

## Kapitel 3

# Prozesse: Beschreibung, Verwaltung und Steuerung



## 3.1 Prozessbeschreibung

- Betriebssysteme bestehen aus Prozessen und arbeiten mit Prozessen (s. Kapitel 2).
- Ein Prozess ist einfach ein Programm in Ausführung, inklusive des aktuellen Werts des Befehlszählers, der Registerinhalte und der Belegungen der Variablen.
- Um einen Prozess zu verwalten, muss das BS folgendes wissen:
  - wo sich der Prozess befindet
  - welche Attribute der Prozess hat: Kennung (ID), Zustand, Speicherort
- **Prozessabbild** im BS enthält:
  - Benutzer-Programm (das der Prozess ausführt) und -Daten
  - Systemstapel (Stack) für Prozeduraufrufe und Parameterübergabe
  - Attribute des Prozesses in einem Prozesskontrollblock (PCB),  
vgl. nächste Folie

# Prozesskontrollblock (PCB)

- Ein Prozess ist repräsentiert durch eine spezielle Datenstruktur, den Prozesskontrollblock (PCB), der im Kern platziert wird und alle relevante Informationen, sog. **Attribute** enthält, z. B.
  - **Prozesscharakteristika:**  
Prozesskennung, Name des Programms
  - **Zustandsinformation:**  
Befehlszähler, Stapelzeiger, Registerinhalte
  - **Verwaltungsdaten:**  
Priorität, Rechte, Statistikdaten
- In größeren Systemen gibt es Hunderte von Prozessen, daher muss mit PCB-Datenstrukturen effizient umgegangen werden
- Es gibt verschiedene Möglichkeiten, je nach Art und Einsatz des Betriebssystems: PCBs einzeln, als Array, Baum, Tabelle, etc.<sup>1</sup>

---

<sup>1</sup>Z.B.: <https://github.com/Stichting-MINIX-Research-Foundation/minix/.../proc.h>

# Prozesse: Vordegrund/Hintergrund, Daemons, etc.

- Beim Booten eines BS werden i.d.R. mehrere Prozesse erzeugt
- Einige davon laufen *im Vordegrund*, d.h. sie kommunizieren mit den Benutzern und führen Arbeiten für diese aus (Kopplung an Terminal/Session/IO)
- Andere Prozesse laufen *im Hintergrund* – sie erfüllen spezielle Funktionen und lassen sich nicht bestimmten Benutzern zuordnen (z.B.: warten/empfangen von e-mails)
- Hintergrund-Prozesse, die Anfragen nach Emails, Webseiten, Drucken u.s.w. bearbeiten, heißen *Daemons*. Große BS haben dutzende davon.
- Auch nach dem Systemstart werden i.d.R. noch Prozesse erzeugt. Laufende Prozesse führen Systemaufrufe aus, um neue Prozesse zu erzeugen, die ihnen bei der Arbeit helfen.
- In interaktiven Systemen starten Benutzer neue Prozesse durch Eingeben eines Kommandos oder das (Doppel-)Klicken eines Icons

# Beispiele: Erfragen der Prozesse

- Linux-Prozesslisting mit ps:  
    `ps -ejlH | less, ps -elF | grep ...`  
    oder `top`
- Windows Task Manager (Strg+Alt+Entf)

# Statische und Dynamische Systeme

- **Statische Betriebssysteme**

Alle Prozesse sind vorher bekannt und statisch definiert.

- Prozesse werden „am Schreibtisch“ definiert. Die PCBs werden als Programmvariablen vereinbart.
- Bsp.: Einfache Systeme, wie z.B. Mikrowellenherd-Steuerung.
- Die Prozesse und ihre PCBs werden beim Systemstart *einmalig* erzeugt.

- **Dynamische Betriebssysteme**

Die Prozesse werden mittels Kernoperationen erzeugt / gelöscht.

```
create_process(id, initial_values)
// Anlegen des Prozesskontrollblocks mittels Kernoperation
// Initialisierung des Prozesses
delete_process(id, final_values)
// Rueckgabe der Endwerte
// Loeschen des Kontrollblocks mittels Kernoperation
```

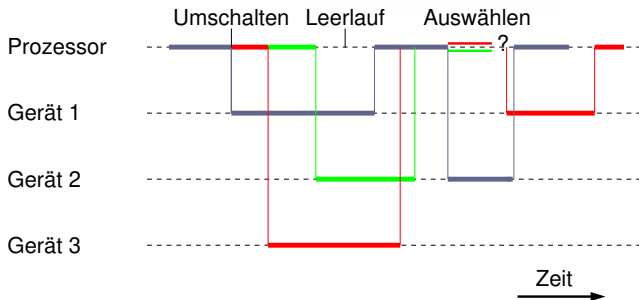
## 3.2 Prozesse und Virtuelle Prozessoren

- Im allgemeinen Fall hat man mehr Prozesse als (physikalische) Prozessoren, was zu einem Wettbewerb zwischen den Prozessen führt
- Damit alle Prozesse auf einem Prozessor vorankommen, schaltet der Prozessor zwischen den Prozessen: sog. **Prozessumschalten**
- Prozessor ist ein aktives Betriebsmittel: er „zieht“ den Prozess an sich, während ein passives BM dem Prozess zugewiesen wird
- Zur Vereinfachung, wird das Umschalten vom Benutzer „versteckt“: jeder Prozess erhält einen sog. virtuellen Prozessor:
  - **realer Prozessor**: aktives Betriebsmittel zur Programm Ausführung
  - **virtueller Prozessor**: eine kontinuierlich scheinende Folge von Nutzungsabschnitten realer Prozessoren zur Programmausführung
- Diese idealisierte Sicht erlaubt es, von einem *Starten*, *Anhalten*, *Warten* oder *Beenden* eines Prozesses zu sprechen, obwohl der ausführende reale Prozessor tatsächlich weder wartet noch anhält
- Analogie: Mensch (Prozessor) führt Aktivitäten (Prozesse) gemäß Anleitungen (Programmen) aus, z.B. Kochen, Hilfe leisten, etc.

