2. Prozesse

- Überblick

2.1 Prozesse

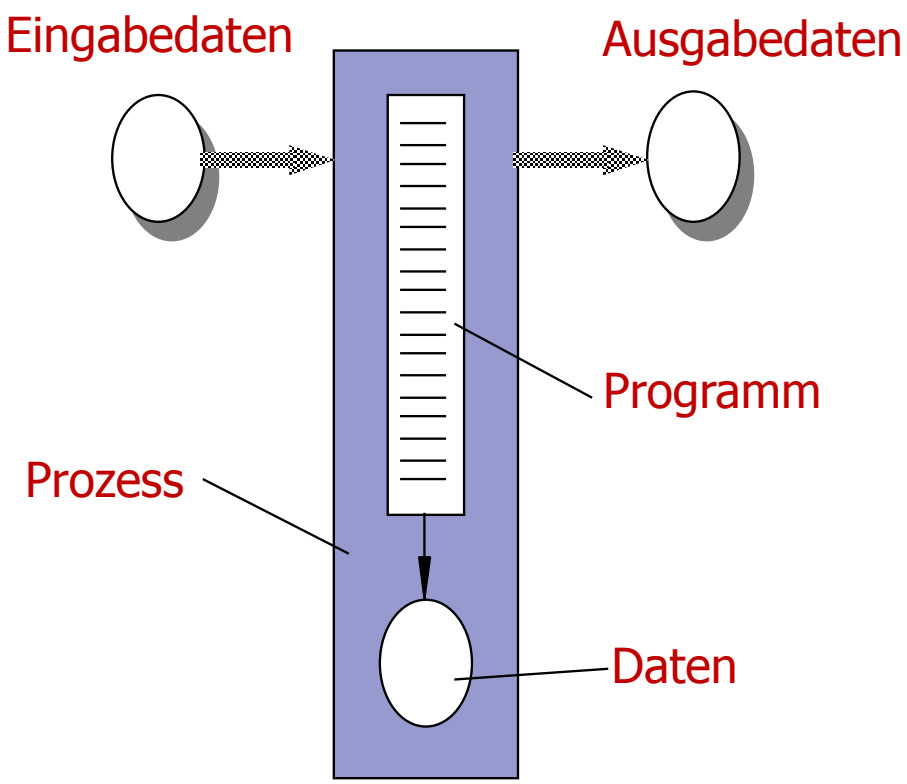
- Alle Anwendungen sind auf die Zuweisung von Prozessor und Speicher angewiesen
  - ⇒Prozessmanagement und Prozessinteraktion sind unverzichtbare Dienste
- Prozesse sind dynamische Objekte, die sequentielle Aktivitäten in einem System repräsentieren
- Ein Prozess (process, task) ist definiert durch
  - Adressraum
  - Verarbeitungsvorschrift, üblicherweise ein Programm
  - Aktivitätsträger, der die Verarbeitungsvorschrift ausführt, in der Regel als Thread bezeichnet

Prozesse haben keinen Zugriff auf Speicher anderer Prozesse -> Sicherheit

- Prozess = virtueller Rechner spezialisiert zur Ausführung eines bestimmten Programms

Beschreibungseinheit Prozess

- Ein Prozess verfügt über
  - Ein- und Ausgabedaten (Parameter) sowie
  - Interne Daten
- Prozess = Beschreibungs-einheit, die für System- und Anwendungssoftware als funktionale und strukturierende Einheit gleichermaßen geeignet ist
- Prozess = laufendes Programm

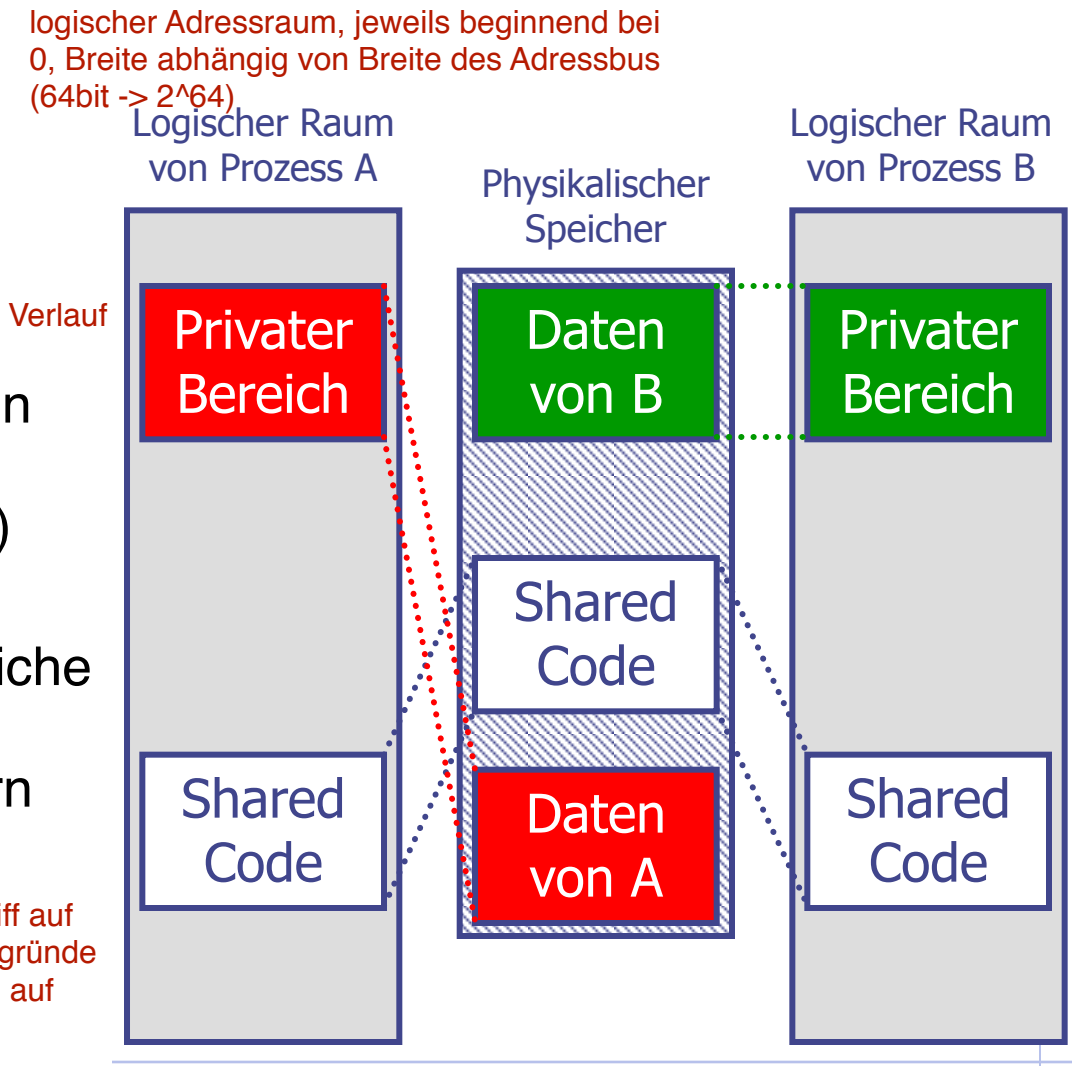


Zusammenhang Prozesse und Programme

Prozesse sind formale getrennt, werden aber von den Betriebssystemen gemischt um sie besonders effizient auszuführen

- Mehrere Prozesse können dasselbe Programm mit unterschiedlichen Daten ausführen
- Beispiel
  - Auf einer Workstation wird ein Webbrowser von zwei Benutzern (lokal und remote) gestartet
  - In beiden Fällen wird der gleiche Browsercode aber mit unterschiedlichen Parametern ausgeführt

Prozesse haben keinen direkten Zugriff auf den physischen Speicher; Sicherheitsgründe und Übertragbarkeit von Programmen auf verschiedene Rechner



Adressräume für Prozesse

- Logischer Adressraum eines Prozesses
  - Gesamtheit aller gültigen Adressen, auf die der Prozess zugreifen darf

  $\Rightarrow$ Adressräume sind gegenseitig geschützt

- Es sind mehrere Relationen zwischen Adressräumen und Prozessen möglich
  - Ein Prozess besitzt genau einen Adressraum (UNIX-Prozess)
  - Mehrere Prozesse teilen sich einen Adressraum (Threads)
  - Ein Prozess wechselt von einem Adressraum zum anderen Adressraum

UNIX-Prozess: ursprüngliche Idee, aber ineffizient

neuer sind Threads

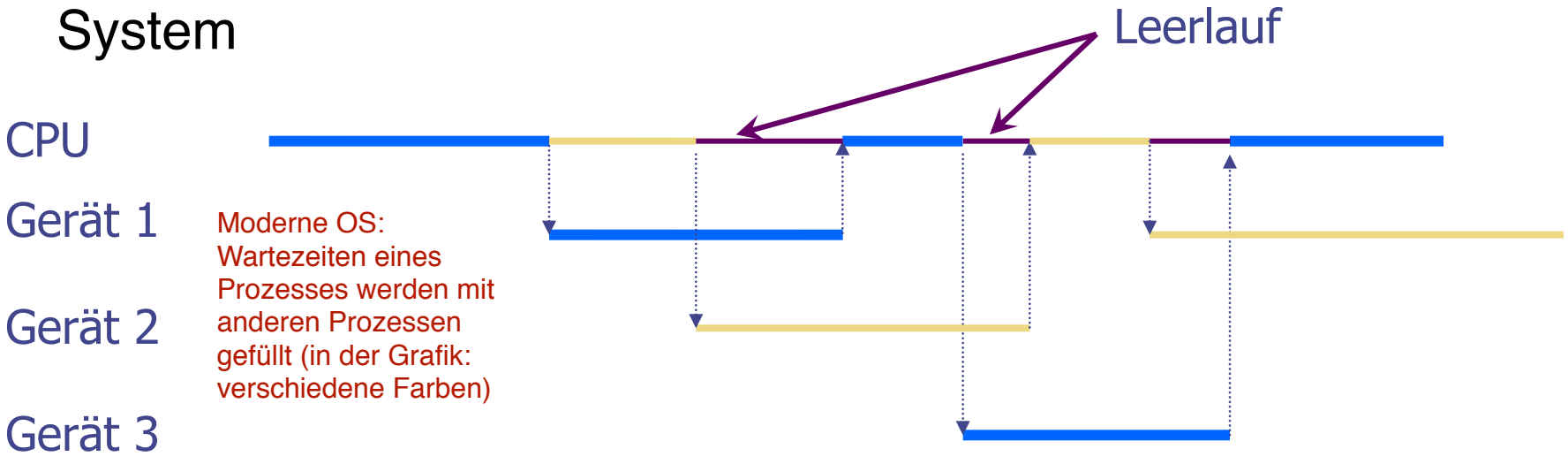
selten wechselt ein Prozess den Adressraum

