UNIX-Schnittstelle

3. Signale, Signalbehandlung, Synchronisation von Threads und Sonstiges

Alle Beispiel-Quellen mittels SVN unter:

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

3.1 Signale und Signalbehandlung

```
Signale:   Softwareinterrupts für Prozesse
           asynchrone:   Timer, Terminalinterrupts, Pipe-Ende
           synchrone:    Programmfehler, Calls
```

Signalbehandlung hat sich im Laufe der UNIX-Entwicklung stark veräendert.

- Bis einschließlich der Version 7 kam es zu Signalverlusten.
- Siganlbehandlung war umständlich und fehlerhaft.
- BSD 4.3 und AT&T V.4 machten jeweils individuelle und unterschiedliche Verbesserungen.
- Standardisierung jetzt durch POSIX.1

Jedes Signal hat einen Namen. Ursprünglich 15 Signale.
Jetzt 31 Signale unterschiedlicher Funkionalität (BSD/AT&T).

Signalquellen:

```
    1. Terminal erzeugt Signale: SIGINT, SIGQUIT, SIGHUP
                                    ^C      ^\      ausschalten
    2. Hardware-Ausnahmen: Division durch Null, Speicherfehler,
                           Busfehler
    3. Kill-Systemruf
    4. Spezielle Softwarezustände im System: Pipeende, Nutzerzeit
                           abgelaufen, Out-of-Band Daten u.s.w.
```

Typische Reaktionen auf Signale:

1. Ignorieren (alle außer SIGKILL/SIGSTOP)
2. Behandlung der Signale durch eingene Funktionen (Nutzer)
3. Default-Behandlung durch Kern
   - terminate
   - terminate and write core
   - ignore
   - continue or stop work

Übersicht über Signale

```
twc = terminate and write core    t   = terminate
i   = ignore                      c   = continue
ti  = terminate or ignore         s   = stop

SIGABRT    6  twc  used by abort, replace SIGIOT in the  future
SIGALRM   14  t    alarm clock
SIGBUS    10  twc  bus error
SIGCHLD   18  i    death of a child  (old: SIGCLD)
SIGCONT   23  c    continue if stopped
SIGEMT     7  twc  EMT instruction
SIGFPE     8  twc  floating point exception
SIGHUP     1  t    hangup
SIGILL     4  twc  illegal instruction (not reset when caught)
SIGINFO       i    status request from keyboard
SIGINT     2  t    interrupt (rubout)
SIGIO         ti   asynchronous I/O
SIGIOT     6  twc  IOT instruction
```



```
SIGKILL    9  t    kill (cannot be caught or ignored)
SIGPHONE  21  i    handset, line status change
SIGPIPE   13  t    write on a pipe with no one to read it
SIGPOLL   22  t    pollable event occured
SIGPROF       t    profiling time alarm
SIGPWR    19  i    power-fail restart
SIGQUIT    3  twc  quit (ASCII FS)
SIGSEGV   11  twc  segmentation violation
SIGSTOP   24  s    stop signal (cannot be caught or ignored)
SIGSYS    12  twc  bad argument to system call
SIGTERM   15  t    software termination signal from kill
SIGTRAP    5  twc  trace trap (not reset when caught)
SIGTSTP   25  s    interactive stop signal
SIGTTIN   26  s    background read attempted
SIGTTOU   27  s    background write attempted
SIGUSR1   16  t    user defined signal 1
SIGUSR2   17  t    user defined signal 2
SIGVTALRM     t    virtual time alarm
SIGWINCH  20  i    window change
SIGXCPU       twc  CPU-limit exceeded
SIGXFSZ       twc  file size limit exceeded
```

Zuverlässige Information über aktuelle Signale:

```
/usr/include/sys/signal.h
```

```
#include <signal.h>

void (*signal(int signo,void(*func)(int)))(int);

typedef void (*sighandler_t)(int);
sighandler_t signal(int signum, sighandler_t handler);
```

Definition (Installation) einer Signalbehandlungsroutine func. Die Signalbehandlungsroutine wird beim Auftreten eines Signals signo aktiviert. Signalbehandlungsroutinen haben einen Integer-Parameter und keinen Rückkehrwert (bei alten Systemen keinen Parameter). Der Integer-Parameter enthält beim Aufruf der Signalbehandlungsroute die Nummer des Signals, das den Aufruf auslöste hat.

Vordefiniert Signalbehandlungsfunktionen:

```
                SIG_DFL   -   Standardbehandlung der Signale
                SIG_IGN   -   Routine zum ignorieren von Signalen
```

Rückkehrwert:

Adresse der vorhergehenden Signalbehandlungsroutine.
SIG_ERR - Fehler beim Setzen der Signalbehandlungsroutine

Bemerkungen:

exec() : private Signalbehandlungsroutinen werden auf SIG_DFL zurückgesetzt. SIG_IGN bleibt erhalten.

Rücksetzen der Signalbehandlungsroutine:

früher: Nach jedem Auftreten eines Signals wurde die Signalbehandlungsroutine auf SIG_DFL gesetzt.

```
sig_int()
{   /* kritischer Bereich */
    signal(SIGINT,sig_int);
    ....
}
main()
{
    .....
    signal(SIGINT,sig_int);
    .....
}
```

später: Signalbehandlungsroutine bleibt bis zum expliziten Umsetzen durch signal() installiert.

heute: wie früher

```
Wirkung von Signalen auf "wartende Systemrufe":
  früher:  alle "wartenden" Systemrufe wurden unterbrochen,
           es erfolgte kein Restart des Systemrufs.
  heute:   systemabhängig
           AT&T    -  unterbrochen, kein Restart
           BSD 4.3 - unterbrochen, kein Restart ist Standard.
                     Restart optional möglich
           POSIX.1 - Unterbrochen, Restart möglich, nicht
                     notwendig

Was kann man in Signalbehandlungsroutinen tun???
     Fast alles. Man muss nur reenterante Funktionen haben.
     Fast alle Systemcalls sind reenterant, aber nicht alle
     Bibliotheksfunktionen.
     Reenterante Systemfunktionen:
       _exit        abort          access    alarm       chdir
       chown        close          creat     dup         dup2
       execle       execve         exit      fcntl       fork
       fstat        getegid        geteuid   getgid      getgroups
       getpgrp      getpid         getppid   getuid      kill
       link         longjmp        lseek     mkdir       mkfifo
       open         pathconf       pause     pipe        read
       rename       rmdir          setgid    setpgid     setsig
       setuid       sigaction      sigaddset sigdelset   sigemptyset
       sigfillset   sigislmember   signal    sigpending  sigprocmask
       sigsuspend   sleep          stat      sysconf     time
       times        umask          uname     unlink      utime
       wait         waitpid        write
```


Unterschiede bei der Behandlung der Signalroutine und der Wiederaufnahme von Systemaufrufen nach Signalen

