Kapitel 2

Programme und Prozesse

Ein Prozess ist ein laufendes Programm. Dies ist die allererste kurze Definition. In diesem Kapitel soll dies genauer betrachtet werden. Dies ist nicht POSIX standardisiert, es wird hier die UNIX Welt beschrieben.

2.1 Einleitung, Übersicht

Ein Programm ist eine Datei, die nachdem sie in den Speicher kopiert worden ist, lauffähig ist. Üblicherweise wird diese Datei mittels Compiler in zwei Schritten erzeugt. Zuerst wird der source code (Endung .c) durch den Compiler in eine object Datei (Endung .o) übersetzt. Diese Datei wird durch den linker mit verschiedenen anderen object Dateien verbunden und das Ergebnis ist ein Programm.

2.2 Format einer Programm Datei

Man kann sich plattformabhängige Informationen unter UNIX mit dem Befehl

```
man a.out
```

besorgen.

2.2.1 linkage class, storage class

Wenn es darum geht ein C Programm in eine Programm Datei umzuwandeln muss man verschiedene Speicherklassen betrachten. Man unterscheidet zwischen *static* Speicherklasse und der *automatic* Speicherklasse.

Die Speicherklasse static bedeutet, dass Objekte von diesem Typ nach dem Erzeugen die gesamte Laufzeit des Programms leben. Z.B. sind globale Variablen in C Programmen von diesem Typ.

Die Speicherklasse automatic bedeutet, dass Objekte von diesem Typ nur innerhalb eines Blocks leben, d.h. nach Abarbeitung dieses Blocks auch wieder verschwinden, z.B. lokale Variablen. Diese Variablen werden auf dem sog. stack gelegt. Variablen von diesen Typ können mehrfach existieren, z.B. bei Rekursion.

Man muss ferner die *linkage class* betrachten. Dies ist der Gültigkeitsbereich eines Namens (Variable oder Funktion). Normalerweise können nicht lokale Variablen und Funktionen auch außerhalb des files, in dem sie definiert werden verwendet werden. Dies wird als *external linkage* bezeichnet. Dies ist die default Einstellung bei C source code. Kann man diese Namen nur innerhalb der Datei verwenden, so spricht man von *internal linkage*.

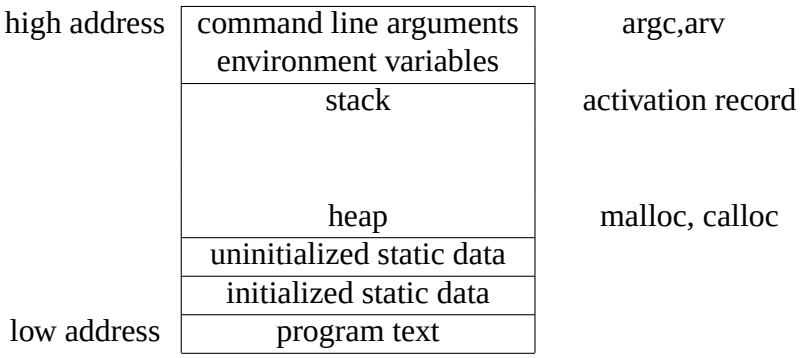
In C gibt es das Schlüsselwort static um diese beiden Dinge zu ändern. Ein automatic Variable (lokale) wird zu einer statischen. Ein Objekt mit der linkage class external wird zu einem internal Objekt.

Dies ist nochmal in der folgenden Tabelle zusammengefasst.

Objekt	Typ	static ändert	mit oder ohne static	storage	linkage class
Variable	lokal	storage class	mit	static	internal
Variable	lokal	storage class	ohne	automatic	internal
Variable	global	linkage class	mit	static	internal
Variable	global	linkage class	ohne	static	external
Funktion	global	linkage class	mit	static	internal
Funktion	global	linkage class	ohne	static	external

2.2.2 Programm im Speicher

Eine typische Implementation kann so aussehen:



Der sog. *activation record* ist die Information, die nötig ist um einen Unterprogrammaufruf zu realisieren. Dies ist die Rückkehradresse, die Parameter (by value), die automatischen Variablen und der Status des Processors. Diese Daten werden vor dem Aufruf gespeichert und nach dem Aufruf der Funktion wieder gelöscht. Hier finden auch die automatischen Variablen Platz. Anders ist es mit den statischen. Sind diese vorbesetzt werden sie schon in den programm file geschrieben. Sind sie nicht vorbesetzt werden sie beim Start in einem eigenen Bereich des Speichers geschrieben. Man sieht diesen Unterschied, wenn man folgende zwei Programme vergleicht:

```
/* 1 */
int i[50000];
main() {
i[3]=3;
while (1);
}
/* 2 */
int i[50000]={1,2,3};
main()
{
i[3]=3;
while (1); }
```