Ausführung von Prozessen

- Einfachste Rechnerbetriebsart ⇒ Stapelbetrieb (batch mode)

  Unterbrechung auch nicht in Wartezeiten, dann blockiert der Prozess die CPU bis er weiter arbeiten kann

  – Der aktive Prozess wird unterbrechungsfrei – ohne Unterbrechung durch andere konkurrierende Prozesse – ausgeführt
  – Mehrere Prozesse werden sequentiell abgearbeitet
- Problem: Während der Kommunikation mit z.B. E/A-Geräten bleibt die CPU ungenutzt ⇒ Leerlaufzeiten und ineffiziente Ausführung, große Aufträge blockieren das gesamte System

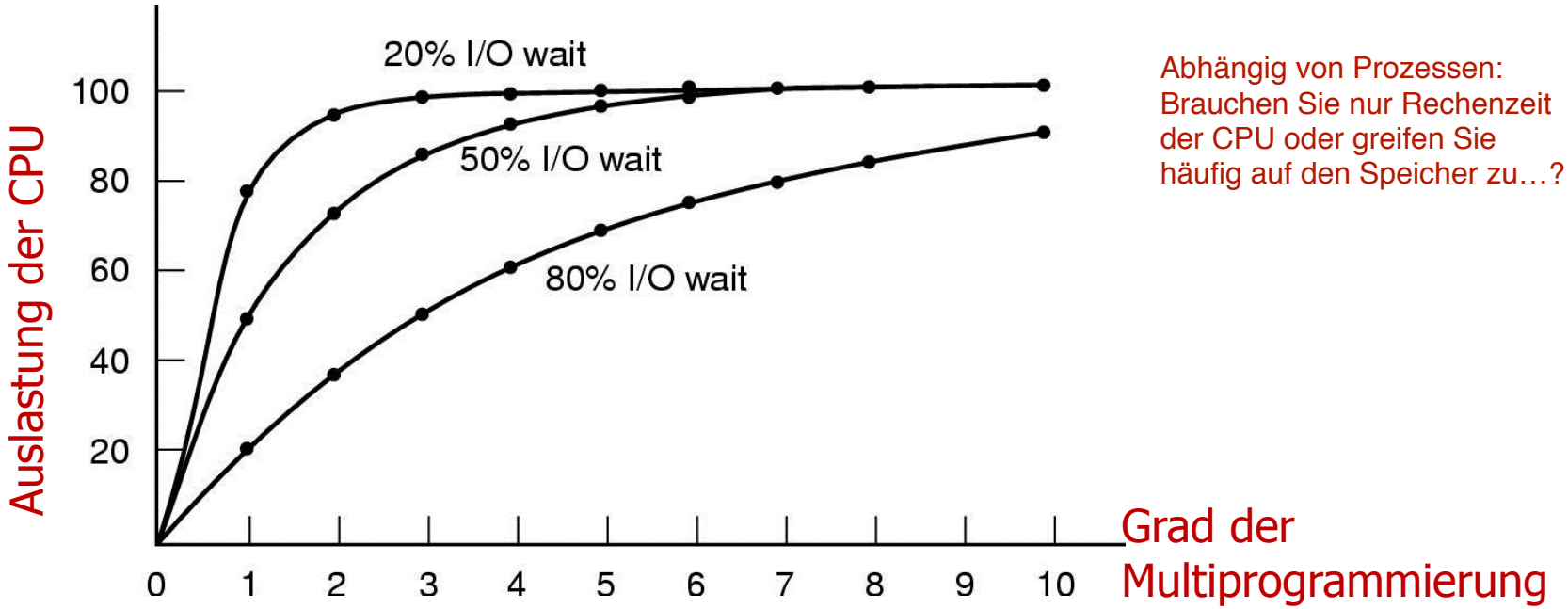


Modellierung der Multiprogrammierung

- Wie viele Prozesse sind für "genau richtige" Auslastung notwendig?
- Keine allgemeine Antwort möglich. Annahmen
  - Ein Prozess verbringt einen Anteil *p* seiner Zeit mit Warten auf E/A-Operationen

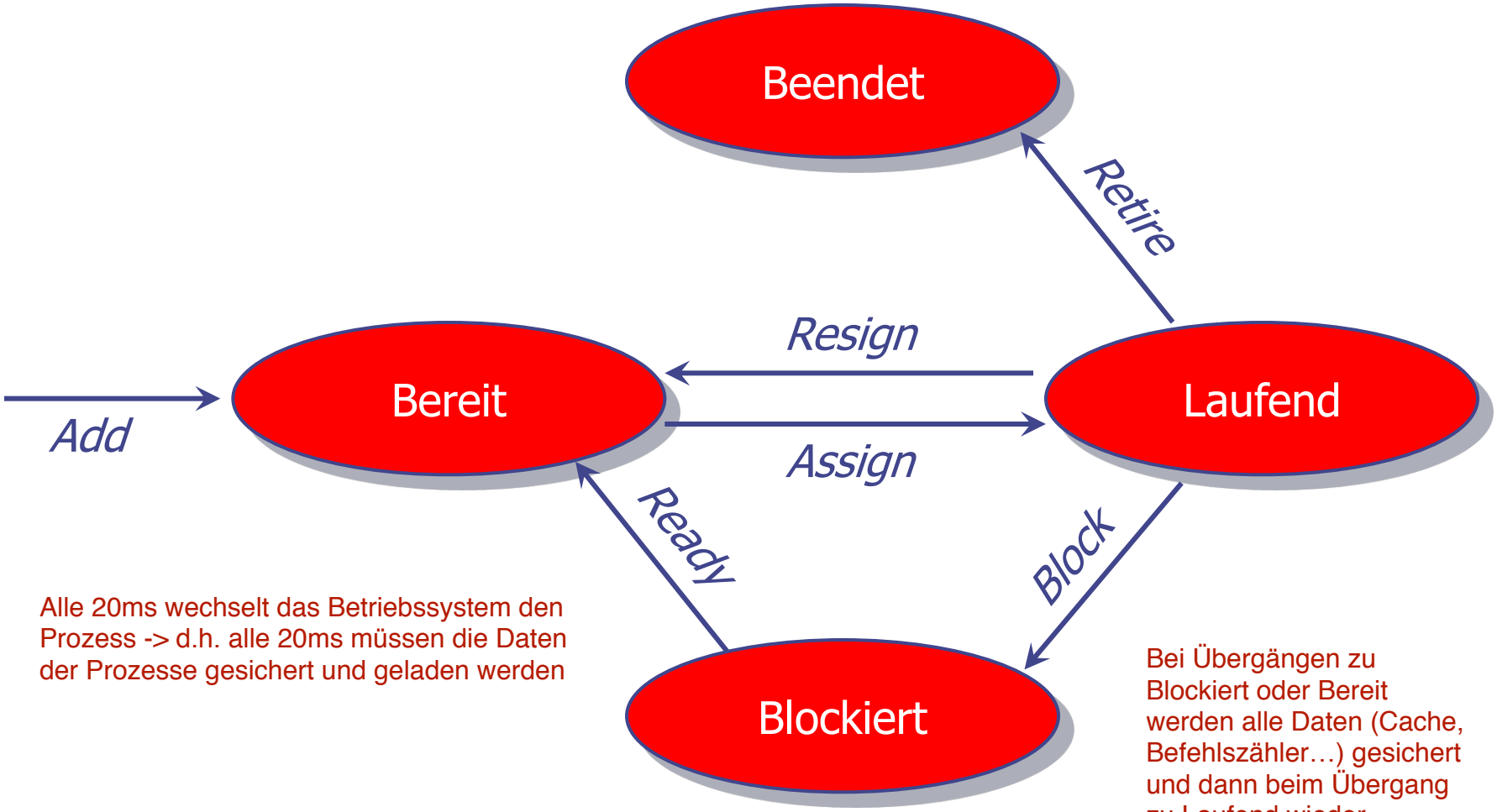
Schätzen, Raten von vermutlichem Nutzungsverhalten, da nicht berechenbar ist, was der Nutzer als nächstes tun wird

  - Wahrscheinlichkeit p^n = *n* Prozesse warten gleichzeitig auf E/A-Ende
  - Ausnutzung der CPU: $A = 1 - p^n$
  - *n* = Grad der Multiprogrammierung (Degree of Multiprogramming)



Abhängig von Prozessen: Brauchen Sie nur Rechenzeit der CPU oder greifen Sie häufig auf den Speicher zu…?

Prozesszustände



Alle 20ms wechselt das Betriebssystem den Prozess -> d.h. alle 20ms müssen die Daten der Prozesse gesichert und geladen werden

Bei Übergängen zu Blockiert oder Bereit werden alle Daten (Cache, Befehlszähler…) gesichert und dann beim Übergang zu Laufend wieder geladen

Prozesszustände

- Ein Prozess kann sich – abhängig vom aktuellen Status – in unterschiedlichen Zuständen befinden
  - Rechnend, Laufend (Running): Der Prozess ist im Besitz des physikalischen Prozessors und wird aktuell ausgeführt
  - Bereit (Ready): Der Prozess hat alle notwendigen Betriebsmittel und wartet auf die Zuteilung des/eines Prozessors
  - Blockiert, Wartend (Waiting): Der Prozess wartet auf die Erfüllung einer Bedingung, z.B. Beendigung einer E/A-Operation und bewirbt sich derzeit nicht um den Prozessor
  - Beendet (Terminated): Der Prozess hat alle Berechnung beendet und die zugeteilten Betriebsmittel freigegeben

