2. Prozesse

- Überblick
  - 2.1 Prozesse, Prozesszustände und Prozessumschaltung
  - 2.2 Threads
  - 2.3 Parallele und nebenläufige Programmiersprachenkonstrukte
  - 2.4 Prozessgraphen

2.1 Prozesse

Qualität eines OS erkennt man an der Prozessverwaltung

- 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 — Raum für Daten
  - Verarbeitungsvorschrift, üblicherweise ein Programm
  - Aktivitätsträger, der die Verarbeitungsvorschrift ausführt, in der Regel als Thread bezeichnet

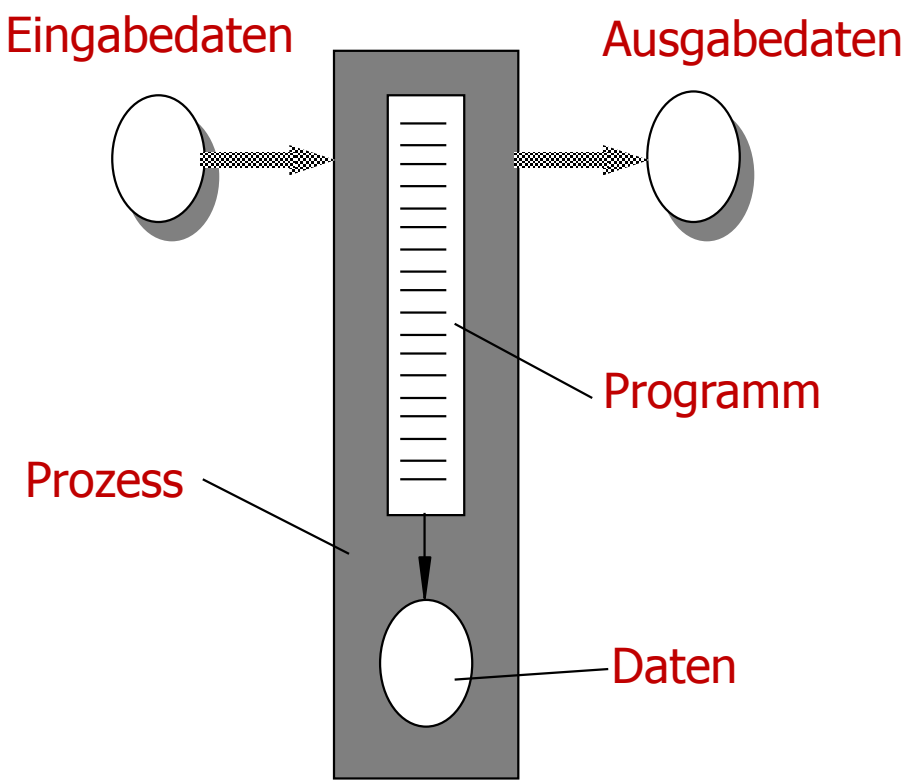
Prozesse sind durch menschenlesbare Namen und Process ID definiert (für OS ist nur die PID interessant)

- Prozess = virtueller Rechner spezialisiert zur Ausführung eines bestimmten Programms

Prozesse erben Rechte von dem Nutzer, in dessen Auftrag sie arbeiten (user, root)

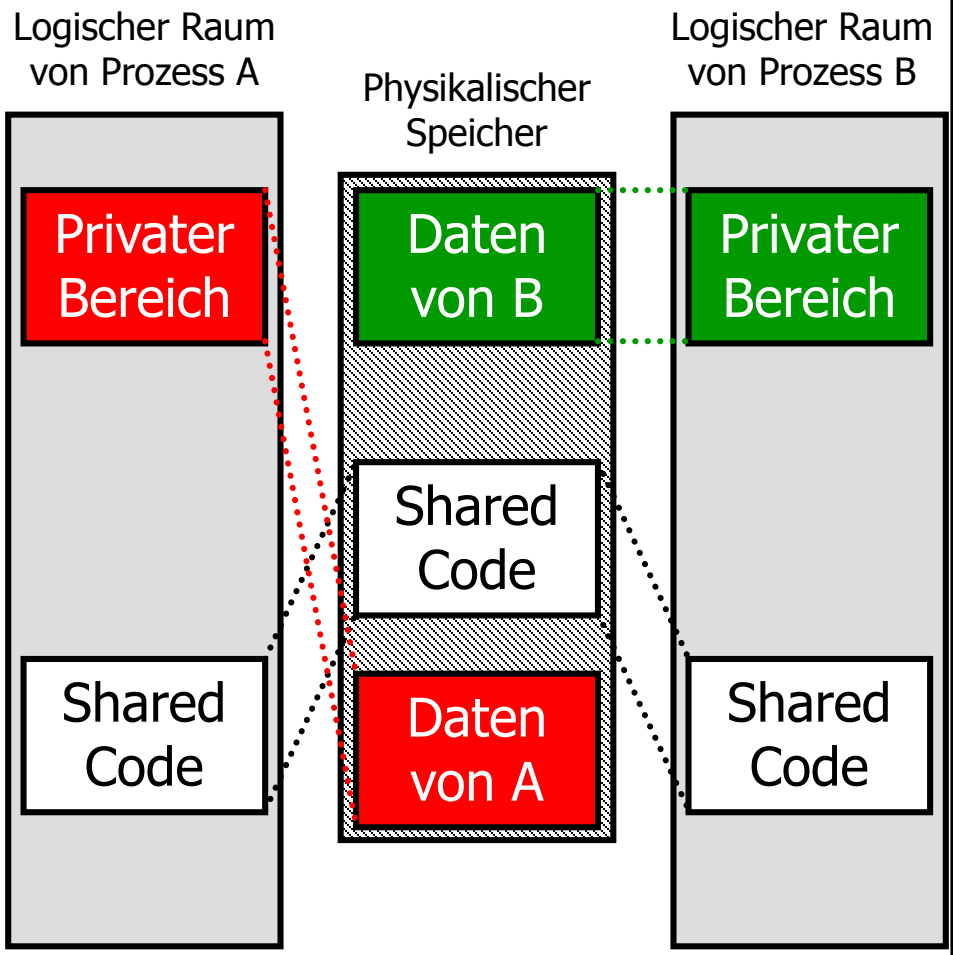
Beschreibungseinheit Prozess

- 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

- 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



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) klassische Architektur
  - Mehrere Prozesse teilen sich einen Adressraum (Threads)
  - Ein Prozess wechselt von einem Adressraum zum anderen Adressraum

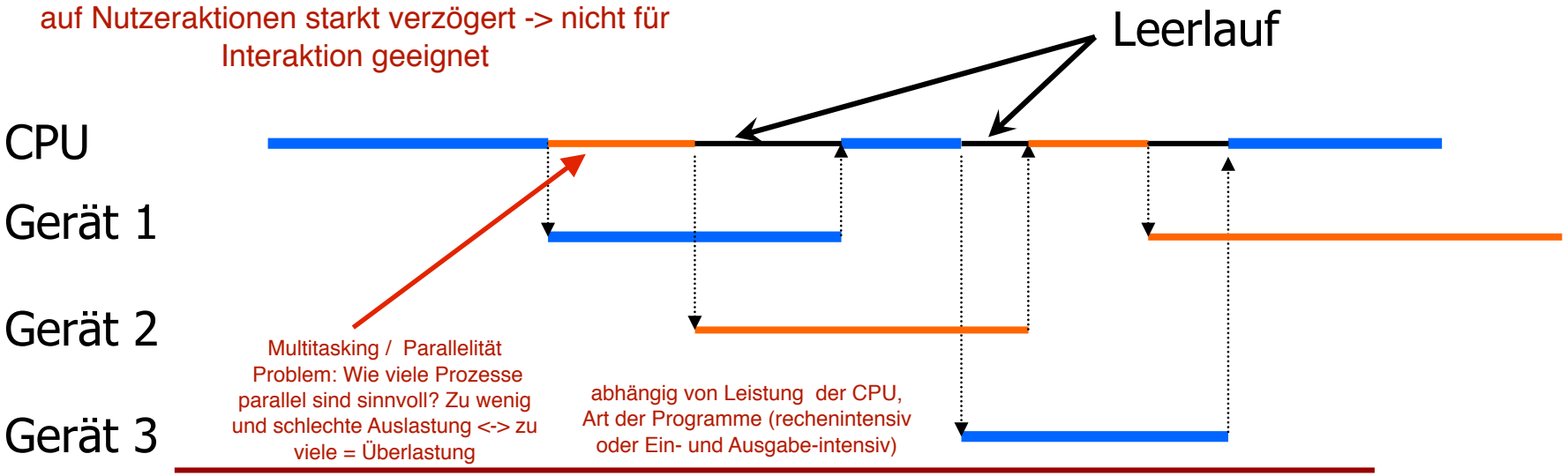
Ausführung von Prozessen

- Einfachste Rechnerbetriebsart ⇒ Stapelbetrieb (batch mode)
  - Der aktive Prozess wird unterbrechungsfrei – ohne Unterbrechung durch andere konkurrierende Prozesse – ausgeführt
  - Mehrere Prozesse werden sequentiell abgearbeitet

