# Vorlesung Betriebssysteme I

## Thema 5: Aktivitäten

Robert Baumgartl

24. November 2015

# Prozesse

**Def.** Ein Prozess ist ein in Ausführung befindliches Programm.

- Lebenszyklus: Erzeugung $\rightarrow$ Abarbeitung $\rightarrow$ Beendigung
- benötigt Ressourcen bei Erzeugung (Hauptspeicher, eineindeutigen Identifikator PID, Programmcode)
- benötigt weitere Ressourcen im Laufe seines Lebens, nicht mehr benötigte Ressourcen gibt er i. a. zurück
- Jeder Prozess besitzt einen virtuellen Prozessor, d. h. CPU wird zwischen allen Prozessen geteilt (jeder erhält CPU für eine gewisse Zeitspanne, vgl. folgende Abbildung)
- Hauptmerkmal: Jeder Prozess besitzt einen *eigenen* Adressraum (jeder Prozess denkt gewissermaßen, er sei allein im System)
- Jeder Prozess besitzt einen Vaterprozess sowie u. U. Kindprozesse

# Virtuelle vs. reale CPU

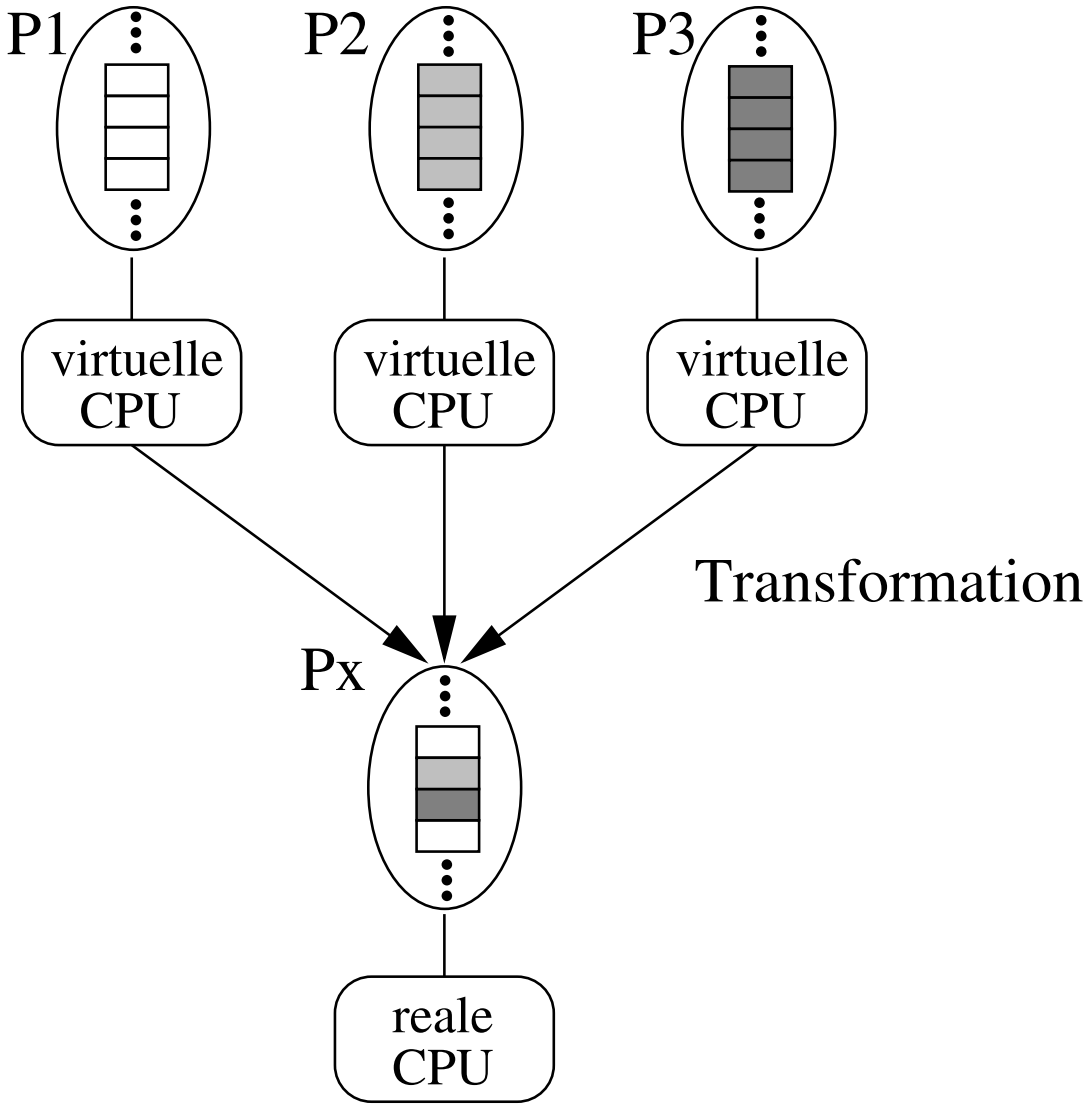



Abbildung: Virtuelle vs. reale CPU

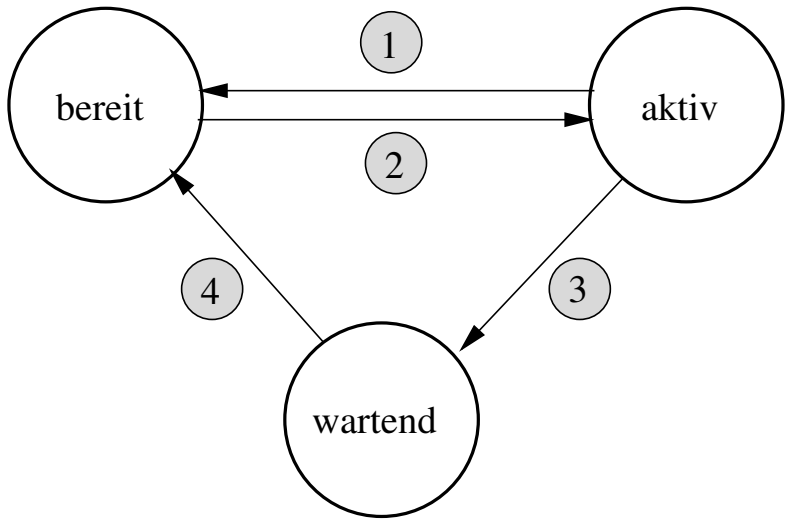
# Zustandsmodell eines Prozesses

Drei grundlegende *Global*zustände werden stets unterschieden:

**aktiv** : Prozess wird abgearbeitet. Er besitzt alle angeforderten Ressourcen und die CPU.

**bereit** : Prozess besitzt alle angeforderten Ressourcen jedoch *nicht* die CPU.

**wartend** : Prozess wartet auf Zuteilung einer durch ihn angeforderten Ressource und wird nicht abgearbeitet.



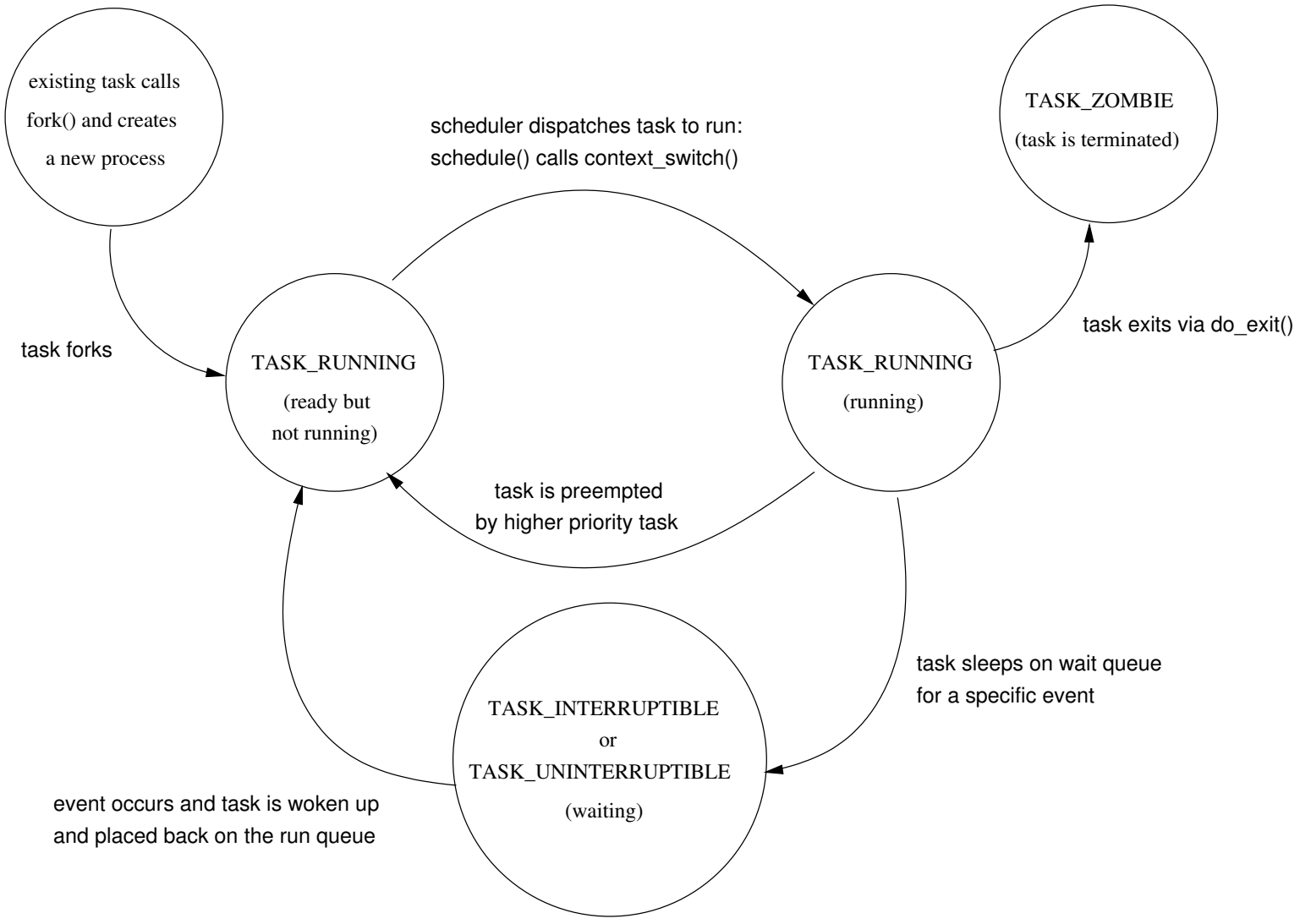
# Zustandsübergänge (Transitionen) bei Prozessen

1. aktiv $\rightarrow$ bereit: Aktiver Prozess wird *verdrängt* (Ursache z. B. höherpriorisierter Prozess wurde bereit oder Zeitscheibe abgelaufen)
2. bereit $\rightarrow$ aktiv: wie 1.
3. aktiv $\rightarrow$ wartend: Aktiver Prozess geht in Wartezustand (er hat eine Ressource angefordert, deren Zuteilung ihm verweigert wurde; er blockiert)
4. wartend $\rightarrow$ bereit: wartender Prozess erhält angeforderte Ressource schließlich zugeteilt.