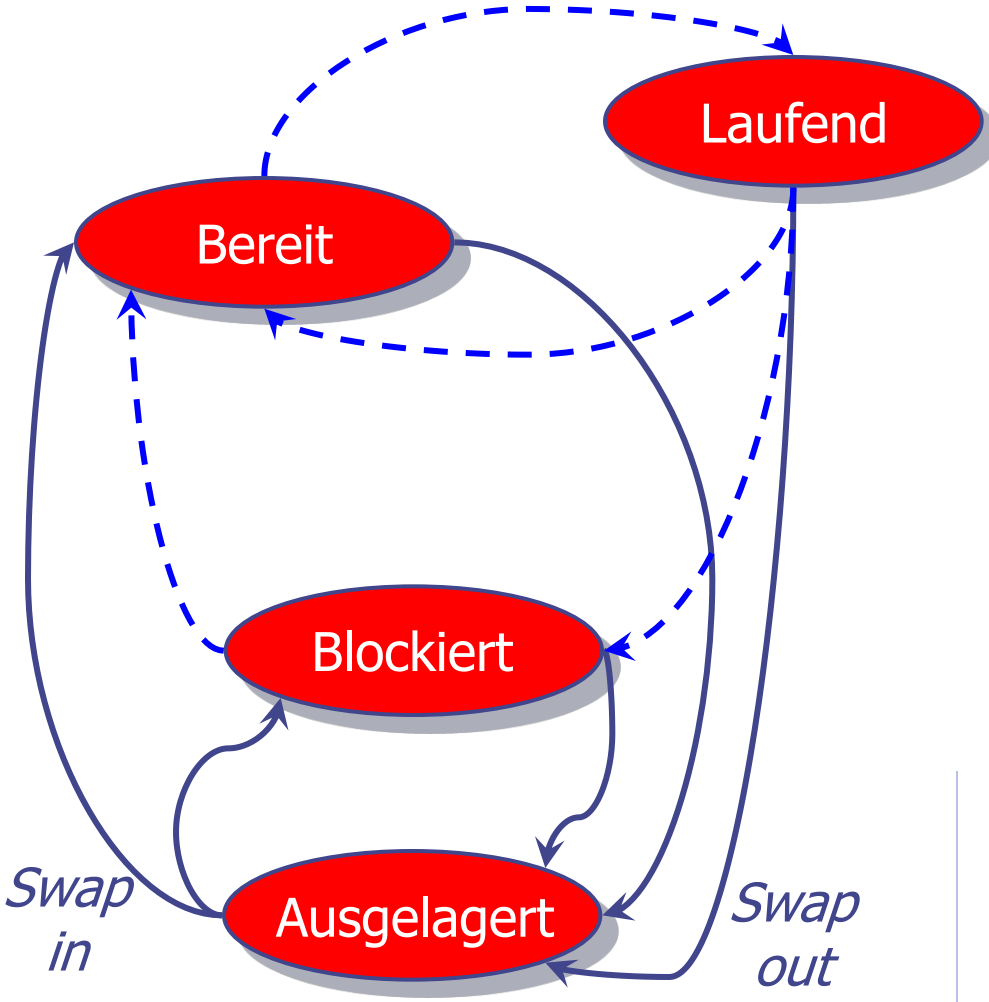
Zustandsübergänge

- Erlaubte Übergänge

Add:	Ein neu erzeugter Prozess wird in die Klasse bereit aufgenommen
Assign:	Infolge des Kontextwechsels wird der Prozessor zugeteilt
Block:	Aufruf einer blockierenden E/A-Operation oder Synchronisation bewirkt, dass der Prozessor entzogen wird
Ready:	Nach Beendigung der blockierenden Operation wartet der Prozess auf erneute Zuteilung des Prozessors
Resign:	Einem laufenden Prozess wird der Prozessor – aufgrund eines Timer-Interrupts, z.B. Zeitscheibe abgelaufen – entzogen
Retire:	Der laufende Prozess terminiert und gibt alle Ressourcen wieder frei

Erweitertes Zustandsmodell

- Wegen Speichermangel werden oft ganze Adressräume ausgelagert (Swapping) ⇒ dem Prozess fehlt auch Arbeitsspeicher
  - Zusatzzustand *Ausgelagert*
  - Zusatzübergänge *swap in* und *swap out*
- Nach der Einlagerung kann der Prozess in den Zustand *Laufend* oder *Blockiert* wechseln, abhängig von aktuellen blockierenden Operationen



Prozesse im Kontext von Betriebssystemen

- Implementierung von Prozessen in BS durch Datenstruktur *Prozesskontrollblock* (Process Control Block, PCB)

⇒PCB = verwaltungs-technischer Repräsentant des Prozesses

- PCB enthält ein Abbild des Registersatzes des realen Prozessors, das den Prozesszustand definiert
  - Prozessidentifikation
  - Bereich für die aktuellen Registerwerte
  - Zustandsvariable (Prozesszustand)
  - Information über Betriebsmittel
  - Hinweise auf Eltern- bzw. Kindprozesse
  - Zugeteilter Prozessor in MIMD-Systemen

Prozesskontext

- Prozesskontext
  - Beschreibt den Zustand einer Funktionseinheit im größeren Detail
  - Auch als Arbeitsumgebung bezeichnet
  - Unterteilung in
    - Ablaufumgebung
    - Verknüpfungsumgebung
- Ablaufumgebung eines Prozesses enthält
  - Befehlszähler, Befehlsregister, Prozessorstatuswort, Adressregister, Seitentabelle, Unterbrechungsmasken, Zugriffsangaben usw.
  - Adressraum, der zusätzlich nach Daten- und Befehlsadressen getrennt sein kann
- Verknüpfungsumgebung besteht aus
  - Datenregistern, Indexregistern, Stapelzeiger usw.

Daten des Prozesses, die beim Wechsel gesichert werden müssen

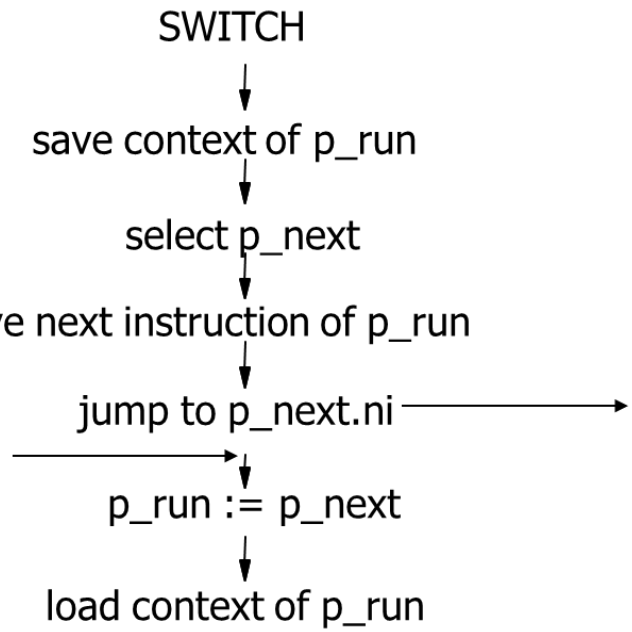
Linux PCB

```
struct task_struct {
volatile long state;
long counter;
long priority;
unsigned long signal;
unsigned long blocked;
unsigned long flags;
int errno;
long debugreg[8];
struct exec_domain *exec_domain;
struct linux_binfmt *binfmt;
struct task_struct *next_task, *prev_task;
struct task_struct 0, *prev_run;
unsigned long saved_kernel_stack;
unsigned long kernel_stack_page;
int exit_code, exit_signal;
unsigned long personality;
int dumpable:1;
int did_exec:1;
int pid;
int pgrp;
int tty_old_pgrp;
int session;
int leader;
int groups[NGROUPS];
struct task_struct *p_opptr, *p_pptr, *p_cptr, *p_ysptr, *p_os
struct wait_queue *wait_chldexit;
unsigned short uid,euid,suid,fsuid;
unsigned short gid,egid,sgid,fsgid;
unsigned long timeout, policy, rt_priority;
unsigned long it_real_value, it_prof_value, it_virt_value;
unsigned long it_real_incr, it_prof_incr, it_virt_incr;
struct timer_list real_timer;
long utime, stime, cutime, cstime, start_time;
unsigned long min_flt, maj_flt, nswap, cmin_flt, cmaj_flt, cnswap;
int swappable:1;
unsigned long swap_address;
unsigned long old_maj_flt; /* old value of maj_flt */
unsigned long dec_flt;
unsigned long swap_cnt;
struct rlimit rlim[RLIM_NLIMITS];
unsigned short used_math;
char comm[16];
int link_count;
struct tty_struct *tty; /* NULL if no tty */
struct sem_undo *semundo;
struct sem_queue *semsleeping;
struct desc_struct *ldt;
struct thread_struct tss;
struct fs_struct *fs;
struct files_struct *files;
struct mm_struct *mm;
struct signal_struct *sig;
};
```

Beispiel eines Prozesskontrollblocks Linux 2.6.11

Prozessumschaltung

- Aktuell aktiver Prozess wird aus Zustand *Laufend* in anderen Zustand versetzt (Zeitscheibe verbraucht, Interrupt, …)
  - Notwendige Aktion: Sicherung des Kontextes
- Ein bereiter Prozess wechselt in den Zustand *Laufend*
  - Notwendige Aktion: Laden des Kontextes des neuen Prozesses



Automatisches Umschalten

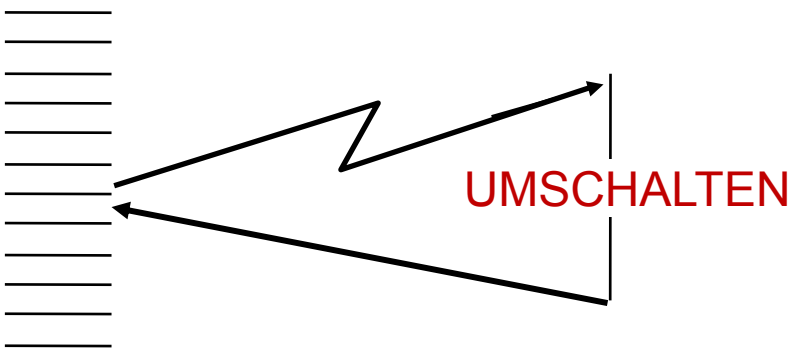
Standard-Timer (Zeitscheibe): 20ms
—> Eindruck von Gleichzeitigkeit vermitteln

- Notwendig: Intervalluhr oder Wecker (timer) als Hardware-Einrichtung mit folgenden Funktionen
  - Vorgabe einer Frist (Stellen des „Weckers”)
  - Unterbrechung bei Fristablauf („Wecken”)
- Programme bleiben unverändert, da das Umschalten von außen ausgelöst wird und zu jedem beliebigen Zeitpunkt stattfinden kann (Unterbrechungen nach wie vor erlaubt)

längere Zeitblöcke (30sec) wären effizienter, aber nicht nutzerfreundlich, da dann Prozesse für diese Zeit nicht reagieren würden nach einem Wechsel

Freiwilliges Umschalten

Automatisches Umschalten



Auswahl des nächsten laufenden Prozesses

- Strategien zur Überführung der Prozesse *bereit* -> *laufend* sind wichtig für die Effizienz eines Systems
- Auswahlprozess beinhaltet die dynamische Auswertung von verschiedener Kriterien, z.B.
  - Prozessnummer (zyklisches Umschalten)
  - Ankunftsreihenfolge
  - Fairness und Priorität (Konstant / Dynamisch)
  - Einhaltung von geforderten Fertigstellungspunkten
- Nach der Wahl müssen die Attribute aller anderen Prozesse angepasst werden ⇒ Detailliert in Kapitel „Scheduling“

Prioritäten: höchste Priorität hat ein Prozess, der gerade Nutzereingaben verarbeitet

Qualität des Scheduling entscheidend für die Geschwindigkeit des Betriebssystems

Bei sicherheitskritischen Anwendungen werden auch Deadlines verwendet, bis zu denen der Prozess auf jeden Fall abgearbeitet sein muss