Vergleicht man die Grösse der beiden files:

```
$ ls -l 1 2
-rwx------ 1 axel lehrstuhl 2811 May 25 12:52 1
-rwx------ 1 axel lehrstuhl 202491 May 25 12:52 2

$
```

So sieht man, dass die initialisierte Datei mit in den Programm file aufgenommen wurde. Der heap ist der Bereich der zur dynamischen Speicherverwaltung zur Verfügung steht. Hier steht Speicher zur Verfügung bis er explizit wieder gelöscht wird. (weder static noch automatic) Es ist klar, dass die static storage class prinzipiell nicht thread safe ist.

2.3 Process ID, User ID

Es sollte klar geworden sein, was ein Programm zu einem Process macht. In UNIX wird ein Process durch die Process ID gekennzeichnet. Dies ist eine eindeutige Zahl. In UNIX Spricht man vom Vater (Erzeuger eines Process) und dem Sohn (= erzeugter Process). Man hat auf die jeweiligen Process IDs mit den Befehlen

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

  pid_t getpid(void);
  pid_t getppid(void);
```

Zugriff auf diese IDs. Ferner wird unter UNIX jedem Benutzer eine eindeutige ID zugewiesen, die sog. *user-ID*. Zu einem Process gehört die user-ID des Benutzer, der diesen Process gestartet hat. Ferner gibt es die sog. *effective user-ID* welche die Privilegien des Processes steuert, diese kann während des Processes geändert werden.

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

  uid_t getuid(void);
  uid_t geteuid(void);
```

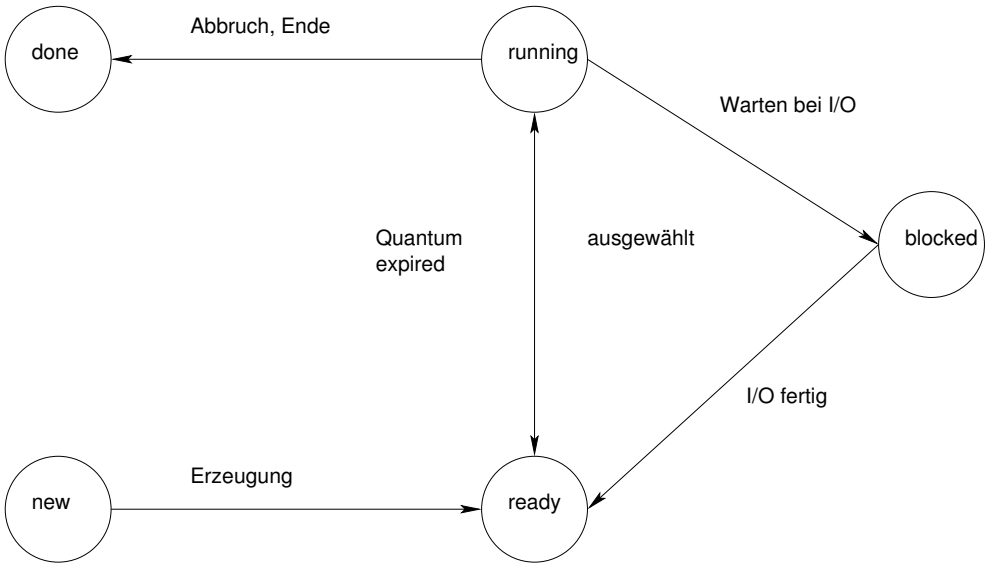
Hierzu ein Programmbeispiel:

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

void main(void)
{
printf("Process ID: %ld\n", (long)getpid());
printf("Parent process ID: %ld\n",
(long)getppid());
printf("Owner user ID: %ld\n", (long)getuid());
}
```

Mit diesem Programm wird die eigene Process ID (PID), die parent PID (PPID) und die user ID des Aufrufers ausgegeben.

2.4 Process Zustände

Ein Process kann im Laufe seines Lebens in verschiedenen Zuständen sein. Man kann dies am einfachsten mit einem Bild zeigen. Dabei werden die möglichen Zustandsänderungen durch Pfeile angedeutet.



Damit sind die 5 wichtigsten Zustände vorgestellt. Ein Wechsel von einem Zustand zu einem anderen kann auch aus anderen Gründen passieren. So ändert der Aufruf von sleep den Zustand auf blocked. Unter *context switch* versteht man das Auslagern eines Processes vom Zustand running und das Einlagern eines neuen Processes. Unter context versteht man all die Dinge, die nötig sind um den Process wieder einzulagern. (z.B. programm counter, Signale, Speicher (statisch, automatisch, dynamisch), ...). Die Informationen über die Processe kann man mit dem UNIX Befehl ps erhalten. Da das process management sehr betriebssystemnah ist, unterscheiden sich die UNIX standards bei der Syntax von ps stark.

2.5 fork

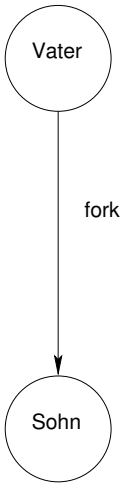
Dies ist die einzige Methode, mit der unter UNIX ein Programm einen neuen Process starten kann.

```
NAME
    fork - create a child process


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

    pid_t fork(void);
```

fork erzeugt eine Kopie des laufenden processes. D.h. man hat nun zwei gleiche Prozesse, sie unterschieden sich lediglich in der PID und der PPID. Um festzustellen ob man sich im Vater oder Sohn process befindet, muss man den Rückgabewert von fork() testen. Er ist 0 im Sohnprocess und die PID des Sohnprocess im Vater process. Dazu ein Beispiel:

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

void main (void)
{
pid_t childpid;
int status;
if ((childpid = fork()) == -1) {
    perror("The fork failed");
    exit(1); }
else if (childpid == 0)
    fprintf(stderr,
          "I am the child with pid = %ld\n",
(long)getpid());
else if (childpid > 0)
    fprintf(stderr,
          "I am the parent with pid = %ld and
child pid = %ld\n",
          (long)getpid(), (long)childpid);

exit(0);
}
```

Die Beziehung Vater, Sohn lässt sich am besten mittels eines Graphen darstellen:



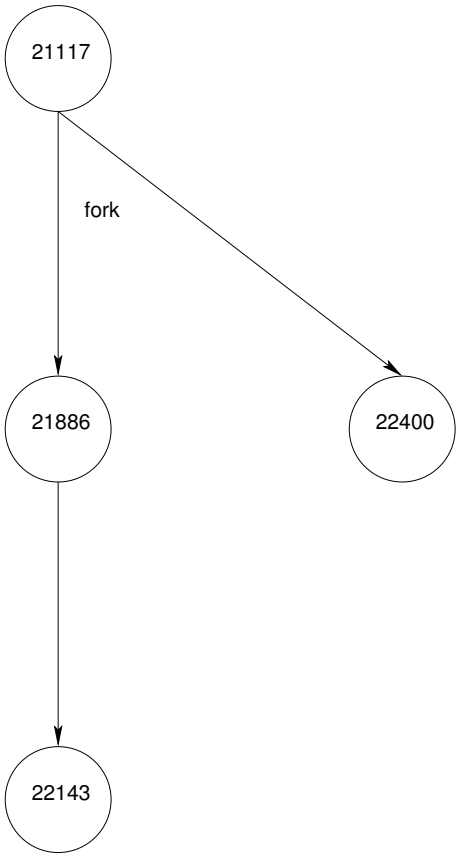
Mit dem fork Aufruf lässt sich jeder beliebiger Baum an Vater Sohn Beziehungen realisieren. Betrachte folgendes Programm:

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

void main (void)
{
int i;
int n=3;
pid_t childpid;

for (i = 1; i < n; i++)
    if ((childpid = fork()) == -1)
        break;

fprintf(stderr, "This is process %ld with
parent %ld\n",
            (long)getpid(), (long)getppid());
}
```

mit folgender Ausgabe

```
This is process 21886 with parent 21117
This is process 22400 with parent 21117
This is process 21117 with parent 14718
This is process 22143 with parent 21886
```

dies ist dann folgender Baum:



Die Ausgabe ist zufällig, da im source code nicht zwischen Vater und Sohn unterschieden wird. Das Programm endet, wenn der erste Process zuende ist, da dann auch die Sohnprocesse beendet werden.

Im Prinzip ist diese Methode geeignet ein UNIX System auszutricksen, da normalerweise die resourcen Verteilung per Process erfolgt. D.h. ein Benutzer, der seine Aufgabe auf viele Processe verteilt ist im Vorteil.

2.6 wait

Dies ist der Systemaufruf, mit dem der Vater warten kann bis der Sohn fertig ist.

