Kapitel 2
Prozesse

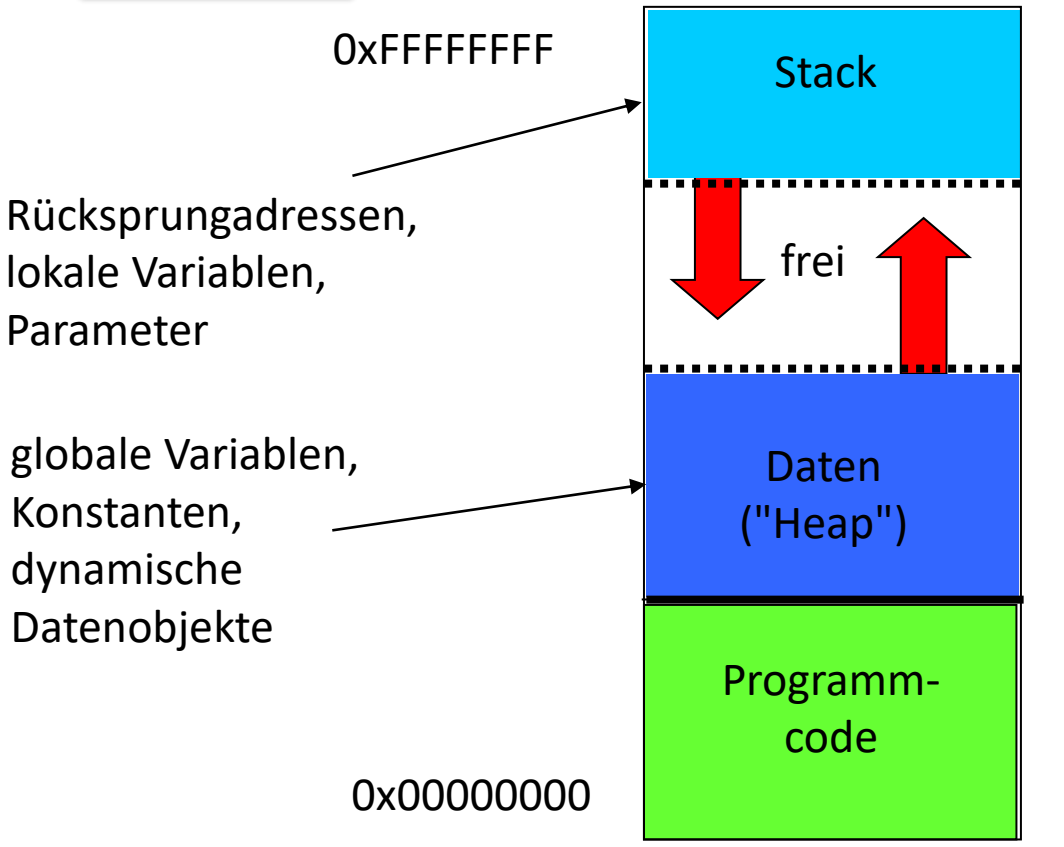
1. *Das Prozessmodell*
2. Das Threadmodell
3. Prozess-Scheduling

Was ist ein Prozess? - Beschreibungsversuche

- Ein Programm in Ausführung
- Eine Instanz eines Programms, das auf dem Computer gerade läuft
- Eine Einheit, der man den Prozessor zuweisen kann, und die vom Prozessor ausgeführt werden kann
- Eine Aktivitätseinheit, die beschrieben werden kann durch:
  - einen Ausführungsfaden *(→ Befehlszähler!)*
  - Zustand *(→ Register, Systemtabellen, …)*
  - zugewiesene Betriebsmittel *(→ CPU, Speicher, Dateien, …)*

Typisches Prozesslayout im Hauptspeicher

Besitzer:
Prozess X

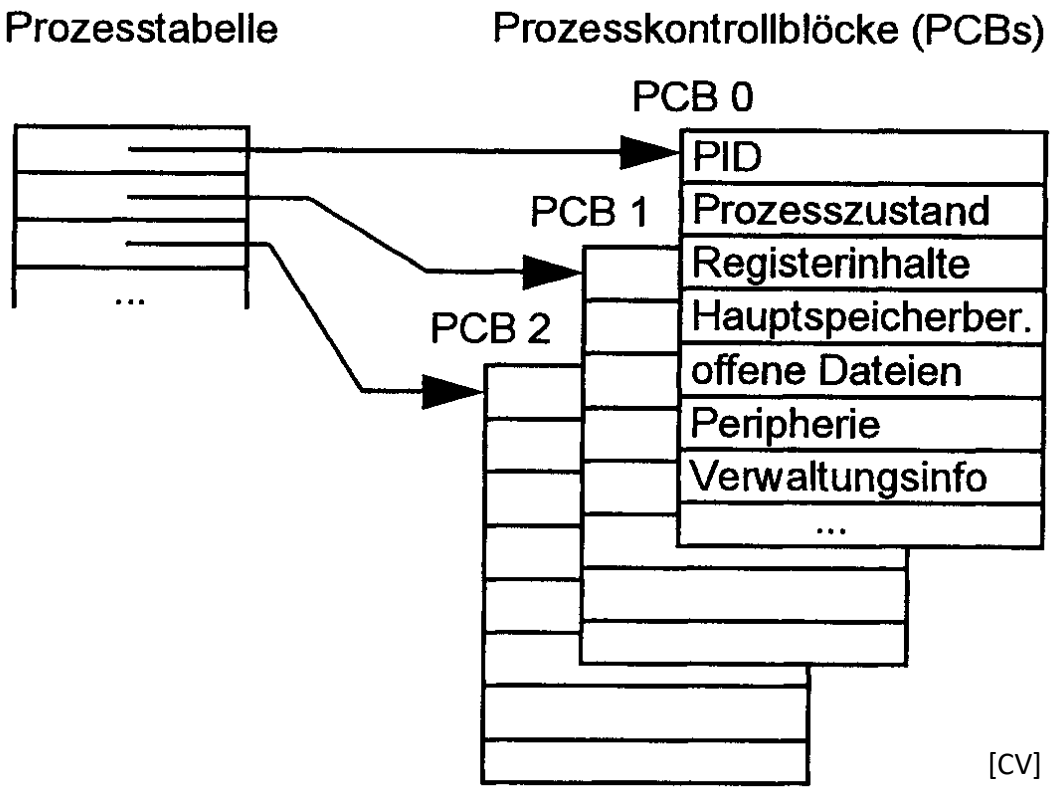


Besitzer:
Betriebssystem

Prozesstabelle
(Zustands-
informationen über
Prozess X)

Implementierung von Prozessen: Prozesstabelle / Prozesskontrollblock (PCB)

- **Prozesstabelle**: für jeden Prozess Verweis auf Datenstruktur zur Speicherung des speziellen Ausführungskontexts: **„Prozesskontrollblock“ (PCB)**



[CV]

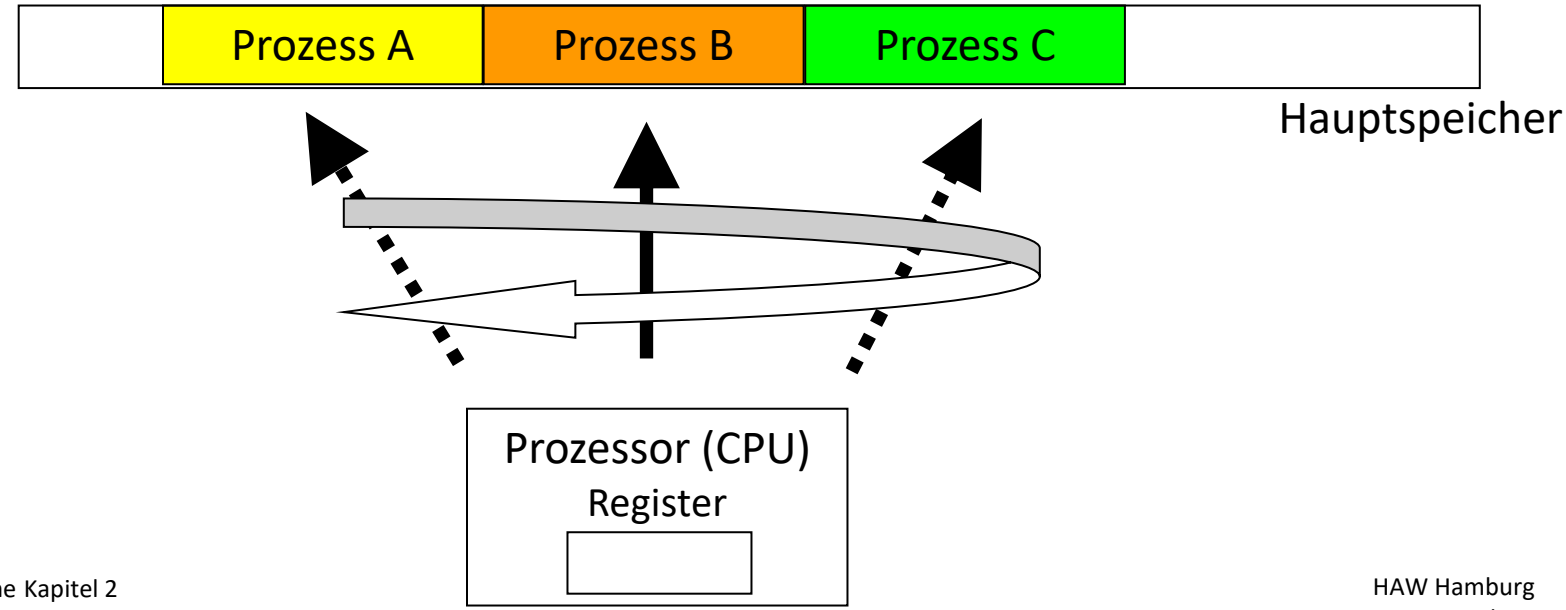
Typische Prozesskontrollblock-Felder (Unix)

Prozessverwaltung	Speicherverwaltung	Dateiverwaltung
Allgemeine Register	Zeiger auf Textsegment	Wurzelverzeichnis
Befehlszähler	Zeiger auf Datensegment	Arbeitsverzeichnis
Programmstatuswort	Zeiger auf Stacksegment	Dateideskriptor
Stackregister		Benutzer-ID
Prozesszustand		Gruppen-ID
Priorität		...
Scheduling-Parameter		
Prozess-ID		
Elternprozess		
Prozessgruppe		
Signale		
Startzeit des Prozesses		
Benutzte CPU-Zeit		
CPU-Zeit der Kindprozesse		
...		