2.2 Thread-Modell

- Das Prozess-Konzept bietet 2 unabhängige Einheiten
  1. Einheit zur Ressourcenbündelung: ein Prozess verfügt über
     - (Virtuellen) Adressraum
     - Beschreibende Datenstrukturen wie PCB
     - Quelltexte und Daten
     - Weitere Betriebsmittel wie E/A-Geräte, Dateien, Kindprozesse …
  2. Ausführungseinheit, der der reale Prozessor zugeteilt wird
     - Ausführungsablauf mit einem oder mehreren Programmen
     - Verzahnte Ausführung mit anderen Prozessen
     - Zustände (bereit, laufend, blockiert, terminiert, …)
     - Priorität
- Bezeichnungen
  - Bündelungseinheit (1): *Prozess* (*Task*)
  - Ausführungseinheit (2): *Leichtgewichtsprozess* (*Thread*)

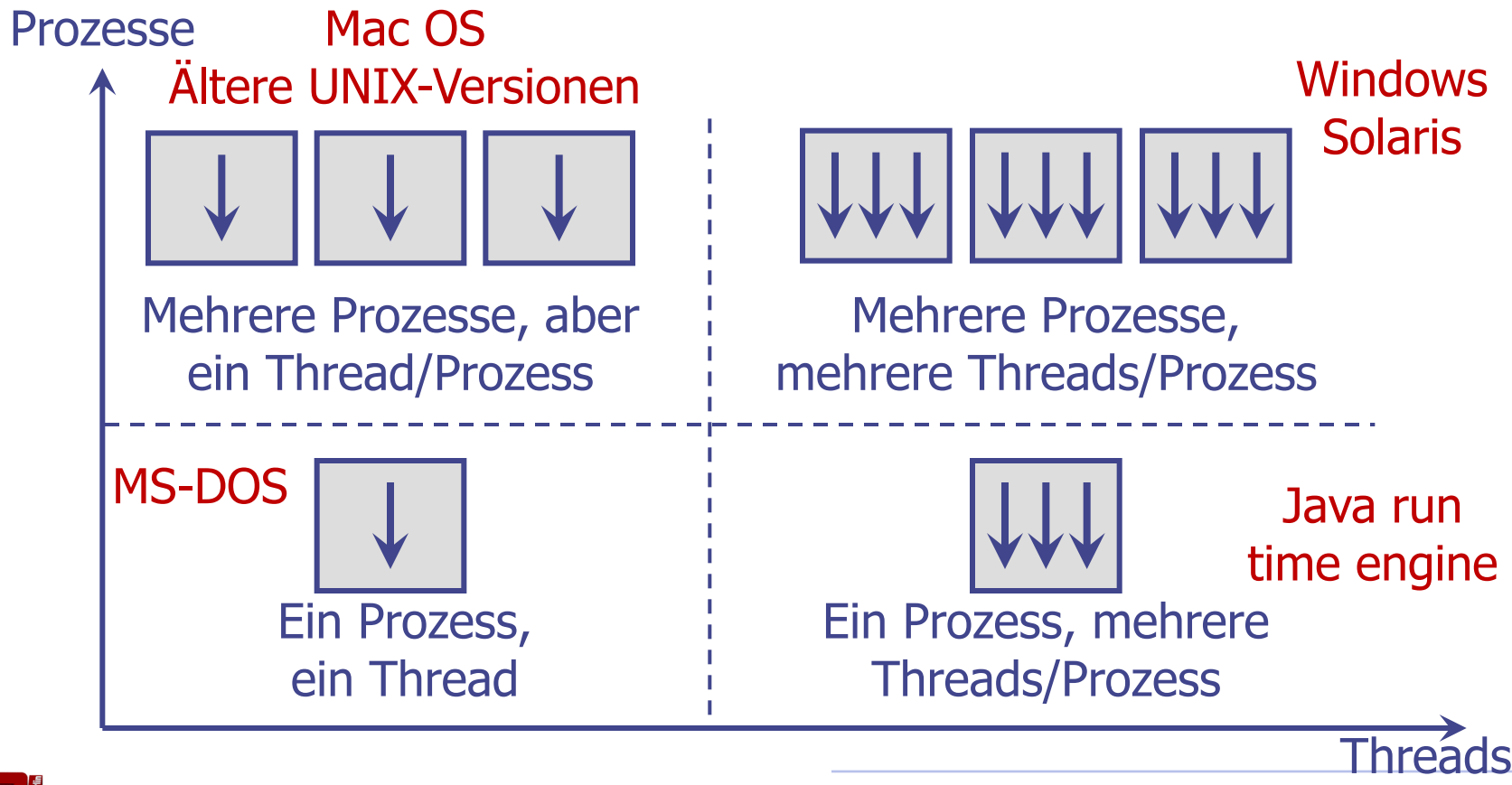
Definition eines Threads

- Thread: Teil eines Prozesses mit folgenden Eigenschaften
  - Keine vollständige Prozesstabelle wie der ursprüngliche Prozess
  - Nebenläufige Ausführung zum Prozess
  - Operiert im selben virtuellen und realen Adressraum
  - Entspricht einem separaten Kontrollfluss dieses Prozesses
- Ein Thread enthält eine eigene Threadtabelle mit separatem Befehlszähler, eigenem Code- und Datenteil und vollständiger Verknüpfungsumgebung

Threads teilen sich einen Adressraum, deswegen funktioniert der Wechsel zwischen Threads eines Prozesses sofort (da keine Daten gespeichert werden müssen)

Zusammenhang Prozesse und Threads

- Multi-Threaded: mehrere Threads innerhalb eines Prozesses
- Single-Threaded: ein Thread/Prozess (klassische Prozesse)



Beispiel zur Nutzung von Threads

- Gegeben: Webserver auf einer dedizierten Maschine
  - Daten vergangener Anfragen werden in Cache solange aufbewahrt, bis der Speicher verbraucht ist
  - Älteste Datensätze werden durch neue ausgetauscht

- Realisierung mit einem Thread
  - Endlosschleife zur Annahme von Anfragen
  - Die Anfragen werden sequentiell bearbeitet
    - Sind die geforderten Daten im Cache ⇒ Kurze Antwortzeit
    - Andererseits wird der Prozess blockiert, bis die Daten von der Festplatte gelesen sind

    ⇒ Leerlauf und geringe CPU-Auslastung

Problem: Nach zu langer Zeit ohne Reaktion, wird Anfrage sicher noch einmal geschickt —> Timeout

Beispiel zur Nutzung von Threads (2)

- Realisierung mit mehreren Threads
  - Thread Dispatcher: liest ankommende Anfragen
  - Thread Worker: bearbeitet eine einzelne Anfrage
- Ablauf
  - Dispatcher empfängt die Anfrage und kreiert/weckt einen Worker
  - Worker wechselt sobald möglich in laufend, überprüft Anfrage
    - Daten im Cache $\Rightarrow$ Bearbeitung sofort
    - Daten auf Festplatte

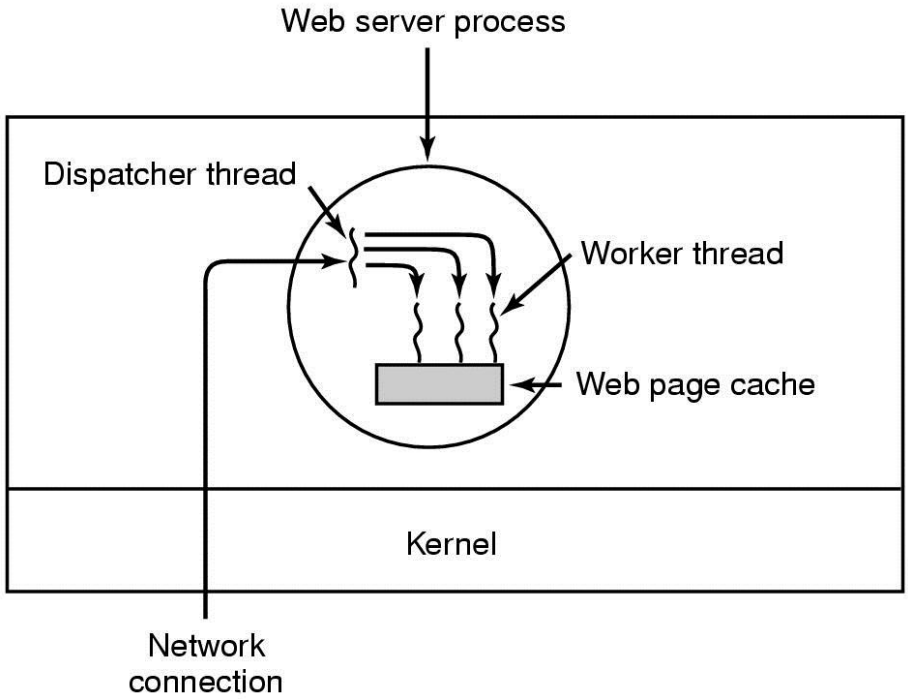
      $\Rightarrow$startet Leseoperation und versetzt sich in Zustand blockiert.
      Leseoperation beendet

      $\Rightarrow$Wechsel in Zustand *Bereit*

      $\Rightarrow$Worker bewirbt sich erneut um die CPU
- Vorteil
  - Hohes Maß an Parallelität zwischen Lese- und Rechenzugriffen

Webserver mit mehreren Threads

Dispatcher nimmt Anfragen entgegen und gibt zurück, dass Anfrage angekommen ist. Dann gibt er die Aufgabe an einen der Worker weiter



Jede Anfrage wird in mehrere Threads mit unterschiedlicher Priorität aufgespalten

Bsp. Textverarbeitung: Threads unterschiedlicher Priorität
- Texteingabe (erscheint sofort)
- Drucken / Dokument senden
- Backup-Kopie erstellen
- Rechtschreibprüfung

Code Worker

```
while(TRUE) {
        wait_for_work(&buf);
        look_in_cache(&puf, &page);
        if(page_not_in_cache(&page);
                read_page_from_disk
                                (&buf,&page);
        return_page(&page); }
```

