UNIX API - Prozesse

Alle Beispiel-Quellen mittels SVN unter:

https://svn.informatik.hu-berlin.de/svn/unix-2014/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
```

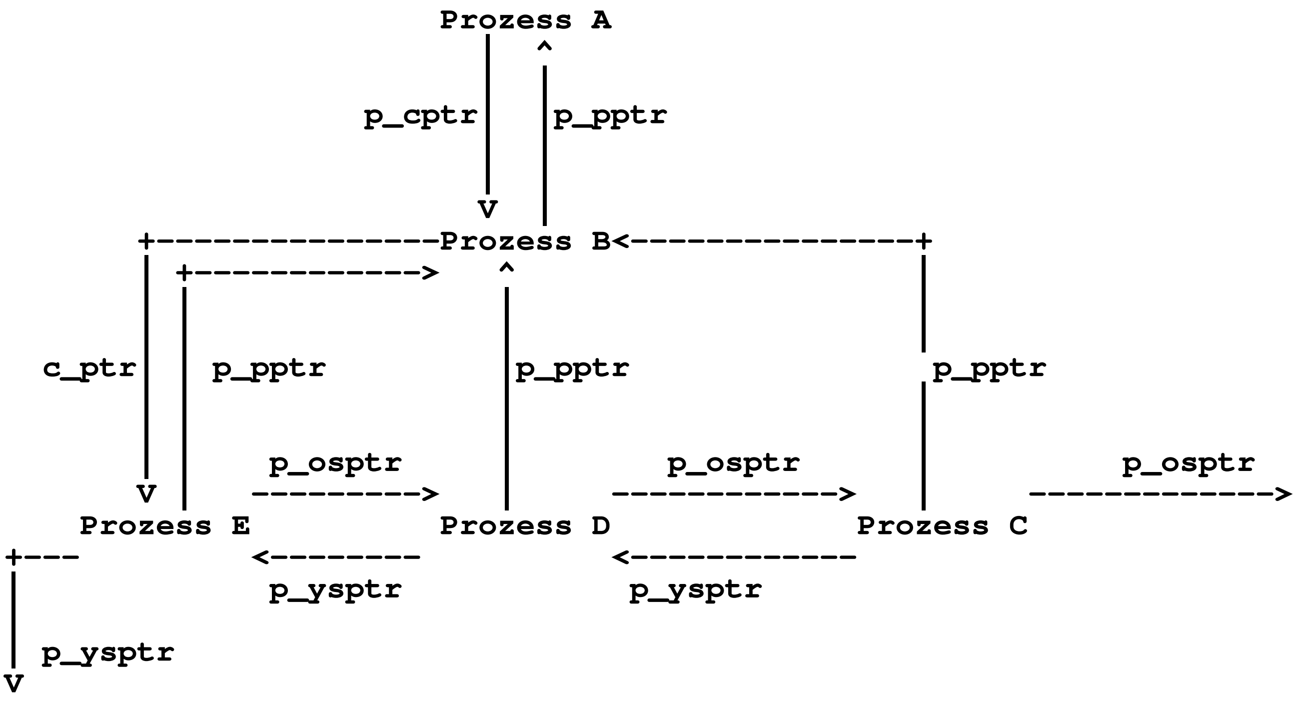

Abbildung: Hierarchie der Prozess-Gruppen

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 Prozesse im UNIX
------------------------------------

                     usermode (1) <-------------------------+
                        ^                                   |
                        |                                   |
                        |    +----------->Interrupthandler  |
                        |    |                              |
   exit                 V    V                              |
  Zombie (9)<------ kernelmode (2)----------->verdrängt (7)-+
                         |    ^
      +------------------+    | Scheduling
      |  warten               |
      V                       V
   sleep (4)---------->ready (3)<------+
   im Speicher         im Speicher     |
      |                   ^            |
      |                   |            |
      |                   |          fork (8)
      |                   |            |
      V                   V            |
    sleep (6)  ---------> ready (5)<---+
    ausgelagert         ausgelagert


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 unter UNIX
-----------------------------
multilevel Feedback-Queue
         Prioritätsebene         Prozesse
   ^   +-------------------------+
S  | 0 |   Swapper               |-----O
Y  |   +-------------------------+
S  |20 |   Platten-I/O           |-----O---O---O
T  |   +-------------------------+
E  |30 |   Wait for Buffer       |-----O---O
M  |   +-------------------------+
   |   |   Wait for Inodes       |-----O
-  |   +-------------------------+
   |   +-------------------------+
   |40 |   Wait for TTY-Input    |-----O---O----O---O---O
   |   +-------------------------+
   |   |   Wait for TTY-Output   |-----O
   |   +-------------------------+
   |   |  Wait for exit of child |-----O
   |   +-------------------------+
   |   +-------------------------+
B  |50 |   User level 0          |-----O<--------------+
E  |   +-------------------------+                     |
N  |   |   User level 1          |-----O----O----O-----O
U  |   +-------------------------+
T  |   |                         |
Z  |   +-------------------------+
E  |   |   User level n          |-----O
R  |   +-------------------------+
Priorität
```



```
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         Prozess B        Prozess C
             nice=0            nice=0           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        Prozess B       Prozess C
            nice=0            nice=-10        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
```



```
   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


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

int main()
{

    printf(" p1-Kommando\n");
    printf("getpid:  %d\n",getpid());
    printf("getepid: %d\n",getppid());
    printf("getuid:  %d\n",getuid());
    printf("geteuid: %d\n",geteuid());
    printf("getgid:  %d\n",getgid());
    printf("getegid: %d\n",getegid());