Funktion	Betriebssystem	Wiederaufsetzen der Signalroutine	Blockier. von Signalen	Restart des Systemrufs
signal	V7, SVR2 SVR3, SVR4	-	-	niemals
sigset, sighold, sigrelse, sigignore, sigpause	SVR3, SVR4	+	+	niemals
signal, sigvec, sigblock, sigsetmask sigpause	4.2 BSD 4.3 BSD,4.3+BSD	+ +	+ +	immer default
sigaction,sigprocmask sigpending,sigsuspend	POSIX.1 SVR4 4.3+BSD	+ + +	+ + +	unbestimmt optional optional

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

int kill(pid_t pid, int signo);                /*immer vorhanden*/
int raise(int signo);                          /*ANSI C, Bibliotheksroutine*/

kill()  sendet das Signal  signo  an einen Prozess oder eine
Prozessgruppe in Abhängigkeit vom Wert von  pid.
      pid  > 0   -  senden des Signals an den Prozess mit der
                   PID  pid
      pid == 0   -  an alle Prozesse der gleichen Prozessgruppe
      pid == -1  -  Nutzerprozess: Signal an alle Prozesse des
                                   Nutzers
                    Superuser: Signal an alle Prozesse ausser
                               0  und  1
      pid < -1   -  an alle Prozesse der Prozessgruppe  abs(pid)
Wenn  signo==0  ist, wird nur die Gültigkeit von  pid geprüft.
Achtung: alte Systeme kennen nur pid > 0 !!!!
raise()  sendet das Signals  signo an den eigenen Prozess.

Rückkehrwerte:
       0 - Ok
      -1 - Fehler
            EINVAL  - signo ist unzulässig
            ESRCH   - kein Prozess oder Prozessgruppe mit
                      entsprechendem PID
            EPERM   - UID des aktuellen Prozesses reicht nicht aus
                      um an den spezifizierten Prozess ein Signal
                      zu senden.
```



```
#include <unistd.h>

unsigned int alarm(unsigned int seconds);

Kern soll nach  seconds  Sekunden ein Signal SIGALRM an den
aktuellen Prozess senden.

Rückkehrwert:
        > 0 -  Anzahl der nicht verbrauchten Sekunden
               des letzten Systemrufes  alarm.
          0 - kein Alarm-Ruf aufgesetzt


#include <unistd.h>

int pause(void)

Pause bis zum Auftreten eines Signals für diesen Prozess.

Rückkehrwert:
    -1 - immer (Signal aufgetreten)
```

Beispiele:

```
Signale/sigusr.c      # manuelles Aufsetzen der Interruptroutine

Signale/reenter.c     # Probleme mit nicht reenteranten Funktionen

Signale/tsleep1.c     # Zeitgeber

Signale/tsleep2.c     # Zeitgeber
```


Die vorgestellten Systemrufe lassen nur relativ einfache Signalbehandlungen zu. Deshalb wurde sowohl im BSD 4.3 als auch im AT&T V.x der Begriff der Signalmenge (32 Bit) eingeführt. Mit Hilfe dieser Signalmengen ist es möglich die Zulassung und Behandlung von Signalen genauer zu spezifizieren. Für die Manipulation der Signalmenge stehen sowohl im BSD 4.3 als auch im AT&T folgende Funktionen zur Verfügung:

```
#include <signal.h>                         BSD 4.3  &  AT&T V.4

int sigemptyset(sigset_t *set);
     alle Signale in Signalmenge  *set  löschen
int sigfillset(sigset_t *set);
     alle Signale in Signalmenge  *set  setzen
int sigaddset(sigset_t *set, int signo);
     Signal signo in Signalmenge  *set  setzen
int sigdelset(sigset_t *set, int signo);
     Signal signo aus Signalmenge  *set  löschen
int sigismember(const sigset_t *set, int signo);
     Testen ob in Signalmenge  *set  das Signal signo enthalten ist
     1 - is member,   0 - is not member
```

```
                                        veraltet
#include <signal.h>                 /*BSD 4.3, AT&T-Bibliothek*/

int sigvec( int signo, struct sigvec *vec, struct sigvec *ovec);
```

sigvec() arbeitet analog wie signal(). Die Signalbehandlunsroutine und das Signalverhalten wird in einer Struktur *vec definiert. Das alte Signalverhalten wird in der Struktur *ovec entsprechend zurückgegeben, wenn ovec!=NULL.

```
   struct sigvec{
        void (*sv_handler)();  /* Signalbehandlungsroutine */
        int sv_mask;           /* aktuelle Signalmaske */
        int sv_flags;          /* Options */
   }
```

Options:
   SV_ONSTACK   - speziellen Signalstack benutzen
   SV_INTERRUPT - kein "Restart" eines Systemrufes beim Auftreten des Signals nach Ausführung der Signalbehandlungsroutine (alte Signalphilosophie)
   SV_RESETHAND - Rücksetzen der Signalbehandlungsroutine auf SIG_DFL, wenn das Signal einmal aufgetreten ist. (alte Sginalphilosophie)

Rückkehrwert:
   0 - kein Fehler aufgetreten
   -1 - Fehler (EFAULT, EINVAL)

Beispiel: Signal/sigusrvec.c # Rücksetzen des Signalhandlers


```
                                                    veraltet
#include <signal.h>                                 /*BSD 4.3*/

int sigsetmask(int mask);
```

sigsetmask() ersetzt die aktuelle Signalmaske des Prozesses durch die durch mask spezifierte Signalmaske. Ein Signal wird geblockt, wenn das korrespondierende Bit gleich 1 ist!!!

Rückkehrwert:
   alte Signalmaske des Prozesses

```
                                                    veraltet
#include <signal.h>                                 /*BSD 4.3*/

int sigblock(int mask);
```

sigblock() fügt die in mask als gesperrt spezifizierten Signale (korrespondierende Bits gleich 1) zu der aktuellen Signalmaske des Prozesses hinzu.

Rückkehrwert:
   alte Signalmaske des Prozesses

```
                                                     veraltet
#include <signal.h>                                  /*BSD 4.3*/

int siggetmask(void);
```

siggetmask() holt die aktuelle Signalmaske eines Prozesses – anzeige der blockierten Signale.

Rückkehrwert:
alte Signalmaske des Prozesses

```
                                                     veraltet
#include <signal.h>                                  /*BSD 4.3*/

int sigmask(int signum);
```

sigmask() liefert als Ergebnis eine Signalmaske, in der das spezifizierte Signal als blockiert (=1) gesetzt ist.

Rückkehrwert:
Signalmaske mit gesperrtem Signal