Code Dispatcher

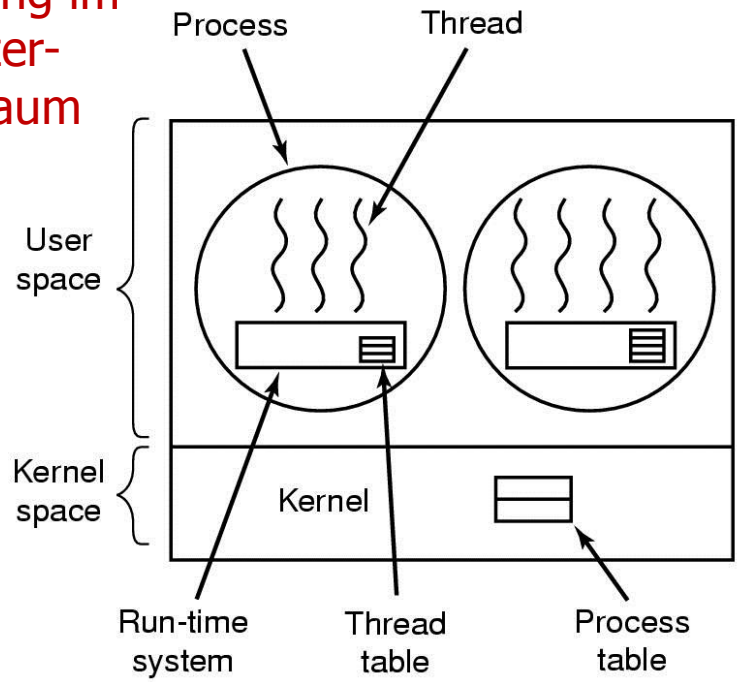
```
while(TRUE) {
                get_next_request(&buf);
                handoff_work(&buf); }
```

Threadtypen

- Grundsätzlich werden Threads aufgeteilt in
  - Kernel-Level-Threads (KL-Threads): realisiert im Kernadressraum
  - User-Level-Threads (UL-Threads): realisiert im Benutzeradressraum
- Hybride Realisierung ist allerdings auch möglich

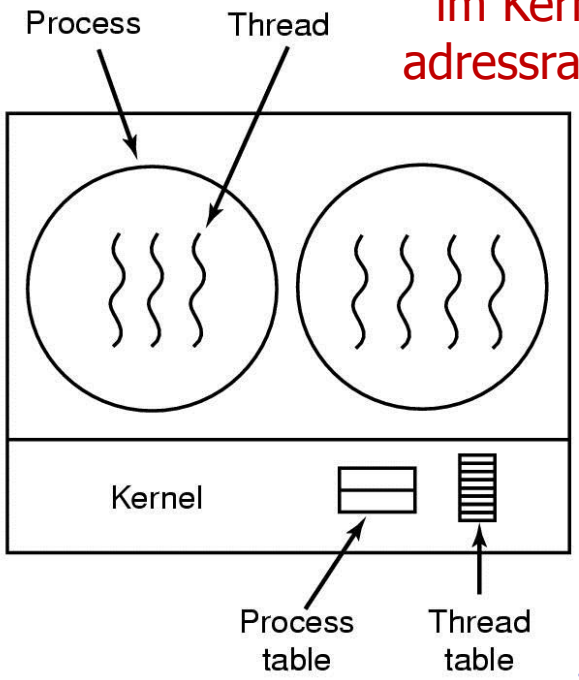
große Freiheit, da das OS nur einen Container für User-Threads zur Verfügung stellt, wie viele Threads mit welchen Aufgaben erstellt und ausgeführt werden, entscheidet der User

Realisierung im Benutzer-adressraum



Threads werden vom OS verwaltet, d.h. alle Threads müssen dort "gemeldet" werden. Vorteil: OS kann entscheiden, dass Prozess mit vielen Threads mehr

Realisierung im Kern-adressraum

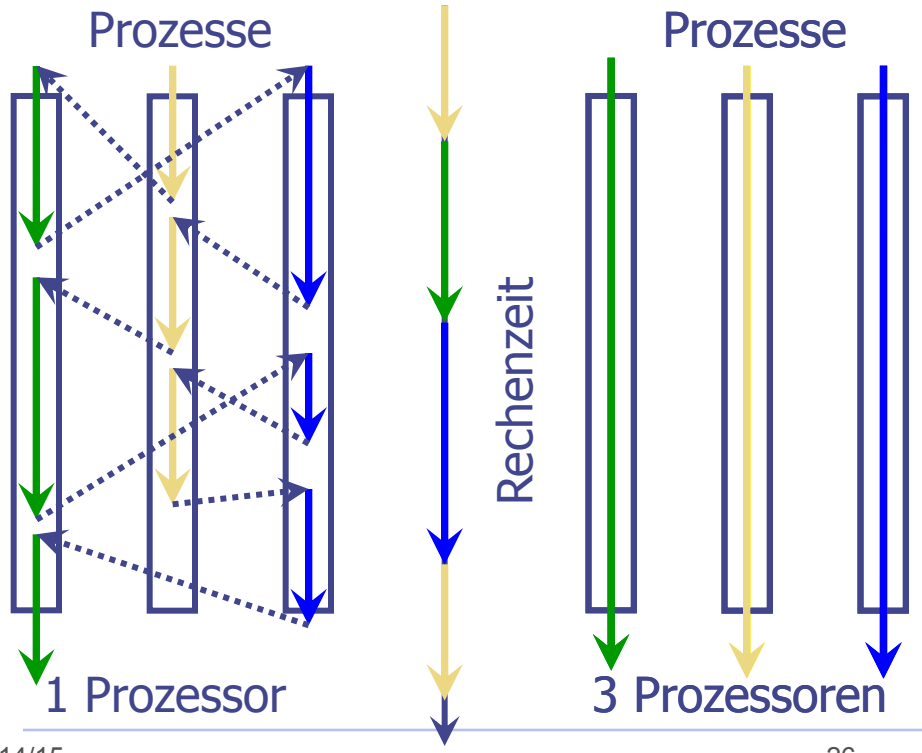


2.3 Nebenläufigkeit und Parallelität

- Nebenläufigkeit (Concurrency = Concurrent Execution)
  - Logische simultane Verarbeitung von Operationsströmen, d.h. es wird der Eindruck erweckt, dass die Prozesse gleichzeitig ablaufen
  - ⇒Verzahnte Ausführung auf einem Einprozessorsystem
- Parallelität
  - Die Operationsströme werden tatsächlich simultan ausgeführt
  - Mehrfache Verarbeitungselemente, d.h. Prozessoren oder andere unabhängige Architekturelemente, sind zwingend notwendig
- Bemerkungen
  - Nebenläufigkeit und Parallelität setzen einen kontrollierten Zugang zu gemeinsamen Ressourcen voraus
  - Nebenläufiges Programm auf Parallelsystem ⇒ paralleles Programm

Zusammenhang Nebenläufigkeit und Parallelität

- Nebenläufigkeit = Zuordnung mehrerer Prozesse zu mindestens einem Prozessor
  Illusion der Gleichzeitigkeit für den Nutzer durch schnellen Wechsel zwischen Prozessen
- Parallelität = Zuordnung mehrerer Prozesse zu mindestens zwei Prozessoren
  echte Gleichzeitigkeit
- Parallelität ist eine Teilmenge der Nebenläufigkeit



Nebenläufigkeit

- Nebenläufigkeit findet in unterschiedlichen Ausprägungen auf der Hardwareebene statt
  - Mehrere Einheiten innerhalb eines Prozessors
    - Instruktionspipeline, mehrfache Recheneinheiten, …
  - Mehrere E/A- und DMA-Kontroller
    - Prozesse und Datentransfers werden nebenläufig ausgeführt
  - Mehrere allgemein einsetzbare Prozessoren
    - Parallele Ausführung von Prozessen

Parallelität

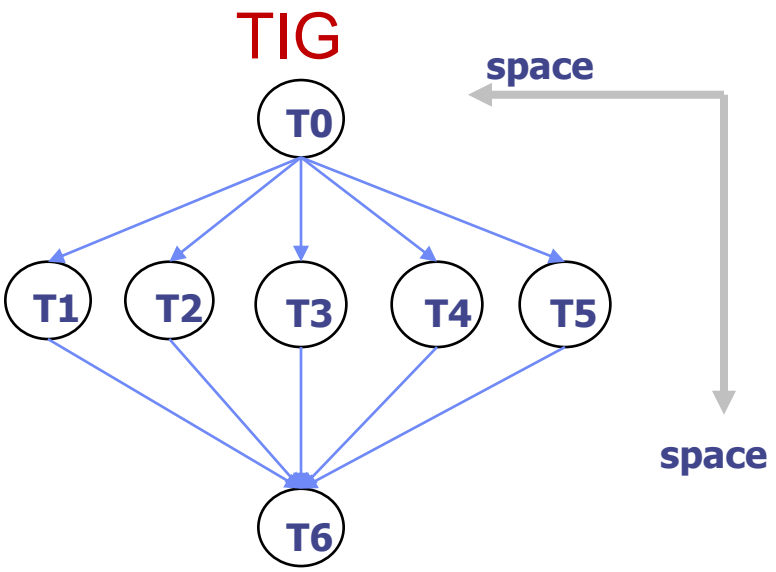
- Parallelität (Parallelism, Parallel processing)
  - Von parallelen Prozessen wird gesprochen, wenn zwei oder mehrere Prozesse tatsächlich simultan ausgeführt werden
  - Dazu sind jedoch zwei oder mehrere aktive Verarbeitungselemente (z.B. Prozessoren) notwendig
- Elementare Kontrollstrukturen in sequentiellen, imperativen Programmiersprachen
  - Sequenz
  - Wiederholung
  - Verzweigung
  - Einschub (Prozedur, Unterprogramm)
- Für alle Grundkonstrukte gibt es analoge Konstrukte für expliziten Parallelismus
- Diese werden explizit vom Programmierer vorgegeben

2.4 Beziehungen zwischen Prozessen

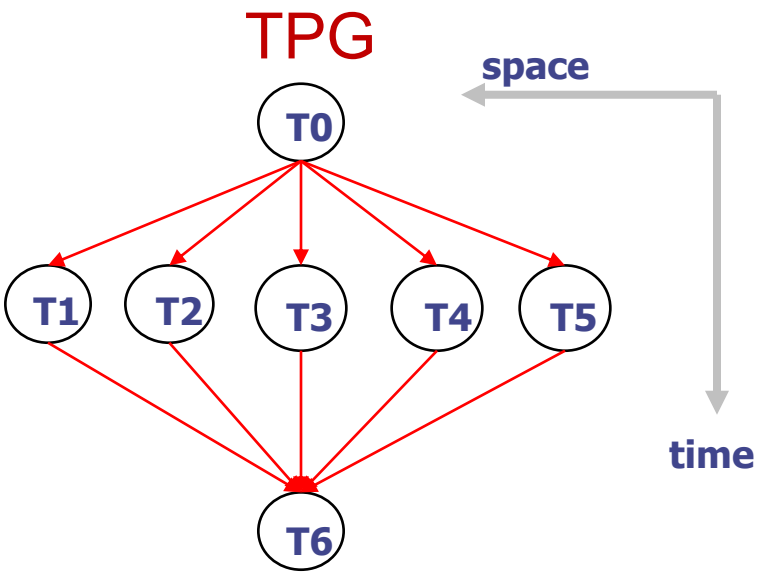
- Prozesse können in diversen Beziehungen stehen:
  - Eltern-Kind-Beziehung: Ein Prozess erzeugt einen weiteren Prozess
  - Vorgänger-Nachfolger-Beziehung: Ein Prozess darf erst starten, wenn ein anderer Prozess beendet ist
  - Kommunikationsbeziehung: Zwei (oder mehr) Prozesse kommunizieren miteinander
  - Wartebeziehung: Ein Prozess wartet auf etwas, was von einem anderen Prozess kommt
  - Dringlichkeitsbeziehung: Ein Prozess ist wichtiger (dringlicher) als ein anderer
- … und viele andere mehr

