

Übungen zu Systemnahe Programmierung in C (SPiC)

Moritz Strübe, Rainer Müller
(Lehrstuhl Informatik 4)



Sommersemester 2013



Signale

Allgemein

Signale zustellen

Signale maskieren

Signale behandeln

Auf Signale warten

Signale vs. Interrupts



- Vergleichbar mit Interrupts beim AVR
- Verwendung von Signalen
 - Ereignissignalisierung des Betriebssystemkerns an einen Prozess
 - Ereignissignalisierung zwischen Prozessen
- Zwei Arten von Signalen
 - synchrone Signale: durch Prozessaktivität ausgelöst (Trap / Falle)
⇒ Zugriff auf ungültigen Speicher, ungültiger Befehl
 - asynchrone Signale: “von außen“ ausgelöst (Interrupts / Unterbrechung)
⇒ Timer, Tastatureingabe



Reaktion auf Signale

- Ign
⇒ ignoriert Signal
- Term
⇒ beendet Prozess
- Core
⇒ erzeugt Core-Dump (Speicherabbild) und beendet Prozess
- Stop
⇒ hält Prozess an
- Cont
⇒ setzt einen angehaltenen Prozess fort
- Signal Handler
⇒ Aufruf einer Signalbehandlungsfunktion, danach Fortsetzung des Prozesses



Ausgewählte POSIX-Signale

- Das Standardverhalten bei den meisten Signalen ist die Terminierung des Prozesses, bei einigen Signalen mit Anlegen eines Core-Dumps
 - SIGALRM (Term): Timer abgelaufen (`alarm(2)`, `setitimer(2)`)
 - SIGCHLD (Ign): Statusänderung eines Kindprozesses
 - SIGINT (Term): Interrupt (Shell: `CTRL-C`)
 - SIGQUIT (Core): Quit (Shell: `CTRL-@`)
 - SIGKILL (nicht behandelbar): beendet den Prozess
 - SIGTERM (Term): Terminierung; Standardsignal für `kill(1)`
 - SIGSEGV (Core): Speicherschutzverletzung
 - SIGUSR1, SIGUSR2 (Term): Benutzerdefinierte Signale
- Siehe auch `signal(7)`



Signal zustellen

- Kommando `kill(1)` aus der Shell

```
1 kill -USR1 <pid>
```

- Parameter: Signalnummer oder Signal ohne "SIG"

- Systemaufruf `kill(2)`

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



Setzen der prozessweiten Signal-Maske

- Konfiguration mit Hilfe einer Variablen vom Typ `sigset_t`
- Der Aufbau dieser Variablen ist nicht festgeschrieben. Daher muss sie mit Hilfsfunktionen konfiguriert werden:
 - `sigemptyset(3)`: Alle Signale aus Maske entfernen
 - `sigfillset(3)`: Alle Signale in Maske aufnehmen
 - `sigaddset(3)`: Signal zur Maske hinzufügen
 - `sigdelset(3)`: Signal aus Maske entfernen
 - `sigismember(3)`: Abfrage, ob Signal in Maske enthalten ist



Setzen der prozessweiten Signal-Maske

■ Setzen einer Maske mit

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

- how: Operation
 - SIG_SETMASK: setzt set
 - SIG_BLOCK: blockiert zusätzlich die in set gesetzten Signale
 - SIG_UNBLOCK: deblockiert die in set gesetzten Signale
- set: Parameter für die Operation
- oset: Speicher für aktuell installierte Maske

■ Beispiel

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

■ Anwendung: z.B. kritische Abschnitte (vgl. cli(), sei())

!! Die prozessweite Signal-Maske wird über exec(3) vererbt !!



sigaction - Signalhandler

■ Konfiguration mit Hilfe der Struktur sigaction

```
1 struct sigaction {  
2     void (*sa_handler)(int); /* Behandlungsfunktion */  
3     sigset_t sa_mask;       /* Signalmaske während der Behandlung */  
4     int sa_flags;          /* Diverse Einstellungen */  
5 }
```

■ Signalbehandlung kann über sa_handler konfiguriert werden:

- SIG_IGN: Signal ignorieren
 - SIG_DFL: Default-Signalbehandlung einstellen
 - Funktionsadresse: Funktion wird in der Signalbehandlung aufgerufen
Als Parameter wird die Signalnummer übergeben
- SIG_IGN und SIG_DFL werden über exec(3) vererbt, nicht aber eine Behandlungsfunktion (nicht möglich, warum?)



sigaction - Maske

- Konfiguration mit Hilfe der Struktur sigaction

```
1 struct sigaction {  
2     void (*sa_handler)(int); /* Behandlungsfunktion */  
3     sigset_t sa_mask;       /* Signalmaske während der Behandlung */  
4     int sa_flags;          /* Diverse Einstellungen */  
5 }
```

- sa_mask wird während der Ausführung des Signalhandlers gesetzt
⇒ sigprocmask()



sigaction - Flags

- Konfiguration mit Hilfe der Struktur sigaction

```
1 struct sigaction {  
2     void (*sa_handler)(int); /* Behandlungsfunktion */  
3     sigset_t sa_mask;       /* Signalmaske während der Behandlung */  
4     int sa_flags;          /* Diverse Einstellungen */  
5 }
```

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



Setzen der Signalbehandlung

■ Konfiguration mit Hilfe der Struktur sigaction

```
1 struct sigaction {  
2     void (*sa_handler)(int); /* Behandlungsfunktion */  
3     sigset_t sa_mask;       /* Signalmaske während der Behandlung */  
4     int sa_flags;          /* Diverse Einstellungen */  
5 }
```

■ Konfiguration Setzen

```
1 #include <signal.h>  
2  
3 int sigaction(int sig, const struct sigaction *act,  
4               struct sigaction *oact);
```

- sig: Signalnummer
- act: Zu installierende Konfiguration
- oact: Speicher für die aktuell installierte Konfiguration



sigaction - Beispiel

■ sigaction

```
1 struct sigaction {  
2     void (*sa_handler)(int); /* Behandlungsfunktion */  
3     sigset_t sa_mask;      /* Signalmaske während der Behandlung */  
4     int sa_flags;          /* Diverse Einstellungen */  
5 }
```

■ Installieren eines Handlers für SIGUSR1

```
1 #include <signal.h>  
2  
3 void my_handler(int sig) {  
4     ...  
5 }  
6  
7 int main(int argc, char * argv[]){  
8     struct sigaction action;  
9     action.sa_handler = my_handler;  
10    sigemptyset(&action.sa_mask);  
11    action.sa_flags = SA_RESTART;  
12    sigaction(SIGUSR1, &action, NULL);  
13    ....
```



Warten auf Signale

- Problem: In einem kritischen Abschnitt auf ein Signal warten
 1. Signal deblockieren
 2. Passiv auf Signal warten (*schlafen legen*)
 3. Signal blockieren
 4. Kritischen Abschnitt bearbeiten
- Operationen müssen atomar am Stück ausgeführt werden!
⇒ gleiche Problematik wie bei den Stromsparmodi des AVR-Prozessors
- Sigsuspend

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

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



POSIX-Signale vs. AVR Interrupts

■ Vergleich

	Interrupts	Signale
Behandlung installieren	ISR()-Makro	<code>sigaction(2)</code>
Auslösung	Hardware	Prozesse mit <code>kill()</code> Betriebssystem
Synchronisation	<code>cli()</code> , <code>sei()</code>	<code>sigprocmask(2)</code>
Warten auf Signale	<code>sei(); sleep_cpu()</code>	<code>sigsuspend(2)</code>

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

