Kapitel 4: Signale

4: Signale

- UNIX-Signale sind Software-Interrupts die an Prozesse zugestellt werden.
- Signale sind Ereignisse die (zu einem unverhersagbaren Zeitpunkt) bei der Ausführung eines Prozesses auftreten können
- Interrupts erlauben asynchrone Kommunikation zwischen dem Kernel (Ring-0) und der CPU.
- Signale erlauben asynchrone Kommunikation zwischen
  - Prozessen (Interprozesskommunikation, IPC) sowie
  - zwischen Kernel und Prozessen.
- Falls die CPU einen Interrupt auslöst (z. B. Divison by Zero) versendet der Kernel an den verursachenden Prozess ein Signal (**`SIGFPE`**). (→ `$ strace ./divisionbyzero`)

Signalkonzept

- Der Prozess kann für Signal **X** einen **Signalhandler Y** einrichten
- Falls Signal **X** eintritt, wird der **Signalhandler Y** aufgerufen
- Falls für ein empfangenes Signal kein Signalhandler eingerichtet wurde, so wird eine Default-Aktion ausgeführt.
  - beenden,
  - stoppen oder
  - wecken
- Ein Signal ist als Variable vom Typ `int` codiert.

  Auflistung aller Signale: `$ man 7 signal`

Terminal-Signale

- `$ man 7 signal`
- Signale können via Terminal ausgelöst werden.
  - Strg + C : SIGINT
  - Strg + \ : SIGQUIT
  - Strg + Z : SIGSTOP
  - `$ fg`: SIGCONT,
  - `$ bg`: SIGCONT, SIGTTIN
- `$ kill -s <signal> <pid>`
- `$ killall -s <signal> <name>`

Signalsichere Funktionen

- Beim Eintritt eines Signals wird die normale Ausführung des Prozesses kurzfristig unterbrochen und der Signalhandler wird abgearbeitet bevor der normale Programmfluss fortgesetzt wird.
- (Signal)sichere (engl. *safe*) Funktionen können bedenkenlos innerhalb eines Signalhandlers aufgerufen werden
- Wird eine **unsichere** Funktion durch ein Signal unterbrochen, darf der Signalhandler keine unsichere Funktion aufrufen; ansonsten ist das Verhalten des Programms **undefiniert.**
- Beispiel einer unsicheren Funktion: `malloc()`.
- Die Signal(7)-Manpage enthält eine Liste von sicheren Funktionen (`$ man 7 signal`).

Begriffe

- Zustellung eines Signals: Aufruf des Signalhandlers
- Hängen eines Signals: Zeitspanne zwischen Erzeugung und Zustellung eines Signals.
- Blockieren eines Signals: Ein Prozess kann Signale blockieren.
  - Ein blockiertes Signal wartet auf Zustellung, bis die Blockade aufgehoben wurde.
  - Blockierte Signale können vom Prozess auch ignoriert werden.
- Signalmaske: Die Signalmaske enthält eine Liste von Signalen die momentan blockiert sind.

Wichtige Anmerkungen 1 und 2

Anmerkung 1: Unaufhaltsame Signale

Die beiden Signale `SIGKILL` und `SIGSTOP` können weder behandelt, ignoriert noch blockiert werden.

Anmerkung 2: Die Zustellungsreihenfolge ist undefiniert

Werden mehrere unterschiedliche Signale parallel an einen Prozess gesandt, so ist deren Zustellungsreihenfolge undefiniert.

Wichtige Anmerkungen 3 und 4

Anmerkung 3: Ungestörte Unterbrechung

Bei der Ausführung eines Signalhandlers wird das behandelte Signal normalerweise ignoriert oder blockiert.

Anmerkung 4: Einmalige Zustellung

Blockierte Signale, die mehrmals auftreten, werden normalerweise nur einmal zugestellt.

4.1: Das alte Linux-Signalkonzept

Signalhandler

```
#include <signal.h>

typedef void (*sighandler_t)(int);

sighandler_t signal(int signum, sighandler_t handler);
```

- Systemcall registriert Signalhandler für das Signal `signum`.
- Bei dem Signalhandler `sighandler_t` handelt es sich um einen Funktionspointer.
  - Rückgabetyp: **`void`**
  - Optionaler Parameter: **`int`**
- Vordefinierte Signalhandler
  - **`SIG_IGN`**: Ignoriert alle Signale bis auf `SIGKILL` und `SIGSTOP`
  - **`SIG_DFL`**: Default-Aktion

Anmerkungen zum Systemcall signal()

- Das Verhalten von `signal()` hängt vom Betriebssystem ab.
- Unter Linux wird das Signal `signum` während der Ausführung des Signalhandlers blockiert.
- Einige UNIX-Systeme setzten den Signalhandler nach einmaliger Ausführung wieder auf `SIG_DFL` zurück.
- Aus Gründen der Portabilität sollte daher auf den Einsatz von `signal()` verzichtet werden.