# Prozessumschaltung

## Prozessumschaltung (Prozesswechsel, *process switching*)

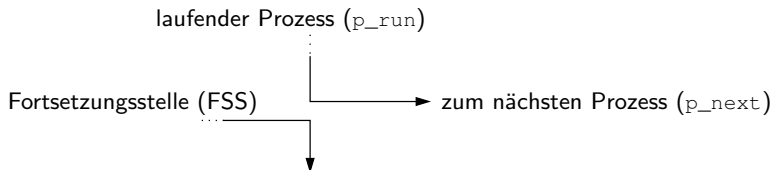
- Prozessumschaltung bedeutet, dass der Prozessor die Bearbeitung des aktuellen Prozesses unterbricht und stattdessen die Bearbeitung eines anderen Prozesses fortführt/anfängt.
- Prozesswechsel bedeutet also einen Übergang von einer Befehlsfolge (Programm) in eine andere.
- Beispiel: drei Prozesse (grau, rot, grün), drei Geräte.  
Wartet ein Prozess auf ein Gerät, wird ein anderer fortgesetzt.





# Umschalten durch Sprung

- Einfachster Fall: Das Umschalten ist „einprogrammiert“ direkt in die Prozesse – es wird direkt zum anderen Prozess gesprungen.
- Eine in aufeinanderfolgenden Speicherzellen stehende Befehlsfolge wird damit verlassen und evtl. später fortgesetzt
- Damit man später die Arbeit am unterbrochenen Prozess wieder korrekt weiterführen kann, muss man sich die Stelle merken.
- Eine Umschaltstelle besteht also mindestens aus
  - Fortsetzadresse (wo wurde die Arbeit unterbrochen)
  - Sprungbefehl (wo soll weitergemacht werden)

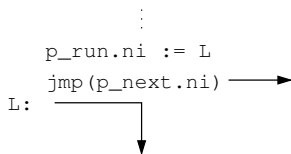


# Umschalten allgemeiner

- Das Umschalten durch direkten Sprung ist wenig flexibel und daher nur in speziellen einfachen Fällen anwendbar.
- I. A. wird das Umschalten wesentlich aufwendiger sein, da folgende wichtige Aspekte beachtet werden müssen
  - **Der Umgang mit Fortsetzstelle:** man weiß oft nicht, von wo aus man wieder zu dem unterbrochenen Prozess zurückkehrt,
  - **Die Auswahl des nächsten Prozesses:** der nächste Prozess, auf den man umschaltet, ist nicht immer der gleiche,
  - **Der Umgang mit dem Prozessorinhalt:** der Prozessor enthält Teile der Prozessbeschreibung, die nicht verloren gehen dürfen.
- Alle drei Aspekte sind gestaltbar und beeinflussen dadurch die Prozessumschaltzeit und somit die Gesamtleistung des Systems
- Wir betrachten im Folgenden diese drei Aspekte genauer

# Merken der Fortsetzstelle

Bevor umgeschaltet wird, merkt man sich die Adresse des nächsten auszuführenden Befehls in einer dafür vorgesehenen Variable `ni` (*next instruction*) des Prozesskontrollblocks.



# Auswahl des nächsten Prozesses

- Bisher gingen wir davon aus, dass feststeht, auf welchen Prozess umgeschaltet wird.
- Meistens ist dieser Prozess jedoch nicht konstant, sondern wird erst zum Zeitpunkt des Umschaltens ausgewählt:
- Mögliche Kriterien:
  - Nummer des Prozesses (z. B. zyklisches Umschalten)
  - Ankunftsreihenfolge (wer früher bereit ist)
  - Priorität (Dringlichkeit)
    - konstant
    - veränderlich
- Die Auswahl des nächsten Prozesses beeinflusst die Verteilung der Prozessorleistung auf die Prozesse und damit die Systemleistung (Performance).

## Auswahl des nächsten Prozesses (Forts.)

Sei  $\text{PSelect}()$  eine Funktion, die gemäß einem gegebenen Kriterium den nächsten Prozess auswählt.

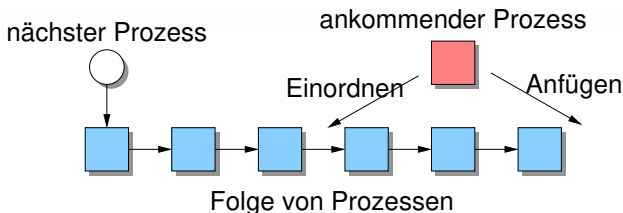
```

      p_next := PSelect()
      p_run.ni := L
      jmp(p_next.ni)
L:

```

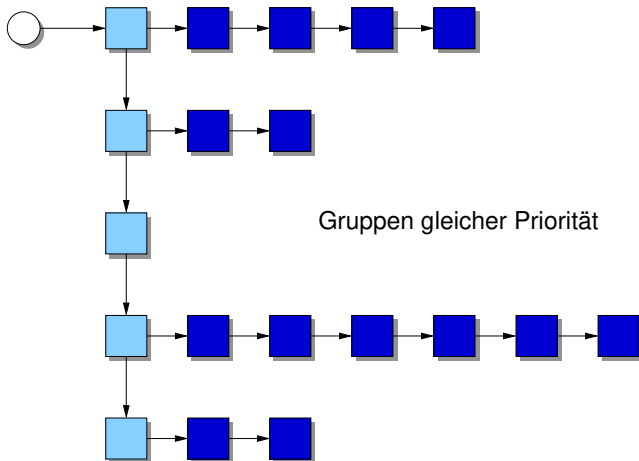
# Auswahl des nächsten Prozesses (Forts.)

- Das Auswahlproblem kann z. B. so gelöst werden, dass die Prozesse bezüglich der Ablaufreihenfolge total geordnet werden.
- Der bzgl. dieser Ordnung erste Prozess wird dann als nächster ( $p_{\text{next}}$ ) aufgegriffen. Eventuell neu ankommende Prozesse werden gemäß der Ordnung eingefügt.
- Die Festlegung der Prozessreihenfolge wird *scheduling* genannt. Spezielle Scheduling-Algorithmen behandeln wir später.



# Auswahl des nächsten Prozesses (Forts.)

- Verwendet man Prioritäten, so kann die Ordnung auch zweidimensional organisiert sein.



