Betriebssysteme

Prozesse und Threads

Literatur Verzeichnis

- Mandl, Peter; Grundkurs Betriebssysteme; 5.Aufl. 2020; Springer Verlag
- Baun, Christian, Betriebssysteme kompakt, 2.Aufl., Springer 2020
- Tanenbaum, Andrew; Moderne Betriebssysteme; 3. Aufl. 2009; Pearson Studium
- Silberschatz et al.; Operating System Concepts; 7.ed; John Wiley 2005
- Siegert, H.J., Baumgarten U.; Betriebssysteme; 5.Aufl. 2001; Oldenbourg Verlag

Gliederung

- Einführung und Abgrenzungen
- Prozessmodell
- Prozesskoordinierung
- Prozess-Lebenszyklus
- Threads

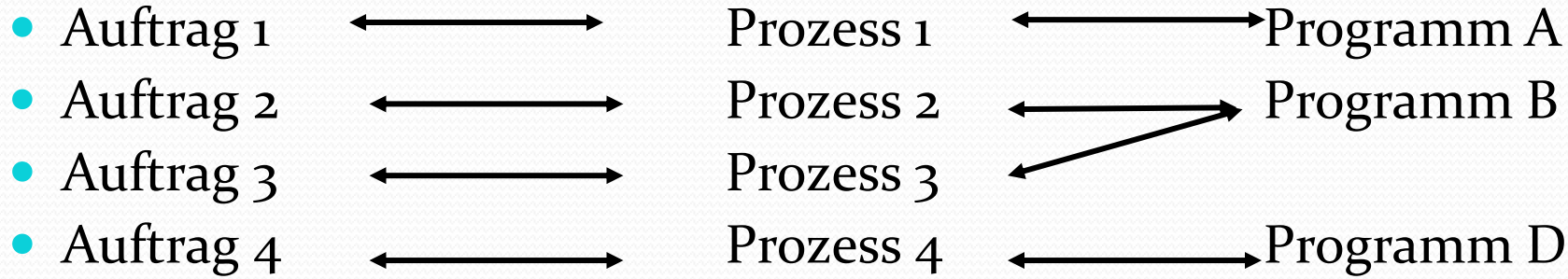
Einführung

- Das Prozess- und Threadmodell sind wesentliche Konzepte der Betriebssystementwicklung
- Prozess sind Betriebssystemmittel, die verwaltet werden müssen
- Threads werden je nach Implementierung vom Betriebssystem oder einer Laufzeitumgebung verwaltet.
- Die Informationen von Prozessen und Threads werden in PCB bzw. TCB verwaltet.

Aufgabe Prozesse

- Erläutern Sie folgende Begriffe und geben Sie dazu auch Beispiele an.
  - Auftrag, Prozess und Programm
  - Multitasking und Multiprocessing
  - Parallelität und Nebenläufigkeit
  - virtuelle Prozessoren.

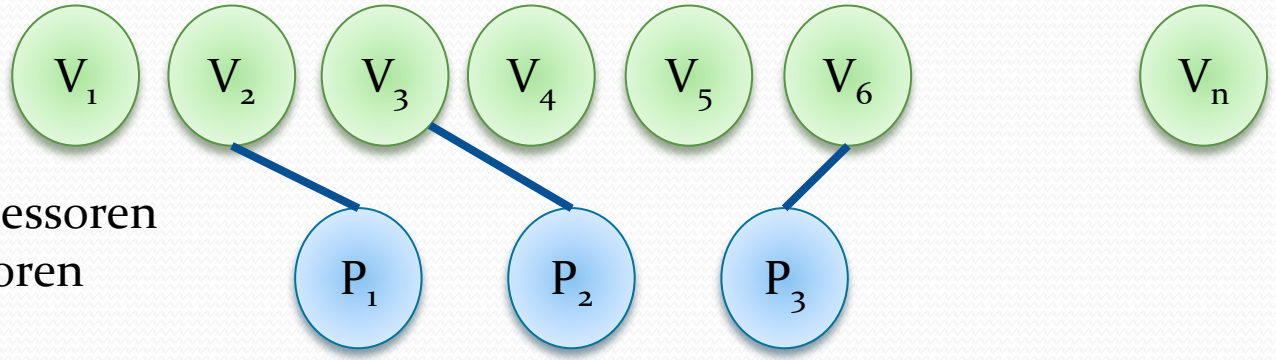
Prozess-Modell



- Auftrag ist der Auslöser einer Arbeitsvorgang für das Betriebssystem
- Prozess bezeichnet die gesamte Zustandsinformation eines laufenden Programms
- Jedes Programm wird in einem Prozess ausgeführt
- Programm ist eine statische Verfahrensvorschrift für die Verarbeitung
- Prozess ist konkrete Instanziierung eines Programms innerhalb eines Rechners

Virtuelle und reale Prozesse

- Das Betriebssystem ordnet im Multiprogramming jedem Prozess ein virtueller Prozessor zu
  - Bei echter Parallelarbeit wird jedem virtuellem Prozessor jeweils ein realer Prozessor zugeordnet
  - Im quasi-parallelen oder nebenläufigen Betrieb wird jedem realen Prozessor für eine gewisse Zeit ein virtueller Prozessor zugeordnet.
  - Eine Prozessumschaltung erfolgt nach einer vorgegebenen Strategie



V: virtuelle Prozessoren
P: reale Prozessoren

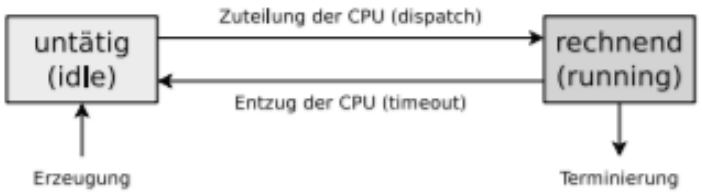
Prozesszustände

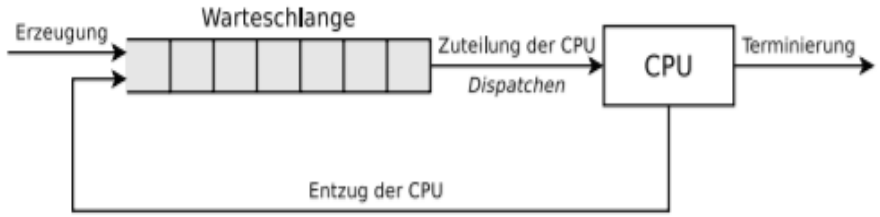
- Prozesse, die bearbeitet werden sollen, müssen verwaltet werden
- Es müssen Ressourcen zugeteilt werden
- Es müssen üblicherweise mehrere Zustände durchlaufen werden
- Die einzelnen Zustände modellieren das Verhalten im System

Prozesszustandsmodell

- **Prozesse warten …**
  - auf den Prozessor (*bereit*)
  - auf eine Nachricht (*blockiert*)
  - auf ein Zeitsignal (*blockiert*)
  - auf Daten des I/O-Geräts (*blockiert*)

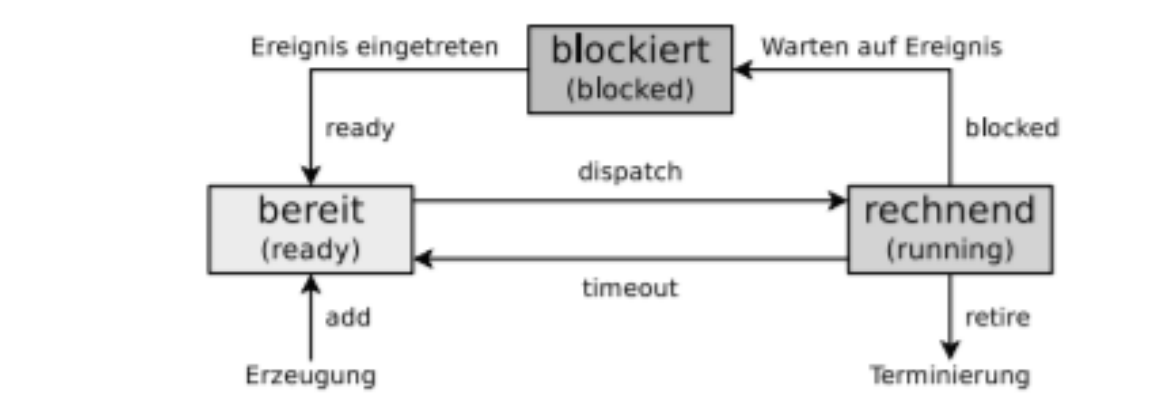
2 Prozesszustandsmodell





- 2 Zustandsmodell
  - Prozesse müssen im Zustand idle in einer Warteschlange gehalten werden.
  - Vorteil: Es ist einfach zu realisieren
  - Aber ein gewichtiger Nachteil Es geht davon aus, dass jeder Prozess sofort rechenfähig ist.
  - Daher trennen des untätigen Zustands in 2 Zustände

3 Prozesszustandsmodell

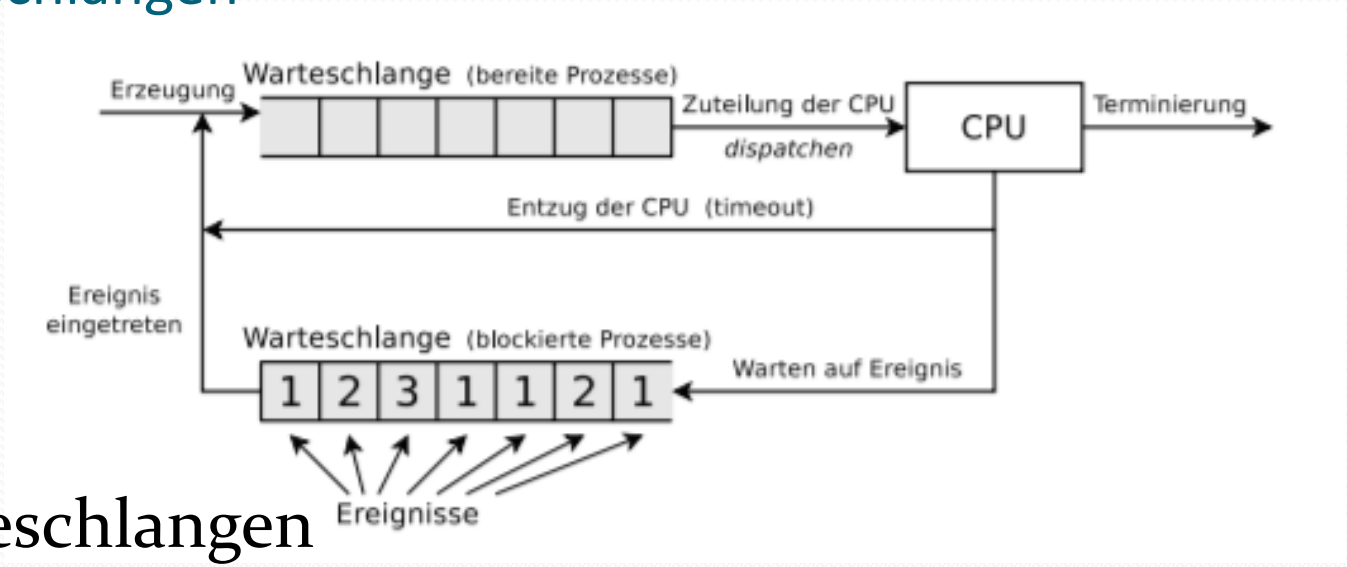


- 3 Zustandsmodell
  - Nun werden 2 Warteschlangen benötigt.