Pause

```
#include <unistd.h>

int pause(void);
```

- **`pause()`** legt den aufrufenden Prozess schlafen, bis ein Signal übermittelt wird, das den Prozess entweder beendet oder eine Funktion aufruft, die das Signal abfängt.
- Der Systemcall **`pause()`** gibt immer `-1` zurück; und `errno` wird auf `EINTR` gesetzt.

Beispiel: Signalhandler

```
1  #include <signal.h>
2  #include <stdio.h>
3  #include <unistd.h>
4
5  void sigint_handler() {
6     puts("Nice try.");
7     signal(SIGINT, SIG_DFL);
8  }
9
10
11 int main() {
12   signal(SIGINT, sigint_handler);
13   while(1) {  pause();  }
14   return 0;
15 }
```

Sleep

```
#include <unistd.h>

unsigned int sleep(unsigned int seconds);
```

- "**`sleep()`** lässt den aktuellen Prozes schlafen, bis `seconds` Sekunden abgelaufen sind oder ein Signal eintrifft, welches nicht ignoriert wird."(`man 3 sleep`)
- Falls die Zeit abgelaufen ist wird `0` zurückgegeben, ansonsten die Restlaufzeit in Sekunden.
- **Merke:** `sleep()` kann durch ein Signal unterbrochen werden.

Beispiel: Sleep im Signalhandler

```
1  #include <signal.h>
2  #include <stdio.h>
3  #include <unistd.h>
4
5  void sig_usr1() {
6    write(STDOUT_FILENO, "USR1\n",5);
7    sleep(4);
8  }
9
10 void sig_usr2() {
11   write(STDOUT_FILENO, "USR2\n",5);
12   sleep(2);
13 }
14
15 int main() {
16   signal(SIGUSR1, sig_usr1);
17   signal(SIGUSR2, sig_usr2);
18   while(1) {
19     printf("%u\n", getpid());
20     sleep(10);
21   }
22 }
```

Beispiel:

```
1  #include <signal.h>
2  #include <stdio.h>
3  #include <unistd.h>
4
5  void sig_usr(int signum) {
6    if(signum == SIGUSR1) write(STDOUT_FILENO, "USR1\n",5);
7    else                  write(STDOUT_FILENO, "USR2\n",5);
8
9    sleep(3);
10 }
11
12 int main() {
13   signal(SIGUSR1, sig_usr);
14   signal(SIGUSR2, sig_usr);
15
16   while(1) { pause();  }
17 }
```

Beispiel: Ignorierung von Signalen 3

```
1  #include <signal.h>
2  #include <stdio.h>
3  #include <unistd.h>
4
5  static int sig1=0;
6  static int sig2=0;
7
8  void sig_usr1() { sig1++; }
9  void sig_usr2() { sig2++; }
10
11 int main() {
12   signal(SIGUSR1, sig_usr1);
13   signal(SIGUSR2, sig_usr2);
14
15   while(1) {
16     printf("%u\n", getpid());
17     if(sig1 != 0) { puts("USR1"); sig1-=1; }
18     if(sig2 != 0) { puts("USR2"); sig2-=1; }
19     sleep(2);
20   }
21 }
```

Signal-Kurzbeschreibung

```
#include <string.h>
#include <signal.h>

char *strsignal(int sig);
void psignal(int sig, const char *s);
```

- **`strsignal()`** gibt Kurzbeschreibung von einem Signal zurück.
- **`psignal()`**: Gibt den String `s` auf `stderr` gefolgt von einem Doppelpunkt und der Beschreibung des Signals `sig`, gefolgt von `\n`, aus.

Beispiel: Signal-Kurzbeschreibung

```
1  #define _POSIX_C_SOURCE 200809L
2
3  #include <string.h>
4  #include <signal.h>
5  #include <stdio.h>
6
7  int main() {
8    for(int i=1;i<10;i++) {
9      puts(strsignal(i));
10   }
11
12   psignal(SIGSTOP, "error␣message");
13   return 0;
14 }
```

Senden von Signalen

```
#include <signal.h>

int kill(pid_t pid, int sig);
```

- **`kill()`** sendet das Signal `sig` an den Prozess mit der PID `pid`.
- Signale können nicht an beliebige Prozesse versendet werden.
- Normalerweise muss die User-ID des Sender- und Empfängerprozesses übereinstimmen.
- `sig==0`: Es wird nur überprüft, ob ein Signal an den Prozess mit der PID `pid` geschickt werden kann.
- Bei Erfolg wird `0` zurück gegeben, ansonsten `-1`.
- Komfortfunktion **`int raise(int sig)`** entspricht **`kill(getpid(), sig)`**.