```
#include <signal.h>                                /*POSIX, BSD 4.3*/

int sigprocmask(int how, sigset_t *set, sigset_t *oset);
```

sigprocmask() erlaubt das Besichtigen und Ändern der aktuellen Signalmaske des Prozesses. Ist set!=NULL, wird die aktuelle Signalmaske entsprechend how verändert. Ist oset!= NULL, wird die unveränderte Signalmaske in *oset abgelegt. how kann folgende Werte annehmen:

- SIG_BLOCK – in *set gesetzte Signale werden in der aktuellen Signalmaske als gesperrt gesetzt.
- SIG_UNBLOCK – in *set gesetzte Signale werden in der aktuellen Signalmaske als nicht gesperrt gesetzt.
- SIG_SETMASK – aktuelle Signalmaske ergibt sich aus *set

Rückkehrwert:
- 0 – ok
- –1 – Fehler (EFAULT, EINVAL)

```
                                                     veraltet
#include <signal.h>                                  /*BSD 4.3*/

int sigpause(int sigmask);
```

sigpause() ersetzt die aktuelle Signalmaske des Prozesses durch die Signalmenge sigmask (blockierte Signale) und wartet auf das Eintreffen eines Signals. Beim Eintreffen eines Signals wird die ursprüngliche Signalmaske wieder gesetzt. sigmask==0 zeigt an, dass alle Signal akzeptiert werden.

Rückkehrwert: immer –1

```
#include <signal.h>                                    /*POSIX, BSD 4.3*/

int sigsuspend(const sigset_t *sigmaskp);

sigsuspend() setzt die aktuelle Signalmaske des Prozesses auf *sigmaskp
und wartet auf das Eintreffen eines Signals. Anschließend wird die
ursprüngliche Signalmaske wieder gesetzt.

Rückkehrwert:
         -1 - immer


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


#include <signal.h>                                 /*POSIX, BSD 4.3*/

int sigpending(sigset_t *set);

sigpending() schreibt in *set die Signale aus der aktuellen
Signalmaske, die blockiert sind und für die ein Signal anliegt.

Rückkehrwert:
            0  -ok
           -1 - Fehler (falsche Adresse)
```



```
#include <signal.h>                                    /*BSD 4.3*/

int sigstack(struct sigstack *ss, struct sigstack *oss);

sigstack()  erlaubt dem Nutzer einen alternativen Stack für die
Signalbehandlung einzurichten. Dieser wird beim auftreten von Sigalen
benutzt und kann vom Nutzer im den Signalbehandlungsroutinen geändert
werden. Ist ss!=NULL wird ein neuer Stack eingerichtet. Ist oss!=NULL
werden die Informationen über den alten Stack nach *oss  geschrieben.

    struct sigstack {
           char *ss_sp;     /*signal stack pointer */
           int  ss_onstack; /* current status, 0 - unused, !=0 used*/
    }

Rückkehrwert:
       0 - ok
      -1 - Fehler (EPERM - Modifizierung eines aktiven Stack)

-------------------------------------------------------------------
                                                          /*BSD 4.3*/
int killpg(int pgrp, int sig);*/

killpg()  sendet ein Signal  sig an alle Prozesse der
Prozessgruppe  pgrp.

Rückkehrwert:
       0 - ok
      -1 - Fehler (EINVAL, EPERM, ESRCH)
```

```
#include <signal.h>                         /*POSIX,AT&T V.4, BSD 4.3*/
int sigaction(int sig, const struct sigaction *act,
              struct sigaction *oact);

sigaction()  arbeitet analog wie  signal(). Die Signalbehandluns-
routine und das Signalverhalten wird in einer Struktur *act
definiert. Das alte Signalverhalten wird in der Struktur  *oact
entsprechend zurückgegeben, wenn  ovec!=NULL.
   struct sigaction{
        void (*sv_handler)();  /* Signalbehandlungsroutine */
        sigset_t sa_mask;     /* aktuelle Signalmaske */
        int sa_flags;         /* Options */
   }
Options:
   SA_ONSTACK   - speziellen Signalstack benutzen
   SA_RESTART   - kein "Restart" eines Systemrufes beim Auftreten
                  des Signals nach Ausführung der Signalbehandlungs-
                  routine (alte Signalphilosophie)
   SA_RESETHAND - Rücksetzen der Signalbehandlungsroutine auf
                  SIG_DFL, wenn das Signal einmal aufgetreten ist.
                  (alte Sginalphilosophie)
   SA_NODEFER   - Signal wird beim Auftreten vom Kern nicht blockiert
   SA_NOCLDWAIT - für SIGCHLD. Kindprozess erzeugt kein Zombie bei
                  exit()
   SA_NOCLDSTOP - für SIGCHLD. Kindprozess sendet kein SIGCHLD,
                  wenn er gestoppt wird.
   SA_SIGINFO   - zusätzliche Informationen werden an die
                  Signalbehandlungsroutine beim Auftreten eines
                  Signals übergeben (weiterer Parameter)
Rückkehrwert: 0 - ok,  -1 - Fehler (EINVAL)
```


Beispiele:

Signal/sigusract.c

```
#include <signal.h>                /*AT&T V.4, BSD 4.3 Bibliothek*/

void (*sigset(int sig, void (*disp)(int)))(int);

sigset()  arbeitet genauso wie  signal() (Setzen einer Signal-
behandlungsroutine. Lediglich wird beim Auftreten eines
Signals  sig  dieses Signal in die Signalmaske des aktuellen
Prozesses aufgenommen, so dass ein erneutes Auftreten des
Signals  sig  unterdrückt wird.

Vordefiniert Funktionen:
                SIG_DFL   -   Standardbehandlung der Signale
                SIG_IGN   -   Routine zum ignorieren von Signalen

Rückkehrwert:
        Adresse der vorhergehenden Signalbehandlungsroutine.
        SIG_ERR - Fehler beim Setzen der Signalbehandlungsroutine

Beispiel: Signal/sigsetusr.c    #  sigset benutzen
---------------------------------------------------------------
#include <signal.h>                                  /*AT&T V.4*/

int sighold(int sig);

sighold()  fügt  das Signal sig in die Signalmaske des aktuellen
Prozesses hinzu, so dass das Signal in Zukunft gesperrt ist.

Rückkehrwert:
          0 - ok
         -1 - Fehler (EINTR, EINVAL)
```



```
#include <signal.h>                                  /*AT&T V.4*/

int sigrelse(int sig);

sigrelse() entfernt das Signal  sig aus der Signalmaske des
aktuellen Porzesses, so dass das Signal in Zukunft wieder auftreten
kann.

Rückkehrwert:
          0 - ok
         -1 - Fehler (EINTR, EINVAL)

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

#include <signal.h>                                  /*AT&T V.4*/

int sigignore(int sig);

sigignore() setzt die Signalbehandlungsroutine für das Signal sig
für den aktuellen Prozess auf  SIG_IGN.