Prozessgraphen

- Prozessbeziehungen werden oft als Graphen dargestellt, meist gerichtet, einige sind azyklisch, (DAG = directed acyclic graph)
- Beispiele
  - Prozesskommunikationsgraph (TIG, task interaction graph)
  - Prozessvorgängergraph (TPG, task precedence graph)



Pfeile definieren Kommunikationsfluss



Pfeile definieren Vorgängerrelation

2.5 Elementare Koordinationsoperationen

- Prozesse, die isoliert und unabhängig voneinander ablaufen, müssen nicht koordiniert werden
- Dies ist allerdings eher selten, da
  - Ein Teil der Betriebsmittel nur exklusiv belegt werden kann
  - Mehrere Prozesse für die Lösung einer gemeinsamen Aufgabe eingesetzt werden
  - Prozesse tauschen Daten aus unterschiedlichen Quellen aus

⇒Unterstützung der Prozessinteraktion ist eine grundlegende Aufgabe der Systemsoftware

- Grundsätzlich gibt es zwei Formen der Interaktion
  - Konkurrenz häufigste Form
  - Kooperation Zusammenarbeit bei einer Aufgabe, z.B. Client-Server

Prozessinteraktion: Konkurrenz

- Konkurrenz
  - Zwei oder mehrere Prozesse bewerben sich gleichzeitig um ein exklusiv benutzbares Betriebsmittel, z.B. Drucker
  - ⇒Synchronisationsmechanismen notwendig

Bsp. für Betriebsmittel, das nur ein Prozess gleichzeitig haben kann: Soundkarte, Drucker, auch Variablen in Programmen

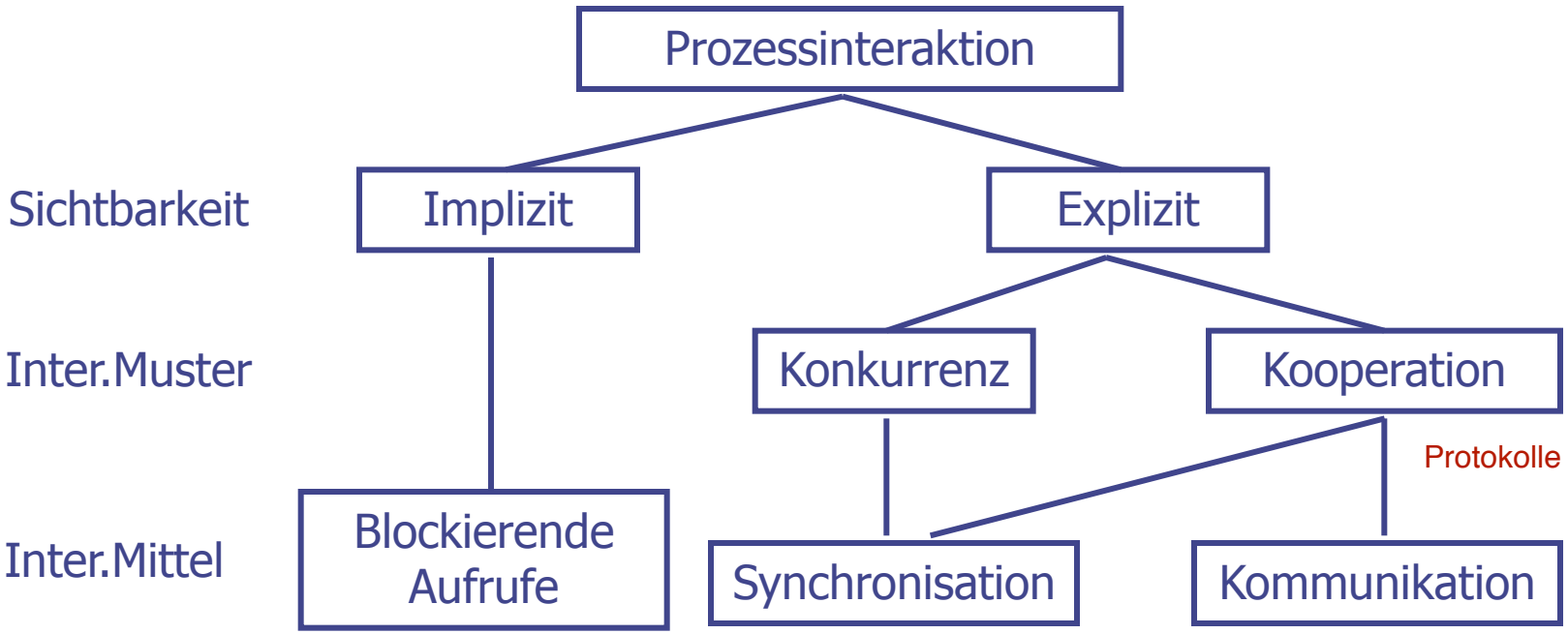
- Durch geeignete Koordinierung muss eine Serialisierung der Zugriffsversuche erreicht werden
- Bei *n* konkurrierenden Prozessen werden *n* - 1 Prozesse z.B. zeitlich verzögert
- Die zeitliche Abstimmung konkurrierender Prozesse wird als Prozesssynchronisation bezeichnet

Prozessinteraktion: Kooperation

- Prozesse tauschen gezielt Informationen untereinander aus, z.B. Erzeuger / Verbraucher-Situation
  - Erzeuger füllt einen Pufferplatz mit Daten
  - Verbraucher entnimmt die Daten aus dem Puffer
- Kooperierende Prozesse müssen
  - Von der Existenz aller anderen beteiligten Prozesse wissen
  - Ausreichende Informationen über diese besitzen, z.B. Art und Funktionalität der Schnittstellen
- Klassisches Beispiel: Client/Server

Explizite / Implizite Prozessinteraktion

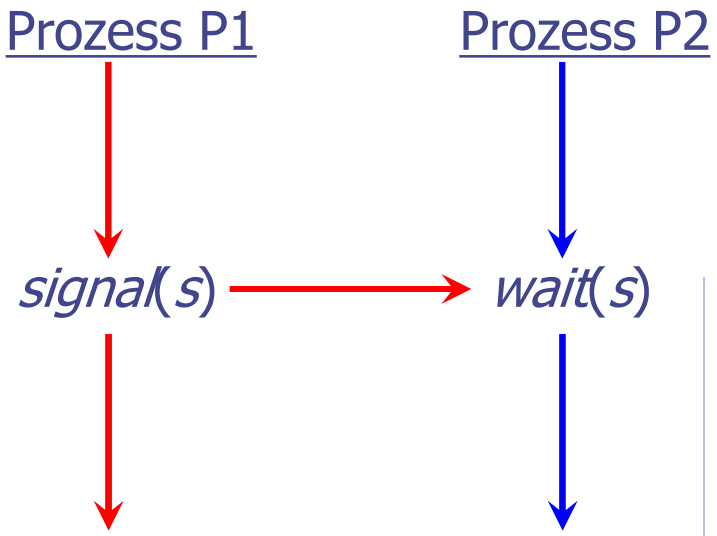
- Implizite Interaktion, wenn ein Prozess eine Systemfunktion aufruft und diese mit anderen Prozessen interagiert
  - ⇒ Der aufrufende Prozess bekommt davon nichts mit
  - ⇒ Wird ggf. für die Dauer der Interaktion geblockt und wieder gestartet, wenn die Ergebnisse vorliegen
- Klassifikation



Signalisierung

- Elementare Aufgabe: Exklusives Sperren/Freigeben einer Variable (Sperrflag) durch konkurrierende Prozesse
  - Voraussetzung: Interagierende Prozesse haben Zugriff auf gemeinsamen Speicherbereich
- Einfachste Form: Herstellung einer Reihenfolgebeziehung (Signalisierung)
  - Prozess P2 wird fortgesetzt, erst nachdem ein Prozess P1 einen bestimmten Abschnitt bearbeitet hat, z.B. warten, bis Prozess A alle Daten auf die Festplatte geschrieben hat
- Operationen signal(s) und wait(s), s Sperrflag

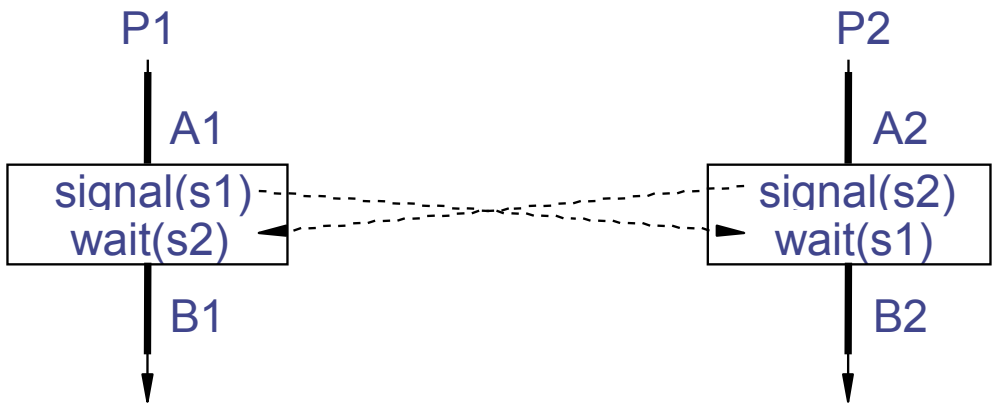
Signal: Bedingung, dass ein Prozess weiter arbeiten kann ist erfüllt -> dies wird signalisiert, dann beginnt der Prozess wieder zu arbeiten



Wechselseitige Synchronisierung

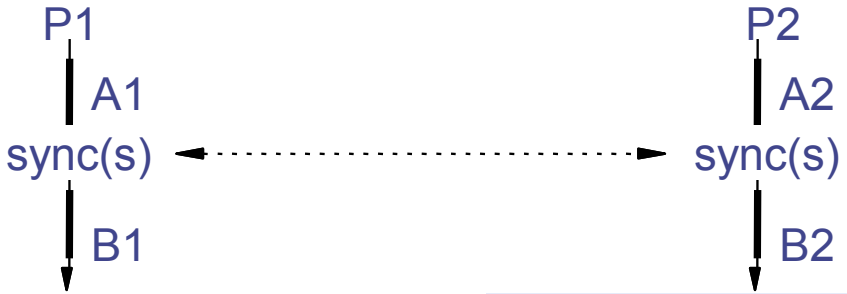
- Symmetrischer Einsatz der Operationen bewirkt, dass sowohl A1 als auch A2 ausgeführt sind, bevor B1 oder B2 ausgeführt werden