@Baun

3 Prozesszustandsmodell

2 Warteschlangen

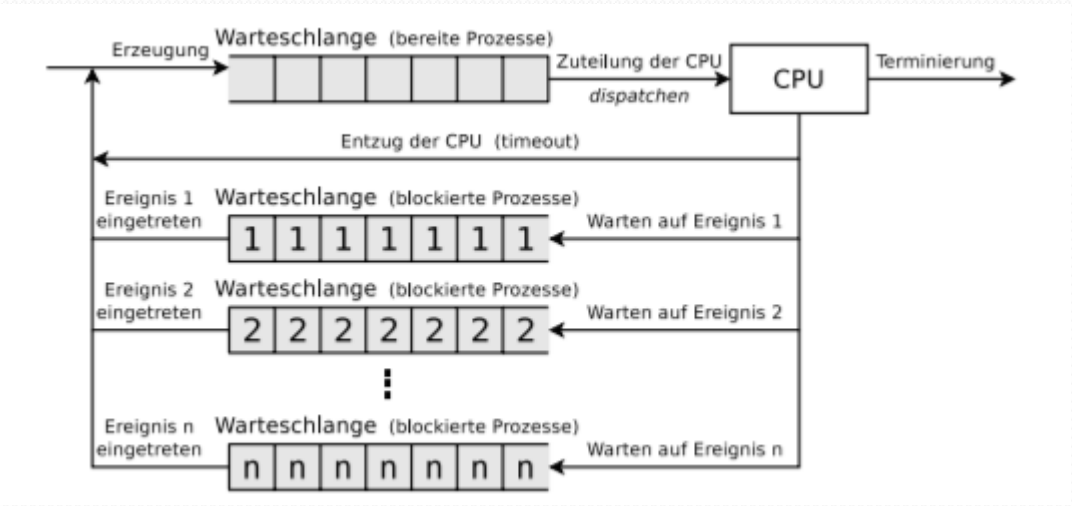


- Warteschlangen
  - Die CPU muss nun für die Prozesse jedes Mal überprüfen, ob das Ereignis für den anstehenden Prozess schon eingetreten ist.
  - Besser eine eigene Warteschlange für jedes Ereignis

@Baun

3 Prozesszustandsmodell

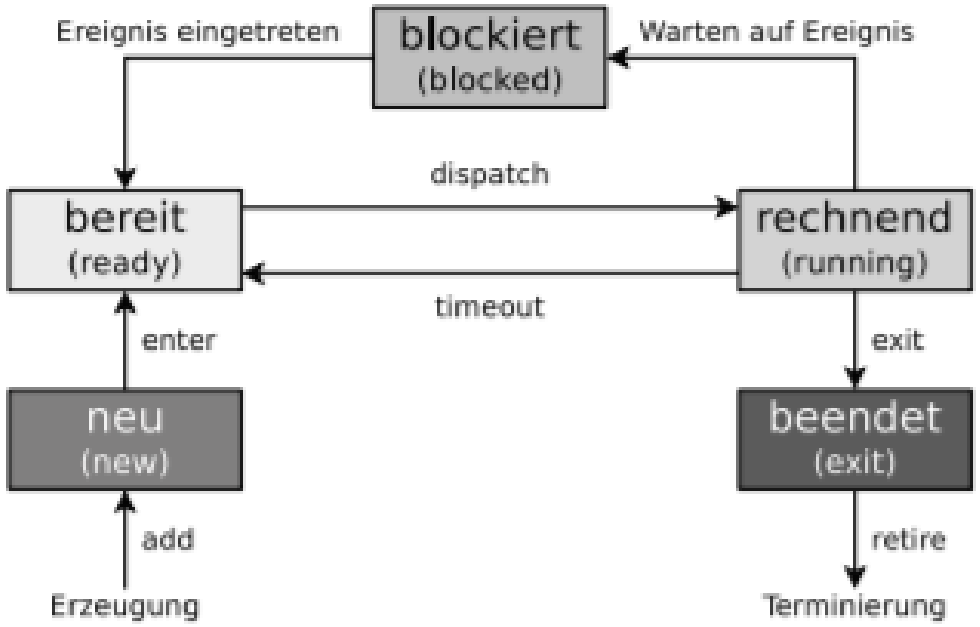
n Warteschlangen



- Warteschlangen
  - Besser eine eigene Warteschlange für jedes Ereignis

@Baun

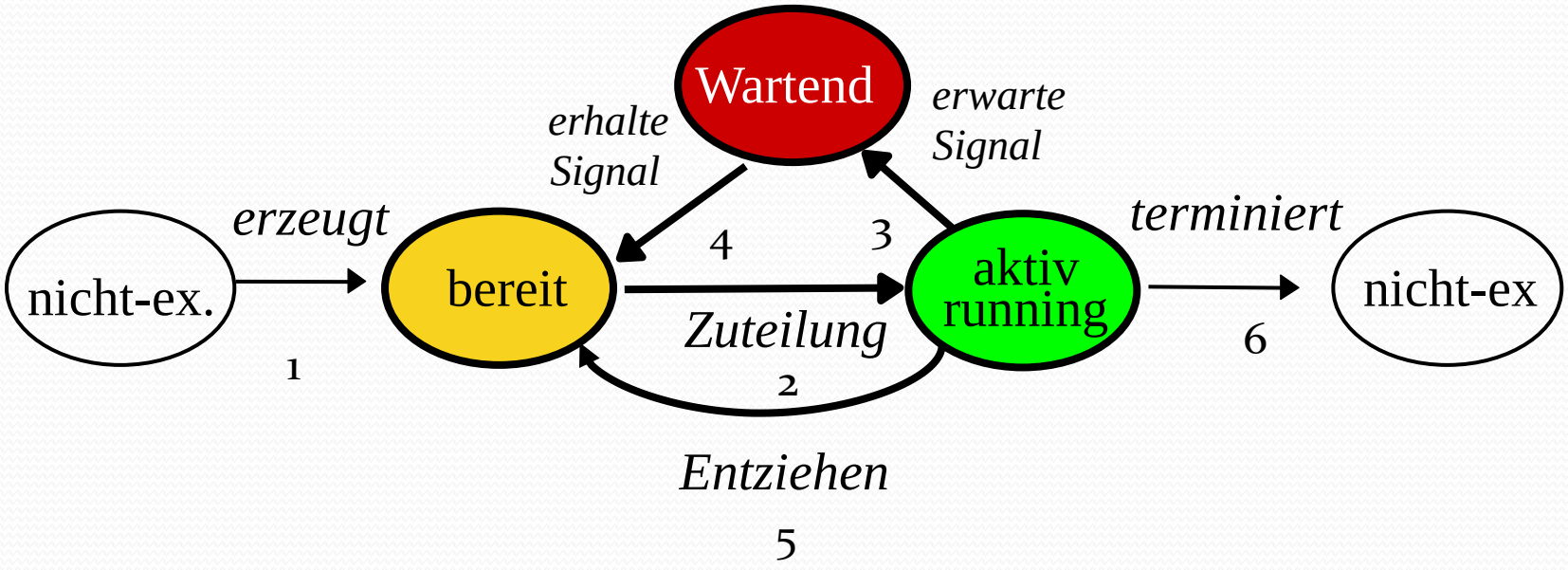
5 Prozesszustandsmodell



- Einführen zweier weitere Zustände
  - Bessere Kontrolle der Ressourcen
  - Bessere Kontrolle der rechenbaren Prozesse

@Baun

Prozesszustandsmodell



Prozesslebenszyklus

- Ein Prozess wird erzeugt:
  - Systemcall fork(Unix)
  - CreateProzess (Windows)
- Der Programmcode und die Daten werden in der Speicher geladen.
- Das Betriebssystem generiert eine eindeutige Identifikation (PID: Process Identification).
- Ein PCB wird in der Prozesstabelle angelegt.
- Übergang von (nicht existent) -> bereit (1)

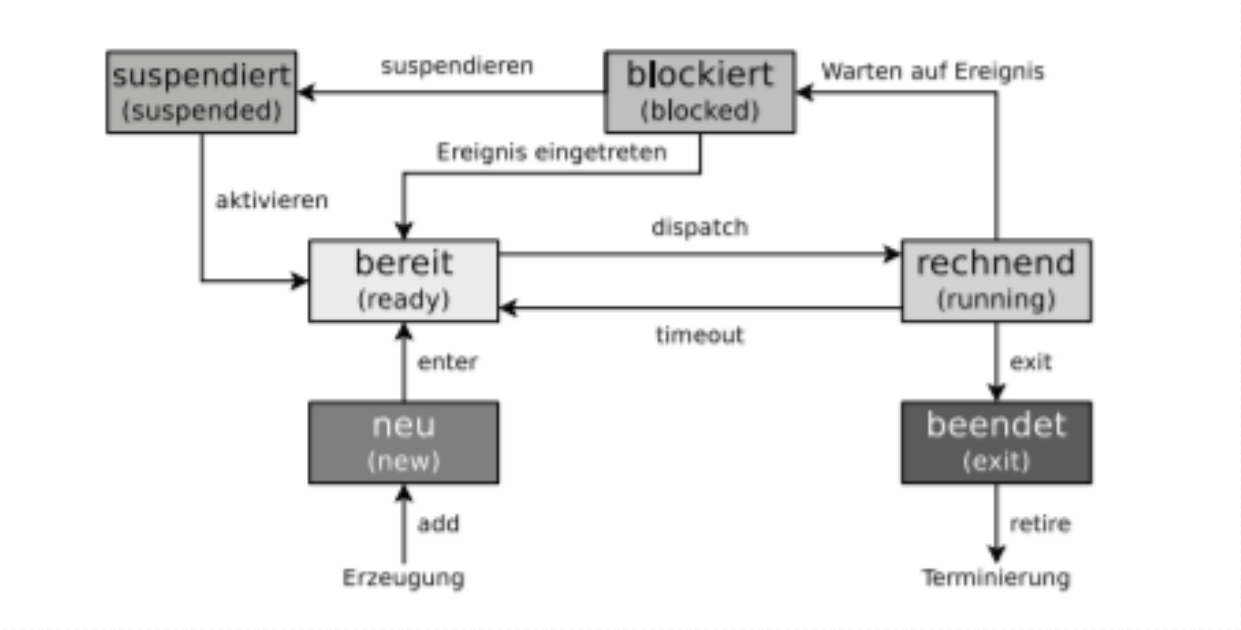
Prozesslebenszyklus

- Das Betriebssystem wählt einen Prozess (bereit) aus und dieser fängt seine Berechnung an (aktiv running). (2)
- Der Prozess wird blockiert (warten auf Input, Betriebsmittel). Er geht von „aktiv running“ in den Zustand „Wartend“. (3)
- Der Blockierungsgrund ist aufgehoben (Betriebsmittel verfügbar) Der Prozess geht von „Wartend“ in den Zustand „bereit“. (4)
- Das Betriebssystem entzieht dem Prozess die CPU und stellt ihn zurück in den Zustand (bereit). (5)
- Der Prozess terminiert. Er geht von „aktiv running“ in den Zustand „nicht-ex“.(6)
  - PCB wird aus der Tabelle entfernt
  - PID wird gelöscht.