    exit(0);
}

    ./p1
    ./p11  - S-BIT
```



```
   Prozesse/p2.c

#include <unistd.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
int main()
{
    char buffer[100];

    write(1," p2-Kommando\n",13);
    sprintf(buffer,"getpid:  %d\n",getpid());
    write(1,buffer,strlen(buffer));
    sprintf(buffer,"getppid:  %d\n",getppid());
    write(1,buffer,strlen(buffer));
    sprintf(buffer,"getuid:  %d\n",getuid());
    write(1,buffer,strlen(buffer));
    sprintf(buffer,"geteuid:  %d\n",geteuid());
    write(1,buffer,strlen(buffer));
    sprintf(buffer,"getgid:  %d\n",getgid());
    write(1,buffer,strlen(buffer));
    sprintf(buffer,"getegid:  %d\n",getegid());
    write(1,buffer,strlen(buffer));
    exit(0);
}

   ./p2
```

```
          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
---------------------

Parent Prozess
u-stuct

 fd0    -------------------> file table 0         +-> vnode-table 43
              ^                file status flags  |     v-node information
 fd1     ----------+           current file offset|     i-node
              |    |           v-node ptr    ------+     current file size
 fd2     --------------+
              |    |   |
              |    v   |
              |    +-------> file table 1         +-> vnode-table 46
              |    ^   |       file status flags  |     v-node information
              |    |   |       current file offset|     i-node
              |    |   |       v-node ptr    ----- +     current file size
Child Prozess |    |   |
u-struct      |    |   |
              |    |   |
 fd0     ------+    |   |
                   |   |
 fd1     ----------+   |
                       v
 fd2     --------------+---> file table 2         +-> vnode-table 56
                               file status flags  |    v-node information
                               current file offset|    i-node
                               v-node ptr  --------+    curren file size
```



```
Filezugriff von Prozessen auf ein File ohne Vererbung
-----------------------------------------------------

Prozess 1
table entry
     fd0
     fd1
     fd2
     fd3     ------------>filetable 3
                             file status flags
                             current-file offset
                             v-node ptr -----------+--> vnode-table 63
                                                   ^      v-node information
                                                   |      i-node
Prozess 2                                          |      current file size
table entry                                        |
     fd0                                           |
     fd1                                           |
     fd2                                           |
     fd3     ------------>filetable 3              |
                             file status flags     |
                             current-file offset   |
                             v-node ptr  ----------+
```

```
               fork1.c

#include <sys/types.h>
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
int   glob1 = 1;   /* externe Variable im initialisierten Datensegment */
char  buf[] = "\nStart von fork1\n\n";
int main()
{
