UNIX-Schnittstelle

1. Prozesse

Alle Beispiel-Quellen mittels SVN unter:

https://svn.informatik.hu-berlin.de/svn/unix2014/Prozesse

1.Vorbemerkungen

Was ist ein Prozess????

1. Menge von Befehlen und Daten einschließlich der aktuellen
   Werte der Prozessorregister.
   (virtuelle Maschine)

2. Tupel von Informationen, das die Arbeit der CPU in jedem
   Zeitpunkt vollständig charakterisiert.
   Problem: Abgrenzung eines Prozesses, Initialzustand.

3. UNIX:
   Ein Programm, das ausgeführt wird. Initialzustand ist in
   einem File gespeichert. Ein Prozess muss Systemressourcen
   wie Speicher und die CPU haben. UNIX unterstützt die Illusion
   von der gleichzeitigen Arbeit von Prozessen.
   (Prozess = ausführbare Instanz eines Programms)
    Problem: Prozess 0 -init- im UNIX???

Betrachtungsweise von Prozessen

- Prozess als aktives Element zur Beschreibung des Organisations-
  prinzips eines Betriebssystems auf einem Rechner und eines
  Algorithmus (Programms)
  Prozesse:
  - werden geboren, leben und sterben
  - sind in ihrer Zahl variabel
  - können Ressourcen belegen und freigeben
  - können einander beeinflussen
  - können zusammenarbeiten
  - können in Konflikt geraten (sich blockieren)
  - Ressourcen teilen
  - voneinander abhängig sein
  - können parallel arbeiten (voneinander unabhängig sein)
    (gleichberechtigt)
  - können hierarchisch voneinader abhängig sein
- Prozess als passives Element, auf das die aktiven Elemente
  (Prozessor und Peripherie) wirken. Ein Prozess erscheint als
  Datenstruktur.
  UNIX-Datenstrukturen für einen Prozess:
  1. Codesegment (Anfangszustand im File, sonst im Speicher
     oder geswappt)
  2. Datensegment (Anfangszustand im File, sonst im Speicher
     oder geswappt)
  3. Stacksegment (im Speicher oder geswappt)
  4. Eintrag in der proc-Liste (immer im Speicher)
     Headerfile: sys/proc.h
  5. user-Struktur (u-Struktur)(im Speicher oder geswappt)
     Headerfile: sys/user.h

proc-Struktur

Identifiers:

p_pid	Prozessnummer
pp_pid	Elternprozessnummer
p_pgrp	Prozessgruppenidentifier
p_uid	Useridentifier
p_gid	Gruppenidentifier
p_suid	effektiver Useridentifier
p_sgid	effektiver Gruppenidentifier

Schedulinginformationen:

p_flag	Flags
SLOAD	Prozess im Hauptspeicher
SSYS	Prozess vom System erstellt (Swapper, Page-Daemon)
SLOCK	Prozess wird gerage ausgelaggert (geswappt)
SSWAP	Rueckkehr nach dem Auslaggern
STRC	Prozess wird "getraced"
SWTED	Prozess wird "getraced"
SOUSIG	Prozess benutzt alten Signalmechanismus
SULOCK	Prozess darf nicht geswappt werden
SPAGE	Prozess wartet auf eine Seite
SKEEP	Kernel fordert den Prozess
SOWEUPC	Prozess wartet auf CPU-Zeit für Systemruf
SOMASK	Signalmaske Wiedererstellen
SWEXIT	Prozess wird beendet
SPHYSIO	Physische E/A-Opertion wird ausgeführt
SVFORK	vfork-Prozess
SNOVM	Child besitzt VS des Elternprozesses vfork()
SVFDONE	Child hat VS des Elternprozesses zurückgegeben vfork()

```
        SPAGI  -  Prozess hat Seiten des VM aus dem Dateisytem
        STIMO  -  TIMEOUT während sleep()
        SSEL   -  Prozess sobald wie möglich auswählen
        SLOGIN -  login-Prozess
    p_stat          Status des Prozesses
        SSLEEP -  Warten auf ein Ereignis
        SRUN   -  Prozess lauffähig
        SIDL   -  Prozess bei der Erzeugung
        SZOMB  -  Zombi-Prozess
        SSTOP  -  Prozess gestoppt
    p_pri           aktuelle Priorität
        PSWP   -  Swapping Priorität  (0)
        PINOD  -  Priorität während des Wartens auf eine Inode (10)
        PRIBIO -  Priorität beim Warten auf Platten-E/A (20)
        PWAIT  -  Priorität beim Warten auf Ressourcen (30)
        PLOCK  -  Priortät beim Warten auf das Locken einer Reseource (35)
        PSLEP  -  Priorität beim Warten auf ein Signal (40)
        PUSER  -  Normale User-Priorität (50)
    p_nice          User nice-Priorität
    p_cpu           neue CPU-Zeit
    p_userpri       berechnet Userpriorität
    p_slptime       Summe der Sleep-Zeiten
Speichermanagment:
    p_textp         Pointer zur Datenstruktur für
                    das Datensegment
    p_pObr          Pointer zur Pagetable
    p_szpt          Zahl der Eintragungen in der Pagetable
    p_addr          Adresse der user-Struktur (u-Struktur)
    p_swaddr        Adresse der user-Struktur
                    während der Prozess geswappt ist
```



```
Synchronisation:
    p_wchan         Prozess auf den gewartet wird
Signalbehandlung:
    p_sig           Maske für hängende Signale
    p_sigignore     Maske für zu ignorierende Signale
    p_sigcatch      Maske für einzufangende Signale
Ressourcenberechnung:
    p_rusage        Zeiger zur Ressourcenstruktur
    p_quota         Zeiger zur Plattennutzungsstruktur
Zeitgebermanagment:
    p_time          Echtzeittimer
Verkettungen in der proc-Struktur:
    p_link, p_rlink  Verkettung in der Run-Queue bzw. Sleep-Queue
    p_nxt           Allgemeine Verkettung (frei, besetzt, Zombi)
    p_pptr      Elternprozess (p_parent)
    p_cptr      1.Kind (p_child)
    p_ysptr     linker Bruder
    p_osptr     rechter Bruder
```