Aufgabe Prozesslebenslauf

- Beschreiben Sie das Prozesszustandsmodel von Unix/Linux.
- Was ist eine Prozessbeschreibung?
  - Welche Daten werden dazu benötigt?
  - Gehen Sie auch auf den Hardware- System- und Nutzer-Kontext ein?
- Was ist ein Kontextwechsel? Erläutern Sie auch den Ablauf eines Kontextwechsel.
- Was versteht man unter Prozesshierarchien? Geben Sie typische Hierarchien für Unix und Linux an. Wie sieht das unter Windows aus?

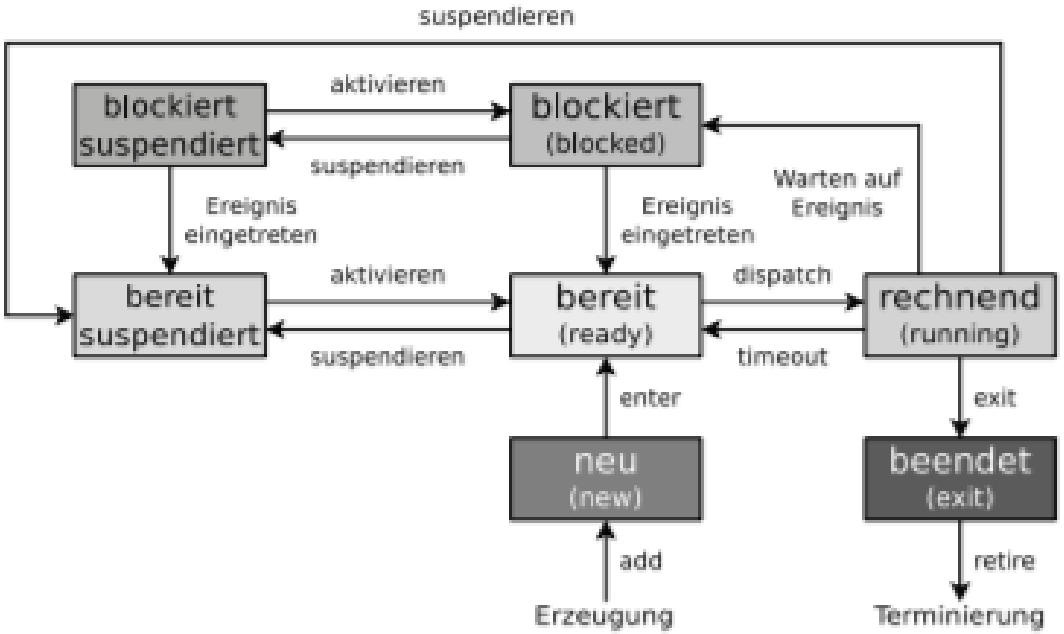
6 Prozesszustandsmodell



- Einführen eines weitere Zustandes
  - Falls nicht genügend Hauptspeicher vorhanden, können Teile eines Prozesses ausgelagert werden (Swap).
  - Der Prozess wird suspendiert.
  - Dadurch hat der rechnende Prozess mehr Speicher.

@Baun

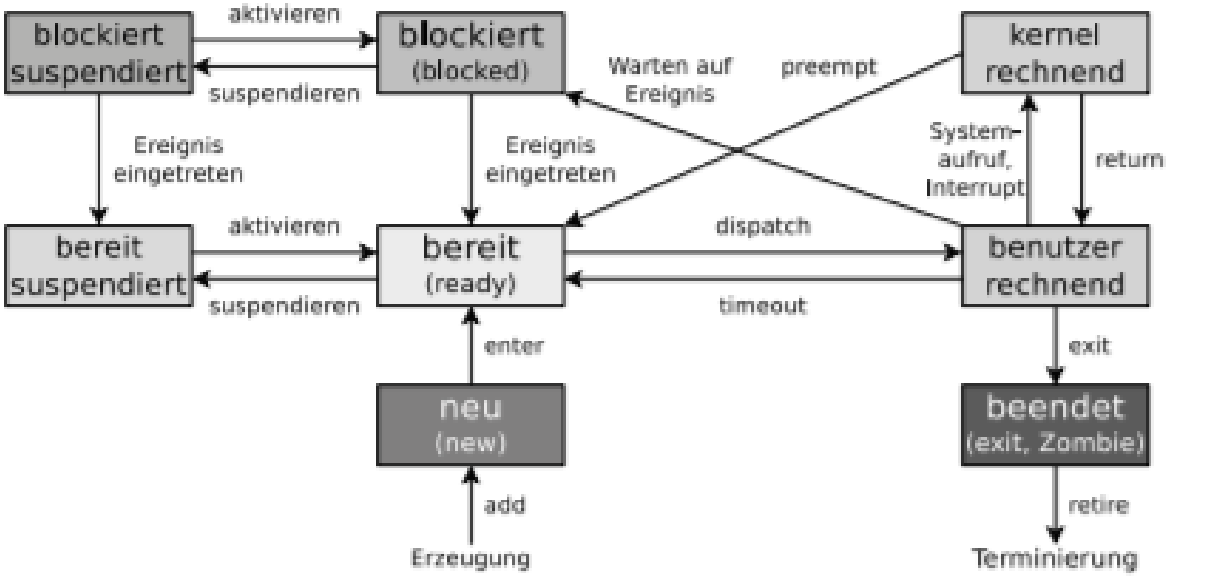
7 Prozesszustandsmodell



- Splitten des Zustands suspendiert
  - Der ausgelagerte Prozess könnte ja auf ein Ereignis warten.

@Baun

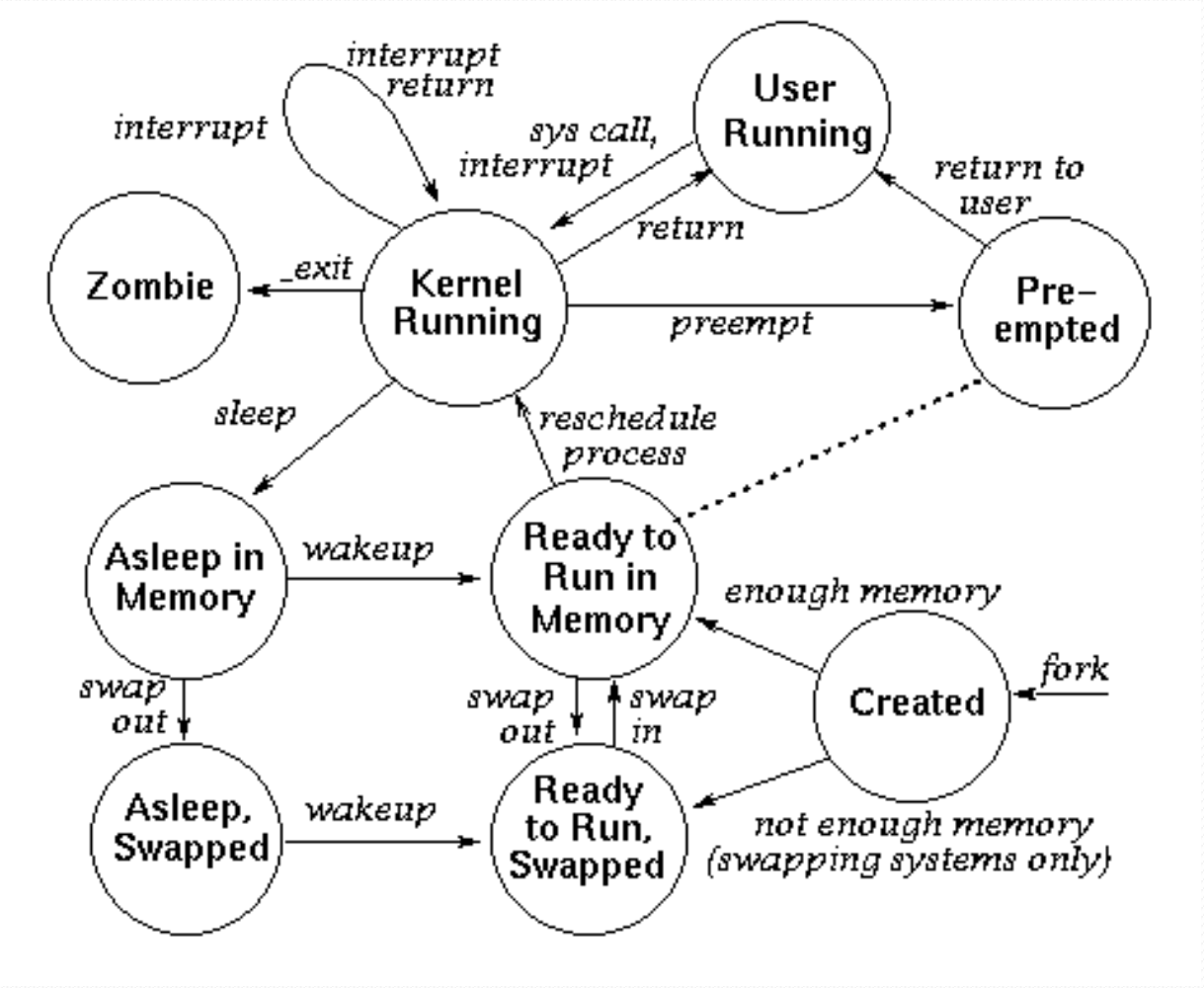
8 Prozesszustandsmodell



- Splitten des Zustands running in user-Mode und kernel-Mode
- Zustandsmodell von Unix.

