

Betriebssysteme (BS)

Prozesse

Olaf Spinczyk

Arbeitsgruppe Eingebettete Systemsoftware

Lehrstuhl für Informatik 12

TU Dortmund

Olaf.Spinczyk@tu-dortmund.de

<http://ess.cs.uni-dortmund.de/~os/>





- Wiederholung
- Prozesse konkret: UNIX-Prozessmodell
 - Shells und E/A
 - UNIX-Philosophie
 - Prozesserzeugung
 - Signale
 - Prozesszustände
- Leichtgewichtige Prozessmodelle
 - „Gewicht“ von Prozessen
 - Leichtgewichtige Prozesse
 - Federgewichtige Prozesse
- Systeme mit leichtgewichtigen Prozessen
 - Windows NT
 - Linux

Silberschatz, Kap. ...
3.1-3.3: Process Concept
21.4: Linux

Tanenbaum, Kap. ...
2.1: Prozesse
10.1-10.3: UNIX u. Linux

Silberschatz, Kap. ...
4: Multithreaded Progr.

Tanenbaum, Kap. ...
2.2: Threads

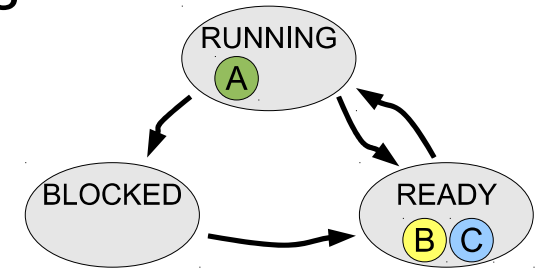
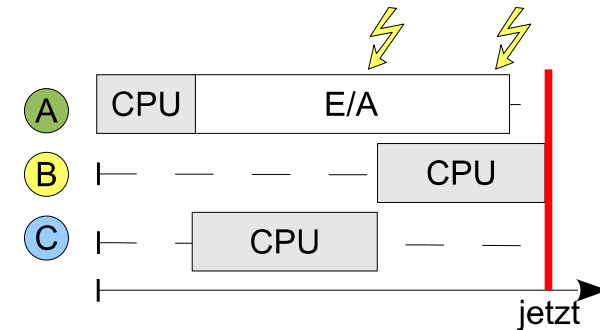


- **Wiederholung**
- Prozesse konkret: UNIX-Prozessmodell
 - Shells und E/A
 - UNIX-Philosophie
 - Prozesserzeugung
 - Signale
 - Prozesszustände
- Leichtgewichtige Prozessmodelle
 - „Gewicht“ von Prozessen
 - Leichtgewichtige Prozesse
 - Federgewichtige Prozesse
- Systeme mit leichtgewichtigen Prozessen
 - Windows NT
 - Linux



Wiederholung: Prozesse ...

- sind „Programme in Ausführung“
 - Dynamisch, nicht statisch
 - Abwechselnde Folge von „CPU-Stößen“ und „E/A-Stößen“
- benötigen „Betriebsmittel“ des Rechners
 - CPU, Speicher, E/A-Geräte
- haben einen Zustand
 - READY, RUNNING, BLOCKED
- werden **konzeptionell** als unabhängige, nebenläufige Kontrollflüsse betrachtet
- unterliegen der Kontrolle des Betriebssystems
 - Betriebsmittel-Zuteilung
 - Betriebsmittel-Entzug





- Wiederholung
- **Prozesse konkret: UNIX-Prozessmodell**
 - Shells und E/A
 - UNIX-Philosophie
 - Prozesserzeugung
 - Signale
 - Prozesszustände
- Leichtgewichtige Prozessmodelle
 - „Gewicht“ von Prozessen
 - Leichtgewichtige Prozesse
 - Federgewichtige Prozesse
- Systeme mit leichtgewichtigen Prozessen
 - Windows NT
 - Linux



UNIX (K. Thompson, D. Ritchie, 1968)