Einrichten eines Weckers

```
#include <unistd.h>

unsigned int alarm(unsigned int seconds);
```

- **`alarm()`** gibt die Sekunden bis zur Auslösung eines vorher geplanten Alarms zurück. War kein Alarm geplant, so wird **`Null`** zurückgegeben.
- Nach `seconds` Sekunden wird das Signal `SIGALRM` gesendet.
- `seconds==0`: Alle eingestellten Alarme werden verworfen.

Beispiel: Wecker

```
1  #include <signal.h>
2  #include <stdio.h>
3  #include <unistd.h>
4
5  #define LINE_MAX 1024
6
7
8  int main() {
9    char line[LINE_MAX+1];
10   int n;
11
12   do {
13     alarm(5);
14     n = read(STDIN_FILENO, line, LINE_MAX);
15     alarm(0);
16
17     if(n>0) write(STDOUT_FILENO, line, n);
18   } while(n>0);
19 }
```

Beispiel: Senden des Null-Signals

```
1  #include <unistd.h>
2  #include <signal.h>
3  #include <stdio.h>
4  #include <stdlib.h>
5
6  int main() {
7    long max = sysconf(_SC_CHILD_MAX);
8
9    for(long i=1; i < max; i++) {
10     if (kill(i, 0) != -1) printf("PID: %ld\n",i);
11   }
12
13   return EXIT_SUCCESS;;
14 }
```

Probleme mit dem Signalkonzept

- Erfragen des aktuellen Signalhandlers ist nicht möglich ohne einen *neuen* Signalhandler einzurichten.
- Mit `signal()` können keine Signale kurzzeitig blockiert werden um diese später zu bearbeiten.

4.2: Das neue Linux-Signalkonzept

Signalmengen

```
1 #include <signal.h>
2
3 int sigemptyset(sigset_t *set);
4 int sigfillset(sigset_t *set);
5 int sigaddset(sigset_t *set, int signum);
6 int sigdelset(sigset_t *set, int signum);
7 int sigismember(const sigset_t *set, int signum);
```

- Im Fehlerfall wird **`-1`** zurückgegeben.
- **`sigemptyset()`**: Entfernt alle Signale aus der Signalmenge `set`
- **`sigfillset()`**: Fügt der Signalmenge `set` alle Signale hinzu.
- **`sigaddset()`**: Fügt das Signal `signum` der Menge `set` hinzu.
- **`sigdelset()`**: Entfernt das Signal `signum` aus Menge `set`.
- **`sigismember()`**: Gibt `1` zurück falls `signum` in `set` enthalten ist.

Beispiel: Signal-Kurzbeschreibung

```
1  #include <signal.h>
2  #include <stdlib.h>
3  #include <stdio.h>
4
5  int main() {
6    sigset_t set;
7    sigemptyset(&set);
8
9    printf("%d\n",sigismember(&set,SIGTERM));
10   sigaddset(&set,SIGTERM);
11   printf("%d\n",sigismember(&set,SIGTERM));
12
13   sigdelset(&set,SIGTERM);
14   printf("%d\n",sigismember(&set,SIGTERM));
15   sigfillset(&set);
16   printf("%d\n",sigismember(&set,SIGTERM));
17
18   return EXIT_SUCCESS;
19 }
```

Einrichtung und Abfrage eines Signalhandlers

```
#include <signal.h>

int sigaction(int signum,
              const struct sigaction *act,
              struct sigaction *oldact);
```

- Bei Erfolg wird `0` zurückgegeben, ansonsten `-1`.
- **`signum`**: Signal für das ein Signalhandler einzurichten oder abzufragen ist.
- **`act`**: Einrichten eines neuen Signalhandlers oder `NULL`.
- **`oldact`**: Falls ungleich `NULL`, wird der momentan eingerichteten Signalhandler zurückgeliefert.

Struktur **sigaction**

```
1 struct sigaction {
2   void      (*sa_handler)(int);
3   ...
4   sigset_t    sa_mask;
5   int         sa_flags;
6 };
```

- **sa_handler**: Adresse des Signalhandlers
- **sa_mask**: Maske von zu blockierenden Signalen
- **sa_flags**: Signaloptionen:
  - SA_NODEFER: Während der Ausführung des Signalhandlers wird das Signal nicht blockiert.
  - SA_RESETHAND: Signalhandler wird nach dem Aufruf zurück auf SIG_DFL gesetzt.