@Baun

Prozesszustandsmodell Beispiel Unix



Prozessbeschreibungen

- Arbeitskontext der Prozesse:
- Benutzerkontext

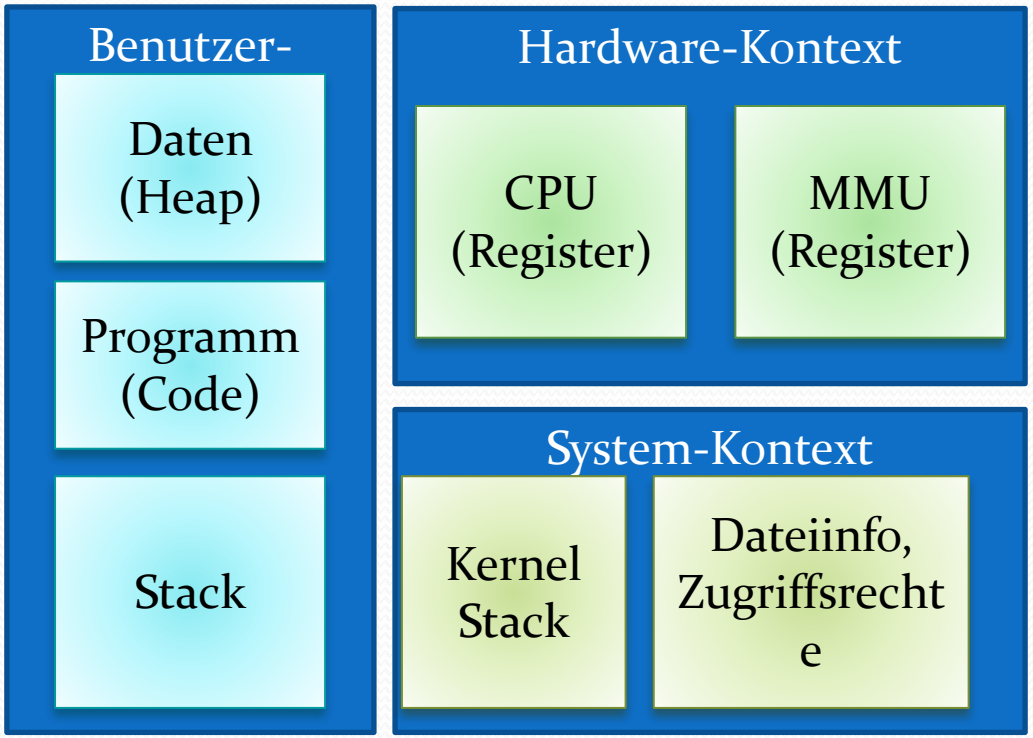
  Speicherreferenzen: Code-, Daten-, Stackadressen im Haupt- bzw. Massenspeicher => virtueller Adressraum
- Hardwarekontext
- Prozessorabbild (Register)
- Systemkontext (Prozesskontrollblöcke)

  Prozesszustand, erwartetes Ereignis, Timerzustand, PID, PID, der Eltern, User/Group-IDs, Betriebsmittel

Prozesskontext

Zum Teil auslagerbar (swappable)

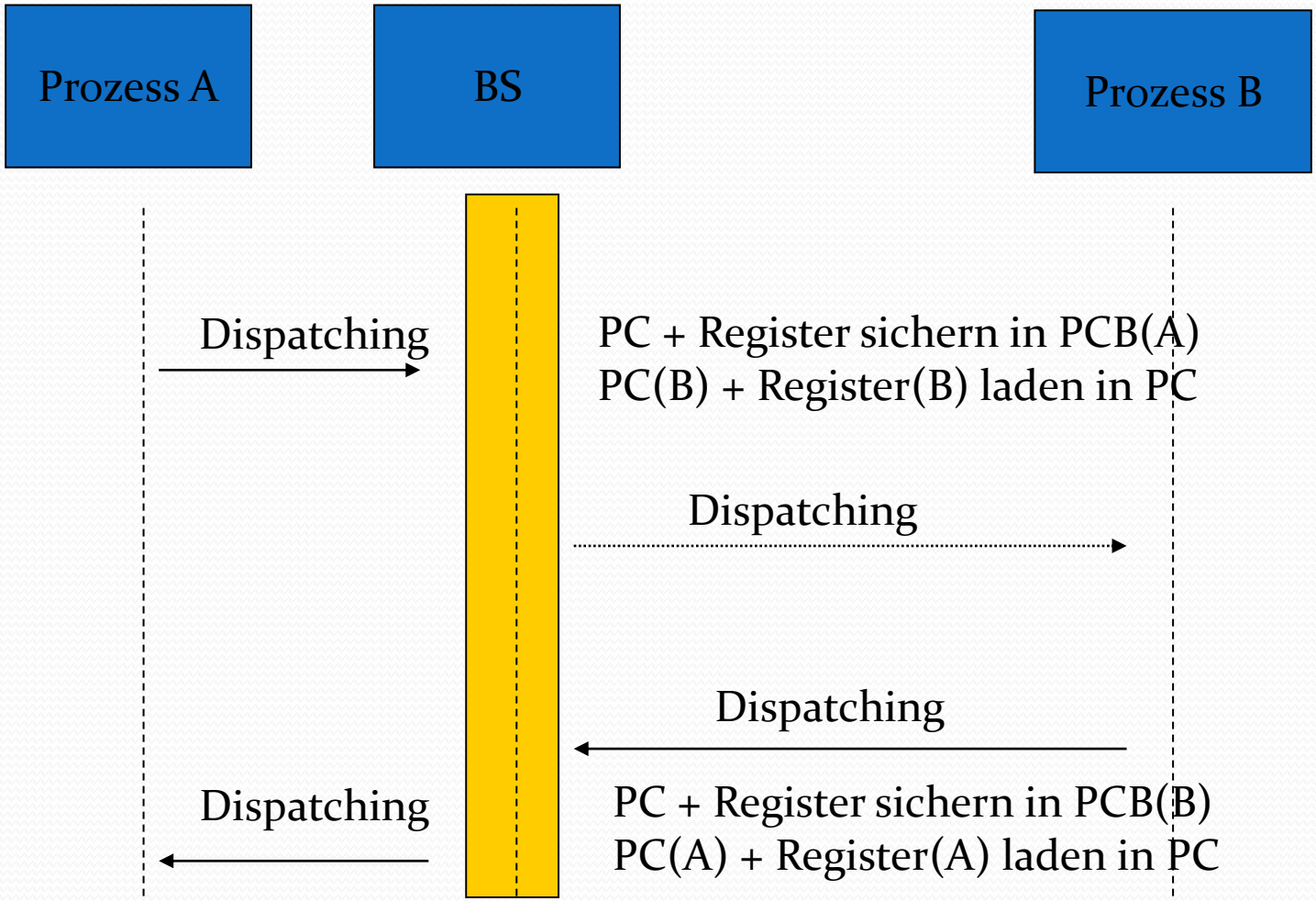
- Benutzerkontext
- Hardwarekontext



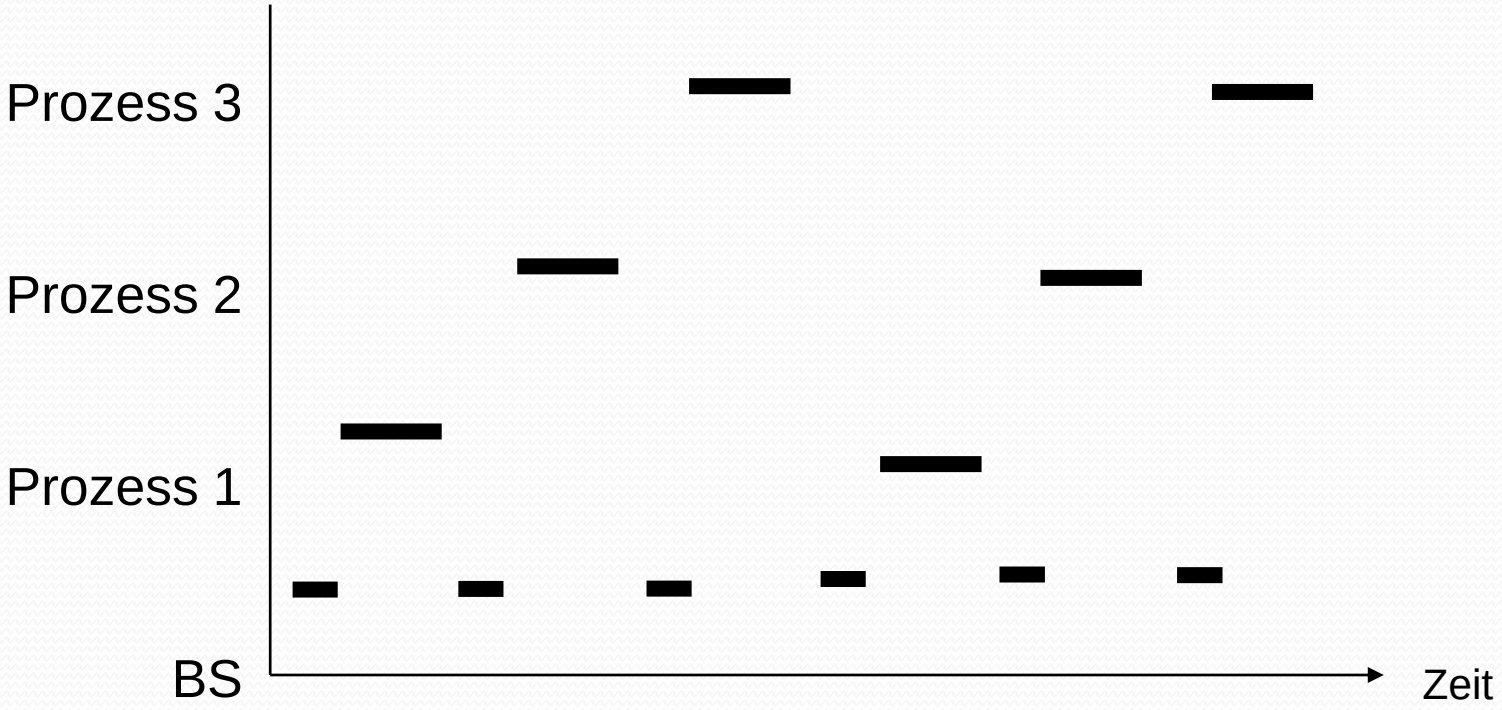
Kontextwechsel

- Wenn das Betriebssystem ein Prozess ablöst, muss der Hardware-Kontext des zu suspendierenden Prozesses für eine neue Aktivierung aufbewahrt werden. (PCB)
- Der Hardware-Kontext des neu zu aktivierenden Prozesses wird aus seinem PCB in die Ablaufumgebung geladen.

Ablauf eines Kontextwechsels



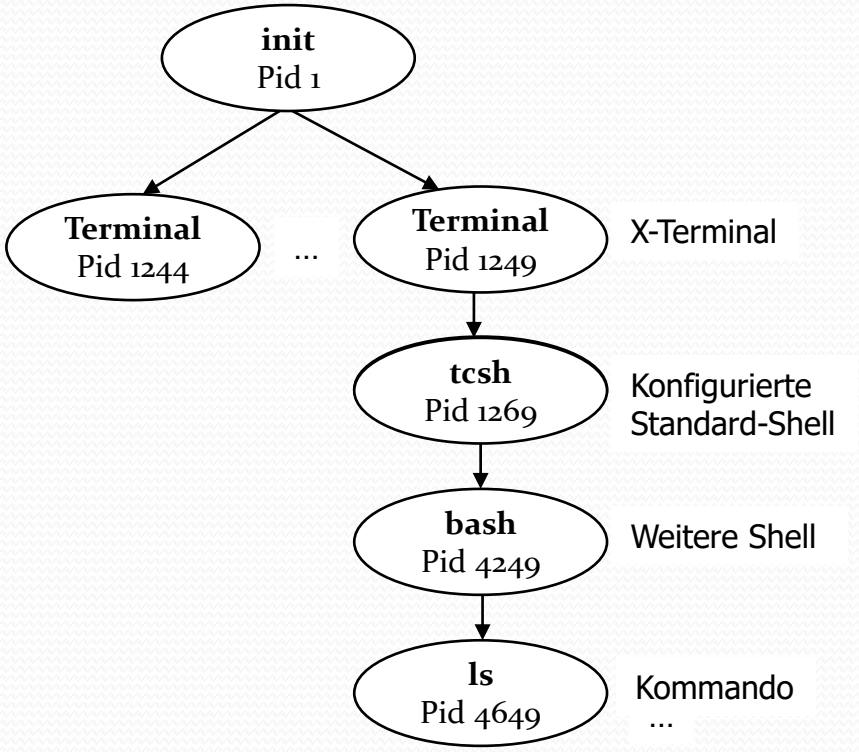
Prozessabarbeitung



Prozesshierarchien

Unix

- Prozesssicht nach dem Login: Kommando pstree
- Ein Prozess mit Bezeichnung **Terminal** als X-Terminal (Terminal-Emulation unter grafischer Oberfläche) läuft



