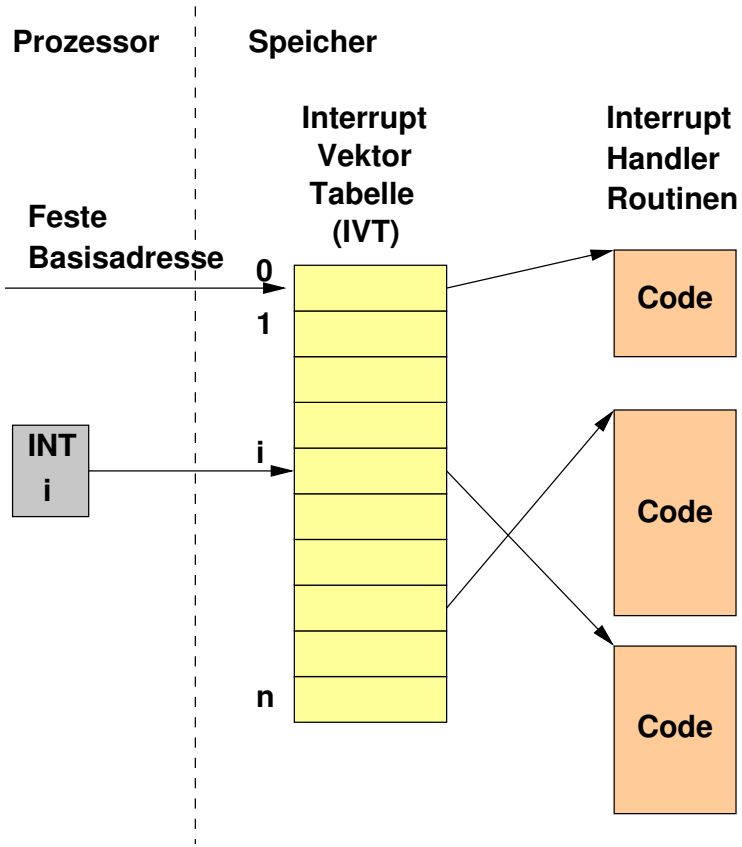
- Wenn der Scheduler die Kontrolle an einen ausgewählten Prozess abgibt - wie bekommt er sie dann wieder zurück?
- Ein Ansatz: Jedes Programm führt „oft genug“ einen Systemaufruf aus, das Betriebssystem ruft bei jedem Systemaufruf den Scheduler auf.
  - **„kooperatives Multitasking“**
  - z.B. in früheren Windows- und MacOS-Versionen
  - Nachteil: Ein Prozess kann nicht gezwungen werden, die Kontrolle abzugeben; Problem bei „bösen“ Programmen.
- Alternative: Preemptives Multitasking
  - benötigt Hardware-Unterstützung
  - Bei Eintreten bestimmter Ereignisse (Ein-/Ausgabe, Ablauf eines Timers, ...) wird der gerade laufende Prozess „von außen“ unterbrochen und Code zur Unterbrechungs-Verarbeitung aufgerufen
  - hierbei kann Aufruf des Schedulers vorgesehen werden