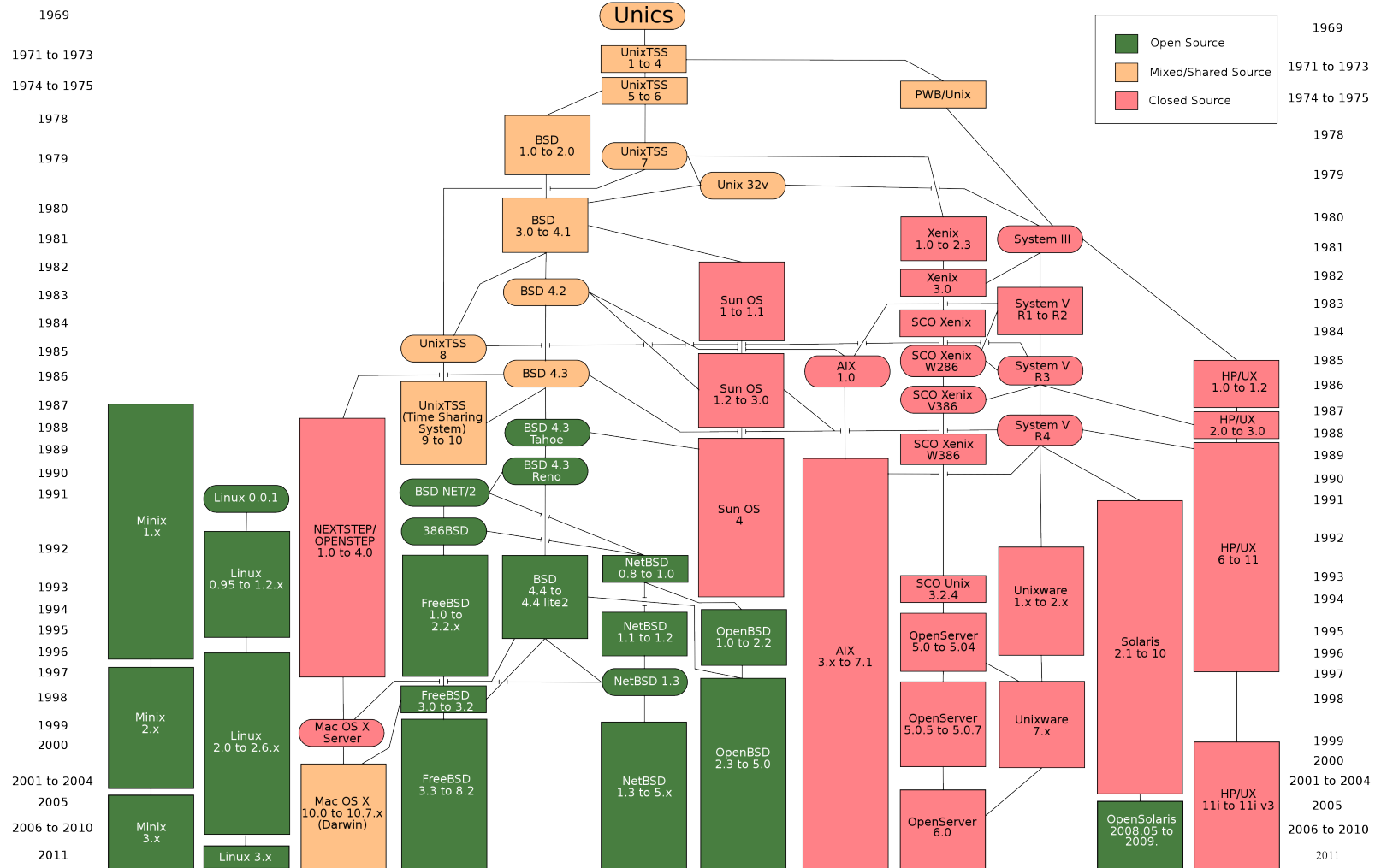
- Eine lange Geschichte ...
- Ursprung: Bell Labs
 - Alternative zu „Multics“
- Version 1 entstand auf einer DEC PDP 7
 - Assembler, 8K 18 Bit Worte
- Version 3 in „C“ realisiert



UNIX (K. Thompson, D. Ritchie, 1968)

• Eine lange Geschichte ...