- Prozesse benutzen Rechenregister des Prozessors zur Ablage von Zwischenergebnissen. Beispiel:
  - <https://de.wikipedia.org/wiki/ARM-Architektur#Registersatz>
  - <http://infocenter.arm.com/...>
- Beim Prozesswechsel durch Sprung würden sie verloren gehen.
- Die „Sprung-Lösung“ kann also nur eingesetzt werden, wenn
  - die Inhalte der Rechenregister im alten Prozess nicht mehr benötigt werden, oder
  - der neue Prozess keine gültigen Registerinhalte erwartet.
- Diese Einschränkungen sind in der Praxis meist unrealistisch.



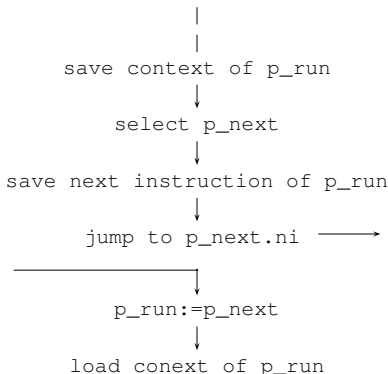
# Prozesskontext

- Außer dem Befehlszähler enthält der Prozessor in seinen Registern eine Menge weiterer prozessspezifischer Daten:
  - Inhalte von Rechenregistern, Indexregistern etc., die den **Zustand** des Prozesses (der Programmbearbeitung) repräsentieren.
  - Inhalte von Adressregistern, Segmenttabellen, Unterbrechungsmasken, Zugriffskontrollinformation etc., die die **Ablaufumgebung** des Prozesses darstellen. Beispiel:  
<https://github.com/Stichting-MINIX-Research-Foundation/.../proc.h>
- Alles zusammen, also die gesamte im Prozessor abgelegte prozessspezifische Information, wird als der **Kontext** des Prozesses (*process context*) bezeichnet.
- Dieser Prozesskontext muss beim Umschalten gerettet und beim Fortsetzen des Prozesses wieder restauriert werden.
- Sofern Daten konstant und im PCB verfügbar sind, kann das Retten entfallen (z. B. von der Segmenttabelle des Prozesses).

- Der Kontextwechsel ist der aufwendigste Teil des Umschaltens.
- Um ihn zu beschleunigen, kann von der Prozessor-Hardware Unterstützung angeboten werden:
  - durch spezielle Befehle, um einen kompletten Registersatz aus dem Prozessor in den Speicher zu schreiben und umgekehrt,
  - durch Bereitstellung mehrerer Registersätze im Prozessor(z. B. 8): beim Umschalten muss u. U. nur das Register geändert werden, das die Nummer des gültigen Registersatzes angibt.
- Prozesswechsel ist dann schnell, wenn nur die Rechenregister umgelaufen werden müssen, aber die Adressierungsumgebung dieselbe bleibt
- Dies ist der Fall bei Threads in einem Adressraum.

# Umschalten (als offene Befehlsfolge)

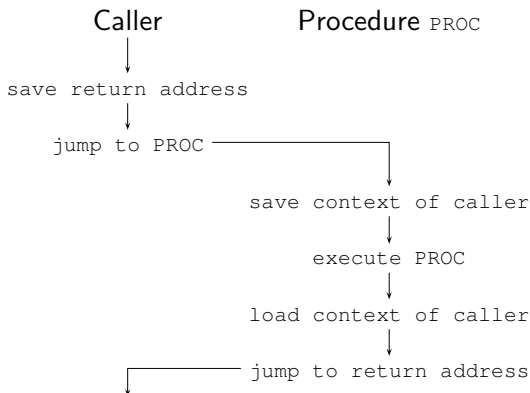
Das Umschalten mit Kontextwechsel hat nun die folgende Form:



Das Umschalten kann auch von einer Bedingung (z. B. binären Variable) abhängig sein: dann spricht man von bedingtem Umschalten

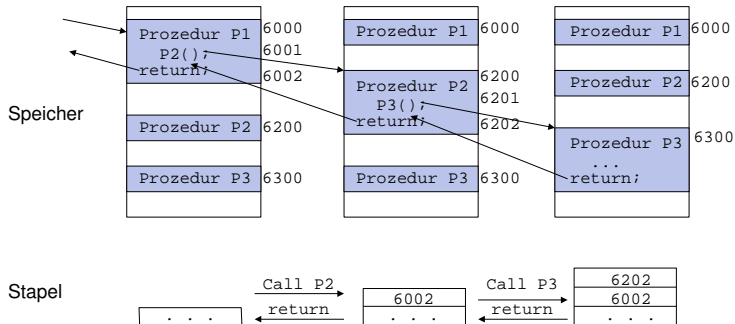
# Prozeduraufrufe

- Das Vorgehen bei der Prozessumschaltung ähnelt dem bei einem Prozeduraufruf
- Prozeduraufruf:



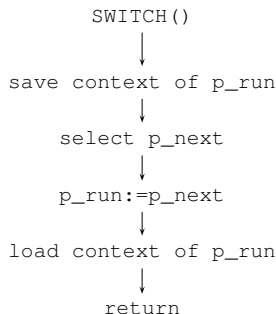
# Prozeduraufrufe (Forts.)

- Rücksprungadressen von Prozeduren werden auf einem Stapel (Stack) abgelegt
- Jeder Prozess hat einen eigenen Stapel für seine Prozeduraufrufe



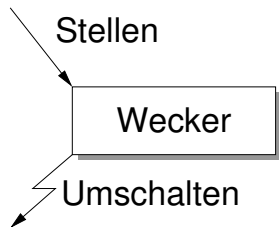
# Umschalten als Prozedur

- Wenn es viele Umschaltstellen gibt, lohnt es sich, das Umschalten als eine Prozedur (wir nennen sie SWITCH) zu organisieren
- So entfallen die Ablage der Fortsetzstelle ( $ni$ ) und der Sprung zum nächsten Prozess: Dies ist dann Bestandteil der Sprung- bzw. der Return-Operation der Prozedur
- Aufruf von SWITCH legt die Fortsetzstelle auf den Stapel von p\_run ab
- `p_run := p_next` bewirkt, dass nun der Stapel von p\_next verwendet wird
- `return` springt an die Fortsetzstelle auf dem neuen (!) Stapel
- Dieser Prozedursprung ist etwas ungewöhnlich: ein Prozess ruft eine Prozedur auf, aus der später ein anderer Prozess zurückkehrt.