Unterbrechungslogik

Hochschule RheinMain



Unterbrechungsbehandlung



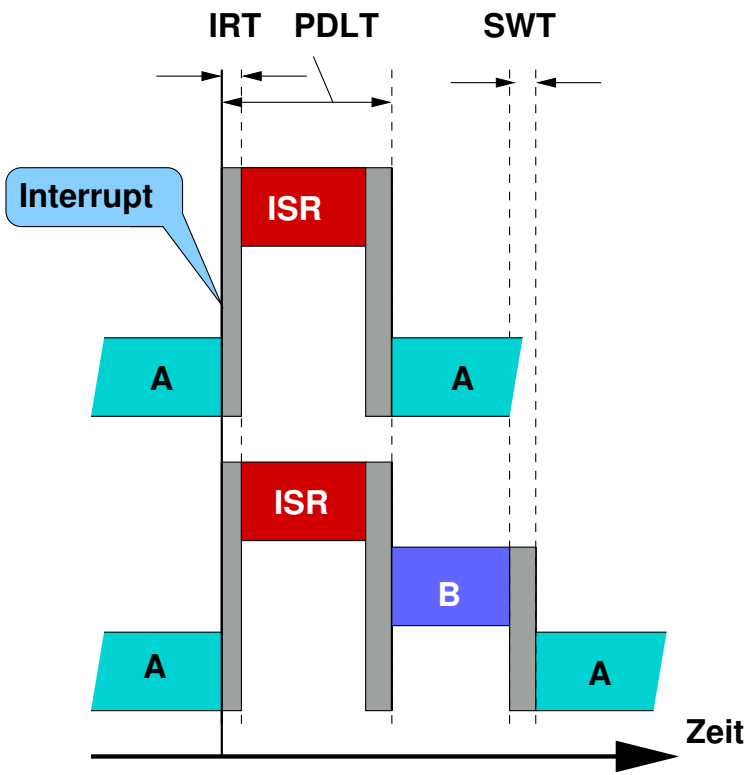
- Interrupt-Handler: Gerätespezifische Routinen zur Behandlung von Interrupts
- Zuordnung von Interrupt(-Nummer) zu Interrupt-Handler über *Interrupt-Vektor-Tabelle* (IVT)

Unterbrechungsbehandlung (2)

Typische Ausführungsschritte:

1. Programmzähler (u.a.) wird durch Hardware auf dem Stack abgelegt.
2. Hardware lädt den neuen Programmzählerinhalt aus dem Unterbrechungsvektor.
3. Eine Assembler-Routine rettet die Registerinhalte.
4. Eine Assembler-Routine bereitet den neuen Stack vor.
5. Eine C-Prozedur markiert den unterbrochenen Prozess als rechenwillig.
6. Der Scheduler bestimmt den Prozess, der als nächster ausgeführt werden soll.
7. Die C-Prozedur gibt die Kontrolle an die Assembler-Routine zurück.
8. Die Assembler-Routine startet den ausgewählten Prozess.

Interrupts aus Sicht des Prozesses



- Laufender Prozess (A) wird unterbrochen
- Interrupt Service Routine (ISR) wird ausgeführt
- Weiterführen, entweder Prozess (A), oder zuvor blockierter Prozess (B) (der durch ISR „rechenwillig“ wurde)
- Relevante Zeitspannen:

  IRT *Interrupt Response Time*
  PDLT *Process Dispatch Latency Time*
  SWT *Process Switch Time*

Beispiel UNIX

Systemaufrufe zur Prozessverwaltung

- `pid_t fork(void)`
  Erzeugen einer Kopie des Prozesses (Child), Parent erhält pid des Kindes zurück oder -1 bei Fehler, Kind erhält 0 als Ergebnis.
- `int execve(char* name, char* argv[], char* envp[])`
  Überlagern des ausgeführten Programms eines Prozesses (Code, Daten, Stack) durch neues Programm. Andere Varianten:`execl`, `execle`, `execlp`, `execv`, `execvp`
- `pid_t getpid(void)`
  Rückgabe der eigenen Prozess ID
- `pid_t getppid(void)`
  Rückgabe der Prozess ID des Elternprozesses.

Beispiel UNIX

Systemaufrufe zur Prozessverwaltung (Forts.)

- `exit(status)`
  Beendet den laufenden Prozess und stellt dem Parent den Exit-Status zur Verfügung.
- `pid_t wait(int* status)`
  Warten auf die Beendigung eines Kindprozesses. Dessen ID wird über den Rückgabewert, sein Status über `status` zurückgegeben.
- `pid_t waitpid(pid_t pid, int* status, int opts)`
  Warten auf das Ende eines bestimmten Kindprozesses, dessen ID über den Parameter pid übergeben wird.