Rückkehrwert:
          0 - ok
         -1 - Fehler (EINTR, EINVAL)
```

```
#include <signal.h>                                     /*AT&T V.4*/

int sigpause(int sig);

sigpause()  entfernt das Signal  sig  aus der Signalmaske des
aktuellen Prozesses und wartet anschliessend auf das eintreffen
eines Signals.

Rückkehrwert:
         -1 - immer
-----------------------------------------------------------------

#include <signal.h>                               /*POSIX,AT&T V.4*/

int sigprocmask(int how, const sigset_t *set, sigset_t *oset);

sigprocmask() erlaubt das Besichtigen und Ändern der aktuellen
Signalmaske des Prozesses. Ist  set!=NULL, wird die aktuelle
Signalmaske entsprechend  how  verändert. Ist oset!= NULL, wird
die unveränderte Signalmaske in  *oset abgelegt.  how kann
folgende Werte annehmen:
     SIG_BLOCK   - in *set gesetzte Signale werden in der aktuellen
                   Signalmaske als gesperrt gesetzt.
     SIG_UNBLOCK - in *set gesetzte Signale werden in der aktuellen
                   Signalmaske als nicht gesperrt gesetzt.
     SIG_SETMASK - aktuelle Signalmaske ergibt sich aus  *set

Rückkehrwert:
     0 - ok
    -1 - Fehler
```



```
#include <signal.h>                                     /*AT&T V.4*/

int sigalstack(const stack_t *ss, stack_t *oss);

sigalstack()  erlaubt dem Nutzer einen alternativen Stack für die
Signalbehandlung einzurichten. Dieser wird beim auftreten von Sigalen
benutzt und kann vom Nutzer im den Signalbehandlungsroutinen
manipuliert werden. Ist ss!=NULL wird ein neuer Stack eingerichtet.
Ist oss!=NULL werden die Informationen über den alten Stack nach
*oss  geschrieben.

    struct sigalstack {
             int *ss_sp;      /*signal stack pointer */
             long ss_size;    /* stacksize in bytes */
             int  ss_flags;   /* current status */
    }
ss_flags:  SS_DISABLE    - Stack nicht benutzt
           SS_ONSTACK    - Prozess benutzt den Stack momentan

Rückkehrwert:
             0 - ok
            -1 - Fehler
```

```
#include <signal.h>                                     /*POSIX,AT&T V.4*/

int sigpending(sigset_t *set);
```

sigpending() schreibt in *set die Signale aus der aktuellen Signalmaske, die blockiert sind und für die ein Signal anliegt.

Rückkehrwert:
 0 -ok
 -1 - Fehler (falsche Adresse)

```
#include <signal.h>                                     /*POSIX, AT&T V.4*/

int sigsuspend(const sigset_t *set);
```

sigsuspend() setzt die aktuelle Signalmaske des Prozesses auf *set und wartet auf das Eintreffen eines Signals. sigsuspend() kehrt nach Abarbeitung einer Signalbehandlungsroutine zurück. Die ursprügliche Signalmaske wird wieder gesetzt.

Rückkehrwert:
 -1 - immer


```
#include <sys/types.h>                                  /*AT&T  V.4*/
#inlcude <sys/siganl.h>
#include <sys/procset.h>

int sigsend(idtype_t idtype, id_t id, int signo);

int sigsendset(procset_t *psp, int signo);
```

Senden eines Signals signo an den/die spezifizierten Prozess(e).
idtype spezifiziert den Typ des nachfolgenden ID id.

idtype	id	Ziel des Signals
P_PID	pid	Prozess
P-PGID	pgid	Prozesse der Prozessgruppe
P_SID	sid	Prozesse der Session
P_UID	uid	Prozesse der Nutzers
P_GID	gid	Prozesse der Gruppe
P_ALL	-	Alle Prozesse
P_TASKID	tid	Prozesse der Task
P_PROJID	prid	Prozesse des Projekts
P_CID	cid	Prozesse der Scheduler-Class
P_CTID	ctid	Prozesse der Contract-Class
P_MY	-	id des aktuellen Prozesse

```
Bei sigsendset wird das Signal  signo  an eine Menge von Prozessen
gesendet, die aus  *psp   berechnet wird.

stuct procset_t {
             idop_t   p_op;   /*
                     POP_DIFF - Alle Prozesse aus der L-Menge und
                                nicht aus der R-Menge ( logische diff )
                     POP_AND  - Alle Prozesse aus der L-Menge und
                                aus der R-Menge  ( logisch und )
                     POP_OR   - Alle Prozesse aus der L-Menge oder
                                aus der R-Menge ( logisches oder )
                     POP_XOR  - Alle Prozesse die entweder in der L-Menge
                                sind oder der R-Menge sind ( logisch xor )
                             */
             idtype p_lidtype; id_t p_lid;  /* 1.Prozessmenge*/
             idtype p_ridtype; id_t p_rid;  /* 2.Prozessmenge*/
}

Rückkehrwert:
     0 - ok
    -1 - Fehler (EINVAL, EPERM, ESRCH, EFAULT)


Beispiele:

    Signale/critical.c
```



```
                     3.2 Synchronisation mit POSIX-Threads
                     =====================================

3.2.1. Thread-Operationen - eine Übersicht
------------------------------------------
Präfix für verschiede Threadbibliotheken:
  SOLARIS-Threads - thr_
  SOLARIS-Pthread, Linux - pthread_
  altes DIGITAL-Interface - cma_ -Wird ab UNIX 4.0c nicht mehr
          unterstützt. Achtung: immer als Erstes cma_init aurufen!!!!

1. Syncronisationshilfen

   Mutex: Objekte für die Syncronisation mehrerer Threads
           mutex - mutual exclusion (gegenseitiger Ausschluss)

   Type: Fast Mutex - genau einmal pro Thread lock und unlock Synchronisation
         über den Zugriff zum Mutexobjekt bei Mehrfachnutzung -> Deadlock

     SOLARIS: mutex_init, mutex_destroy, mutex_lock,
              mutex_trylock, mutex_unlock
     LIN/SOL: pthread_mutex_init, pthread_mutex_destroy,
              pthread_mutex_lock, pthread_mutex_trylock,
              pthread_mutex_unlock
     SOLARIS: pthread_mutex_getprioceiling, pthread_mutex_getprioceiling
```

```
Type: Bedingter Mutex - recursiver Zugriff erlaubt
      Synchronisation erfolgt über den Wert des Mutexobjektes

  SOLARIS: cond_init, cond_destroy, cond_wait,
           cond_signal, cond_broadcast, cond_timedwait
  LIN/SOL: pthread_cond_init, pthread_cond_destroy,
           pthread_cond_wait, pthread_cond_signal,
           pthread_cond_broadcast, pthread_cond_timedwait

Semaphore-Operationen
  SOLARIS: sema_init, sema_destroy, sema_wait,
           sema_trywait, sema_post