  int   var;    /* lokale Variable im Stacksegment */
  pid_t pid; var = 1;
  if (write(1, buf, sizeof(buf)-1) != sizeof(buf)-1) exit(-1);
  printf("vor fork: pid = %d, glob1 = %d, var = %d\n\n",
               getpid(), glob1, var);
  if ( (pid = fork()) < 0)
        exit(-2);
  else if (pid == 0) {  /* im Kindprozess */
        glob1++;                /* Variablen modifizieren */
        var++;
  } else
        sleep(2);       /* im Elternprozess */
  printf("nach fork: pid = %d, glob1 = %d, var = %d\n\n",
               getpid(), glob1, var);
  exit(0);
}

   ./fork1
   ./fork1 > xxx
   cat xxx          # ?????
```



```
                fork2.c

#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
int   glob1 = 1;   /* external variable in initialized data */
char  buf[] = "\nSchreiben auf stdout\n\n";
int main()
{
  int   var,fd;    /* lokale Variable im Stacksegment */
  pid_t pid; var = 1;
  if (write(1, buf, sizeof(buf)-1) != sizeof(buf)-1) exit(-1);
  printf("\nvor fork: pid = %d, glob1 = %d, var = %d\n\n",
                        getpid(), glob1, var);
  fd=open("test-file-vorlesung",O_RDWR+O_CREAT,0644);
  if ( (pid = fork()) < 0) exit(-2);
  else if (pid == 0) {  /* im Kindprozess */
        glob1++;   var++; write(fd,"*******CHILD*******\n",20);
  } else {
        sleep(2);       /* im Elternprozess */
        lseek(fd,01,0);
        write(fd,"PARENT\n",7);
  }
  printf("nach fork: pid = %d, glob1 = %d, var = %d\n\n",
                       getpid(), glob1, var);
  exit(0);
}
```

```
./fork2.c
cat test-file-vorlesung

vi fork2.c    # lseek auskommentieren

./fork2
cat test-file-vorlesung
```



```
#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()
```

```
      vfork1.c     - Programm mit vfork

#include <sys/types.h>
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
int   glob1 = 1;   /* externe Variable im initialisierten Datensegment */
char  buf[] = "\nStart von fork1\n\n";
int main()
{
  int   var;    /* lokale Variable im Stacksegment */
  pid_t pid;
  var = 1;
  if (write(1, buf, sizeof(buf)-1) != sizeof(buf)-1) exit(-1);
  printf("vor vfork: pid = %d, glob1 = %d, var = %d\n\n", getpid(), glob1, va
  if ( (pid = vfork()) < 0)
        exit(-2);
  else if (pid == 0) {  /* im Kindprozess */
        glob1++;                /* Variablen modifizieren */
        var++;
  } else
        sleep(2);       /* im Elternprozess */
  printf("nach vfork: pid = %d, glob1 = %d, var = %d\n\n", getpid(), glob1, v
  exit(0);
}

     ./vfork1
```



```
   fork-test.c           /          vfork-test.c

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#define MYFORKCALL fork            #define MYFORKCALL vfork
#define XXX "fork"                 #define XXX "vfork"
#define ZKL 100000
char buff1[64000];
char buff2[64000];
char buff3[64000];
char buff4[64000];
int vglob=1;
int main()
{
  int pid,i,local;   char buf[512];
  local=1;
  printf("Test von: %s\n", XXX);
  for (i=0;i<ZKL;i++) {
    if ( (pid=MYFORKCALL()) == 0) { /*child*/
      vglob++; local++; close(1); exit(0);
    }
    if (pid < 0) {
      fprintf(stderr,"%s-Fehler nach %d %s-Zyklen\n",XXX,XXX,i);
      exit(-2);
    }
    wait(NULL);
  }
  printf("vglob: %d, local: %d\n",vglob,local);
}
```