(wird im Praktikum vertieft)

Threads

Hochschule RheinMain

Motivation

- Prozesserzeugung, Prozessumschaltung und Prozesskommunikation sind teuer
  - → rechenzeitaufwändig zur Laufzeit
  - Verluste auch durch Cache-Misses
- Wie nutzt man mehrere Prozessoren eines Multiprozessors für eine Applikation?
  - Z.B. mehrere kooperierende Prozesse auf verschiedenen Prozessoren
  - Muss auf Applikationsebene ausprogrammiert werden !
- Wie strukturiert man einen Server-Prozess, der Anforderungen von mehreren Klienten bedienen kann?
  - Ein Server-Prozess → keine Parallelität
  - Multiplexing für verschiedene Klienten von Hand
    → komplexe Programmierung

Lösung

- Einführung von billiger Nebenläufigkeit in einem Prozessadressraum durch „Leichtgewichtsprozesse“, sogenannte **Threads**.

Definition

Prozess (Einheit der Betriebsmittelverwaltung)

- ausführbares Programm, das Code und globale Daten definiert.
- privater Adressraum.
  Code und Daten über Adressraum zugreifbar.
- Menge von Betriebsmitteln
  - geöffnete Objekte, Betriebssystem-Objekte wie z.B. Timer, Signale, Semaphore
  - dem Prozess durch das Betriebssystem als Folge der Programmausführung zugeordnet.
- Menge an Threads

Threads (Aktivitätsträger)

- Idee einer „parallel ausgeführten Programmfunktion“
- Eigener Prozessor-Context (Registerinhalte usw.)
- Eigener Stack (i.d.R. zwei, getrennt für user und kernel mode)
- eigener kleiner privater Datenbereich (Thread Local Storage)
- Threads eines Prozesses nutzen gemeinsam Programm, Adressraum und alle Betriebsmittel.

Schnittstellen

- Vornehmlich: **POSIX pthreads** (s.u.)
- Daneben:
  - Java: Klasse Thread in java.lang
    - ★ Z.B. Implementierung der Schnittstelle `java.lang.Runnable` und Implementierung von Methode `run()`.
    - ★ Thread-Modell in JVM implementiert
  - C++: Boost Threads verbreitet
  - Windows:
    - ★ C-Schnittstelle für Windows API, u.a. `CreateThread(...)`
  - LinuxThreads (veraltet)
    - ★ `clone()` System Call
      → Erzeugung eines Prozesses mit Angabe detaillierter Flags, was gemeinsam mit erzeugendem Prozess genutzt werden soll
    - ★ Gehört nicht zum UNIX Standard
      → Programme nicht portierbar

POSIX Threads (pthreads)

POSIX: Portable Operating System Interface (for unIX)

- Familie internationaler Standards ISO/IEC 9945, ursprünglich spezifiziert durch IEEE Computer Society als IEEE 1003
- C-API-Spezifikationen → Kompatibilität auf Quellcode-Ebene (kein ABI)
- Umfassende Funktionalitäten[2], hier nur Thread Interface („pthreads“) betrachtet.
- Auf vielen Systemen verfügbar, insbesondere auch Multiprozessorsystemen, z.B. Linux, MacOS X, FreeBSD, Solaris
- Neben C / C++ auch für andere Prgrammiersprachen
- Teilweise Bestandteil der libc
- ca. 50 Funktionen

[2] POSIX Base Definitions, System Interfaces, and Commands and Utilities (which include POSIX.1, extensions for POSIX.1, Real-time Services, Threads Interface, Real-time Extensions, Security Interface, Network File Access and Network Process-to-Process Communications, User Portability Extensions, Corrections and Extensions, Protection and Control Utilities and Batch System Utilities.