Quelle: Wikipedia

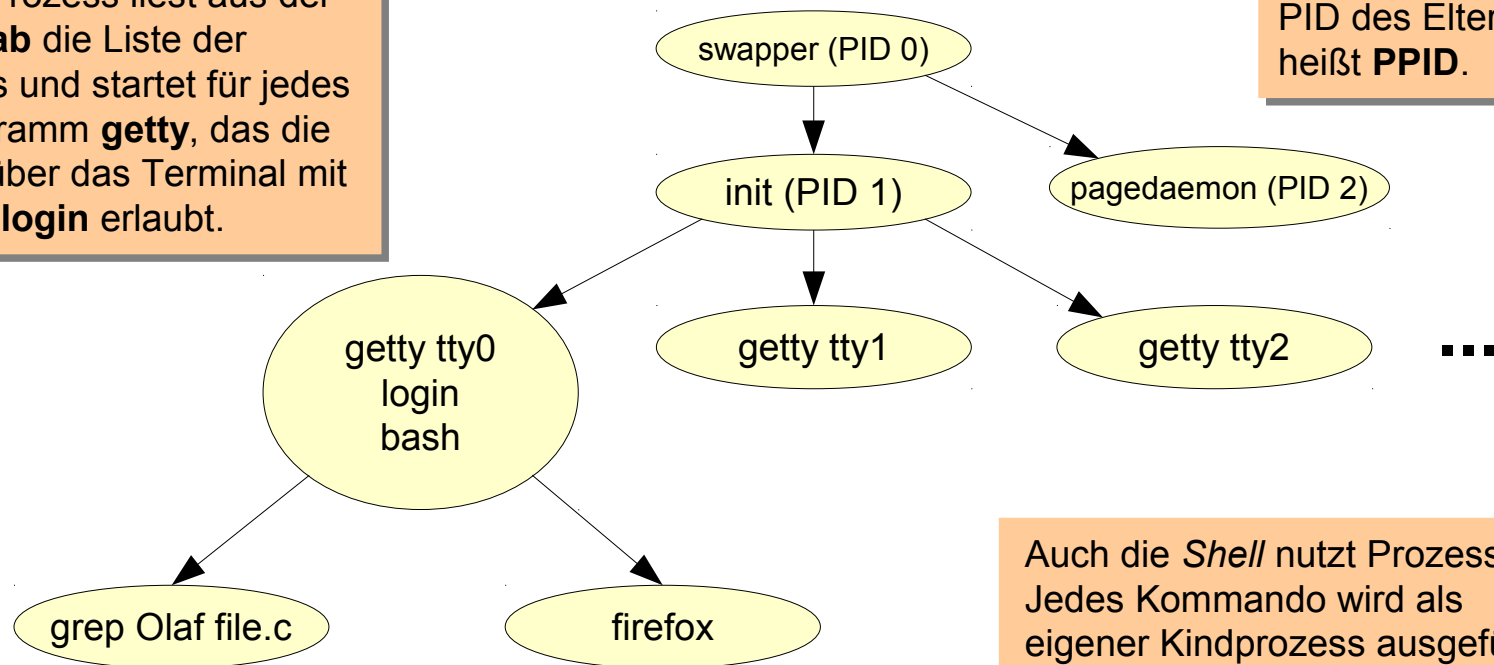




UNIX-Prozesse ...

- sind primäres **Strukturierungskonzept** für Aktivitäten
 - Anwendungsprozesse und Systemprozesse
- können leicht und schnell weitere Prozesse erzeugen
 - Elternprozess → Kindprozess
- bilden eine **Prozess-Hierarchie**:

Der **init**-Prozess liest aus der **/etc/inittab** die Liste der Terminals und startet für jedes das Programm **getty**, das die Einwahl über das Terminal mit Hilfe von **login** erlaubt.



Jeder UNIX-Prozess hat eine eindeutige Nummer (Prozess-ID, **PID**). Die PID des Elternprozesses heißt **PPID**.

Auch die *Shell* nutzt Prozesse: Jedes Kommando wird als eigener Kindprozess ausgeführt.



UNIX-Shells

- „Schale“ (*shell*), die den „Kern“ (*kernel*) umgibt
- Textbasierte Nutzerschnittstelle zum Starten von Kommandos:
 - Suche im Dateisystem entsprechend \$PATH (z.B. /usr/bin:/bin:...)

```
olaf@xantos:~> which emacs
/usr/bin/emacs
```

Das Kommando **which** zeigt an, wo ein bestimmtes Kommando gefunden wird.

- Jedes ausgeführte Kommando ist ein eigener Kindprozess
- Typischerweise blockiert die Shell bis das Kommando terminiert
- Man kann aber auch Kommandos stoppen und fortsetzen („*job control*“) oder sie im Hintergrund ausführen ...



UNIX-Shells: Job Control

```
olaf@xantos:~> emacs foo.c
```

Ctrl-Z

```
[1]+  Stopped      emacs foo.c
olaf@xantos:~> kate bar.c &
[2] 19504
olaf@xantos:~> jobs
[1]+  Stopped      emacs foo.c
[2]-  Running      kate bar.c &
olaf@xantos:~> bg %1
[1]+  emacs foo.c &
olaf@xantos:~> jobs
[1]-  Running      emacs foo.c &
[2]+  Running      kate bar.c &
```

- Kommando wird gestartet
- die *Shell* blockiert

- Kommando wird gestoppt
- die *Shell* läuft weiter

- Durch das **&** am Ende wird **kate** im Hintergrund gestartet

- **jobs** zeigt alle gestarteten Kommandos an

- **bg** schickt ein gestopptes Kommando in den Hintergrund



Standard-E/A-Kanäle von Prozessen

- Normalerweise verbunden mit dem *Terminal*, in dem die *Shell* läuft, die den Prozess gestartet hat:
 - **Standard-Eingabe** Zum Lesen von Benutzereingaben (Tastatur)
 - **Standard-Ausgabe** Textausgaben des Prozesses (Terminalfenster)
 - **Standard-Fehlerausgabe** Separater Kanal für Fehlermeldungen (normalerweise auch das *Terminal*)
- Praktisch alle Kommandos akzeptieren auch Dateien als Ein- oder Ausgabekanäle (statt des *Terminals*)
- *Shells* bieten eine einfache Syntax, um die Standard-E/A-Kanäle umzuleiten ...



Standard-E/A-Kanäle umleiten

Umleitung der Standard-Ausgabe
in die Datei „l1“ mit >

```
olaf@xantos:~> ls -l > l1
olaf@xantos:~> grep "Sep 2007" < l1 > l2
olaf@xantos:~> wc < l2
 2  18 118
```

Umleitung der Standard-Eingabe
auf die Datei „l2“ mit <

Das gleiche noch etwas kompakter ...

```
olaf@xantos:~> ls -l | grep "Sep 2007" | wc
 2  18 118
```

Mit | (*pipe*) verbindet die Shell die Standard-Ausgabe
des linken mit der Standard-Eingabe des rechten Prozesses.