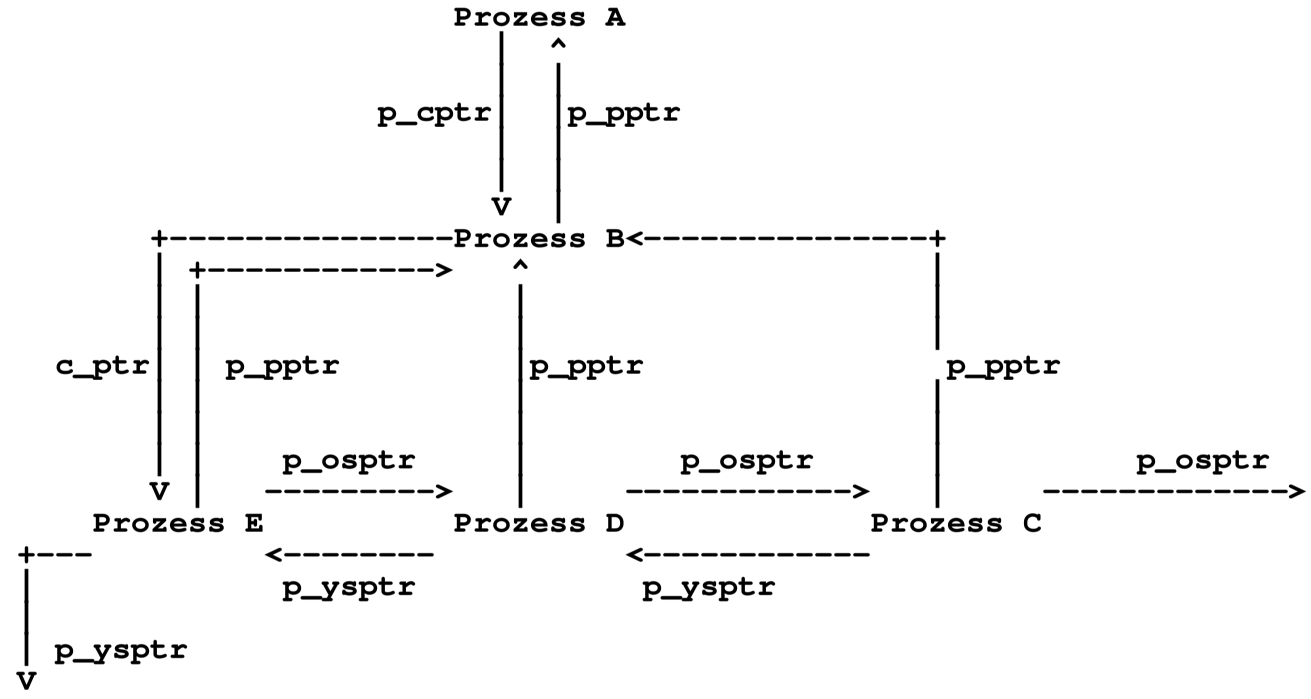
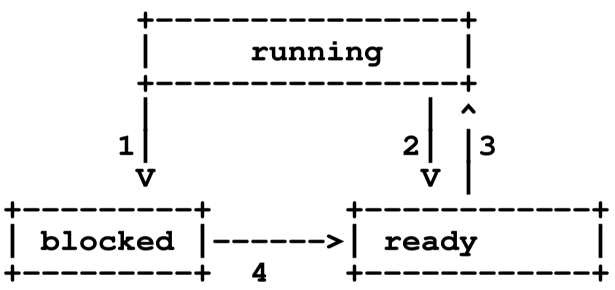


Abbildung: Hierarchie der Prozess-Gruppen

user-Struktur (u-Struktur)

- **Ausführungsstatus**
  user-Modus, kernel-Modus, Register, Adressen
- **Status der Systemrufe**
- **Deskriptortabellen (E/A, Netzwerk, ...)**
- **Abrechnungsinformationen**
- **Ressourcensteuerung**
- **Kernel Process Stack**
- **u-Struktur wird geswappt, 2-6 KB**

Statuswechsel eines Prozesses (einfaches Modell)



1. Prozess wartetet auf ein externes Ereignis
   (read, write, Signal, ...)

2. Scheduler übergibt die Steuerung an einen anderen
   Prozess (Priorität)

3. Prozess erhält Steuerung vom Scheduler (Priorität)

4. Externes Ereignis ist eingetroffen. Prozess kann wieder
   aktiviert werden.

Statuswechsel eines Prozesse in der Realität

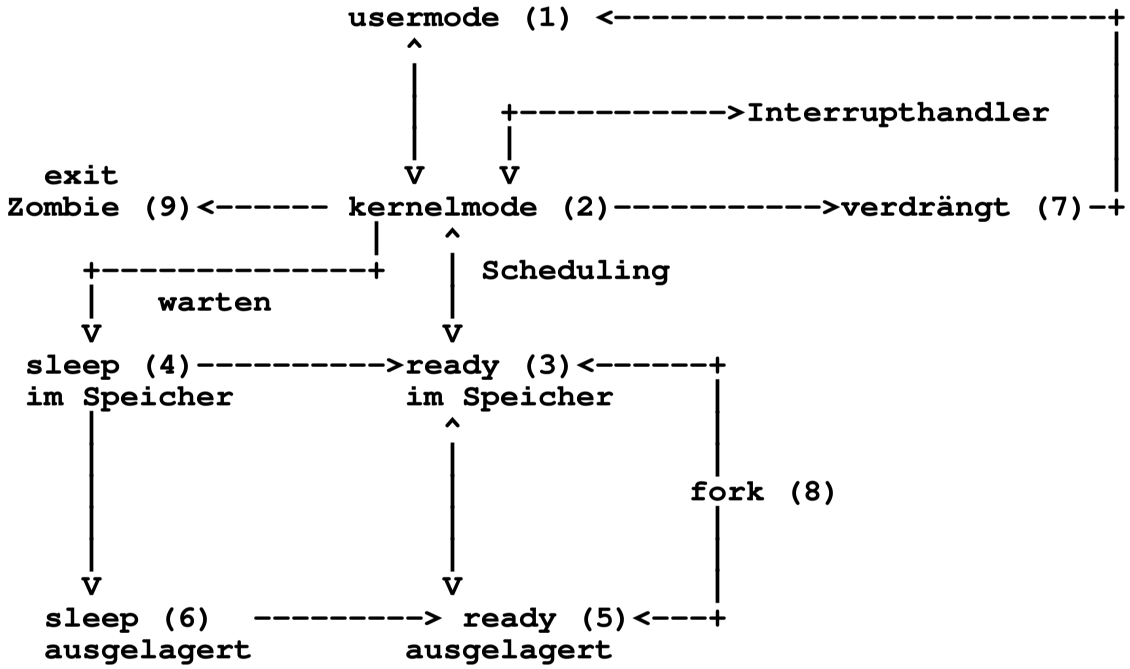


Abbildung: Prozesszustände und mögliche Übergänge

1. Prozess arbeitet im "user mode". Kein Zugriff auf systeminterne Daten. Jederzeit unterbrechbar.
2. Prozess arbeitet im "kernel mode". Zugriff auf systeminterne Daten. Nicht unterbrechbar!?
3. Prozess ist bereit. Wartet auf Prozessorzuteilung.
4. Prozess im HS, schläft weil er auf ein Ereignis wartet.
5. Prozes ist bereit weiter zu arbeiten, aber ist ausgelagert. wartet auf Einlagerungen durch Swapper.
6. Prozess wartet auf Ereignis und der Swapper hat ihn aus dem HS entfernt (auf Platte)
7. Prozess wurde verdrängt (Zeitgeberinterrupt)
8. Prozess wurde durch fork() erzeugt
9. Prozess hat einen Systemruf exit ausgeführt und wartet auf die Beendigung(Zombi).