effizienteste Betriebsart -> Hochleistungsrechner

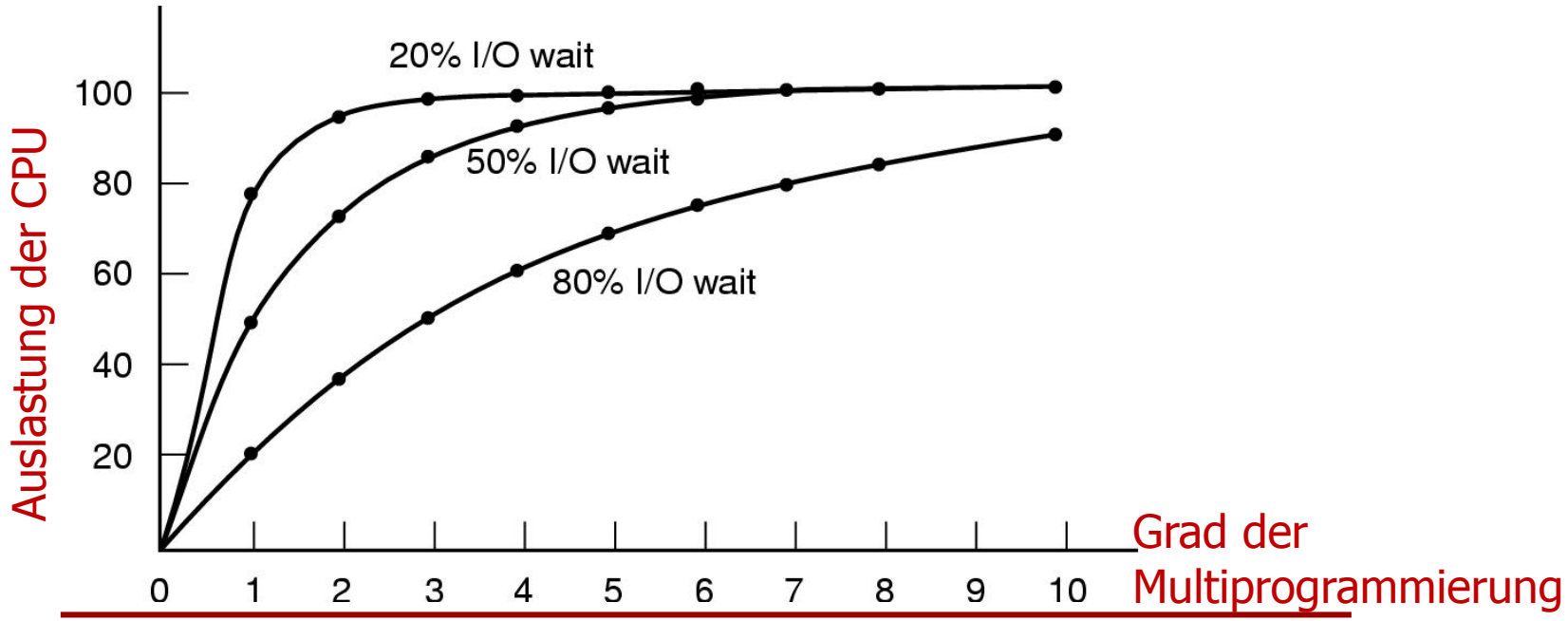
- Problem: Während der Kommunikation mit z.B. E/A-Geräten bleibt die CPU ungenutzt ⇒ Leerlaufzeiten und ineffiziente Ausführung, große Aufträge blockieren das gesamte System

Problem außerdem: nicht responsiv, d.h.Reaktion auf Nutzeraktionen starkt verzögert -> nicht für Interaktion geeignet



Modellierung der Multiprogrammierung

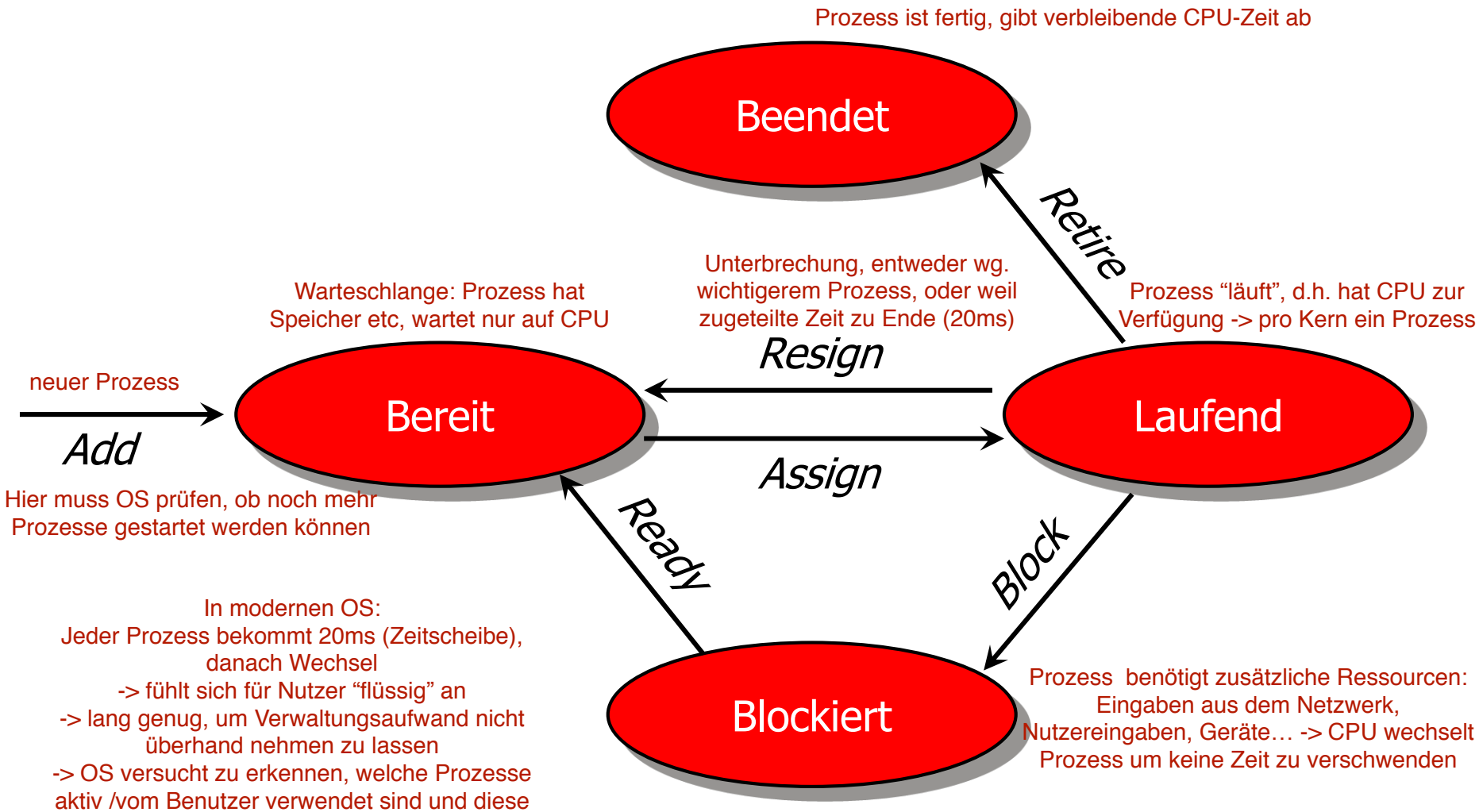
- Wie viele Prozesse sind für "genau richtige" Auslastung notwendig?
- Keine allgemeine Antwort möglich. Annahmen
  - Ein Prozess verbringt einen Anteil p seiner Zeit mit Warten auf E/A-Operationen
  - Wahrscheinlichkeit p^n = n Prozesse warten gleichzeitig auf E/A-Ende
  - Ausnutzung der CPU: $A = 1 - p^n$
  - n = Grad der Multiprogrammierung (Degree of Multiprogramming)



Wichtig!!

Prozesszustände

aktive Prozesse wechseln ständig zwischen “laufen” und “schlafen”



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

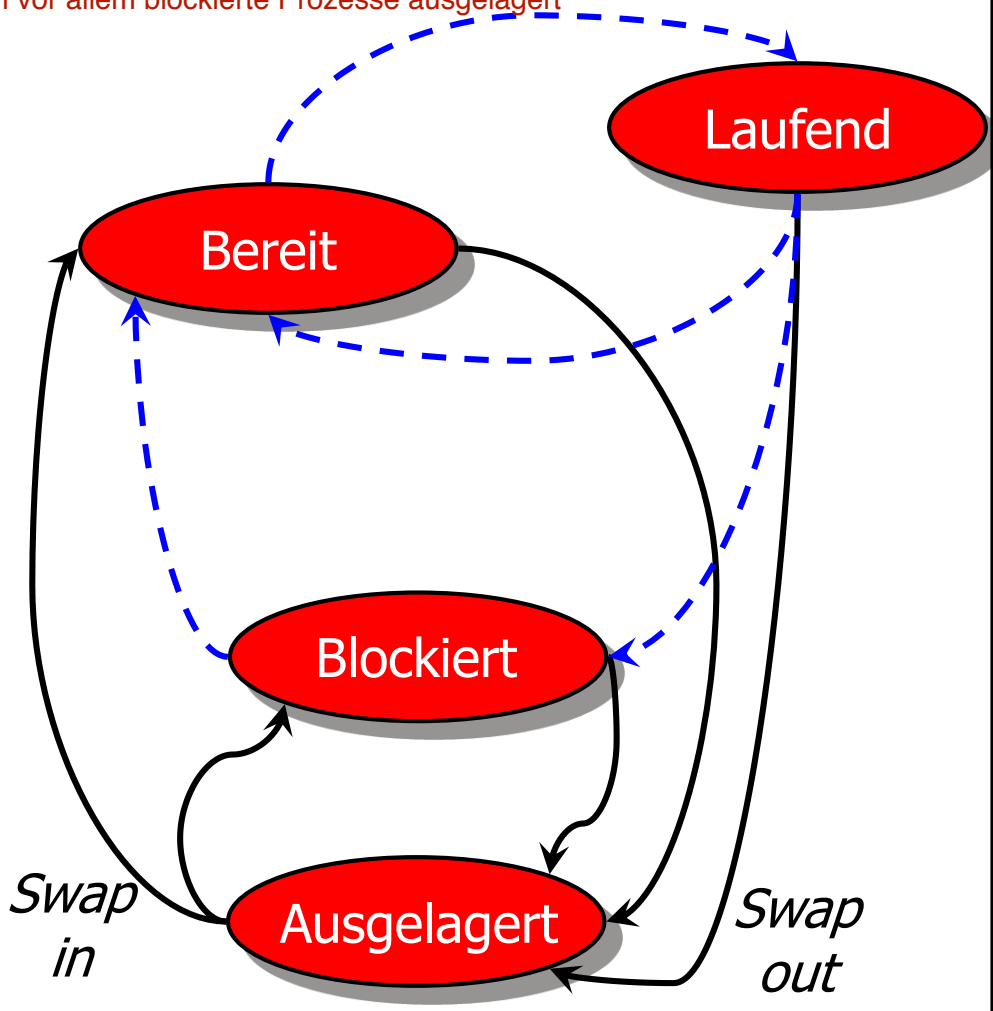
Erweitertes Zustandsmodell

Daten auf Festplatte ausgelagert, wenn RAM voll
-> es werden vor allem blockierte Prozesse ausgelagert