Mehrfach lesbare und einzeln schreibbare Locks
  SOLARIS: rwlock_init, rwlock_destroy, rw_rdlock,
           rw_wrlock, rw_trylock, rw_trywrlock,
           rw_unlock

Signalbehandlung für Threads
  SOLARIS:      thr_sigsetmask, thr_kill, pthread_kill,
  LIN/Solaris:  pthread_sigmask, pthread_testcancel,
                pthread_setcancelstate, pthread_setcanceltype

Handling von private Thread-Daten
  SOLARIS: thr_keycreate, thr_setspecific, thr_getspecific
  LIN/SOLARIS: pthread_key_create, pthread_key_delete,
               pthread_getspecific, pthread_setspecific
```



```
2. Manipulation von Attributen für Variable

  LIN/SOLARIS:
           pthread_mutexattr_init, pthread_mutexattr_destroy,
           pthread_mutexattr_gettype, pthread_mutexattr_settyp
           pthread_condattr_init, pthread_condattr_destroy,
  SOLARIS:
           pthread_mutexattr_getprioceiling,
           pthread_mutexattr_setprioceiling,
           pthread_mutexattr_getprotocol,
           pthread_mutexattr_setprotocol,
           pthread_mutexattr_getpshared,
           pthread_mutexattr_setpshared,
           pthread_condattr_getshared,
           pthread_condattr_setshared
```

3.2.2. Thread Bibliotheksrufe (SOLARIS/DEC/LINUX/POSIX)

```
-------------------------------------------------------
Include-File: thread.h  (pthread.h für POSIX-Threads)

   typedef unsigned int thread_t;
   typedef unsigned int thread_key_t;
   size_t               thr_min_stack(void);
   #define      THR_MIN_STACK   thr_min_stack()
   /*  thread flags (one word bit mask) */
   #define      THR_BOUND               0x00000001
   #define      THR_NEW_LWP             0x00000002
   #define      THR_DETACHED            0x00000040
   #define      THR_SUSPENDED           0x00000080
   #define      THR_DAEMON              0x00000100
   thread_t     thr_self();
   void         thr_yield(void);
   ...
Bemerkungen zur Compilierung:

SOLARIS/LIN:
     pthreads:

       gcc [ flag ... ] file ...  -lpthread [ library ... ]

SOLARIS:
     thread:

       gcc [ flag ... ] file ...  -lthread [ library ... ]
```


1. Syncronisationshilfen

Operationen mit Mutexobjekten (Mutex)

Mutex's werden dazu benutzt die Abarbeitung von Threads zu Serialisieren. Sie stellen sicher, das immer nur ein Thread in einem kritischen Bereich ist (Mehrfachausschluss). Wenn man Mutex's in SharedMemory-Bereichen plaziert, kann man damit auch Threads von verschiedenen Prozessen synchronisieren. Über Mutex's werden Threads bezüglich des Zugriffs auf kritische Daten synchronisiert. Ein Mutex sollte immer nur für einen kritischen Bereich benutzt werden.

Mutexobjekt erzeugen

```
  SOLARIS:
    int mutex_init(mutex_t *mp, int type, void *arg);
  LIN/SOLARIS:
    int pthread_mutex_init(pthread_mutex_t *mp,
                           pthread_mutexattr_t attr);
```

[pthread_]mutex_init() erzeugt und initialisiert ein Mutexobjekt mp.

type kann folgende Werte annehmen:

```
  USYNC_PROCESS - Synchronisation von mehreren Prozessen, einer
                  darf nur mutex_init() abarbeiten, arg ignoriert
  USYNC_THREAD  - Synchronisaton von Threads in einem Prozess,
                  arg ignoriert.
```

Bemerkung: potentiell können weitere Typen spezifiziert werden, dann wird arg zur Initialisierung von mp benutzt.

```
    attr spezifiziert ebenfalls den Type eines Mutexobjektes
    (pthread_mutexattr_init()).
      Default-Werte für attr: NULL (SOLARIS/LIN)

  Rückkehrwerte:
         0  -  Ok
         EINVAL - falsches Argument
         EFAULT - mp oder arg zeigen auf unzulässige Adressen

  Vereinfachung:
     #include <pthread.h>
     pthread_mutex_t mutexobjekt=PTHREAD_MUTEX_INITIALIZER;
            entspricht:
     pthread_mutex_t mutexobjekt;
     pthread-mutex_init(mutexobjekt,NULL);
```

Mutexobjekt streichen

```
  SOLARIS:
    int mutex_destroy(mutex_t *mp);
  LIN/SOLARIS:
    int pthread_mutex_destroy(pthread_mutex_t *mp);

    [pthread_]mutex_destroy() hebt alle Zustände bezüglich des
    spezifizierten Mutexobjektes mp auf.

  Rückkehrwerte:
         wie [pthread_]mutex_init()
```


Mutexobjekt sperren

```
  SOLARIS:
    int mutex_lock(mutex_t *mp);
  LIN/SOLARIS:
    int pthread_mutex_lock(pthread_mutex_t *mp);

    Sperren des Mutexobjektes auf das mp zeigt. Wenn das Mutexobjekt
    bereits gesperrt war, wartet der aktuelle Thread bis das
    Mutexobjekt freigegeben wird.
  Rückkehrwerte:
        wie [pthread_]mutex_init()
```

Mutexobjekt prüfen und sperren

```
  SOLARIS:
    int mutex_trylock(mutex_t *mp);
  LIN/SOLARIS:
    int pthread_mutex_trylock(pthread_mutex_t *mp);

    [pthread_]mutex_trylock versucht das Mutexobjekt auf das mp zeigt zu
    sperren. Dazu wird als erstes geprüft ob das Mutexobjekt bereits
    gesperrt war. Ist dies der Fall, kehrt [pthread_]mutex_trylock() mit
    EBUSY zurück, andernfalls wird der Thread Eigentümer des Mutexobjektes,
    das Mutexobjekt wird gesperrt und der Thread setzt die Arbeit fort.
  Rückkehrwert:
       0  -  Ok
       EINVAL - falsches Argument
       EFAULT - mp oder arg zeigen auf unzulässige Adressen
       EBUSY  - Mutex war bereits gelockt
```

```
Mutexobjekt entsperren

  SOLARIS:
    int mutex_unlock(mutex_t *mp);
  LIN/SOLARIS:
    int pthread_mutex_unlock(pthread_mutex_t *mp);

    [pthread_]mutex_unlock() gibt nach einem [pthread_]mutex_lock() bzw.
    erfolgreichem [pthread_]mutex_trylock() das Mutexobjekt wieder frei.
  Rückkehrwert:
       0  -  Ok
       EINVAL - falsches Argument
       EFAULT - mp oder arg zeigen auf unzulässige Adressen

Zur Verwaltung der Attribute von Mutexobjekten stehen folgende
Systemrufe zur Verfügung:
  LIN/SOLARIS:
    int pthread_mutexattr_init(pthread_mutexattr_t *attr);
    int pthread_mutexattr_destroy(pthread_mutexattr_t *attr);
  SOLARIS:
    int pthread_mutexattr_setpshared(pthread_mutexattr_t *attr,
                 int process-shared);
    int pthread_mutexattr_getpshared(const pthread_mutexattr_t *attr,
            int *process-shared);
    nicht unterstützt
         pthread_mutexattr_getprioceiling
         pthread_mutexattr_getprotocol
         pthread_mutexattr_setprioceiling
         pthread_mutexattr_setprotocol
```



```
Bedingte Mutexobjekte
  Bedingte Mutexobjekte erlauben einen mehrfachen Zugriff auf
  ein Mutexobjekt.  Die Synchronisation erfolgt über Bedingungen,
  die an ein Mutexobjekt gebunden sind.