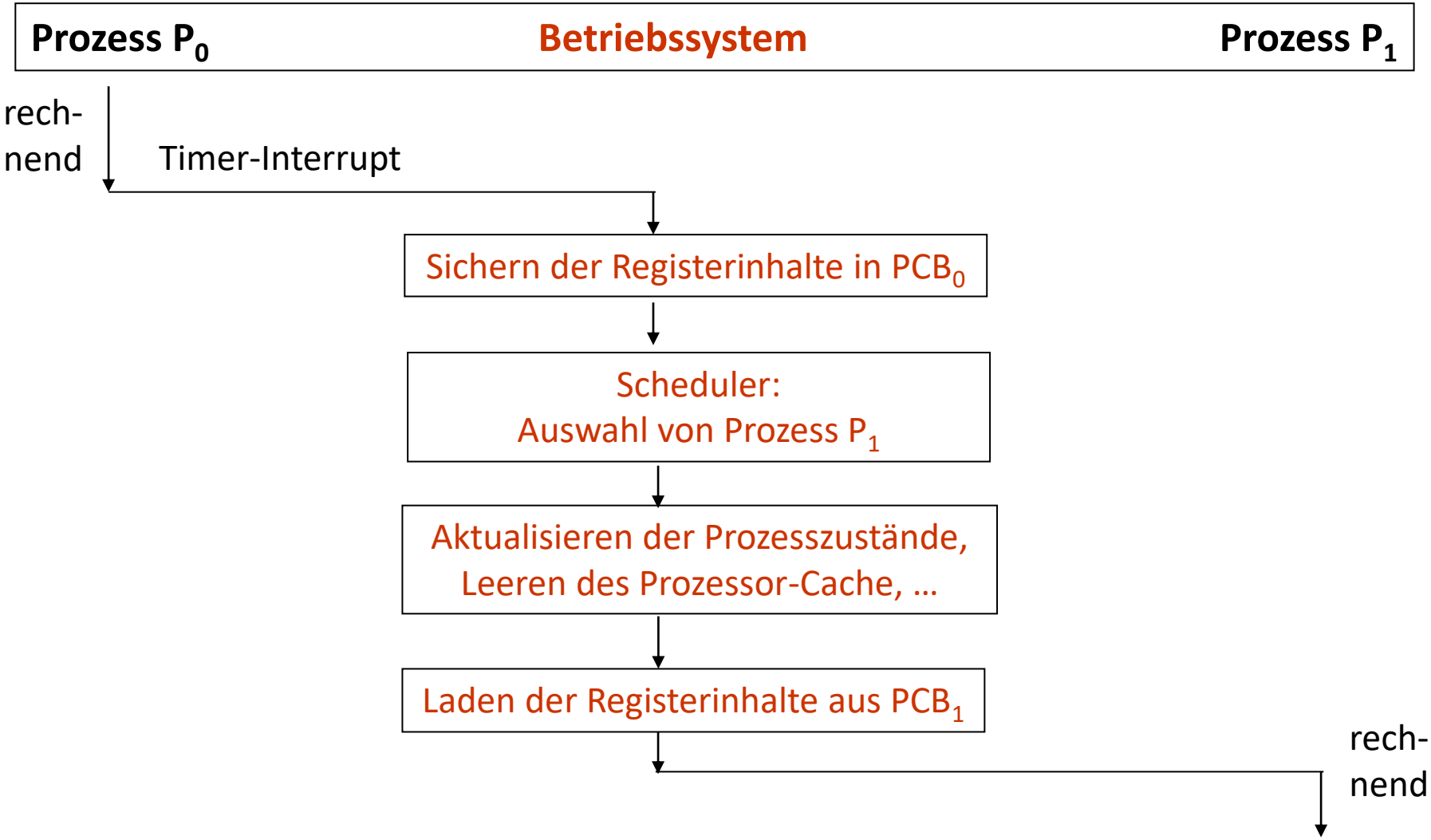
[AT]

Multiprogramming (Wdh.)

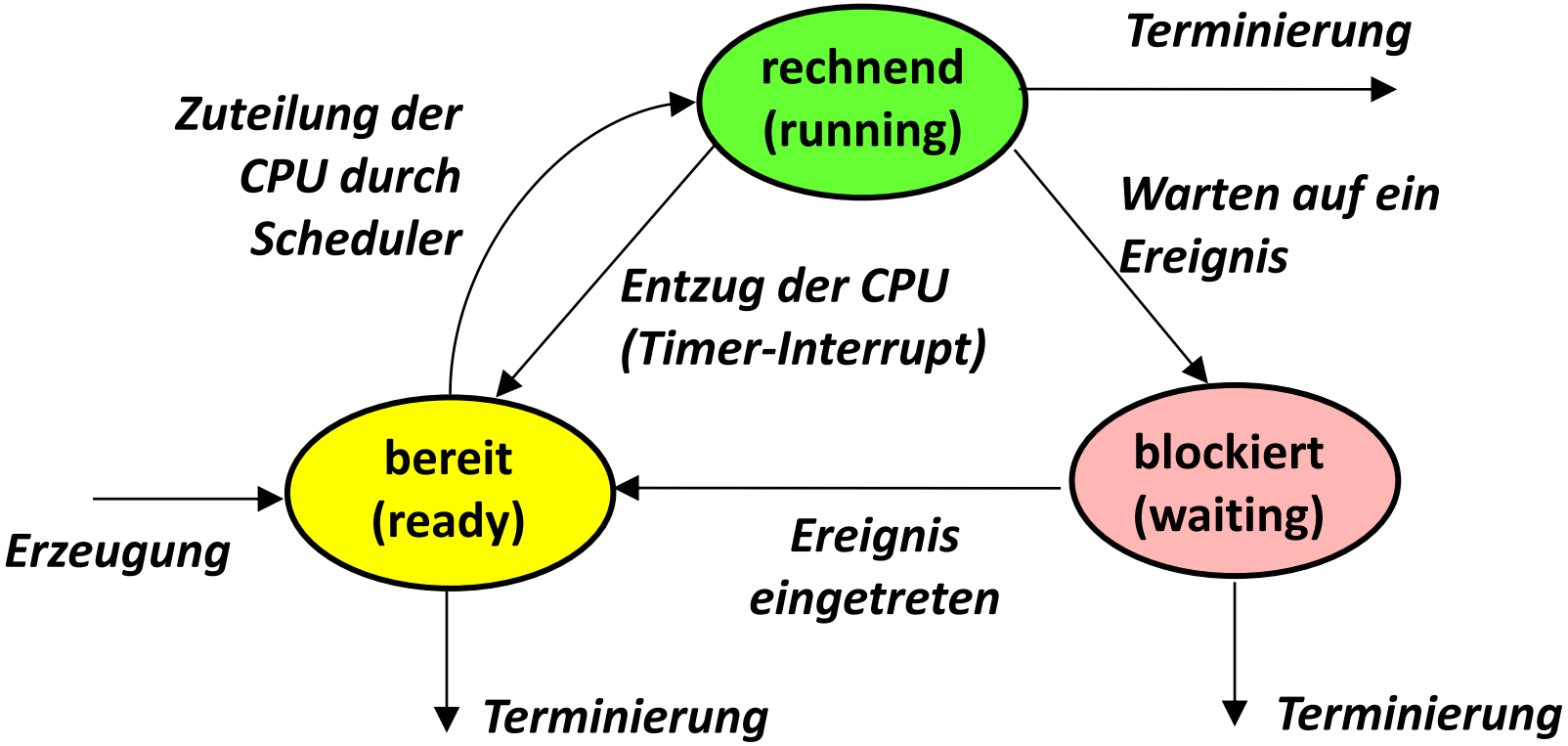
- Mehrere Prozesse gleichzeitig im Speicher
- Werden parallel ausgeführt
- Konkurrieren um Betriebsmittel (insbesondere CPU!)
- Regelmäßiger Wechsel des aktiven Prozesses
  (→ CPU-Besitz) mit „Retten“ der **Registerinhalte** („Kontext“)
  Zuteilung der CPU an einen Prozess durch „Scheduler“



Ablauf eines Prozesswechsels („Context Switch“) nach Timer-Interrupt



Prozesszustände bei Multiprogramming



Prozesserzeugung

- Wann wird ein neuer Prozess erzeugt?
  1. Initialisierung des Systems („Booten“)
     - Vordergrundprozesse (z.B. für Login)
     - Hintergrundprozesse („Daemons“, „Services“)
  2. Systemaufruf zum Erzeugen eines Prozesses durch einen anderen Prozess
  3. Benutzeranforderung (Starten eines Programms)
  4. Start eines Batch-Jobs („Task-Planer“)
- Implementierung der Prozesserzeugung
  - Grundsätzlich durch **Systemaufruf** (Fall 2)
  - „Eltern“-Prozess und „Kind“-Prozess haben getrennte Adressräume (eigene Speicherbereiche)!

Prozesserzeugung in UNIX/Linux

- Jeder Prozess hat eine eindeutige Nummer:
  Prozess-ID (PID)
- Nur der erste Prozess im System wird in der Initialisierungsphase „handgeformt“: **`init`** (PID 0)
- Alle anderen Prozesse entstehen durch den Systemaufruf **fork()**
- Ein mit dem „fork“-Aufruf neu erzeugter Prozess (**Kind-Prozess**) stellt eine Kopie des ausführenden Prozesses dar (**Eltern-Prozess**).
  - Ein Elternprozess braucht seine Ressourcen nicht zwischen seinen Kindern aufzuteilen.
  - Jeder Kind-Prozess wird separat durch UNIX geschedult
  - Die Eltern-Kind-Beziehung bleibt jedoch erhalten (Zeiger!)