Pthreads API

API-Aufrufe zum Thread-Management

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

- Erzeugen eines Threads

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

- Sich selbst beenden

```
void pthread_exit(void *retval);
```

- Thread-Identifier des aktuellen Threads ermitteln

```
pthread_t pthread_self(void);
```

- Warten auf Beendigung eines Threads

```
int pthread_join(pthread_t th, void **thread_return);
```

- Beenden eines anderen Threads

```
int pthread_cancel(pthread_t thread);
```

Beispiel: Thread Erzeugen/Löschen


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

void my_thread(int *param);

main(int argc, char *argv[])
{
  pthread_t thread;
  int arg = atoi(argv[1]);

  pthread_create(&thread,
       NULL,
       (void*)my_thread,
       (void*)&arg);
  .....

  pthread_join(thread, NULL);
  return 0;
}
```

```
void my_thread(int* pcount)
{
  int i;
  int count = *pcount;
  for(i = 0; i < count; i++)
    do_fun();
}
```

Beispiel: Thread Erzeugen/Löschen

Hochschule RheinMain

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

void my_thread(int *param);

main(int argc, char *argv[])
{
  pthread_t thread;
  int arg = atoi(argv[1]);

  pthread_create(&thread,
        NULL,
        (void*)my_thread,
        (void*)&arg);
  .....

  pthread_join(thread, NULL);
  return 0;
}
```

Thread erzeugen und starten

```
void my_thread(int* pcount)
{
  int i;
  int count = *pcount;
  for(i = 0; i < count; i++)
    do_fun();
}
```

Beispiel: Thread Erzeugen/Löschen


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

void my_thread(int *param);

main(int argc, char *argv[])
{
  pthread_t thread;
  int arg = atoi(argv[1]);

  pthread_create(&thread,
        NULL,
        (void*)my_thread,
        (void*)&arg);
  .....

  pthread_join(thread, NULL);
  return 0;
}
```

Thread erzeugen und starten

Attribute, default falls NULL

```
void my_thread(int* pcount)
{
  int i;
  int count = *pcount;
  for(i = 0; i < count; i++)
    do_fun();
}
```

Beispiel: Thread Erzeugen/Löschen


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

void my_thread(int *param);

main(int argc, char *argv[])
{
  pthread_t thread;
  int arg = atoi(argv[1]);

  pthread_create(&thread,
        NULL,
        (void*)my_thread,
        (void*)&arg);

  .....

  pthread_join(thread, NULL);
  return 0;
}

void my_thread(int* pcount)
{
  int i;
  int count = *pcount;
  for(i = 0; i < count; i++)
    do_fun();
}
```

Thread erzeugen und starten

Attribute, default falls NULL

Zeiger auf Thread-Code

Beispiel: Thread Erzeugen/Löschen


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

void my_thread(int *param);

main(int argc, char *argv[])
{
  pthread_t thread;
  int arg = atoi(argv[1]);

  pthread_create(&thread,
        NULL,
        (void*)my_thread,
        (void*)&arg);

  .....

  pthread_join(thread, NULL);
  return 0;
}
```

```
void my_thread(int* pcount)
{
  int i;
  int count = *pcount;
  for(i = 0; i < count; i++)
    do_fun();
}
```

Thread erzeugen und starten

Attribute, default falls NULL

Zeiger auf Thread-Code

Argument (beliebiger Zeiger)

Beispiel: Thread Erzeugen/Löschen

Hochschule RheinMain

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

void my_thread(int *param);

main(int argc, char *argv[])
{
  pthread_t thread;
  int arg = atoi(argv[1]);

  pthread_create(&thread,
        NULL,
        (void*)my_thread,
        (void*)&arg);
  .....

  pthread_join(thread, NULL);
  return 0;
}

void my_thread(int* pcount)
{
  int i;
  int count = *pcount;
  for(i = 0; i < count; i++)
    do_fun();
}
```

Thread erzeugen und starten

Attribute, default falls NULL

Zeiger auf Argument (beliebiger Zeiger)