Geschwindigkeit fork und vfork

```
time ./fork-test
time ./vfork-test
```



```
#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	

```
       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 und _exit:

myexit.c

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
void my_exit1()
   { printf("1. exit handler\n"); }
void my_exit2()
   { printf("2. exit handler\n"); }
void my_exit3()
   { printf("3. exit handler\n"); }
int main()
{ if (atexit(my_exit1) != 0) {
     fprintf(stderr,"can't register my_exit1\n"); exit(255);
  }
  if (atexit(my_exit2) != 0) {
     fprintf(stderr,"can't register my_exit2\n"); exit(254);
  }
  if (atexit(my_exit3) != 0) {
     fprintf(stderr,"can't register my_exit3\n"); exit(253);
  }
  printf("main is done\n");
  exit(0);      /*   Standard-exit, mit privaten exit-Routinen*/
  _exit(0);      /*   sofortiges Ende, keine privaten exit-Routinen*/
}

   ./myexit
   ./myexit > xxxx
   cat xxxx
      exit-Routinen tauschen und noch einmal
```


myexit1.c - mit fflush

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

void my_exit1()
   { printf("1. exit handler\n"); }
void my_exit2()
   { printf("2. exit handler\n"); }
void my_exit3()
   { printf("3. exit handler\n"); }
int main()
{ if (atexit(my_exit1) != 0) {
     fprintf(stderr,"can't register my_exit1\n"); exit(255);
  }
  if (atexit(my_exit2) != 0) {
     fprintf(stderr,"can't register my_exit2\n"); exit(254);
  }
  if (atexit(my_exit3) != 0) {
     fprintf(stderr,"can't register my_exit3\n"); exit(253);
  }
  printf("main is done\n");
  fflush(stdout);                /* leeren der Puffer */
  _exit(0);                      /* sofortiges Ende ist nicht schädlich */
}

      ./myexit1
      ./myexit1 >xxxx
      cat xxxx
```

myexit2.c – mit write

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

char mystr[100];
void my_exit1()
   { write(1,"1 exit handler\n",15); }
void my_exit2()
   { write(1,"2 exit handler\n",15); }
void my_exit3()
   { write(1,"3 exit handler\n",15); }
int main()
{ if (atexit(my_exit1) != 0) {
     strcpy(mystr,"can't register my_exit1\n");
     write(2,mystr,strlen(mystr)); exit(255);
  }
  if (atexit(my_exit2) != 0) {
     strcpy(mystr,"can't register my_exit2\n");
     write(2,mystr,strlen(mystr)); exit(254);
  }
  if (atexit(my_exit3) != 0) {
     strcpy(mystr,"can't register my_exit3\n");
     write(2,mystr,strlen(mystr)); exit(253);
  }
write(1,"main is done\n",13);    /*keine Puffer*/
 _exit(0);
}
```