Die UNIX-Philosophie

Doug McIlroy, der Erfinder der UNIX-Pipes, fasste die Philosophie hinter UNIX einmal wie folgt zusammen:

"This is the Unix philosophy:

- *Write programs that do one thing and do it well.*
- *Write programs to work together.*
- *Write programs to handle text streams, because that is a universal interface."*

Für gewöhnlich wird das abgekürzt:

„Do one thing, do it well.“



UNIX-Prozesssteuerung: *System Calls*

Ein erster Überblick ...

- **getpid** (2) liefert PID des laufenden Prozesses
- **getppid** (2) liefert PID des Elternprozesses (PPID)
- **getuid** (2) liefert die Benutzerkennung des laufenden Prozesses (UID)

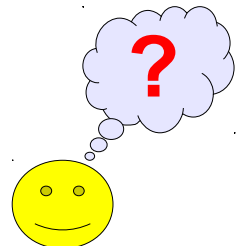
- **fork** (2) erzeugt neuen Kindprozess
- **exit** (3), **_exit** (2) beendet den laufenden Prozess
- **wait** (2) wartet auf die Beendigung eines Kindprozesses
- **execve** (2) lädt und startet ein Programm im Kontext des laufenden Prozesses



UNIX-Prozesse im Detail: fork()

System Call: **pid_t fork (void)**

- Dupliziert den laufenden Prozess (Prozesserzeugung!)
- Der Kindprozess erbt ...
 - Adressraum (code, data, bss, stack)
 - Benutzerkennung
 - Standard-E/A-Kanäle
 - Prozessgruppe, Signaltabelle (dazu später mehr)
 - Offene Dateien, aktuelles Arbeitsverzeichnis (dazu viel später mehr)
- Nicht kopiert wird ...
 - *Process ID (PID), Parent Process ID (PPID)*
 - anhängige Signale, *Accounting*-Daten, ...
- Ein Prozess ruft **fork** auf,
aber zwei kehren zurück

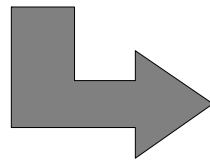




Verwendung von fork()

```
... /* includes */
int main () {
    int pid;
    printf("Elternpr.: PID %d PPID %d\n", getpid(), getppid());
    pid = fork(); /* Prozess wird dupliziert!
                  Beide laufen an dieser Stelle weiter. */

    if (pid > 0)
        printf("Im Elternprozess, Kind-PID %d\n", pid);
    else if (pid == 0)
        printf("Im Kindprozess, PID %d PPID %d\n",
              getpid(), getppid());
    else
        printf("Oh, ein Fehler!\n"); /* mehr dazu in der TÜ */
}
```



```
olaf@xantos:~> ./fork
Elternpr.: PID 7553 PPID 4014
Im Kindprozess, PID 7554 PPID 7553
Im Elternprozess, Kind-PID 7554
```




Diskussion: Schnelle Prozesserschöpfung

- Das Kopieren des Adressraums kostet viel Zeit
 - Insbesondere bei direkt folgendem **exec..()** pure Verschwendung!
- Historische Lösung: **vfork**
 - Der Elternprozess wird suspendiert, bis der Kindprozess **exec..()** aufruft oder mit **_exit()** terminiert.
 - Der Kindprozess benutzt einfach Code und Daten des Elternprozesses (kein Kopieren!).
 - Der Kindprozess darf keine Daten verändern.
 - teilweise nicht so einfach: z.B. kein **exit()** aufrufen, sondern **_exit()**!
- Heutige Lösung: **copy-on-write**
 - Mit Hilfe der MMU teilen sich Eltern- und Kindprozess dasselbe Code- und Datensegment. Erst wenn der Kindprozess Daten ändert, wird das Segment kopiert.
 - Wenn nach dem **fork()** direkt ein **exec..()** folgt, kommt das nicht vor.
 - Heutzutage benutzen **fork** und **vfork copy-on-write**.