# Automatisches Umschalten

- In vielen Fällen ist es nicht möglich oder nicht sinnvoll, Umschaltstellen explizit in die Prozesse einzubauen.
- Wünschenswert wäre ein **automatisches Umschalten**.
- Dazu wird oft eine **Intervalluhr** oder Wecker (*timer*) benutzt – eine Hardware-Einrichtung (E/A-Gerät) mit den folgenden Funktionen:
  - Vorgabe einer Frist („Wecker“ stellen)
  - Unterbrechung bei Fristablauf („Wecken“)



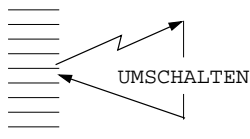
# Automatisches Umschalten (Forts.)

- Beim automatischen Umschalten können die Programme unverändert bleiben.
- Das Umschalten wird gewissermaßen von außen ausgelöst und kann zu jedem beliebigen Zeitpunkt stattfinden (Unterbrechungen dürfen nicht abgeschaltet sein.)

Freiwilliges Umschalten



Automatisches Umschalten

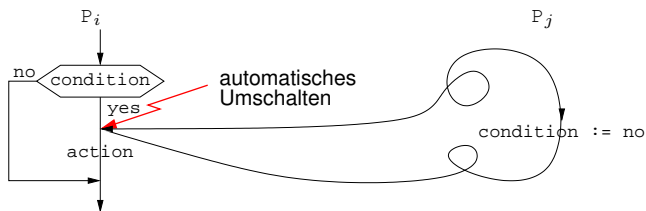


- Es kommt zu einer **verzahnten Ausführung** der Programme



### 3.3 Verzahnte Ausführung und kritischer Abschnitt

- Potentielles Problem bei Verzahnung: Es gibt Programmstellen, an denen eine Aktion abhängig von einer Bedingung durchgeführt wird.
- Aber: Man kann nicht ausschließen, dass zwischen der Bedingungsabfrage und der Aktion selbst ein Umschalten stattfindet und vor der Rückkehr zum unterbrochenen Prozess ein anderer Prozess die Bedingung ändert, (z. B. so etwas wie "Druckausgabe fertig?"):



- Das Testen+Aktion ist ein **kritischer Abschnitt**, der nicht unterbrochen werden darf, sonst ist fehlerhaftes Verhalten möglich


# Kern als kritischer Abschnitt

- **Regel für kritische Abschnitte:** Wenn ein Prozess einen kritischen Abschnitt betritt, darf kein weiterer Prozess eine dazu in potentiellm Konflikt stehende Aktion ausführen.
- Man spricht von **gegenseitigem Ausschluss** (*mutual exclusion*).
- Im BS-Kern müssten alle möglichen Konfliktstellen identifiziert und entsprechend geschützt werden.
- Da im Kern viele kritische Abschnitte vorhanden sind, kann man einfach den **gesamten Kern als kritischen Abschnitt** auffassen: Kernoperationen dürfen nur „an einem Stück“ ausgeführt werden
- **Vorsicht:** In einem Mehrprozessorsystem kann es trotz Unterbrechungsverbot zu einer verzahnten Ausführung von Kernoperationen kommen, wenn sie simultan auf zwei Prozessoren bearbeitet werden, deren Speicherzugriffe verzahnt ablaufen.
- Für diesen Fall müssen wir den gegenseitigen Ausschluss am Kern durch eine sogenannte **Kernsperr**e erzwingen.

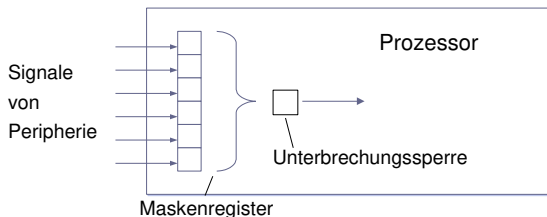
# Kernausschluss

- Die Realisierung des Kernausschlusses ist davon abhängig, ob Unterbrechungen möglich sind und ob mehrere Prozessoren vorhanden sind.
- Daher haben wir vier Fälle zu unterscheiden:
  - Fall 1: Einprozessorsystem ohne Unterbrechungen
  - Fall 2: Einprozessorsystem mit Unterbrechungen
  - Fall 3: Mehrprozessorsystem ohne Unterbrechungen
  - Fall 4: Mehrprozessorsystem mit Unterbrechungen
- Der **Fall 1** erfordert keinerlei spezielle Sicherungsmaßnahmen, da es keinen Anlass gibt, eine Kernoperation zu verlassen

kernel operation



## Fall 2: Einprozessorsystem mit Unterbrechungen



- Die Kernoperation kann in diesem Fall durch ein `disable interrupt` und `enable interrupt` geklammert werden.

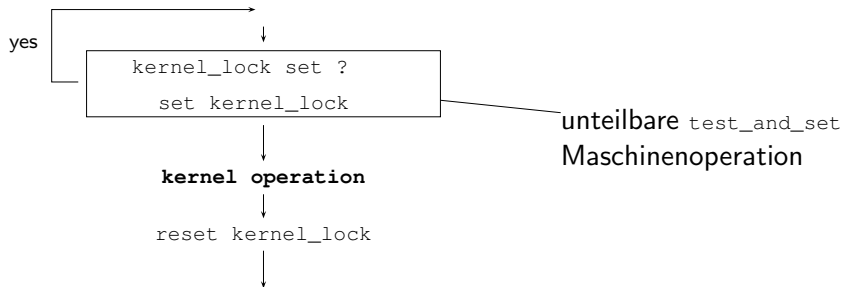
```
      |  
  disable interrupt  
      |  
  kernel operation  
      |  
  enable interrupt  
      |  
      ↓
```