Thread-Code

Warten auf Ende des Thread

Beispiel: Thread Erzeugen/Löschen

Hochschule RheinMain

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

void my_thread(int *param);

main(int argc, char *argv[])
{
  pthread_t thread;
  int arg = atoi(argv[1]);

  pthread_create(&thread,
        NULL,
        (void*)my_thread,
        (void*)&arg);
  .....

  pthread_join(thread, NULL);
  return 0;
}

void my_thread(int* pcount)
{
  int i;
  int count = *pcount;
  for(i= 0; i < count; i++)
    do_fun();
}
```

Thread erzeugen und starten

Attribute, default falls NULL

Zeiger auf Argument (beliebiger Zeiger)

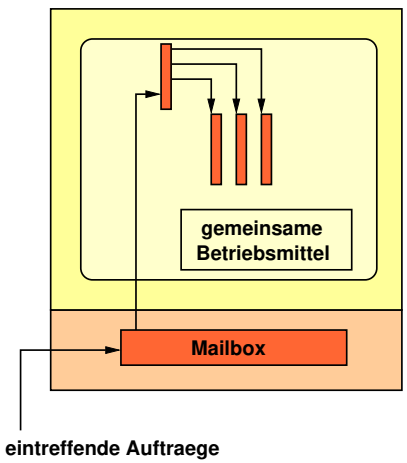
Zeiger auf Speicher für Returnwert (NULL falls nicht erwünscht)

Warten auf Ende des Thread

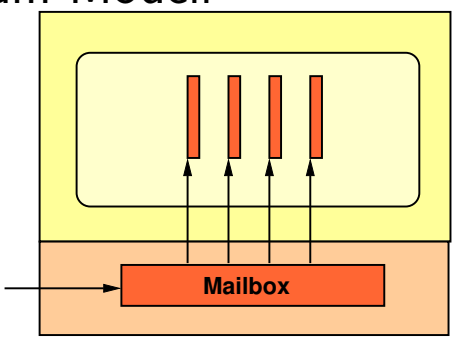
Kooperationsformen

- Bei **Einprozessorsystemen** liefern Threads **keinen** Performancegewinn
- Dennoch ist die Verwendung von Threads zur Organisation nebenläufiger Prozesse sinnvoll