Initialisieren eines bedingten Mutexobjektes
  SOLARIS:
    int cond_init(cond_t *cvp, int type, void *arg);
  LIN/SOLARIS:
    int pthread_cond_init(pthread_cond_t *cond,
                const pthread_condattr_t *attr);
    Initialisieren eines bedingten Mutexobjektes *cond mit den Werten
    des Attributes attr. Defaultwerte für attr sind:
      SOLARIS:  NULL     und LIN: pthread_condattr_default
    Bedingte Mutexobjekte sollten global addressiert werden.
  Rückkehrwerte:
       0   -  OK
      EFAULT - Illegale Adresse      EINVAL - unzulässiger Wert

Streichen eines bedingten Mutexobjektes
  SOLARIS:
     int cond_destroy(cond_t *cond);
  LIN/SOLARIS:
     int pthread_cond_destroy(pthread_cond_t *cond);

    Freigeben eines bedingten Mutexobjektes. Das Verhalten des
    Systems ist unbestimmt, wenn noch ein Thread dieses
    Objekt benutzt([pthread_]cond_timedwait, [pthread_]cond_wait).

  Rückkehrwerte: siehe pthread_cond_init
```

```
Warten auf eine Signalisierung
   SOLARIS:
     int cond_wait(cond_t *cond, mutex_t *mp);
   LIN/SOLARIS:
     int pthread_cond_wait(pthread_cond_t *cond,
                     pthread_mutex_t *mp);

     Warten auf ein Signale für das spezifizierte bedingte Mutexobjekt
     cond . Das zuvor mit [pthread_]mutex_lock() gesperrte Mutexobjekt
     mp  wird für die Zeit des Wartens freigegeben.
     [pthread_]cond_wait()  wird durch [pthread_]cond_signal() oder
     [pthread_]cond_broadcast() aktiviert.
   Rückkehrwerte: siehe pthread_cond_init

Warten auf eine Signalisierung mit Timelimit
  SOLARIS:
     int cond_timedwait(cond_t *cond, mutex_t *mp,
          timestruc_t *abstime);
  LIN/SOLARIS:
     int pthread_cond_timedwait(pthread_cond_t *cond,
                     pthread_mutex_t *mp,
                     const struct timespec *abstime);

    Warten auf ein Signal für das spezifizierte bedingte Mutexobjekt
    cond . Das zuvor mit [pthread_]mutex_lock() gesperrte Mutexobjekt
    mp  wird für die Zeit des Wartens freigegeben. Es wird ein Fehler
    zurückgegeben, wenn die spezifizierte Absolute Zeit  *abstime
    abgelaufen ist, ohne das [pthread_]cond_timedwait() durch
    [pthread_]cond_signal() oder cond_broadcast() aktiviert wurde.
  Rückkehrwerte: siehe pthread_cond_init
```



```
Senden eines Signals für eine Bedingung
  SOLARIS:
     int cond_signal(cond_t *cond);
  LIN/SOLARIS:
     int pthread_cond_signal(pthread_cond_t *cond);

    [pthread_]cond_signal() sendet ein Signal an das Bedingungsobjekt
    *cond.  Wenn ein Thread mit [pthread_]cond_wait() oder
    [pthread_]cond_timedwait() wartet wird dieser aktiviert und das
    zugehörige Mutexobjekt gesperrt.  Wenn mehrere Threads warten,
    wird entsprechend der Schedulingstrategie, der nächste Thread
    aktiviert.

  Rückkehrwerte: siehe pthread_cond_init()

Sendes Broadcast Signals für eine Bedingung
  SOLARIS:
    int cond_broadcast(cond_t *cond);
  LIN/SOLARIS:
    int pthread_cond_broadcast(pthread_cond_t *cond);

    [pthread_]cond_broadcast() sendet ein Signal an das Bedingungsobjekt
    *cond.  Wenn ein Thread mit [pthread_]cond_wait() oder
    [pthread_]cond_timedwait() wartet wird dieser aktiviert und das
    zugehörige Mutexobjekt gesperrt.  Wenn mehrere Threads warten,
    werden alle Threads aktiviert.

  Rückkehrwerte: siehe pthread_cond_init
```

Zur Verwaltung der Attribute von bedingten Mutexobjekten stehen folgende Sytemfunktionen zur Verfügung:

```
  LIN/SOLARIS:
    int pthread_condattr_create(pthread_condattr_t *attr);
    int pthread_condattr_delete(pthread_condattr_t *attr);
  SOLARIS:
    int pthread_condattr_init(pthread_condattr_attr *attr);
    int pthread_condattr_destroy(pthread_condattr_t *attr);
    int pthread_condattr_getpshared(pthread_condattr_t *attr,
         int *process-shared);
    int pthread_condattr_setpshared(pthread_condattr_t *attr,
         int process-shared);
```

Weiter ältere Synchronisationsfunktionen (Solaris)

```
   Signalbehandlung für Threads
     thr_sigsetmask, thr_kill

   Handling von private Thread-Daten
     thr_keycreate
     thr_setspecific
     thr_getspecific
```


Beispiele:

```
        Signale/simple_mutex
        Signale/sig1
        Signale/sig2
        Signale/stat_sigwait
```

3.3. Sonstiges

```
#include <ucontext.h>                              /*AT&T V.4, BSD 4.3*/

int getcontext(ucontext_t *ucp);

int setcontext(ucontext_t *ucp);
```

Die Funktion getcontext() ermöglicht das Erfassen des aktuellen Prozesszustandes in der Struktur ucontext_t. Mit der Funktion setcontext() kann der zuvor gerettet Prozesszustand wieder hergestellt werden. Mit Hilfe dieser Funktionen ist sowohl ein schnelles Umschalten innerhalb des Prozesses möglich als auch das Setzen der Prozessumgebung innerhalb einer Signalbehandlungsroutine.

Rückkehrwert:

```
 0 - ok (setcontext()  kehrt nicht zurück)
-1 - Fehler, ucp keine gültige Adresse
```