Prozess-Scheduling

Früher: Kein Problem - Batchjobs, mehrere Partitionen, Partitionen mit fester Priorität

Jetzt: Problematisch - Kombinationen von Batchjobs mit Dialogprozessen, Timesharing

Problem: Wer darf wann den Prozessor wie lange benutzen?

Kriterien:

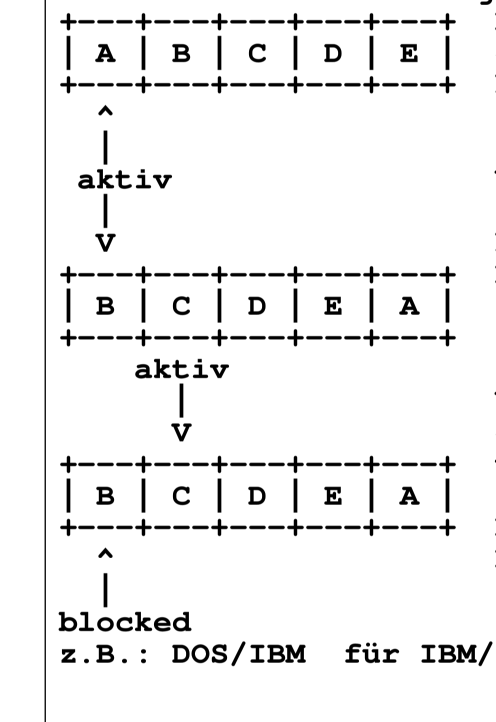
1. Fairplay: Sichern, dass jeder Prozess einmal die CPU bekommt.
2. Effektivität: Auslastung der CPU mit 100%.
3. Anwortzeitverhalten: Minimieren der Antwortzeiten für interaktive Prozessen.
4. Durchsatz: Maximum von Batchjobs pro Stunde.
5. Verweilzeit: Minimieren der Zeit, die der Batchjob im Rechner benötigt.

Schedulingverfahren

1. Round Robin Scheduling
2. Mehrfach Schlangen (multiple queues)
3. Shortest Job First
4. Zeit-Überwachte-Steuerung (apriori scheduling)
5. Zweistufiges Scheduling

1. Round Robin Scheduling

alt, einfach, faire, oft benutzt
Auf einer Prozesschlange basierend.



Der am weitesten vorn stehende Prozess erhält die Steuerung. Alle Prozesse haben gleiche Zeitquanten.

Wenn das Zeitquantum des aktiven Prozesses abgelaufen ist, wird er vorn entfernt und hinten wieder angestellt. Alle anderen Prozesse rücken einen Platz nach vor.

Wird der aktive Prozess blockiert ohne das sein Zeitquantum abgelaufen ist, wird der nächste Prozess in der Schlange aktiv. Der blockierte Prozess bleibt vorn. Er wird wieder aktiv, sobald die Blockierung nicht mehr vorliegt.

z.B.: DOS/IBM für IBM/360 - 1968

Interessante Frage: Wie gross ist das Zeitquantum zu wählen?

Prozesswechselzeit!!!!!!!! z.B. 2 ms

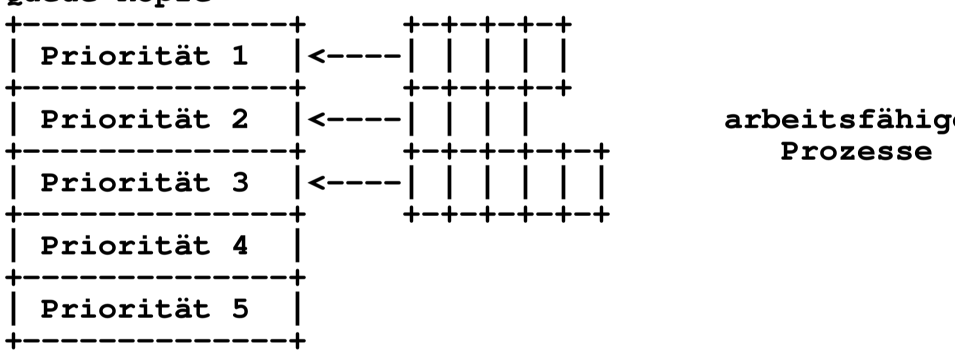
Quantum	% Verwaltungszeit	Wartezeit bei 5 Prozessen
8 ms	20%	32 ms
18 ms	10%	72 ms
98 ms	2%	392 ms
5 sec	0%	20 sec

--> Je kürzer die Quanten - je geringer die Effektivität
Je grösser die Quanten - je grösser die Reaktionszeit
übliche Zeitquanten: 100 - 200 ms

2. Mehrfach Schlangen (multiple queues)

Problem: Viel Zeit wurde für das Ein-/Auslagern von Prozessen verbraucht. Es war sinnvoll CPU-intensive Prozesse länger im Speicher zu halten und grössere Quanten zu verteilen.

Queue-Köpfe



Priorität	Quanten
1	1
2	2
3	4
4	8
5	16

Wenn ein Prozess sein Quantum verbraucht hat, wird er in die folgende Prioritätenklasse eingeordnet. Dadurch wird die Zahl der Swappvorgänge stark verringert gegenüber "round robin". Kurze Jobs werden stark bevorteilt.

Problem: Prozesse die abwechselnd interaktiv und CPU-intensiv sind werden bezüglich ihrer Reaktionszeit immer langsamer.

Lösung: Bei einer Terminaleingabe wird die Priorität auf 1 gesetzt.