# Zustandsübergänge cont'd

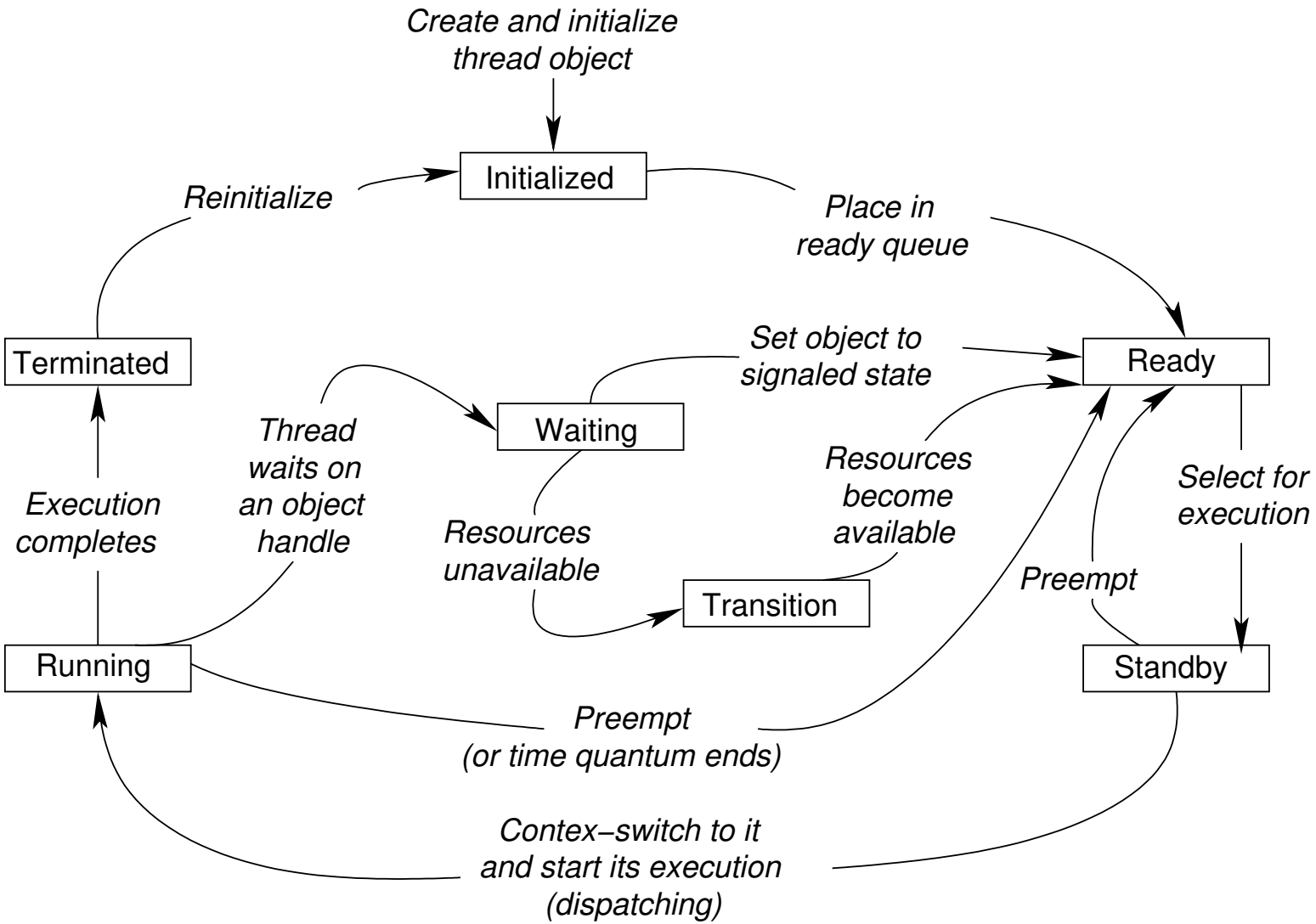
- bereit $\rightarrow$ wartend: unmöglich (ein bereiter Prozess kann nichts tun, also auch keine Ressource anfordern, die ihm verweigert wird)
- wartend $\rightarrow$ aktiv: nicht sinnvoll (Prozess erhält eine Ressource, auf die er wartet, rückgebender aktiver Prozess würde für Ressourcenrückgabe „bestraft“)
- Es gibt stets einen aktiven Prozess (CPU kann nicht „leerlaufen“), falls keine Nutzarbeit anliegt $\rightsquigarrow$ Idle-Prozess
- Jede Ressourcenanforderung wird irgendwann erfüllt.
- Prozesszustandsdiagramme in realen Systemen sehen häufig komplexer aus (sind es aber nicht).