```
./myexit2
./myexit2 > xxxx
cat xxxx
```

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
```

```
Beispiele für exec:
-------------------

          echoall.c - Ausgabe Parameter und Environment  (Hilfsprogramm)

#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[])
{
   int         i;
   char      **ptr;
   extern char   **environ;
   printf("-------------PID: %d----------\n",getpid());
   printf("-------------Argumente----------\n");
   for (i = 0; i < argc; i++)
           /* echo all command-line args */
      printf("argv[%d]: %s\n", i, argv[i]);
   printf("-------------Environment----------\n");
   for (ptr = environ; *ptr != 0; ptr++)
           /* and all env strings */
      printf("%s\n", *ptr);
   exit(0);
}

       ./echoall | more
       ./echoall -asdf 1324 -f --dd=5 +6 | more
```



```
   exec1.c   - Aufruf von echoall

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

char   *env_init[] = { "USER=unknown", "PATH=/tmp", NULL };

int main()
{
   pid_t   pid;

   if ( (pid = fork()) < 0) {
      fprintf(stderr,"fork error"); exit(-1);
   }
   else if (pid == 0) {   /* specify filename, inherit environment */
      printf("\n\n\nexeclp('./echoall\', ..., NULL)\n");
      if (execlp("./echoall",
                "echoall", "only 1 arg", (char *) 0) < 0){
         fprintf(stderr,"execle error\n");
         exit(-5);
      }
   }
   if (waitpid(pid, NULL, 0) < 0) {
      fprintf(stderr,"wait error\n"); exit(-2);
   }
```

```
   if ( (pid = fork()) < 0) {
      fprintf(stderr,"fork error\n");
      exit(-3);
   }
   else if (pid == 0) {   /* specify pathname, specify environment */
      printf("\n\n\nexecle('./echoall','-echoall', ....,NULL)\n");
      if (execle("./echoall",
               "-echoall", "myarg1", "MY ARG2", (char *) 0,
               env_init) < 0) {
         fprintf(stderr,"execlp error\n");
         exit(-4);
      }
   }
   if (waitpid(pid, NULL, 0) < 0) {
      fprintf(stderr,"wait error");
      exit(-2);
   }
   exit(0);
}

   ./exec1 | more
```



```
        pwd-test.c    - Aufruf von pwd aus einem Programm heraus und einlesen
                        des Ergebnisses von pwd
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main()
{
  int pid; int pipefd[2]; char buf[512];

  pipe(pipefd);
  if ( (pid=vfork()) == 0) { /*child*/
      close(pipefd[0]); close(1); dup2(pipefd[1],1);
      execl("/bin/pwd","pwd",NULL);
      exit(-1);
  }
  if (pid < 0) {
      fprintf(stderr,"no vfork\n"); exit(-2);
  }
  close(pipefd[1]);
  if ((pid=read(pipefd[0],buf,512)) <=0) {
     fprintf(stderr,"error: %d\n",pid); exit(-3);
  }
  close(pipefd[0]);
  buf[pid]=0;
  printf("\n\npwd: %s\n\n",buf);
}

    ./pwd-test
```

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

     setreuid(ruid,euid)         setuid(uid)                 seteuid(uid)
           root                      root                        root
          |  |                      |  |  |                        |
          |  |           +----------+  |  +-----------+            |
          |  |           |          |  |  |           |            |
          |  +----------------------+  |  +-----------|------------+
          |              |          |  |  |           |
          |  +-----------+          |  |  |           +------------+
          |  |                      |  |  |                        |
          V  V                      V  V  V                        V
         REAL  ---setreuid:u--->  EFFEKTIVER <--setrueid:u ----  SAVED
         UID   <--setreuid:u----     UID  <-- exec mit S-Bit -->  UID
          |                         ^    ^                         |
          |                         |    |                         |
          |                         |    |                         |
          +- setuid:u,seteuid:u ----+    +-- setuid:u,seteuid:u ---+

     :u   - normaler Nutzer
```


Beispiel getuid, geteuid:

```
-------------------------

  changeuid.c  - File erzeugen mit euid

#include  <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdlib.h>
#include <string.h>

cf(fname)
  char *fname;
{
    int fd;
    fd=open(fname,O_RDWR|O_CREAT,0644);
    write(fd,fname,strlen(fname));
    write(fd,"\n",1);
    close(fd);
}

main()
{  int uid;
   int euid;
   int res;

   uid=getuid();
   euid=geteuid();
```

```
   printf("Bei Programstart: ") ;
   printf("real uid = %d, effective uid = %d\n", uid, euid);
   printf("Erzeuge file1\n");
   cf("file1");
   res=seteuid(uid);
   if (res) printf("res setuid %d\n",res);
   printf("seteuid(%d): real uid = %d, effective uid = %d\n", uid,getuid(), g
   printf("Erzeuge file2\n");
   cf("file2");
   res=seteuid(euid);
   if (res) printf("res seteuid %d\n",res);
   printf("seteuid(%d): real uid = %d, effective uid = %d\n", euid, getuid(),
   printf("Erzeuge file3\n");
   cf("file3");
   res=seteuid(uid);
   if (res) printf("res seteuid %d\n",res);
   printf("seteuid(%d): real uid = %d, effective uid = %d\n", uid, getuid(),
   printf("Erzeuge file4\n");
   cf("file4");
   exit(0);
}

      rm file*
      ./changeuid
      ls -lisa file*
      rm file*
      ls -lisa changeuid0
      ./changeuid0   - root mit S-Bit
      ls -lisa file*
      rm file*
```


Wirkung von S-Bit bei system-Ruf und fork/exec-Ruf

pruids.c   - Ausgabe von uid und euid