- Wegen Speichermangel werden oft ganze Adressräume ausgelagert (Swapping) ⇒ dem Prozess fehlt auch Arbeitsspeicher
  - Zusatzzustand *Ausgelagert*
  - Zusatzübergänge *swap in* und *swap out*

möglichst vermeiden! -> häufiger Grund für Einfrieren eines Systems (dauert lange)

- 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

neue Version: Folien 12-14 zu Concurrency
- Nebenläufigkeit: Wechsel zwischen verschiedenen Prozessen so dass Illusion von Gleichzeitigkeit entsteht
- Parallelität: Prozesse werden gleichzeitig auf verschiedenen Prozessoren ausgeführt

-> real ist nebenläufig nicht viel langsamer als parallel wegen IO-Wartezeiten

Sprünge zwischen Prozessen werden durch Interrupts gesteuert (Interrupt -> springe zu nächstem Prozess)
OS entscheidet wohin gesprungen wird (welcher Prozess hat höchste Priorität?)
Vor Sprung muss Fortsetzstelle (FSS) gemerkt werden = aktueller Zustand des Prozesses
FSS des nächsten Prozesses ist Sprungziel

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.

```
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
- Arten der Prozessumschaltung

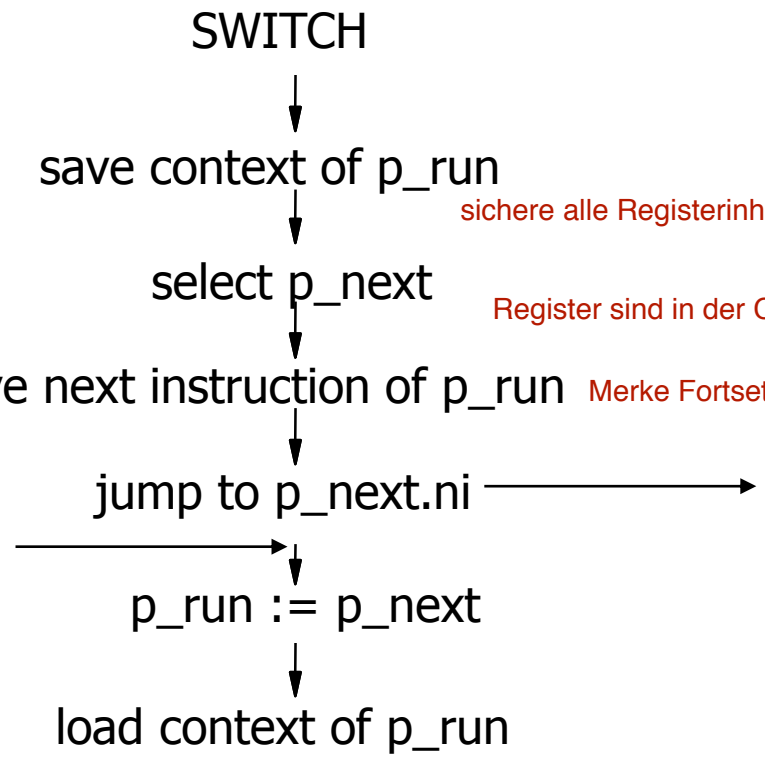
  OS selbst wechselt explizit
  - Explizit: aktiver Prozess initiiert selbst die Umschaltung
  - Unbedingt: gezielte Übergabe/Auswahl durch übergeordnete Instanz

  Umschaltung, außer wenn gerade in kritischem Bereich
  - Bedingt: Umschaltung erst bei Erfüllung einer Bedingung möglich
  - Automatisch: Prozessumschaltung wird durch ein äußeres Ereignis (Interrupt) ausgelöst
- Alle Zustandsveränderungen werden in die Prozesstabelle eingetragen

Umschalten (als offene Befehlsfolge)

- Befehlsfolge zum Umschalten

alle 20ms ausgeführt -> wenig Zeit für das Umschalten!
-> je besser das OS, desto schneller kann es umschalten (effizienter)



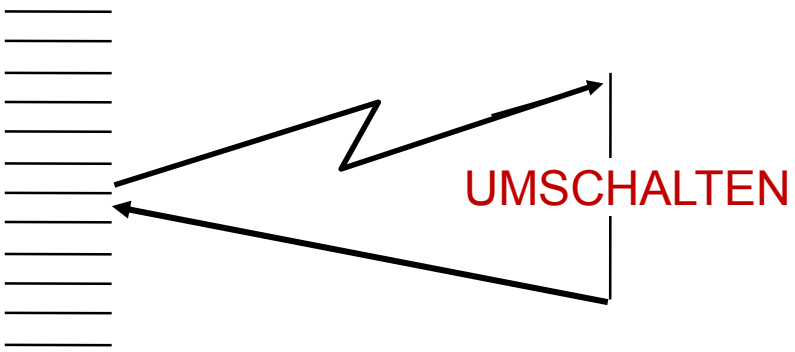
Prozessauswahl (Teil des OS) -> entscheidet über nächsten Prozess

Automatisches Umschalten

- 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)

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) Round Robin
  - Ankunftsreihenfolge First Come, First Serve
  - Fairness und Priorität (Konstant / Dynamisch)
  - Einhaltung von geforderten Fertigstellungspunkten Deadlines

idR bestes Verfahren: keine Zeit verschwenden, jeder kommt mal dran
Wartezeit erhöht Priorität

- Nach der Wahl müssen die Attribute aller anderen Prozesse angepasst werden ⇒ Detailliert in Kapitel „Scheduling“

Prozesse sollen nicht “verhungern”
-> nie drankommen

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

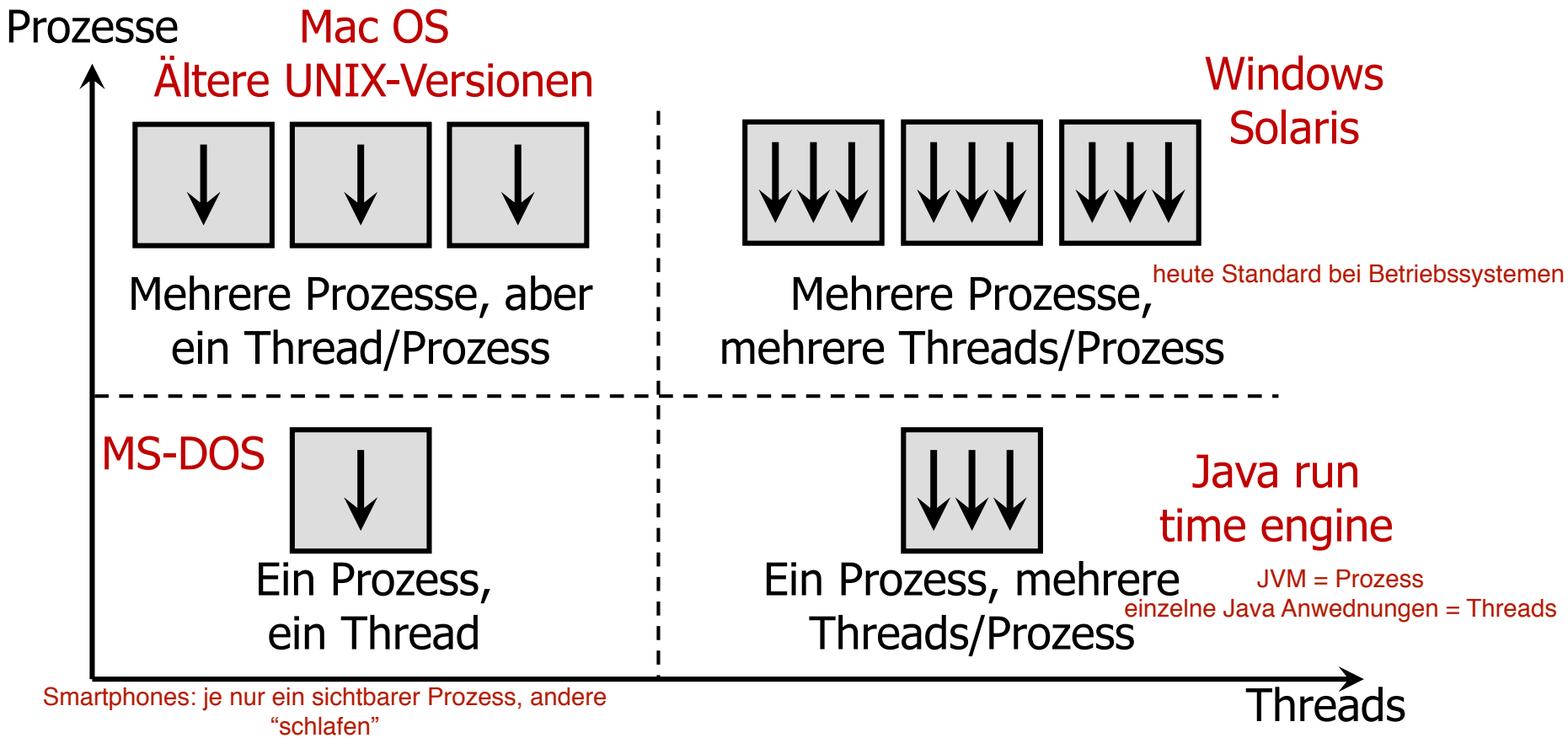
"Mini-Prozess"

- 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

Wurzel = Prozess
Kinder = Threads

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

Zeitdauer nur in der Bearbeitungsphase unterschiedlich

Anfrage:
Annahme - Bearbeitung - Ausgabe

Beispiel zur Nutzung von Threads (2)

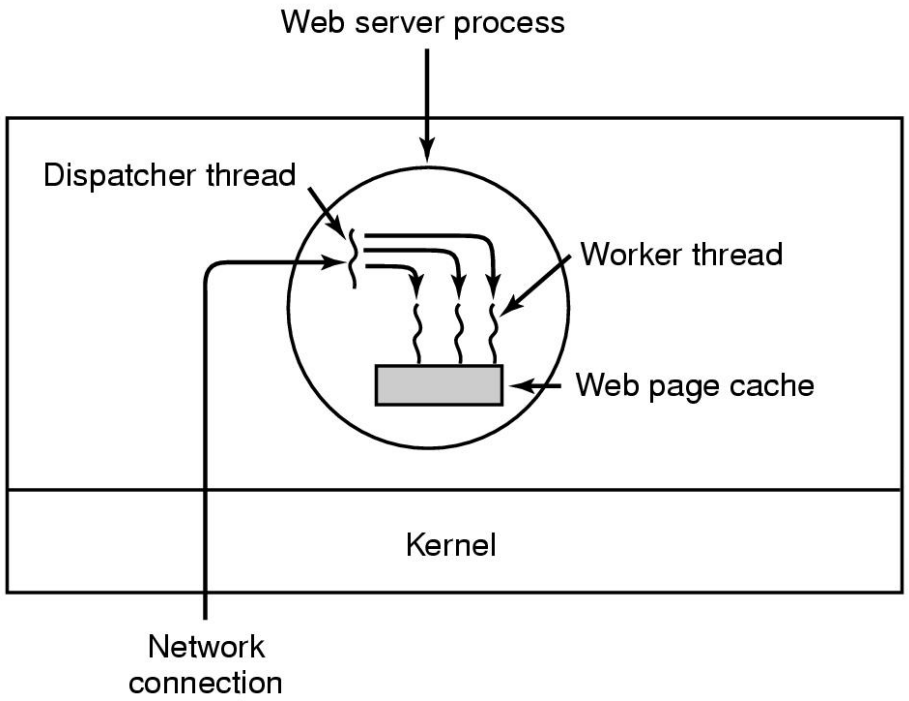
- Realisierung mit mehreren Threads
  - Thread Dispatcher: liest ankommende Anfragen
  - Thread Worker: bearbeitet eine einzelne Anfrage
- Ablauf
  - Dispatcher empfängt die Anfrage und kreiert/weckt einen Worker
  - Worker wechselt sobald möglich in laufend, überprüft Anfrage
    - Daten im Cache ⇒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

      ⇒startet Leseoperation und versetzt sich in Zustand blockiert. Leseoperation beendet

      ⇒Wechsel in Zustand *Bereit*

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

Webserver mit mehreren Threads



Dispatcher: Ein Thread nimmt ausschließlich Anfragen an und antwortet (ACK)
Worker: nimmt Anfragen entgegen, es gibt pro Anfrage einen Thread

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); }
```