# Prozesszustände im Linux-Kernel 2.6




Quelle: Robert Love, Linux Kernel Development, 2005

# Prozesszustände im Windows NT/2000/XP

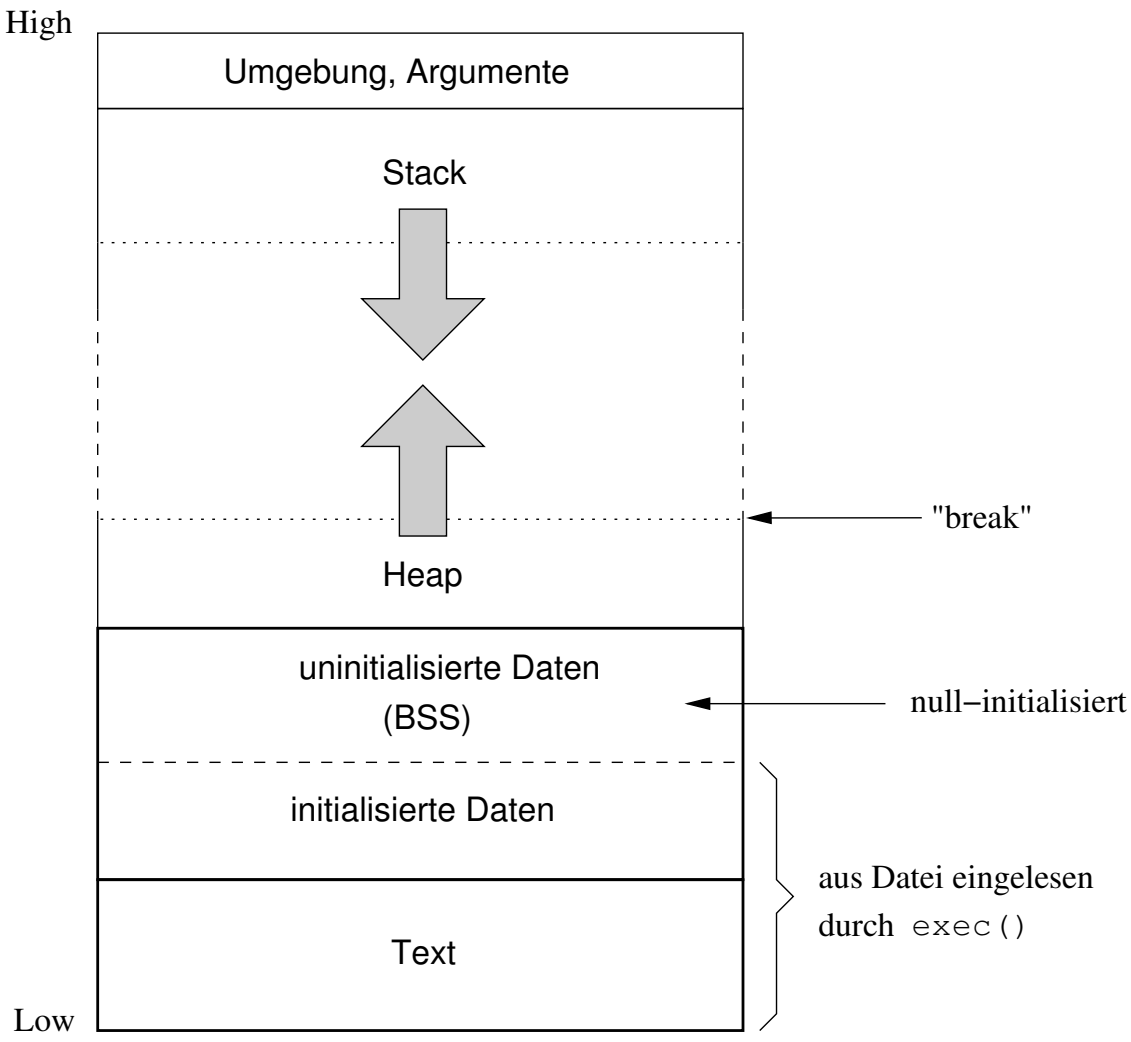



Quelle: David Solomon, Inside Windows 2000, Microsoft Press, 2000

# Speicherabbild

- jeder Prozess besitzt eigenen Adressraum (Größe systemabhängig, typisch $2^{32}$ Bytes)
- Adressraum ist exklusiv (Ausnahme: *Shared-Memory-Segmente*)
- Bestandteile (Abb. 10) eines Adressraums in UNIX:
  - Text: Programmcode
  - Data: initialisierte Daten
  - BSS: uninitialisierte Daten, “Heap”
  - Stack