Beispiel: sigaction

```
1  #include <signal.h>
2  #include <stdio.h>
3  #include <unistd.h>
4  #include <string.h>
5
6  void sig_foobar(int signum) {
7    if(signum == SIGTERM) puts("foo");
8    else                  puts("bar");
9    sleep(3);
10 }
11
12 int main() {
13   sigset_t set;
14   sigfillset(&set);
15   struct sigaction st;
16   memset(&st,0,sizeof(struct sigaction));
17   st.sa_handler=sig_foobar;
18   st.sa_mask = set;
19   sigaction(SIGTERM, &st, NULL);
20   sigaction(SIGUSR1, &st, NULL);
21   while(1) {  printf("%u\n", getpid()); sleep(2); }
22   return 0;
23 }
```

Ungestörtes Arbeiten

Manchmal ist es notwendig Signale zu blockieren, um kritische Codeabschnitte **ungestört** aufrufen zu können:

- Einleitung einer Vollbremsung
- Auslösen des Feueralarms
- Verhinderung einer Kernschmelze
- ...

Signale blockieren

```
#include <signal.h>

int sigprocmask(int how, const sigset_t *set, sigset_t *
    oldset);
```

- Abfragen und modifizieren der aktiven Signalmaske
- `set==NULL, oldset!=NULL`
  Signalmaske ohne Änderungen erfragen.
- `set!=NULL, oldset==NULL`
  Signalmaske ohne Erfragen ändern.
- `set!=NULL, oldset!=NULL`
  Signalmaske mit Erfragen ändern.
- `how` Sei `S` die aktuelle Signalmaske:
  - `SIG_BLOCK`: $S \leftarrow S \cup$ `set`
  - `SIG_UNBLOCK`: $S \leftarrow S \setminus$ `set`
  - `SIG_SETMASK`: $S \leftarrow$ `set`

Beispiel: Ungestörtes Arbeiten

```
1  #include <signal.h>
2  #include <stdio.h>
3  #include <stdlib.h>
4  #include <unistd.h>
5
6  int main() {
7    sigset_t mask, oldmask;
8
9    sigfillset(&mask);
10   sigprocmask(SIG_SETMASK, &mask, &oldmask);
11
12   /* Critical code */
13   puts("Replace this line with critical code");
14   sleep(30);
15
16   sigprocmask(SIG_SETMASK, &oldmask, NULL);
17   pause(); /* Wait for a signal */
18
19   return EXIT_SUCCESS;
20 }
```

Frage: Warum können hier Signale verloren gehen?

Erfragen von hängenden Signalen

```
#include <signal.h>

int sigpending(sigset_t *set);
```

- Bei Erfolg wird `0`, ansonsten `-1` zurückgegeben.
- Schreibt die Menge der momentan hängenden (blockierten) Signalen an die Adresse `set`.
- Hinweis: Bei `set` handelt es sich um eine Bitmaske.
  - Das i-te Bit ist gesetzt $\Longrightarrow$ Signal *i* hängt.
  - Das i-te Bit ist **nicht** gesetzt $\Longrightarrow$ Signal *i* hängt nicht.
- **Beispiel:** `set==10` $\Longrightarrow$ Signal 3 und 1 hängen.

Beispiel: Hängendes Signal

```
1  #include <stdio.h>
2  #include <signal.h>
3  #include <unistd.h>
4  #include <stdlib.h>
5
6  void handler(int signum) {
7    if(signum == SIGINT) puts("SIGINT␣intercepted.");
8    signal(SIGINT, SIG_DFL);
9  }
10
11 int main() {
12   sigset_t set, sigintset;
13   signal(SIGINT, handler);
14   sigemptyset(&sigintset);
15   sigaddset(&sigintset, SIGINT);
16
17   sigprocmask(SIG_BLOCK, &sigintset, NULL);
18   puts("SIGINT␣blocking:␣activated");
19   sleep(10);
20
21   sigpending(&set);
22   if (sigismember(&set, SIGINT)) puts("SIGINT␣is␣pending");
23
24   sigprocmask(SIG_UNBLOCK, &sigintset, NULL);
25   puts("SIGINT␣blocking:␣disabled");
26   sleep(30);
27
28   return EXIT_SUCCESS;
29 }
```

Zusammenfassung

Nach diesem Kapitel sollten Sie . . .

- . . . wissen was ein Signal ist.
- . . . den Zustand eines Prozesses mittels Signale zu verändern.
- . . . das Linux-Signalkonzept verinnerlicht haben.
- . . . in der Lage sein einen Signal-Handler einzurichten.
- . . . verstanden haben wie Signale blockiert werden können.