## Fall 3: Mehrprozessorsystem ohne Unterbrechungen

- Das Problem besteht (wiederum) darin, dass zwischen der Auswertung einer Bedingung und der davon abhängigen Aktion andere Operationen die Bedingungsvariable ändern können.
- Man könnte Sperroperationen einführen, die jedoch selbst wiederum kritische Abschnitte darstellen, die unteilbar ablaufen müssen, d. h. sind unter gegenseitigen Ausschluss zu stellen, usw. . . .
- Um diese Rekursion aufzulösen, benutzt man die von der Hardware bereitgestellten **unteilbaren Operationen**, die den Wert einer Bedingungsvariable abfragen und sie gleichzeitig setzen
- Derartige Maschinenbefehle haben z. T. unterschiedliche Namen (compare-and-swap, fetch-and-add, . . .) und auch unterschiedliche Semantik.
- Wir werden die Operation `test_and_set` verwenden:
  - `test_and_set(reg, x) := {load reg, x; x:=1}`
  - Unabhängig von ihrem Wert wird die Variable x auf 1 gesetzt, der alte Wert wird im Register getestet und „ja“ (x=1) oder „nein“ (x=0) zurückgegeben

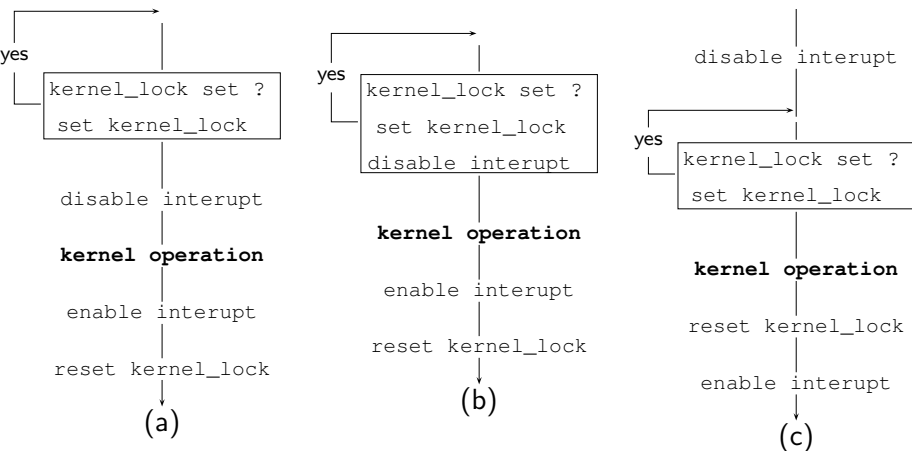
## Fall 3: Unteilbare Maschinenoperationen

- Bei Verwendung des `test_and_set`-Befehls: Ist die Kernsperre belegt, so wird in einer Mini-Schleife der Wert immer wieder abgefragt.
- Man nennt dies „aktives Warten“ (*busy waiting*).
- Eine solche Sperre wird auch als *spin lock* bezeichnet.
- Aktives Warten ist eine Verschwendung von Rechenkapazität, wird aber in BS oft toleriert, da Kernoperationen relativ kurz sind.



## Fall 4: Mehrprozessorsystem mit Unterbrechungen

- Hier müssen nun beide Techniken, d. h. Unterbrechungssperre und Kernsperre gemeinsam zum Einsatz kommen.
- Dazu wollen wir die folgenden drei Lösungen diskutieren:



# Diskussion der Lösungen

- **Lösung (a):**

Beachte: Eine Unterbrechungsbehandlung ist eine Kernoperation, für deren Durchführung die Kernsperrung ebenfalls benötigt wird!

Gibt es nun direkt nach dem Setzen der Kernsperrung eine Unterbrechung, so wird in der Unterbrechungsbehandlung vergeblich versucht, die Kernsperrung zu setzen. Der Prozess würde an dieser Stelle „hängen“.

- **Lösung (b):**

Ideal wäre deshalb eine Operation, die unteilbar sowohl die Kernsperrung als auch die Unterbrechungssperre setzt. Leider findet man solche Operationen bei keinem Prozessor.

- **Lösung (c):**

Es muss daher zuerst die Unterbrechungssperre gesetzt werden und dann erst die Kernsperrung. Lösung (c) ist die einzig korrekte

**Beachte:** Unterbrechungssperre ist zwar nützlich im Kern des BS, eignet sich jedoch nicht für gegenseitigen Ausschluss in Benutzerprozessen



## 3.4 Prozesszustände: Motivation

- Wenn ein Prozess nicht fortgesetzt werden kann, wird der Prozessor freigegeben.
- In diesem Fall wird auf einen anderen Prozess umgeschaltet. Nun kann dieser Prozess u. U. auch nicht fortsetzbar sein, weil er z. B. ebenfalls auf das Ende einer E/A-Operation wartet. Deshalb würde er den Prozessor sofort wieder freigeben.
- So könnte man eine Reihe von Prozessen „ausprobieren“, bis man irgendwann auf einen stößt, der tatsächlich fortsetzbar ist.
- Um diese Suche nach einem fortsetzbaren Prozess zu beschleunigen, fasst man die Prozesse nach ihrem Zustand (fortsetzbar, nicht fortsetzbar) zu Teilmengen zusammen:
  - Zustand **rechnend** (running): Prozesse, die gerade auf einem Prozessor bearbeitet werden
  - Zustand **bereit** (ready): Prozesse, die zwar fortsetzbar sind, aber gerade nicht bearbeitet werden
  - Zustand **wartend** (waiting): Prozesse, die nicht fortsetzbar sind, weil sie auf das Eintreten einer Bedingung warten