# Prinzipieller Adressraumaufbau eines Prozesses



# Prozessverwaltung

- Prozesse werden unterbrochen und fortgesetzt (Wechsel zwischen *bereit* und *aktiv*)
- $\rightarrow$ alle Informationen, die für Fortsetzung benötigt werden (= *Mikro*zustand), müssen archiviert werden
- $\rightarrow$ *Prozesstabelle* aka Process Control Block (PCB)
- konkrete Ausprägung der Parameter stark systemabhängig
- Beispiel eines Eintrags: Tabelle 1
- Linux: `struct task_struct` in `include/linux/sched.h`; ca. 1.7 kBytes groß

# Mikrozustand eines Prozesses

| **Prozessverwaltung** | **Speicherverwaltung** | **Dateiverwaltung** |
|---|---|---|
| Register | Zeiger auf Text-Segment | Wurzelverzeichnis |
| Befehlszeiger | Zeiger auf Data-Segment | Arbeitsverzeichnis |
| Flagregister | Zeiger auf Stack | offene Dateideskriptoren |
| Globalzustand | | User ID |
| Priorität | | Gruppen-ID |
| Prozess-ID | | |
| ID des Vaters | | |
| Zeitstempel | | |
| erhaltene CPU-Zeit | | |

Tabelle: Typischer Eintrag in der Prozesstabelle

# Informationen zu Prozessen: das Kommando `ps`

gibt tabellarisch zu jedem Prozess des Nutzers aus

- PID (Prozess-ID)
- TTY (das zugehörige Terminal)
- Zustand (Status) des Prozesses
- die bislang konsumierte CPU-Zeit
- das zugrundeliegende Kommando

# Kommando `ps` (Fortsetzung)

Kommandoswitches von `ps`, die Sie brauchen werden:

- `-A` listet alle Prozesse
- `r` listet alle bereiten Prozesse (, die sich die CPU teilen)
- `X` gibt Inhalt des Stackpointers und einiger weiterer Register aus
- `f` zeichnet Verwandtschaftsverhältnisse mit ASCII-Grafik (besser: `pstree`-Kdo.)
- `-l` langes Format (zusätzlich UID, Parent PID, Priorität, Größe)

Ein falscher Kommandozeilenparameter gibt eine kurze Zusammenfassung der gültigen Switches aus.

Achtung: Die Syntax der Optionen von `ps` ist kompliziert; manchmal mit vorangestelltem '-', manchmal ohne.

# Weitere wichtige Prozess-Kommandos

- `top` - kontinuierliche Prozessbeobachtung
- `pstree` - (text-)grafische Veranschaulichung von Prozessverwandschaften
- `pgrep` - Suche nach Prozessen mittels regulärer Ausdrücke

  **Beispiel:**

  ```
  pgrep -l "[[:alpha:]]*d\>"
  ```

  listet die PID und Namen aller Daemon-Prozesse
- `nice` - Setzen der Prozesspriorität
- `kill` - Senden von Signalen

# Erzeugung von Prozessen

- Nur ein Prozess kann einen anderen Prozess erzeugen (lassen), z. B. durch
  - Doppelklick auf ein Icon
  - Eingabe eines Kommandos
  - Abarbeitung eines Skriptes
  - Bootvorgang des Rechners
- Mechanismus: Systemruf
  - UNIX: `fork()`
  - Win32: `CreateProcess()`
- erzeugter Prozess landet zunächst im *Bereit*-Zustand

# Beispiel: Prozesserzeugung im Shellskript

```
#!/bin/bash

# number of xterms to start
if [ "$1" == "" ]
then
    iterations=1
else
    iterations=$1
fi

# do the (dirty) work
for (( count=0; count < $iterations; count++))
do
  xterm &
done

# finish(ed)
exit 0
```

# Erzeugung eines Unix-Prozesses mittels `fork()`

```
pid_t  fork(void);
```

- erzeugt *identische* Kopie des rufenden Prozesses, mit differierendem PID und PPID (*Parent Process Identificator*)
- *beide* Prozesse setzen nach `fork()` fort und sind fortan unabhängig voneinander
- Es ist nicht vorhersehbar, ob Vater oder Sohn zuerst `fork()` verlassen
- Resultat:
  - Vater: -1 im Fehlerfalle, PID des Sohnes ansonsten
  - Sohn: 0
- Vater-Sohn-Verwandschaft
- Vater und Sohn arbeiten identischen Code ab, haben aber private Variablen