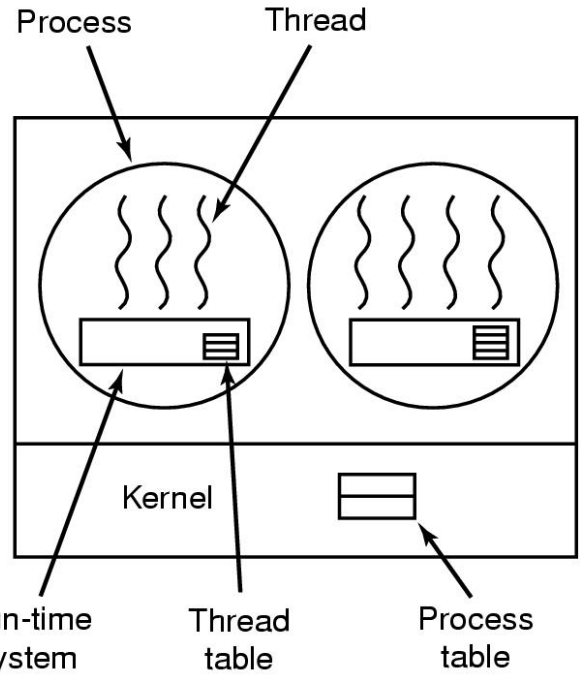
Code Dispatcher

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

Threadtypen

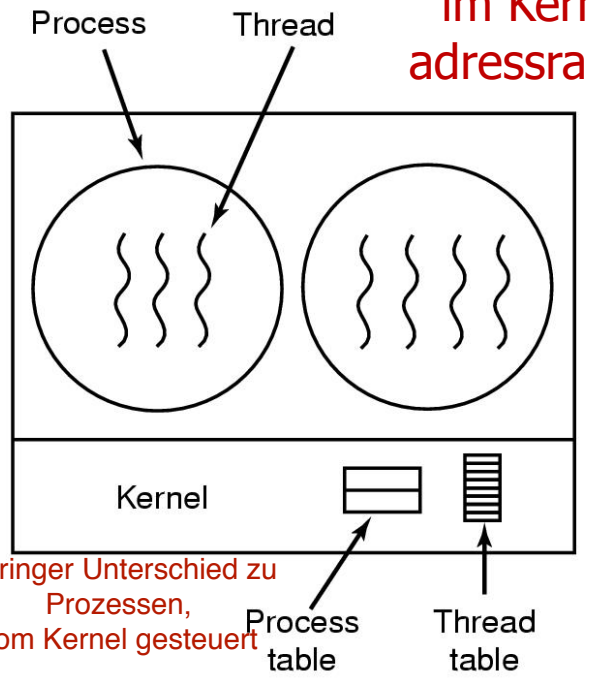
- Grundsätzlich werden Threads aufgeteilt in
  - Kernel-Level-Threads (KL-Threads): realisiert im Kernadressraum
  - User-Level-Threads (UL-Threads): realisiert im Benutzeradressraum
- Hybride Realisierung ist allerdings auch möglich

Realisierung im Benutzer-adressraum



Freiheit für den Benutzer:
Wie will er Threads nutzen?
-> evtl. Konflikte

Realisierung im Kern-adressraum



Geringer Unterschied zu Prozessen, vom Kernel gesteuert

Eigenschaften von KL-Threads

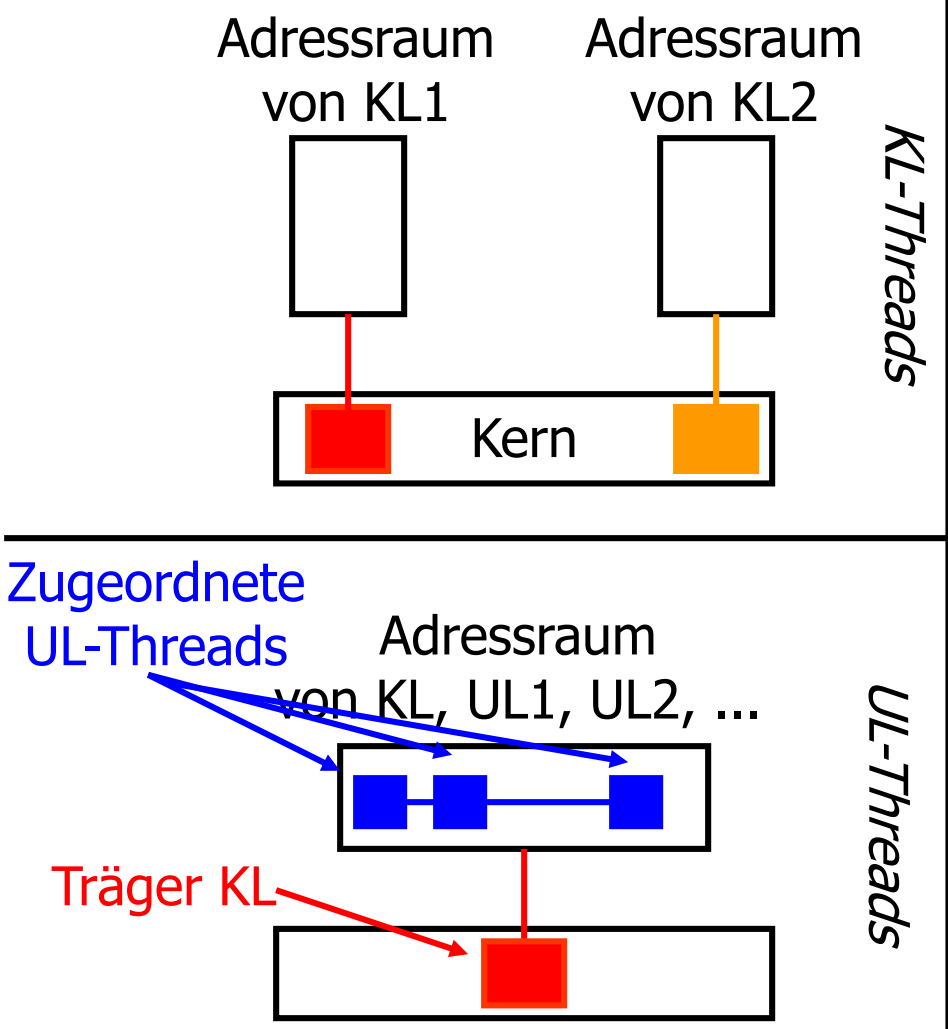
- KL-Threads haben folgende Eigenschaften
  - Werden im Kern des Betriebssystems (der Systemsoftware) realisiert
  - Umschaltung erfordert Aufruf von Verwaltungsfunktionen des Betriebssystems $\Rightarrow$ Kontrolle wird für den Umschaltzeitraum an das Betriebssystem übergeben
  - Im Allgemeinen erfordert eine Umschaltung von KL-Threads auch den Wechsel des Adressraums
- Beispiele für Betriebssysteme mit Unterstützung für KL-Threads
  - Windows, Solaris 2, BeOS, Tru64 (früher DigitalUNIX)