Prozesserzeugung in UNIX/Linux: Systemaufrufe (1)

- **fork**
  - Aufruf: **`fork()`**
  - Erzeugt einen neuen (Kind-)Prozess als identische Kopie des ausführenden (Eltern-)Prozesses
  - Beide Prozesse machen mit der Anweisung nach dem fork - Aufruf weiter.
  - Einziger Unterschied:
    - Rückgabewert = 0 für den neuen (Kind-)Prozess und
    - Rückgabewert = Kind-PID für den Eltern-Prozess

Prozesserzeugung in UNIX/Linux: Systemaufrufe (2)

- **exec** (+ Varianten)
  - Aufruf: **`exec(Programm, Parameter, ..)`**
  - Der ausführende Prozess wird durch das neue Programm „überlagert“ (Code, Daten und Stack).
  - Anschließend Ausführung des ersten Befehls des „neuen“ Programmcodes.
  - Der Prozesskontext (PID) bleibt der alte!
- **waitpid**
  - Aufruf **`waitpid (PID, *Status, Optionen)`**
  - Der ausführende Prozess wartet solange, bis der Kindprozess PID beendet ist.
  - PID= -1 Warten, bis irgendein Kindprozess terminiert

Prozesserzeugung in UNIX/Linux: Eine einfache Beispiel-Shell

```
while (TRUE) {                                  /* Endlosschleife */
  type_prompt();                                /* Prompt ausgeben */
  read_command(&command, &params);              /* Eingabezeile von Tastatur lesen */

  PID = fork();                                 /* Kind erzeugen */
  if (PID < 0) {
        printf("Unable to fork");               /* Fehlerbedingung */
        continue;                               /* Schleife wiederholen */
  }
  if (PID > 0) {
        waitpid (PID, &status, 0);              /* Elternprozess wartet auf Kind */
     } else {
        /* PID == 0 */
        exec (command, params);                 /* Das Kind-Programm starten */
     }
}
```

Beispiel: Debugging des Shell-Codes (1)

Elternprozess 4711

```
while (TRUE) {
   type_prompt();
   read_command(&command, &params);

   PID = fork();

if (PID > 0) {
      waitpid (PID, &status, 0);
   } else {
      exec (command, params);
   }
}
```

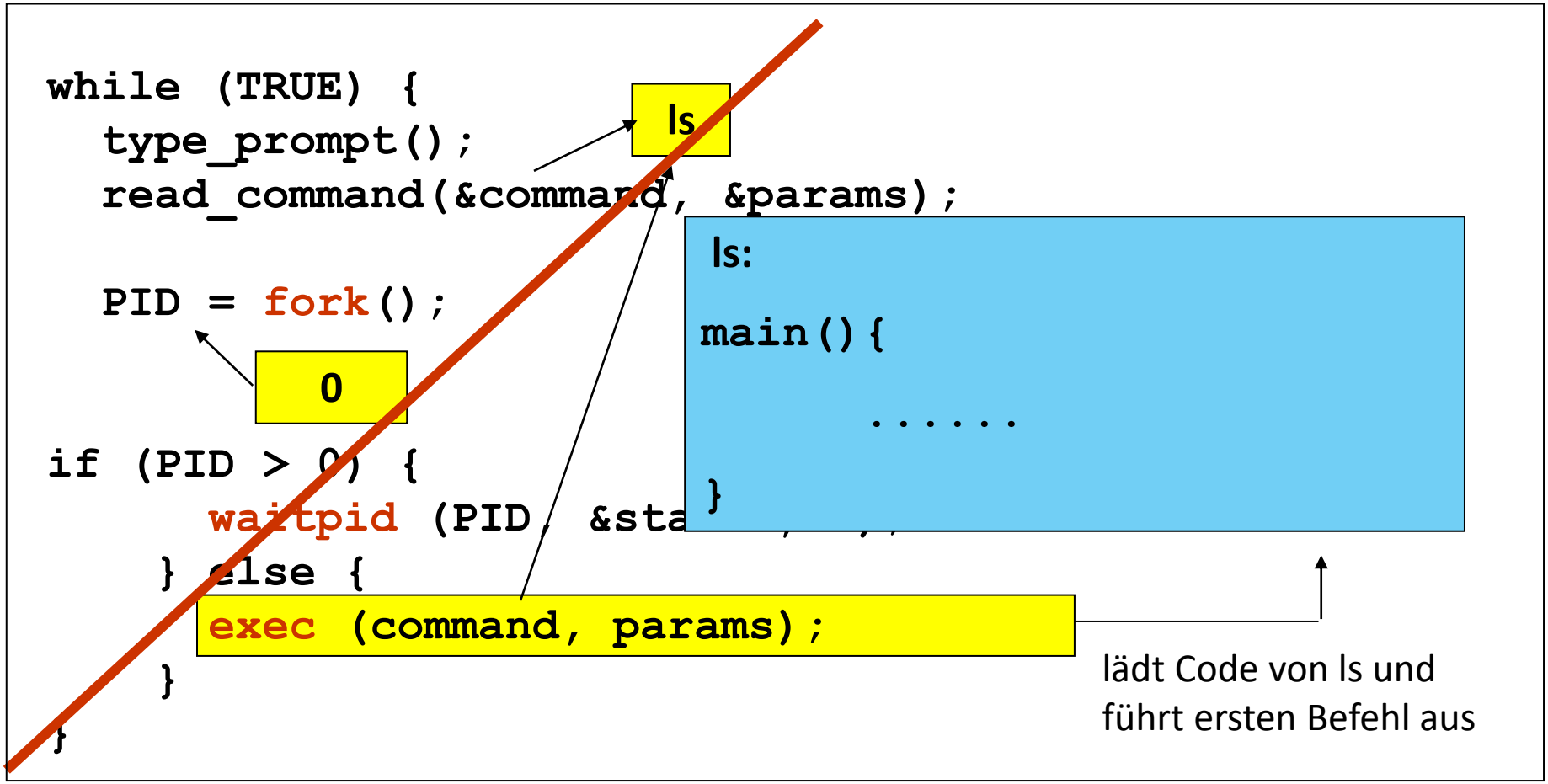
ls

4712

erzeugt parallel laufenden Kindprozess mit PID = 4712

Beispiel: Debugging des Shell-Codes (2)

Kindprozess 4712



Prozesserzeugung in Windows

- Prozesse sind als Objekte implementiert!
- Windows-API-Systemaufruf: **`CreateProcess`** `(..)`
  - 10 Parameter: Datei, …
  - Erzeugt neuen Prozess und lädt Programmcode (fork + exec)
  - Rückgabe: „handle“ auf den neuen Prozess (Objektreferenz)
- Prozessverwaltung
  - Keine *automatische* Eltern-Kind-Beziehung nach Erzeugung
  - Prozessgruppierung durch Auftrags-Objekte möglich (CreateJobObject)
- Funktion **`WaitForSingleObject`**: Warten auf verschiedene Ereignisse (u.a. Prozessende) möglich

Prozessbeendigung („Terminierung“)

- Wann wird ein Prozess beendet?
  1. Freiwilliges Beenden: Programm fertig
  2. Freiwilliges Beenden wegen selbst entdeckten Fehlers
  3. Unfreiwilliges Beenden (durch das Betriebssystem) wegen schweren Fehlers
  4. Unfreiwilliges Beenden durch einen anderen Prozess („kill“ / „TerminateProcess“)

- Funktionen: exit (UNIX), ExitProcess (Windows)

Zusammenfassung: Prozessmodell

- Prozess-Definition / Multiprogramming
- Prozesszustände
- Implementierung von Prozessen
  - Prozesstabelle / Prozesskontrollblock (PCB)
  - Prozesswechsel
- Prozess-Erzeugung und –terminierung
  - Beispiele: fork/exec, CreateProcess