```
#include  <stdio.h>
#include  <stdlib.h>
int main()
{
   printf("real uid = %d, effective uid = %d\n", getuid(), geteuid());
   exit(0);
}

      ./pruids
```

```
    setid.c    - Ausführung eines Kommandos  mittels 'system'
                  (Umschreibung von fork und exec)


#include   <stdio.h>
#include   <stdlib.h>
int main(int argc, char *argv[])
{
   int      status;
   if (argc < 2) {
      fprintf(stderr,"command-line argument required\n");
      exit(-1);
   }
   if ( (status = system(argv[1])) != 0) {
      fprintf(stderr,"system() error\n");
      exit(-2);
   }
   fprintf(stderr,"status: %d\n",status);
   exit(0);
}



      ./setid  ./pruids
      ./setid0  ./pruids      - root mit S-Bit
```



```
    setidn.c   - Ausführung eines Kommandos mittels fork und exec

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

int main(int argc, char *argv[])
{
   int      status, pid; extern int errno;
   if (argc < 2) {
      fprintf(stderr,"command-line argument required\n"); exit(-1);
   }
   fprintf(stderr,"Kommando: %s\n",argv[1]);
   if ( (pid = fork()) == 0 ) {
      if (pid < 0 ) {
           fprintf(stderr,"fork error\n"); exit(0);
      }
      execlp(argv[1],argv[1],(char *)0);
      fprintf(stderr,"execle error: %d\n",errno); exit(1);
   }
   if (pid < 0 ) {
      fprintf(stderr,"fork error\n"); exit(0);
   }
   waitpid(pid,&status,0);
   fprintf(stderr,"status: %d\n",status); exit(0);
}
```

```
./setidn   ./pruids
./setidn0  ./pruids     - root mit S-Bit
```


Betrachtungen zu "defunct"

```
   fork3.c       fork und wait erzeugen  defunct-Prozesse bei linux,
                       bei Solaris ok

/* server test  */
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <sys/wait.h>
#include <string.h>
#include <errno.h>

pid_t   spid,cpid1,cpid2;

int main()
{
  char buffer[512];
  char out[512];
  int lgt,status,res;

  spid=getpid();
  sprintf(out,"\nServer (%d) startet\n",spid);
  write(2,out,strlen(out));
```

```
  for (;;) {
     write(1,"\n: ",3);
     if (( lgt=read(0,buffer,512)) <=0) {
        sprintf(out,"\nServer ended\n");
        write(2,out,strlen(out)); exit(-1);
     }
     if (lgt <= 1) goto wait;
     if ( (cpid1 = fork()) < 0) {
          sprintf(out,"\nserver can't fork\n");
          write(2,out,strlen(out)); exit(-2);
     }
     if (cpid1 == 0) {  /* child 1 */
        cpid1=getpid();
        sprintf(out,"\n   Client 1 (%d) start.",cpid1);
        write(2,out,strlen(out));
        if ( (cpid2 = fork()) < 0) {
             sprintf(out,"\n   Client 1 (%d) can't fork",cpid1);
             write(2,out,strlen(out)); exit(-2);
        }
        if (cpid2 == 0 ) { /* child 2 */
            cpid2=getpid();
            sprintf(out,"\n      Client 2 (%d(%d)) start.\n",cpid2,cpid1);
            write(2,out,strlen(out)); sleep(5); res='a'-buffer[0];
            sprintf(out,"\n      Client 2 (%d(%d)) end with %d.",
                        cpid2,cpid1,res);
            write(2,out,strlen(out)); exit(res);
        }
```