Prozesshierrachien

Linux

- Prozesse nach dem Booten (nach Tanenbaum)

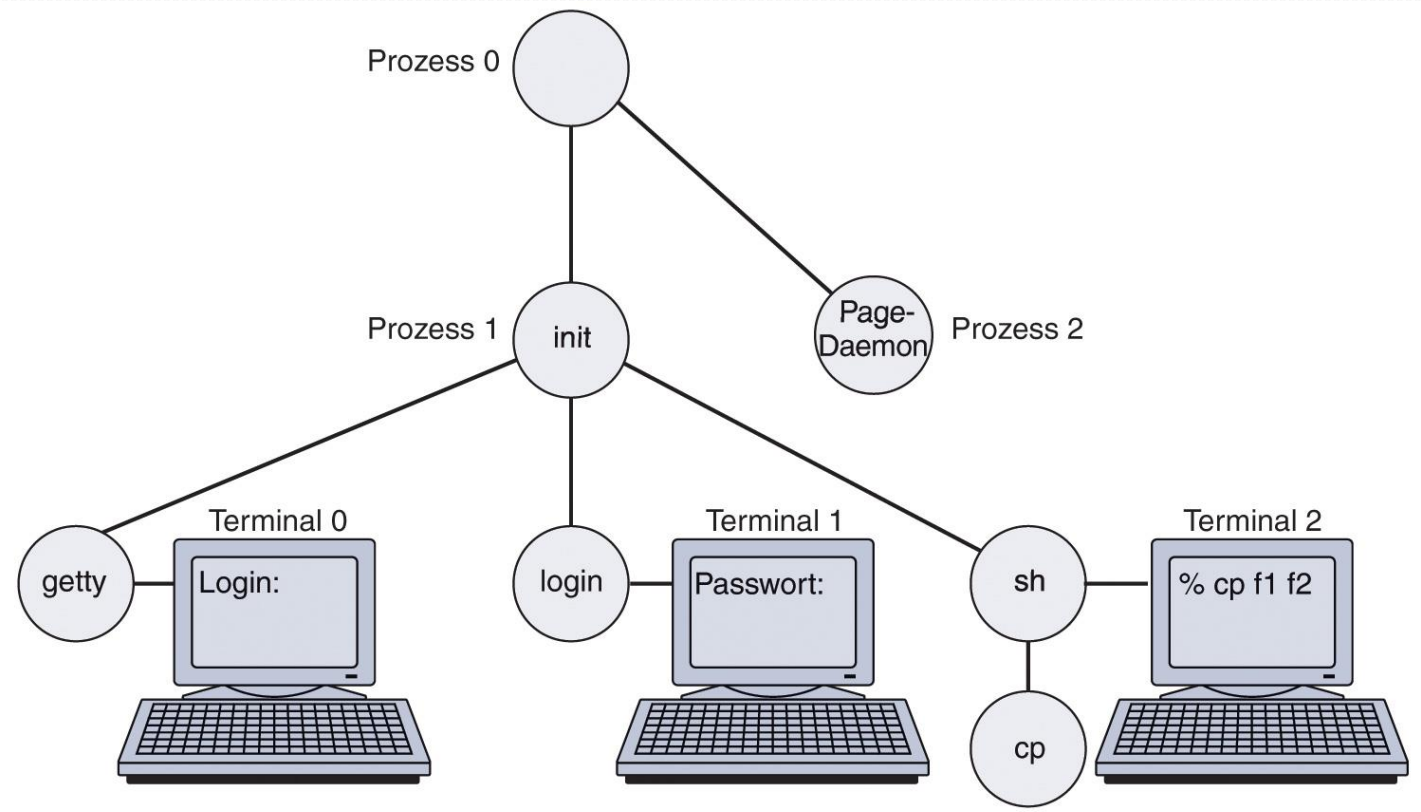


Abbildung 10.11: Folge von Prozessen, die zum Booten von einigen Linux-Systemen verwendet wird

Prozesshierrachien

Linux und Unix

- Initialisierungsprozess (init)
  - Ist im Bootimage vorhanden. Er liest aus einer Datei wie viele Terminals vorhanden sind und startet für jedes Terminal einen neuen Prozess.
- Für jeden Benutzer ein Loginprozess
- Kommandointerpreter
  = Shells pro Benutzer
- Anwendungen

Prozesshierrachien

Windows

- Unter Windows gibt es keine Prozesshierarchie.
- Alle Prozesse gleichwertig
- Ein Elternprozess erhält ein Token, um sein Kindprozess zu steuern.
- Dieser Token kann weitergegeben werden.

Prozesse

- Traditionelles Modell: Prozess hat
  - einen Adressraum
  - einen Ausführungsfaden

- Manchmal wünschenswert:
  - mehrere „Ausführungsfäden“ parallel zu haben
  - wie eigenständige Prozesse, aber gemeinsamer Adressraum

Prozesse

- Ein Prozess bündelt zusammengehörige Ressourcen.

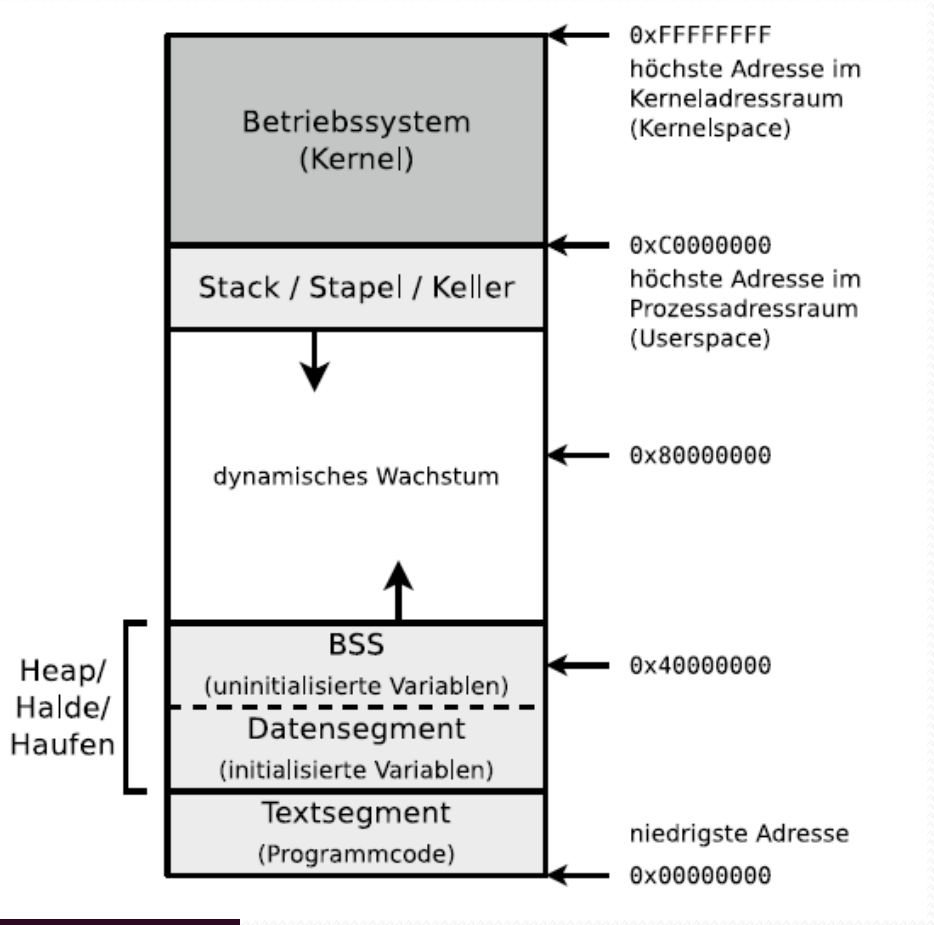
- Ressourcen:
  - Adressraum
  - Offene Files
  - Kinderprozesse
  - Accounting

Adressraumverwaltung

- Linux in einem 32-Bit System
- standardmäßigen Aufteilung des virtuellen Adressraums auf einem 32 Bit-System
  - 25% für Kernel
  - 75% für Usermodus
- Ein Prozess kann damit bis zu 3GB Speicher verwenden.
- Textsegment: Programmcode
- Heap ist dynamisch und besteht aus
  - Datensegment (initialisierten Variablen und Konstanten)
  - BSS(Block Started by Symbol) (nicht initialisierten Variablen, Speicheranfragen zur Laufzeit)
- Stack (Speicherplatz für Unterprogrammaufrufe)
- Ausgabe in Linux:

-> size *

```
ks@ubuntu:~/programme$ size a.out bsp_fork bsp_fork2
   text    data     bss     dec     hex filename
   2171     640       8    2819     b03 a.out
   1607     608       8    2223     8af bsp_fork
   1794     616       8    2418     972 bsp_fork2
   1579     608       8    2195     893 fork_3
   2138     648     280    3066     bfa hello
```



@Baun

Threads

- Prozess besitzt Ausführungsfaden (thread)
-
- Thread: werden für die Ausführung auf CPU verwaltet
- Threads besitzen:
  - Befehlszähler (PC)
  - Register
  - Stack
- => mehrere Fäden innerhalb eines Prozesses

Prozesse-Threads

Elemente pro Prozess	Elemente pro Thread
Adressraum	Befehlszähler
Globale Variable	Register
Geöffnete Dateien	Stack
Kindprozesse	Zustand
Ausstehende Signale	
Signale und Signalroutinen	
Verwaltungsinformationen	

Abbildung 2.12: Die erste Spalte führt Elemente auf, die alle Threads des Prozesses teilen. Die zweite Spalte zeigt Elemente, die zu einem individuellen Thread gehören.