A und B sind Abschnitte in einem Prozess, hier ist A jeweils die Voraussetzung für B



- Prozesse P1 und P2 **synchronisieren** sich an dieser Stelle

⇒Zusammenfassung zu Operation **sync (Rendezvous)**



Kritische Abschnitte

- Definition Kritischer Abschnitt (Kritischer Bereich)
  - Operationsfolgen, bei denen eine nebenläufige oder verzahnte Ausführung zu Fehlern führen kann
- Beispiele

  es muss sichergestellt werden, dass nur je ein Prozess gleichzeitig in diesem Bereich ist
  - Zugriff auf exklusiv benutzbare Betriebsmittel
  - Veränderungen gemeinsam benutzter Variablen
- Definition mit Sperrvariablen

  Sperrvariablen funktionieren wie eine Schranke:
  Ein Prozess, der den kritischen Bereich betritt schließt die Schranke

  …

  Sperren (*Sperrvariable*)

  Freigeben (*Sperrvariable*)

  } Kritischer Bereich / Abschnitt

  …
- Durch ein Sperrflag soll sichergestellt werden, dass sich ein einziger Prozess im kritischen Bereich befindet

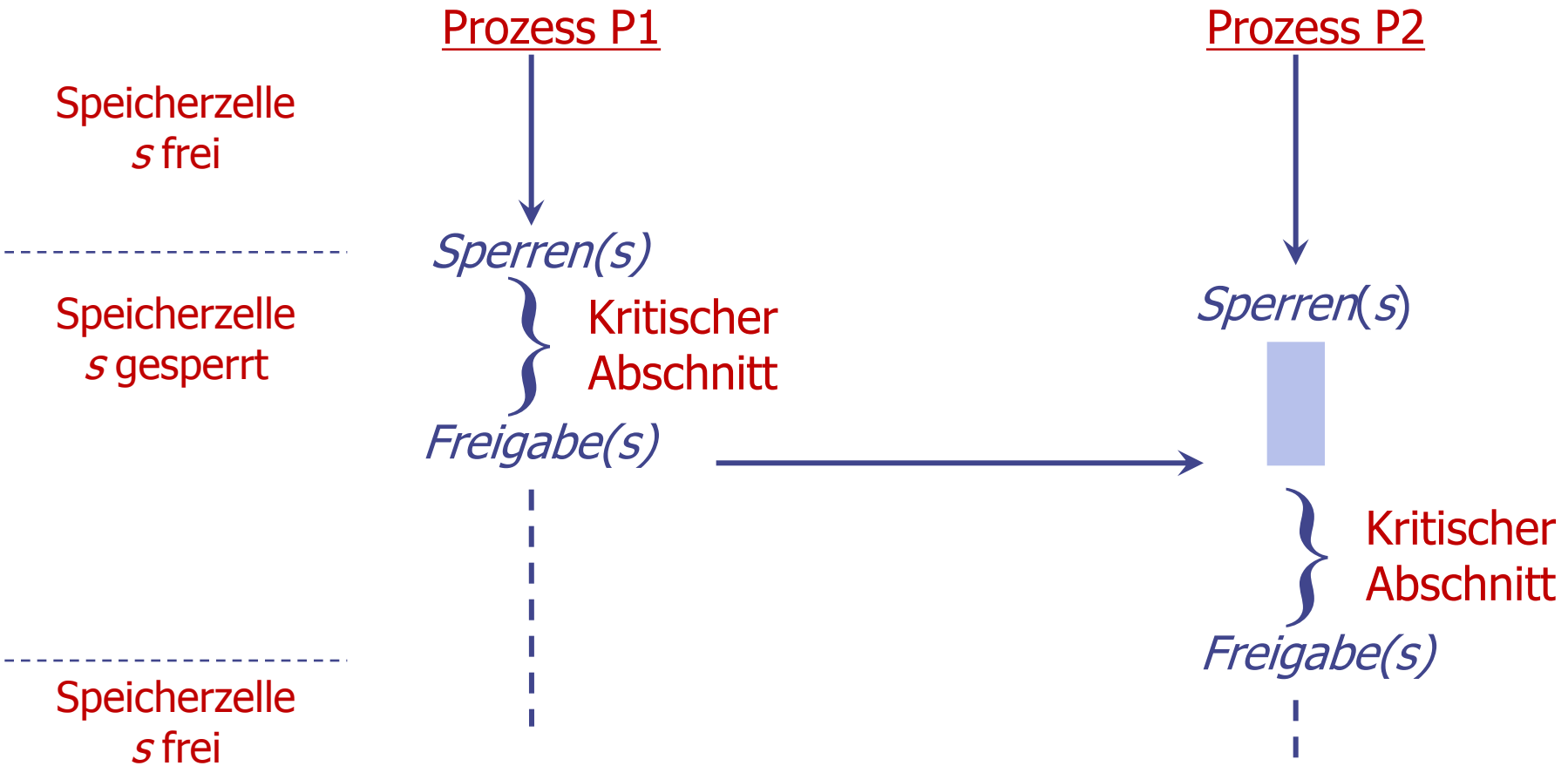
Prozesssynchronisation mittels kritischer Abschnitte

- Ein kritischer Abschnitt darf von nur einem Prozess betreten werden
  - ⇒Prozesse müssen sich gegenseitig ausschließen (*mutual exclusion*)
- Weitere Anforderungen an Behandlung von kritischen Abschnitten

  um Sicherheit zu garantieren: Keine Annahmen treffen ("gambling")
  - Keine Annahmen über Prozessorgeschwindigkeit
  - Keine Annahmen über Anzahl und Reihenfolge von Prozessen
  - Keine Verzögerung von Prozessen in unkritischen Bereichen
  - Keine Verklemmung, d.h. Prozesse dürfen sich nicht gegenseitig blockieren
  - Ein Prozess muss nach endlicher Zeit den kritischen Bereich betreten können
  - Endliche Aufenthaltszeit im kritischen Bereich

Kritische Abschnitte mit Sperrflags

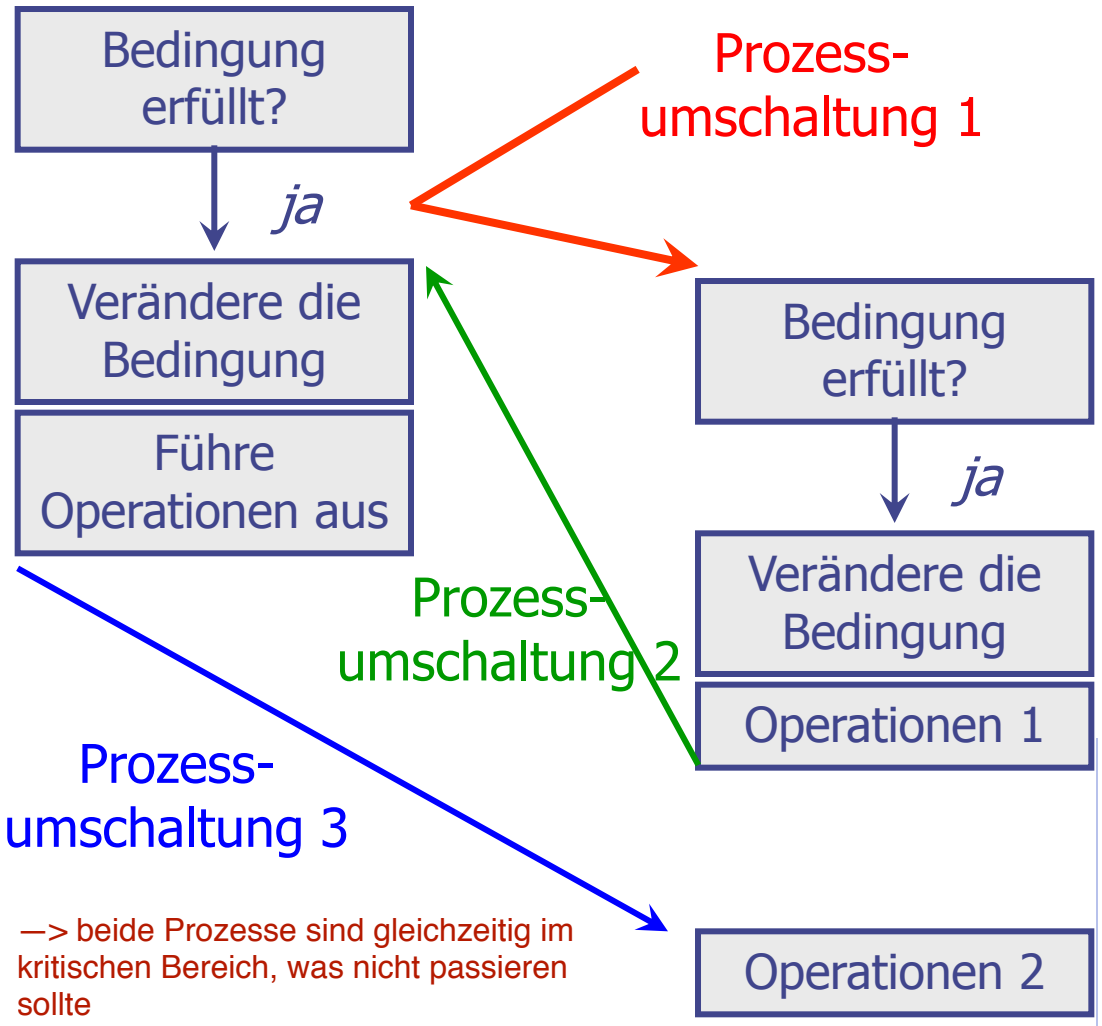
- Kritische Abschnitte können nur unter der Bedingung „keine Nebenläufigkeit“ mit Sperrflags gesichert werden



Probleme durch verzahnte Ausführung

- Bei verzahnter Ausführung kann nicht ausgeschlossen werden, dass zwischen
  - Abfrage der Bedingung UND
  - Darauf folgender Operation

  ein Umschalten stattfindet und ein anderer Prozess die Bedingung ändert



Beispiel: Zimmerbelegung im Hotel

- Es sei ein Hotel mit *anzahl* Zimmern gegeben und jedem Zimmer seien die Attribute *status* (*frei, belegt*) und Gastname zugeordnet
- Routine zur Abfrage/Belegung der Zimmer von einem Terminal

```
[1] Warte auf Signal vom Terminal
[2] if (Freizimmer>0) {  //Initial Freizimmer = anzahl;
[3]    i=SucheZimmer();
[4]    Zimmer[i].status=belegt;
[5]    Zimmer[i].gast=enterName();
[6]    Freizimmer --;
[7]    PrintMessage(Zimmer i reserviert); }
[8] else PrintMessage(Hotel belegt);
```

- Für zwei Buchungsprozesse A und B sind beispielsweise folgende Konstellationen – abhängig von der Verzahnung – möglich
  - A1…A8B1…B8
  - B1…B8A1…A8
  - A1A2A3 B1B2B3 A4A5A6A7A8 B4B5B6B7B8
- Verzahnte Ausführung hinterlässt inkonsistenten Datenbestand. Warum?