- Beispiel: Server

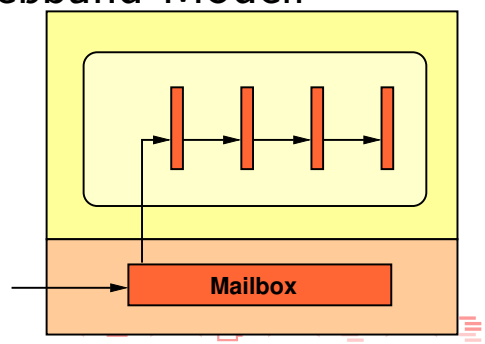
(a) Verteiler/Arbeiter-Modell



(b) Team-Modell



(b) Fließband-Modell

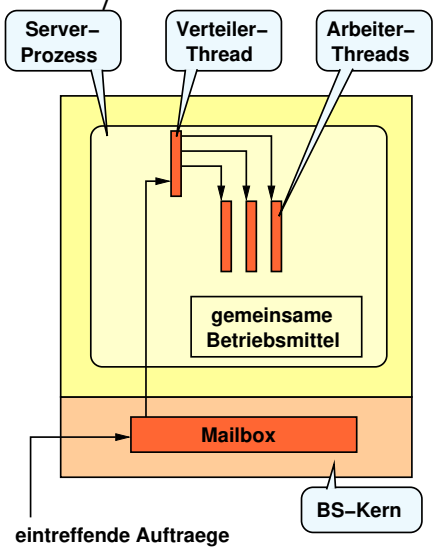


aus Tanenbaum: Moderne Betriebssysteme

Kooperationsformen

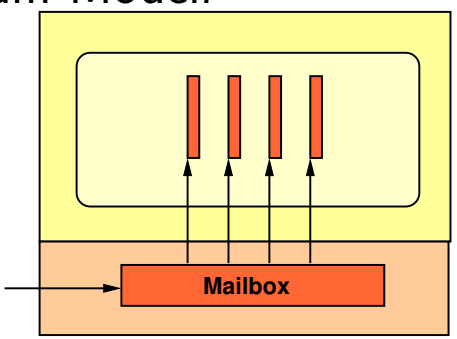
- Bei **Einprozessorsystemen** liefern Threads **keinen** Performancegewinn
- Dennoch ist die Verwendung von Threads zur Organisation nebenläufiger Prozesse sinnvoll
- Beispiel: Server

(a) Verteiler/Arbeiter-Modell

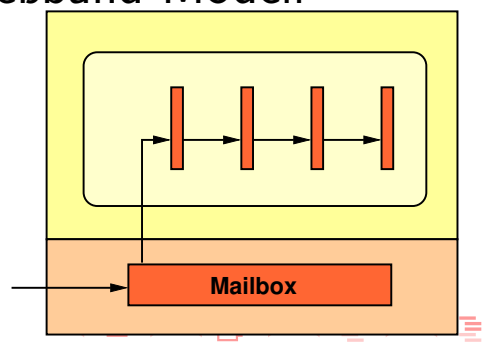


aus Tanenbaum: Moderne Betriebssysteme

(b) Team-Modell



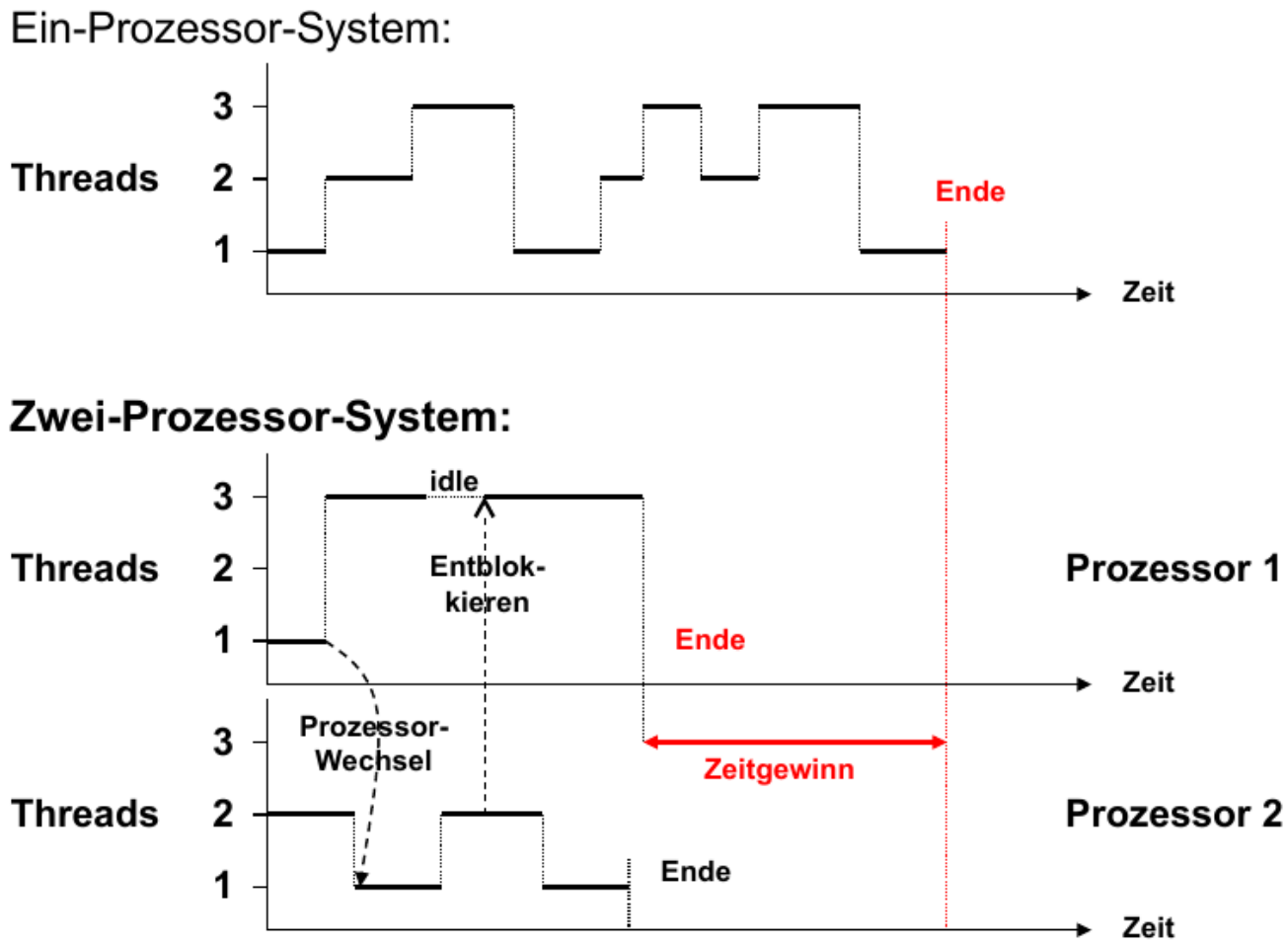
(b) Fließband-Modell



Kooperationsformen

Hochschule RheinMain

- Bei **Mehrprozessorsystemen** können Prozesse **nur** mithilfe von Threads und durch nebenläufige Implementierung Performancegewinne erzielen

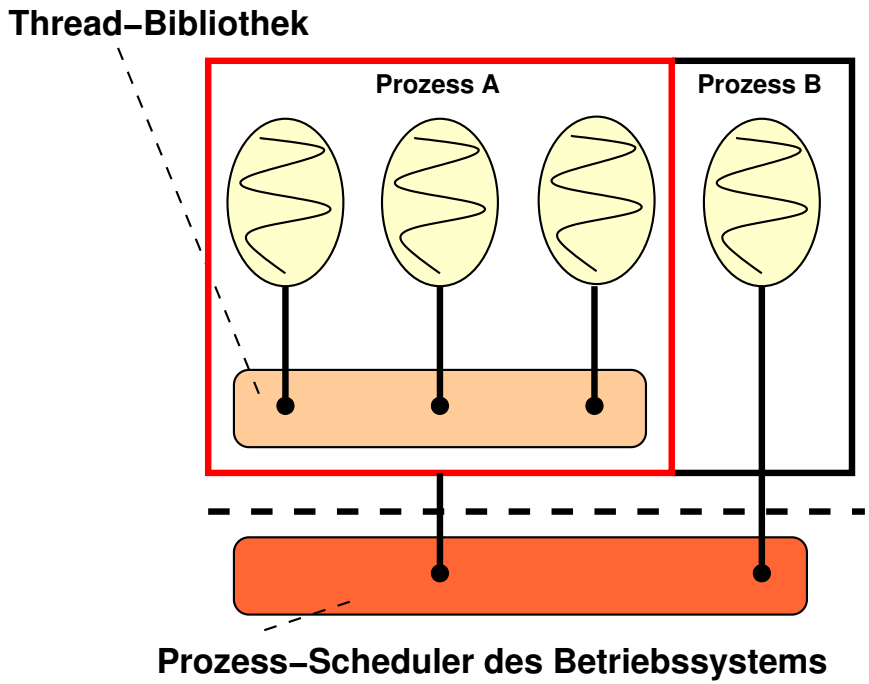


Implementierung von Threads

Hochschule RheinMain

Thread-Bibliothek (*User level threads*)