```
#include <sys/time.h>                            /*BSD 4.3, AT&T V.4*/

int getitimer (int which, struct itimerval *value);

int setitimer (int which, struct itimerval *value,
                          struct itimerval *ovalue);
```

BSD unterstützt für jeden Prozess drei Intervaltimer:

1. Realtime-Intervalltimer - SIGALRM
2. virueller Intervalltimer - SIGVTALRM
3. Intervalltimer für Profiling - SIGPROF

getitimer() fragt jeweils einen der Intervalltimer ab. setitimer() setzt einen der Intervalltimer. Durch which wird der jeweilige Intervalltimer bestimmt:

```
ITIMER_REAL    - Realtime-Intervalltimer
ITIMER_VIRTUAL - virtueller Intervalltimer
ITIMER_PROF    - Profiling Timer
```

*oval!=NULL - alter Wert des jeweiligen Timers.

Rückkehrwert:

```
 0 - ok
-1 - Fehler
     EFAULT - falsche Adresse,  EINVAL - falscher Wert
```

```
Zeit- und Prioritätsverwaltung
-------------------------------

int acct(char *path)

Ein- bzw. Ausschalten des Accountsystems des UNIX.
*path - Filename des Accountfiles. Einschalten des Accountsystems.
*path == NULL - Ausschalten des Accountsystems.

Rückkehrwert:
   0 - ok
  -1 - Fehler
         EACCES, EFAULT, EINVAL, EIO, ELOOP, ENOENT,
         ENOTDIR, EPERM EROFS EBUSY, ENOENT

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

#include <sys/time.h>

int adjtime(struct timeval *delta, struct timeval *olddelta);

adjtime()  dient zur kontinuierlichen Synchronisation der Zeit.
Dabei wird die Systemzeit schrittweise um die in *delta angegebenen
Mikrosekunden korregiert (positiv - vorgestellt, negativ - rückgestellt).
*olddelta enthält die Angaben der restliche Mikrosekunden von der
letzten Änderung.

Rückkehrwert:
   0 - ok
  -1 - Fehler
```



```
include <sys/time.h>

int settimeofday(struct timeval *tp, struct timezone *tzp);
int gettimeofday(struct timeval *tp, struct timezone *tzp);

settimeofday()  setzt den Systemzeitgeber.
gettimeofday()  liest den Systemzeitgeber.
   struct timeval {
                      long tv_sec;  /* Sekunden seit 1.1.1970 */
                      long tv_usec; /* und Mikrosekunden */
                 }
   struct timezone {
                     int tz_minuteswest; /* minuten westlich
                                           von Greenwich */
                     int tz_dsttime;    /* Type DST */
                  }
Rückkehrwert:
     0 - ok
    -1 - Fehler

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

#include <sys/typese.h>
#inlcude <time.h>

int stime(const time_t *tp);

stime()  setzt den Systemzeitgeber auf den Wert *tp, der die
Zahl der Sekunden seit 1.1.1970 00:00:00 Uhr angibt. Nur für SU.
```

```
#include <sys/typese.h>

#inlcude <time.h>

time_t time(time_t *tlock)       (veraltet, heute gettimeofday)
```

time() gibt den die Zeit in Sekunden zurück, die seit dem 1.1.1970 00:00:00 Uhr vergangen ist. Rückgabe erfolgt sowohl in *tlock als auch als Rückkehrwert.

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

clock_t times(struct tms *buffer);
```

times gibt die verbrauchte CPU-Zeit in Ticks in der Struktur *buffer zurück.

```
       struct tms {
              clock_t   tms_utime;  /* Nutzerzeit */
              clock_t   tms_stime;  /* Systemzeit */
              clock_t   tms_cutime;  /* Nutzerzeit Kind*/
              clock_t   tms_cstime;  /* Systemzeit Kind*/
       }
```

Der Rückkehrwert ist die vergangene Zeit seit dem Systemstart in Ticks.


```
#include <sys/label.h>
#include <sys/audit.h>

int auditsvc(inf fd, int limit);

int auditon(int condition);

int audit(audit_record_t *record)
```

auditsvc() spezifiziert den Filedescritor und den Speicherplatz für das Audit-Log des Kerns. limit spezifiziert den freien Platten-speicherplatz in Prozent und fd ist der Filedescriptor eines zum Schreiben eröffneten Files.
auditon() ein- bzw. ausschalten des Audit-Log.
audit() schreiben eines Datensatzes in das Audit-Log-File.

Rückkehrwert:
- 0 – ok
- -1 – Fehler

```
#include <sys/label.h>
#include <sys/audit.h>

int setuseraudit(int uid, audit_state_t *state);
int setaudit (audit_state_t *state);
```

Festlegen des UID für die Audit-Sätze geschrieben werden sollen. setaudit() UID des aktuellen Prozesses.

Rückkehrwert:
 0 – ok
 -1 – Fehler

```
int profile(short *buf, int bufsiz, void (*offset)(), int scale);
```

Einschalten des Profiling. buf – Adresse des Puffers für Statistik. bufsize – Länge des Puffers in Byte, offset ist der Beginn des zu testenden Bereich. scale dient der Skalierung (0xffff – 1:1 Abbildung, 0x2 – 1 Eintrag, 0,1 – ausschalten).

Rückkehrwert:
 0 – ok
 -1 – Fehler


```
#include <signal.h>          /*BSD*/
#include <sys/ptrace.h>          /*BSD*/
#include <sys/wait.h>          /*BSD*/
#include <unistd.h>         /*V4.x*/
#include <sys/types.h>          /*V4.x*/

int ptrace(enum ptracereq request, int pid, char *addr, int data
           [,char *addr2]);
```

ptrace() erlaubt dem aktuellen Prozess die Abarbeitung eines anderen Prozesses pid zu steuern und in ihm Veränderungen vorzunehmen.
Nur für SU.

Rückkehrwert:
 0 – ok
 -1 – Fehler (EIO, EPERM, EPERM)

```
int nice(int incr);
```

Änderung der aktuellen Priorität. incr > 0 Verringerung der Priorität.
incr < 0 Erhöhung der Priorität (nur SU).