Umschaltung von KL-Threads

- Adressraumwechsel ist mit bedeutendem Zusatzaufwand verbunden
  - Instruktionen an MMU zum Neuladen der gesicherten Segmenttabellen und der Seitentabellenregister
  - Indirekte Verzögerung aufgrund des kalten Caches, d.h. eine bestimmte Vorlaufzeit ist notwendig, bis die abgestrebte Cachetrefferrate beim Datenzugriff erzielt wird
- Weitere Nachteile
  - Umschaltung wird durch eine Unterbrechung initiiert ⇒ Befehle wie *trap* gehören zu den aufwendigsten des Befehlssatzes
  - Übergabe der Umschaltungskontrolle an den Kern und Aktivierung der Verwaltungsfunktionen des Betriebssystems ⇒ zusätzlicher Verwaltungsaufwand
- KL-Threads = schwergewichtige Threads

UL-Threads

- Vollständige Realisierung im Adressraum der Anwendung
- KL-Thread als Träger
  - ⇒UL-Threads sind der Prozessorverwaltung und dem Betriebssystemkern völlig unbekannt
- Umschaltung zwischen den UL-Threads ähnelt einem Prozeduraufruf
  - ➢ Kein Adressraumwechsel
  - ⇒Leichtgewichtsprozesse



Wechsel relativ schnell & einfach im Vgl. zu Kernel-Threads
Nutzer trägt Verantwortung für Steuerung der Threads
-> OS arbeitet nach Standardmustern, nicht unbedingt angepasst an konkretes Problem, Anwednungsprogramm weiß besser, was sinnvoll ist

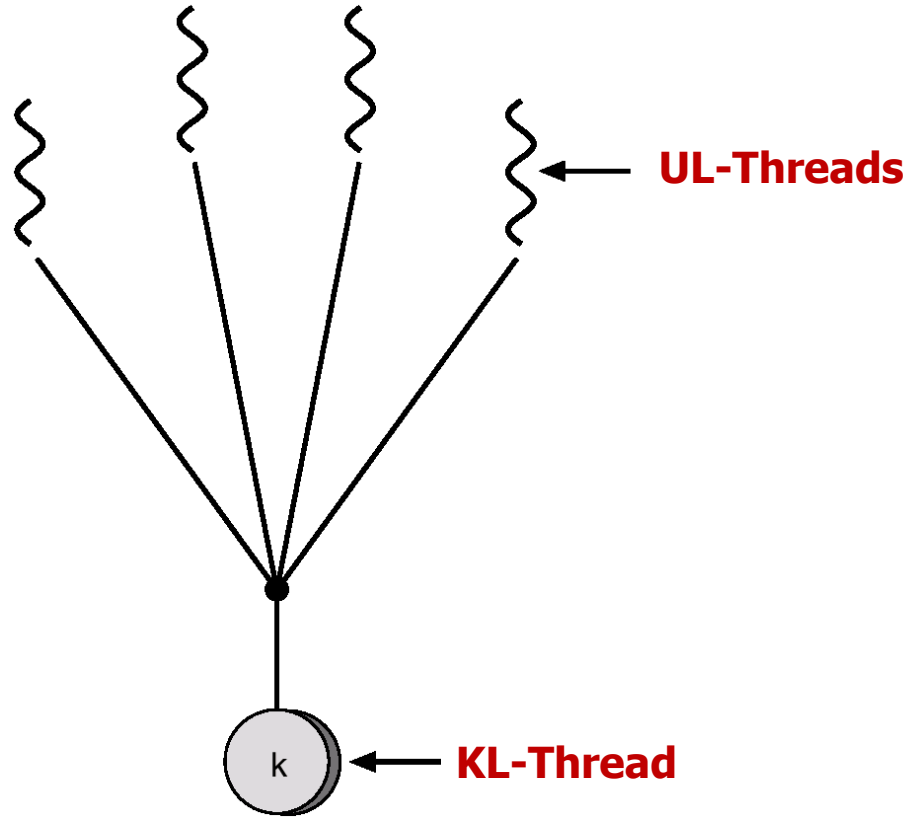
Umschaltung von UL-Threads

- UL-Threads benötigen eine Laufzeitumgebung mit Funktionen zur Verwaltung der Threads (Blockierung, Umschaltung, Scheduling, Erzeugung/Löschung, ...)
  - Threadtabelle: Analog zur Prozesstabelle mit Informationen wie Befehlszähler, Stapelzeiger, Register, Zustand ...
  - Einsetzbar auch bei Betriebssystemen ohne Thread-Unterstützung
- Ein UL-Thread benötigt bestimmte Betriebsmittel (BM)
  - Aufruf der entsprechenden Funktion im Laufzeitpaket
  - Überprüfung, ob der Thread blockiert werden muss
    - Nein $\Rightarrow$ BM werden zur Verfügung gestellt
    - Ja $\Rightarrow$ Speicherung der Threaddaten in der Threadtabelle
      - Falls möglich: Auswahl eines bereiten Threads aus der gleichen Tabelle, Laden der Threaddaten und Ausführung
      - Andernfalls: Blockierender Systemaufruf, bis die benötigten Betriebsmittel zur Verfügung gestellt werden

Multithreading-Modelle

- Modelle für hybride Unterstützung von KL- und UL-Threads
  - Zuordnung vieler UL-Threads zu einem KL-Thread (many-to-one)
  - Zuordnung eines UL-Threads zu einem KL-Thread (one-to-one)
  - Zuordnung mehrerer UL- zu mehreren KL-Threads (many-to-many)

Modell: Many-to-One



Ein Prozess mit Ressourcen
mehrere Threads als “Untermieter”
z.B. JVM
-> heute selten verwendet

- Ein KL-Thread als Träger für UL-Threads
- Effiziente Verwaltung im Benutzerraum
- Blockierende Systemaufrufe blockieren alle Threads
- Nicht geeignet für Multiprozessoren (Nur der KL-Thread wird zugeordnet)

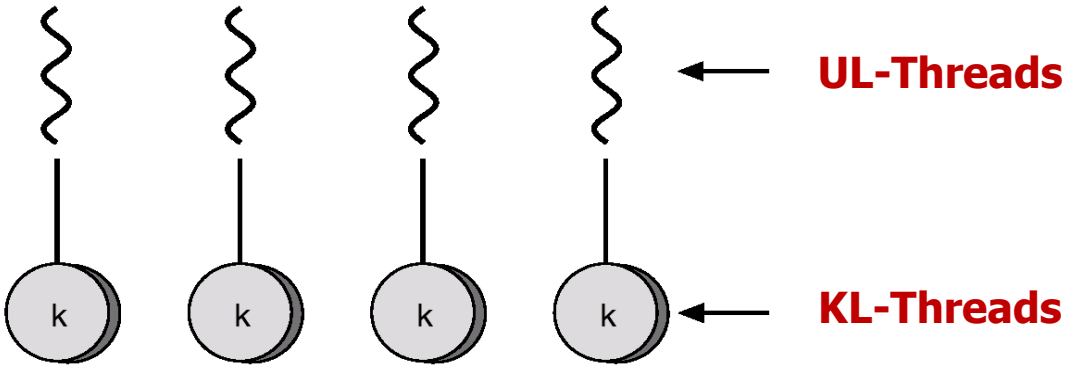
  OS sieht die Threads nicht!
- Beispiele:
  - Green Threads (Solaris 2)
  - Bibliotheken für BS ohne Threadunterstützung

Nachteil: blockierende Aufrufe blockieren alle

Multithreading-Modelle: One-to-One

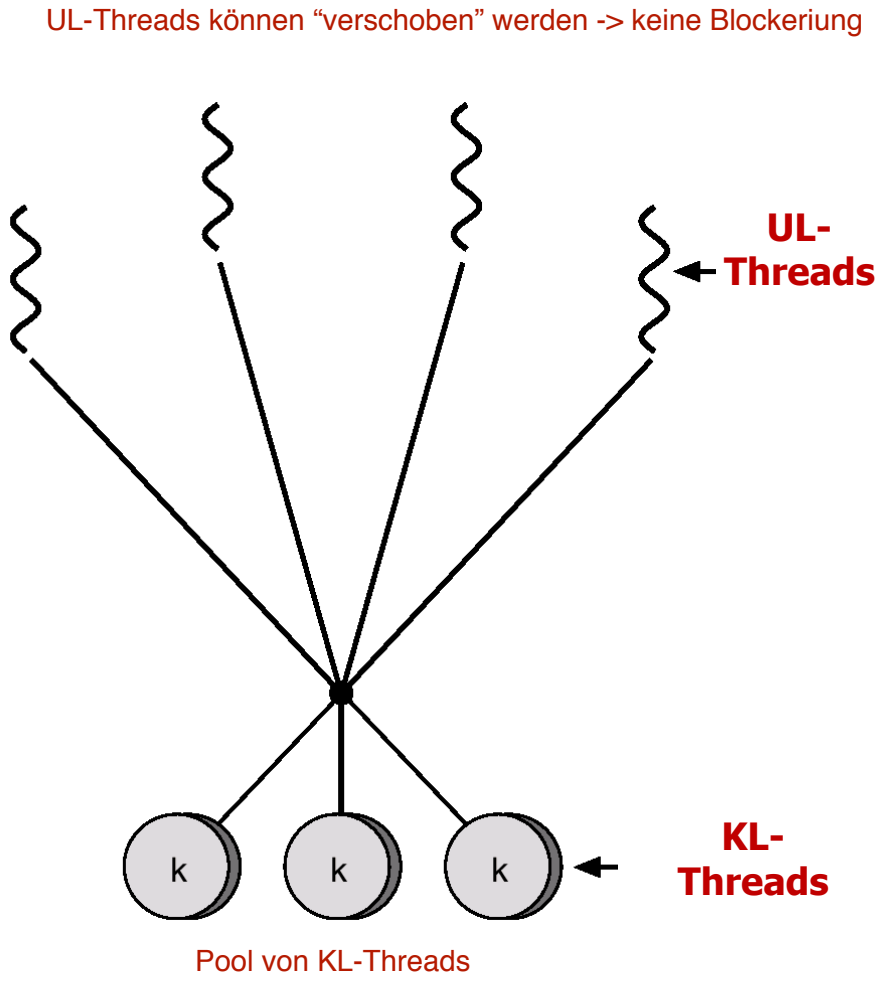
- Direkte Abbildung der UL-Threads auf KL-Threads
- Keine Blockierung aller UL-Threads durch blockierende Systemaufrufe eines UL-Threads
- Mehrere Threads werden in Multiprozessoren parallel ausgeführt
- Wichtiger Nachteil: Erzeugung/Löschung eines Threads genauso aufwendig wie entsprechende Prozessoperationen $\Rightarrow$ Die meisten BS mit diesem Modell beschränken die Anzahl möglicher Threads
- Beispiele: Windows