Thread-Modell

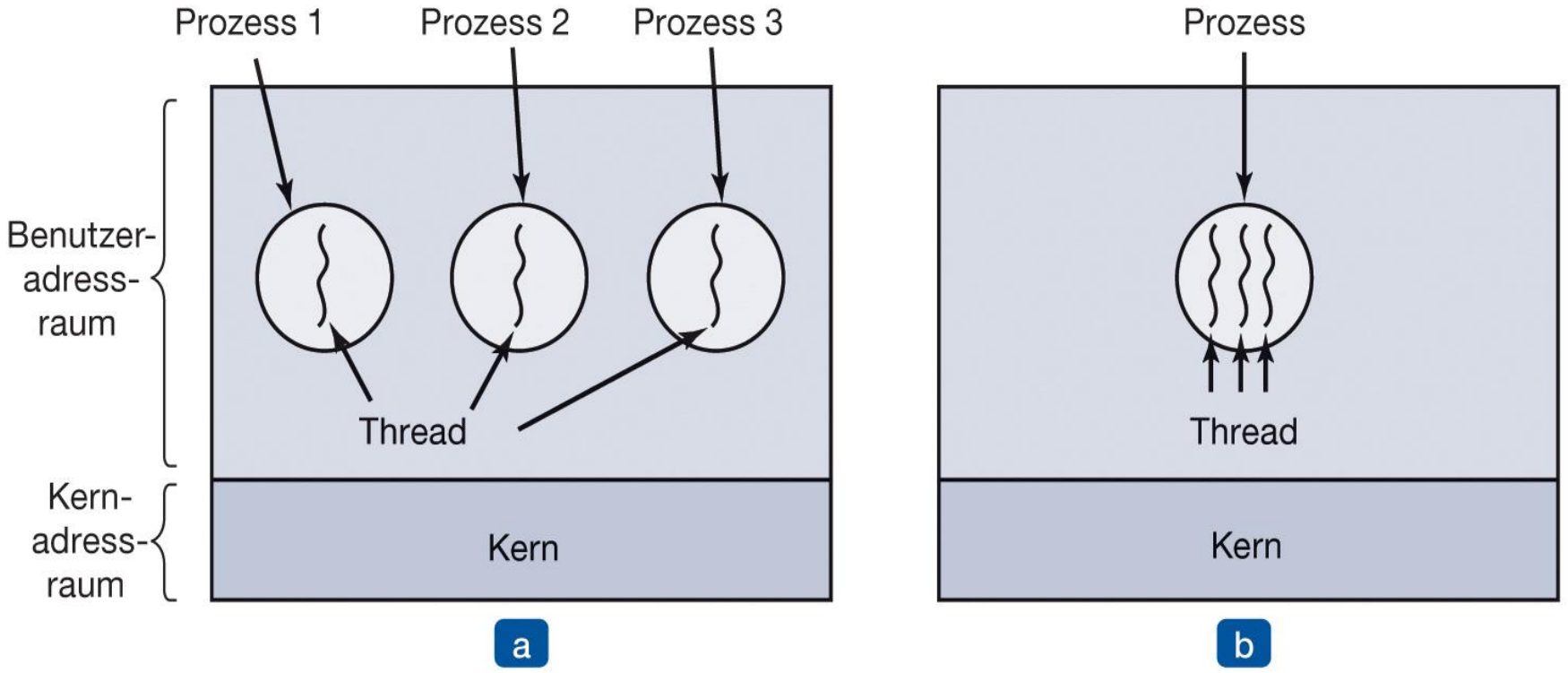


Abbildung 2.11: (a) Drei Prozesse mit je einem Thread (b) Ein Prozess mit drei Threads

Aufgaben zu Threads

- Warum werden Threads eingesetzt. Nennen Sie auch Beispiele, wo man Threads nutzt.
- Nennen Sie Vor- und Nachteile von Threads
- Threads lassen sich im user oder kernel-Mode implementieren. Was sind die Vor- und Nachteile dieser Implementierung?
- Diskutieren Sie das Prozessmodell von Windows

Threads

- **Vorteile:**
- Prozesswechselzeiten sind kürzer als zwischen normalen Prozessen (kleinerer Kontext, keine Umschaltung zwischen unterschiedlichen Betriebsmittelausstattungen)
- Gemeinsame Nutzung von Betriebsmitteln wird erleichtert (insbesondere eines gemeinsamen Speicherbereichs)

  Beispiele:
  - Tabellenkalkulationsprogramm mit Subprozessen für Eingabe und Berechnung
  - Serverprozesse in Client/Server-Anwendungen: Bereitstellung bestimmter Betriebsmittel
- Mit Wartezeiten verbundene Ein/Ausgabe-Aufträge können von eigenen Subprozessen abgewickelt

Threads

- **Nachteile bei der Strukturierung eines Anwendungssystems:**
- Koordination der Subprozesse und Zugriff auf gemeinsame Betriebsmittel nur durch Programmierer
- Ohne Nutzung von Wartezeiten für Ein-/Ausgabe nehmen sich Subprozesse nur gegenseitig Rechenzeit weg, ohne den Gesamtprozessablauf zu beschleunigen
- Ein Wechsel zwischen Subprozessen verschiedener Prozesse kostet genauso viel Zeit wie ein normaler Prozesswechsel.

Thread-Implementierung

- Es gibt zwei Möglichkeiten Threads zu implementieren
- Implementieren auf Benutzerebene
  - Eine Threadbibliothek übernimmt das Scheduling und das Umschalten.
  - Threadtabelle wird im Benutzerspeicher verwaltet.
- Implementieren auf Kernelebene
  - Verwalten im Kernelmodus
  - Kernel stellt die Methoden bereit
  - Keine spezielle Threadbibliothek erforderlich
  - Threadtabelle wird im Kernelspeicher verwaltet.

Thread-Implementierung

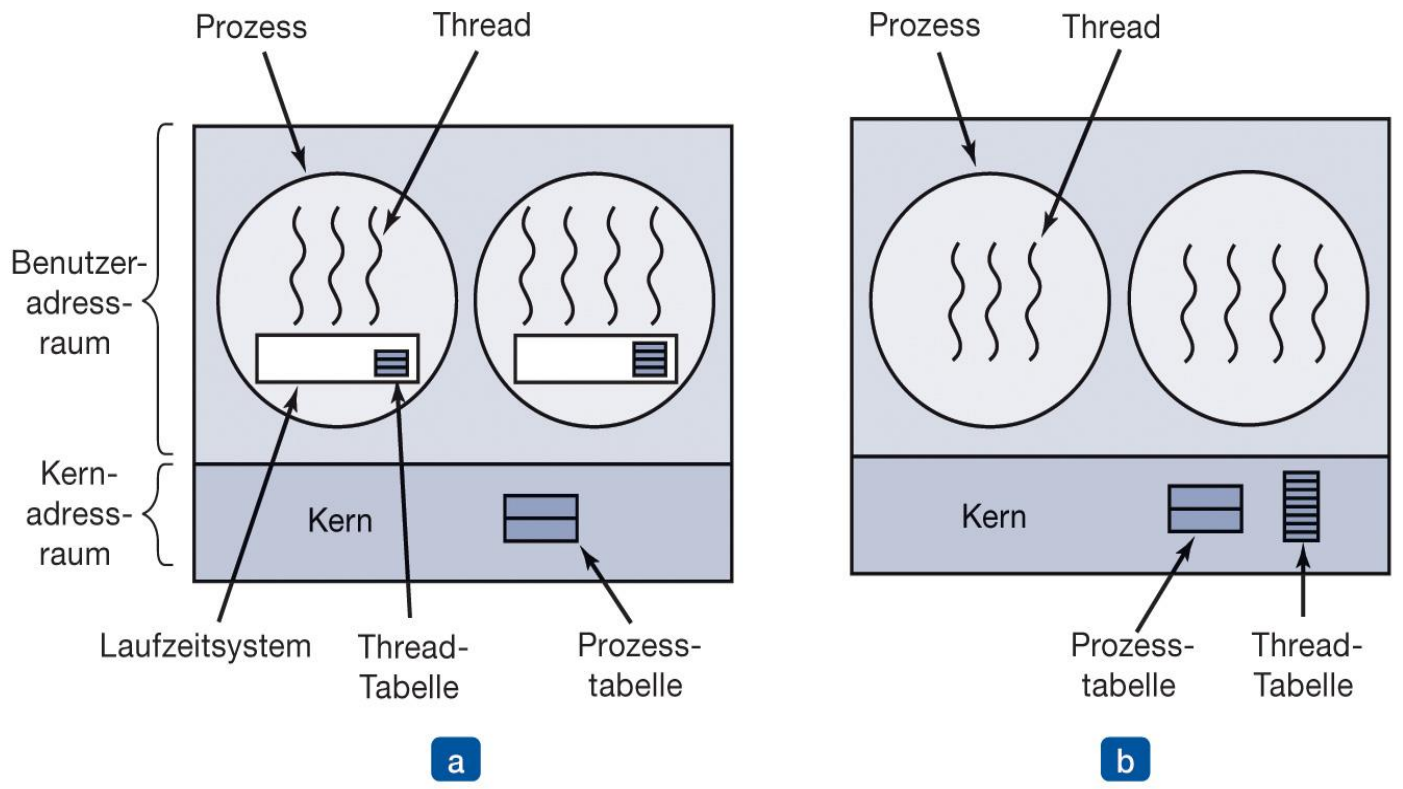
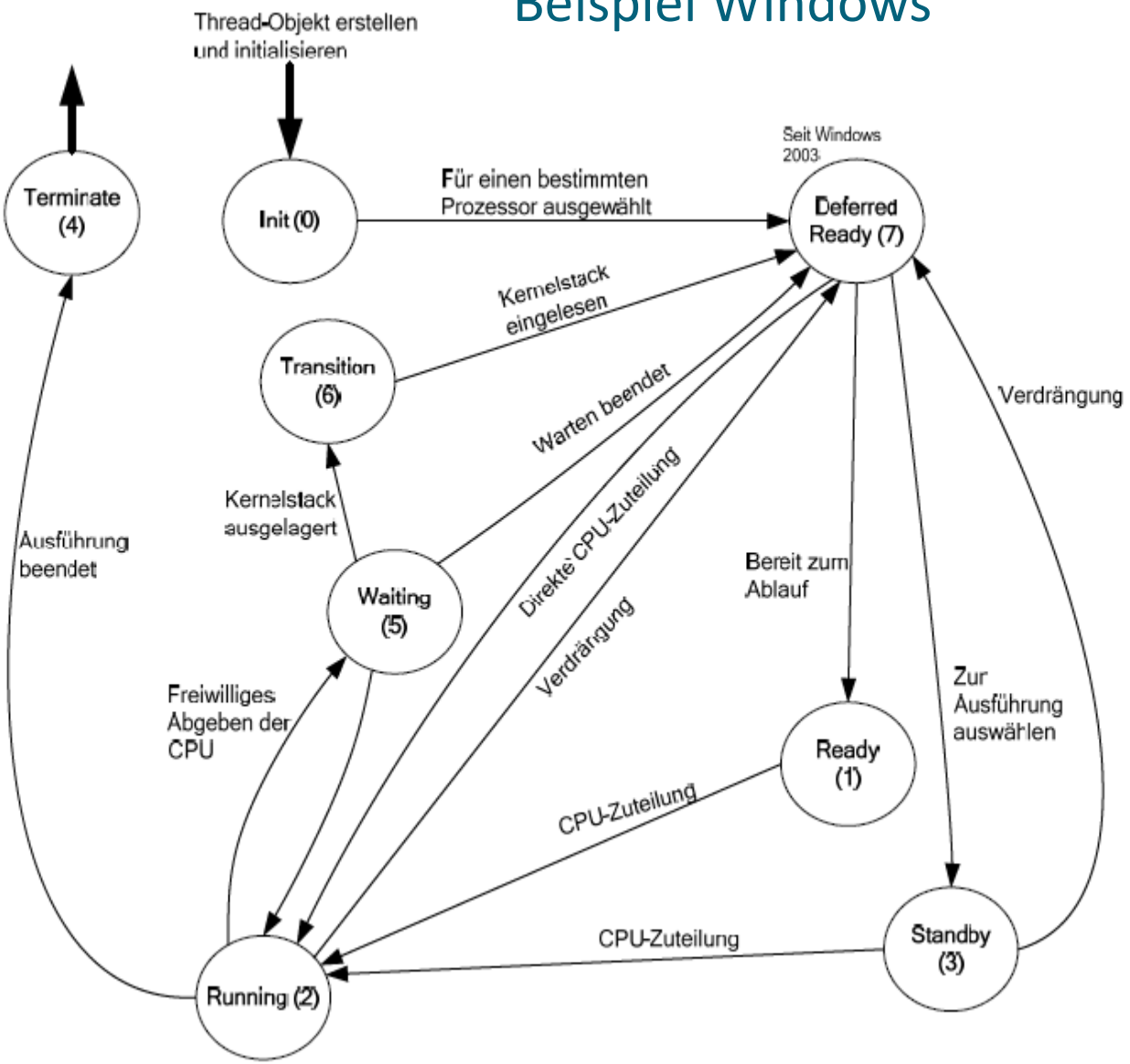