# Prozesszustände aus “PS“(Linux):

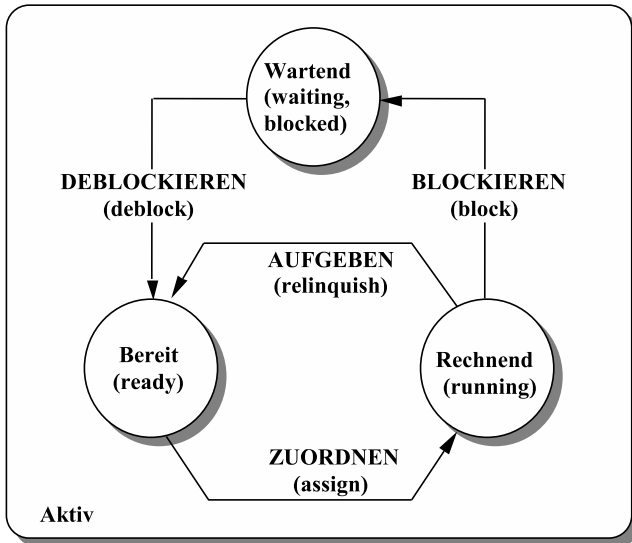
## PROCESS STATE CODES

Here are the different values that the **s**, **stat** and **state** output specifiers (header "STAT" or "S") will display to describe the state of a process:

D	uninterruptible sleep (usually IO)
R	running or runnable (on run queue)
S	interruptible sleep (waiting for an event to complete)
T	stopped by job control signal
t	stopped by debugger during the tracing
W	paging (not valid since the 2.6.xx kernel)
X	dead (should never be seen)
Z	defunct ("zombie") process, terminated but not reaped by its parent

**Abbildung:** Prozesszustände aus dem PSManual

# Prozesszustände und Übergänge (in einem statischen System)

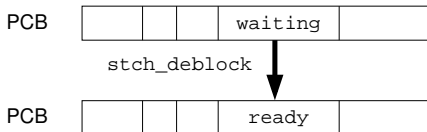


# Zustandswechseloperationen

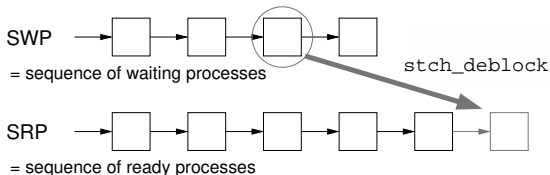
- Für alle **Zustandsübergänge** (*state changes*) sind entsprechende Operationen im Kern vorgesehen:
  - **Aufgeben** (*relinquish*)  
Freiwilliges Umschalten auf einen anderen Prozess. Der bisher rechnende Prozess bleibt fortsetzbar, d. h. geht in den Zustand „bereit“ über.  
Bsp.: Freiwilliges Aufgeben ist in Java mit `Thread.yield()` möglich.
  - **Zuordnen** (*assign*)  
Aufgreifen des nächsten Prozesses aus der „Bereit-Menge“ zur Fortsetzung auf dem Prozessor.
  - **Blockieren** (*block*)  
Verlassen des Prozessors wegen Nichterfüllung einer Bedingung (bedingtes Umschalten): Der Prozess geht in den Zustand „wartend“ bzw. „schlafend“ über.
  - **Deblockieren** (*deblock*)  
Ist die Bedingung erfüllt, auf die der blockierte Prozess gewartet hat, so wird er durch diesen Übergang wieder in die Menge der bereiten Prozesse eingefügt.

# Zustandswechseloperationen: die Durchführung

- Die Durchführung der Übergänge hängt davon ab, wie die Prozesszustände realisiert sind.
- Neben der Änderung des Zustands (als Operation auf der PCB-Datenstruktur) muss auch das Umschalten stattfinden.
- Beispiel: `stch_deblock` („stch“ steht für „state change“)
  - Prozesszustand als Attribut im PCB



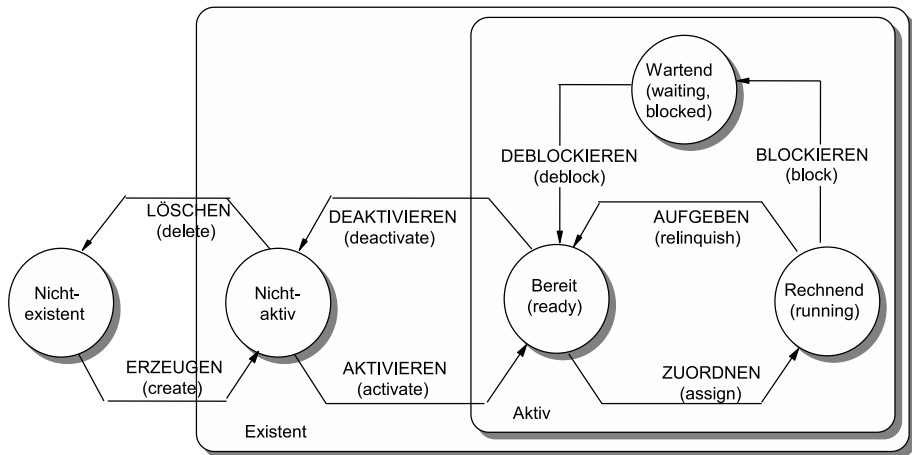
- Prozesszustand als Zugehörigkeit zu einer Liste



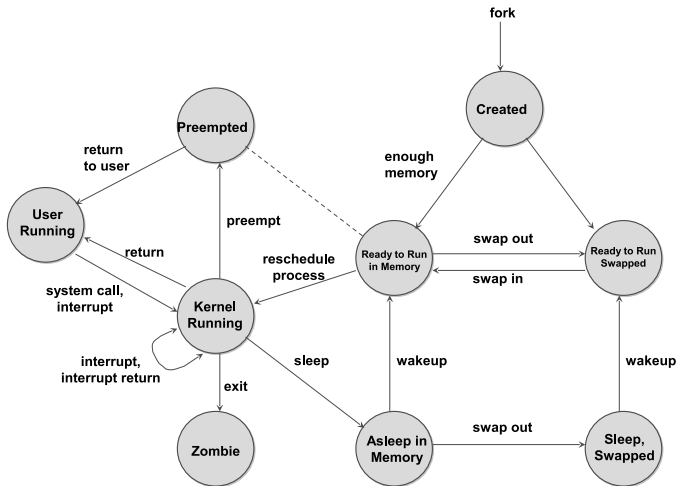
# Zustände und Übergänge (in einem dynamischen System)

- In dynamischen Systemen ist die Menge der am Geschehen teilnehmenden Prozesse variabel.
- Dynamische Systeme können auf zwei Weisen entstehen:
  - Aktivieren / Deaktivieren
    - Ein Prozess kann zwar definiert sein (es existiert ein PCB), auch seine Programm- und Datenbereich sind vielleicht schon vorhanden, aber der Prozess ruht, d. h. er ist nicht aktiv.
    - Wir unterscheiden zwischen aktiven und nichtaktiven Prozessen.
    - Übergänge zwischen diesen Zuständen sind durch die Operationen **Aktivieren** und **Deaktivieren** möglich.
  - Erzeugen / Löschen
    - Prozesse existieren bei Systemstart noch nicht, sondern müssen explizit erzeugt und ggf. gelöscht werden.
    - Dafür sind die Operationen **Erzeugen** und **Löschen** vorgesehen.