OS kennt Zuordnung von Prozessen und Threads



Multithreading-Modelle: Many-to-Many

- Mehrere UL-Threads auf gleich viele/weniger KL-Threads
  - Anzahl der KL-Threads wird durch die Anwendung oder in Abhängigkeit von Zielhardware (1-CPU vs. n-CPU) bestimmt
  - Kompromiss zwischen Modellen
    - Keine Blockierung durch Systemaufrufe
    - Parallele UL-Threads partiell möglich
    - Moderate Erhöhung des Verwaltungsaufwands
    - Keine Beschränkung der Threadanzahl
  - Beispiele: Solaris 2, HP-UX, Tru64 UNIX, IRIX



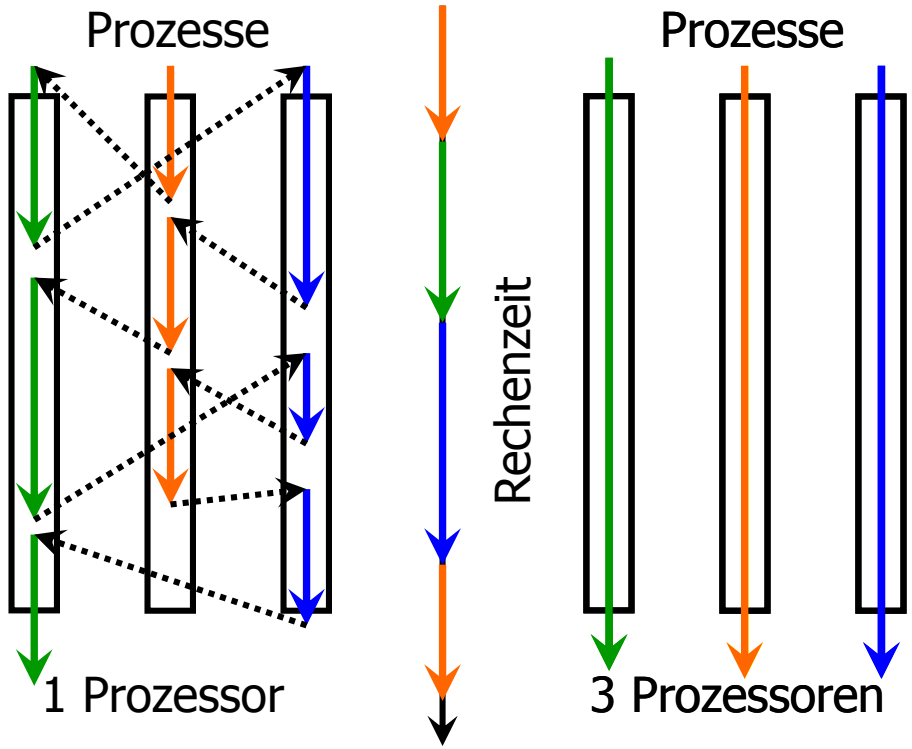
2.3 Nebenläufigkeit und Parallelität

- 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
- Parallelität = Zuordnung mehrerer Prozesse zu mindestens zwei Prozessoren
- 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
  - Explizit = Programmierer sieht bewusst und gezielt nebenläufige Kontrollflüsse vor

Paralleles Verzweigen

- UNIX-Konzept `fork/join` (auch `fork/wait`) ermöglicht Erzeugung einer perfekten Kopie (Child) des aufrufenden Prozesses (Parent) mit folgenden Eigenschaften
  - Gleiches Programm
  - Gleiche Daten (gleiche Werte in Variablen)
  - Gleicher Programmzähler (nach der Kopie)
  - Gleicher Eigentümer
  - Gleiches aktuelles Verzeichnis
  - Gleiche Dateien geöffnet (selbst Schreib-, Lesezeiger ist gemeinsam)
- Zusammenführung der beiden Zweige mit `join` (oder `wait`)

Beispiel `fork/join`

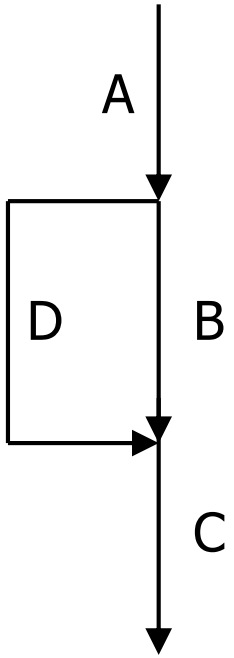
- Unterscheidungsmerkmal
  - `fork()` liefert verschiedene Prozessnummern für Parent (!=0) / Child (0)

```
pid_t p;                    Parent
...
p= fork();
if( p == (pid_t)0 ) {
/* child */ ...
}
else
{
/* parent */ ...
} else {
/* error */...
}
```

```
pid_t p;                    Child
...
p= fork();
if( p == (pid_t)0 ) {
/* child */ ...
}
else
{
/* parent */ ...
} else {
/* error */...
}
```

Beispielhafter Ablauf von `fork/join`

```
    A
    fork f
    B
    join f
    C
    end
f:  D
    end
```

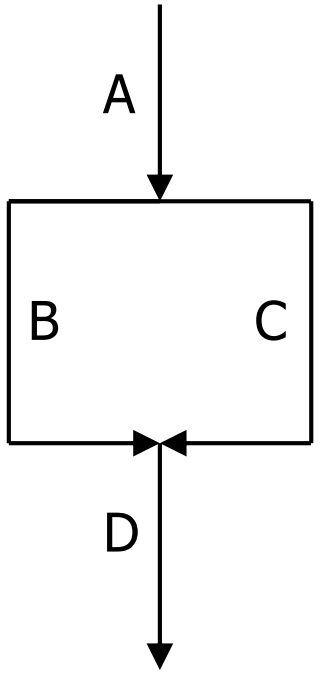


Parallele Anweisungen

- Unabhängige Befehle werden in Blöcke zusammengefasst
- Ausführungsreihenfolge wird explizit als irrelevant gekennzeichnet
- Analogon zu `begin/end` oder `{ }`:
  - `parbegin/parend`,
  - `cobegin/coend`,
  - …

Beispiel

```
A;
parbegin
   B;
   C;
parend
D;
```

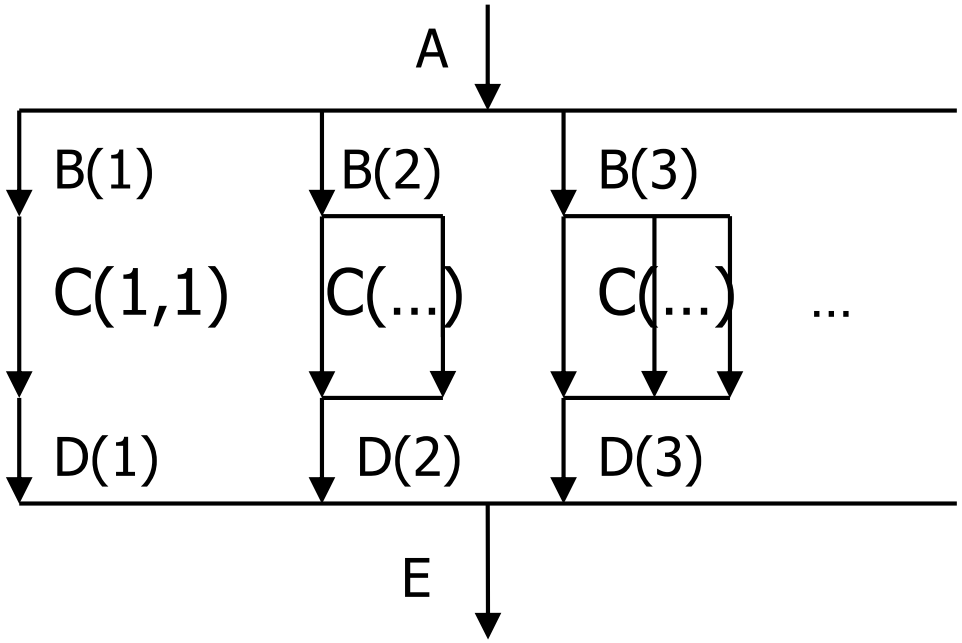


Parallele Schleifen

- Sind die Schleifendurchläufe unabhängig voneinander, so können n Schleifendurchgänge, n=Anzahl verfügbarer Prozessoren, parallel durchgeführt werden
- Häufige Schlüsselwörter: `pardo/parend`, `doall/endo`, ...