UNIX-Prozesse im Detail: `_exit()`

System Call: **`void _exit (int)`**

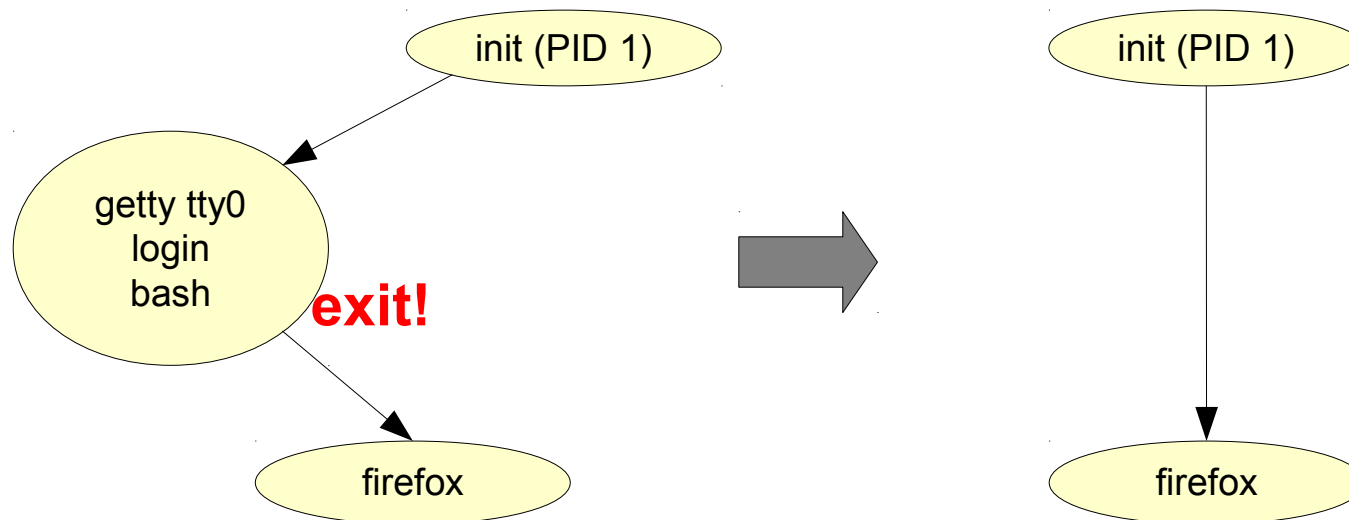
- Terminiert den laufenden Prozess und übergibt das Argument als „*exit status*“ an den Elternprozess.
 - Aufruf kehrt nicht zurück!
- Gibt die belegten Ressourcen des Prozesses frei.
 - offene Dateien, belegter Speicher, ...
- Sendet dem eigenen Elternprozess das Signal SIGCHLD.
- Die Bibliotheksfunktion **`exit`** (3) räumt zusätzlich noch die von der libc belegten Ressourcen auf
 - Gepufferte Ausgaben werden beispielsweise herausgeschrieben!
 - Normale Prozesse sollten **`exit`** (3) benutzen, nicht **`_exit`**.



Diskussion: Verwaiste Prozesse

(engl. „*orphan processes*“)

- Ein UNIX-Prozess wird zum Waisenkind, wenn sein Elternprozess terminiert.
- Was passiert mit der Prozesshierarchie?



**init (PID 1) adoptiert alle verwaisten Prozesse.
So bleibt die Prozesshierarchie intakt.**



UNIX-Prozesse im Detail: wait()

System Call: **pid_t wait (int *)**

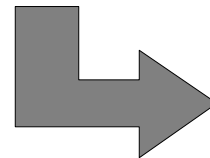
- Blockiert den aufrufenden Prozess bis ein Kindprozess terminiert. Der Rückgabewert ist dessen PID. Über das Zeigerargument erhält der Aufrufer u.a. den „*exit status*“.
- Wenn ein Kindprozess bereits terminiert ist, kehrt der Aufruf sofort zurück.



Verwendung von wait()

```
... /* includes, main() { ... */  
pid = fork(); /* Kindprozess erzeugen */  
if (pid > 0) {  
    int status;  
    sleep(5); /* Bibliotheksfunktion: 5 Sek. schlafen */  
    if (wait(&status) == pid && WIFEXITED(status))  
        printf ("Exit Status: %d\n", WEXITSTATUS(status));  
}  
else if (pid == 0) {  
    exit(42);  
}  
...
```

Ein Prozess kann auch von außen „getötet“ werden, d.h. er ruft nicht **exit** auf. In diesem Fall würde **WIFEXITED** 0 liefern.



```
olaf@xantos:~> ./wait  
Exit Status: 42
```



Diskussion: *Zombies*

- Bevor der *exit status* eines terminierten Prozesses mit Hilfe von **wait** abgefragt wird, ist der ein „Zombie“.
- Die Ressourcen solcher Prozesse können freigegeben werden, aber die Prozessverwaltung muss sie noch kennen.
 - Insbesondere der *exit status* muss gespeichert werden.

```
olaf@xantos:~> ./wait &
olaf@xantos:~> ps
  PID TTY          TIME CMD
 4014 pts/4        00:00:00 bash
17892 pts/4        00:00:00 wait
17895 pts/4        00:00:00 wait <defunct>
17897 pts/4        00:00:00 ps
olaf@xantos:~> Exit Status: 42
```

Beispielprogramm
von eben während
der 5 Sekunden
Wartezeit.

Zombies werden
von **ps** als
<defunct>
dargestellt.