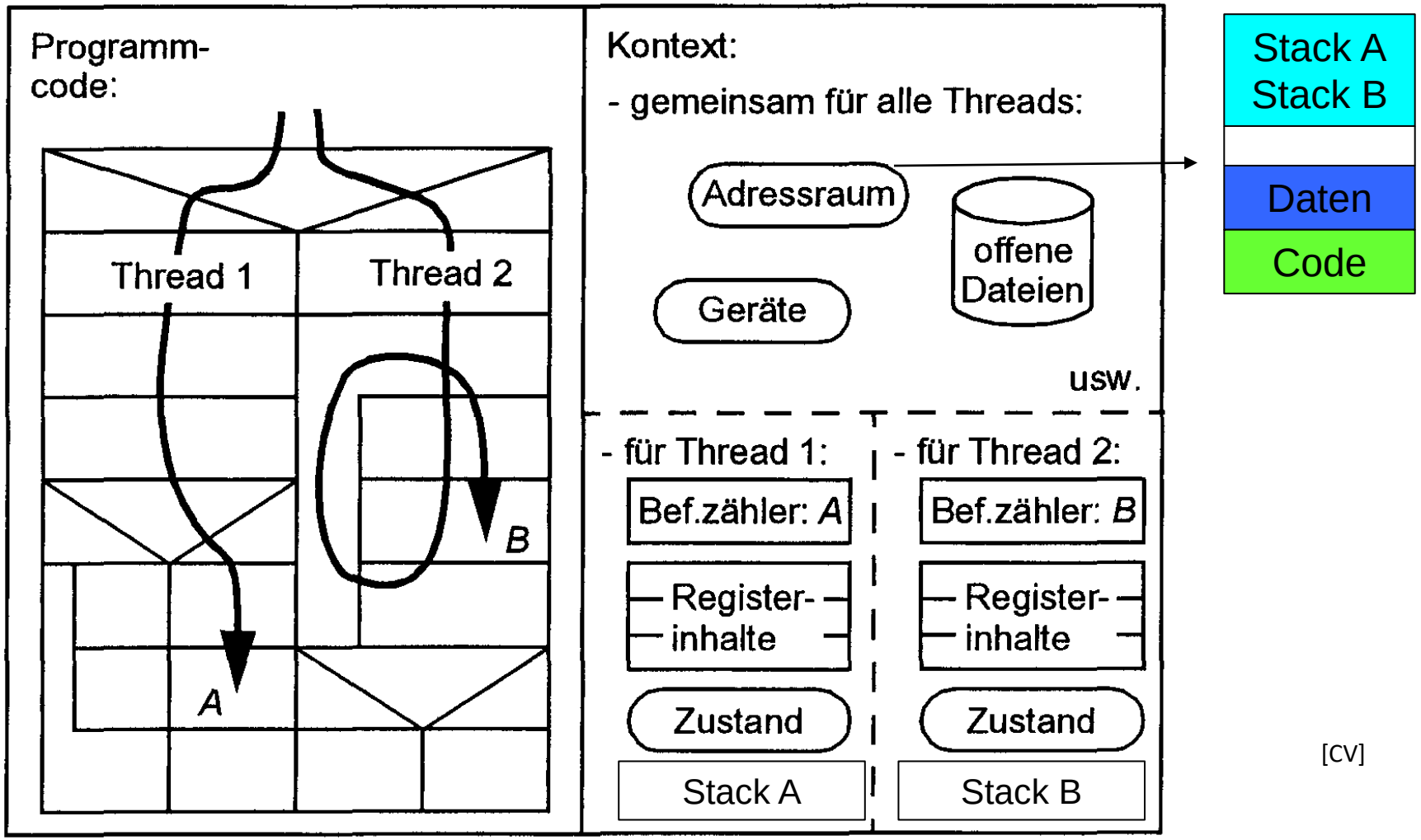
Kapitel 2
Prozesse

1. Das Prozessmodell
2. *Das Threadmodell*
3. Prozess-Scheduling

Erweiterung des Prozessmodells durch „Threads“

- **Thread = Prozess-“Faden“ („leichtgewichtiger Prozess“)**
- unabhängiger Subprozess, der innerhalb eines normalen Prozesses (Programms) abläuft
- verfügt über einen eigenen Programmzähler, Stack sowie Zustandsinformationen (eigener „Ausführungsfaden“)
- Alle Threads eines Prozesses laufen in demselben Adressraum, sie können deshalb globale Variable gemeinsam nutzen.
- Ein Prozess kann mehrere Threads enthalten.
- Threads laufen wie die normalen Prozesse asynchron ab
- Threads können sich ebenso in einem der Zustände rechnend, bereit oder blockiert befinden.
- Alle Threads haben Zugriff auf die Betriebsmittel des Prozesses (geöffnete Dateien, Kindprozesse, Signale usw.)

Threads eines Prozesses



[CV]

Analogie Prozess / Haus

Prozess	Haus
➢ Ein **Prozess** beinhaltet die Ressourcen für „seine“ Threads.	➢ Ein **Haus** stellt die Einrichtungen (Küche, Dusche etc.) für „seine“ Bewohner bereit.
➢ Ein **Thread** läuft in genau einem Prozess ab (solange er existiert).	➢ Ein **Bewohner** lebt in dem Haus und wird es nie verlassen – von der Geburt bis zum Tod (hier hinkt der Vergleich).

Daraus ergibt sich:

➢ Mehrere Threads können gleichzeitig in einem Prozess ablaufen.	➢ Mehrere Bewohner können gleichzeitig in einem Haus leben.
➢ Die Threads eines Prozesses greifen auf dieselben Ressourcen (Speicher, Dateien, ..) zu. Koordination ist notwendig (z.B. via Synchronisationsmechanismen)	➢ Die Bewohner benutzen dieselben Einrichtungen gleichzeitig. Koordination ist notwendig (z.B. via „BESETZT“ Schild)
➢ Threads laufen auf einer CPU ab. Ein Prozess ist passiv – er ist nur ein Container für Ressourcen.	➢ Nur die Bewohner „tun was“, nicht das Haus.

Wozu sind Threads zu gebrauchen?

- Parallele Aktionen innerhalb eines Anwendungsprogramms werden möglich!
- Beispiele ?
  - …
  - …
  - …
- Erzeugung über spezielle Systemaufrufe (z.B. Windows-API) oder Programmiersprachenkonstrukte (z.B. JAVA-Klasse „Thread“)
- Parallelverarbeitung bei Multiprozessoreinsatz wird unterstützt!

Vorteile von Threads gegenüber Prozessen

- Die Programmierung paralleler Aktionen wird vereinfacht
  - Threads realisieren Nebenläufigkeit innerhalb eines Anwendungsprogramms
    → Sequentielle Programmierung paralleler Aktionen
  - Threads können über den gemeinsamen Speicher ihres Prozesses (Zugriff auf dieselben Variablen und Objekte) zusammenarbeiten
- Die Ausführung paralleler Aktionen wird beschleunigt
  - Thread-Erzeugung kostet deutlich weniger Zeit als Prozess-Erzeugung
  - Das Gleiche gilt für das Thread-Ende
  - Das Umschalten zwischen den Threads eines Prozesses ist schneller als das Umschalten zwischen Prozessen

ABER: Die Code-Ausführung selbst ist bei Threads und Prozessen gleich schnell!

Threadverwaltung / -scheduling (1)

.. durch Betriebssystem oder Benutzerprozess?

- Thread-Scheduling durch Betriebssystem-Kernel („Kernel Level Threads“) :
  - Die Threads aller Prozesse sind gleichwertig.
  - Verwaltung einer vollständigen Thread-Tabelle durch den Kernel *(Registerinhalte, Zustand, ...)*
  - Nach einem blockierenden Thread (z.B. E/A-Anforderung) kann direkt ein anderer Thread desselben Prozesses aktiv werden

  Moderne Betriebssysteme unterstützen üblicherweise Kernel Level Threads (z.B. Windows, z/OS)

Threadverwaltung / -scheduling (2)

.. durch Betriebssystem oder Benutzerprozess?

- Thread-Scheduling durch Benutzerprozess selbst („User Level Threads“):
  - Der Kernel kennt keine Threads.
  - Verwaltung einer Thread-Tabelle durch den Benutzerprozess
  - Ein blockierender Thread blockiert den gesamten Prozess und damit alle anderen Threads dieses Prozesses

Das Java-Threadmodell

- JAVA unterstützt **Multithreading**
- Bis Java 1.2: User level threads
  - Thread-Verwaltung durch JVM – JAVA Virtual Machine
- Ab Java 1.3: **Kernel Level-Threads**
  - Erzeugen von Betriebssystem-Threads (falls vom BS unterstützt) oder eigenen Prozessen

Java-Threads

- Die **`main`**-Methode eines Java-Programms wird bereits durch einen Thread ausgeführt
  - ohne explizite Thread-Verwendung läuft nur der eine "Haupt"-Thread ab *("main")*
- Zum Erzeugen weiterer Threads gibt es im Package **`java.lang`** ...
  - die Klasse **Thread**
    - Einzige von einer abgeleiteten Klasse **zu redefinierende Methode**: **`void run()`**
  - das Interface **Runnable** (als Alternative)
    - Einzige **zu implementierende Methode**: **`void run()`**
- **`void run()`**
  - muss mit dem Sourcecode für den eigenen Thread (re-)definiert werden (die eigentlichen Aktionen)

Konstruktoren der Klasse `Thread` (Auswahl)

Es wird jeweils ein neues Java-Threadobjekt erzeugt, aber noch nicht gestartet!

- **`Thread()`**
  - Erzeugt ein neues Java-Threadobjekt mit Namen "Thread-x"
    (x wird von der Java-VM hochgezählt)
- **`Thread(Runnable target)`**
  - Erzeugt ein neues Java-Threadobjekt, wobei die **`run`**-Methode des übergebenen Objekts **`target`** beim (späteren) Starten ausgeführt wird
  - Das übergebenen Objekt muss das Runnable-Interface implementieren!

Methode der Klasse `Thread` zum Starten eines neuen Threads

- **`void start()`**
  - erzeugt, initialisiert und startet den Thread auf Betriebssystem-Ebene
  - ruft die Methode **`run()`** automatisch auf
  - **`start()`** kehrt sofort zurück, der neue Thread läuft parallel
  - Niemals **`run()`** direkt aufrufen!

➔ Beispiele: ThreadTest1a / ThreadTest1b

Methoden der Klasse `Thread` zum Beenden eines Threads

- „Normales“ Ende: **`run`**-Methode ist fertig!
- Beenden durch einen anderen Thread:
  - ***`void stop()`***
    - nicht mehr verwenden (unsicher)!! ➔ Beispiel: ThreadTest2
  - **Stattdessen**:
  - **`void interrupt()`**
    - Setzt für den Thread ein Interrupt-Flag (entspricht Setter einer Objektvariablen vom Typ **`boolean`**) *und "weckt" ihn, falls er blockiert ist*
  - **`boolean isInterrupted()`**
    - Liefert den Wert des Interrupt-Flags für den Thread (entspricht Getter einer Objektvariablen vom Typ **`boolean`**)
    - Muss vom Thread-Code abgefragt werden
    - ➔ Beispiel: ThreadTest3

Ein Thread kann nach Beendigung nicht neu gestartet werden!

Methoden der Klasse `Thread` zum Anhalten und Fortsetzen eines Threads

- `static void sleep(long milliseconds) throws InterruptedException`
- `static void sleep(long milliseconds, long nanoseconds) throws InterruptedException`

➢ Hält den ausgeführten Thread für die angegebene Zeit an

➢ ***throws InterruptedException:*** Ein zwischendurch erfolgender Aufruf von **`interrupt()`** (durch einen anderen Thread) erzeugt eine **`InterruptedException`**, „weckt“ den ausführenden Thread und muss von diesem behandelt werden (catch)!