- Thread-Funktionen / Kontextwechsel in Bibliothek auf Applikationsebene implementiert
- Betriebssystem kennt nach wie vor nur übliche Prozesse
- 1:n-Zuordnung: BS „sieht“ nur ganzen Prozess
  - ++ leicht / nachträglich zu implementieren („*retrofit*“)
  - - - Keine Nutzung von Mehrprozessor-Architekturen



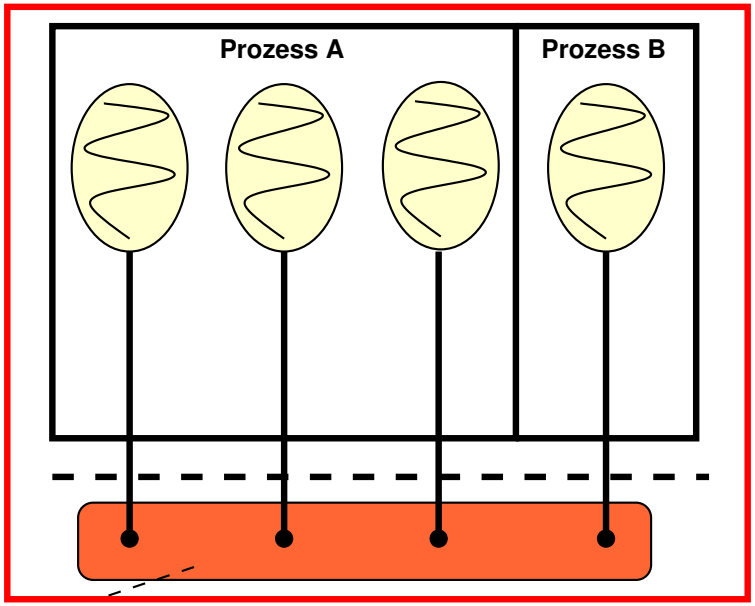
- Beispiel: pthreads Implementierung in OSF/DCE
- POSIX: „*process scope*“

Implementierung von Threads (2)

Hochschule RheinMain

Im BS-Kern (*Kernel level threads*)

- Betriebssystem unterstützt Threads
- Threads sind die Einheiten, denen Prozessoren zugeordnet werden („*schedulable entities*“).
- 1:1-Zuordnung (allgem.: m:n)
  - ++ Nutzbarkeit von Mehrprozessor-Architekturen
  - (- -) Kernel Unterstützung erforderlich



- Beispiele: pthreads Implementierung in Linux, , FreeBSD, Solaris, ...
- POSIX: „*system scope*“

Beispiel: Linux (ab Kernel 2.6)

Native POSIX Thread Library (NPTL)

- Federführung Red Hat
- Ziele:
  - Konformität zu POSIX Pthreads
  - Gute Multiprozessor-Performance
  - Niedrige Erzeugungskosten
  - Kompatibilität zu LinuxThreads
- 1:1-Implementierung
  - Kernel verwaltet Prozesse
  - **pthread_create()** führt zu neuem Prozess unter Nutzung von **clone()**
  - Spezielle Kernel-Unterstützung und viel Optimierung im Kernel (z.B. sog. Futexes)

Zusammenfassung

Was haben wir in Kap. 3 gemacht?

- Konzept des sequentiellen Prozesses (Wichtig!).
- Strukturierung von Aktivität durch eine Menge von sequentiellen Prozessen, die zueinander nebenläufig ausgeführt werden.
- Betriebssystem bietet Anwendungsprogrammierern ein solches Prozesskonzept an der Dienstschnittstelle zur Strukturierung von Anwendungen.
- Das Betriebssystem kann Prozesse auch intern zur Strukturierung höherer Funktionalität nutzen.
- Ansätze besprochen, wie Betriebssystem das Prozesskonzept implementiert (prinzipiell und speziell am Beispiel UNIX).
- Thread-Konzept als performante „Leichtgewichtsprozesse“ vorgestellt.