- Wikipedia: In 1999 the *Library of Congress* entered it into the United States *National Film Registry* with other films deemed „historically, culturally or aesthetically important.“





UNIX Prozesse im Detail: `execve()`

System Call: `int execve (const char *kommando,
const char *args[],
const char *envp[])`

- Lädt und startet das angegebene Kommando.
- Der Aufruf kehrt nur im Fehlerfall zurück.
- Der komplette Adressraum wird ersetzt.
- Es handelt sich aber weiterhin um denselben Prozess!
 - Selbe PID, PPID, offenen Dateien, ...
- Die **libc** bietet einige komfortable Hilfsfunktionen, die intern `execve` aufrufen: **`execl`**, **`execv`**, **`execlp`**, **`execvp`**, ...



Verwendung von exec..()

```
... /* includes, main() { ... */
char cmd[100], arg[100];
while (1) {
    printf ("Kommando?\n");
    scanf ("%99s %99s", cmd, arg);
    pid = fork(); /* Prozess wird dupliziert!
                   Beide laufen an dieser Stelle weiter. */

    if (pid > 0) {
        int status;
        if (wait(&status) == pid && WIFEXITED(status))
            printf ("Exit Status: %d\n", WEXITSTATUS(status));
    }
    else if (pid == 0) {
        execlp(cmd, cmd, arg, NULL);
        printf ("exec fehlgeschlagen\n");
    }
    ...
}
```



Diskussion: Warum kein `forkexec()`?

- Durch die Trennung von **fork** und **execve** hat der Elternprozess mehr Kontrolle:
 - Operationen im Kontext des Kindprozesses ausführen
 - Voller Zugriff auf die Daten des Elternprozesses
- Shells nutzen diese Möglichkeit zum Beispiel zur ...
 - Umleitung der Standard-E/A-Kanäle
 - Aufsetzen von *Pipes*



Signale

- Mit Hilfe von Signalen können Prozesse über Ausnahmesituation informiert werden
 - ähnlich wie Hardwareunterbrechungen
- Beispiele:
 - **SIGINT** Prozess abbrechen (z.B. bei Ctrl-C)
 - **SIGSTOP** Prozess anhalten (z.B. bei Ctrl-Z)
 - **SIGWINCH** Fenstergröße wurde geändert
 - **SIGCHLD** Kindprozess terminiert
 - **SIGSEGV** Speicherschutzverletzung des Prozesses
 - **SIGKILL** Prozess wird getötet
 - ...
- Die Standardbehandlung (terminieren, anhalten, ...) kann für die meisten Signale überdefiniert werden.
 - siehe **signal** (2)



UNIX-Prozesszustände

- ein paar mehr als wir bisher kannten...