➢ Der catch-Aufruf setzt das Interrupt-Flag automatisch auf **`false`** zurück → Interrupt-Flag muss im catch-Block ggf. neu gesetzt werden, falls der Zustand im Code mit **`isInterrupted()`** abgefragt wird!

➔ Beispiel: ThreadTest4

Weitere Methoden der Klasse `Thread`

- `static Thread currentThread()`
  - Liefert das Java-Thread-Objekt des ausgeführten Threads
- `String getName()`
  - Liefert den Namen des Threads
- `void setName(String name)`
  - Weist den Namen des Threads zu
- `boolean isAlive()`
  - Liefert `true`, wenn der Thread gestartet, aber noch nicht beendet wurde
- `void join() throws InterruptedException`
  - Hält den ausgeführten Thread an, bis dieser Thread (das Zielobjekt beim Aufruf) beendet ist.

➔ Beispiel: ThreadTest5

Threads in Linux

- Linux-Threads sind Prozesse, die teilweise den gleichen Kontext besitzen (allg. „Tasks“ mit „Task-ID“)
- Für den Kernel kein Unterschied zu „normalen“ Prozessen
- Threaderzeugung kann durch speziellen fork() – Befehl - heißt **clone()** - erreicht werden
- clone() bietet die Möglichkeit, den Anteil des gemeinsamen Prozesskontexts/Adressraums über Parameter festzulegen
- Verbreitetes API: POSIX-Threads („pthreads“)
  → eigene Bibliothek *(benutzt in Linux „clone()“)*

Im Original-UNIX gibt es keine Threads!

Threads in Windows

- Vollständige Implementierung von Threadobjekten
- Zugriff über Windows-API-Funktionen
  (CreateThread, GetCurrentThread, SuspendThread, ResumeThread, Sleep, CloseHandle, …)

- Alle Threads sind Kernel Level-Threads
  - Scheduling durch den Kern

➔ Beispiel: → WindowsThreadTest.c

Zusammenfassung: Threadmodell

- Was ist ein Thread?
- Nutzen / Vorteile von Threads
- Kernel Level / User Level Threads
- Threads in
  - JAVA
  - Linux
  - Windows

Kapitel 2
Prozesse

1. Das Prozessmodell
2. Das Threadmodell
3. *Prozess-Scheduling*

Prozess-Scheduling: Grundlagen

- Scheduling (Erstellen eines Ablaufplans) findet im Betriebssystem an unterschiedlichen Stellen statt
- Meist ist das Scheduling von Prozessen (CPU-Zuteilung) gemeint; Algorithmen gelten auch für Kernel Level-Threads!
- Grundproblem: Welcher Prozess/Thread bekommt wann für wie lange die CPU? ➔ Planungs- und Entscheidungproblem des „Prozess-Schedulers“
- Kategorien von Scheduling-Algorithmen:
  - Stapelverarbeitung („Batch-Processing“)
  - Interaktive Systeme
  - Echtzeitsysteme
- Weitere Frage: CPU-Scheduling in Multiprozessorsystemen?

Arten von Scheduling – Algorithmen

- Nicht-unterbrechend („non-preemptive“)
  - Wenn ein Prozess im Zustand „Running“ ist, wird er ausgeführt, bis er terminiert oder selbst blockiert durch Warten auf ein Ereignis
  - Im kooperativem Multiprogramming (Windows 3.11) konnte der Prozess die CPU auch freiwillig abgeben
- Unterbrechend („preemptive“)
  - Ein laufender Prozess kann vom Betriebssystem unterbrochen werden („Running“ $\Rightarrow$ „Ready“)
  - System ist zuverlässiger, da kein Prozess die CPU monopolisieren kann
- Moderne Betriebssysteme verwenden unterbrechende Scheduling-Algorithmen

Ziele von Scheduling-Algorithmen

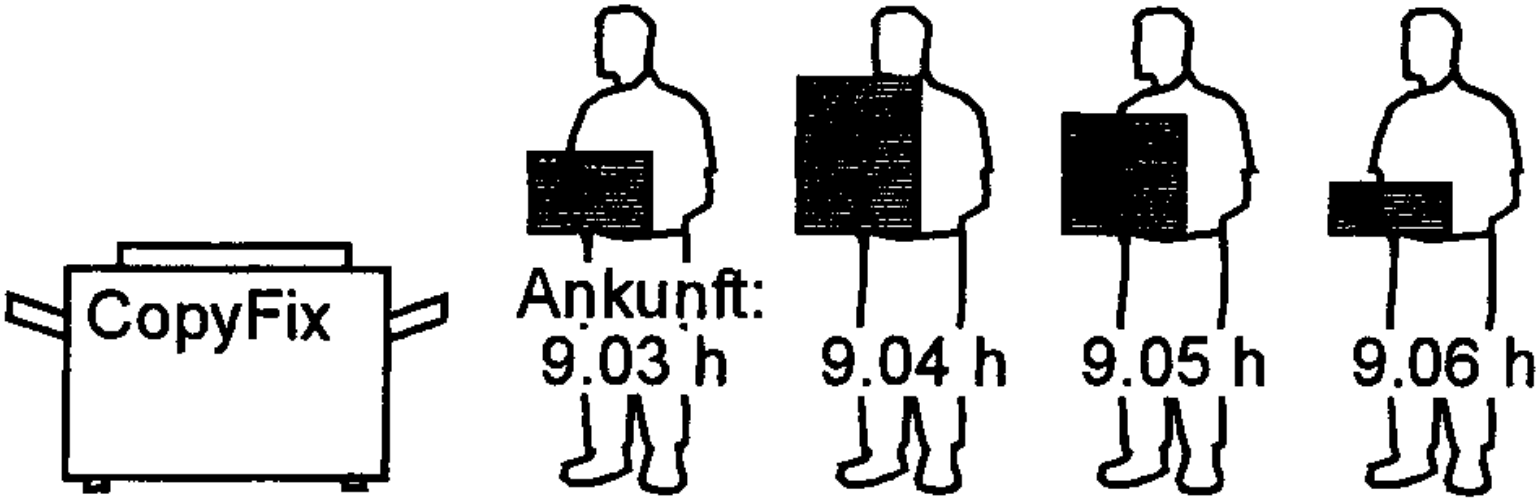
- Alle Systeme
  - Fairness - jeder Prozess bekommt Rechenzeit der CPU
  - Policy Enforcement - Strategien werden sichtbar durchgeführt
  - Balance - alle Teile des Systems sind gleichmäßig ausgelastet
- Stapelverarbeitungssysteme
  - Durchsatz - maximiere nach Jobs pro Stunde
  - Turnaround-Zeit - minimiere die Zeit vom Start bis zur Beendigung
  - CPU-Belegung - belege die CPU konstant mit Jobs
- Interaktive Systeme
  - Antwortzeit - antworte schnellstmöglich auf Anfragen
  - Proportionalität - auf die Bedürfnisse des Nutzers eingehen
- Echtzeitsysteme
  - Meeting Deadlines - keine Daten verlieren
  - Predictability - Qualitätsverlust bei Multimedia vermeiden

Scheduling-Algorithmen: Überblick

Algorithmen können auch kombiniert werden!

- Stapelverarbeitungssysteme
  - First-Come First-Served
  - Shortest Job First
  - Shortest Remaining Time Next
- Interaktive Systeme
  - Round-Robin („Zeitscheibenverfahren“)
  - Prioritätenbasiertes Scheduling
    - statisch / dynamisch
    - mit mehreren Prioritätsklassen / Warteschlangen
- Echzeitsysteme
  - Rate Monotonic Scheduling (für periodische Prozesse)
  - Earliest Deadline First

First-Come First-Served (FCFS)



[CV]

- Jeder Prozess wird in der Reihenfolge der Ankunft (Entstehung) in die Ready Queue (Warteschlange mit Prozessen im „Bereit“-Zustand) eingetragen
- Wenn ein Prozess endet (oder blockiert), wird jeweils der älteste aus der Ready Queue entnommen

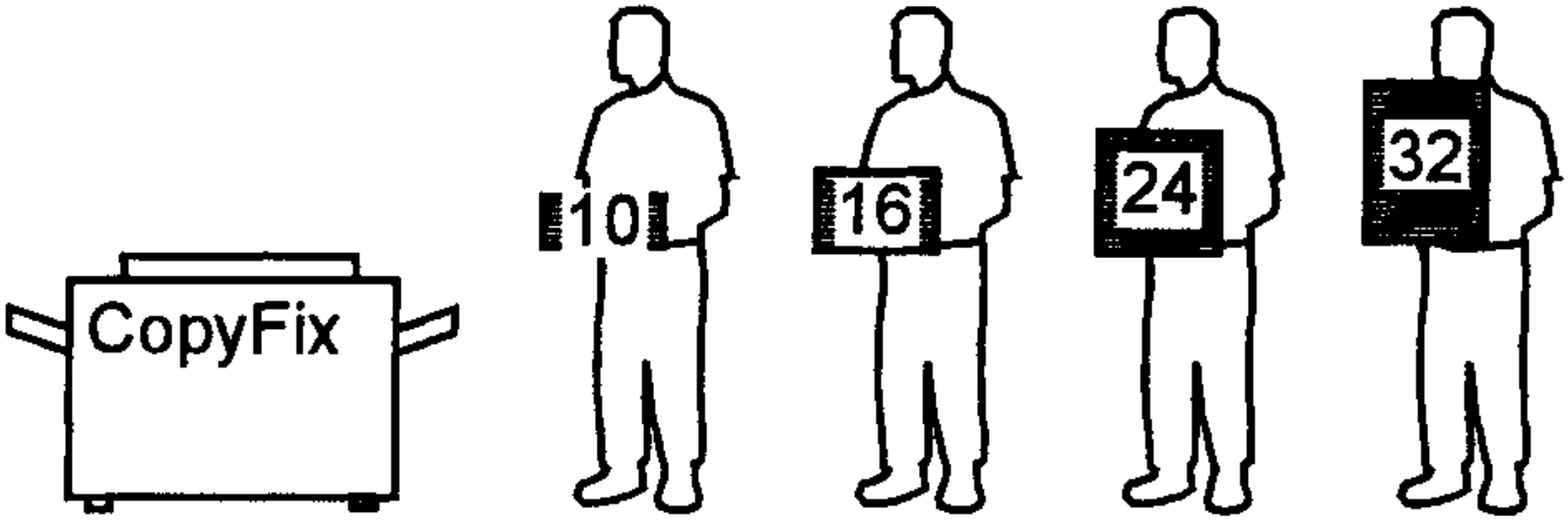
First-Come First-Served (FCFS)