Kluge Leute können diesen Algorithmus "austricksen" und damit andere Leute ernsthaft behindern.

```
3. Shortest Job First
---------------------
Prinzip für Batch-Job-Systeme
Der kürzeste Job wird als erster realisiert.
Beispiel:
     Job A: 8 min (a)           Job B: 4 min (b)
     Job C: 4 min (c)           Job D: 4 min (d)
Jobreihefolge: B C D A
interessant: mittlere Verweilzeit eines Jobs???
     (4*b + 3*c + 2*d + a) / 4 =
     ( 16 +  12 +   8 + 8) / 4 = 11
Dieses Verfahren ist auch gut für interaktive Prozesse geeignet,
wenn jede Kommandoausführung ein Prozess (Job) ist.
Problem: Woher weiss man vorher, welches der kürzeste Prozess ist????
Lösung: Schätzen, Messen und Rechnen (gewichtete Summen)
         innerhalb von Takten
          Prozess erhält  S0 sec im 0.Takt
          Prozess  verbraucht T0 sec im 0.Takt
          Prozess erhält S1 = T0 sec im 1. Takt
          Prozess verbraucht T1 sec im 1.Takt
          Prozess erhält S2 = a*T0 + (1-a)*T1  sec im 2.Takt
   usw.       Sn = a*Tn-2   +  (1-a)*Tn-1  sec im n.Takt.
      für a = 1/2  ergeben sich folgende Schätzungen:
     S1 = T0
     S2 = T0/2  + T1/2
     S3 = T0/4  + T1/4  + T2/2
     S4 = T0/8  + T1/8  + T2/4 + T3/2
     S5 = T0/16 + T1/16 + T2/8 + T3/4 + T4/2
    Damit wird die Vergangenheit eines Prozesses unterdrückt.
    a= 1/2 erweist sich damit als günstig (auch gut berechenbar)
```



```
4. Zeit-Überwachte-Steuerung (apriori scheduling)
-------------------------------------------------
Ziel: gleichverteilte CPU-Zeit über die gesamte Sitzung
      eines Nutzers
dazu notwendig: Login-Zeit, Anzahl der Nutzer, bisher
                verbrauchte Zeit

Prioritätsbestimmung:
                   verbrauchte CPU-Zeit
Priorität  = ----------------------------
                      Login-Zeit
                   -----------------
                   Anzahl der Nutzer

   0.5 - Prozess verbrauchte weniger als ihm zustand,
         hohe Priorität
   2.0 - Prozess verbrauchte mehr als ihm zustand,
         niedrige Priorität.
Der Prozess mit dem kleinsten Quotienten erhält die höchste
Priorität.
```

5. Zweistufiges Scheduling

Problem: Nicht alle Prozesse die arbeiten wollen, haben Platz im Hauptspeicher.

---> Prozesse müssen durch einen Scheduler geswappt werden.

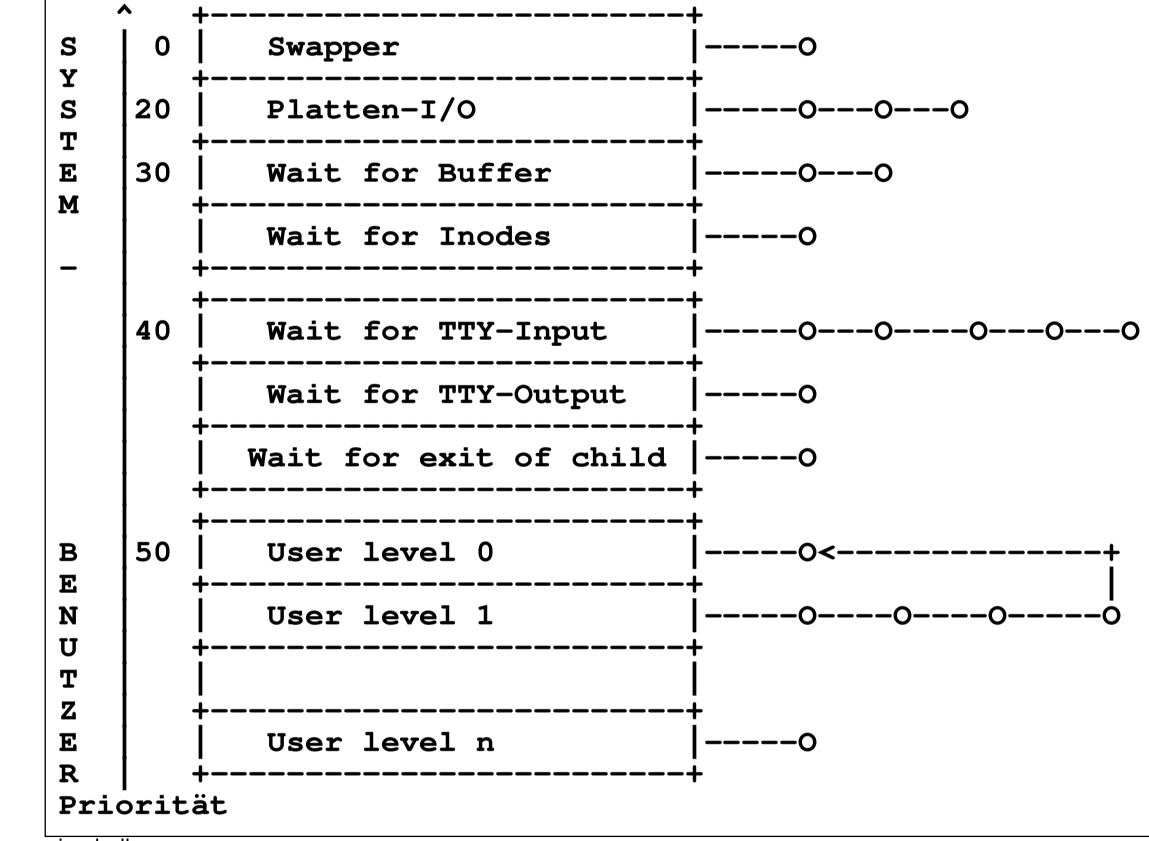
----> es existieren zwei Scheduler:

a) für Prioritätssteuerung der Prozesse im HS
   (verwaltet kleine Quanten)
   realisiert Strategien 1. bis 4.

b) für das Swappen
   (verwaltet grosse Quanten)
   Kriterien für Swapping:
   1. Wie lange ist der Prozess im HS bzw.Platte?
   2. Wieviel CPU-Zeit hat der Prozess verbraucht?
   3. Wie gross ist der Prozess?
   4. Wie hoch ist die Priorität des Prozesses?

Scheduling unter UNIX

multilevel Feedback-Queue



Prioritätsberechnung:

- **p_cpu** – Schätzung der aktuellen CPU-Auslastung des Prozesses
- **p_nice** – Nutzerspezifischer Wichtungsfaktor
  -20<=p_nice<=20 – negative Werte steigern die Priorität
- **PUSER** – Prioritätskonstante, die garantiert, dass die Nutzerpriorität nicht höher ist als die niedrigste Systempriorität
- p_userpri = PUSER + p_cpu / 4 + 2 * p_nice
  Diese Berechnung erfolgt in gewissen Zeitabständen
  Bei jeder Berechnung wird p_cpu halbiert

Beispiel ohne Nice, PUSER=60:

Intervall	Prozess A nice=0		Prozess B nice=0		Prozess C nice=0		
	60		60		60		PUSER+2*nice
	userpri	cpu	userpri	cpu	userpri	cpu	
0	60	0	60	0	60	0	
	x	60		0		0	
1	75	30	60	0	60	0	
		0	x	60		0	
2	67,5	15	75	30	60	0	
		0		0	x	60	
3	63,75	7,5	67,5	15	75	30	
	x	60		0		0	
4	76,9	33,6	63,75	7,5	67,5	15	
		0	x	60		0	
5	68,4	16,8	76,9	33,6	63,75	7,5	
		0		0	x	60	

