

U8 POSIX-Signale

- Zwei Arten von Signalen
 - ◆ synchrone Signale: durch Prozessaktivität ausgelöst (Analogie: Traps)
 - ◆ asynchrone Signale: "von außen" ausgelöst (Analogie: Interrupts)

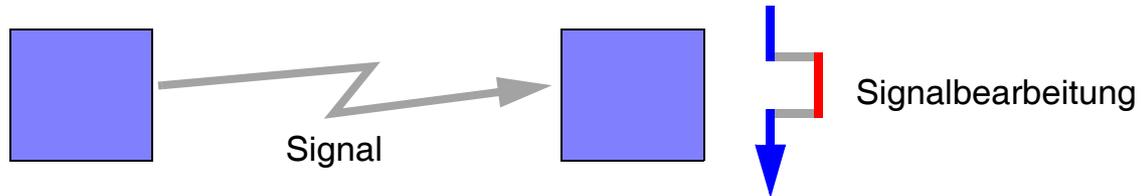
- Zwecke von Signalen
 - ◆ Ereignissignalisierung des Betriebssystemkerns an einen Prozess
 - ◆ Ereignissignalisierung zwischen Prozessen

1 Reaktion auf Signale

- abort
 - ◆ erzeugt Core-Dump (Segmente + Registerkontext) und beendet Prozess
- exit
 - ◆ beendet Prozess, ohne einen Core-Dump zu erzeugen
- ignore
 - ◆ ignoriert Signal
- stop
 - ◆ stoppt Prozess
- continue
 - ◆ setzt einen gestoppten Prozess fort
- signal handler
 - ◆ Aufruf einer Signalbehandlungsfunktion, danach Fortsetzung des Prozesses

2 POSIX-Signalbehandlung

- Signal bewirkt Aufruf einer Funktion



- ◆ nach der Behandlung läuft Prozess an unterbrochener Stelle weiter

- Systemschnittstelle

- ◆ sigaction
- ◆ sigprocmask
- ◆ sigsuspend
- ◆ kill

3 Signalhandler installieren: sigaction

■ Prototyp

```
#include <signal.h>

int sigaction(int sig, /* Signal */
              const struct sigaction *act, /* Handler */
              struct sigaction *oact /* Alter Handler */ );
```

- Handler bleibt solange installiert, bis ein neuer Handler mit `sigaction` installiert wird

■ sigaction-Struktur

```
struct sigaction {
    void (*sa_handler)(int); /* Behandlungsfunktion */
    sigset_t sa_mask; /* Signalmaske während der Behandlung */
    int sa_flags; /* Diverse Einstellungen */
}
```

3 Signalhandler installieren: sigaction Handler (`sa_handler`)

- Signalbehandlung kann über `sa_handler` eingestellt werden:
 - `SIG_IGN` Signal ignorieren
 - `SIG_DFL` Default-Signalbehandlung einstellen
 - *Funktionsadresse* Funktion wird in der Signalbehandlung aufgerufen und ausgeführt
 - ▮ `SIG_IGN` und `SIG_DFL` werden über **`exec(3)`** vererbt, nicht aber eine Behandlungsfunktion (nicht möglich, warum?)

- Mit `sa_flags` lässt sich das Verhalten beim Signalempfang beeinflussen
 - bei uns gilt: `sa_flags=SA_RESTART`

3 Signalhandler installieren: sigaction Maske (sa_mask)

- verzögerte Signale
 - ◆ während der Ausführung der Signalhandler-Prozedur wird das auslösende Signal blockiert
 - ◆ bei Verlassen der Signalbehandlungsroutine wird das Signal deblockiert
 - ◆ es wird maximal eine Zustellung je Signal zwischengespeichert

- mit `sa_mask` in der `struct sigaction` kann man zusätzliche Signale während der Behandlung des Signals blockieren

- Auslesen und Modifikation von Signal-Masken des Typs `sigset_t` mit:
 - ◆ `sigaddset()`: Signal zur Maske hinzufügen
 - ◆ `sigdelset()`: Signal aus Maske entfernen
 - ◆ `sigemptyset()`: Alle Signale aus Maske entfernen
 - ◆ `sigfillset()`: Alle Signale in Maske aufnehmen
 - ◆ `sigismember()`: Abfrage, ob Signal in Maske enthalten ist