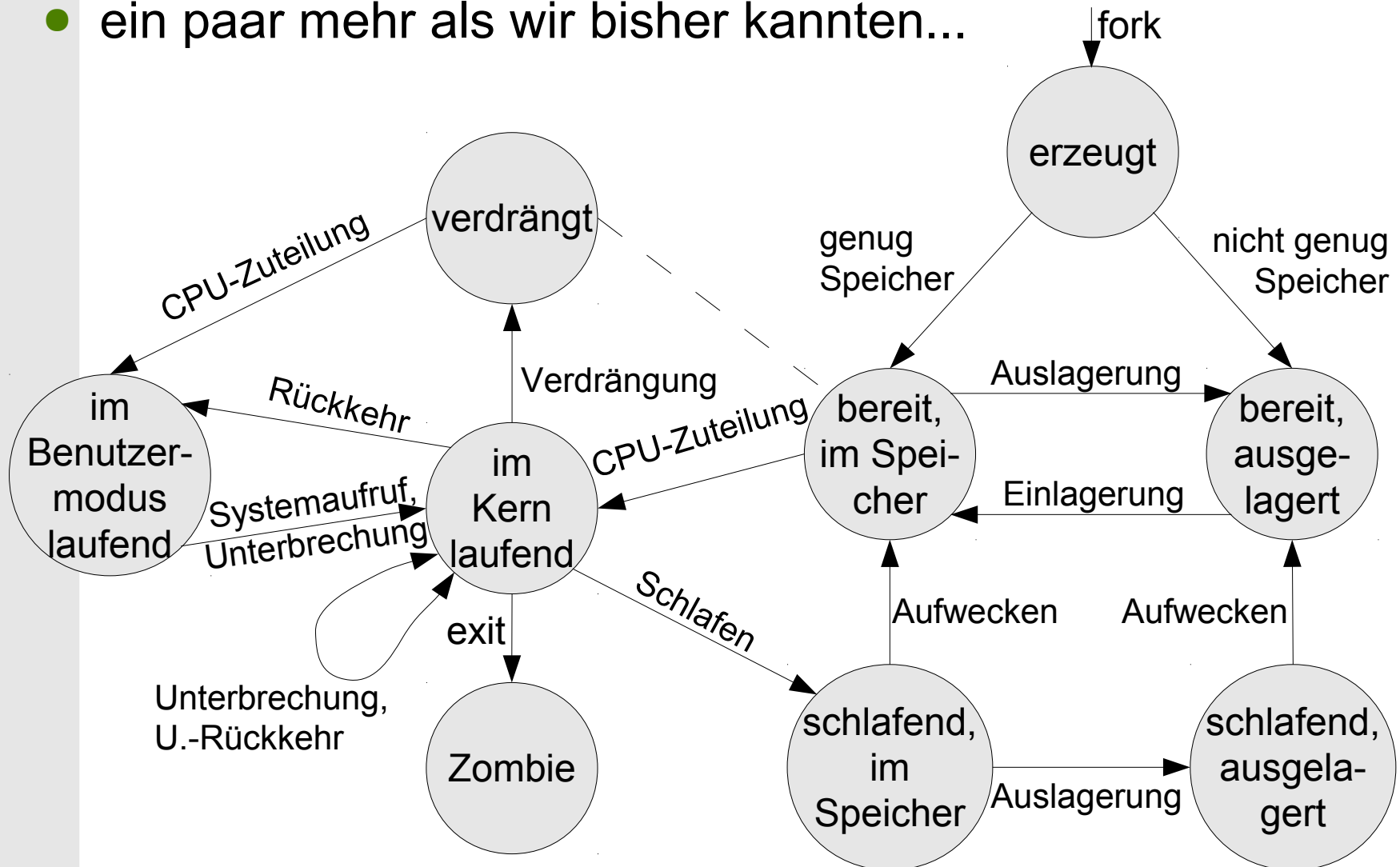


Bild in Anlehnung an M. Bach „UNIX – Wie funktioniert das Betriebssystem?“



Inhalt

- Wiederholung
- Prozesse konkret: UNIX-Prozessmodell
 - Shells und E/A
 - UNIX Philosophie
 - Prozesserzeugung
 - Signale
 - Prozesszustände
- **Leichtgewichtige Prozessmodelle**
 - „Gewicht“ von Prozessen
 - Leichtgewichtige Prozesse
 - Federgewichtige Prozesse
- Systeme mit leichtgewichtigen Prozessen
 - Windows NT
 - Linux



Das „Gewicht“ von Prozessen

- Das **Gewicht** eines Prozesses ist ein bildlicher Ausdruck für die Größe seines Kontexts und damit die Zeit, die für einen Prozesswechsel benötigt wird.
 - CPU-Zuteilungsentscheidung
 - alten Kontext sichern
 - neuen Kontext laden
- Klassische UNIX-Prozesse sind „schwergewichtig“.



Leichtgewichtige Prozesse (*Threads*)

- Die 1:1-Beziehung zwischen Kontrollfluss und Adressraum wird aufgebrochen.
 - Eng kooperierende *Threads* (deutsch „Fäden“) können sich einen Adressraum teilen (*code + data + bss*, nicht *stack*!).
- **Vorteile:**
 - Aufwändige Operationen können in einen leichtgewichtigen Hilfsprozess ausgelagert werden, während der Elternprozess erneut auf Eingabe reagieren kann.
 - Typisches Beispiel: Webserver
 - Programme, die aus mehreren unabhängigen Kontrollflüssen bestehen, profitieren unmittelbar von Multiprozessor-Hardware.
 - Schneller Kontextwechsel, wenn man im selben Adressraum bleibt.
 - Je nach *Scheduler* eventuell mehr Rechenzeit.
- **Nachteil:**
 - Programmierung ist schwierig: Zugriff auf gemeinsame Daten muss koordiniert werden.



Federgewichtige Prozesse

(engl. *User-Level Threads*)