Prozessreihenfolge: A B C A B C

p_userpri = PUSER + p_cpu / 4 + 2 * p_nice

Beispiel mit Nice, PUSER=60:

Intervall	Prozess A nice=0		Prozess B nice=-10		Prozess C nice=5		
	60		40		70		PUSER+2*nice
	userpri	cpu	userpri	cpu	userpri	cpu	
0	60	0	40	0	70	0	
		0		x 60		0	
1	60	0	55	30	70	0	
		0		x 60		0	
2	60	0	62,5	45	70	0	
	x	60		0		0	
3	75	30	51,25	22,5	70	0	
		0		x 60		0	
4	67,5	15	60,625	41,25	70	0	
		0		x 60		0	
5	63,75	7,5	65,312	50,625	70	0	
	x	60		0		0	

Prozessreihenfolge B B A B B A

1.2 Systemrufe zur Prozessteuerung

Prozessidentifikation

Erzeugung eines neuen Prozesses

Prozessbeendigung

Programmaufruf

Änderung der Identität eines Prozesses

1.2.1 Prozessidentifikation

Jeder Prozess hat eine eindeutige Prozessnummer (nicht negative Integerzahl)
Bestimmte Prozesse haben feste Prozessnummern.
z.B.
0 – sched oder swapper
1 – init (Elternprozess aller Prozesse)
Prozessnummer muss man für verschiedene Aktionen wissen!!!

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

pid_t getpid(void)
            Gibt die Nummer des eigenen Prozesses zurück.

pid_t getppid(void)
            Gibt die Nummer des Elternprozesses zurück.

uid_t getuid(void)
            Gibt den wirklichen (real) UID des Prozesses zurück.

uid_t geteuid(void)
            Gibt den effektive UID des Prozesses zurück.

gid_t getgid(void)
            Gibt den wirklich (real) GID des Prozesses zurück.

gid_t getegid(void)
            Gibt den effektiven GID des Prozesses zurück.
```

Aktueller Wertebereich für pid,uid und gid

Linux 2.4
    pid 0 .. 32.000
    gid,uid: 0 .. 65.000

Linux 2.6
    pid 0 .. 1.000.000.000
    gid,uid: 0 .. 4.000.000.000

Solaris 2.8
    pid 0 .. 30.000
    gid,uid 0 .. 2.147.483.647

Beispiele getuid, geteuid

   Prozesse/p1.c
   Prozesse/p2.c

1.2.2 Erzeugung eines Prozesses