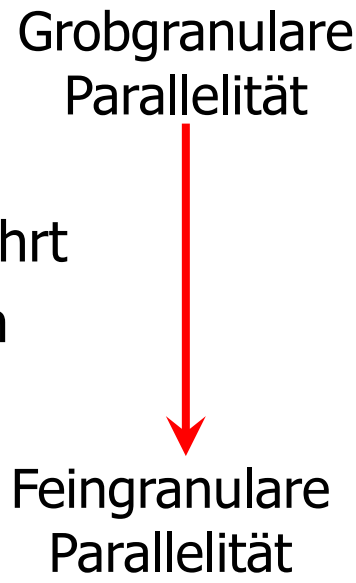
Beispiel

```
A;
pardo i:=1 to n
  B(i);
  pardo j:=1 to i
       C(i,j);
  parend
  D(i);
parend
E;
```



Granularität von Parallelität und Nebenläufigkeit

- Parallelität und Nebenläufigkeit können auf verschiedenen Abstraktionsebenen realisiert werden
  - Prozesse laufen parallel ab
  - Parallele Threads innerhalb eines Prozesses
  - Einzelne Anweisungen werden parallel ausgeführt
  - Einzelne Operationen werden in eine Reihe von Teilbefehlen zerlegt und parallel ausgeführt



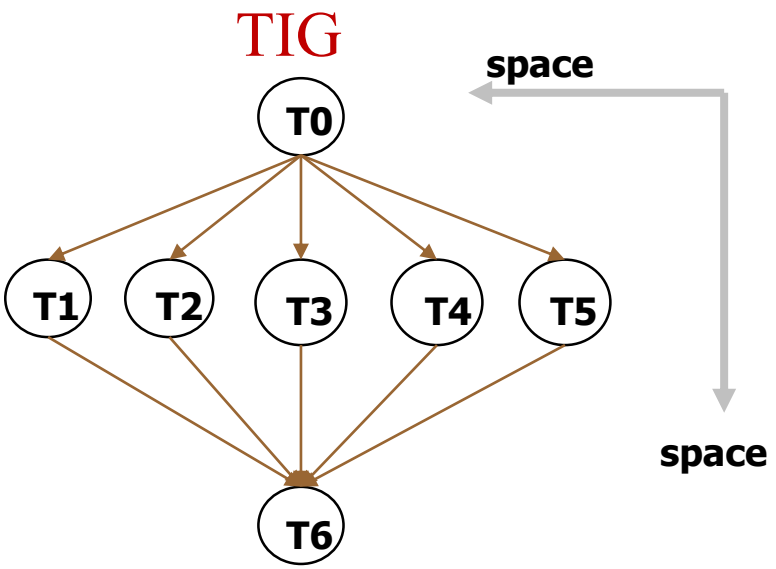
- Die Länge der parallelen Aktivitäten wird oft mit dem Begriff Granularität charakterisiert

2.4 Beziehungen zwischen Prozessen

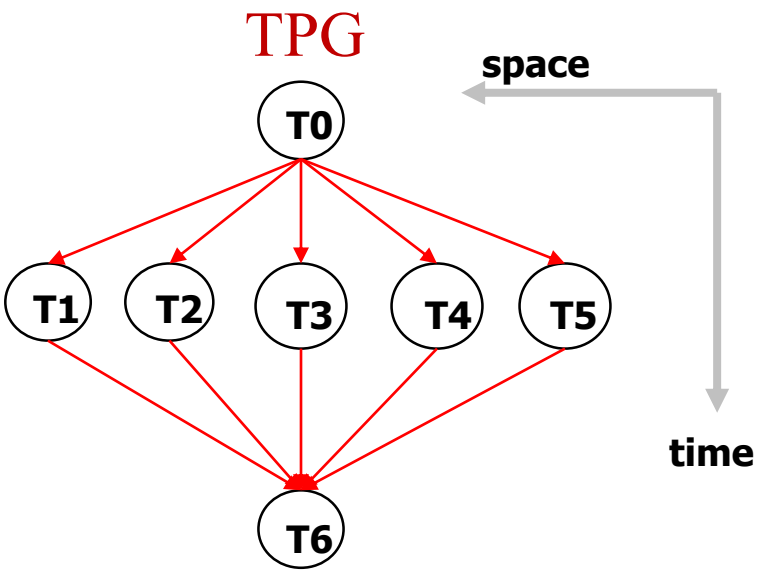
- Prozesse können in diversen Beziehungen stehen:
  - Eltern-Kind-Beziehung: Ein Prozess erzeugt einen weiteren Prozess
  - Vorgänger-Nachfolger-Beziehung: Ein Prozess darf erst starten, wenn ein anderer Prozess beendet ist
  - Kommunikationsbeziehung: Zwei (oder mehr) Prozesse kommunizieren miteinander
  - 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

Grundoperationen mit Prozessen

- Statische Betriebssysteme
  - Alle Prozesse sind a-priori bekannt und definiert
  - Prozesse werden für eine bestimmte Anwendung realisiert
  - Beschreibende Datenstrukturen (PCB) werden von einem Konfigurationsprogramm einmalig erzeugt
- Dynamische Betriebssysteme
  - Neue Prozesse können während der Laufzeit hinzukommen bzw. terminieren
  - Dies wird mit folgenden Kernoperationen realisiert
    - `create_process(id, initialValues)`

      ```
      // Anlegen des Prozesskontrollblocks
      // Initialisierung des Prozesses
      ```

    - `delete_process(id, finalValues)`

      ```
      // Rückgabe der Endwerte
      // Löschen des Kontrollblocks
      ```

Ereignisse zur Erzeugung von Prozessen

- Vier Ereignisse zur Erzeugung von Prozessen
  1. Initialisierung des Systems: Meistens Hintergrundprozesse (Daemons) wie Terminaldienst, Mailserver, Webserver, …
  2. Prozesserzeugung durch andere Prozesse: Aufteilung des Prozesses in mehrere nebenläufig oder parallel auszuführende Aktivitäten, die als eigene Prozesse initialisiert werden
  3. Benutzerbefehle zum Starten eines Prozesses (Kommandozeile oder grafische Oberfläche)
  4. Initialisierung einer Stapelverarbeitung (bei Mainframes)
- Technischer Ablauf in allen Fällen gleich
  - Bestimmter Prozess analysiert die Eingabe (z.B. von Benutzern oder Konfigurationsdateien)
  - Prozess sendet einen Systemaufruf zur Prozesserzeugung und teilt dem BS mit, welches Programm darin ausgeführt werden soll

Prozesserzeugung bei UNIX und Windows

- Systemaufruf bei UNIX ist `fork`
  - Exakte Kopie des aufrufenden Prozesses wird erzeugt mit gleichen Umgebungsvariablen, Speicherabbild, geöffneten Dateien
  - Üblicherweise ruft der Kindsprozess einen Befehl wie `execve` auf, um das Speicherabbild zu wechseln und ein neues Programm auszuführen
- Systemaufruf bei Windows ist `CreateProcess` mit 10 Parametern wie auszuführendes Programm, Parameter, Sicherheitseinstellungen, Priorität, Spezifikation über die zu erzeugenden Fenster, Zeiger auf Datenstruktur für die Prozessdaten
- In beiden Systemen haben Vater- und Kindprozess getrennte Adressräume
  - Speicheränderungen z.B. vom Kindprozess sind für Vater nicht sichtbar
  - Kommunikation über gemeinsame Dateien allerdings möglich

Erzeugen neuer Prozesse: fork (1)

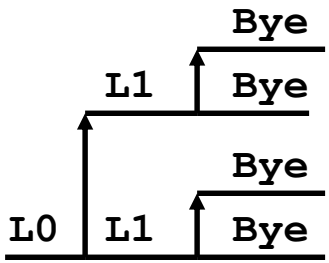
- Hauptpunkte
  - Vater und Kind führen selben Code aus
    - Unterscheidung mittels return-Wert
  - Starten vom selben Zustand, aber jeder hat eine private Kopie
    - Inklusive der gemeinsamen Ausgabedateideskriptoren

```
void fork1() {
    int x = 1;
    pid_t pid = fork();
    if (pid == 0) {
       printf("Child has x = %d\n", ++x);
    } else {
       printf("Parent has x = %d\n", --x);
    }
    printf("Bye from process %d with x = %d\n", getpid(), x);
}
```

Beispiel: fork (2)

- Sowohl Vater als auch Kind können weiter forken

```
void fork2()
{
    printf("L0\n");
    fork();
    printf("L1\n");
    fork();
    printf("Bye\n");
}
```



Zombies

- Beenden von Prozessen: exit
  - `void exit(int status)`
    - Beendet einen Prozess., Rückgabewert im Normalfall 0
- Abwicklung
  - Wenn ein Prozess terminiert, dann bindet er noch Systemressourcen
    - Verschiedene Tabellen innerhalb des BS
    - Ein solcher Prozess wird Zombie genannt

Neue Programme starten: exec

- int execl(char *path, char *arg0, char *arg1, ..., 0)
- Lädt und startet ausführbares Programm
  - path ist der Pfad, wo sich die ausführbare Datei befindet
  - arg0 ist der Name des Prozesses (Programmname)
  - arg1, ..., argn sind die eigentlichen Argumente
  - Liste der Argumente ist nullterminiert (char *) 0
  - Rückgabewert -1 im Fehlerfall

```
main() {
    if (fork() == 0) {
        execl("/usr/bin/cp", "cp", "foo", "bar", (char *)0);
    }
    wait(NULL);
    printf("copy completed\n");
    exit();
}
```