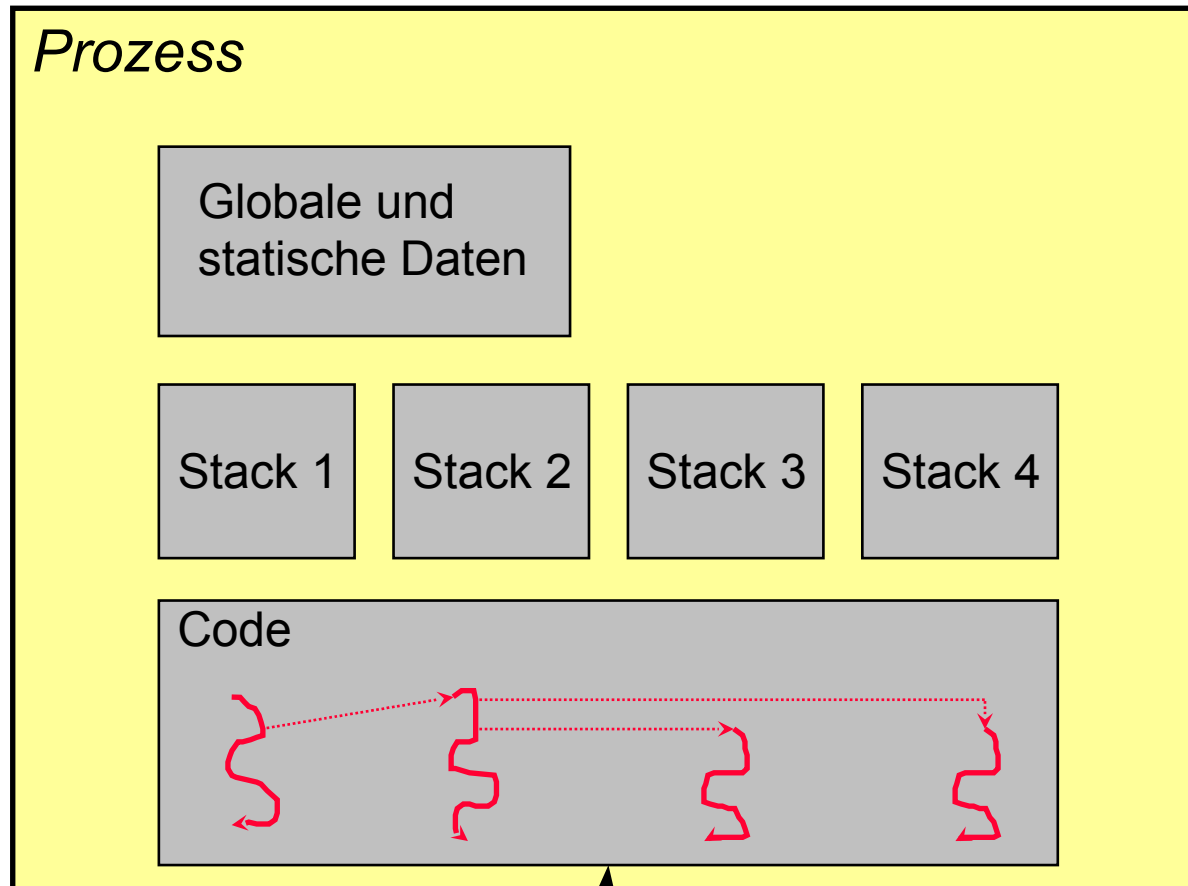
- Werden komplett auf der Anwendungsebene implementiert. Das Betriebssystem weiß nichts davon.
 - Realisiert durch Bibliothek: *User-Level Thread Package*
- **Vorteile:**
 - Extrem schneller Kontextwechsel: Nur wenige Prozessorregister sind auszutauschen. Ein *Trap* in den Kern entfällt.
 - Jede Anwendung kann sich das passende *Thread-Package* wählen.
- **Nachteile:**
 - Blockierung eines federgewichtigen Prozesses führt zur Blockierung des ganzen Programms.
 - Kein Geschwindigkeitsvorteil durch Multi-Prozessoren.
 - Kein zusätzlicher Rechenzeitanteil.



- Wiederholung
- Prozesse konkret: UNIX-Prozessmodell
 - Shells und E/A
 - UNIX Philosophie
 - Prozesserzeugung
 - Signale
 - Prozesszustände
- Leichtgewichtige Prozessmodelle
 - „Gewicht“ von Prozessen
 - Leichtgewichtige Prozesse
 - Federgewichtige Prozesse
- **Systeme mit leichtgewichtigen Prozessen**
 - Windows NT
 - Linux



Threads in Windows (1)



Ein Prozess enthält 1 bis N Thread, die auf denselben globalen Daten operieren.



Threads in Windows (2)

- **Prozess:** Umgebung und Adressraum für *Threads*
 - Ein Win32-Prozess enthält immer mindestens 1 *Thread*
- **Thread:** Code ausführende Einheit
 - Jeder *Thread* verfügt über einen eigenen Stack und Registersatz (insbesondere PC)
 - *Threads* bekommen vom *Scheduler* Rechenzeit zugeteilt
- Alle *Threads* sind *Kernel-Level Threads*
 - *User-Level Threads* möglich („*Fibers*“), aber unüblich
- Strategie: Anzahl der *Threads* gering halten
 - Überlappte (asynchrone) E/A



Threads in Linux

- Linux implementiert **POSIX Threads** in Form der ***pthread***-Bibliothek
- Möglich macht das ein Linux-spezifischer *System Call*

Linux System Call:

```
int __clone (int (*fn)(void *), void *stack,  
            int flags, void *arg)
```

- Universelle Funktion, parametrisiert durch **flags**
 - CLONE_VM Adressraum gemeinsam nutzen
 - CLONE_FS Information über Dateisystem teilen
 - CLONE_FILES Dateideskriptoren (offene Dateien) teilen
 - CLONE_SIGHAND Gemeinsame Signalbehandlungstabelle
- Für Linux sind alle *Threads* und Prozesse intern „*Tasks*“
 - Der *Scheduler* macht also keinen Unterschied



Zusammenfassung

- Prozesse sind die zentrale Abstraktion für Aktivitäten in heutigen Betriebssystemen.
- UNIX-Systeme stellen diverse *System Calls* zur Verfügung, um Prozesse zu erzeugen, zu verwalten und miteinander zu verknüpfen.
 - alles im Sinne der Philosophie: „*Do one thing, do it well.*“
- Leichtgewichtige Fadenmodelle haben viele Vorteile
 - in UNIX-Systemen bis in die 90er Jahre nicht verfügbar
 - in Windows von Beginn an (ab NT) integraler Bestandteil