Betriebssystemgestützte Mechanismen für Prozessinteraktion

- Betriebssystemunterstützung für Prozesssynchronisation
  - Semaphore (Dijkstra, 1965)
  - Monitore
- Operationen Sperren/Freigabe wirken direkt auf die Prozesszustände (Bereit, Laufend, Blockiert, Beendet)
- Voraussetzung
  - Operationen der Form test_and_set sind als atomare Operationen realisiert

Semaphore

= Schranke mit Warteliste

- Semaphore stellen eine Zählsperre dar und bestehen aus
  - Nicht-negativ initialisiertem Zähler
  - Liste mit Verweisen auf involvierte Prozesse
- Bei negativen Zählerwerten (*nach Dekrementierung des Zählers! siehe Source Code*) werden die anfragenden Prozesse blockiert, bis mind. ein Prozess den kritischen Bereich verlässt
- Grundoperation P(*s*) (*Passieren des Semaphors*)
  - Der aktuelle Wert des Semaphorzählers wird dekrementiert
  - Passieren der Sperre und Eintritt in den kritischen Bereich
- Grundoperation V(*s*) (*Verlassen des kritischen Bereichs*)
  - Austritt aus dem kritischen Bereich
  - Der aktuelle Wert des Semaphorzählers wird inkrementiert
- Alternative Namen sind DOWN/UP

Prozesse, die auf Semaphore warten, sind im Zustand blockiert (bekommen also auch keine CPU-Zeit) und verschwenden so keine Zeit

Beispielrealisierung der Operationen P und V

```
void P (Semaphor s) {
  zaehler(s)--;
  if (zaehler(s)<0) { /* Prozess(e)
  im kritischen Bereich */
        Zustand des aktuellen
          Prozesses Ta sichern
        Blockiere den Prozess Ta
          und füge Ta in die
          zugeordnete
          Warteschlange
        Wähle bereiten Prozess Tb
        Kontext von Tb laden;
  }
}
```

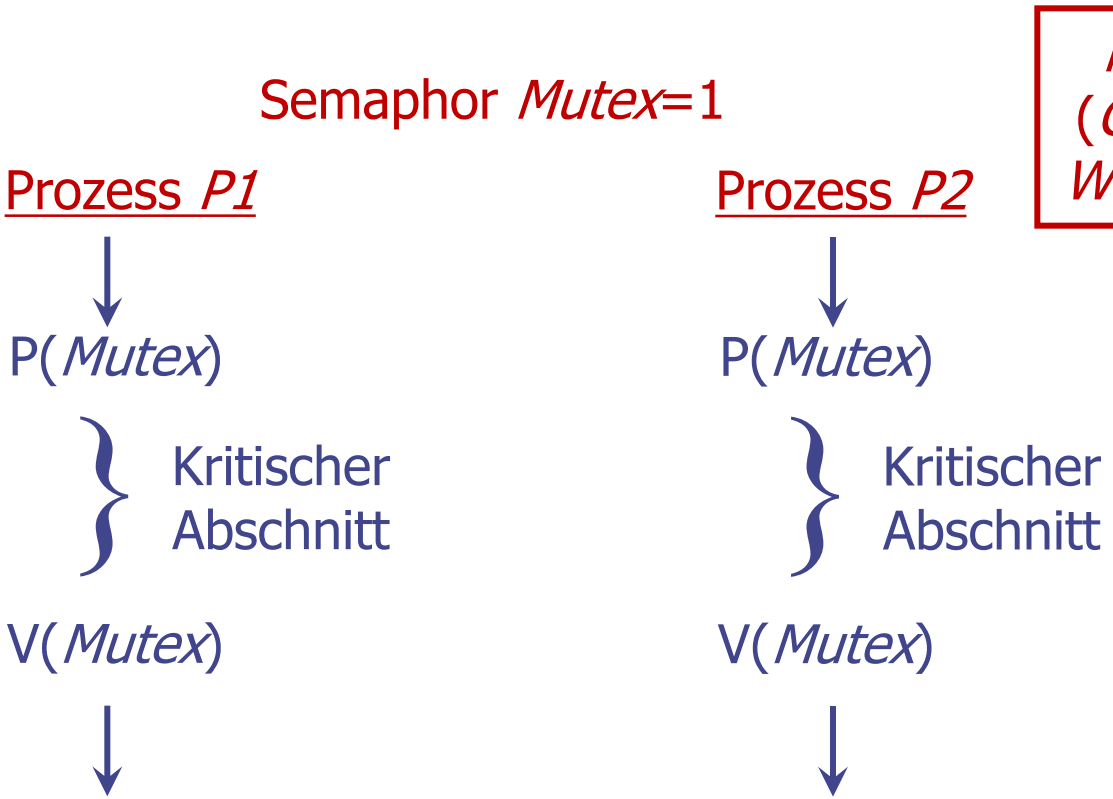
```
void V (Semaphor s) {
    if (zaehler(s)<0) { /*
    Prozess(e) im Zustand
    blockiert vorhanden */
        Bestimme z.B. den am
          längsten wartenden
          Prozess Tc
        Versetze Tc in den
          Zustand bereit
    }
    zaehler(s)++;
}
```

Funktionsweise der Semaphore

- P-Operation löst bei einem Zählerstand kleiner 0 (*nach Dekrementierung!*) eine Blockade des aktuellen Prozesses und eine zwangsläufige Umschaltung zu einem bereiten Prozess aus
  - Initialisierung des Zählers bestimmt, wie viele Prozesse sich gleichzeitig im kritischen Bereich aufhalten dürfen
    - Zähler = 1 $\Rightarrow$ binäre Variable, gegenseitiger Ausschluss
    - Zähler = k>1 $\Rightarrow$ k Prozesse dürfen im kritischen Bereich sein, z.B. k = Anzahl von FTP-Benutzern
- V-Operation ermöglicht bei jedem Aufruf den Einsatz eines blockierten Prozesses aus zugehöriger Warteschlange
  - Ankommende Prozesse werden üblicherweise in der Reihenfolge des Eintreffens in die Warteschlange eingefügt (FIFO)
  - Abweichungen von der FIFO-Reihenfolge sind zum Beispiel beim Echtzeitbetrieb notwendig, etwa Berücksichtigung von Prioritäten

Beispiel: Einfacher kritischer Abschnitt

- Zwei Prozesse *P1* und *P2* konkurrieren um den Eintritt in den kritischen Bereich



Mutex=Mutual Exclusion (*Gegenseitiger Ausschluss, Wechselseitiger Ausschluss*)

Semaphor

```
struct Semaphore {
  int count;          // process counter
  Queue *wp;  // count=1: free, count<=0: occupied
}                              // if count<0 : |count| is the

                      // number of waiting processes
void init (Semaphore *s, int i) {
    s->count = i;     // set i=1 for mutual exclusion
    s->wp = NULL;
}
void P(Semaphore *s) {
   s->count--;
   if (s->count < 0) block(s->wp);  // enqueue process
}
void V(Semaphore *s) {
   s->count++;
   if (s->count <= 0) deblock(s->wp) // deblock first of
}                                          // queue
```

Beispiel: Erzeuger – Verbraucher-System

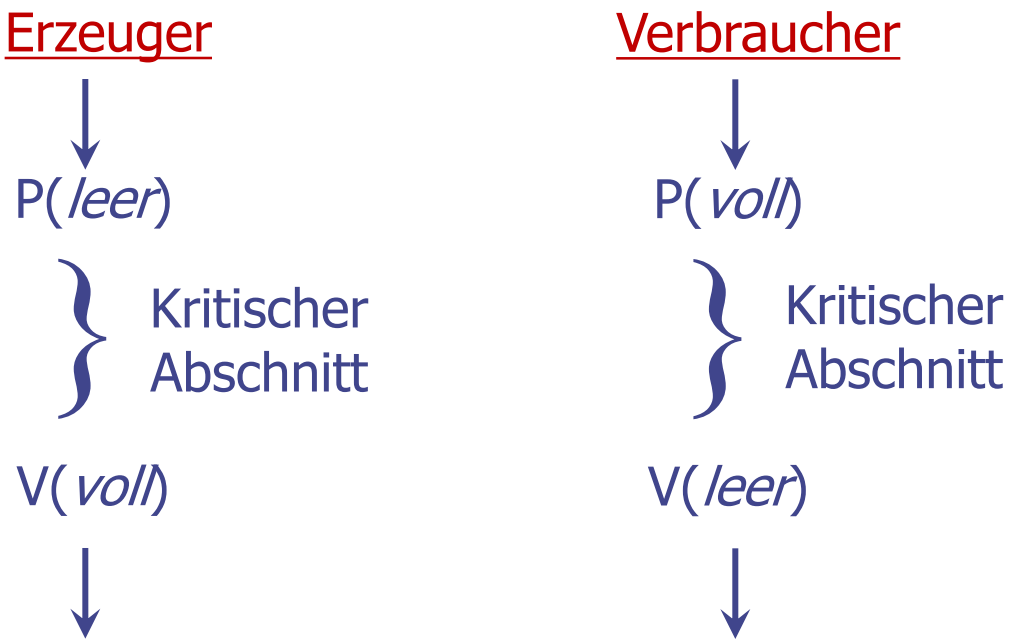
- Zwei Prozesse kommunizieren über gemeinsamen Puffer
  - Der Erzeuger füllt den Puffer
  - Der Verbraucher konsumiert den Pufferinhalt
- Nebenbedingungen
  - Der Puffer hat eine beschränkte Aufnahmekapazität, d.h. der Erzeuger darf nichts hinzufügen, wenn der Puffer voll ist
  - Der Verbraucher darf nicht auf den Puffer zugreifen, wenn dieser leer ist
- Mögliche Ereignisse
  - Der Puffer ist voll $\Rightarrow$ Semaphor *voll* wird eingeführt
  - Der Puffer ist leer $\Rightarrow$ Semaphor *leer* wird eingeführt

Erzeuger – Verbraucher mit Semaphoren

- Durch die Anordnung der Semaphore *voll* und *leer* „*Überkreuz*“ wird eine abwechselnde Nutzung des Puffers gewährleistet

es können beliebig viele Plätze im Puffer sein

Semaphor *leer*=1, *voll*=0



überkreuz gekoppelt: wenn “leer”, wird Erzeuger ausgeführt, füllt den Platz und erhöht “voll”