- Nicht-unterbrechend (CPU-Freigabe nur durch Blockierung oder Terminierung)
- **Nachteil:** Ein (u.U. kurzer) Prozess muss möglicherweise sehr lange Zeit warten, bevor er gestartet wird
- **Nachteil:** Bevorzugt rechenintensive Prozesse gegenüber I/O-intensiven Prozessen

➢ FCFS ist als primäres Verfahren nicht interessant

➢ Wird in Kombination mit anderen Verfahren genutzt

Shortest Job First (SJF)



[CV]

- Prozess mit kürzester erwarteter Laufzeit wird als nächstes ausgewählt
- Ein kurzer Prozess überholt längere in der Queue
- Ebenfalls nicht-unterbrechend

Shortest Job First (SJF)

- **Voraussetzung**: Laufzeiten sind vorher bekannt
  (→ ggf. durch Erfahrungswerte oder Benutzerangaben)!
- Wenn die angegebene Laufzeit überschritten ist, kann der Prozess abgebrochen werden
- **Nachteil**: Große Jobs können verhungern
- Wenn mehrere Jobs gleichzeitig anstehen, ist SJF beweisbar optimal bzgl. der Minimierung der mittleren Gesamtwartezeit!
  (→ Beweis über Beispiel)

Shortest Remaining Time Next

- Der Prozess mit der kürzesten Restlaufzeit wird als nächster ausgewählt.
- Unterbrechende Version von „Shortest Job First “
- Restlaufzeit muss bekannt sein oder die Prozessor-Zeit muss geschätzt werden aufgrund des bisherigen Verhaltens.
- Optimiert für den Fall unterschiedlicher Ankunftszeiten
- Gut für neue Prozesse mit kurzer Laufzeit!

Round-Robin (Zeitscheibenverfahren)

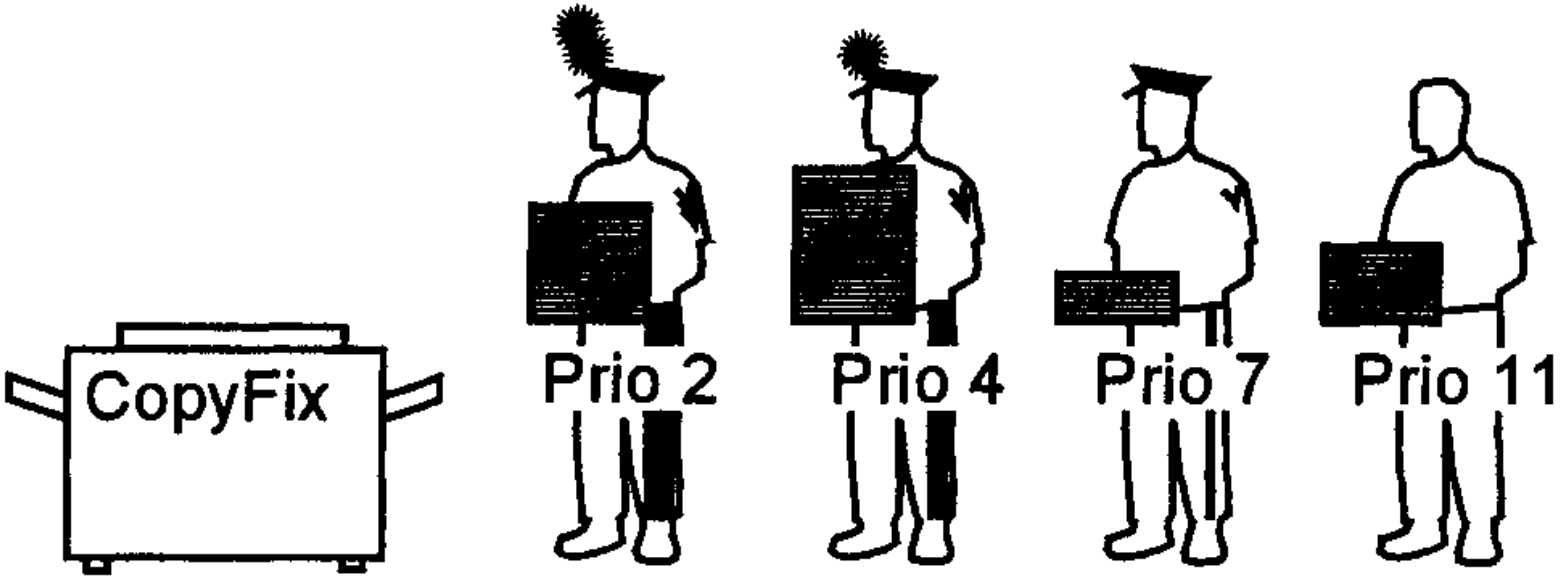
[CV]

- Die gesamte CPU-Zeit wird gleichmäßig auf alle Prozesse verteilt
- Eine feste Zeitscheibe („Quantum“) von x ms wird für alle Prozesse festgelegt
- Jeder Prozess bekommt für x ms die CPU und wird nach Ablauf dieser Zeitscheibe unterbrochen
- Jeder Prozess erhält die CPU erst wieder, wenn zwischenzeitlich alle anderen Prozesse ihre Zeitscheibe verbrauchen durften

Round-Robin - Implementierung

- Regelmäßige Unterbrechung durch einen Timer-Interrupt (Timer – Interrupt wird periodisch generiert)
- Wenn ein Interrupt erfolgt, wird der aktuell laufende Prozess zurück in die „Ready“-Queue gestellt und der nächste Prozess wird ausgesucht (→ Prozesswechsel!)
- Blockiert ein Prozess in „seiner“ Zeitscheibe, bekommt der nächste Prozess (Thread) die CPU
- Optimierung der Zeitscheibengröße (Quantum)?
  - zu kurz: viele Prozesswechsel, viel Overhead
  - zu lang: evtl. schlechte Antwortzeiten
  - Erfahrungswert: ~20 – 50 ms

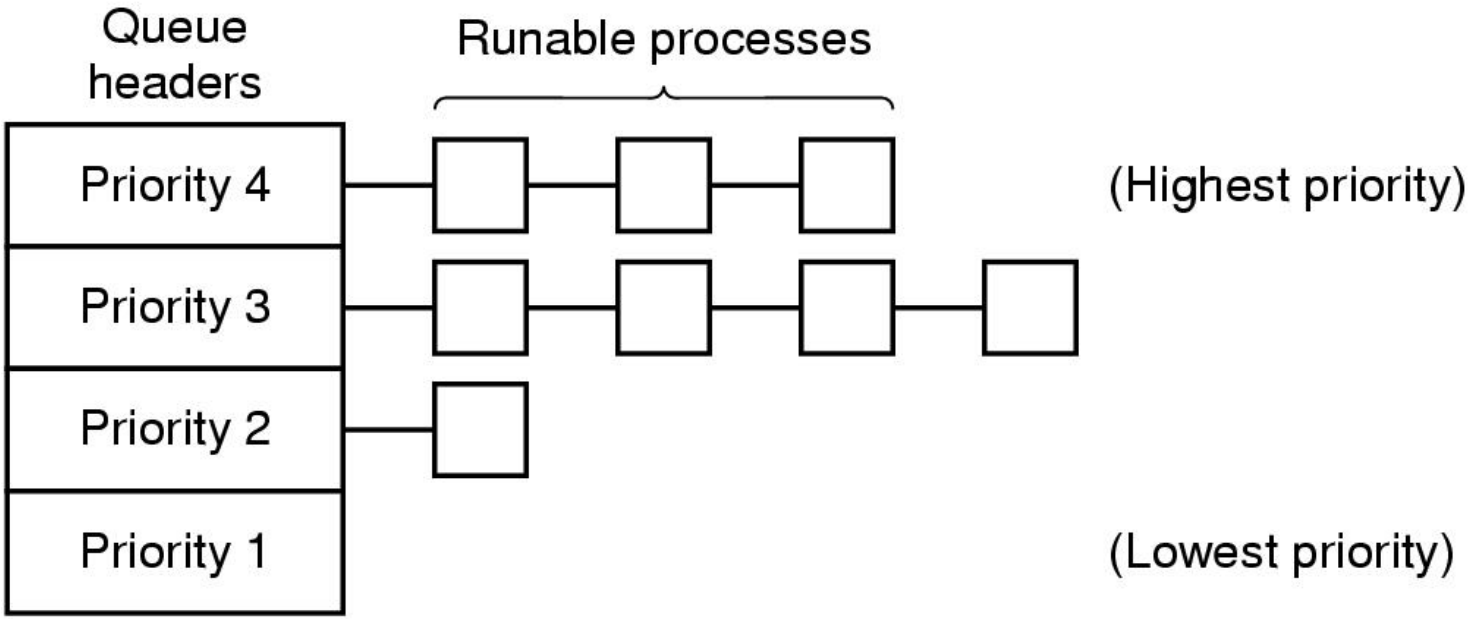
Prioritätenbasiertes Scheduling



[CV]

- Zuweisen von Prioritäten an Prozesse
- Bevorzugung des Prozesses mit der höchsten Priorität

Scheduling mit Prioritätsklassen



[AT]

- Realisierung i.d.R. durch eine Warteschlange pro Prioritätsklasse.
- Die ankommenden Prozesse werden je nach Prioritätsklasse in die entsprechende Warteschlange eingereiht.
- Prozesse einer Klasse sind erst „dran“, wenn alle höher priorisierten Klassen-Warteschlangen leer sind!
- Scheduling innerhalb einer Klasse: meist Round-Robin-Verfahren (ggf. mit unterschiedlichen Zeitscheiben)!