# Zustandsdiagramm vollständig (Schema)



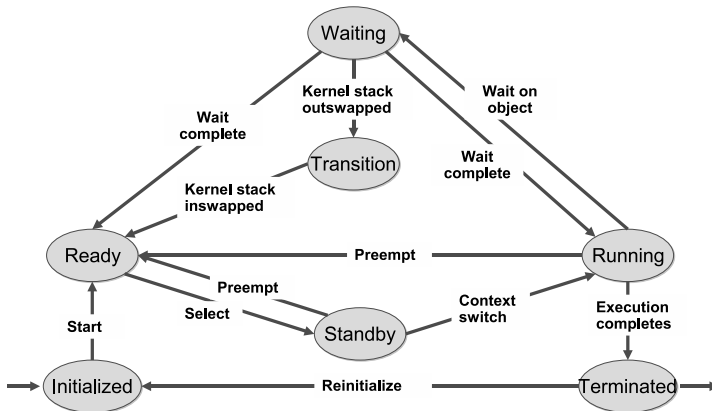
# Schema-Ausprägung: Unix



Konkrete Ausprägung des vorherigen Schemas in Unix: feiner untergliedert, z. B. Unterscheidung zwischen Prozessen deren Daten im Hauptspeicher liegen (*Ready to Run in Memory*) oder auf die Festplatte ausgelagert sind (*Ready to Run Swapped*)



# Schema-Ausprägung: Windows NT

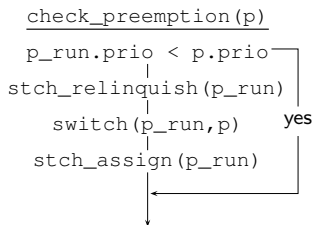


## 3.5 Prioritäten und Verdrängung

- In der bisherigen Betrachtung bleibt ein Prozess in Bearbeitung, bis:
  - er freiwillig den Prozessor aufgibt,
  - durch eine Uhr-Unterbrechung ein Umschalten erzwungen wird,
  - er wegen einer nichterfüllten Bedingung nicht weiterarbeiten kann.
- In vielen Einsatzgebieten sind nicht alle Prozesse von gleicher Wichtigkeit/Dringlichkeit, was zur Verwendung von Prioritäten führt.
- **Prioritäts-Regel:** Zu jedem Zeitpunkt wird der Prozess mit der höchsten Priorität bearbeitet.
- **Konsequenz:** BS wartet nicht, bis eine der drei obigen Situationen eintritt, sondern führt ein Umschalten sofort durch, sobald ein Prozess höherer Priorität auftaucht, d. h. „bereit“ wird.
- Man sagt, der rechnende Prozess wird durch den dringlicheren Prozess verdrängt (*preempted*).

# Verdrängungsprüfung

- Die Prioritäts-Regel besagt, dass keiner der bereiten Prozesse eine höhere Priorität besitzen darf als einer der rechnenden.
- Wenn wir annehmen, dass die Prioritäts-Regel eingehalten wurde und die Prioritäten konstant sind, so kann eine Verletzung nur stattfinden, wenn ein neuer Prozess die Bereitmenge betritt.
- Nach dem Zustandsdiagramm gibt es drei Übergänge in die Bereitmenge:
  - Aufgeben (*relinquish*)
  - Deblockieren (*deblock*)
  - Aktivieren (*activate*)
- Diese Operationen prüfen mit `check_preemption()`, ob der nun bereit gewordene Prozess eine höhere Priorität besitzt als der rechnende. Ist das der Fall, so wird auf den dringlicheren Prozess umgeschaltet.

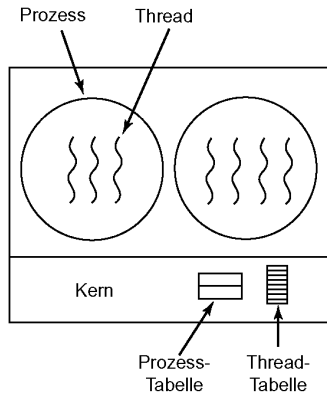
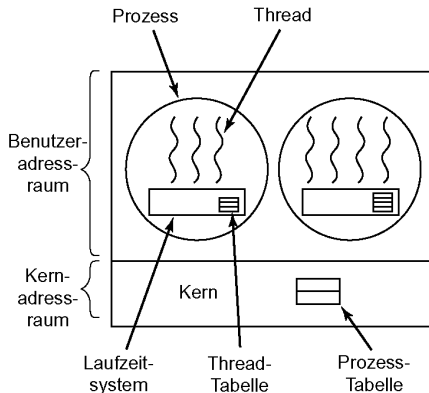


## 3.6 Prozesse vs. Threads

- Prozess – Programm in Ausführung, aber auch eine Gruppe dazu gehörender Ressourcen (gemeinsam verwaltet):
  - Adressraum (Quellcode/Befehlszähler und Daten/Register/ Stack)
  - geöffnete Dateien (Files)
  - erzeugte Kindprozesse, etc.
- Thread – ein “leichtgewichtiger Prozess”, zusammen mit:
  - Befehlszähler
  - Register, die lokale Variablen enthalten
  - Stapel (Keller, Stack) für aufgerufene Prozeduren
- I.d.R. laufen mehrere Threads in einem Prozess, d.h. in demselben Adressraum
- *Multithreading* = mehrere Threads in einem Adressraum, um mehrere gleichzeitige Aktivitäten zu programmieren:
  - wegen der Struktur der Anwendung mit mehreren Aktivitäten;
  - um höhere Performanz zu erreichen: Überlappung durch Umschalten vom blockierten zum bereiten Thread, kleinerer Aufwand beim Erzeugen/Löschen als bei Prozessen.