# Typischer Einsatz von `fork()`

```
int main(int argc, char* argv[])
{
  pid_t ret;

  ret = fork();
  if (ret == -1) {
    printf("fork() failed. Stop.\n");
    exit(EXIT_FAILURE);
  }
  if (ret == 0) { /* Sohn */
    printf("Ich bin der Sohn!\n");
    exit(EXIT_SUCCESS);
  }
  else { /* Vater */
    printf("Ich bin der Vater!\n");
    printf("Der PID des Sohnes betraegt %d.\n", ret);
    exit(EXIT_SUCCESS);
  }
}
```

# Wieviel Prozesse schlafen?

```c
#include <unistd.h>

int main(void)
{
  fork();
  fork();
  fork();
  sleep(60);

  return 0;
}
```

# Variablen sind privat

```
int var = 42;

int main(int argc, char* argv[])
{
  pid_t ret;
  if ((ret = fork())== -1) {
    printf("fork() failed. Stop.\n");
    exit(EXIT_FAILURE);
  }
  if (ret == 0) { /* Sohn */
    var = 32168;
    printf("Sohns 'var' hat den Wert %d.\n", var);
    sleep(5);
    printf("Sohns 'var' hat (immer noch) den Wert %d
        .\n", var);
    exit(EXIT_SUCCESS);
  }
  else { /* Vater */
    sleep(2);
    printf("Vaters 'var' hat den Wert %d.\n", var);
    exit(EXIT_SUCCESS);
  }
```

# Die Bibliotheksfunktion `system()`

```
int system (const char* string);
```

- führt das Kommando `string` mittels `/bin/sh -c` aus
- `string` kann Kommando und dessen Parameter enthalten
- kehrt erst zurück, wenn Kommando beendet wurde
- kombiniert `fork()` und `exec()`

# Überlagerung des Prozessabbilds mittels `execl()`

- `execl()` übernimmt (u. a.) eine Pfadangabe einer ausführbaren Binärdatei als Parameter
- ersetzt den aktuell abgearbeiteten Programmcode durch diese Binärdatei
- springt diesen Code sofort an und beginnt, diesen abzuarbeiten
- kehrt nur im Fehlerfalle zurück (z. B. bei falscher Pfadangabe)
- Rückkehr in Ausgangsprozess unmöglich (!)
- Systemruf-Familie: 5 Rufe mit sehr ähnlicher Semantik (`execl()`, `execle()`, `execv()`, `execlp()` und `execvp()`)
- erzeugt *keinen* neuen Prozess

# Überlagerung des Prozessabbilds mittels `execl()`

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

int main(int argc, char* argv[])
{
  int ret;

  printf("%s vor Aufruf von execl()\n", argv[0]);
  ret = execl("/bin/ls", "ls", NULL);
  if (ret == -1) {
    printf("execl() ging schief. Und nun?\n");
    exit (EXIT_FAILURE);
  }
  /* wird nicht erreicht ! */
  printf("%s nach Aufruf von execl()\n", argv[0]);
  exit (EXIT_SUCCESS);
}
```

# Beendigung von Prozessen

Beendigung kann selbst oder durch anderen Prozess erfolgen (falls dieser die Rechte dazu besitzt)

- Selbstbeendigung:
  - Verlassen von `main()`,
  - `return` innerhalb von `main()`,
  - `exit()` an beliebiger Stelle im Programm, z. B. als Folge eines Fehlers
- Fremdbeendigung:
  - Zustellung eines Signals durch anderen Prozess
  - fataler Fehler durch den Prozess selbst (Division durch Null, illegale Instruktion, Referenz eines ungültigen Zeigers, . . . )

# Möglichkeit zur Beendigung: durch das System

```c
#include <stdio.h>

int main(int argc, char* argv[])
{
  int ret = 42;
  int x = 0;

  ret = ret / x;
  printf("Geschafft!\n");
  return 0;
}
```

Abarbeitung:

```
robge@ilpro121:~> ./div-by-zero
Gleitkomma-Ausnahme
```

# Möglichkeit der Beendigung: exit (mit Rückkehrcode)

Listing 1: Generierung eines Rückkehrcodes (`retval.c`)