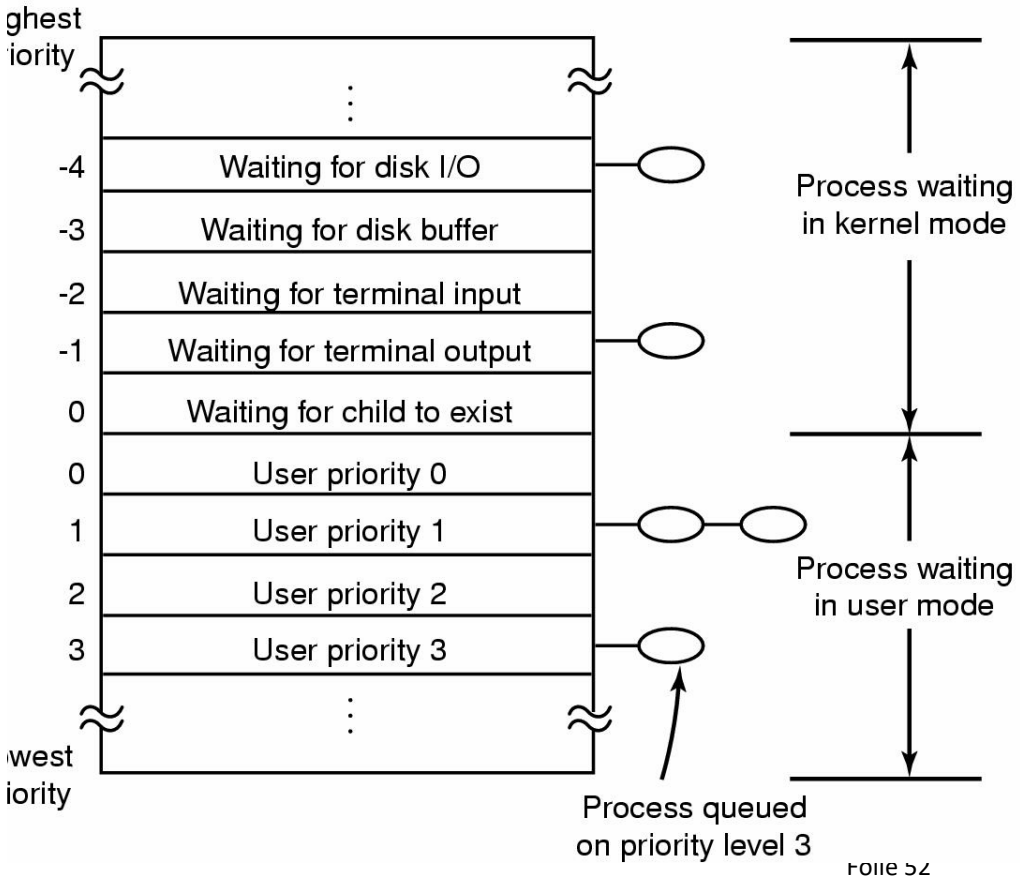
Scheduling mit dynamischen Prioritäten („Multilevel Feedback Queueing“)

- Regelmäßige Neuberechnung der Prioritäten
  → Prozesse können in eine andere Klasse verschoben werden
- Mögliche Kriterien für die Neuberechnung:
  - Alter des Prozesses
  - Bisherige Wartezeit des Prozesses
  - Verhalten des Prozesses (z.B. regelmäßiges Verbrauchen der gesamten Zeitscheibe)
  - Status eines Prozesses (Systemprozess/Benutzerprozess)
  - …
- Vorteil: „Verhungern“ von Prozessen wird vermieden

→ Standard-Algorithmus für moderne Betriebssysteme!

Traditionelles UNIX-Scheduling

- Multilevel-Feedback mit Round-Robin in den einzelnen Prioritätsklassen
- Hohe Prioritätsklasse ➔ niedrige Priorität
- Prioritäten werden 1-mal pro Sekunde neu berechnet nach der Formel:
  Prioritätsklasse = CPU-Verbrauch + Nice + Basispriorität
- Korrektur der Priorität
  bevorzugt Interaktive Prozesse
- Priorität von Systemprozessen (<= 0) ist immer höher als die Priorität von User-Prozessen (>= 0)
- Mit einem „Nice“-Wert (immer > 0) kann ein Benutzer die Priorität eines seiner Prozesse heruntersetzen
- Scheduling-Algorithmen bei Linux mittlerweile deutlich komplexer



Windows Scheduling

- Multilevel-Feedback mit Round-Robin in den einzelnen Prioritätsklassen
- Windows schedult nur Threads
- 32 Prioritätklassen (0-31) in 2 Kategorien angeordnet:
  - „Echtzeit“ (Klasse 31-16) [hohe Priorität]
    - Für System-Threads (Kernel-Mode)
    - Feste Prioritäten, Round-Robin bei gleicher Priorität
  - Variabel (Klasse 0-15) [niedrige Priorität]
    - Für Benutzer-Threads
    - Variable Prioritäten, je nach Thread-Verhalten
    - Priorität steigt bei Warten auf Ereignis
    - Priorität sinkt bei Verbrauch einer ganzen Zeitscheibe

Steuerung der Benutzer-Threadpriorität über Windows-API

- Windows-API: Zuweisen von Prioritäten möglich für
  - einen Prozess (→ Basispriorität für alle Threads dieses Prozesses): `SetPriorityClass` (6 mögliche Werte)
  - einen Thread (→ Priorität relativ zu den anderen Threads desselben Prozesses): `SetThreadPriority` (7 mögliche Werte)
- Abbildung der 42 möglichen Win32-Prioritäten auf die 32 Windows-Prioritätsklassen:

		Win32 process class priorities					
		Realtime	High	Above Normal	Normal	Below Normal	Idle
Win32 thread priorities	Time critical	31	15	15	15	15	15
	Highest	26	15	12	10	8	6
	Above normal	25	14	11	9	7	5
	Normal	24	13	10	8	6	4
	Below normal	23	12	9	7	5	3
	Lowest	22	11	8	6	4	2
	Idle	16	1	1	1	1	1

JAVA Thread - Scheduling

- Verwendung von Prioritätsklassen
- Realisierung der Java-Thread-Priorität über Betriebssystem-API
- Methoden der Klasse Thread für das Scheduling:

```
public void setPriority(int newPrio)
```

  - Weist dem Thread eine Priorität zu
    - zwischen Thread.MIN_PRIORITY (üblich: 1) und Thread.MAX_PRIORITY (üblich: 10)
    - Default: Thread.NORM_PRIORITY (üblich: 5)

```
public int getPriority()
```

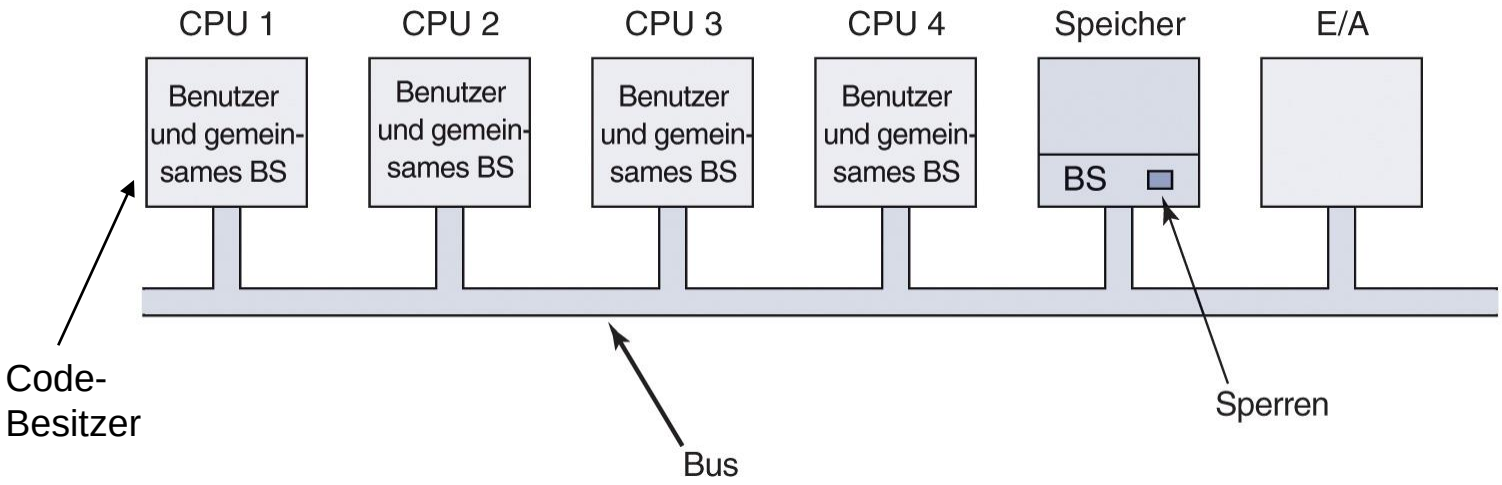
  - Liefert die aktuelle Priorität

→ *Aber: Das Betriebssystem trifft die Scheduling-Entscheidungen nach seinen Regeln!*