3 Signalhandler installieren: Beispiel

■ Beispiel:

```
#include <signal.h>
void my_handler(int sig) { ... }
...
struct sigaction action;
sigemptyset(&action.sa_mask);
action.sa_flags = SA_RESTART;
action.sa_handler = my_handler;
sigaction(SIGUSR1, &action, NULL);
```

4 Signal zustellen

■ Systemaufruf **kill(2)**

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

■ Kommando **kill(1)** aus der Shell (z. B. **kill -USR1 <pid>**)

5 Ausgewählte POSIX-Signale

Das Defaultverhalten bei den meisten Signalen ist die Terminierung des Prozesses, bei einigen Signalen mit Anlegen eines Core-Dumps.

- **SIGALRM**: Timer abgelaufen (`alarm(2)`, `setitimer(2)`)
- **SIGCHLD** (ignore): Statusänderung eines Kindprozesses
- **SIGINT**: Interrupt; (Shell: CTRL-C)
- **SIGQUIT** (core): Quit; (Shell: CTRL-\)
- **SIGKILL** (nicht behandelbar): beendet den Prozess
- **SIGTERM**: Terminierung; Standardsignal für `kill(1)`
- **SIGSEGV** (core): Speicherschutzverletzung
- **SIGUSR1**, **SIGUSR2**: Benutzerdefinierte Signale

6 Ändern der prozessweiten Signal-Maske

```
int sigprocmask(int how, /* Verknüpfung der Masken */
               const sigset_t *set, /* neue Maske */
               sigset_t *oset /* Speicher für alte Maske */ );
```

■ how:

- ◆ **SIG_BLOCK**: blockiert Signale (Maske |= *set)
- ◆ **SIG_SETMASK**: blockiert Signale der Maske (Maske = *set)
- ◆ **SIG_UNBLOCK**: deblockiert Signale der Maske (Maske &= ~(*set))

■ Beispiel

```
sigset_t set;
sigemptyset(&set);
sigaddset(&set, SIGUSR1);
sigprocmask(SIG_BLOCK, &set, NULL); /* Blockiert SIGUSR1 */
```

- Anwendung: Sperren der Signalbehandlung in kritischen Abschnitten
Vgl.: Sperren der Interruptbehandlung in kritischen Abschnitten (**cli()**, **sei()**)
- Die prozessweite Signal-Maske wird über **exec (3)** vererbt.

7 Warten auf Signale

- Problem: Prozess will in einem kritischen Abschnitt auf ein Signal warten
 - Signal deblockieren
 - Auf Signal warten
 - Signal blokieren
 - Kritischen Abschnitt bearbeiten
- Operationen müssen atomar am Stück ausgeführt werden!
 - ➡ gleiche Problematik wie bei den Stromsparmodi des AVR-Prozessors
- Sigsuspend

```
#include <signal.h>
int sigsuspend(const sigset_t *mask);
```

- (1) `sigsuspend(mask)` setzt `mask` als Signal-Maske
- (2) Der Prozess blockiert bis zum Eintreffen eines Signals
- (3) Gegebenenfalls wird der Signalhandler ausgeführt
- (4) `sigsuspend` restauriert die ursprüngliche Signal-Maske und kehrt zurück

8 POSIX-Signale vs. Interrupts

	Signale	Interrupts
Behandlung installieren	sigaction()	ISR () -Makro der C-Bibliothek
Behandlungsfunktion	Signalhandler	Interrupthandler
Auslösung	durch Prozesse mit kill() oder durch das Betriebssystem	durch die Hardware
Synchronisation	sigprocmask()	cli(), sei()
Warten auf IRQ/Signal	pause()	sleep_cpu()
	sigsuspend()	sei() + sleep_cpu()

- Signale und Interrupts sind sehr ähnliche Konzepte auf unterschiedlichen Ebenen
- Viele Probleme treten in beiden Fällen auf und sind konzeptionell identisch zu lösen