```
        /* child 1 nach fork */
        res=wait(&status);
        if (res < 0) {
              fprintf(stderr,"\n   res: %d, errno: %d, Text: %s",
                             res,errno,strerror(errno));
        }
        sprintf(out,
        "\nChild 1 (%d) nach wait auf Client 2 (%d). PID: %d, Status: %d\n",
          cpid1,cpid2,res,status>>8);
        write(2,out,strlen(out));
        exit(status>>8);

     }
     /* Server nach fork */
     sprintf(out,"\nServer (%d) start client %d",spid,cpid1);
     write(2,out,strlen(out));
  wait: while ((res=waitpid(-1,&status,WNOHANG)) >0) {
        sprintf(out,"\nServer wait successfull. PID: %d, Status: %d",
                    res,status>>8);
        write(2,out,strlen(out));
     }
     sprintf(out,"\nServer wait failed. PID: %d, errno: %d, msg: %s",
                 res,errno>>8,strerror(errno));
     write(2,out,strlen(out));
  }
  exit(0);
}

    ./fork3
```

```
    fork4.c       Verbesserung unter linux, Problem bei Solaris

/* server test  */
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <sys/wait.h>
#include <string.h>
#include <signal.h>
#include <errno.h>

pid_t   spid,cpid1,cpid2;

void endchild( int sig )
{
  char out[512];
  int res,status;
  int apid;

  apid=getpid();

  /* catch exit-codes from all ending childs */
  while ((res=waitpid(-1,&status,WNOHANG)) >0) {
     if (apid == spid ) {
       sprintf(out,
               "\nsigend1: Server=%d, waitpid successfull: PID: %d, Status: %
```



```
               ,apid,res,status>>8);
     } else {
       sprintf(out,
               "\n   sigend1: Client 1=%d, waitpid successfull: PID: %d, Stat
               ,apid,res,status>>8);
     }
     write(2,out,strlen(out));
     }
     if (apid == spid ) {
              sprintf(out,"\nsigend2: Server=%d, waitpid failed: PID: %d, Sta
     } else {
              sprintf(out,"\n   sigend2: Client 1=%d, waitpid failed: PID: %d
     }
     write(2,out,strlen(out));

}

int main()
{
  char buffer[512];
  char out[512];
  int lgt,status,res;
  struct sigaction newsig,oldsig; /* signalhandling*/

  spid=getpid();
  sprintf(out,"\nServer (%d) startet",spid);
  write(2,out,strlen(out));
  /* init signalhandling  for catch end of process */
  newsig.sa_handler = endchild;
  sigemptyset(&newsig.sa_mask);
```

```
  newsig.sa_flags= 0;
  if (sigaction(SIGCHLD, &newsig, &oldsig) < 0) {
      sprintf(out,"\nServer can't catch SIGCHLD\n");
      write(2,out,strlen(out));
      exit(1);
  }

  for (;;) {
     write(1,"\n: ",3);
     if (( lgt=read(0,buffer,512)) <=0) {
        if ( errno == EINTR) {
           continue;
        }
        sprintf(out,"\nServer ended\n");
        write(2,out,strlen(out));
        exit(-1);
     }
     if (lgt <= 1) continue;
     if ( (cpid1 = fork()) < 0) {
          sprintf(out,"\nserver can't fork");
          write(2,out,strlen(out));
          exit(-2);
     }
     if (cpid1 == 0) {
        /* child 1 */
        cpid1=getpid();
        sprintf(out,"\n   Client 1 (%d) start.",cpid1);
        write(2,out,strlen(out));
        if ( (cpid2 = fork()) < 0) {
             sprintf(out,"\n   Client 1 (%d) can't fork\n",cpid1);
```