➔ TestThread6

Multiprozessor-Scheduling

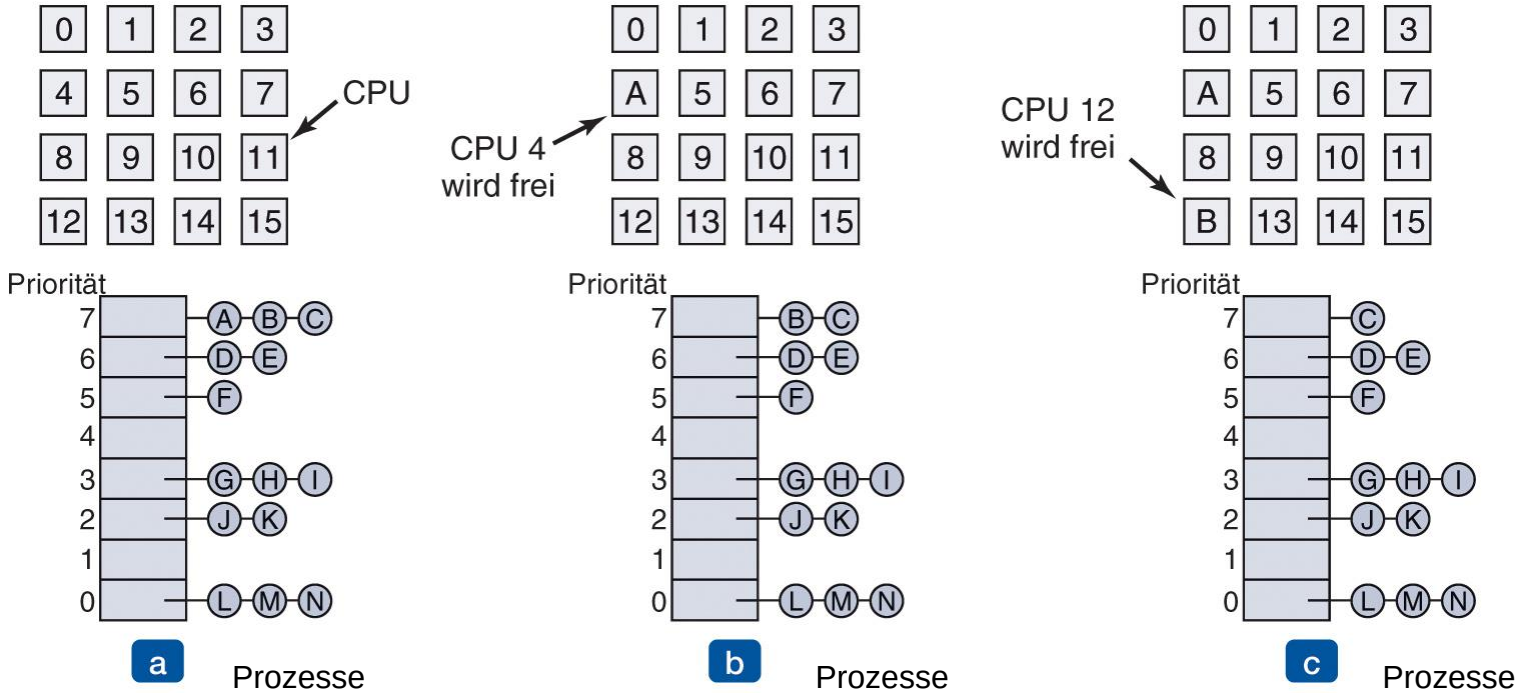
- Mehrere CPUs vorhanden
- Häufigstes Verfahren: **Symmetrisches Multiprozessing (SMP)**
- Das Betriebssystem liegt nur *einmal* im Hauptspeicher
- Alle CPUs haben Zugriff auf das Betriebssystem
- Nur *eine* CPU gleichzeitig führt Betriebssystemcode (eines spezifischen Bereichs) aus *(geschützt durch Sperren → Synchronisation)*
- Scheduling-Verfahren:
  - Time-Sharing mit gemeinsamen Prioritäts-Warteschlangen
  - Time-Sharing mit eigenen Prioritäts-Warteschlangen
  - *Gang*-Scheduling



[AT]

Multiprozessor-Scheduling: Time-Sharing mit gemeinsamen Prioritäts-Warteschlangen

- Es gibt nur *eine* gemeinsame Ready-Queue mit Prioritäts-Warteschlangen
- Sobald eine CPU frei wird: Zuordnung des am höchsten priorisierten Prozesses (Threads) zu der CPU

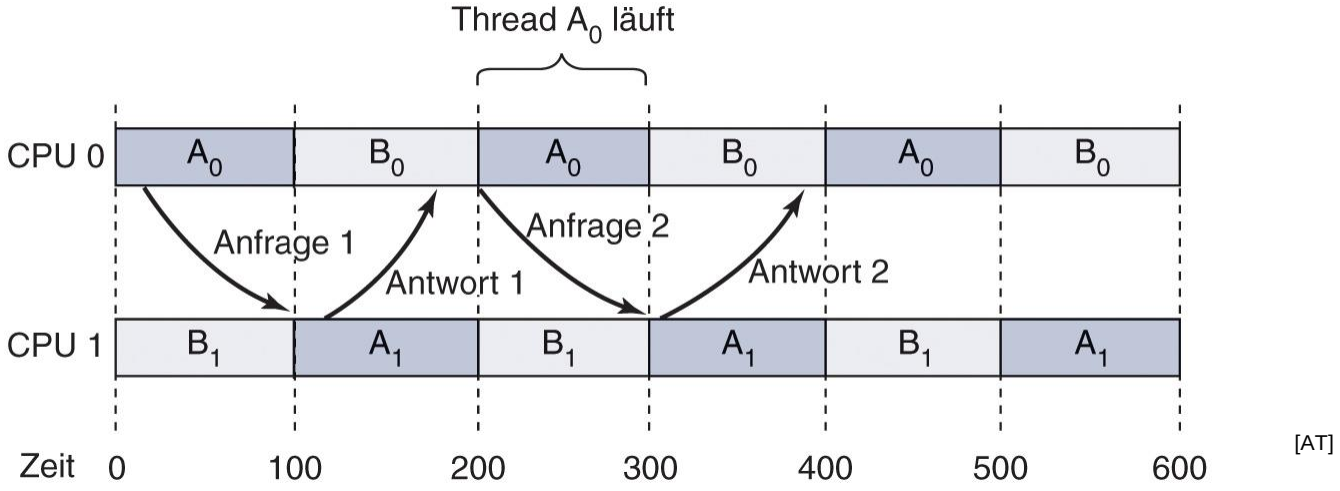


Multiprozessor-Scheduling: Time-Sharing mit eigenen Prioritäts-Warteschlangen

- Bei Erzeugung wird ein Prozess (Thread) fest einer CPU zugeordnet *(z.B. der CPU mit der kürzesten Warteschlange)*
- Jede CPU hat *eigene* Prioritäts-Warteschlangen.
- Nur, falls die Warteschlangen einer CPU komplett leer sind, darf sie Prozesse (Threads) von einer anderen CPU *stehlen*
- Vorteil: Nutzung von vorhandenen Cache-Blocks einer CPU möglich
  - Gleichmäßige Auslastung trotzdem gewährleistet
  - Notwendiger Zugriff auf fremde Warteschlangen i.A. selten

Multiprozessor-Scheduling: Optimierung für Threads mit gemeinsamer Kommunikation

- Problem: Scheduling von Threads (z.B. desselben Prozesses), die häufig miteinander kommunizieren
- Bei Verlagerung auf verschiedene CPUs muss ggf. erst abgewartet werden, bis der andere Thread seine CPU erhält



[AT]

- Optimal: Alle zusammengehörigen Threads (A_0+A_1, B_0+B_1) können **gleichzeitig** laufen!

Multiprozessor-Scheduling: *Gang*-Scheduling

- Zusammengehörige Threads werden als Gruppe (*Gang*) geschedult
- Alle Gang-Mitglieder laufen gleichzeitig auf verschiedenen CPUs (falls genug CPUs vorhanden sind)
- Alle Threads auf allen CPUs beginnen und beenden **gleichzeitig ihre Zeitscheiben**!

Zeitabschnitt	CPU 0	CPU 1	CPU 2	CPU 3	CPU 4	CPU 5
0	A_0	A_1	A_2	A_3	A_4	A_5
1	B_0	B_1	B_2	C_0	C_1	C_2
2	D_0	D_1	D_2	D_3	D_4	E_0
3	E_1	E_2	E_3	E_4	E_5	E_6
4	A_0	A_1	A_2	A_3	A_4	A_5
5	B_0	B_1	B_2	C_0	C_1	C_2
6	D_0	D_1	D_2	D_3	D_4	E_0
7	E_1	E_2	E_3	E_4	E_5	E_6

[AT]

Unix/Windows-Multiprozessorscheduling

- Konzeptioneller Unix / Windows-Standard:
  Time-Sharing mit gemeinsamen Prioritäts-Warteschlangen
- Zusätzlich bei Windows:
  Bei Mehrbenutzerbetrieb (Terminalserver-Mode mit *Remote Desktop-Protokoll* RDP):
  Dynamic Fair-Share Scheduling mit **Gang-Scheduling** für alle Threads einer Benutzer-Session

Zusammenfassung: Prozess-Scheduling

- Wo und wann findet Scheduling statt?
- unterbrechende / nicht-unterbrechende Algorithmen
- Ziele von Scheduling-Algorithmen
- Stapelverarbeitungssysteme
  - First-Come First-Served
  - Shortest Job First
  - Shortest Remaining Time Next
- Interaktive Systeme
  - Round-Robin („Zeitscheibenverfahren“)
  - Prioritätenbasiertes Scheduling
    - statisch / dynamisch
    - mit mehreren Prioritätsklassen / Warteschlangen
- Echzeitsysteme
  - Rate Monotonic Scheduling (für periodische Prozesse)
  - Earliest Deadline First
- UNIX / Windows / JAVA - Scheduling
- Multiprozessorscheduling

Ende des 2. Kapitels: Was haben wir geschafft?

2. Prozesse

- 2.1 Das Prozessmodell
- 2.2 Das Threadmodell
- 2.3 Prozess-Scheduling