Abbildung 2.16: (a) Ein Thread-Paket auf Benutzerebene (b) Ein Thread-Paket, verwaltet vom Kern

Threadzustandsmodell
Beispiel Windows

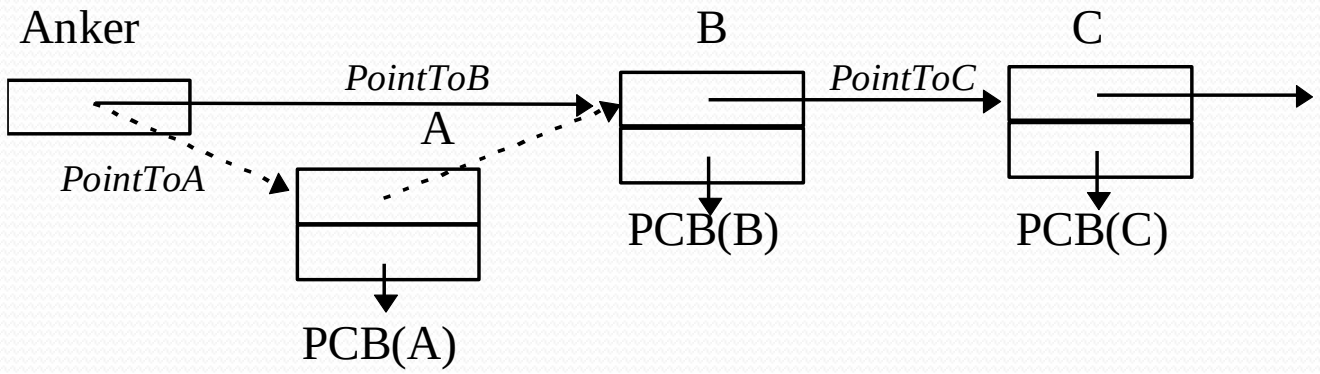


Prozesskoordinierung

- Konkurrenz
- mind. zwei Prozesse
- ein Betriebsmittel
- exklusiv
- Kritischer Abschnitt
- exklusiver Zugriff
- keine Parallelität

Prozesskoordinierung

- **Situation**: Warteschlange „Bereit“ für Prozessbeschreibungen (PCB)



Einhängen

(1) Lesen des Ankers: PointToB

(2) Setzen des NextZeigers:=PointToB

3) Setzen des Ankers:=PointToA

Aushängen

(1) Lesen des Ankers: PointToB

(2) Lesen des NextZeigers:PointToC

(3) Setzen des Ankers:=PointToC

Prozesskoordinierung

- **Problem**: („Race conditions“: kontextbedingt, nicht-wiederholbare Effekte durch „überholende“ Prozesse)

  a) Unterbrechung beim Aushängen von B durch Einhängen von A

  ⇒ Prozess A ist **weg**!

  b) Unterbrechung beim Einhängen von A durch Aushängen von B

  ⇒ Prozess B **bleibt** erhalten!

Konkurrenz - Anforderungen

- Zwei Prozesse dürfen **nicht gleichzeitig** in ihren kritischen Abschnitten sein (*mutual exclusion*).
- Jeder Prozess, der am Eingang eines kritischen Abschnitts wartet, muss irgendwann den Abschnitt auch betreten dürfen: **kein ewiges Warten** darf möglich sein (*fairness condition*).
- Ein Prozess darf außerhalb eines kritischen Abschnitts einen anderen Prozess **nicht blockieren**.
- Es dürfen **keine Annahmen** über die Abarbeitungsgeschwindigkeit oder Anzahl der Prozesse bzw. Prozessoren gemacht werden.

Konkurrenz- kritische Abschnitte

Vier Bedingungen für korrekte und effiziente Lösungen:

- 1. Keine zwei Prozesse gleichzeitig in kritischem Abschnitt
- 2. Keine Annahmen über relative Geschwindigkeit oder Anzahl der CPUs
- 3. Kein Prozess außerhalb seines kritischen Abschnitts darf andere Prozesse behindern oder blockieren
- 4. Kein Prozess darf ewig auf seinen Eintritt in den kritischen Abschnitt warten

Konkurrenz- kritische Abschnitte

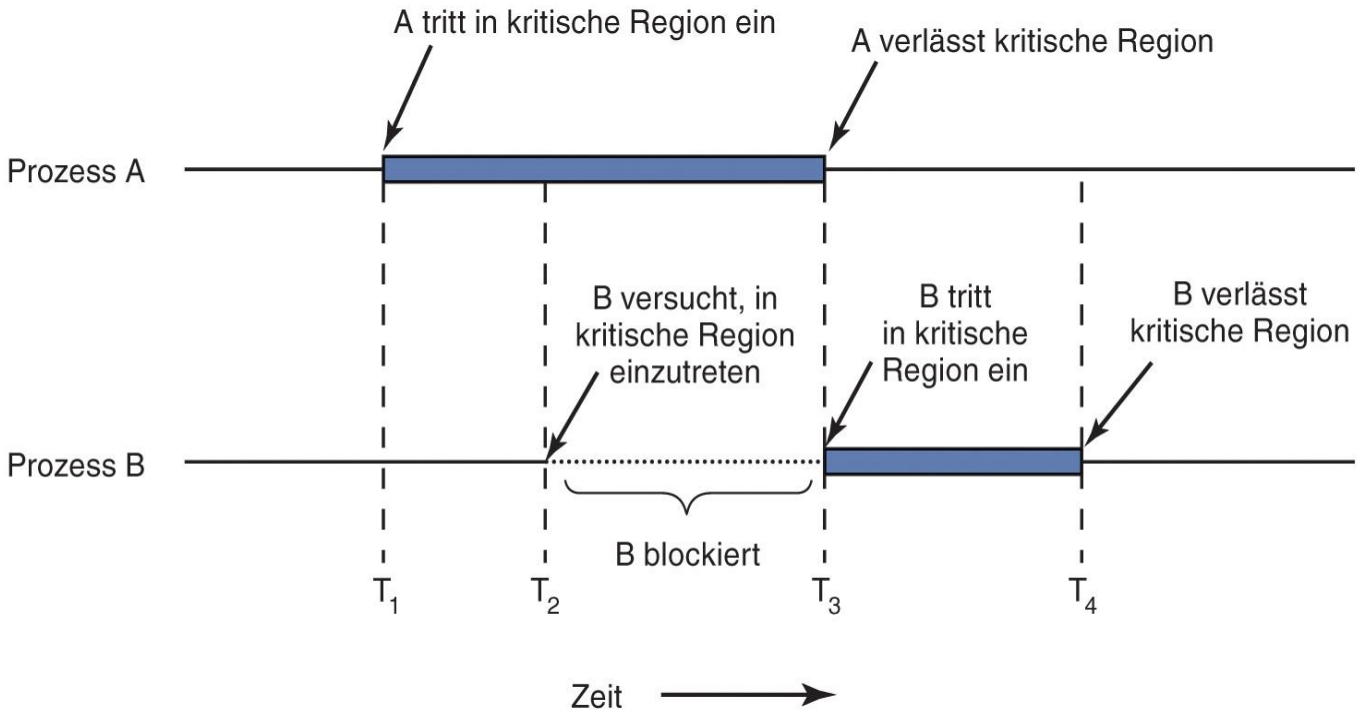


Abbildung 2.22: Wechselseitiger Ausschluss unter Verwendung von kritischen Regionen

Semaphore

- Semaphore wurden von Dijkstra1962 entwickelt und basieren auf Sperrmechanismen.
- Höherwertiges Konzept zur Lösung des Mutual-Exclusion-Problems.
- Semaphore verwalten intern
  - eine Warteschlange für Prozesse bzw. Threads, die am Eingang warten müssen.
  - Einen Semaphorzähler, der angibt wie viele Prozesse in den kritischen Abschnitt dürfen.
  - Für den Eintritt und für den Austritt gibt es zwei Operation
    - **P():** wird beim Eintritt in den kritischen Abschnitt aufgerufen. Der Semaphorzähler wird dabei um 1 vermindert, sofern er größer als 0 ist. Wenn der Zähler 0 ist, wird der Zutritt verwehrt.
    - **V():** wird beim Verlassen des kritischen Abschnitt aufgerufen. Der Semaphorzähler wird um 1 erhöht.

Konkurrenz- kritische Abschnitte

Passieren P(s) waitFor (signal)

- Aufruf *vor* kritischem Abschnitt, warten falls besetzt

Verlassen V(s) send (signal)

- Aufruf *nach* kritischen Abschnitt, Aktivieren eines wartenden Prozesses

Voraussetzung/Forderung:

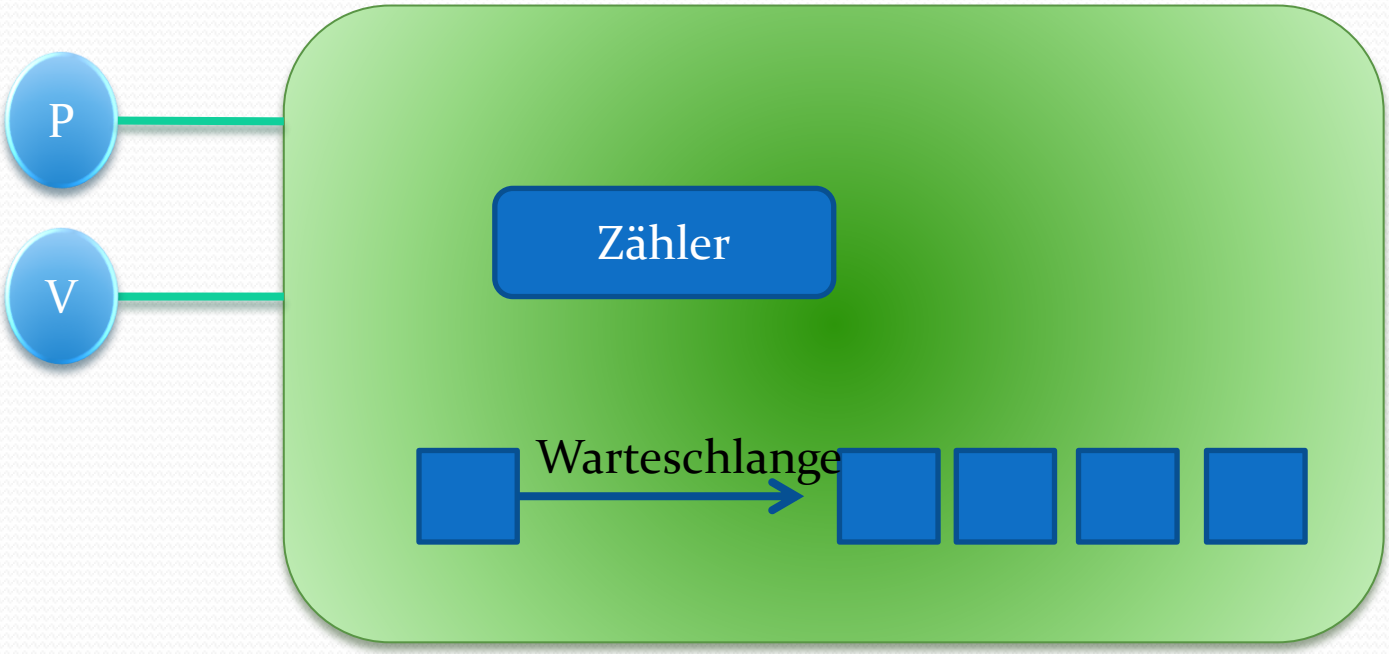
- „Atomare Aktion“:
- Entweder **vollständige** Transaktion oder gar **keine**! („roll back“ bei Abbruch)

Es gibt 2 Typen von Semaphore

- Binäre Semaphore (Mutex) mit zwei Zustände (locked, unlocked).
- Zählsemaphore. Dies ist der allgemeine Semaphore mit beliebig vielen Zuständen.