```
#include <sys/types.h>    für System V
#include <unistd.h>

pid_t fork(void)
```

Erzeugen eines neuen Prozesses, der in all seinen Eigenschaften
und Zugriffsrechten dem alten Prozess entspricht (Kopie des alten)
Achtung bei der Benutzung von Threads: Hier verhält sich fork
je nach benutzter Thread-Art (SOLARIS, POSIX) unterschiedlich.

Unterschiede zwischen Eltern- und Kindprozess:

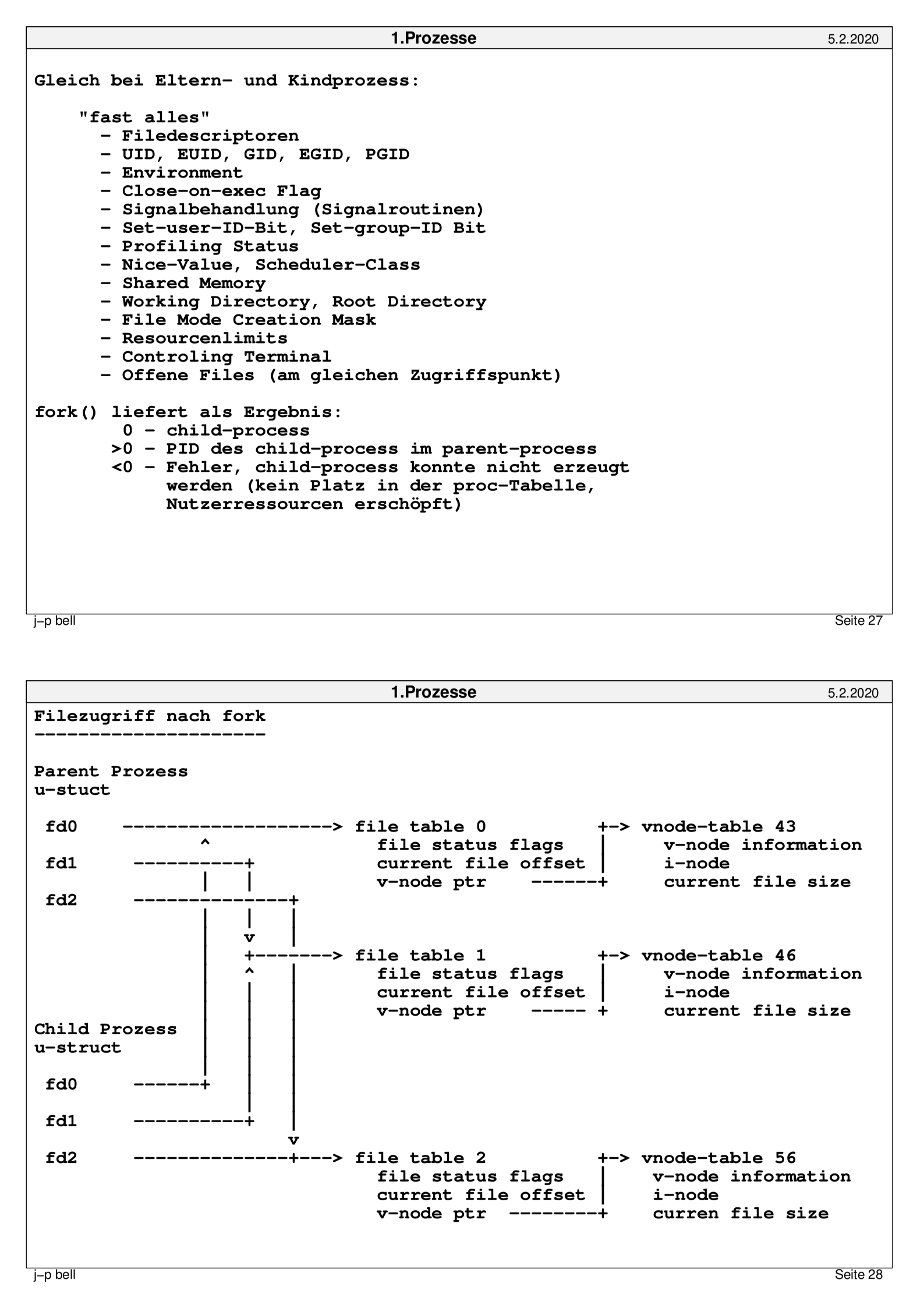
- Rückkehrkode des child-processes ist 0
  Rückkehrkode des parentprocess ist PID des
  child-process
- child-process hat andere PID und andere PPID
- eigenes Stack-Segment, eigenes Daten-Segment
- child-process:
  - prozesspezifische Zeitangaben auf 0
  - Semaphoreoperation clears
  - Process-Locks, Text-Segment-Locks
    und File-Locks aufgehoben.
  - keine hängenden Signale
- eventuell Threads

Gleich bei Eltern- und Kindprozess:

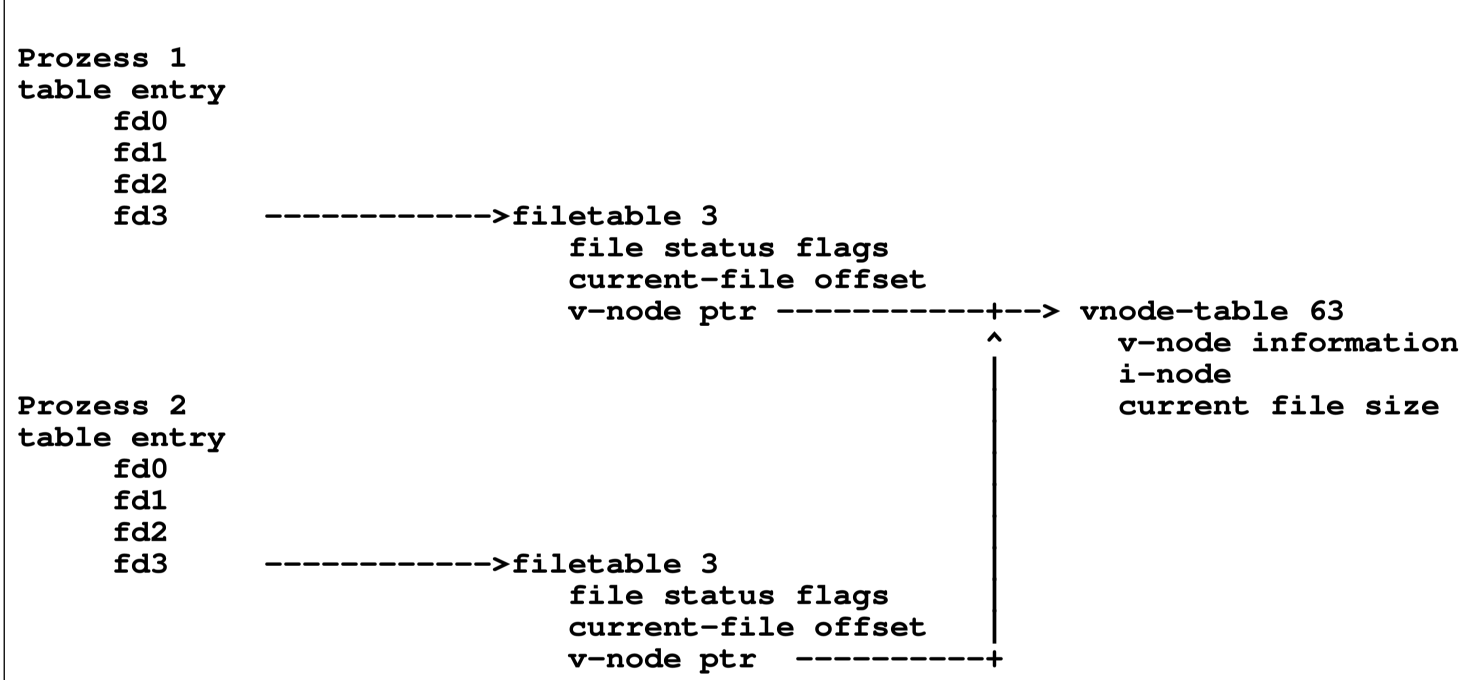
"fast alles"
- Filedescriptoren
- UID, EUID, GID, EGID, PGID
- Environment
- Close-on-exec Flag
- Signalbehandlung (Signalroutinen)
- Set-user-ID-Bit, Set-group-ID Bit
- Profiling Status
- Nice-Value, Scheduler-Class
- Shared Memory
- Working Directory, Root Directory
- File Mode Creation Mask
- Resourcenlimits
- Controling Terminal
- Offene Files (am gleichen Zugriffspunkt)

fork() liefert als Ergebnis:
- 0 - child-process
- >0 - PID des child-process im parent-process
- <0 - Fehler, child-process konnte nicht erzeugt werden (kein Platz in der proc-Tabelle, Nutzerressourcen erschöpft)

Filezugriff nach fork



Filezugriff von Prozessen auf ein File ohne Vererbung




```
#include <sys/types.h>    für System V
#include <unistd.h>

pid_t int fork1(void)

Erzeugen eines neuen Prozesses.  Funktionalität analog    fork().
Neuer Systemruf nur für Solaris. Variante von fork().
  bis Solaris10: -lthread:  alle Threads werden gedoppelt.
                 -lpthread: nur der aktuelle Thread wird gedoppelt.
  ab Solaris10:  nur der aktuell Thread wird gedoppelt.

Rückkehrkode: wie fork()


-------------------------------------------------------------


#include <sys/typedef.h>

pid_t int vfork(void)

Erzeugen eines neuen Prozesses.  Funktionalität analog    fork().
Aber ohne Kopie des Datensegmentes und des Stacksegmentes.
Kindprozess benutzt Daten- und Stacksegment des Elternprozesses.
Seiteneffekte, wenn Kindprozess mehr als EXEC macht!!!!!

Rückkehrkode: wie fork()
```

```
#include <sys/types.h>    für System V
#include <unistd.h>

pid_t int forkall(void)

Erzeugen eines neuen Prozesses.  Funktionalität analog    fork().
Sehr neuer Systemruf für Solaris 10. Es werden alle momentan
laufenden Threads kopiert.

Rückkehrkode: wie fork()

------------------------------------------------------------

Verhalten von fork() bei der Benutzung von Threads:

                  fork       vfork       fork1       forkall
Linux             a,d+s       a            -           -
Solaris8/9
    SOLARIS       a,d+s       t          t,d+s         -       (-lthread)
    POSIX         t,d+s       t          t,d+s                 (-lpthread)
Solaris10         t,d+s       t          t,d+s       a,d+s


Beispiele für fork und vfork:
-----------------------------
      Prozesse/fork1.c
      Prozesse/fork2.c
      Prozesse/vfork1.c, Prozesse/fork-test.c, Prozesse/forkv-test.c
```



```
       1.2.3 Prozessbeendigung
       -----------------------

Arten der Prozessbeendigung:

   1. Normales Prozessende

         a) return im Hauptprogramm ----> exit()

         b) Aufruf von exit
            - Abarbeitung von mit
                     int atexit(void (*func)(void))
              definierten privaten exit-Routinen (ANSI-C)
            - E/A-Endebehandlungen
            - Aufruf von _exit

         c) _exit() Aufruf
            UNIX-spezifische Prozessendebehandlung

   2. Abnormales Prozessende

          a) Aufruf der Funktion abort
                       void abort(void)
             diese Funktion erzeugt ein Signal SIGABRT
             (POSIX.1, ANSI C) siehe Stevens: Program 10.18

          b) Ende über Signalbehandlung
```

```
#include <stdlib.h>

void exit(int exitcode)            ANSI-C   libc

#include <unistd.h>

void _exit(int exitcode)           POSIX.1  Systemruf

Reguläres beenden eines Prozesses.

