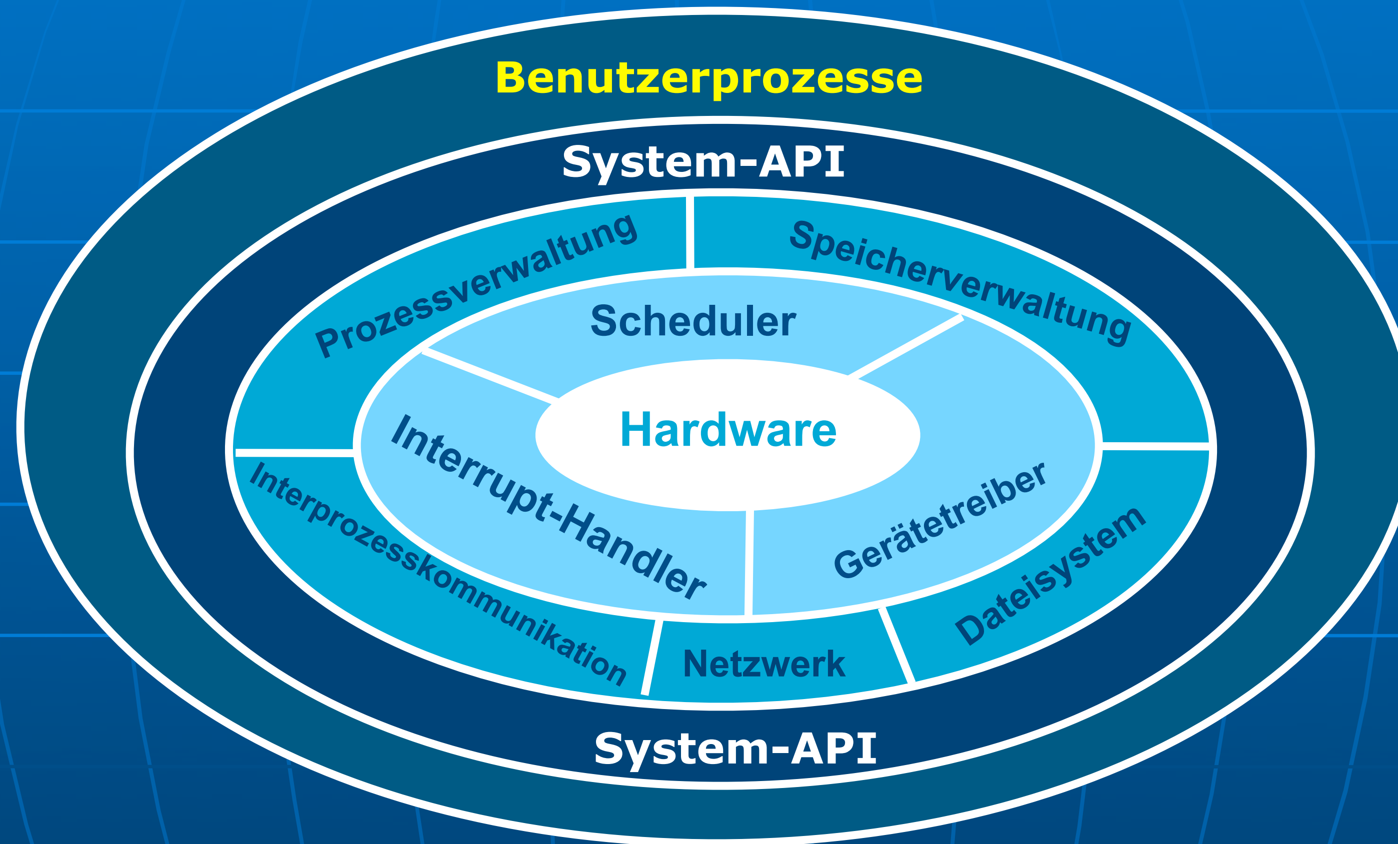
Betriebssystemkonzepte



Margarita Esponda

WS 2011/2012

Betriebssystemkonzepte

- Grundfunktionalität von Betriebssystemen
- Betriebssystemdienste
- Einige grundlegende Konzepte
- Systemaufrufe
- Prozessverwaltung
- Konzepte der Nebenläufigkeit

Grundfunktionalität von Betriebssystemen

Prozessverwaltung

- Erzeugen, Starten, Anhalten und Löschen von Prozessen
- Zuweisung von Ressourcen zu Prozessen
- Schutz von Prozessen, durch Isolierung
- Kommunikation zwischen Prozessen

Geräteverwaltung

- Steuerung der Geräte (Driver)
- Abstraktion von Hardwaredetails
- Schnittstelle für höhere Schichten

Speicherverwaltung

- Verwaltung des Hauptspeichers und des virtuellen Speichers
- Teilung und Schutz von Speicherbereichen

Dateiverwaltung und Objektverwaltung

Netzkommunikation

Energieverwaltung

Betriebssystemdienste

Minimale Dienste aus der Sicht eines Benutzers:

Benutzerschnittstellen

- CLI Komandozeile-Interpreter
- Batch-Schnittstelle
- Graphische Benutzerschnittstelle

Shell-Skriptsprache

CDE X-Windows

KDE

GNOME

Programmausführung

Ein- und Ausgabeoperationen

Dateiverwaltung

Kommunikation

- Gemeinsame Speicher
- Nachrichtenverkehr

Fehlererkennung-System

Betriebssystemdienste

Weitere interne Dienste für alle Benutzer:

Ressourcen-Verwaltung

- CPU-Zeit
- Speicherplatz
- Ein-/Ausgabegeräte

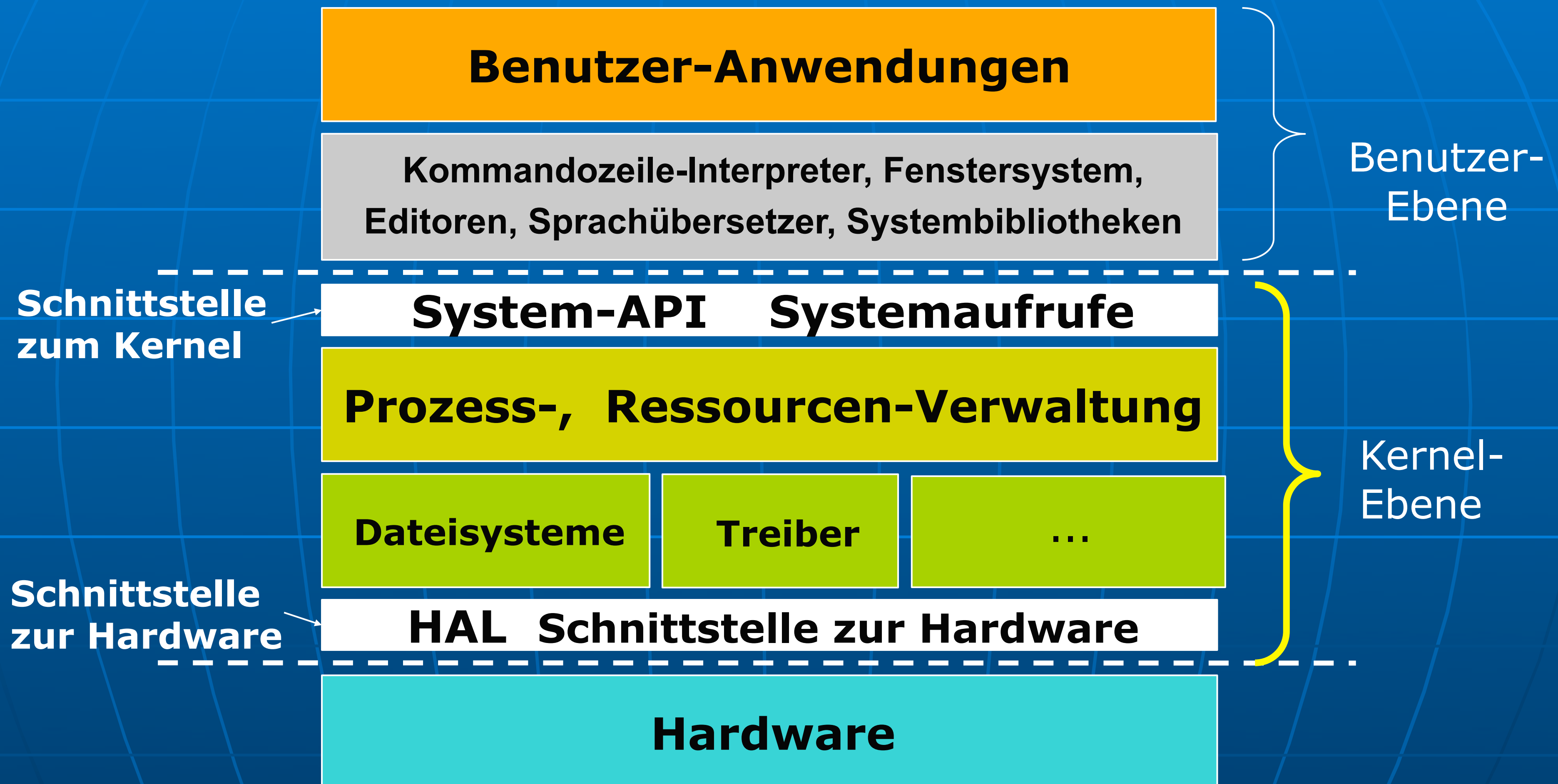
Buchhaltung (Accounting)

- Verbrauch von Ressourcen
- Statistik für Systemanalyse und Optimierungen

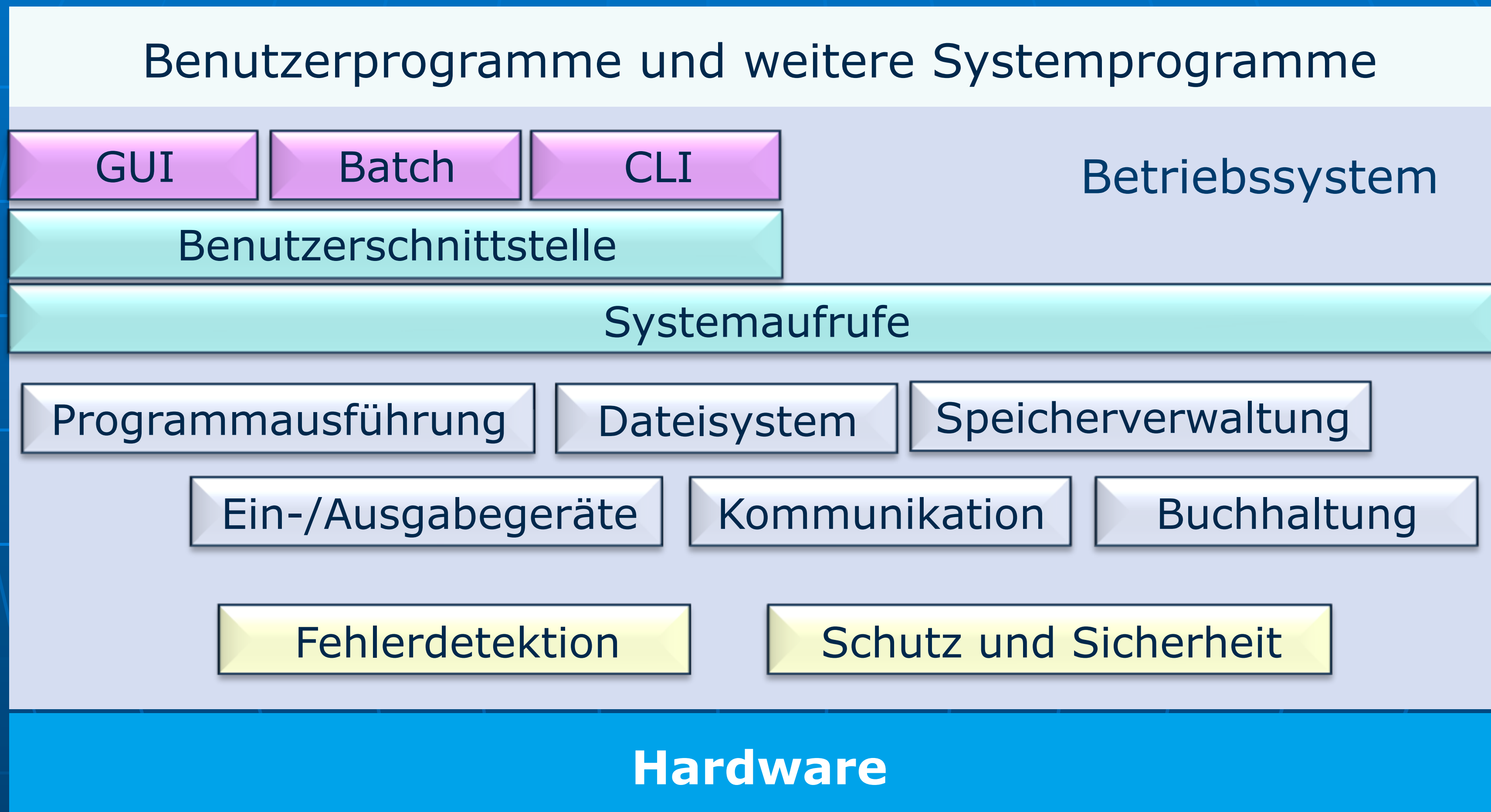
Schutz und Sicherheit

- Kontrollierter Zugriff auf alle Ressourcen
- Informationsschutz
- Sicherheit gegen Angriffe durch das Netz

Schichten-Struktur eines Betriebssystems



Betriebssystemdienste



Betriebssystemdienste

Kommandozeile-Interpreter

- manchmal als Teil der Kernel implementiert
- meistens als getrenntes Programm (Linux/Unix)

Die meisten Kommandointerpreter bieten eine einfache **Skript-Sprache.**

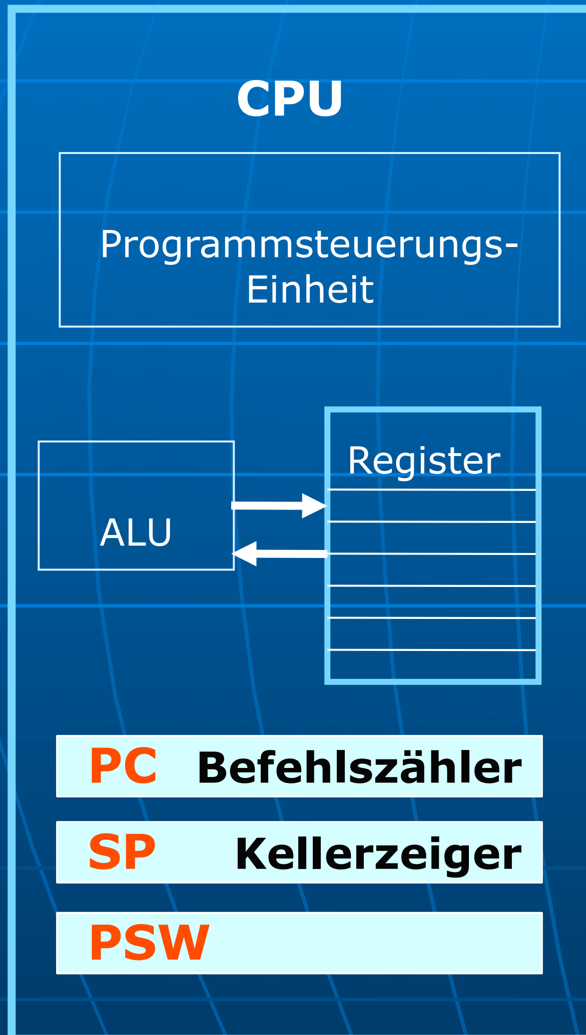
Zwei Möglichkeiten, Kommandos zu interpretieren:

- Die Kommandos sind im Interpreter fest programmiert
  Builtin-Funktionen
- Die Kommandos werden ausgeführt, indem entsprechende Systemprogramme aufgerufen werden (UNIX)

Betriebssystemkonzepte

- Grundfunktionalität von Betriebssystemen
- Betriebssystemdienste
- Einige grundlegende Konzepte
- Systemaufrufe
- Prozessverwaltung
- Konzepte der Nebenläufigkeit

Prozessor



Befehlszähler **PC** — Speicheradresse des nächsten Befehls.

Kellerzeiger **SP** — Zeigt das Ende des aktuellen Kellers im Speicher.

PSW Programm-Statuswort

↓

Priorität-Bit

Ausführungsmodus

Kernel- oder Benutzermodus

Ausführung in zwei Modi

Aus Sicherheitsgründen ist nur das Betriebssystem berechtigt, einige Hardware- und Softwareteile zu benutzen. Um das zu gewährleisten, unterstützt die Hardware zwei Ausführungsmodi.

Kernmodus

Das Betriebssystem arbeitet normalerweise im Kernmodus.

Benutzermodus

Anwendungsprogramme, inklusive Compiler, Editoren, Programmierumgebungen, arbeiten in diesem Modus.

Ausführung in zwei Modi

- Fast alle Prozessoren besitzen zwei Ausführungsmodi
- Eingebettete Prozessoren haben oft keinen Kernmodus
- Java-basierte Betriebssysteme gewährleisten Schutz durch den Interpreter und nicht durch die Hardware

Systemmodus (Kernel-Modus)

Jeder Befehl kann ausgeführt werden

Das Betriebssystem hat Zugriff auf alle Rechnerkomponenten

Benutzermodus (User-Modus)

Eingeschränkter Zugriff

Nur ein Teil der CPU-Befehle kann ausgeführt werden

Nur ein Teil der Hardware kann benutzt werden

Prozess

Ein Prozess ist das zentrale Konzept in jedem Betriebssystem.

Ein Prozess ist ein Programm in Ausführung mit zusätzlicher Kontextinformation.

Prozesse können unterbrochen werden.

Unterbrechungen

(Traps und Interrupts)

Der normale Ablauf der Ausführung von Anweisungen in der CPU wird durch **Traps** oder **Interrupts** unterbrochen.

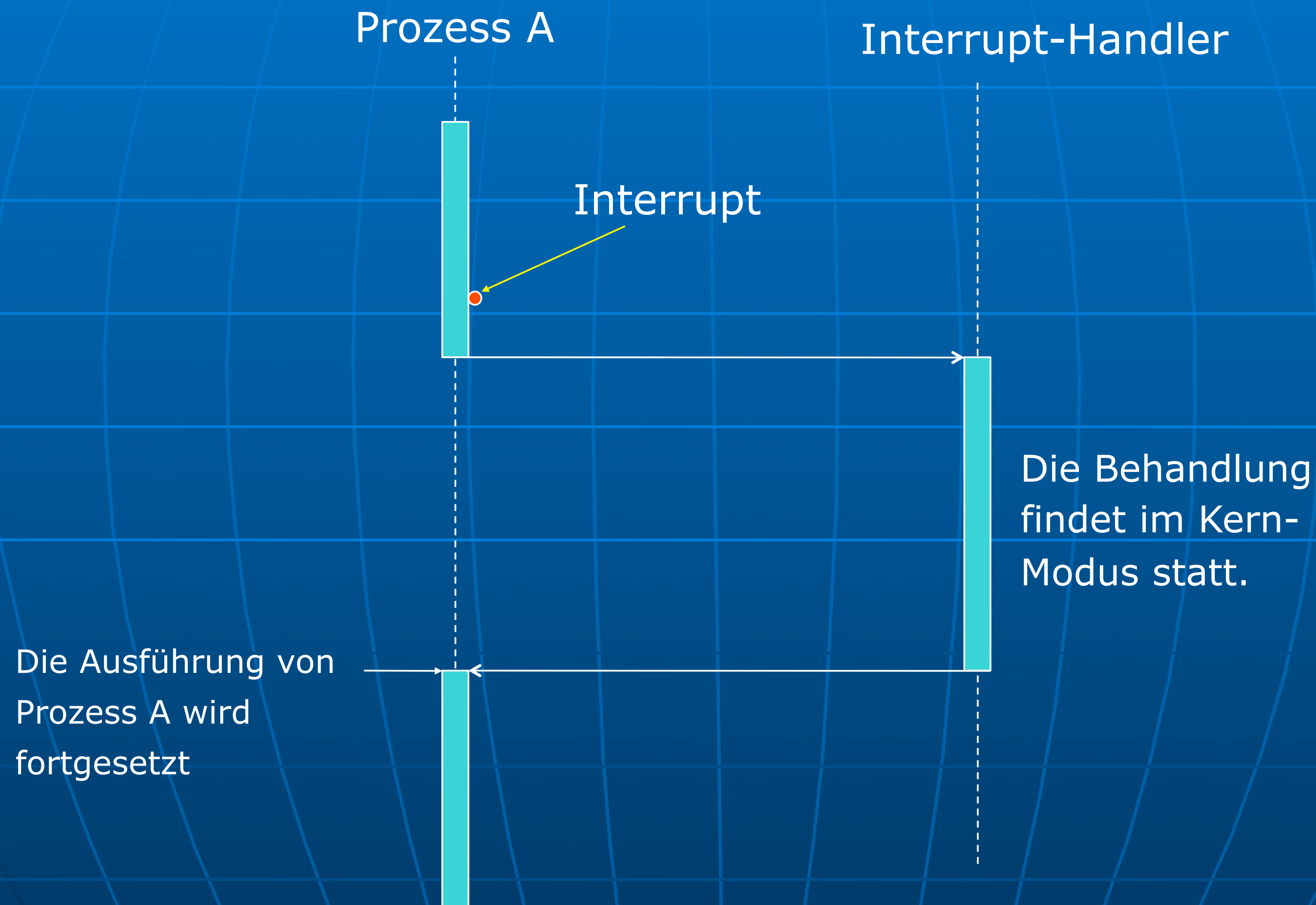
Traps oder **Exceptions** sind normalerweise Fehler, die während der Ausführung von Anweisungen innerhalb der CPU festgestellt werden.

Beispiele:
- Teilung durch 0
- Unerlaubte Speicherzugriffe
- Illegaler Befehlscode
- Stapelüberlauf

Interrupts werden normalerweise von Ein-/Ausgabegeräten signalisiert.

Beim Auftreten von Traps oder Interrupts speichert die CPU den aktuellen Befehlszähler (**PC**) und das Prozessorstatuswort (**PSW**).

Unterbrechungen



Unterbrechungen in Pintos

Zwei Kategorien:

Interne Unterbrechungen (*internal interrupts*)

Synchrone Unterbrechungen

weil diese während der Ausführung von Anweisungen innerhalb des Prozessors verursacht und festgestellt werden

Externe Unterbrechungen (*external interrupts*)

Asynchrone Unterbrechungen

verursacht von Hardware außerhalb des Prozessors

- system timer
- keyboard
- serial port
- disk

Unterbrechungen in 80x86 Architektur

Die 80x86 CPU-Architektur unterstützt **256** verschiedene "*interrupts*"

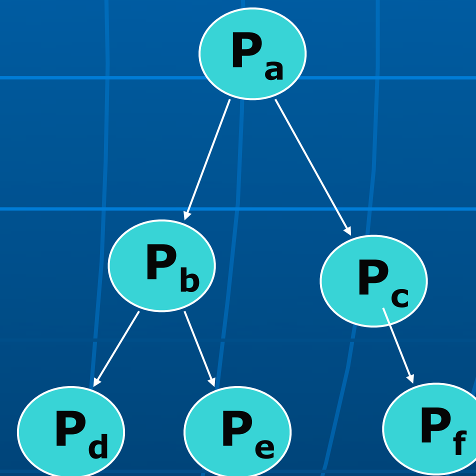
⟹ es gibt 256 "*interrupt handler*", die in der **IDT** "*interrupt descriptor table*" definiert sind.

Prozesshierarchie

In vielen Betriebssystemen, wenn ein Prozess einen anderen Prozess erzeugt, wird der neue Prozess als Kindprozess des erzeugenden Prozesses registriert.

Das führt zu Prozesshierarchien.

Unter Unix bilden alle Prozesse, die zu einem Baum gehören, eine Prozess-Gruppe.

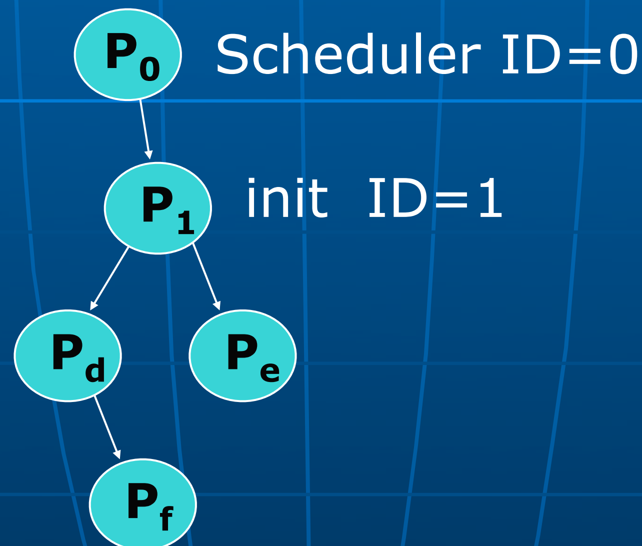


Windows-Prozessmodell

- Windows verwendet ein etwas komplexeres Prozessmodell.
- Ein neuer Prozess wird durch einen Systemaufruf namens **CreateProcess**() erzeugt, der einen leeren Prozess erzeugt, dem anschließend eine Aufgabe zugewiesen werden muss.
- Außerdem ist jeder Prozess im Benutzermodus mit einer numerischen Priorität ausgestattet.
- Der erzeugende Prozess bekommt den **Prozess-Handle** des neuen Prozesses. Ein Prozess-Handle kann weitergegeben werden, sodass hier **keine richtige Prozess-Hierarchie** entsteht.
- Alle Prozesse werden auf der gleichen Ebene erzeugt.

Das Unix-Prozessmodell

- Prozesse können mit Hilfe der **fork-**Funktion erzeugt werden.
- Der erste Prozess, der auf dem Rechner gestartet wird, heißt **init** und bekommt die Prozess-ID (**PID) 1.**
- Jeder Prozess außer dem **init-**Prozess besitzt einen Elternprozess.



Prof. Dr. Margarita Esponda

Das Unix-Prozessmodell

- Wenn der Vaterprozess vor dem Kindprozess beendet wird, werden alle Kindprozesse von dem **init**-Prozess adoptiert.
- Wenn ein Kindprozess dagegen vorher beendet wird, wird er nicht komplett aus dem Speicher und aus der Prozesstabelle entfernt, sondern bleibt als sogenannter **Zombie**-Prozess bestehen.
- **Zombie**-Prozesse bleiben im Speicher, bis die entsprechenden Vaterprozesse den Systemaufruf **wait4**() durchführen.
- Auf diese Weise können die Vaterprozesse auch den Endstatus seiner Kindprozesse untersuchen.

Das Unix-Prozessmodell

- Jeder Prozess reagiert auf eine Reihe verschiedener Signale, die irgendwo in der Betriebssystembibliothek als Konstanten definiert sind.
- Signale werden mithilfe des Systemaufrufs **kill**() an einen Prozess gesendet.
- Einige Signale sind:
  - **SIGTERM** beendet den Prozess normal
  - **SIGKILL** erzwingt einen sofortigen Abbruch
  - **SIGHUP** weist darauf hin, dass eine Verbindung unterbrochen worden ist
  - …

Das Unix-Prozessmodell

- Prozesse können jederzeit selbst die Kontrolle abgeben, indem diese den Systemaufruf **pause**() durchführen. In diesem Fall kann der Prozess nur durch ein Signal wieder geweckt werden.

User- und Group-ID

- *Neben der Prozess-ID besitzt jeder Prozess in einem UNIX-System eine User-ID (**UID**) und eine Group-ID (**GID**). Diese beiden Informationen sind für die Systemsicherheit wichtig.*

- *Prozesse können überwacht werden mit den Programmen **ps**, **pstree** und **top**.*

Das Unix/Linux-Prozessmodell

Zur Verwaltung sind einem Prozess folgende Kennwerte zugeordnet:

PID (process identifier) eine eindeutige, einmalige Kennnummer

PPID (parent process ID)

UID, GID (user ID bzw. group ID) Benutzer-ID und Gruppe-ID

NI Nice-Wert für Grundpriorität

PRI Priorität

CMD Kommando, durch das der Prozess gestartet wurde

TIME verbrauchte Rechenzeit

TTY zugeordnetes Terminal

RSS (resident set size) Größe des Programms im Arbeitsspeicher

...

Betriebssystemkonzepte

- Grundfunktionalität von Betriebssystemen
- Betriebssystemdienste
- Einige grundlegende Konzepte
- Systemaufrufe
- Prozessverwaltung
- Konzepte der Nebenläufigkeit

Systemaufrufe

Eine der grundlegenden Aufgaben eines Betriebssystems ist das zur Verfügungstellen einer normierten Programmierschnittstelle (**System-API**).

Mit Hilfe von **System-API**s werden komplexe Hardwaredetails verborgen, die Programmierung von System- und Benutzeranwendungen wird leichter, und die Programme werden sicherer und portabel.

Systemaufrufe

Eine **System-API** stellt eine Reihe von Funktionen zur Verfügung, innerhalb derer oft ein oder mehrere Systemaufrufe stattfinden.

Die Systemaufrufe sind Dienste, die vom Betriebssystem selber aufgeführt werden.

Die drei meist verwendeten APIs sind:

Win32 API	Windows
POSIX API	UNIX, Linux und Mac OS X
Java API	für die Java virtuelle Maschine

Klassen von Systemaufrufen

Die Systemaufrufe werden je nach Aufgabenbereich meistens in fünf Hauptkategorien eingeteilt.

Systemaufrufe

Prozesssteuerung
- Prozesse starten, beenden oder unterbrechen
- Prozesse synchronisieren

Speicherverwaltung
- Dynamische Speicherallokation

Dateisystem
- Dateien öffnen, schließen, erzeugen oder löschen
- Eigenschaften von Dateien lesen oder ändern
- Zugriffsrechte ändern

Ein-/Ausgabegeräte
- Geräte reservieren oder freigeben
- Lesen und schreiben auf verschiedenen Geräten

Netzkommunikation
- Verbindung herstellen, Nachrichten senden, usw.

Implementierung von Systemaufrufen

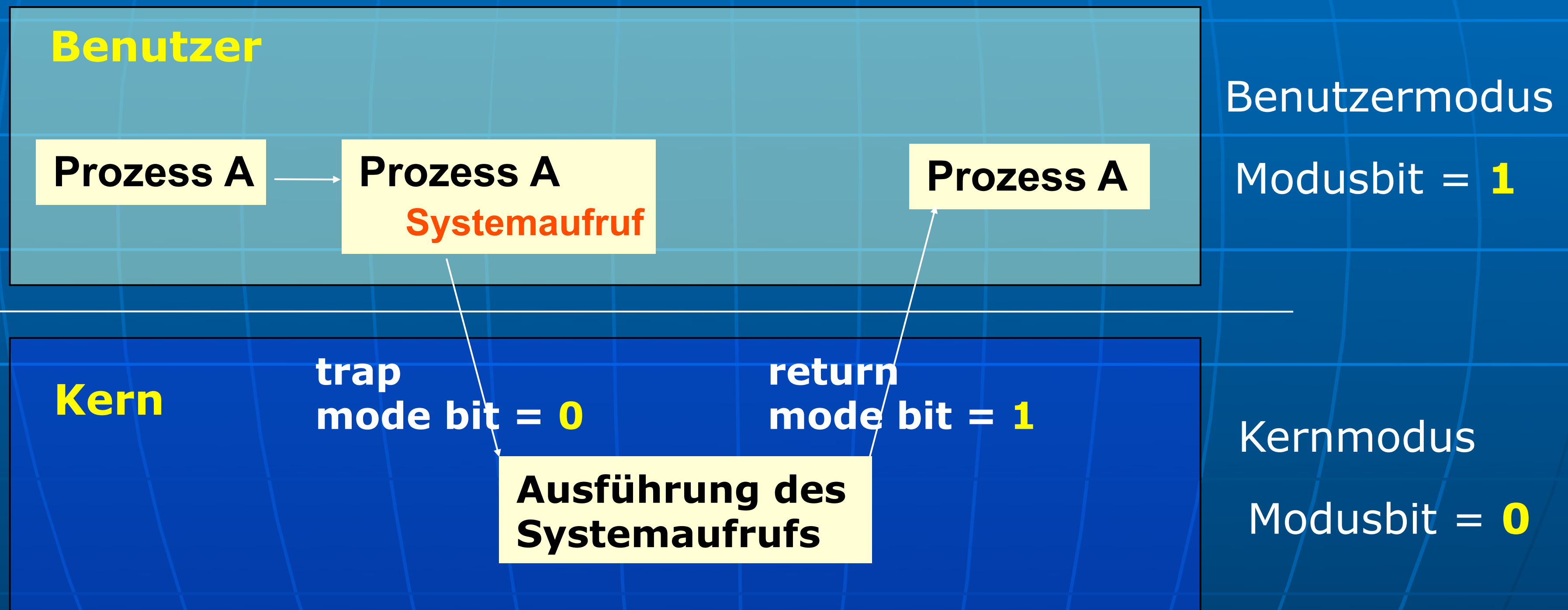
Jeder Systemaufruf bekommt eine Zahl zugeordnet.

Diese Zahl wird als Index in einer Tabelle verwendet, mit Hilfe derer der Systemaufruf stattfindet.

Jeder Systemaufruf verursacht eine **Unterbrechung oder Trap**. Der Benutzerprozess wird unterbrochen, und die Kontrolle wird vom Betriebssystem übernommen.

Ablauf eines Systemaufrufs

Systemaufrufe werden im Kernmodus vom Betriebssystem ausgeführt.



Beispiel eines Systemaufrufs

Benutzermodus

```
#include <stdio.h>
main()
{
    printf("My first C-Program");
}
```

```
    /* C Standardbibliothek */
int printf( const char *format, ...){
  ...
}
```

Kernmodus

trap

return

write()-Systemaufruf

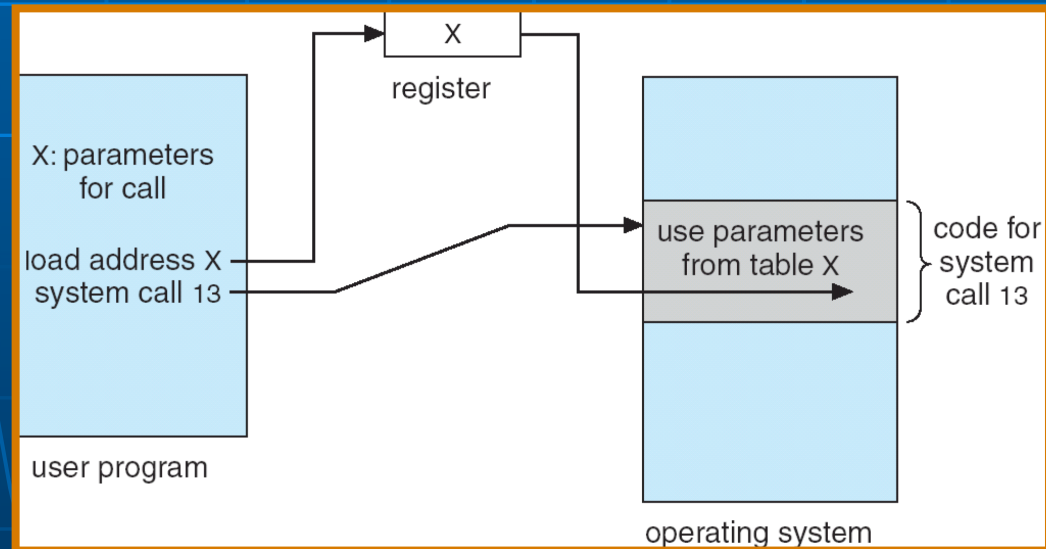
Systemaufrufe

Parameterübergabe

1. Direkt durch die Register.
2. Die Adresse einer Tabelle, in der die Parameter sind, wird in ein Register geschrieben.
3. Durch den Ausführungsstapel des Programms.

2.

Linux und Solaris



aus Silberschatz

Systemaufrufe für Prozesssteuerung

Linux-API

Kurzer Überblick der Systemaufrufe für Prozessmanipulation.

Prozesserzeugung
- **fork()** Dupliziert einen Prozess
- **exec()** Überlagert einen Prozess durch Laden eines neuen Programms

Prozesssynchronisation
- **wait()** Wartet, bis ein anderer Prozess beendet wird

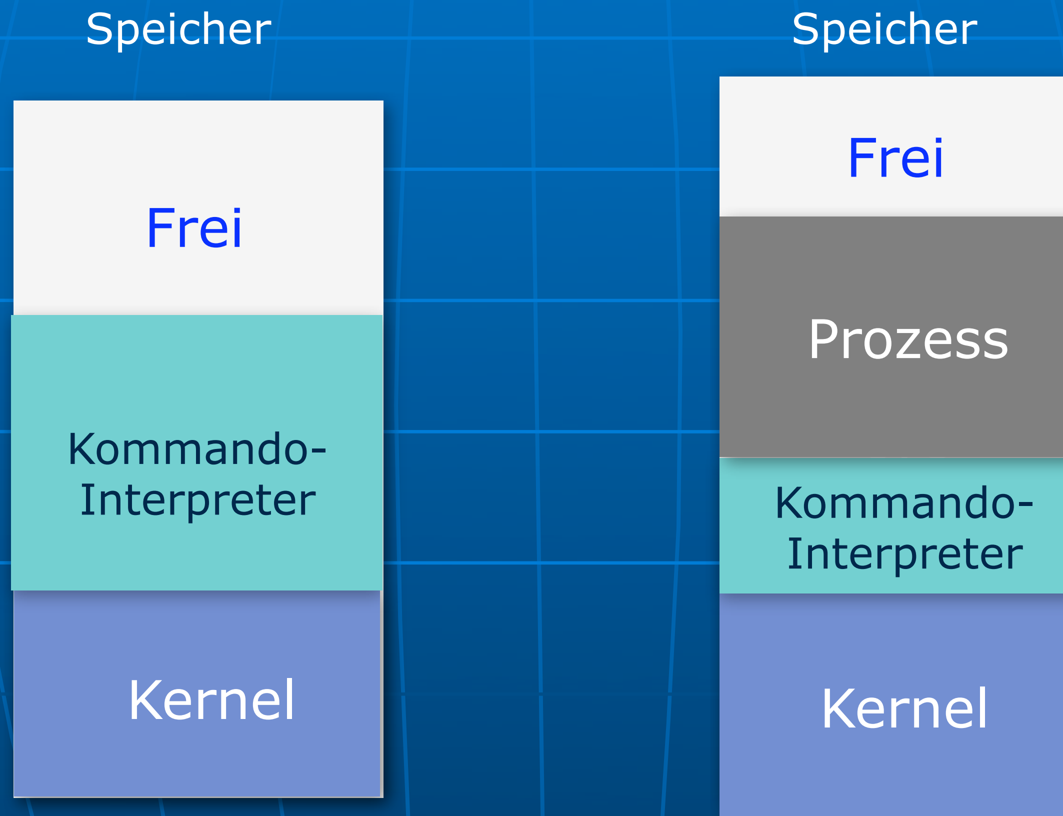
Prozessbeendigung
- **exit()** Beendet einen Prozess

Prozessinformation
- **getpid()** Ermittelt das Prozess-ID
- **getppid()** Ermittelt das Prozess-ID des Vaterprozesses

Kommandozeile-Interpreter

MS-DOS

Einbenutzersystem

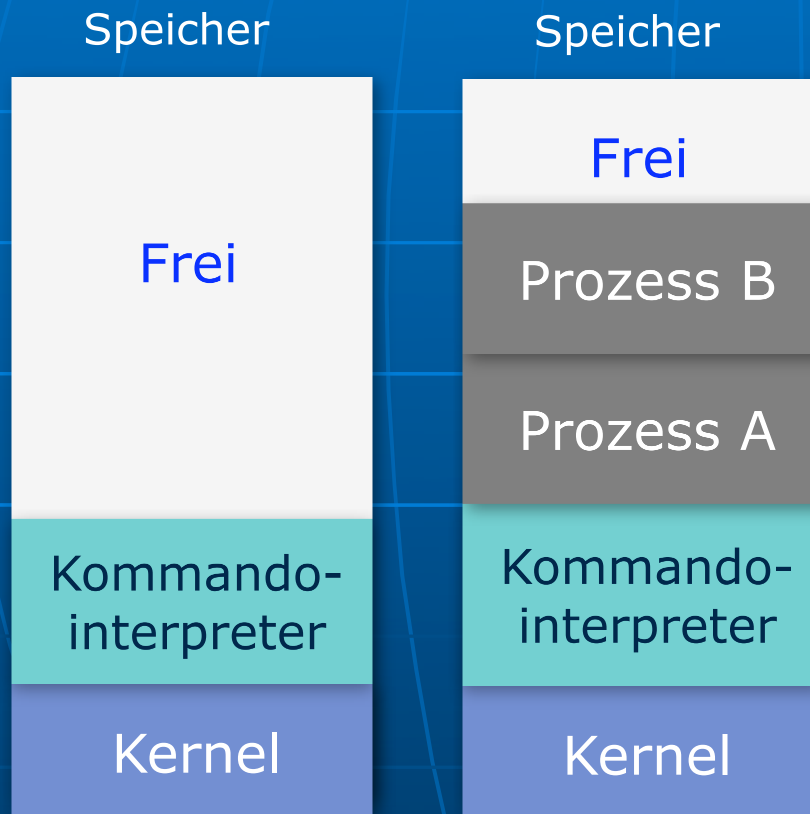


Der neue Prozess wird teilweise in den Speicherbereich des Kommando-Interpreters überlagert.

Kommandozeile-Interpreter

freeBSD

Mehrbenutzersystem



Prozesse bekommen nach einer **exec**-Funktion einen neuen Speicherbereich.

Unmittelbar nach einer **fork**-Funktion teilen Vater und Kind die gleiche Umgebung. Wenn aber Vater oder Kind etwas verändern, werden nur die veränderten Teile entsprechend kopiert.

copy-on-write

Systemaufrufe für Prozesssteuerung

Linux-API

Prozesserzeugung

pid = fork()	Der fork-Systemaufruf erzeugt ein Kind-Prozess, der fast ein exakter Klon des Vater-Prozesses ist. Ein Kind-Prozess beginnt seine Ausführung, sobald fork beendet ist. Vater- und Kind-Prozess laufen weiter simultan. Nur am Rückgabewert der fork-Funktion erkennt ein Prozess, ob er der Vater oder der Sohn ist.

Innerhalb des Kind-Prozesses gibt die fork()-Funktion den Rückgabewert **0**.

Innerhalb des Vater-Prozesses gibt die fork()-Funktion den **PID** des Kind-Prozesses als Rückgabewert.

PID = Prozessidentifikation innerhalb der Betriebssystemverwaltung

Prozesserzeugung

Linux-API

Beispiel:

Vater-Prozess

```
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>

int main() {
  int i;

  if ( fork()==0 )
      for (i=0; i<10000; i++)
           printf( "_" );
  else
      for (i=0; i<10000; i++)
           printf( "P" );

   return 0;
}
```

PID≠0

Diese Schleife wird vom Vater-Prozess ausgeführt.

Kind-Prozess

```
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>

int main() {
  int i;

  if ( fork()==0 )
      for (i=0; i<10000; i++)
           printf( "_" );
  else
      for (i=0; i<10000; i++)
           printf( "P" );

   return 0;
}
```

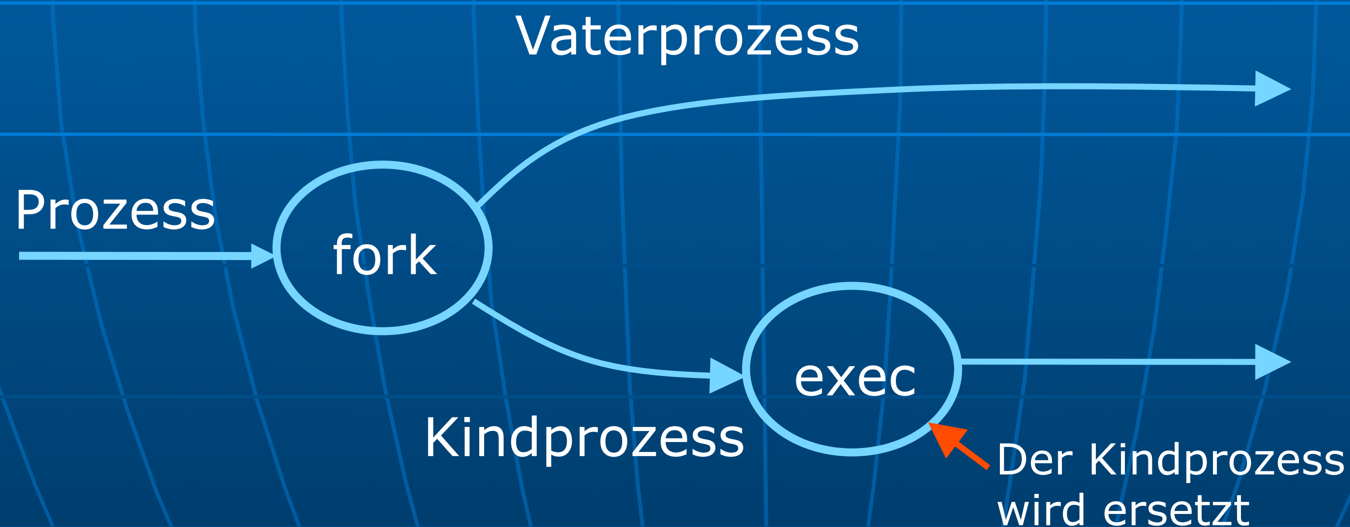
0

Diese Schleife wird vom Kind-Prozess ausgeführt.

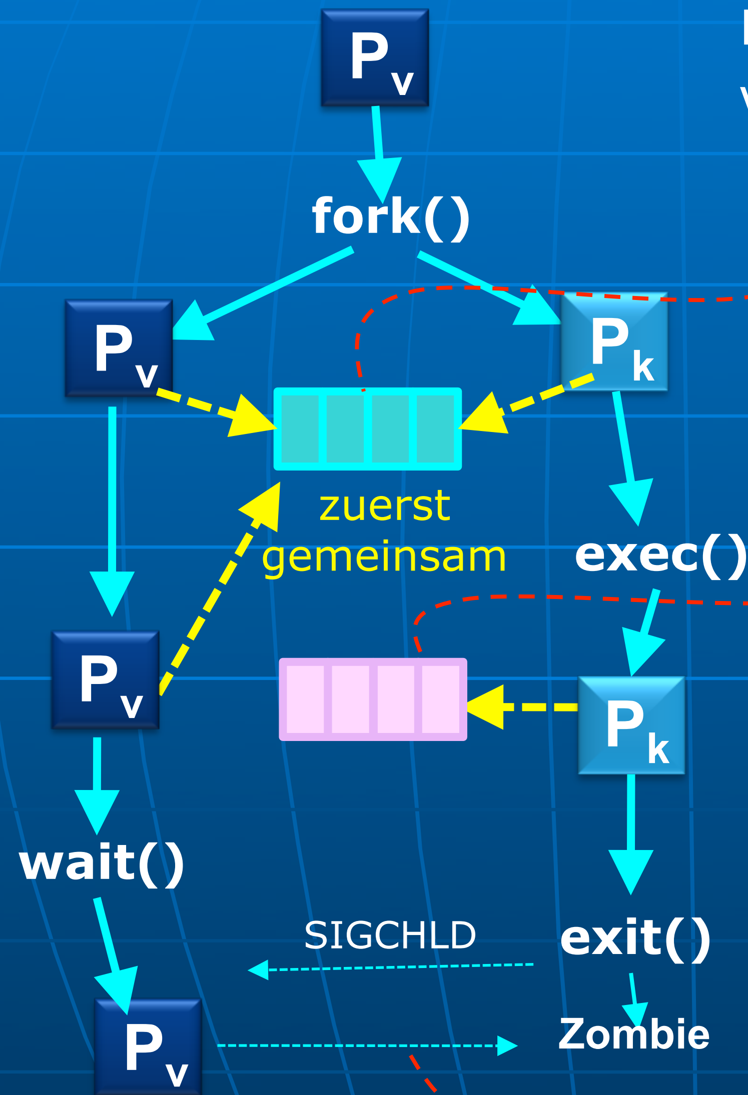
Prozesserzeugung

Linux

Der **fork**-Systemaufruf wurde aus Effizienzgründen so implementiert und wird erst sinnvoll, wenn wir den neu erzeugten Kindprozess mit Hilfe eines der **exec**-Systemaufrufe durch das Laden eines neuen Programms überladen.



Prozesserzeugung in Linux



Hier wird das **copy-write-** Verfahren verwendet.

Das Kindprogramm bekommt nach der **exec**-Funktion eine neue Speicherumgebung

Der Vaterprozess entfernt den Kindprozess aus der Prozesstabelle

wait, waitpid

```
long waitpid(p id_t pid,  unsigned int *stat_addr,  int options)
```

Die waitpid-Funktion wartet auf die Beendigung eines durch pid angegebenen Prozesses. Der exit-Code wird in die stat_addr-Adresse geschrieben.

Die Funktion fragt mit einer Schleife alle Kindprozesse des aktuellen Prozesses ab, ob sie zur angegebenen PID passen.

Wenn pid

> 0 wartet auf den Kindprozess mit einer **PID** gleich **pid**

= 0 wartet auf jeden Kindprozess, dessen Prozessgruppennummer (PGRP) gleich der PGRP des rufenden Prozesses ist.

= -1 wartet auf jeden Kindprozess.

Prozesserzeugung

Win32-API

```
#include <stdio.h>
#include <windows.h> // aus Silberschatz

int main(void) {
    STARTUPINFO sInfo;
    PROCESS_INFORMATION pInfo;
    ZeroMemory( &sInfo, sizeof(sInfo) );
    sInfo.cb = sizeof( sInfo );
    ZeroMemory( &pInfo, sizeof(pInfo) );

      /* Ein neuer Prozess mit dem Programm p.exe wird gestartet*/
    if( !CreateProcess( NULL, "C:\\p.exe",
                        NULL, NULL, FALSE, 0, NULL,
                        NULL, &sInfo, &pInfo ) )
    {   printf( "Fehler im CreateProcess()-Systemaufruf" );
        return -1;
    } else {
        WaitForSingleObject(pInfo.hProcess, INFINITE);
        printf( "Prozess beendet! \n" );
        CloseHandle( pInfo.hProcess );
        CloseHandle( pInfo.hThread );
    }
}
```

Prozesserzeugung

Linux

Die Familie der **exec**-Funktionen

execl, execv, execle, execve, execlp, execvp

Beim Aufruf einer **exec**-Funktion wird das aufrufende Programm von dem als Argument neu angegebenen Programm komplett ersetzt.

Das ursprüngliche Programm hört auf zu laufen, nachdem das Programm, das in der **exec**-Funktion spezifiziert wird, gestartet werden kann.

Bedeutung

e	Die Umgebungsvariablen als Vektor.
l	Die Kommandozeilenargumente in Form eines Vektors.
v	Die Kommandozeilenargumente in Form eines Vektors.
p	Ein Dateiname anstatt eines Pfadnamens als Argument zum Aufruf des Programms.

Prozessersetzung

Linux-API

execl (char *pathname, char *arg_0 , ... , NULL)

Der Systemaufruf **execl** überlagert die Prozessumgebung des aufrufenden Prozesses mit dem Programm ***pathname*** und startet seine Ausführung. Die nächsten Parameter sind die Argumente des neuen aufgerufenen Programms. Der letzte Parameter muss immer **Null** sein und kennzeichnet das Ende der Parameterliste.

Beispiel:

```
#include <stdio.h>
#include <unistd.h>

int main() {
        return execl ( "/bin/ls", "ls", "-l", NULL);
}
```

Prozessersetzung

Linux-API

execv (char *pathname, char *argv[])

Der Systemaufruf **execv** überlagert die Prozessumgebung des aufrufenden Prozesses mit dem Programm ***pathname*** und startet seine Ausführung. Der zweite Parameter ist die Argumentliste des neuen aufgerufenen Programms.

Beispiel:

```
#include <stdio.h>
#include <unistd.h>


int main() {
        char* argumente[4] = { "ls", "-l", "/usr/bin", NULL };
        return execv ( "/bin/ls", argumente );
}
```

Warten auf einen Prozess

Linux

Die **wait-** und **waitpid**-Funktionen

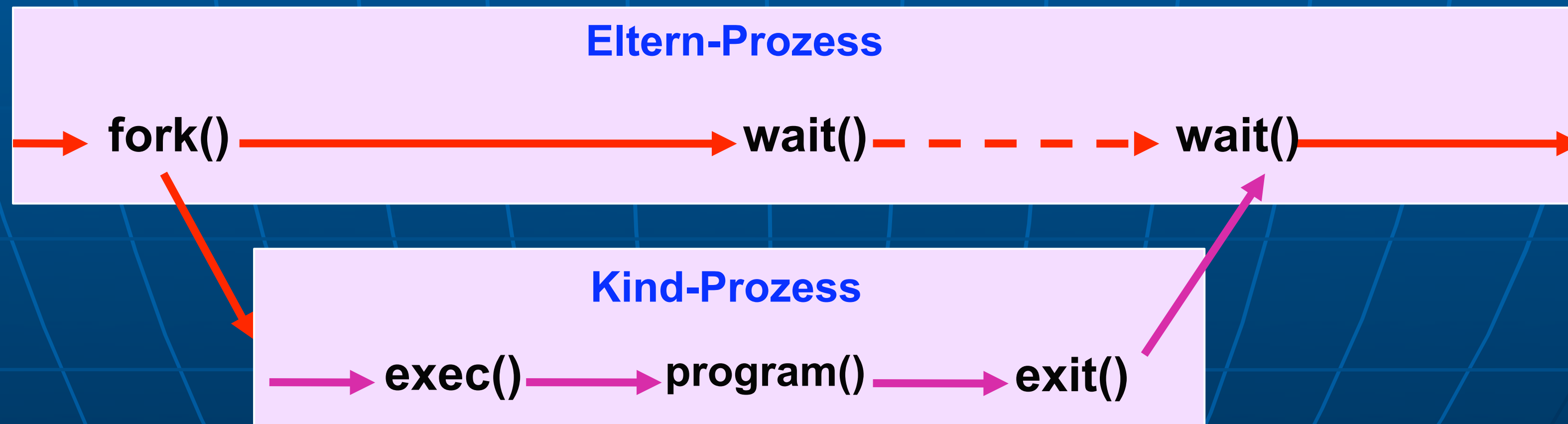
```
pid_t  wait ( int *status );
```

wartet auf das Ende irgendeines Kindes

```
pid_t waitpid ( pid_t pid,  int *status,  int options );
```

wartet auf das Ende des Kindes mit der Prozessnummer = pid

In beiden Fällen wird in die Variable ***status*** der Endzustand und der Beendigungsgrund des Kindprozesses zurück geschrieben.



Prof. Dr. Margarita Esponda

Warten auf einen Prozess

```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <stdlib.h>

int main() {
  pid_t  pid;

  switch ( pid = fork () )  {
  case -1:    printf ("Fehler bei fork()\n");
              return -1;
  case  0:    sleep (5);   /* ein Nickerchen */
              printf ( "--- im Kindprozess ---\n" );
              break;
  default:    printf ( "--- im Elternprozess ---\n" );
              sleep (2);   /* ein Nickerchen */
              waitpid(pid,0,0);
  }
  return  0;
}
```

Prozess beenden

Linux

Die exit- und kill-Funktionen

Ein Prozess endet normalerweise, wenn:

- er selbst den Systemaufruf exit (int ret_value) ausführt.
- durch ein return- im Hauptprogramm (main).
- wenn main ohne ein return endet, erzeugen die meisten Compiler einen Code, der den Wert 0 zurückliefert.

Durch die Verwendung der kill-Funktion können auch Prozesse beendet werden, indem man ihnen das Signal SIGKILL sendet.

```
int kill ( pid_t victim_id, int signal );
```

Shell

Kommandozeilen-Interpreter

Der Kommandozeilen-Interpreter oder *Schell* ist ein Programm, das in der Lage ist Kommandos aus einfachen Eingabezeilen zu lesen, zu interpretieren und auszuführen.

Der Kommandozeilen-Interpreter ist eine gute Hilfe, um mit einigen Systemaufrufen vertraut zu werden.

Unix

sh minimale Funktionalität

csh, ksh, bash, pdksh, zsh, usw.

Windows

MS-DOS

Ein einfaches Shell

Kommandozeilen-Interpreter

Mit Hilfe der fork- und exec-Systemaufrufe kann man bereits einen einfachen Kommandozeilen-Interpreter programmieren.

```
while ( TRUE ) {
      type_prompt( );                                  /* gibt einen Prompt aus */
      read_command (command, parameters)  /* liest eine Kommandozeile */

      if (( fork ) != 0 ) {                            /* erzeugt einen Kindprozess */
            waitpid( -1, &status, 0 );
      } else {
            execve ( command, parameters, 0 );  /* führt eine Kommandozeile
                                                        aus */
      }
}
```

Einige Linux-API- und Win32-API-Systemaufrufe

	Windows	Unix
Prozesssteuerung	CreateProcess() ExitProcess() WaitForSingleObject()	fork() exit() wait()
Dateiverwaltung	CreateFile() ReadFile() WriteFile() CloseHandle()	open() read() write() close()
Gerätesteuerung	ReadConsole() WriteConsole()	read() write()
Information	GetCurrentProcessID()	getpid()
Kommunikation	CreatePipe()	pipe()
Schutz	SetFileSecurity() SetSecurityDescriptorGroup()	chmod() chown()

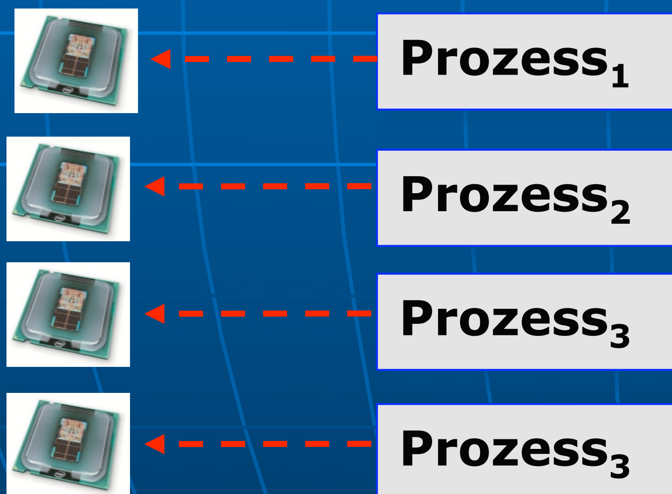
Betriebssystemkonzepte

- Grundfunktionalität von Betriebssystemen
- Betriebssystemdienste
- Einige grundlegende Konzepte
- Systemaufrufe
- Konzepte der Nebenläufigkeit
- Prozessverwaltung

Parallelität

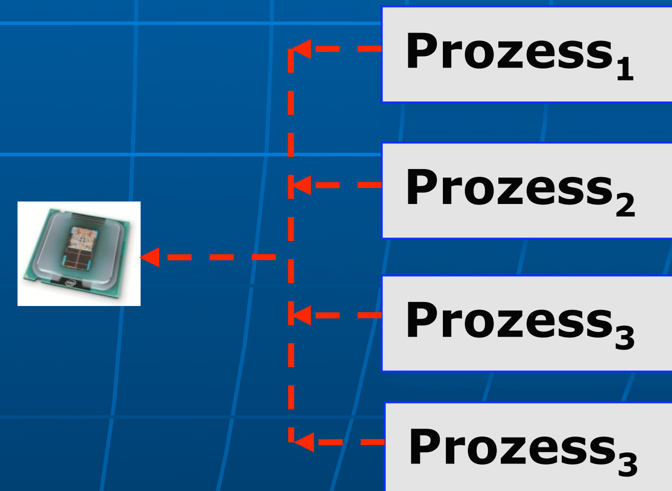
Echte Parallelität

Prozesse laufen auf unterschiedlichen CPUs



Virtuelle Parallelität

Mehrere Prozesse laufen auf der selben CPU



Grundlegende Begriffe

"single program" **In der Steinzeit der PCs durfte man nur einzelne Programme streng sequentiell ausführen.**

"task switching" **Mehrere Programme liegen im Hauptspeicher. Durch Umschaltung setzt man die Ausführung eines der Programme fort.**

"multitasking" **Das Betriebssystem schaltet die Programme um. Man hat das Gefühl, dass mehrere Programme parallel laufen.**

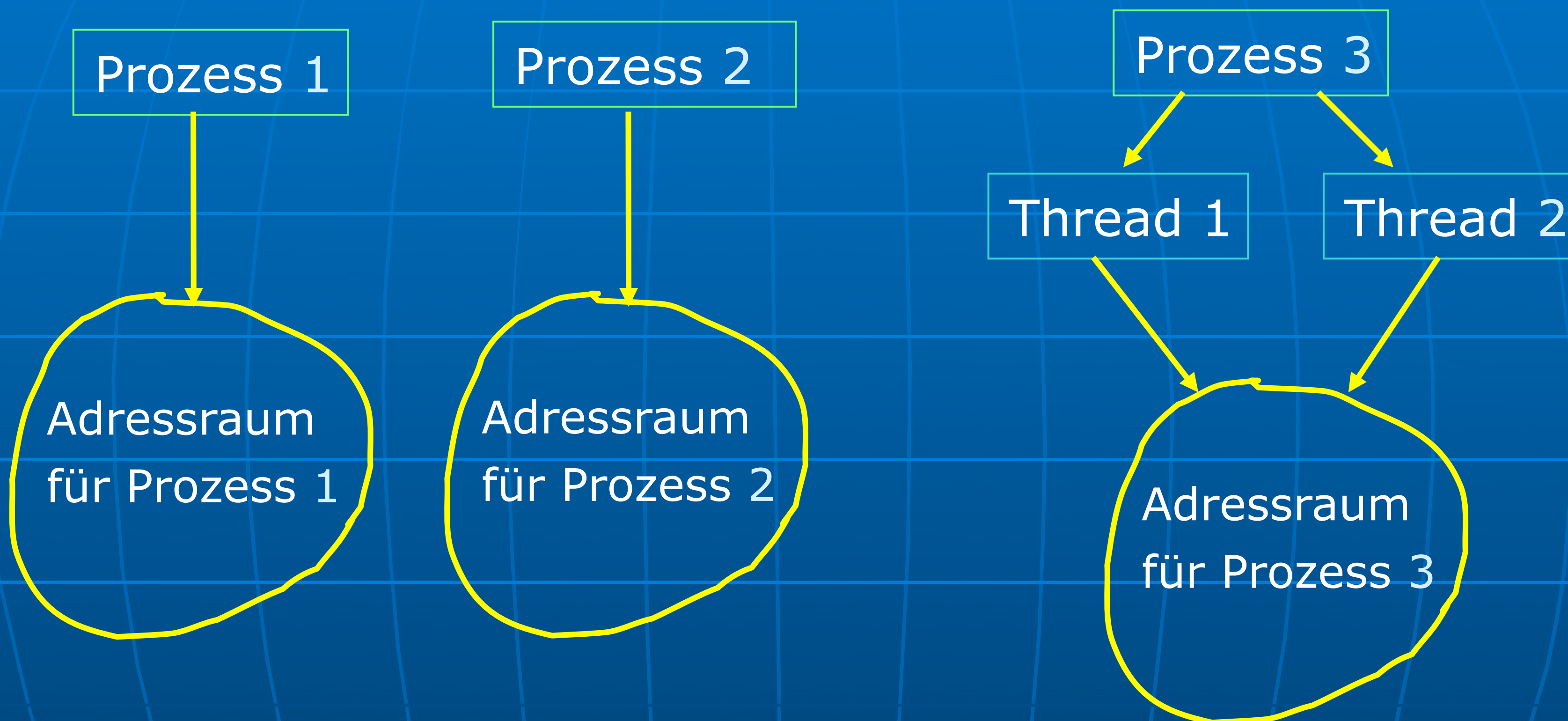
"multithreading" **Es gibt mehrere Programmfäden, die parallel innerhalb eines Programms ausgeführt werden.**

"Multitasking" vs. "Multithreading"

aus der Sicht eines Benutzers

"Multitasking"	"Multithreading"
mehrere Prozesse (process)	mehrere Programmfäden (thread)
jeder Prozess hat eine eigene Umgebung	alle Threads laufen in der gleichen Umgebung
Kein gemeinsamer Adressraum	gemeinsamer Adressraum !!!
Verwaltung durch das Betriebssystem	Verwaltung innerhalb eines Prozesses

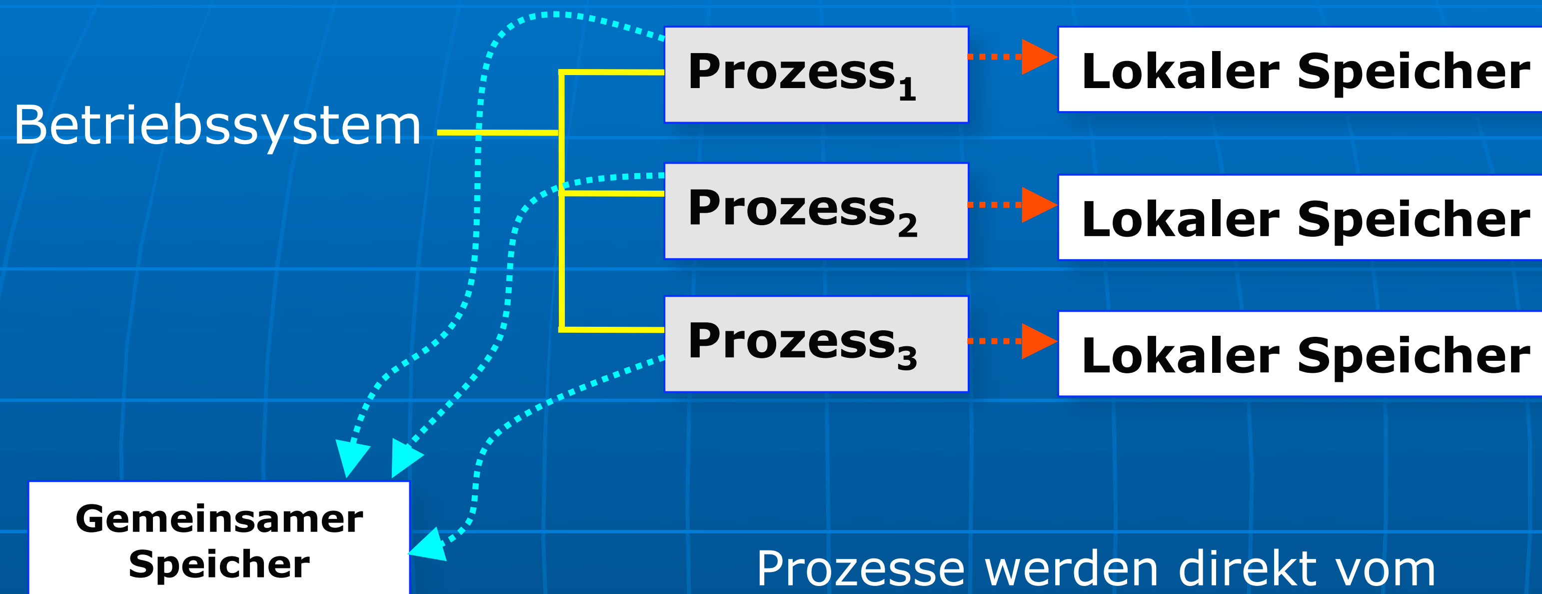
Multitasking + Multithreading



aus der Sicht eines Benutzers

Multitasking

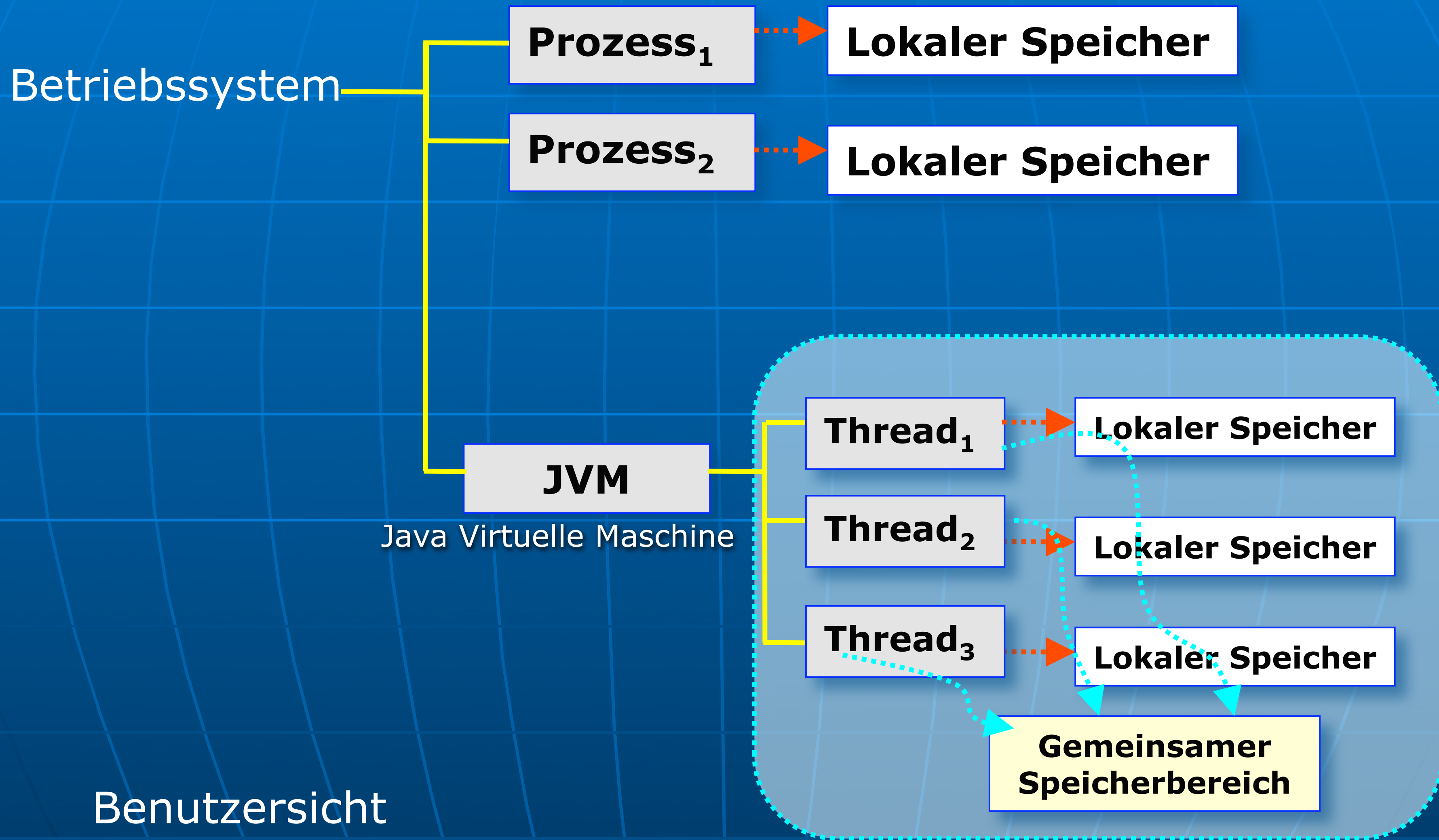
Benutzersicht



Prozesse werden direkt vom Betriebssystem verwaltet.

Jeder Prozess bekommt einen eigenen Speicherbereich vom Betriebssystem zugeteilt.

Multitasking + Multithreading



Es gibt kein Java-Programm ohne Threads.

Betriebssystem

Java-Anwendung

garbage collector

main-Thread

GUI

setVisible(true)

event-dispatcher

Betriebssystem

Browser

garbage collector

Java-Applet

Java-Applet

init-Thread

init-Thread

Threads mit gemeinsamen Daten

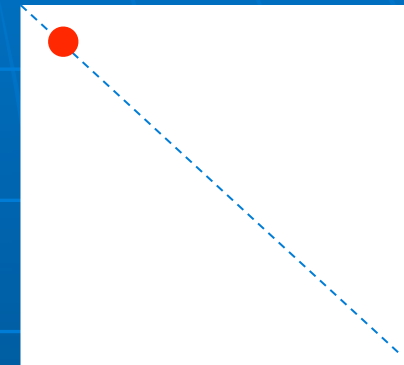
Synchronisationsprobleme

T1

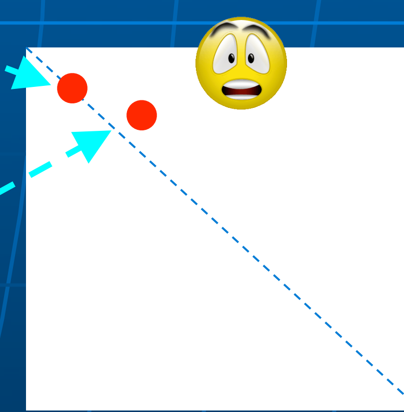
```
public void run(){
  .
  .
  x = x + 1;
  y = y + 1;
  paint();
  .
}
```

T2

```
public void run(){
  .
  .
  x = x * 2;
  y = y * 2;
  paint();
  .
}
```



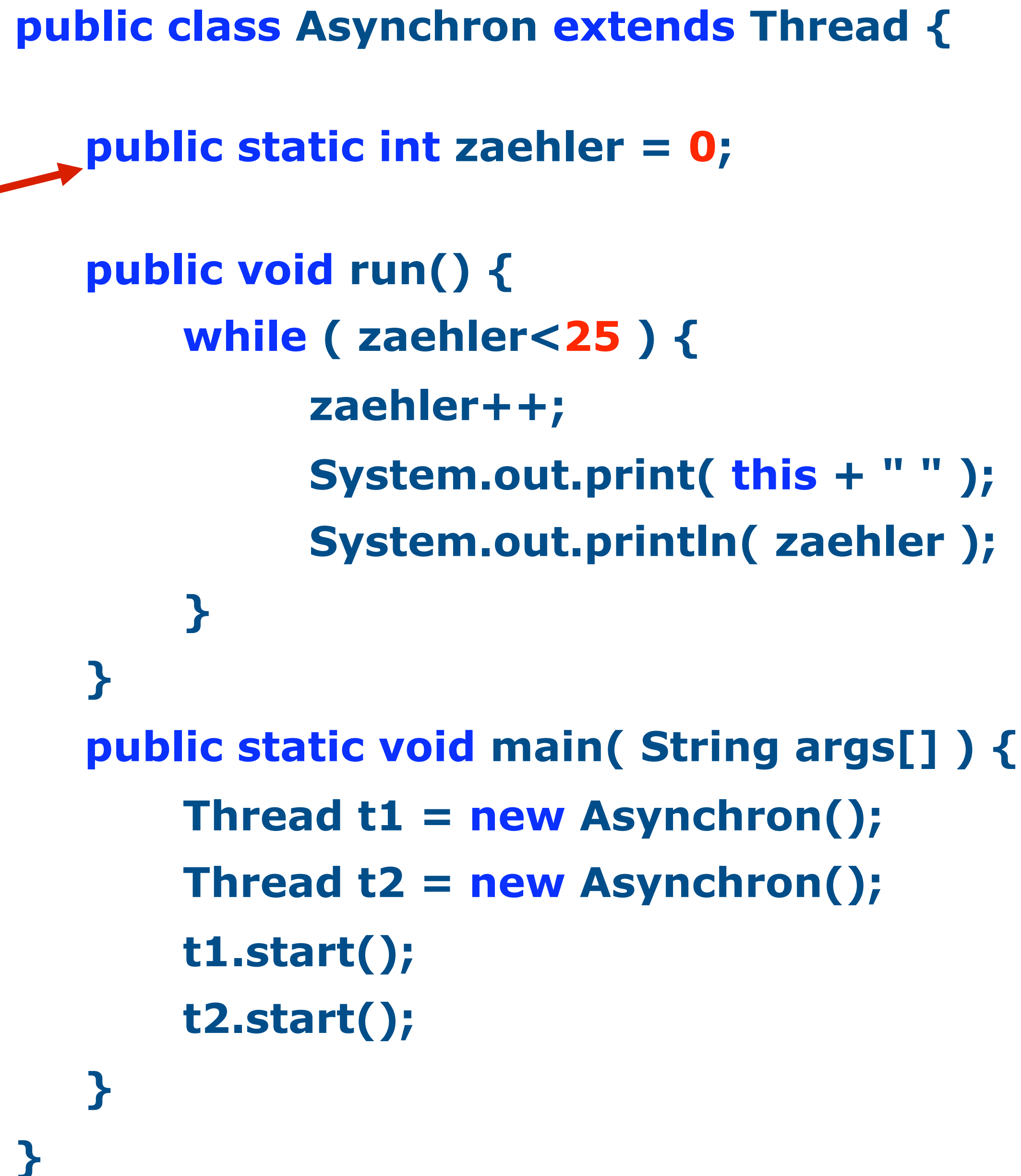
			3, 3
T1:	x = x + 1;		4, 3
	x = x * 2;	T2:	8, 3
	y = y * 2;	T2:	8, 6
	paint();	T2:	8, 6
T1:	y = y + 1;		8, 7



Synchronisationsprobleme

Klassenvariable

Gemeinsame Variable für alle Objekte der Klasse Asynchron.



Synchronisationsprobleme

```
public void run() {
   while ( zaehler<25 )
   {
      zaehler++;
      System.out.print( this + " " );
      System.out.println( zaehler );
   }
}
```

```
public void run() {
   while ( zaehler<25 )
   {
      zaehler++;
      System.out.print( this + " " );
      System.out.println( zaehler );
   }
}
```

Synchronisationsprobleme

```
public void run() {
   while ( zaehler<25 )
   {
      zaehler++;
      System.out.print( this + " " );
      System.out.println( zaehler );
   }
}
```

```
public void run() {
   while ( zaehler<25 )
   {
      zaehler++;
      System.out.print( this + " " );
      System.out.println( zaehler );
   }
}
```

Synchronisationsprobleme

```
Thread[Thread-0,5,main] 1
Thread[Thread-0,5,main] 2
Thread[Thread-0,5,main] 3
Thread[Thread-0,5,main] 4
Thread[Thread-0,5,main] 5
Thread[Thread-0,5,main] 6
Thread[Thread-1,5,main] Thread[Thread-0,5,main] 8
8
Thread[Thread-0,5,main] 9
Thread[Thread-0,5,main] 10
Thread[Thread-0,5,main] 11
Thread[Thread-0,5,main] 12
Thread[Thread-0,5,main] 13
Thread[Thread-0,5,main] 14
Thread[Thread-0,5,main] 15
Thread[Thread-0,5,main] 16
Thread[Thread-0,5,main] 17
Thread[Thread-0,5,main] 18
Thread[Thread-0,5,main] Thread[Thread-1,5,main] 20
20
Thread[Thread-0,5,main] Thread[Thread-1,5,main] 22
22
Thread[Thread-0,5,main] Thread[Thread-1,5,main] 24
Thread[Thread-0,5,main] 25
```

Synchronisationsprobleme

```
public void run() {
   while ( zaehler<25 )
   {
      zaehler++;
      System.out.print( this + " " );
      System.out.println( zaehler );
   }
}
```

```
public void run() {
   while ( zaehler<25 )
   {
      zaehler++;
      System.out.print( this + " " );
      System.out.println( zaehler );
   }
}
```

Synchronisationsprobleme

zaehler 26

t0

t1

```
public void run() {
   while ( zaehler<25 )
   {
      zaehler++;
      System.out.print( this + " " );
      System.out.println( zaehler );
   }
}
```

```
public void run() {
   while ( zaehler<25 )
   {
      zaehler++;
      System.out.print( this + " " );
      System.out.println( zaehler );
   }
}
```

Wechselseitiger Ausschluss muss gewährleistet werden.

T1

```
public void run(){
  .
  .
  x = x + 1;
  y = y + 1;
  paint();
  .
}
```

Kritischer Abschnitt

T2

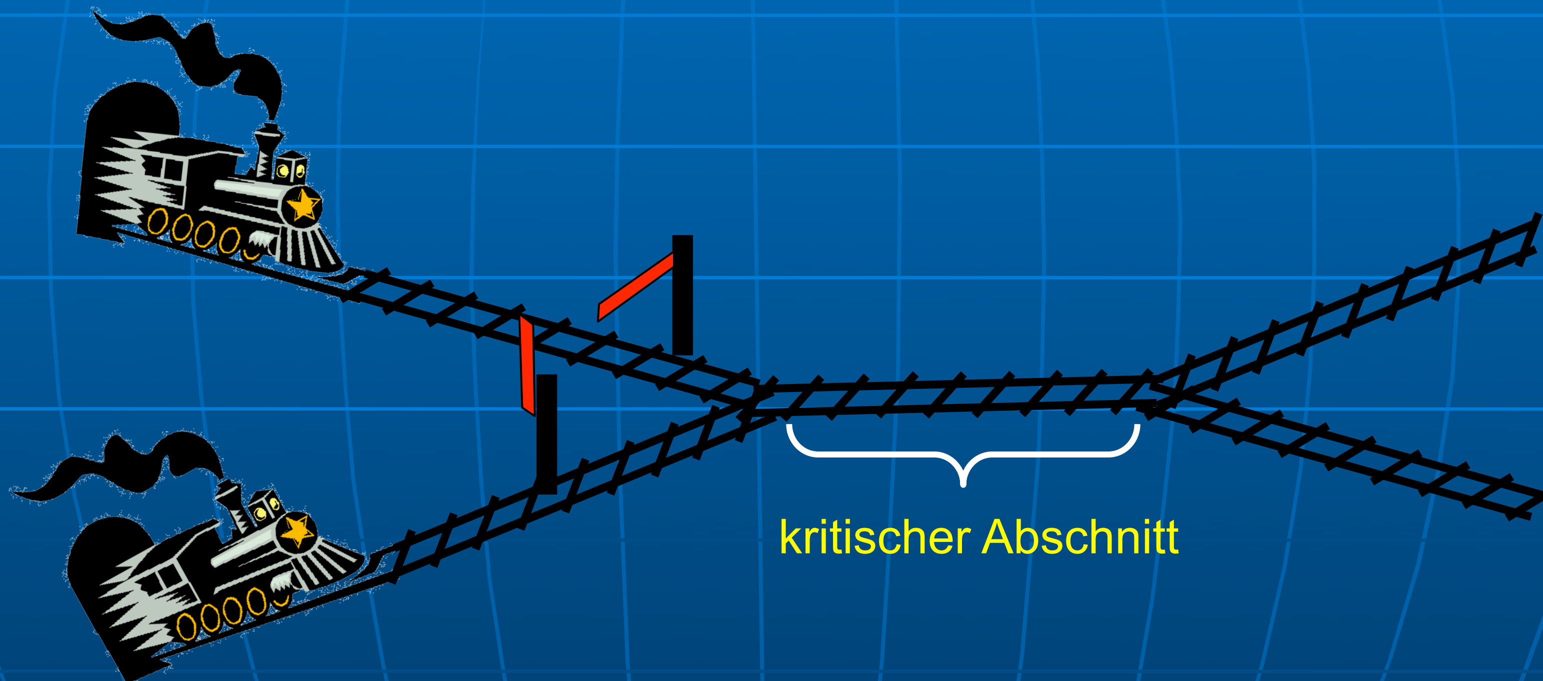
```
public void run(){
  .
  .
  x = x * 2;
  y = y * 2;
  paint();
  .
}
```

Kritischer Abschnitt

nur ein Thread im kritischen Abschnitt

Kritische Abschnitte

Wechselseitiger Ausschluss muss gewährleistet werden, wenn gemeinsame Ressourcen benutzt werden.



Kritische Abschnitte

Eines der wichtigsten Ziele bei der Entwicklung von Betriebssystemen ist es, **geeignete** und **effiziente** Synchronisationsmechanismen anzubieten, um den Wechselseitigen Ausschluss zu garantieren.

Das Vermeiden von kritischen Abschnitten reicht allein nicht aus, um eine effiziente Ausführung von parallelen Prozessen zu garantieren.

Kritische Abschnitte

Was ist eine gute Lösung?

- Keine zwei Prozesse/Threads dürfen in ihren kritischen Regionen sein.
- Es dürfen keine Annahmen über die Geschwindigkeit und Anzahl der CPUs gemacht werden.
- Kein Prozess/Thread, der außerhalb seines kritischen Abschnitts läuft, darf anderen Prozessen den Eintritt zum kritischen Abschnitt blockieren.
- Kein Prozess darf ewig auf seinen kritischen Abschnitt warten.
- **Die Lösung soll auch bei Mehrprozessorsystemen funktionieren.**