```
             write(2,out,strlen(out));
             exit(-2);
        }
        if (cpid2 == 0 ) {
            /* child 2 */
            cpid2=getpid();
            sprintf(out,"\n      Client 2 (%d(%d)) start.\n",cpid2,cpid1);
            write(2,out,strlen(out));
            sleep(5);
            res='a'-buffer[0];
            sprintf(out,"\n      Client 2 (%d(%d)) end with %d.",cpid2,cpid1,
            write(2,out,strlen(out));
            exit(res);
        }
        /* child 1 nach fork */
        res=wait(&status);

        sprintf(out,"\n   Client 1 (%d) nach wait auf Client 2 (%d) PID: %d,
        write(2,out,strlen(out));
        exit(status>>8);

     }
     /* Server nach fork */
     sprintf(out,"\nServer (%d) start client 1 %d",spid,cpid1);
     write(2,out,strlen(out));
  }
  exit(0);
}
```

```
    fork5.c       Korekte Fassung mit Signalhandling


/* server test  */
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <sys/wait.h>
#include <string.h>
#include <signal.h>
#include <errno.h>

pid_t   spid,cpid1,cpid2;

void endchild( int sig )
{
  char out[512];
  int res,status;
  int apid;

  apid=getpid();
  /* catch exit-codes from all ending childs */
  while ((res=waitpid(-1,&status,WNOHANG)) >0) {
    if (res >= 0 ) {
         sprintf(out,"\nsigend: Server=%d, waitpid successfull: PID: %d, Stat
```



```
     } else {
         sprintf(out,"\nsigend: Server=%d, waitpid failed: PID: %d, Status: %
     }
     write(2,out,strlen(out));
  }
}

int main()
{
  char buffer[512];
  char out[512];
  int lgt,status,res;
  struct sigaction newsig,oldsig; /* signalhandling*/

  spid=getpid();
  sprintf(out,"\nServer (%d) startet",spid);
  write(2,out,strlen(out));
  /* init signalhandling  for catch end of process */
  newsig.sa_handler = endchild;
  sigemptyset(&newsig.sa_mask);
  newsig.sa_flags= 0;
  if (sigaction(SIGCHLD, &newsig, &oldsig) < 0) {
      sprintf(out,"\nServer can't catch SIGCHLD\n");
      write(2,out,strlen(out));
      exit(1);
  }

  for (;;) {
     write(1,"\n: ",3);
 M1: lgt=read(0,buffer,512);
```

```
    if (lgt < 0) {
           if ( errno == EINTR)  {
                goto M1;
           }
           sprintf(out,"\nServer error: %d : %s\n",errno,strerror(errno));
           write(2,out,strlen(out));
           exit(-1);
    }
    if (lgt ==0) {
       sprintf(out,"\nServer ended\n");
       write(2,out,strlen(out));
       exit(-1);
    }
    if (lgt == 1 ) continue;
    if ( (cpid1 = fork()) < 0) {
         sprintf(out,"\nserver can't fork\n");
         write(2,out,strlen(out));
         exit(-2);
    }
    if (cpid1 == 0) {
       /* child 1 */
       cpid1=getpid();
       sprintf(out,"\n   Client 1 (%d) start.",cpid1);
       sigaction(SIGCHLD, &oldsig,&newsig);
       write(2,out,strlen(out));
       if ( (cpid2 = fork()) < 0) {
            sprintf(out,"\n   Client 1 (%d) can't fork",cpid1);
            write(2,out,strlen(out));
            exit(-2);
       }
```



```
       if (cpid2 == 0 ) {
           /* child 2 */
           cpid2=getpid();
           sprintf(out,"\n      Client 2 (%d(%d)) start.\n",cpid2,cpid1);
           write(2,out,strlen(out));
           sleep(5);
           res='a'-buffer[0];
           sprintf(out,"\n      Client 2 (%d(%d)) end with %d.",cpid2,cpid1,
           write(2,out,strlen(out));
           exit(res);
       }
       /* child 1 nach fork */
       res=wait(&status);

       sprintf(out,"\n   Client 1 (%d) nach wait auf Client 2 (%d) PID: %d,
       write(2,out,strlen(out));
       exit(status>>8);

    }
    /* Server nach fork */
    sprintf(out,"\nServer (%d) start client 1 %d",spid,cpid1);
    write(2,out,strlen(out));
  }
  exit(0);
}
```