Rückkehrwert:
 >=0 – neuer nice-Wert (ok
 -1 – Fehler

```
#include <sys/time.h>                              /*BSD*/
#include <sys/resource.h>

int getpriority(int which, int who);
int setpriority(int which, int who, int niceval);

Holen bzw. setzen der Prozesspriorität.
        which                who
      PRIO_PROCESS           PID         Prozess
      PRIO_PGRP              PPGRID      Prozessgruppe
      PRIO_USER              UID         Nutzer
niceval   -  Priorität (-20  .. 19)
Rückkehrwert:
         setpriority :  0  - ok       getpriority: Nicewert
                       -1  - Fehler
-----------------------------------------------------------------------
#include <sys/types.h>                    /*AT&T V.4*/
#include <sys/priocntl.h>
#include <sys/rtpriocntl.h>
#include <sys/tspriocntl.h>

long priocntl(idtype_t, idtype, id_t id, int cmd,  ... /*arg*/);
long priocntlset(procset_t *psp, int cmd, ... /*arg*/);

Holen bzw. setzen der Prozesspriorität. cmd  spezifiziert die Funktion.
Die Argumente werden entsprechend dem Kommando spezifiziert und enthalten
bestimmte Prioritätsangaben.
Rückkehrwert:
       0 - ok
      -1 - Fehler(EAGAIN, EFAULT, EINVAL, ENOMEM, EPERM, ERANGE, ESRCH)
```



```
Systemsteuerung und Systeminformationen
---------------------------------------

int getdomainname(char *name, int namelen);

getdomainname gibt den NIS-Domainname in *name zurück.
namelen  ist die maximale Lenge. Domainname darf maximal 64
Zeichen lang sein.
Rückkehrwert:
     0 - ok
    -1 - Fehler

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

int setdomainname(char *name, int namelen);

setdomainname erlaubt das Setzen des NIS-Domainname *name mit
der Länge  namelen.
Rückkehrwert:
     0 - ok
    -1 - Fehler
```

```
int gethostname(char *name, int namelen);

Holen des Host-Name in *name mit der maximalen Länge namelen.
Rückkehrwert:
     0 - ok
    -1 - Fehler

Beispiel:  Signal/gethost.c   # Hostnamen auslesen



int sethostname(char *name, int namelen);

Setzen des Host-Namen *name mit der Länge namelen.
Rückkehrwert:
     0 - ok
    -1 - Fehler



long gethostid(void);
Auslesen des Host-ID's (32-Bit-Zahl).

Beispiel:     Signal/hostid.c     #  Host-ID auslesen
```



```
#include <sys/utsname.h>
int uname(struct utsname *name);

Holt in der Struktur utsname den Namen und weitere Angaben über das
System.
    struct utsname {
           char sysname[9];   /*OS-Name*/
           char nodename[9];  /*Host-Name*/
           char nodeext[65-9];/*langer Host-Name*/
           char release[9];
           char version[9];
           char machine[9];
     }
Rückkehrwert:
        0 - ok
       -1 - Fehler
```

```
#include <sys/systeminfo.h>                  /*AT&T V.4*/

long sysinfo(int cmd, char *buf, long count);
```

Kopieren von Informationen über das aktuelle System auf dem der Prozess läuft in den Puffer *buf mit der Länge count.
cmd spezifiziert die gewünschte Information.

```
   SI_SYSNAME          -  OS-Name
   SI_HOSTNAME         -  Hostname
   SI_SET_HOSTNAME     -  setzen Hostname
   SI_RELEASE          -  Release
   SI_VERSION          -  Version
   SI_MACHINE          -  Maschinentype
   SI_ARCHITECTURE     -  Architektur
   SI_HW_PROVIDER      -  Hersteller
   SI_HW_SERIAL        -  Seriennummer
   SI_SRPC_DOMAIN      -  Domainname
   SI_SET_SRPC_DOMAIN  -  setzen Domainname
```



```
#include <sys/time.h>
#include <sys/resource.h>

int getrlimit(int resource, struct rlimit *rlp);
int setrlimit(int resource, struct rlimit *rlp);
```

Lesen und setzen von Ressourcenlimits.

```
  resource:  RLIMIT_CPU    -  CPU-Zeit in Sekunden
             RLIMIT_FSIZE  -  Grösste Filelänge
             RLIMIT_DATA   -  Grösse Datensegment
             RLIMIT_STACK  -  Grösse Stacksegment
             RLIMIT_CORE   -  Grösste Länge eines Core-Files
             RLIMIT_RSS    -  Grösster blegter HS
             RLIMIT_NOFILE-   Anzahl der Deskriptoren

  struct rlimit {
            int rlim_cur;   /*current (soft)limit */
            int rlim_max;   /*hard limit */
   }
```

Rückkehrwert:
- 0 – ok
- -1 – Fehler

Beispiel: Signal/getres.c

```
#include <sys/time.h>
#include <sys/resource.h>

int getrusage(int who, struct rusage *rusage);
```

Auslesen der verbrauchten Ressourcen für den aktuellen Prozess (who==RUSAGE_SELF) bzw für die Kindprozesse (who==RUSAGE_CHILDREN).

```
  struct rusage {
           structt timeval ru_utime;  /* user time */
           struct timeval ru_stime;   /*sys time */
           int ru_maxrss; /* max an residenten Seiten */ int ru_ixrss;
           int ru_idrss;  /* durchschn. Seitenzahl*/     int ru_isrss;
           int ru_minflt; /* Page faults, no I/O */
           int ru_majflt; /* Page faults with I/O */
           int ru_nswap;  /* spaps */
           int ru_inblock;/* Block inputs */
           int ru_oublock;/* Block outputs */
           int ru_msgsnd; int ru_msgrcv; int ru_nsignals;
           int ru_nvcsw;  /* contextswitchings for wait */
           int ru_nivcsw; /* contextswitchings for nice */
}
```



```
#include <sys/reboot.h>                              /* BSD */

int reboot(int howto, char *bootargs);
```

System neu starten.

```
  howto:
     RB_HALT      - halt
     RB_ASKNAME   - Standardkernel booten
     RB_SINGLE    - reboot und in Single-User-Mode gehen
     RB_DUMP      - coredump anlegen vor dem Reboot
     RB_STRING    - Übergabe spezieller Informationen in
                    *bootargs
```

```
#include <sys/uadmin.h>                /* AT&T V.4 */

int uadmin(int cmd, int fcn, int mdep);
```

Ausführen spezieller Systemsteueroperationen.

```
     cmd - Kommando
          A_SHUTDOWN
          A_REBOOT
          A_REMOUNT
     fcn - Unterkommandos
          AD_HALT
          AD_BOOT
          AD_IBOOT
```

```
#include <sys/syscall.h>

int syscall(int number,.../*arg*/);
```

syscall erlaubt die Übergabe eines beliebigen Systemrufes an den Kernel.

```
#include <unistd.h>

long sysconf(int name);
```

sysconf() erlaubt die Bestimmung spezieller Systemkonstanten wärend der Laufzeit.
z.B.

name	Bedeutung
_SC_ARG_MAX	maximale Zahle der Argumente
_SC_CHILD_MAX	maximale Zahl der Prozesse pro UID
_SC_CLK_TCK	Ticks pro Sekunde
_SC_OPEN_MAX	maximale Zahl der Files pro Prozess
_SC_JOB_CONTROL	Jobcontrol unterstützt
_SC_SAVED_IDS	Saved ID's unterstützt
_SC_VERSION	POSIX-Version

Beispiel: Signal/sysconf.c