# Realisierung von Threads in Betriebssystemen

- Zwei Möglichkeiten, Threads im BS zu realisieren: im Benutzer-adressraum oder im Kern (oder Hybride von beiden)



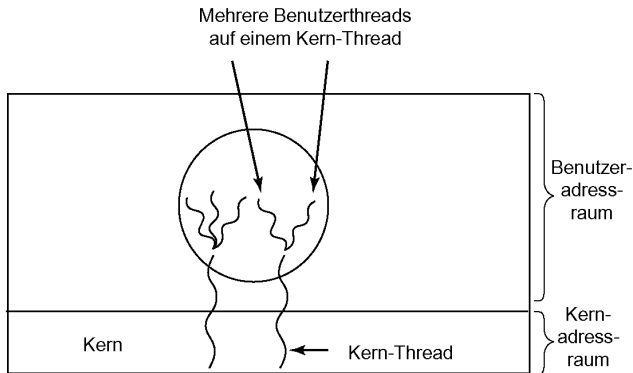
Quelle: Tanenbaum 2003.

# Zwei Realisierungs-Ansätze: Vergleich

- Erster Ansatz: Thread-Packet im Benutzeradressraum:
  - Der Kern weiss nichts von den Threads
  - Der Ansatz benötigt ein Laufzeitsystem (Prozeduren thread-create, thread-exit, etc.), das eine Thread-Tabelle verwaltet
  - Vorteil: Läuft auf jedem Betriebssystem, gute Performance
  - Nachteil 1: Wird ein Thread blockiert, muss der Kern den ganzen Prozess (alle Threads) blockieren, da er ja sie nicht einzeln sieht
  - Nachteil 2: Es gibt keine Unterbrechungen innerhalb eines Prozesses ⇒ wie sollte der Prozessor zwischen den Threads umschalten?  
Die Nachteile werden mit verschiedenen Tricks behoben.
- Zweiter Ansatz: Thread-Verwaltung im Kern:
  - Kein Laufzeitsystem und keine Thread-Tabelle in jedem Prozess nötig
  - Der Kern besitzt eine Thread-Tabelle für alle Threads
  - Nachteil: Systemaufrufe im Kern sind viel langsamer als Laufzeitsystemaufrufe
  - Vorteil: Beim Blockieren eines Threads kann ein weiterer im gleichen Prozess aktiviert werden

# Threads in BS: Hybride Methode

- Man versucht, die Vorteile beider o.g. Methoden zu verbinden
- Ein Weg: Verwendung von Kern-Threads, in jedem von diesen laufen evtl. mehrere Benutzer-Level-Threads
- Es gibt auch mehrere andere Ansätze



Quelle: Tanenbaum 2003.

# Prozesse & Threads in Programmiersprachen

- Moderne Programmiersprachen enthalten oft ein Prozess- oder Threadkonzept zur Formulierung von nebenläufigen Aktivitäten innerhalb von Programmen (z. B. Java Threads <sup>2</sup>)
- Es gibt auch Programmbibliotheken, die eine Programmiersprache um ein Prozesskonzept ergänzen (z. B. Pthreads für C).
- Es ergeben sich folgende Fragen:
  - Wie verhalten sich diese Prozesse zu den BS-Prozessen?
  - Wie unterstützt das Betriebssystem solche Prozesse?

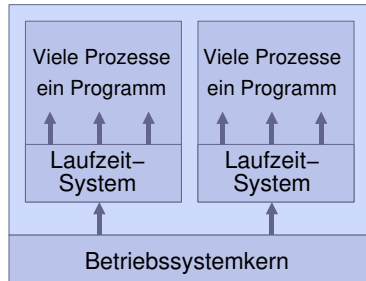
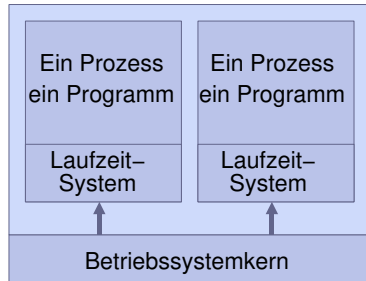
---

<sup>2</sup>Multithreading in Java wird in unserer VL „Multithreading und Networking im Java-Umfeld“ ausführlich behandelt



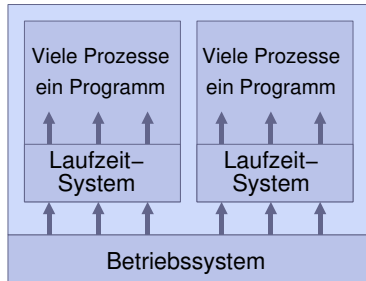
# Prozesse in BS und Programmiersprachen

1. Klassischer Mehrprogrammbetrieb:  
Unabhängige Prozesse in eigenen Adressräumen.  
Bsp.: Programmierung ohne Threads unter Unix
2. Programmiersprache mit Threads, ohne BS-Unterstützung: Threads sind für BS nicht sichtbar, es behandelt das Gesamtprogramm als einen Prozess.  
Bsp.: Java auf nicht-modernem BS

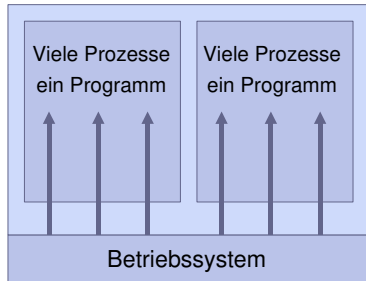


# Prozesse in BS und Programmiersprachen (Forts.)

3. Programmiersprachliche Prozesse werden 1:1 auf BS-Prozesse abgebildet.  
Bsp.: Java auf Unix



4. Die Programmiersprache unterstützt keine Prozesse. Programme bestehen aus BS-Prozessen, die sich einen Adressraum teilen.  
Bsp.: C mit der Pthreads-Bibliothek



# Prozessverwaltung – Zusammenfassung

Bis jetzt wurden folgende Teile des Kerns behandelt:

