

## 4. Foliensatz

# Betriebssysteme und Rechnernetze

Prof. Dr. Christian Baun

Frankfurt University of Applied Sciences  
(1971–2014: Fachhochschule Frankfurt am Main)  
Fachbereich Informatik und Ingenieurwissenschaften  
[christianbaun@fb2.fra-uas.de](mailto:christianbaun@fb2.fra-uas.de)

# Lernziele dieses Foliensatzes

- Am Ende dieses Foliensatzes kennen/verstehen Sie...
  - Was ein **Prozess** aus Sicht des Betriebssystems ist
  - Welche Informationen der **Prozesskontext** im Detail enthält
    - **Benutzerkontext**
    - **Hardwarekontext**
    - **Systemkontext**
  - die unterschiedlichen Prozesszustände anhand verschiedener **Zustands-Prozessmodelle**
  - wie das **Prozessmanagement** mit **Prozesstabellen**, **Prozesskontrollblöcken** und **Zustandslisten** funktioniert
  - welche Schritte Betriebssysteme beim **Erstellen von Prozessen** (via fork oder exec) oder **Löschen von Prozessen** durchführen
  - was **Forkbomben** sind
  - die **Struktur von UNIX-Prozessen im Speicher**
  - was **Systemaufrufe** (System Calls) sind und wie sie funktionieren

Übungsblatt 4 wiederholt die für die Lernziele relevanten Inhalte dieses Foliensatzes

# Prozess und Prozesskontext

## Wir wissen bereits...

- Ein **Prozess** (lat. *procedere* = voranschreiten) ist eine Instanz eines Programms, das ausgeführt wird
  - Prozesse sind dynamische Objekte und repräsentieren sequentielle Aktivitäten im Computer
  - Auf Computern sind immer mehrere Prozesse in Ausführung
  - Die CPU wird im raschen Wechsel zwischen den Prozessen hin- und hergeschaltet
- 
- Ein Prozess umfasst außer dem Programmcode noch seinen **Kontext**
  - 3 Arten von Kontextinformationen speichert das Betriebssystem:
    - **Benutzerkontext**
      - Daten im zugewiesenen Adressraum (virtuellen Speicher)  $\implies$  Foliensatz 2
    - **Hardwarekontext** ( $\implies$  Folie 4)
      - Register in der CPU
    - **Systemkontext** ( $\implies$  Folie 5)
      - Informationen, die das Betriebssystem über einen Prozess speichert
  - Die Informationen im Hardwarekontext und Systemkontext verwaltet das Betriebssystem im **Prozesskontrollblock** ( $\implies$  Folie 6)

# Hardwarekontext

- Der **Hardwarekontext** umfasst die Inhalte der Register in der CPU zum Zeitpunkt der Prozess-Ausführung
- Register, deren Inhalt bei einem Prozesswechsel gesichert werden muss:
  - Befehlszähler (*Program Counter, Instruction Pointer*) – enthält die Speicheradresse des nächsten auszuführenden Befehls
  - Stackpointer – enthält die Speicheradresse am Ende des Stacks
  - Basepointer – zeigt auf eine Adresse im Stack
  - Befehlsregister (*Instruction Register*) – speichert den aktuellen Befehl
  - Akkumulator – speichert Operanden für die ALU und deren Resultate
  - Page-table base Register – Adresse wo die Seitentabelle des laufenden Prozesses anfängt
  - Page-table length Register – Länge der Seitentabelle des laufenden Prozesses

Einige dieser Register wurden in Foliensatz 2 vorgestellt

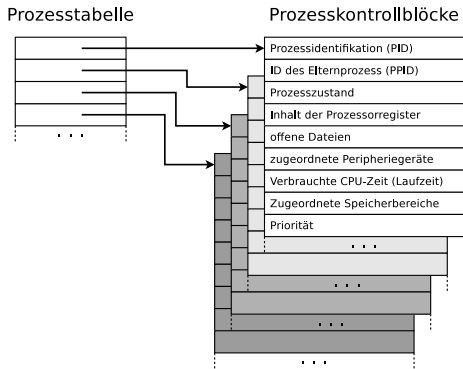
# Systemkontext

- Der **Systemkontext** sind die Informationen, die das Betriebssystem über einen Prozess speichert
- Beispiele:
  - Eintrag in der Prozesstabelle
  - Prozessnummer (PID)
  - Prozesszustand
  - Information über Eltern- oder Kindprozesse
  - Priorität
  - Identifier – Zugriffsrechte auf Ressourcen
  - Quotas – Zur Verfügung stehende Menge der einzelnen Ressourcen
  - Laufzeit
  - Geöffnete Dateien
  - Zugeordnete Geräte

# Prozesstabelle und Prozesskontrollblöcke

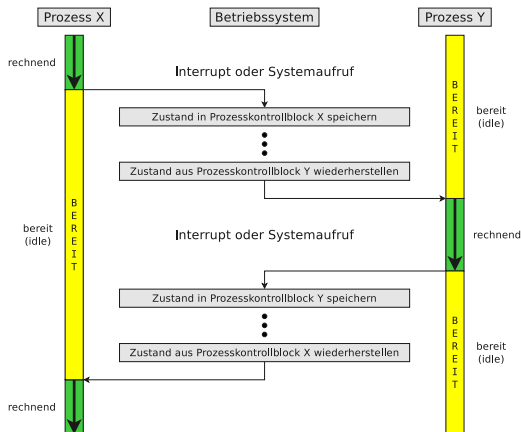
- Jeder Prozess hat seinen eigenen Prozesskontext, der von den Kontexten anderer Prozesse unabhängig ist

- Zur Verwaltung der Prozesse führt das Betriebssystem die **Prozesstabelle**
  - Es ist eine Liste aller existierenden Prozesse
  - Sie enthält für jeden Prozess einen Eintrag, den **Prozesskontrollblock**



# Prozesswechsel

- Beim Prozesswechsel werden der Systemkontext und der Hardwarekontext ( $\Rightarrow$  Inhalt der CPU-Register) im Prozesskontrollblock gespeichert
- Erhält ein Prozess Zugriff auf die CPU, wird sein Kontext mit dem Inhalt des Prozesskontrollblocks wiederhergestellt
- Jeder Prozess ist zu jedem Zeitpunkt in einem bestimmten **Zustand**  $\Rightarrow$  Zustandsdiagramm der Prozesse



# Prozesszustände

Wir wissen bereits...

Jeder Prozess befindet sich zu jedem Zeitpunkt in einem Zustand

- Wie viele unterschiedliche Zustände es gibt, hängt vom Zustands-Prozessmodell des Betriebssystems ab

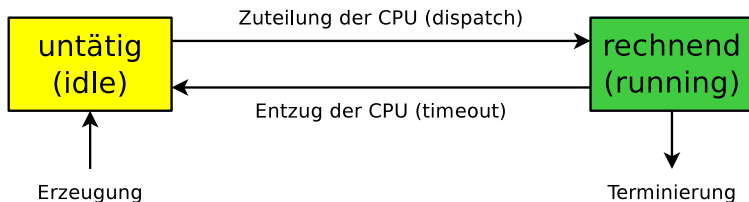
Frage

Wie viele Prozesszustände braucht ein Prozessmodell mindestens?



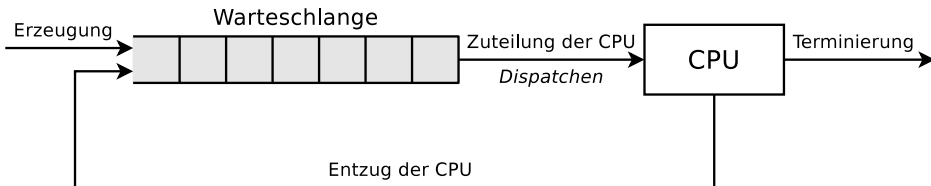
## 2-Zustands-Prozessmodell

- Prinzipiell genügen 2 Prozesszustände
  - **rechnend** (*running*): Einem Prozess wurde die CPU zugeteilt
  - **untätig** (*idle*): Die Prozesse warten auf die Zuteilung der CPU



## 2-Zustands-Prozessmodell (Implementierung)

- Die Prozesse im Zustand **untätig** müssen in einer Warteschlange gespeichert werden, in der sie auf ihre Ausführung warten
  - Die Liste wird nach Prozesspriorität oder Wartezeit sortiert



Die Priorität (anteilige Rechenleistung) hat unter Linux einen Wert von -20 bis +19 (in ganzzahligen Schritten). -20 ist die höchste Priorität und 19 die niedrigste Priorität. Die Standardpriorität ist 0. Normale Nutzer können Prioritäten von 0 bis 19 vergeben. Der Systemverwalter (root) darf auch negative Werte vergeben.

- Dieses Modell zeigt auch die Arbeitsweise des **Dispatchers**
  - Aufgabe des Dispatchers ist die Umsetzung der Zustandsübergänge
- Die Ausführungsreihenfolge der Prozesse legt der **Scheduler** fest, der einen **Scheduling-Algorithmus** (siehe Foliensatz 5) verwendet

# Konzeptioneller Fehler des 2-Zustands-Prozessmodells

- Das 2-Zustands-Prozessmodell geht davon aus, dass alle Prozesse immer zur Ausführung bereit sind
    - Das ist unrealistisch!
  - Es gibt fast immer Prozesse, die **blockiert** sind
    - Mögliche Gründe:
      - Warten auf die Eingabe oder Ausgabe eines E/A-Geräts
      - Warten auf das Ergebnis eines anderen Prozesses
      - Warten auf eine Reaktion des Benutzers
  - Lösung: Die untätigen Prozesse werden in 2 Gruppen unterschieden
    - Prozesse die **bereit** (ready/idle) sind
    - Prozesse die **blockiert** (blocked) sind
- ⇒ 3-Zustands-Prozessmodell

# 3-Zustands-Prozessmodell

- Jeder Prozess befindet sich in einem der folgenden Zustände:

- **rechnend** (*running*):

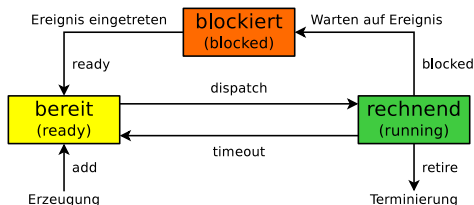
- Der Prozess hat Zugriff auf die CPU und führt auf dieser Instruktionen aus

- **bereit** (*ready/idle*):

- Der Prozess könnte unmittelbar Instruktionen auf der CPU ausführen und wartet aktuell auf die Zuteilung der CPU

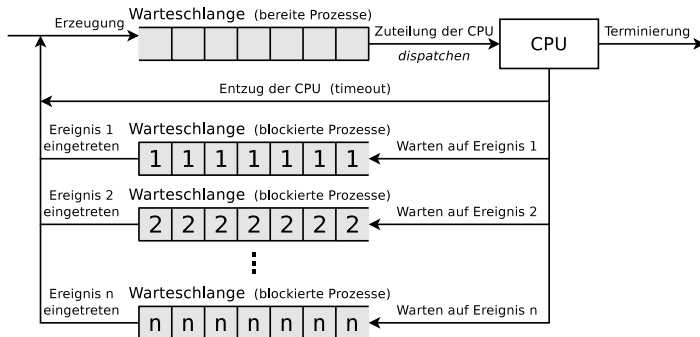
- **blockiert** (*blocked*):

- Der Prozess kann momentan nicht weiter ausgeführt werden und wartet auf das Eintreten eines Ereignisses oder die Erfüllung einer Bedingung
- Dabei kann es sich z.B. um eine Nachricht eines anderen Prozesses oder eines Eingabe-/Ausgabegeräts oder um das Eintreten eines Synchronisationsereignisses handeln



# 3-Zustands-Prozessmodell – Realisierung

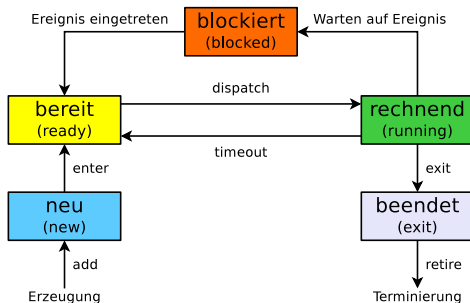
- In der Praxis implementieren Betriebssysteme (z.B. Linux) mehrere Warteschlangen für Prozesse im Zustand **blockiert**



- Beim Zustandsübergang wird der Prozesskontrollblock des Prozesses aus der alten Zustandsliste entfernt und in die neue Zustandsliste eingefügt
- Für Prozesse im Zustand **rechnend** existiert keine eigene Liste

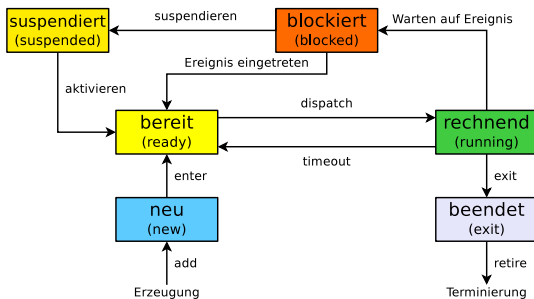
## 5-Zustands-Prozessmodell

- Es ist empfehlenswert, das 3-Zustands-Prozessmodell um 2 weitere Prozesszustände zu erweitern
  - neu** (*new*): Der Prozess (Prozesskontrollblock) ist erzeugt, wurde aber vom Betriebssystem noch nicht in die Warteschlange für Prozesse im Zustand **bereit** eingefügt
  - exit**: Der Prozess ist fertig abgearbeitet oder wurde beendet, aber sein Prozesskontrollblock existiert aus verschiedenen Gründen noch
- Grund für die Existenz der Prozesszustände **neu** und **exit**:
  - Auf manchen Systemen ist die Anzahl der ausführbaren Prozesse limitiert, um Speicher zu sparen und den Grad des Mehrprogrammbetriebs festzulegen



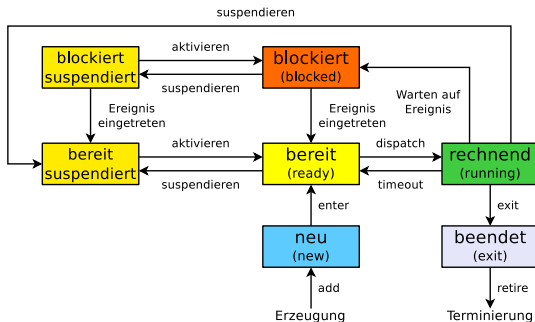
## 6-Zustands-Prozessmodell

- Ist nicht genügend physischer Hauptspeicher für alle Prozesse verfügbar, müssen Teile von Prozessen ausgelagert werden  $\Rightarrow$  **Swapping**
- Das Betriebssystem lagert Prozesse aus, die im Zustand **blockiert** sind
- Dadurch steht mehr Hauptspeicher den Prozessen in den Zuständen **rechnend** und **bereit** zur Verfügung
  - Es macht also Sinn, das 5-Zustands-Prozessmodell um den Prozesszustand **suspendiert** (*suspended*) zu erweitern



## 7-Zustands-Prozessmodell

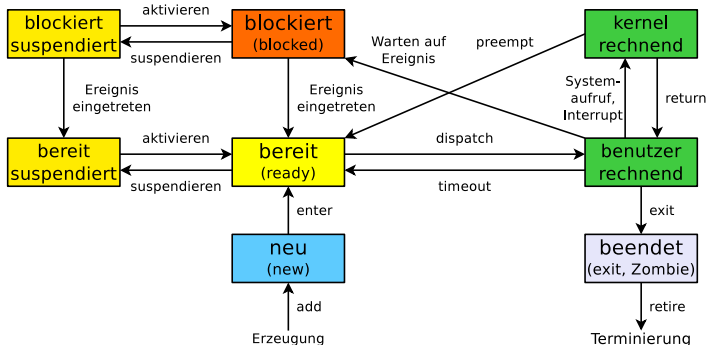
- Wurde ein Prozess suspendiert, ist es besser, den frei gewordenen Platz im Hauptspeicher zu verwenden, um einen ausgelagerten Prozess zu aktivieren, als ihn einem neuen Prozess zuzuweisen
  - Das ist nur dann sinnvoll, wenn der aktivierte Prozess nicht mehr blockiert ist
- Im 6-Zustands-Prozessmodell fehlt die Möglichkeit, die ausgelagerten Prozesse zu unterscheiden in:
  - blockierte ausgelagerte Prozesse
  - nicht-blockierte ausgelagerte Prozesse





# Prozessmodell von Linux/UNIX (etwas vereinfacht)

- Der Zustand **rechnend** (*running*) wird unterteilt in die Zustände...
  - **benutzer rechnend** (*user running*) für Prozesse im Benutzermodus
  - **kernel rechnend** (*kernel running*) für Prozesse im Kernelmodus



Ein **Zombie-Prozess** ist fertig abgearbeitet (via Systemaufruf `exit`), aber sein Eintrag in der Prozesstabelle existiert so lange, bis der Elternprozess den Rückgabewert (via Systemaufruf `wait`) abgefragt hat

# Prozesse unter Linux/UNIX erzeugen mit fork (1/2)

- Der Systemaufruf `fork()` ist die üblicherweise verwendete Möglichkeit, einen neuen Prozess zu erzeugen
- Ruft ein Prozess `fork()` auf, wird eine identische Kopie als neuer Prozess gestartet
  - Der aufrufende Prozess heißt **Vaterprozess** oder **Elternprozess**
  - Der neue Prozess heißt **Kindprozess**
- Der Kindprozess hat nach der Erzeugung den gleichen Programmcode
  - Auch die Befehlszähler haben den gleichen Wert, verweisen also auf die gleiche Zeile im Programmcode
- Geöffnete Dateien und Speicherbereiche des Elternprozesses werden für den Kindprozess kopiert und sind unabhängig vom Elternprozess
  - Kindprozess und Elternprozess besitzen ihren eigenen Prozesskontext

Mit `vfork` existiert eine Variante von `fork`, die nicht den Adressraum des Elternprozesses kopiert, und somit weniger Verwaltungsaufwand als `fork` verursacht. Die Verwendung von `vfork` ist sinnvoll, wenn der Kindprozess direkt nach seiner Erzeugung durch einem anderen Prozess ersetzt werden soll. Für diese Vorlesung ist `vfork` nicht relevant.

## Prozesse unter Linux/UNIX erzeugen mit fork (2/2)

- Ruft ein Prozess `fork()` auf, wird eine exakte Kopie erzeugt
  - Die Prozesse unterscheiden sich nur in den Rückgabewerten von `fork()`

```
1 #include <stdio.h>
2 #include <unistd.h>
3 #include <stdlib.h>
4
5 void main() {
6     int rueckgabewert = fork();
7
8     if (rueckgabewert < 0) {
9         // Hat fork() den Rückgabewert -1, ist ein Fehler aufgetreten.
10        // Speicher oder Prozesstabelle sind voll.
11        ...
12    }
13    if (rueckgabewert > 0) {
14        // Hat fork() einen positiven Rückgabewert, sind wir im Elternprozess.
15        // Der Rückgabewert ist die PID des neu erzeugten Kindprozesses.
16        ...
17    }
18    if (rueckgabewert == 0) {
19        // Hat fork() den Rückgabewert 0, sind wir im Kindprozess.
20        ...
21    }
22 }
```

# Prozessbaum

- Durch das Erzeugen immer neuer Kindprozesse mit `fork()` entsteht ein beliebig tiefer Baum von Prozessen ( $\implies$  Prozesshierarchie)
- Das Kommando `ps tree` gibt die laufenden Prozesse unter Linux/UNIX als Baum entsprechend ihrer Vater-/Sohn-Beziehungen aus

```
$ ps tree
init--Xprt
  |-acpid
  ...
  |-gnome-terminal--4*[bash]
    |
    |   |-bash---su---bash
    |   |-bash+-gv---gs
    |   |
    |   |   |-pstree
    |   |   |-xterm---bash---xterm---bash
    |   |   |-xterm---bash---xterm---bash---xterm---bash
    |   |   `--xterm---bash
    |   |-gnome-pty-helpe
    |   `--{gnome-terminal}
  |-4*[gv---gs]
```

# Informationen über Prozesse unter Linux/UNIX

```
$ ps -eFw
```

UID	PID	PPID	C	SZ	RSS	PSR	STIME	TTY	TIME	CMD
root	1	0	0	51286	7432	2	Apr11	?	00:00:03	/sbin/init
root	1073	1	0	90930	6508	0	Apr11	?	00:00:00	/usr/sbin/lightdm
root	1551	1073	0	60913	6772	2	Apr11	?	00:00:00	lightdm --session-child 14 23
bnc	2143	1551	0	1069	1560	0	Apr11	?	00:00:00	/bin/sh /etc/xdg/xfce4/xinitrc
bnc	2235	2143	0	85195	18888	3	Apr11	?	00:00:11	xfce4-session
bnc	2284	2235	0	110875	45256	3	Apr11	?	00:06:20	xfce4-panel --display :0.0
bnc	2389	2235	0	129173	47904	0	Apr11	?	00:00:26	xfce4-terminal --geometry=80x24
bnc	2471	2389	0	5374	5360	2	Apr11	pts/0	00:00:00	bash
bnc	2487	1	5	316370	395892	0	Apr14	?	00:08:58	/opt/google/chrome/chrome
bnc	2525	2389	0	5895	6620	3	Apr11	pts/5	00:00:00	bash
bnc	3105	2284	0	597319	257520	0	Apr11	?	00:05:22	kate -b
bnc	3122	3105	0	5364	5156	2	Apr11	pts/6	00:00:00	/bin/bash
bnc	11196	2471	0	269491	181048	0	Apr14	pts/0	00:00:25	okular bsrn_vorlesung_04.pdf
bnc	16325	1	0	346638	146872	3	10:31	?	00:00:16	evince BA.pdf
bnc	17384	2525	1	223478	61312	2	10:39	pts/5	00:00:49	dia
bnc	19561	2471	0	9576	3340	0	11:20	pts/0	00:00:00	ps -eFw

- C (CPU) = CPU-Belastung des Prozesses in Prozent
- SZ (Size) = Virtuelle Prozessgröße = Textsegment, Heap und Stack (siehe Folie 31)
- RSS (Resident Set Size) = Belegter physischer Speicher (ohne Swap) in kB
- PSR = Dem Prozess zugewiesener CPU-Kern
- STIME = Startzeitpunkt des Prozesses
- TTY (Teletypewriter) = Steuerterminal. Meist ein virtuelles Gerät: pts (pseudo terminal slave)
- TIME = Bisherige Rechenzeit des Prozesses auf der CPU (HH:MM:SS)

# Unabhängigkeit von Eltern- und Kindprozess

- Das Beispiel zeigt, dass Eltern- und Kindprozess unabhängig voneinander arbeiten und unterschiedliche Speicherbereiche verwenden

```
1 #include <stdio.h>
2 #include <unistd.h>
3 #include <stdlib.h>
4
5 void main() {
6     int i;
7     if (fork())
8         // Hier arbeitet der Vaterprozess
9         for (i = 0; i < 5000000; i++)
10            printf("\n Vater: %i", i);
11     else
12         // Hier arbeitet der Kindprozess
13         for (i = 0; i < 5000000; i++)
14            printf("\n Kind : %i", i);
15 }
```

```
Kind : 0
Kind : 1
...
Kind : 21019
Vater: 0
...
Vater: 50148
Kind : 21020
...
Kind : 129645
Vater: 50149
...
Vater: 855006
Kind : 129646
...
```

- In der Ausgabe sind die Prozesswechsel zu sehen
- Der Wert der Schleifenvariablen *i* beweist, dass Eltern- und Kindprozess unabhängig voneinander sind
  - Das Ergebnis der Ausführung ist nicht reproduzierbar

Ausführung auf nur einem CPU-Kern...

```
$ taskset --cpu-list 1 ./fork_beispiel2
```

# Die PID-Nummern von Eltern- und Kindprozess (1/2)

```
1 #include <stdio.h>
2 #include <unistd.h>
3 #include <stdlib.h>
4
5 void main() {
6     int pid_des_Kindes;
7
8     pid_des_Kindes = fork();
9
10    // Es kam zu einem Fehler --> Programmabbruch
11    if (pid_des_Kindes < 0) {
12        perror("\n Es kam bei fork() zu einem Fehler!");
13        exit(1);
14    }
15
16    // Vaterprozess
17    if (pid_des_Kindes > 0) {
18        printf("\n Vater: PID: %i", getpid());
19        printf("\n Vater: PPID: %i", getppid());
20    }
21
22    // Kindprozess
23    if (pid_des_Kindes == 0) {
24        printf("\n Kind:  PID: %i", getpid());
25        printf("\n Kind:  PPID: %i", getppid());
26    }
27 }
```

- Das Beispiel erzeugt einen Kindprozess
- Kindprozess und Vaterprozess geben beide aus:
  - Eigene PID
  - PID des Vaters (PPID)

## Die PID-Nummern von Eltern- und Kindprozess (2/2)

- Die Ausgabe ist üblicherweise mit dieser vergleichbar:

```
Vater: PID: 20952
Vater: PPID: 3904
Kind: PID: 20953
Kind: PPID: 20952
```

- Gelegentlich kann man folgendes Ereignis beobachten:

```
Vater: PID: 20954
Vater: PPID: 3904
Kind: PID: 20955
Kind: PPID: 1
```

- Der Elternprozess wurde vor dem Kind-Prozess beendet
  - Wird der Elternprozess vor dem Kindprozess beendet, bekommt er `init` als neuen Elternprozess zugeordnet
  - Elternlose Prozesse werden immer von `init` adoptiert

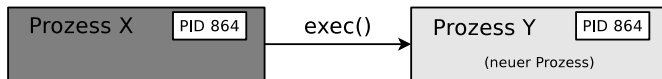
**init (PID 1) ist der erste Prozess unter Linux/UNIX**

Alle laufenden Prozesse stammen von `init` ab  $\implies$  `init` = Vater aller Prozesse



# Prozesse ersetzen mit exec

- Der Systemaufruf `exec()` ersetzt einen Prozess durch einen anderen
  - Es findet eine **Verkettung** statt
  - Der neue Prozess erbt die PID des aufrufenden Prozesses
- Will man aus einem Prozess heraus ein Programm starten, ist es nötig, zuerst mit `fork()` einen neuen Prozess zu erzeugen und diesen mit `exec()` zu ersetzen
  - Wird vor einem Aufruf von `exec()` kein neuer Prozess mit `fork()` erzeugt, geht der Elternprozess verloren
- Schritte einer Programmausführung in der Shell:
  - Die Shell erzeugt mit `fork()` eine identische Kopie von sich selbst
  - Im neuen Prozess wird mit `exec()` das eigentliche Programm gestartet



## Beispiel zum Systemaufruf exec

```
$ ps -f
UID          PID    PPID  C STIME TTY          TIME CMD
user         1772    1727  0 May18 pts/2        00:00:00 bash
user        12750    1772  0 11:26 pts/2        00:00:00 ps -f

$ bash

$ ps -f
UID          PID    PPID  C STIME TTY          TIME CMD
user         1772    1727  0 May18 pts/2        00:00:00 bash
user        12751    1772 12 11:26 pts/2        00:00:00 bash
user        12769   12751  0 11:26 pts/2        00:00:00 ps -f

$ exec ps -f
UID          PID    PPID  C STIME TTY          TIME CMD
user         1772    1727  0 May18 pts/2        00:00:00 bash
user        12751    1772  4 11:26 pts/2        00:00:00 ps -f

$ ps -f
UID          PID    PPID  C STIME TTY          TIME CMD
user         1772    1727  0 May18 pts/2        00:00:00 bash
user        12770    1772  0 11:27 pts/2        00:00:00 ps -f
```

- Durch das exec hat ps -f die Bash ersetzt und deren PID (12751) und PPID (1772) übernommen

# Ein weiteres Beispiel zu exec

```
1 #include <stdio.h>
2 #include <unistd.h>
3
4 int main() {
5     int pid;
6     pid = fork();
7
8     // Wenn die PID!=0 --> Elternprozess
9     if (pid) {
10         printf("...Elternprozess...\n");
11         printf("[Eltern] Eigene PID:      %d\n", getpid());
12         printf("[Eltern] PID des Kindes: %d\n", pid);
13     }
14     // Wenn die PID=0 --> Kindprozess
15     else {
16         printf("...Kindprozess...\n");
17         printf("[Kind]   Eigene PID:      %d\n", getpid());
18         printf("[Kind]   PID des Vaters: %d\n", getppid());
19
20         // Aktuelles Programm durch "date" ersetzen
21         // "date" wird der Prozessname in der Prozesstabelle
22         execl("/bin/date", "date", "-u", NULL);
23     }
24     printf("[%d ]Programmende\n", getpid());
25     return 0;
26 }
```

- Der Systemruf `exec()` existiert nicht als Bibliotheks-funktion
- Aber es existieren mehrere Varianten der Funktion `exec()`
- Eine Variante ist `execl()`

Hilfreiche Übersicht über die verschiedene Varianten der Funktion `exec()`

<http://www.cs.uregina.ca/Links/class-info/330/Fork/fork.html>

# Erklärung zum exec Beispiel

```
$ ./exec_beispiel
...Elternprozess...
[Eltern] Eigene PID:      25492
[Eltern] PID des Kindes:  25493
[25492 ]Programmende
...Kindprozess...
[Kind]   Eigene PID:      25493
[Kind]   PID des Vaters:  25492
Di 24. Mai 17:16:48 CEST 2016
```

```
$ ./exec_beispiel
...Elternprozess...
[Eltern] Eigene PID:      25499
[Eltern] PID des Kindes:  25500
[25499 ]Programmende
...Kindprozess...
[Kind]   Eigene PID:      25500
[Kind]   PID des Vaters:  1
Di 24. Mai 17:17:15 CEST 2016
```

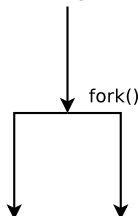
- Der Kindprozess wird nach der Ausgabe seiner PID mit `getpid()` und der PID seines Elternprozesses mit `getppid()` durch `date` ersetzt
- Wird der Elternprozess vor dem Kindprozess beendet, bekommt der Kindprozess `init` als neuen Elternprozess zugeordnet

Seit Linux Kernel 3.4 (2012) und Dragonfly BSD 4.2 (2015) können auch andere Prozesse als `PID=1` neue Elternprozesse eines verweisten Kindprozesses werden  
<http://unix.stackexchange.com/questions/149319/new-parent-process-when-the-parent-process-dies/177361#177361>

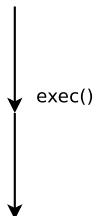
## 3 Möglichkeiten um einen neuen Prozess zu erzeugen

- **Prozessvergabelung** (*forking*): Ein laufender Prozess erzeugt mit `fork()` einen neuen, identischen Prozess
- **Prozessverkettung** (*chaining*): Ein laufender Prozess erzeugt mit `exec()` einen neuen Prozess und beendet (terminiert) sich damit selbst, weil er durch den neuen Prozess ersetzt wird
- **Prozesserzeugung** (*creation*): Ein laufender Prozess erzeugt mit `fork()` einen neuen, identischen Prozess, der sich selbst mit `exec()` durch einen neuen Prozess ersetzt

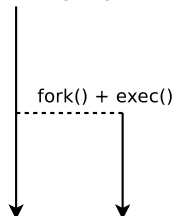
Prozessvergabelung



Prozessverkettung



Prozesserzeugung



# Spaß haben mit Forkbomben

- Eine Forkbombe ist ein Programm, das den Systemaufruf `fork` in einer Endlosschleife aufruft
- Ziel: So lange Kopien des Prozesses erzeugen, bis kein Speicher mehr frei ist
  - Das System wird unbenutzbar

## Forkbombe in Python

```
1 import os
2
3 while True:
4     os.fork()
```

## Forkbombe in C

```
1 #include <unistd.h>
2
3 int main(void)
4 {
5     while(1)
6         fork();
7 }
```

## Forkbombe in PHP

```
1 <?php
2 while(true)
3     pcntl_fork();
4 ?>
```

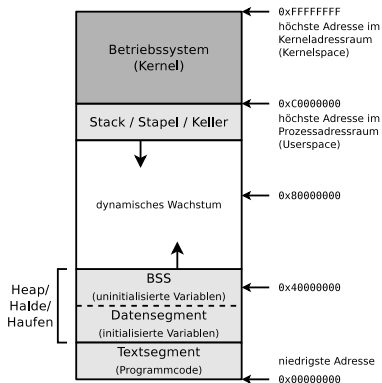
- Einzige Schutzmöglichkeit: Maximale Anzahl der Prozesse und maximalen Speicherverbrauch pro Benutzer limitieren

# Struktur eines UNIX-Prozesses im Speicher (1/6)

- Standardmäßige Aufteilung des virtuellen Speichers auf einem Linux-System mit 32-Bit-CPU
  - 1 GB sind für das System (Kernel)
  - 3 GB für den laufenden Prozess

Die Struktur von Prozessen auf 64 Bit-Systemen unterscheidet sich nicht von 32 Bit-Systemen. Einzig der Adressraum ist größer und damit die mögliche Ausdehnung der Prozesse im Speicher

- Das **Textsegment** enthält den Programmcode (Maschinencode)
- Können mehrere Prozessen teilen
  - Muss also nur einmal im physischen Speicher vorgehalten werden
  - Ist darum üblicherweise nur lesbar (*read only*)
- Liest `exec()` aus der Programmdatei

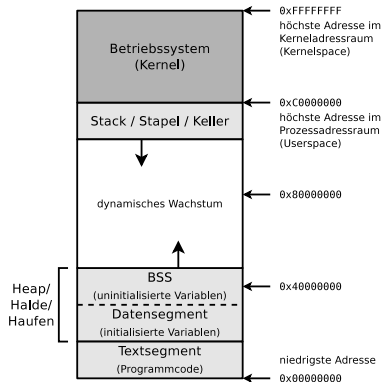


## Quellen

UNIX-Systemprogrammierung, *Helmut Herold*, Addison-Wesley (1996), S.345-347  
 Betriebssysteme, *Carsten Vogt*, Spektrum (2001), S.58-60  
 Moderne Betriebssysteme, *Andrew S. Tanenbaum*, Pearson (2009), S.874-877

# Struktur eines UNIX-Prozesses im Speicher (2/6)

- Der **Heap** wächst dynamisch und besteht aus 2 Teilen:
  - 1 **Datensegment**
  - 2 **BSS**
- Das **Datensegment** enthält **initialisierte** Variablen und Konstanten
  - Enthält alle Daten, die ihre Werte in globalen Deklarationen (außerhalb von Funktionen) zugewiesen bekommen
    - Beispiel: `int summe = 0;`
  - Liest `exec()` aus der Programmdatei



## Quellen

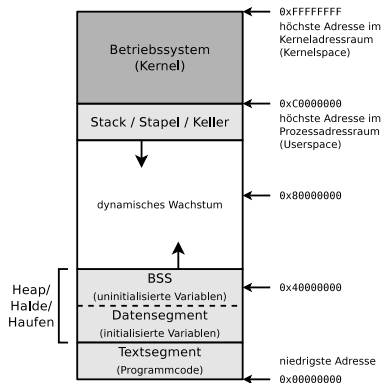
UNIX-Systemprogrammierung, *Helmut Herold*, Addison-Wesley (1996), S.345-347  
Betriebssysteme, *Carsten Vogt*, Spektrum (2001), S.58-60  
Moderne Betriebssysteme, *Andrew S. Tanenbaum*, Pearson (2009), S.874-877

Der Userspace in der dargestellten Speicherstruktur der Prozesse entspricht dem Benutzerkontext (siehe Folie 3). Das ist der vom Betriebssystem zugewiesene virtuelle Adressraum (virtuelle Speicher)  $\Rightarrow$  siehe Foliensatz 2



# Struktur eines UNIX-Prozesses im Speicher (3/6)

- Der Bereich **BSS** (*Block Started by Symbol*) enthält **nicht initialisierte** Variablen
- Enthält globale Variablen (Deklaration ist außerhalb von Funktionen), denen kein Anfangswert zugewiesen wird
  - Beispiel: `int i;`
- Zudem kann hier der Prozess dynamisch zur Laufzeit Speicher allokieren
  - Unter C mit der Funktion `malloc()`
- Alle Variablen im BSS initialisiert `exec()` mit 0

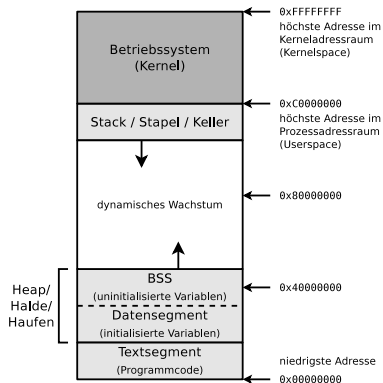


## Quellen

UNIX-Systemprogrammierung, *Helmut Herold*, Addison-Wesley (1996), S.345-347  
Betriebssysteme, *Carsten Vogt*, Spektrum (2001), S.58-60  
Moderne Betriebssysteme, *Andrew S. Tanenbaum*, Pearson (2009), S.874-877

# Struktur eines UNIX-Prozesses im Speicher (4/6)

- Der **Stack** dient zur Realisierung geschachtelter Funktionsaufrufe
  - Enthält auch die Kommandozeilenargumente des Programmaufrufs und Umgebungsvariablen
- Arbeitet nach dem Prinzip LIFO (Last In First Out)

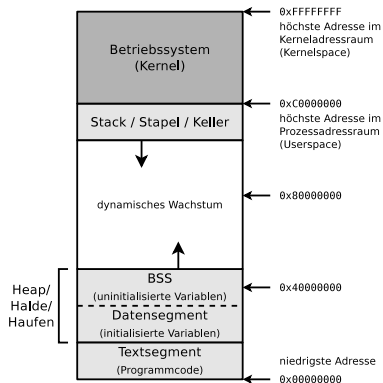


## Quellen

UNIX-Systemprogrammierung, *Helmut Herold*, Addison-Wesley (1996), S.345-347  
 Betriebssysteme, *Carsten Vogt*, Spektrum (2001), S.58-60  
 Moderne Betriebssysteme, *Andrew S. Tanenbaum*, Pearson (2009), S.874-877

# Struktur eines UNIX-Prozesses im Speicher (5/6)

- Mit jedem Funktionsaufruf wird eine Datenstruktur mit folgendem Inhalt auf den Stack gelegt:
  - Aufrufparameter
  - Rücksprungsadresse
  - Zeiger auf die aufrufende Funktion im Stack
- Die Funktionen legen auch ihre lokalen Variablen auf den Stack
- Beim Rücksprung aus einer Funktion wird die Datenstruktur der Funktion aus dem Stack entfernt



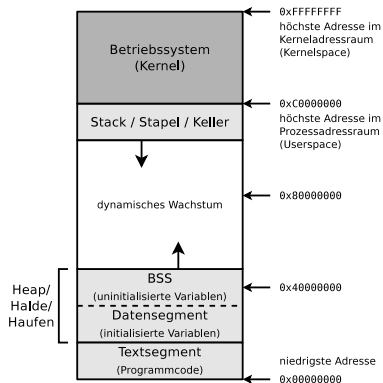
## Quellen

UNIX-Systemprogrammierung, *Helmut Herold*, Addison-Wesley (1996), S.345-347  
 Betriebssysteme, *Carsten Vogt*, Spektrum (2001), S.58-60  
 Moderne Betriebssysteme, *Andrew S. Tanenbaum*, Pearson (2009), S.874-877

# Struktur eines UNIX-Prozesses im Speicher (6/6)

- Das Kommando `size` gibt die Größe (in Bytes) von Textsegment, Datensegment und BSS von Programmdateien aus
  - Die Inhalte von Textsegment und Datensegment sind in den Programmdateien enthalten
  - Alle Inhalte im BSS werden bei der Prozesserzeugung auf den Wert 0 gesetzt

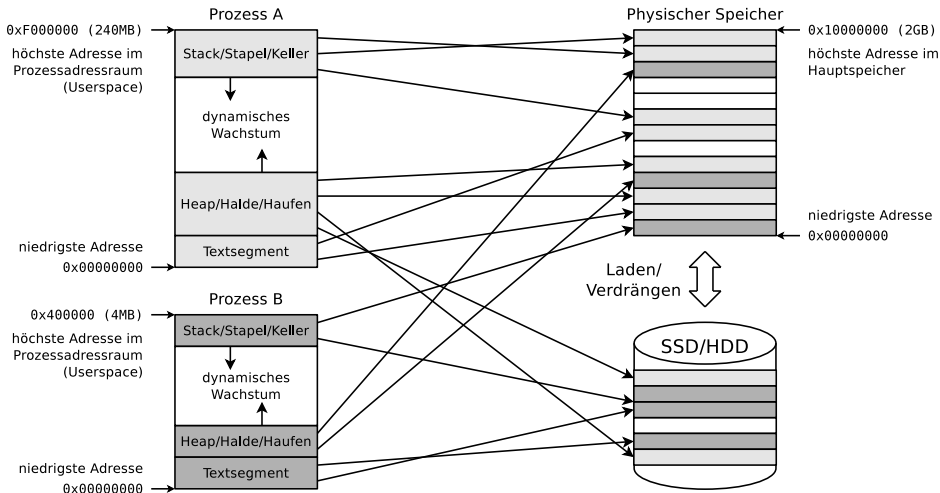
\$	size	/bin/c*				
	text	data	bss	dec	hex	filename
	46480	620	1480	48580	bdc4	/bin/cat
	7619	420	32	8071	1f87	/bin/chac1
	55211	592	464	56267	dbcb	/bin/chgrp
	51614	568	464	52646	cda6	/bin/chmod
	57349	600	464	58413	e42d	/bin/chown
	120319	868	2696	123883	1e3eb	/bin/cp
	131911	2672	1736	136319	2147f	/bin/cpio



## Quellen

UNIX-Systemprogrammierung, *Helmut Herold*, Addison-Wesley (1996), S.345-347  
Betriebssysteme, *Carsten Vogt*, Spektrum (2001), S.58-60  
Moderne Betriebssysteme, *Andrew S. Tanenbaum*, Pearson (2009), S.874-877

# Wiederholung: Virtueller Speicher (Foliensatz 2)

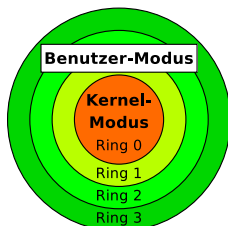


Quelle: [http://cseweb.ucsd.edu/classes/wi11/cse141/Slides/19\\_VirtualMemory.key.pdf](http://cseweb.ucsd.edu/classes/wi11/cse141/Slides/19_VirtualMemory.key.pdf)

Die Ablage der Prozesse im (hier 2 GB großen) physischen Speicher erfolgt durch den virtuellen Speicher nicht in fortlaufender Weise und auch nicht zwangsläufig ständig im Hauptspeicher

# Benutzermodus und Kernelmodus

- x86-kompatible CPUs enthalten 4 Privilegienstufen
  - Ziel: Stabilität und Sicherheit verbessern
  - Jeder Prozess wird in einem Ring ausgeführt und kann sich nicht selbstständig aus diesem befreien



## Realisierung der Privilegienstufen

- Das Register CPL (Current Privilege Level) speichert die aktuelle Privilegienstufe
- Quelle: Intel 80386 Programmer's Reference Manual 1986  
<http://css.csail.mit.edu/6.858/2012/readings/i386.pdf>

- In Ring 0 (= **Kernelmodus**) läuft der Betriebssystemkern
  - Hier haben Prozesse vollen Zugriff auf die Hardware
  - Der Kern kann auch physischen Speicher adressieren ( $\implies$  Real Mode)
- In Ring 3 (= **Benutzermodus**) laufen die Anwendungen
  - Hier arbeiten Prozesse nur mit virtuellem Speicher ( $\implies$  Protected Mode)

## Moderne Betriebssysteme verwenden nur 2 Privilegienstufen (Ringe)

Grund: Einige Hardware-Architekturen (z.B: Alpha, PowerPC, MIPS) enthalten nur 2 Stufen

# Systemaufrufe (1/2)

Wir wissen bereits...

Alle Prozesse außerhalb des Betriebssystemkerns dürfen ausschließlich auf ihren eigenen virtuellen Speicher zugreifen

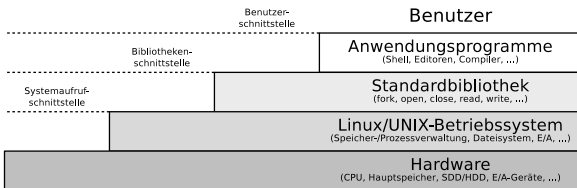
- Muss ein Prozess im Benutzermodus eine höher privilegierte Aufgabe erfüllen (z.B. Zugriff auf Hardware), kann er das dem Kernel durch einen **Systemaufruf** mitteilen
  - Ein Systemaufruf ist ein Funktionsaufruf im Betriebssystem, der einen Sprung vom Benutzermodus in den Kernelmodus auslöst (⇒ **Moduswechsel**)

## Moduswechsel

- Ein Prozess gibt die Kontrolle über die CPU an den Kernel ab und ist unterbrochen bis die Anfrage fertig bearbeitet ist
  - Nach dem Systemaufruf gibt der Kernel die CPU wieder an den Prozess im Benutzermodus ab
  - Der Prozess führt seine Abarbeitung an der Stelle fort, an der der Prozesswechsel zuvor angefordert wurde
- 
- Die Leistung eines Systemaufrufs wird im Kernel erbracht
    - Also außerhalb des Adressraums des aufrufenden Prozesses

# Systemaufrufe (2/2)

- **Systemaufrufe** (System Calls) sind die Schnittstelle, die das Betriebssystem den Prozessen im Benutzermodus zur Verfügung stellt
  - Systemaufrufe erlauben den Prozessen im Benutzermodus u.a. Prozesse und Dateien zu erzeugen und zu verwalten und auf Hardware zuzugreifen

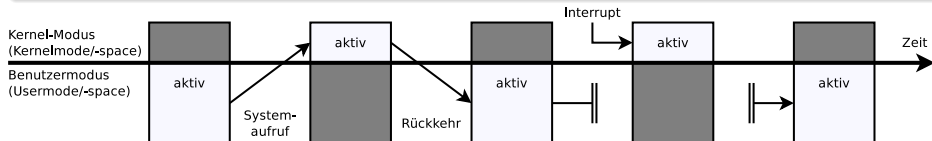


Einfach gesagt...

Ein Systemaufruf ist eine Anfrage eines Prozesses im Benutzermodus an den Kernel, um einen Dienst des Kernels zu nutzen

## Vergleich zwischen Systemaufrufen und Interrupts

Interrupts sind Unterbrechungen, die Ereignisse außerhalb von Prozessen im Benutzermodus auslösen





# Ein Beispiel für einen Systemaufruf: `ioctl()`

- Mit `ioctl()` setzen Linux-Programme gerätespezifische Befehle ab
  - Er ermöglicht Prozessen die Kommunikation mit und Steuerung von:
    - Zeichenorientierten Geräten (Maus, Tastatur, Drucker, Terminals, ...)
    - Blockorientierten Geräten (SSD/HDD, CD-/DVD-Laufwerk, ...)
- Syntax:

```
ioctl (Filedeskriptor, Aktionsanforderung, Integer-Wert oder Zeiger auf Daten);
```

- Einige typische Einsatzszenarien von `ioctl()`:
  - Diskettenspur formatieren
  - Modem oder Soundkarte initialisieren
  - CD auswerfen
  - Status- und Verbindungsinformationen der WLAN-Schnittstelle auslesen
  - Auf Sensoren via Inter-Integrated Circuit (I<sup>2</sup>C) Datenbus zugreifen

## Gute Übersichten über Systemaufrufe

Linux: <http://www.digilife.be/quickreferences/qrc/linux%20system%20call%20quick%20reference.pdf>

Linux: <http://syscalls.kernelgrok.com>

Linux: [http://www.tutorialspoint.com/unix\\_system\\_calls/](http://www.tutorialspoint.com/unix_system_calls/)

Windows: <http://j00ru.vexillum.org/ntapi>

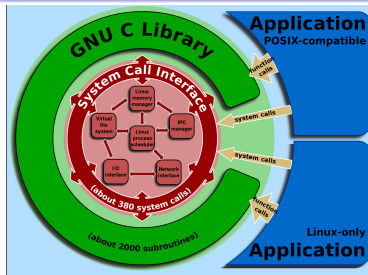
# Systemaufrufe und Bibliotheken

- Direkt mit Systemaufrufen arbeiten ist **unsicher** und **schlecht portabel**
- Moderne Betriebssysteme enthalten eine Bibliothek, die sich logisch zwischen den Benutzerprozessen und dem Kern befindet

## Beispiele für solche Bibliotheken

GNU C-Bibliothek glibc (Linux), C Standard Library (UNIX), C Library Implementation (BSD), Native API ntdll.dll (Windows)

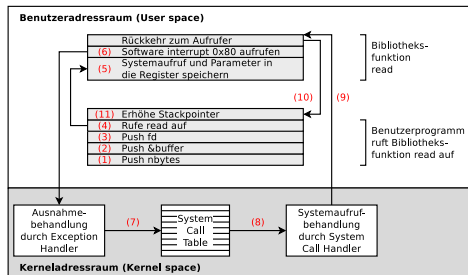
- Die Bibliothek ist zuständig für:
  - Kommunikationsvermittlung der Benutzerprozesse mit dem Kernel
  - Moduswechsel zwischen Benutzermodus und Kernelmodus
- Vorteile, die der Einsatz einer Bibliothek mit sich bringt:
  - Erhöhte **Portabilität**, da kein oder nur sehr wenig Bedarf besteht, dass die Benutzerprozesse direkt mit dem Kernel kommunizieren
  - Erhöhte **Sicherheit**, da die Benutzerprozesse nicht selbst den Wechsel in den Kernelmodus durchführen können



Bildquelle: Wikipedia  
(Shmuel Csaba Otto Traian, CC-BY-SA-3.0)