   Die niederwertigen 8 Bits von "exitcode" werden als
   Resultat  an den Parentprozess übergeben.
   Dateien werden abgeschlossen.

Probleme bei der Übergabe des Resultats:

  a) Parentprozess wartet mit wait() ---> kein Problem
  b) Parentprozess wartet nicht:
     a) Parentprozess existiert nicht:
          Resultat wird an init-Prozess übergeben.
     b) Parentprozess existiert noch:
          Resultat wird in der proc-Tabelle gespeichert.
          Alle anderen Informationen über den Prozess
          werden gestrichen.  Es entsteht ein "Zombie-Prozess".
          Dieser wird gestrichen wenn der Parentprozess
          das Resultat mit wait() abholt.

Nebenwirkungen: SIGHUB für alle Prozesse der Prozessgruppe
```



```
Beispiele für exit:
-------------------
    Prozesse/myexit.c
    Prozesse/myexit1.c
    Prozesse/myexit2.c
```

Warten auf Prozessende eines Kindprozesses

Wenn ein Prozess beendet oder gestoppt wird, wird ein Terminationkode gebildet. Dieser Kode beinhaltet den Exitkode bzw. die Signalnummer, die das Beenden bzw. das Stoppen des Kindprozesses bewirkte.
Der Kode wird in der proc-Table gespeichert.

Systemrufe:

```
pid_t wait(int *statloc)   POSIX.1, SVR4, BSD 4.3

pid_t waitpid(pid_t pid, int *statloc, int options)
                                    POSIX.1, SVR4, BSD 4.3
```

Warten auf das Ende oder das Stoppen (Stop-Signal) eines Kindprozesses des aktuellen Prozesse.


```
#include <sys/types.h>
#include <sys/wait.h>

pid_t wait(int *statloc))
```

Wartet auf das erste Ende eines beliebigen Kindprozesses, sofern der aktuelle Prozess einen Kindprozess besitzt.
*statloc enthält den Terminationkode, wenn statloc != NULL

Terminationkode: <high order 8 bits> <low order 8 bits>

- Kindprozess durch Signal gestoppt: <Signalnummer><0x7F>
- exit, _exit: <niederwertigen 8 Bit des Exitkodes><0x00>
- Kindprozess durch Signal abgebrochen: <0x00><Signalnummer>
- Kindprozess durch Signal abgebrochen und Dump erzeugt: <0x00><Signalnummer + 0x80>

Rückkehrkode:

- >=0 - Prozessnummer des Kindprozesses
- < 0 - Fehler, kein Kindprozess vorhanden oder falscher Parameter

```
#include <sys/types.h>
#include <sys/wait.h>

pid_t waitpid(pid_t pid, int *statloc, int options)

Bedingtes warten auf das Prozessende eines spezifizierten
Prozesses.

 pid  >  0 - warten auf das Ende des Prozesses mit der
             PID == pid
 pid == -1 - warten auf das Ende eines Kindprozesse (entspricht wait())
 pid ==  0 - warten auf das Ende eines Kindprozesses der eigenen
             Prozessgruppe
 pid  < -1 - warten auf jeden Prozess mit der
             ProzessengruppenID==|pid|

 *statloc enthält den Terminationkode (siehe wait())
 options:
        WCONTINUED - warten auf das Ereignis
        WNOHANG    - auf das Ereignis nicht warten
        WUNTRACED  - nur gestoppte Prozesse melden, die noch
                     nicht gemeldet wurden
Rückkehrkode:
      >=0 - Prozessnummer des Kindprozesses
      < 0 - Fehler, kein Kindprozess vorhanden oder falscher
            Parameter
```



```
Weiter Systemrufe für das Warten auf Kindprozesse
-------------------------------------------------

Voraussetzungen:
   #include <sys/types.h>
   #include <sys/wait.h>
   #include <sys/time.h>
   #include <sys/resource.h>

pid_t wait3(int *statloc, int options, struct rusage *rusage)
                                                SVR4, BSD 4.3

pid_t wait4(pid_t pid, int *statloc, int options,
            struct rusage *rusage)              BSD 4.3

*rusage: Ressourcenangaben über den beendeten oder gestoppten
         Prozess (CPU-Zeit, Hauptspeicherbedarf, Seitenfehler,
         Signale, ...)

wait3()
   entspricht wait() unter der Berücksichtigung von options.

wait4()
   entspricht waitpid().
```

1.2.4 Programmausführung

exec() umfasst eine Familie von Systemrufen, die zum Laden und Starten eines neuen Programms innerhalb eines Prozesses dient.

Grundfunktionen:

- exec() überlagert im aufrufenden Prozess das aktuelle Codesegment und das Datensegment.
- Das Stacksegment wird zurückgesetzt
- Eröffnete Dateien bleiben eröffnet.
- Ignorierte Signale bleiben ignoriert.
- Alle anderen Signale werden zurückgesetzt.

Folgende prozesspezifischen Werte bleiben unverändert:

nice-Wert	Schedulerwerte	Filecreationmask
PID	Semaphore	Ressourcelimits
PPID	Session ID	Controlterminal
PGID	traceflag	Prozesssignalmaske
Alarmzeit	Workingdirectory	hängende Signale
Prozesszeit	Rootdirectory	

Rückgabewerte:
- \- keiner, wenn alles ok
- <0 bei Fehlern

EXEC-Systemrufe:

```
#include <unistd.h>

int execl(const char *pathname, const char *arg0,...(char *) 0);