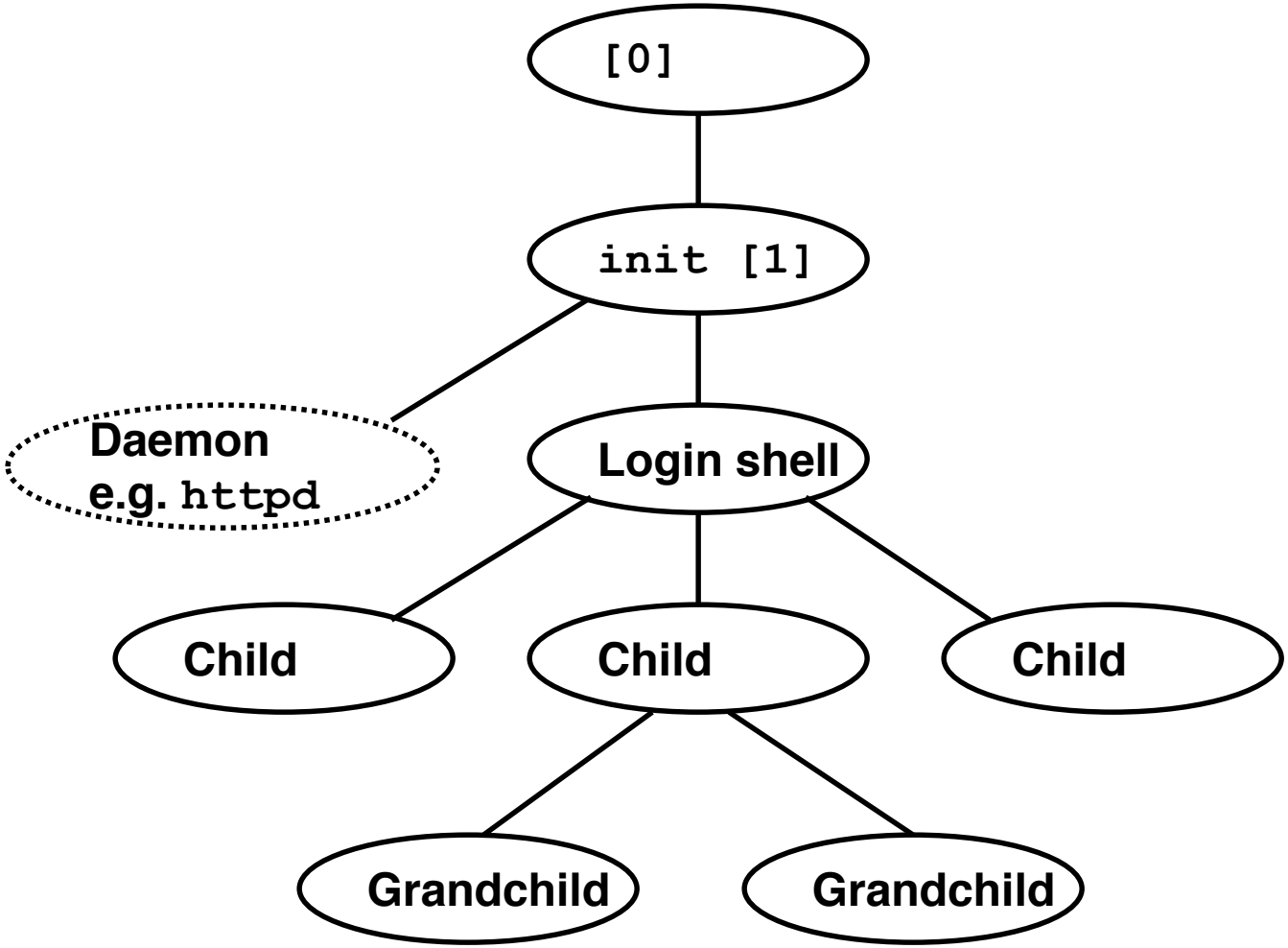
Prozesshierarchien

- Erzeugen die Kindprozesse weitere Prozesse (Kindkind… prozesse), so entsteht eine Prozesshierarchie
  - UNIX: Vaterprozess und Kindprozesse bilden eine Familie, d.h. Signale werden an alle Prozesse in der Familie verteilt und jeder Prozess entscheidet über Annahme und Verwertung
  - Bei UNIX-Initialisierung wird `init` gestartet und erzeugt die Terminals (Anzahl definiert in Konfigurationsdatei)
    - Nach Anmeldung erzeugen die Terminals `shell`-Prozesse
    - Shells erzeugen neue Prozesse bei Eingabe von Befehlen, …

    ⇒ Alle Prozesse gehören zu einem Baum mit `init` als Wurzel
  - Windows: kein Konzept einer Prozesshierarchie
    - Vaterprozess kann Kindprozess über ein Handle steuern
    - Allerdings darf der Handle an andere Prozesse weitergegeben

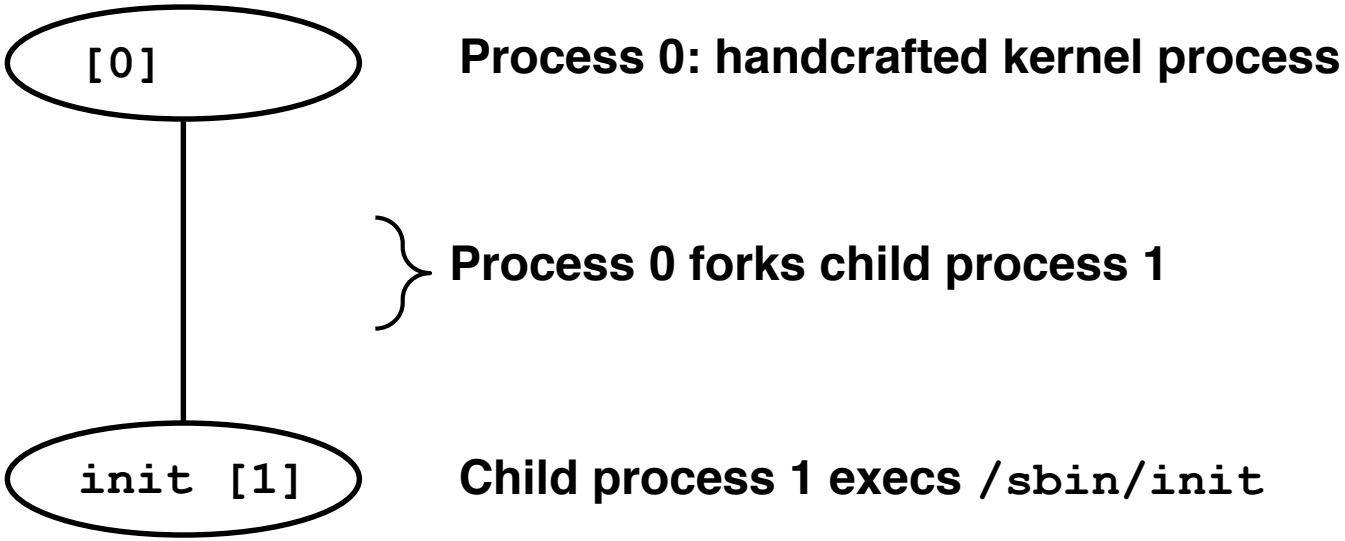
    ⇒ Prozesshierarchie wird außer Kraft gesetzt

Unix Prozesshierarchie

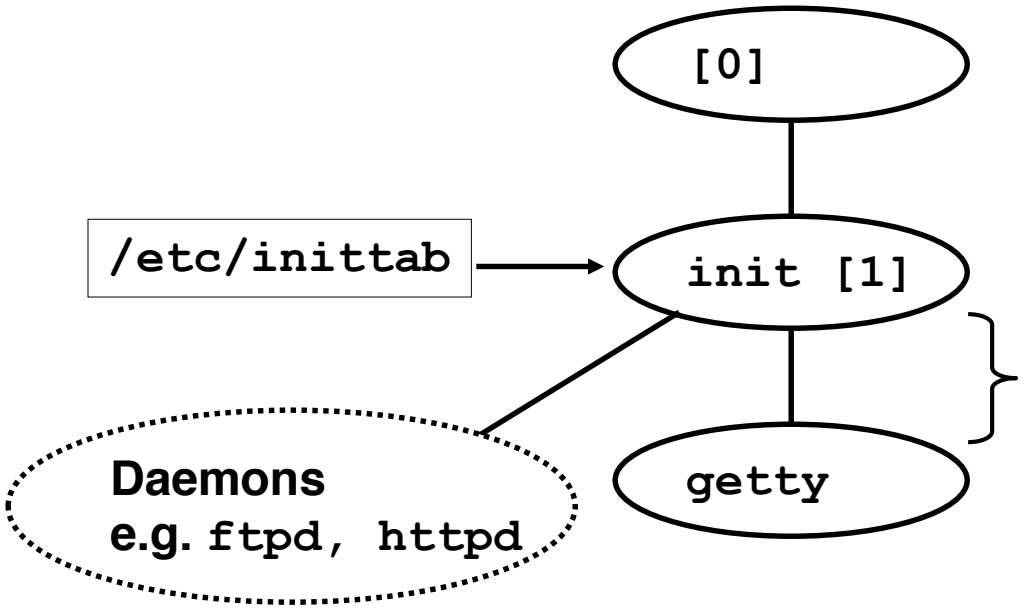


Unix Start: Schritt 1

- Drücken des Resetknopfs lädt den PC mit der Adresse eines kleinen Bootstrapprogramms
  - Bootstrapprogramm lädt den Bootblock (Plattenblock 0)
  - Bootblockprogramm lädt das Kernelbinärprogramm (z.B.: /boot/vmlinux)
  - Bootblockprogramm übergibt die Kontrolle an den Kernel
- Kernel kreiert per Hand die Datenstrukturen für Prozess 0

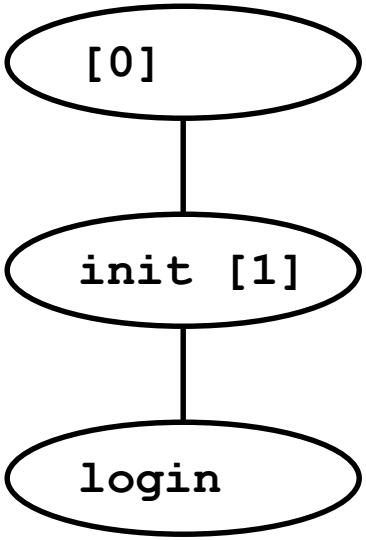


Unix Start: Schritt 2



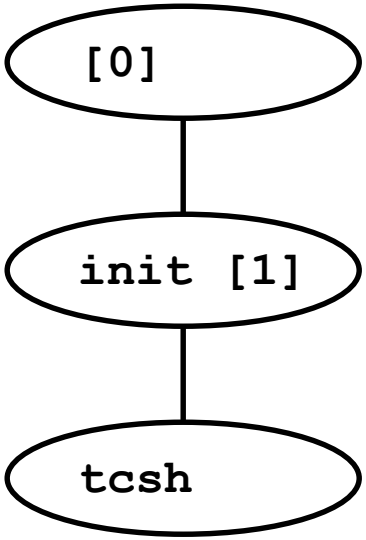
init forkt und exect Daemons per /etc/inittab, und forkt und exect ein getty Programm für die Konsole

Unix Start: Schritt 3



Der getty Prozess exect ein login Programm

Unix Start: Schritt 4



login liest login und passwd.
wenn OK, führt eine *shell aus*
wenn nicht OK,
führt weiteren getty aus