```
NAME
    wait - wait for child processes to stop or
terminate

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

     pid_t wait (int *statptr);
```

Der aufrufende Process (Vater) wartet bis irgendein Sohn Process endet, oder der aufrufende Process selber ein Signal erhält. Wurde erfolgreich gewartet ist der Rückgabewert die PID des Sohns. Ansonsten ist der Rückgabewert -1 und errno wird gesetzt. Mit statptr kann man den Rückgabewert des Sohns lesen. Dieser wird mit exit() oder return() (in der Funktion main()) gesetzt. Folgender Programmausschnitt zeigt wie diese verwendet werden können:

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

pid_t child;
int status;

while(((child = wait(&status)) == -1) && (errno
== EINTR)) ;

if (child == -1)
    perror("Could not wait for child");
else if (!status)
    printf("Child %ld terminated normally,
return status is zero\n",
          (long)child);
else if (WIFEXITED(status))
    printf("Child %ld terminated normally,
return status is %d\n",
          (long)child, WEXITSTATUS(status));
else if (WIFSIGNALED(status))
    printf("Child %ld terminated due to signal
not caught\n",
    (long)child);
```

Es gibt POSIX Macros (WIF...) um den return Status des Sohn processes zu überprüfen.

2.7 Aufgabe

In welcher Reihenfolge (Programmteil) kommt die Ausgabe? Warum?

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

int i;
int n;
pid_t childpid;
int status;

for (i = 1; i < n; ++i)
    if ((childpid = fork()) <= 0) break;

for( ; ; ) {
    childpid = wait(&status);
    if ((childpid == -1) && (errno != EINTR))
            break;
}

fprintf(stderr, "I am process %ld, my parent is %ld\n",
         (long)getpid(), (long)getppid());
```

Und wie ist es hier?

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

int i;
int n;
pid_t childpid;
int status;
pid_t waitreturn;

for (i = 1; i < n; ++i)
     if (childpid = fork())
          break;
while(childpid != (waitreturn = wait(&status)))
     if ((waitreturn == -1) && (errno != EINTR))
          break;

fprintf(stderr, "I am process %ld, my parent is %ld\n",
         (long)getpid(), (long)getppid());
```

2.8 exec

Mit diesem Systemaufruf wird kein neuer Prozess erzeugt, sondern der aktuelle Prozess wird durch einen neuen überlagert. Dies ist die Methode, mit der die shell in UNIX einen neuen Prozess startet. Zuerst wird ein fork gemacht, dann wird der Sohn Prozess durch exec überlagert. Es gibt verschiedene Varianten des exec Aufrufs. Die execl Familie übergibt den Befehl und die Argumente als Liste. Die execv Famile übergibt die Argumente als Vektor. Dazu ein kleines Programm:

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

void main(void)
{
pid_t childpid;
int status;

if ((childpid = fork()) == -1) {
    perror("Error in the fork");
    exit(1);
}

else if (childpid == 0) { /* child code */
    if (execl("/usr/bin/ls", "ls", "-l", NULL)
< 0) {
        perror("Exec of ls failed");
        exit(1); }
}

else if (childpid != wait(&status)) {/* parent
code */
    perror("A signal occurred before the child
exited");
    exit(1);}
exit(0);
}
```

Nun die Syntax der execl Familie:

```
SYNOPSIS

  #include <unistd.h>



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

  int execle (const char *path, const char *arg0,
      ..., const char *argn,(char *0), const char
*envp[]);

  int execlp (const char *file, const char *arg0,
      ..., const char *argn,(char *)0);
```

Es handelt sich um einen Aufruf mit einer variablen Anzahl von Parametern. Das Ende wird am letzten Parameter erkannt, der NULL sein muss. Bei execle wird im allerletzten Parameter das environment übergeben. Wird exec in der anderen Form verwandt bleibt das bisherige environment erhalten. Bei execlp wird zum Suchen des Befehls der Pfad (PATH im environment) verwendet.

Nun die execv Familie:

```
SYNOPSIS

  #include <unistd.h>



  int execv (const char *path, char *const *argv);

  int execve (const char *path, char *const *argv,
       char *const *envp);

  int execvp (const char *file, char *const *argv);
```

Hier werden die Argument als argv Vektor übergeben. Die drei Varianten sind wie bei execl. Bei der Verwendung der execv Familie kann die bereits vorgestellte Routine makeargv von Nutzen sein.

2.8.1 Was bleibt bei exec erhalten?

Es ist klar, dass der Programmtext, Variable, Stack und Heap überschrieben werden. Das environment bleibt erhalten ausser bei exec..e Aufrufen. Nachfolgend eine Liste der Attribute, die erhalten bleiben:

Attribut	System Aufruf
PID	`getpid()`
PPID	`getppid()`
Process Group ID	`getpgid()`
Session membership	`getsid()`
real user ID	`getuid()`
real group ID	`getgid()`
zusätzliche group ID	`getgroups()`
Zeit aus alarm	`alarm()`
aktuelles directory	`getcwd()`
root directory	
Maske beim Erzeugen von files	`umask()`
Maske für Signale	`sigprocmask()`
nicht abgearbeitete Signale	`sigpending()`
bisher benötigte Zeit	`times()`

2.9 Hintergrund Prozesse, Dämonen

Von den meisten shells aus kann man Befehle im *Hintergrund* ausführen. Dies bedeutet, dass keine Kommunikation mit dem Terminal mehr stattfindet. Man sagt der Prozess hat kein *controlling Terminal*. *Dämonen* sind Hintergrundprozesse, die unbegrenzt laufen. Typischerweise warten sie auf Ereignisse, und behandeln diese. Z.B. nfs-server, Druckerspooler, WWW-server, etc. Um als Programmierer einen Hintergrundprozess zu starten verwendet man den Befehl setsid().

```
NAME
   setsid - creates a session and sets the
process group ID

SYNOPSIS

  #include <unistd.h>



  pid_t setsid(void);
```

Durch den Aufruf von setsid wird der Process zu einem sog. process group leader einer neuen session. Der Process bekommt eine eigene session ID, und da ein Terminal nur eine session unterstützt, hat der Prozess nun kein controlling Terminal mehr, und ist somit ein Hintergrund Process. Das folgende Programm erzeugt einen Hintergrund Process:

```
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
int makeargv(char *s, char *delimiters, char ***argvp);

void main(int argc, char *argv[]) {
  char **myargv;
  char delim[] = " \t";
  pid_t childpid;

  if (argc != 2) {
    fprintf(stderr, "Usage: %s string\n", argv[0]);
    exit(1);
    }

  if ((childpid = fork()) == -1) {
    perror("The fork failed");
    exit(1);
    }
  else if (childpid == 0) {
/* child becomes a background process */
    if (setsid() == -1)
      perror("Could not become a session leader");
    else if (makeargv(argv[1], delim, &myargv) < 0)
      fprintf(stderr, "makeargv error\n");
    else if (execvp(myargv[0], &myargv[0]) < 0)
      perror("The exec of command failed");
     exit(1); /* child should never return */
    }
   exit(0); /* parent exits */
}
```

Auch für einen Dämon Process noch ein Beispiel. Das folgende Programm schaut regelmässig nach ob eine mail angekommen ist. Wenn eine mail da ist piept das Terminal (=ctl-g).

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#define MAILFILE "/var/mail/oshacker"
#define SLEEPTIME 10

void main(void) {
  int mailfd;

  for( ; ; ) {
    if ( (mailfd = open(MAILFILE, O_RDONLY)) !=
-1) {
      fprintf(stderr,"%s", "\007");
      close(mailfd);
      }
    sleep(SLEEPTIME);
    }
}
```

Dies ist wieder ein Programm, welches man nicht unbedingt als Beispiel nehmen sollte. Benutzername und directory sind hard-coded. Die korrekte Methode ist mit getuid die UID zu bekommen und mit getpwuid den Benutzernamen. Da das directory für mails auf UNIX systemen nicht einheitlich geregelt ist, sollte das Programm versuchen die environment Variablen MAIL, MAILCHECK, MAILDIR, MAILPATH zu verwenden.