```
#include <stdlib.h>

int main(int argc, char* argv[])
{
  if (argc==2) {
    exit (atoi(argv[1]));
  }
  else {
    exit(42);
  }
}
```

Listing 2: Abfrage des Rückkehrcodes im Shellskript

```
#!/bin/bash
./retval 14
echo $?
./retval
echo $?
```

# Synchronisation mittels `wait()`

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

- bringt den rufenden Prozess in den Wartezustand
- dieser wird (automatisch) wieder verlassen, wenn ein (*beliebiger*) Kindprozess terminiert
- falls kein Kindprozess existiert, wird einfach fortgesetzt
- status enthält Statusinformationen zum Kindprozess (u. a. Rückkehrcode)
- Resultat:
  - -1 bei Fehler
  - PID des beendeten Kindprozesses ansonsten

$\rightarrow$ zur Synchronisation zwischen Vater und Sohn nutzbar

# Beispiel 1 zu `wait()`

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

int main(int argc, char* argv[])
{
  pid_t ret;

  ret = fork();
  if (ret == -1) {
    perror("fork");
    exit(EXIT_FAILURE);
  }
  if (ret == 0) { /* Sohn */
    printf("Sohn geht schlafen...\n");
    sleep(10);
    printf("Sohn erwacht und endet.\n");
    exit(EXIT_SUCCESS);
  }
  else {          /* Vater */
    printf("Vater wartet auf Sohns Ende.\n");
    ret = wait(NULL);
    if (ret == -1) {
      perror("wait");
      exit(EXIT_FAILURE);
    }
    printf("Vater endet (nach Sohn).\n");
    exit(EXIT_SUCCESS);
  }
}
```

# Beispiel 2 zu `wait()`

```c
#include <stdio.h>
#include <unistd.h>
#include <sys/wait.h>

int main(int argc, char* argv[])
{
  sleep(20);
  fork();           /* 1. */
  sleep(20);
  fork();           /* 2. */
  wait(NULL);
  sleep(20);
  fork();           /* 3. */
  sleep(20);
  return 0;
}
```

Wann sind welche Prozesse im System?

# fork(), exec() und wait() zusammen: eine Shell

Eine Shell tut im Prinzip nichts weiter als:

1: **loop**
2: Kommando $\rightarrow$ von `stdin` einlesen
3: `fork()`
4: Sohnprozess überlagert sich selbst mit Kommando &&
Vater wartet auf die Beendigung des Sohnes
5: **end loop**

Beispiel: `minishell.c` (extern, da zu groß)

# Windows: `CreateProcess()`

- keine Verwandtschaft zwischen Prozessen $\rightarrow$ keine Hierarchie
- legt neuen Adressraum an ($\rightarrow$ neuer Prozess)
- startet in diesem einen Thread, der das angegebene Programm ausführt
- gewissermaßen Hintereinanderausführung von `fork()` und `exec()`

```
BOOL CreateProcess (
  LPCTSTR lpApplicationName,    // pointer to name of executable module
  LPSTR lpCommandLine,          // pointer to command line string
  LPSECURITY_ATTRIBUTES lpProcessAttributes,
  LPSECURITY_ATTRIBUTES lpThreadAttributes,
  BOOL bInheritHandles,         // handle inheritance flag
  DWORD dwCreationFlags,
  LPVOID lpEnvironment,         // pointer to new environment block
  LPCTSTR lpCurrentDirectory,   // pointer to current directory name
  LPSTARTUPINFO lpStartupInfo,
  LPPROCESS_INFORMATION lpProcessInformation
);
```

# Ist das alles zu *Aktivitäten*?

Mitnichten!

- `vfork()`, `clone()`, ...
- Threads
- Coroutinen und Fibers
- Kommunikation
- Synchronisation

# Was haben wir gelernt?

1. Begriff des Prozesses
2. Zustände und Transitionen zwischen ihnen
3. Prozesserzeugung in Unix mittels `fork()`
4. Überlagerung des Prozessabbilds mittels `exec()`
5. Methoden der Prozessbeendigung
6. einfache Synchronisation mittels `wait()`