Semaphore

- Konzept Semaphore



Semaphore Code Beispiel p()

Software Pseudo-Code für P()

```
// initialisieren des Semaphorzählers
s:= x; // mit x >0
void P(){ // kritischer Abschnitt
  if(s >= 1) {
      s := s-1; // ausführender Prozess geht weiter
      } else {
      // ausführender Prozess wird gestoppt und in
eine
      // dem Semaphore zugeordnete Warteliste
eingetragen.
      }
}
```

Semaphore Code Beispiel V()

Software Pseudo-Code für V

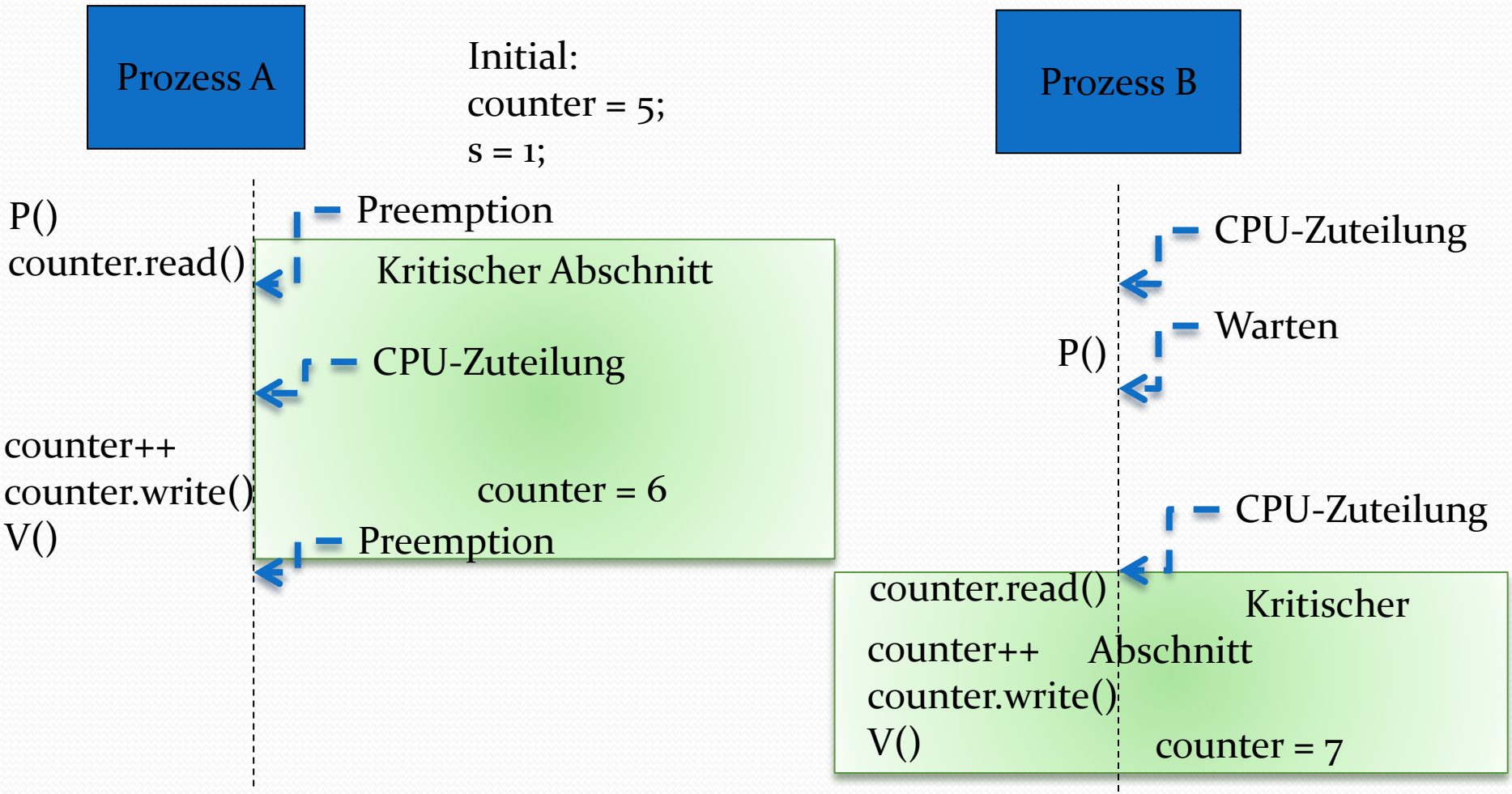
```
void V(){ // dies ist auch ein kritischer Abschnitt
   s:= s+1;
   if( Warteliste ist nicht leer) {
        //aus der Warteliste wird ein Prozess
ausgewählt und
        //aufgeweckt
}
```

Semaphore Code Beispiel Nutzung

- **Software Pseudo-Code**

```
// falls der kritische Abschnitt besetzt, dann warten
P();
// kritischer Abschnitt beginnt
z = z+3;
Write(z);
// kritischer Abschnitt endet
// Verlassen des kritischen Abschnitts
// Aufwecken eines Prozesses
V();
…
```

Semaphore Beispiel
Vermeidung von Lost-Update



Semaphore Beispiele

- Kooperation
- Parallele Prozesse
- Bewusste Zusammenarbeit

- Ereignissynchronisation

  $\Rightarrow$ Reihenfolge hängt von Ereignissen ab

Interprozesskommunikation (IPC)

  $\Rightarrow$ Austausch von Ergebnissen

Lösung Ereignissynchronisation: z.B. Semaphore

Aufgabe Erzeuger/Verbraucher

Welche Probleme können auftreten?
Versuchen Sie, diese Probleme mit einem Semaphor zu Lösen.
Wie könnte ein Code-Fragment aussehen?
Welche Probleme sehen Sie bei der Verwendung von Semaphoren?

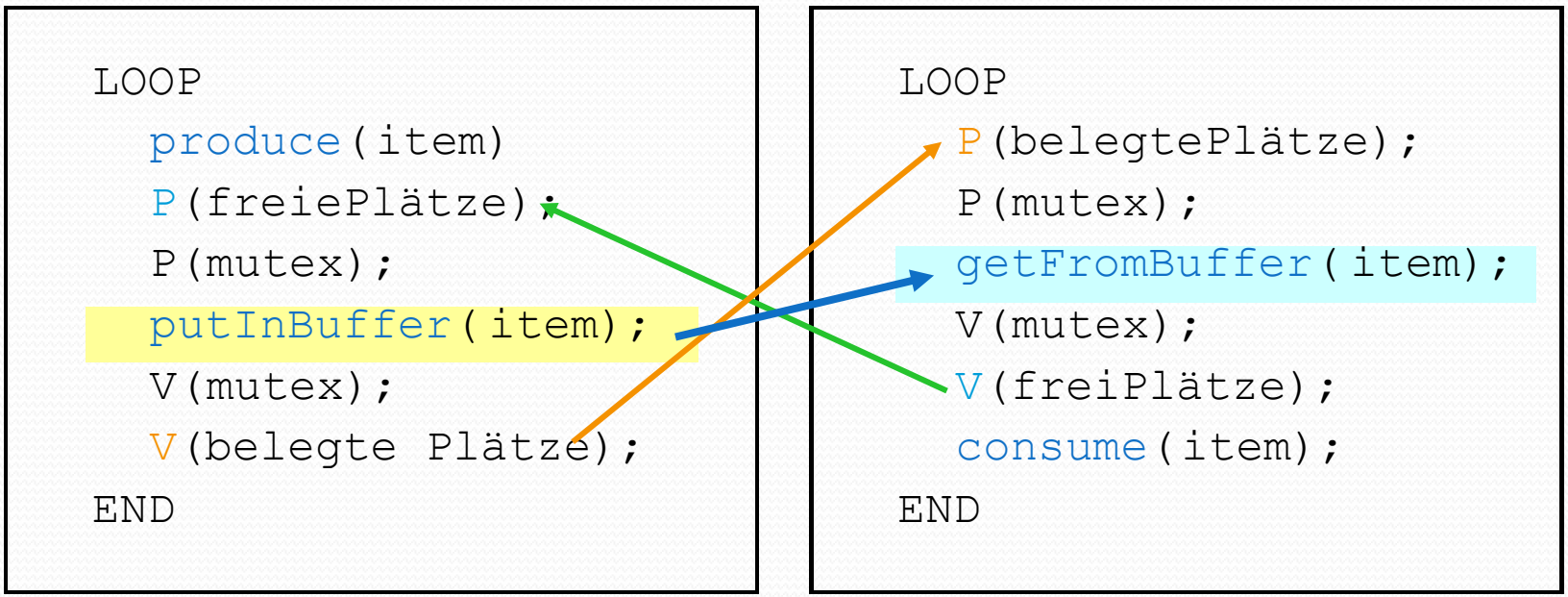
Semaphore Beispiele

Erzeuger

```
LOOP
   produce(item)
   P(freiePlätze);
   P(mutex);
   putInBuffer(item);
   V(mutex);
   V(belegte Plätze);
END
```

Verbraucher

```
LOOP
   P(belegtePlätze);
   P(mutex);
   getFromBuffer(item);
   V(mutex);
   V(freiPlätze);
   consume(item);
END
```



Semaphore Beispiele

- **Vertauschung** : alle sind nur gleichzeitig im krit. Abschnitt
- **V(); … krit. Abschnitt …; P();**

- **Replikation** oder Weglassen: ewiges Warten
- **P(); … krit. Abschnitt …; P();**

Philosophenproblem

Fünf Philosophen sitzen an einem runden Tisch.
Jeder hat einen Teller mit Essen vor sich.
Zum Essen benötigt jeder Philosoph zwei Gabeln. Da es nur 5 Gabeln gibt, können nicht alle gleichzeitig essen.

Die Philosophen denken über Probleme nach. Wenn einer hungrig ist nimmt er die rechte und linke Gabel und isst. Wenn er satt ist legt er die Gabel wieder hin und beginnt zu denken.

Hier kann es zu Verklemmung kommen, wenn alle 5 Philosophen gleichzeitig die linke Gabel nehmen. Dann ist die jeweils rechte Gabel besetzt und keiner kann essen und sie alle müssen verhungern.
Wie kann man dieses Problem sinnvoll lösen?