        Feste Anzahl von Parametern

int execv(const char *pathname, char *const argv[]);

        variable Anzahl von Parametern

int execle(const char *pathname, const char *arg0, ...,
                        (char *) 0, char *const envp[]);
        wie execl, aber mit setzen der Umgebungsvariablen

int execve(const char *pathname, char *const argv[],
                                    char *const envp[]);
        wie execv, aber mit setzen der Umgebungsvariablen

int execlp(const char *filename, const char *arg0,...,
                                              (char *) 0);
        wie execl, aber Programm wird gesucht

int execvp(const char *filename, char *const argv[]);

        wie execv, aber Programm wird gesucht
```

Beispiel für exec: **exec1.c, echoall.c**

- Prozesse/echoall.c
- Prozesse/exec1.c
- Prozesse/pwd-test.c

1.2.5 Änderung von ID's von Prozessen

Folgende ID's sind änderbar:

UID	– User ID
SUID	– Saved User ID
GID	– Group ID
EUID	– Effective User ID
EGID	– Effective Group ID
PGID	– Processgroup ID
SID	– Session ID

Systemrufe zum Abfragen von ID's:

getuid(), geteuid(),
getgid(), getegid(),
getpid(), getppid(),
getsid(), getprgp()

Systemrufe zum Ändern von ID's:

setuid(), seteuid(), setreuid(),
setgid(), setegid(), setregid(),
setpgid(), setpgrp(), setsid()

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

int setuid(uid_t uid);

           Setzen des UID's für den aktuellen Prozess.

int setgid(gid_t gid);

           Setzen des GID's für den aktuellen Prozess.

    setuid() wirden unter folgenden Bedingungen erfolgreich
    abgearbeitet:

      1. aktueller UID == 0 (root):
                 setuid() setzt aktuellen UID auf uid und EUID auf uid.
      2. aktueller UID != 0 und (uid == UID oder uid == SUID):
                 setuid() setzt EUID auf uid.
    setgid() arbeitet analog.

    Rückkehrkode:
         0 - ok
        <0 - Fehler
```



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

int seteuid(uid_t uid);

       Setzen des effektiven UID's für den aktuellen Prozess.

int setegid(gid_t gid);

       Setzen des effektiven GID's für den aktuellen Prozess.

    seteuid() wirden unter folgenden Bedingungen erfolgreich
    abgearbeitet:

      1. aktueller UID == 0 (root):
                 seteuid() setzt aktuellen EUID auf uid.
      2. aktueller UID != 0 und (uid == UID oder uid == SUID):
                 seteuid() setzt EUID auf uid.
    setegid() arbeitet analog.

    Rückkehrkode:

            0 - ok
           <0 - Fehler
```

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

int setreuid(uid_t ruid, uid_t euid);          BSD 4.3

       Setzen von UID und EUID.

int setregid(gid_t rgid, gid_t egid);          BSD 4.3

       Setzen von GID und EGID.

    Der Superuser kann sowohl uid und euid gleichzeitig setzen.
    Der normale User kann lediglich einen Wechsel zwischen EUID
    und UID veranlassen. Hiermit können Programme, die mit S-Bit
    laufen zwischen priveligierten Modus und User-Modus hin- und
    herschalten.

    Rückkehrkode:

      0 - ok
     <0 - Fehler
```



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

pid_t setpgid(pid_t pid, pid_t pgid)

         Setzen des Pozessgruppen ID's für den Prozess pid auf den
         Wert pgid. Wenn pid==0, ist der aktuelle Porzess gemeint. Wenn
         pgid==0, wird der PGID des mit pid spezifierten Prozess benutzt.
         Prozessgruppen werden für Signalverteilung und zur Synchronisation
         des Zugriffs auf das Controling Terminal benuzt.

         Rückkehrkode:   0 - ok
                         <0 - Fehler

pid_t setpgrp(void)

         Der aktuelle Prozess wird Prozessgruppenführer. Der Prozess-
         gruppen ID und der aktuelle Session ID ergeben sich aus dem
         PID des aktuellen Prozesses. Identisch mit setpgid(0,0).

         Rückkehrkode: Nummer der aktuellen Prozessgruppe

pid_t setsid(void)

         Der aktuelle Prozess wird Gruppenführer einer neuen Session.
         Prozessgruppe ohne Controling Terminal.
         Der aktuelle SID ergibt sich aus dem aktuellen PID.

         Rückkehrkode: Nummer der aktuellen Sessiongroup
```

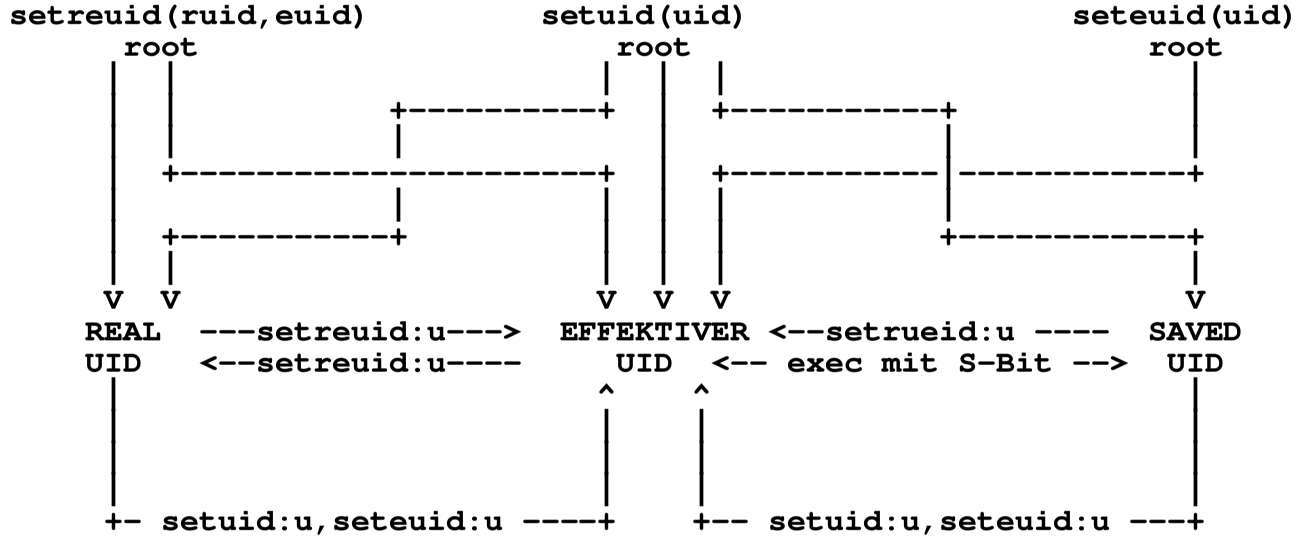
Übersichten

Ändern der User-IDs (UID, EUID, SUID) bei exec() und setuid()

ID	exec		setuid(nuid)	
	S-Bit=0	S-Bit=1	superuser	user
UID	-	-	nuid	-
EUID	-	ID vom Programm	nuid	nuid
SUID	EUID	UID	nuid	-

UID - Wirklicher UID (Real UID)
EUID - Effektiver UID, bestimmt Zugriffsrechte
SUID - Geretteter EUID

Funktionen, die unterschiedliche User-IDs setzen



:u - normaler Nutzer

Beispiel getuid, geteuid:

Prozesse/changeuid.c
Prozesse/pruids.c
Prozesse/setid.c
Prozesse/setidn.c