2.10 environment

Um Processen systemspezifische Informationen zukommen zu lassen gibt es das environment. Dies ist eine Liste von Paaren Name=Wert. Der Name gibt die environment Variable an und Wert ist ein string. Unter POSIX.2 sind folgende Namen und ihre Bedeutung festgelegt.:

Varaible	Bedeutung
HOME	login directory
LANG	Sprache
LC_ALL	
LC_COLLATE	
LC_CTYPE	
LC_MONETARY	
LC_NUMERIC	
LC_TIME	
LOGNAME	Benutzername zu diesem Prozess
PATH	Suchpfad
TERM	Terminal Information für Ausgaben
TZ	Zeitzone

Es gibt für jeden Process eine globale Variable char **environ die den aktuellen Wert enthält.

```
#include <stdio.h>
#include <stdlib.h>
extern char **environ;

void main(int argc, char *argv[])
{
int i;
printf("The environment list for %s is\n",
argv[0]);
for(i = 0; environ[i] != NULL; i++)
  printf("environ[%d]: %s\n", i, environ[i]);
exit(0);
}
```

Um zu Testen ob eine environment Variable gesetzt ist gibt es den Aufruf getenv.

```
NAME

  getenv - get an environment variable

SYNOPSIS
  #include <stdlib.h>

  char *getenv(const char *name);
```

Daher hätte unser mail-Dämon besser so ausgesehen:

```
...
#include <stdlib.h>
#define MAILDEFAULT "/var/mail"
char *mailp = NULL;

if (getenv("MAILPATH") == NULL)
  mailp = getenv("MAIL");
if (mailp == NULL)
  mailp = MAILDEFAULT;
....
```

Der Wert von mailp sollte kopiert werden, da der Zeiger bei einem erneuten Aufruf von getenv verändert werden kann.

2.11 Beenden von Processen

Am Ende eines Process werden vom System die benötigten Resourcen wieder freigegeben. Dies sind virtueller Speicher, locks, Signale, offene files. Dananch wird der Process beendet und ein wait Aufruf beim Vater wird beantwortet. Hat der Process selber Söhne, so werden diese zu *Zombies*, und werden vom Process mit der Nummer 1 (=init) adoptiert. init macht ab und zu ein wait um auch diese Processe ordentlich sterben zu lassen. Von einem normalen Ende eines Processes spricht man wenn der Process mit return in main, oder exit(), oder _exit() beendet wird.

```
NAME
  exit, atexit, _exit - Terminates a process

LIBRARY                 Standard C Library
(libc.so, libc.a)

SYNOPSIS

  #include <stdlib.h>
    int atexit( void (*function) (void));
    void exit(int status);
  #include <unistd.h>
     void _exit(int status);
```

Mit exit und _exit kann eine Zahl an den aufrufenden Process übergeben werden. Gelesen wird dieser mit wait(*status). Mit return kann dieser Wert nur übergeben werden, wenn main nicht als void declariert wurde. Die Funktion atexit() installiert einen exit handler, der am Ende des Processes aufgerufen wird. Dabei ist die Routine _exit lowlevel. Sie wird von exit aufgerufen nachdem exit die mit atexit installierten handler aufgerufen hat. Man kann mehrere exithandler stack-mässig definieren. Auch hierzu noch ein Beispiel:

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/times.h>
#include <limits.h>

static void show_times(void)
{
  struct tms times_info;
  double ticks;

  if ((ticks = (double) sysconf(_SC_CLK_TCK)) < 0)
    perror("Cannot determine clock ticks per second");
  else if (times(&times_info) < 0)
    perror("Cannot get times information");
  else {
    fprintf(stderr, "User time: %8.3f seconds\n",
        times_info.tms_utime/ticks);
    fprintf(stderr, "System time: %8.3f seconds\n",
        times_info.tms_stime/ticks);
    fprintf(stderr, "user time(child): %8.3f seconds\n",
        times_info.tms_cutime/ticks);
    fprintf(stderr, "sys time(child): %8.3f seconds\n",
        times_info.tms_cstime/ticks);
    }
}

void main(void)
{
  if (atexit(show_times)) {
    fprintf(stderr, "Cannot install show_times exit
handler\n");
    exit(1);
    }
/* rest of main program goes here */
}
```

Am Ende wird eine Zeitstatistik ausgegeben. Unter abnormal Termination versteht man jedes andere Beenden. Dies kann durch den Aufruf von abort() passieren, oder aber durch Signale (z.B. ctl-c).