# Schritt für Schritt (1/4) – read(fd, buffer, nbytes);

- In Schritt **1-3** legt der Benutzerprozess die Parameter auf den Stack
- In **4** ruft der Benutzerprozess die **Bibliotheksfunktion** für read (⇒ nbytes aus der Datei fd lesen und in buffer speichern) auf
- Die Bibliotheksfunktion speichert in **5** die Nummer des Systemaufrufs im *Accumulator Register* EAX (32-Bit) bzw. RAX (64-Bit)
  - Die Bibliotheksfunktion speichert die Parameter des Systemaufrufs in den Registern EBX, ECX und EDX (bzw. bei 64-Bit: RBX, RCX und RDX)

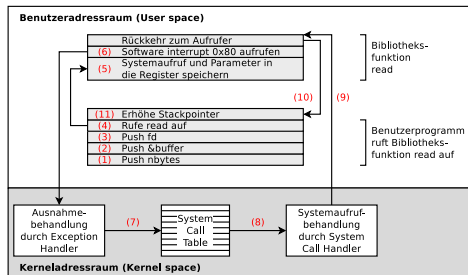


Quelle dieses Beispiels

Moderne Betriebssysteme, Andrew S. Tanenbaum, 3.Auflage, Pearson (2009), S.84-89

# Schritt für Schritt (2/4) – read(fd, buffer, nbytes);

- In **6** wird der Softwareinterrupt (Exception) 0x80 (dezimal: 128) ausgelöst, um vom Benutzermodus in den Kernelmodus zu wechseln
  - Der Softwareinterrupt unterbricht die Programmausführung im Benutzermodus und erzwingt das Ausführen eines Exception-Handlers im Kernelmodus

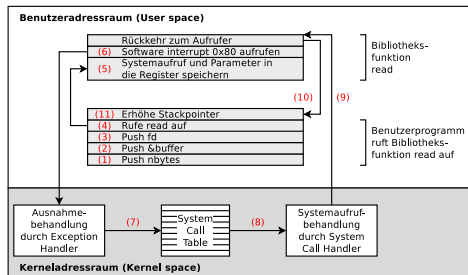


Der Kernel verwaltet die *System Call Table*, eine Liste mit allen Systemaufrufen

Jedem Systemaufruf ist dort eine eindeutige Nummer und eine Kernel-interne Funktion zugeordnet

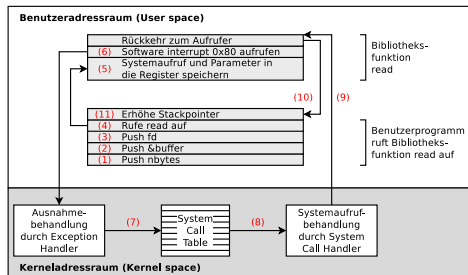
# Schritt für Schritt (3/4) – read(fd, buffer, nbytes);

- Der aufgerufene Exception-Handler ist eine Funktion im Kernel, die das Register EAX (bzw. RAX) ausliest
- Die Exception-Handler-Funktion ruft in **7** die entsprechende Kernel-Funktion aus der System Call Table mit den in den Registern EBX, ECX und EDX (bzw. RBX, RCX und RDX) liegenden Argumenten auf
- In **8** startet der Systemaufruf



# Schritt für Schritt (4/4) – read(fd, buffer, nbytes);

- In **9** gibt der Exception-Handler die Kontrolle an die Bibliothek zurück, die den Softwareinterrupt auslöste
- Die Funktion kehrt danach in **10** zum Benutzerprozess so zurück, wie es auch eine normale Funktion getan hätte
- Um den Systemaufruf abzuschließen, muss der Benutzerprozess in **11** genau wie nach jedem Funktionsaufruf den Stack aufräumen
- Anschließend kann der Benutzerprozess weiterarbeiten



# Beispiel für einen Systemaufruf unter Linux

- Systemaufrufe werden wie Bibliotheksfunktionen aufgerufen
  - Der Mechanismus ist bei allen Betriebssystemen ähnlich
  - In einem C-Programm ist kein Unterschied erkennbar

```
1 #include <syscall.h>
2 #include <unistd.h>
3 #include <stdio.h>
4 #include <sys/types.h>
5
6 int main(void) {
7     unsigned int ID1, ID2;
8
9     // Systemaufruf
10    ID1 = syscall(SYS_getpid);
11    printf ("Ergebnis des Systemaufrufs: %d\n", ID1);
12
13    // Von der glibc aufgerufener Systemaufruf
14    ID2 = getpid();
15    printf ("Ergebnis der Bibliotheksfunktion: %d\n", ID2);
16
17    return(0);
18 }
```

```
$ gcc SysCallBeispiel.c -o SysCallBeispiel
$ ./SysCallBeispiel
Ergebnis des Systemaufrufs: 3452
Ergebnis der Bibliotheksfunktion: 3452
```

# Auswahl an Systemaufrufen

## Prozess- verwaltung

fork	Neuen Kindprozess erzeugen
waitpid	Auf Beendigung eines Kindprozesses warten
execve	Einen Prozess durch einen anderen ersetzen. PID beibehalten
exit	Prozess beenden

## Datei- verwaltung

open	Datei zum Lesen/Schreiben öffnen
close	Offene Datei schließen
read	Daten aus einer Datei in den Puffer einlesen
write	Daten aus dem Puffer in eine Datei schreiben
lseek	Dateipositionszeiger positionieren
stat	Status einer Datei ermitteln

## Verzeichnis- verwaltung

mkdir	Neues Verzeichnis erzeugen
rmdir	Leeres Verzeichnis entfernen
link	Neuen Verzeichniseintrag (Link) auf eine Datei erzeugen
unlink	Verzeichniseintrag löschen
mount	Dateisystem in die hierarchische Verzeichnisstruktur einhängen
umount	Eingehängtes Dateisystem aushängen

## Verschiedenes

chdir	Aktuelles Verzeichnis wechseln
chmod	Dateirechte für eine Datei ändern
kill	Signal an einen Prozess schicken
time	Sekunden seit dem 1. Januar 1970 („Unixzeit“) ausgeben



# Systemaufrufe unter Linux

- Die Liste mit den Namen der Systemaufrufe im Linux-Kernel...
  - befindet sich im Quelltext von Kernel 2.6.x in der Datei:  
arch/x86/kernel/syscall\_table\_32.S
  - befindet sich im Quelltext von Kernel 3.x, 4.x und 5.x in diesen Dateien:  
arch/x86/syscalls/syscall\_[64|32].tbl oder  
arch/x86/entry/syscalls/syscall\_[64|32].tbl

```
arch/x86/syscalls/syscall_32.tbl
```

```
...
1      i386   exit      sys_exit
2      i386   fork      sys_fork
3      i386   read      sys_read
4      i386   write     sys_write
5      i386   open      sys_open
6      i386   close     sys_close
...
```

## Anleitungen, wie man eigene Systemaufrufe realisiert

```
https://www.kernel.org/doc/html/v4.14/process/adding-syscalls.html
https://brennan.io/2016/11/14/kernel-dev-ep3/
https://medium.com/@jeremyphilemon/adding-a-quick-system-call-to-the-linux-kernel-cad55b421a7b
https://medium.com/@ssreehari/implementing-a-system-call-in-linux-kernel-4-7-1-6f98250a8c38
http://tldp.org/HOWTO/Implement-Sys-Call-Linux-2.6-i386/index.html
http://www.ibm.com/developerworks/library/l-system-calls/
```