Übungen zu Systemnahe Programmierung in C (SPiC) – Sommersemester 2025

Übung 12

Maxim Ritter von Onciul
Eva Dengler

Lehrstuhl für Informatik 4
Friedrich-Alexander-Universität Erlangen-Nürnberg

Lehrstuhl für Informatik 4
Systemsoftware

Friedrich-Alexander-Universität
Technische Fakultät

Vorstellung Aufgabe 7

Signale

Signale

- Verwendung von Signalen
  - Ereignissignalisierung des Betriebssystemkerns an einen Prozess
  - Ereignissignalisierung zwischen Prozessen
- Vergleichbar mit Interrupts beim AVR
- Zwei Arten von Signalen
  - Synchrone Signale: Durch Prozessaktivität ausgelöst (Trap)
    $\Rightarrow$ Zugriff auf ungültigen Speicher, ungültiger Befehl
  - Asynchrone Signale: “Von außen” ausgelöst (Interrupt)
    $\Rightarrow$ Timer, Tastatureingabe
- Standardbehandlungen für Signale bereits vorhanden

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

```
01 kill -USR1 <pid>
```

  - Parameter: Signalnummer oder Signal ohne “SIG”

- Systemaufruf `kill(2)`

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

Setzen der prozessweiten Signalmaske

- Konfiguration mit Hilfe einer Variablen vom Typ `sigset_t`
- Hilfsfunktionen konfigurieren die Signalmaske
  - `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
- Gesetzte Signale werden blockiert
- AVR-Analogie: `EIMSK`-Register

Setzen der prozessweiten Signalmaske

- Setzen einer Maske mit

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

- how: Operation
  - SIG_SETMASK: Setzt eine absolute Signalmaske
  - SIG_BLOCK: Blockiert Signale relativ zur aktuell gesetzten Maske
  - SIG_UNBLOCK: Deblockiert Signale relativ zur aktuell gesetzten Maske
- oset: Speichert Kopie der vorherigen Signalmaske (optional)
- Die Signalmaske wird bei fork(2)/exec(3) vererbt
- Signalbehandlung kann über sa_handler konfiguriert werden:
  - SIG_IGN: Signal ignorieren
  - SIG_DFL: Default-Signalbehandlung einstellen
  - Funktionspointer
- SIG_IGN und SIG_DFL werden nach exec(3) beibehalten, Funktionspointer nicht. Warum?
- AVR-Analogie: ISR(..), Alarmhandler

Setzen der prozessweiten Signalmaske

Beispiel

```
01 sigset_t set;
02 sigemptyset(&set);
03 sigaddset(&set, SIGUSR1);
04 sigprocmask(SIG_BLOCK, &set, NULL); /* Blocks SIGUSR1 */
```

- AVR-Analogie: Sperren kritischer Abschnitte (`cli()`, `sei()`)

sigaction – Signalhandler

- Konfiguration mit Hilfe der Struktur `sigaction`

```
01 struct sigaction {
02   void (*sa_handler)(int); // Behandlungsfunktion
03   sigset_t sa_mask;        // Zusätzlich blockierte Signale
04   int sa_flags;            // Diverse Einstellungen
05 }
```

- Signalbehandlung kann über `sa_handler` konfiguriert werden:
  - `SIG_IGN`: Signal ignorieren
  - `SIG_DFL`: Default-Signalbehandlung einstellen
  - Funktionspointer
- `SIG_IGN` und `SIG_DFL` werden über `exec(3)` vererbt, Funktionspointer nicht. Warum?
- AVR-Analogie: `ISR(..)`, Alarmhandler

sigaction – Maske

- Konfiguration mit Hilfe der Struktur `sigaction`

```
01 struct sigaction {
02   void (*sa_handler)(int); // Behandlungsfunktion
03   sigset_t sa_mask;        // Zusätzlich blockierte Signale
04   int sa_flags;            // Diverse Einstellungen
05 }
```

- Während Signalbehandlung sind folgende Signale blockiert:
  - Signalmaske bei Eintreffen des Signals
  - Zusätzlich: Auslösendes Signal
  - Zusätzlich: Signale in `sa_mask`

⇒ Synchronisation mehrerer Signalhandler durch `sa_mask`

sigaction – Flags

- Konfiguration mit Hilfe der Struktur `sigaction`

```
01 struct sigaction {
02   void (*sa_handler)(int); // Behandlungsfunktion
03   sigset_t sa_mask;        // Zusätzlich blockierte Signale
04   int sa_flags;            // Diverse Einstellungen
05 }
```

- `man signal` zeigt die Funktion `signal()`, die NICHT verwendet werden sollte.
- `sa_flags` beeinflussen das Verhalten beim Signalempfang
- Bei uns gilt: `sa_flags=SA_RESTART`

Setzen der Signalbehandlung

- Konfiguration mit Hilfe der Struktur `sigaction`

```
01 struct sigaction {
02   void (*sa_handler)(int); // Behandlungsfunktion
03   sigset_t sa_mask;        // Zusätzlich blockierte Signale
04   int sa_flags;            // Diverse Einstellungen
05 }
```

- Konfiguration Setzen

```
01 #include <signal.h>
02
03 int sigaction(int sig, const struct sigaction *act,
04     struct sigaction *oact);
```

sigaction – Beispiel

```
01 struct sigaction {
02   void (*sa_handler)(int); // Behandlungsfunktion
03   sigset_t sa_mask;        // Zusätzlich blockierte Signale
04   int sa_flags;            // Diverse Einstellungen
05 }
```

- Installieren eines Handlers für SIGUSR1

```
01 #include <signal.h>
02 static void my_handler(int sig) {
03     // [...]
04 }
05
06 int main(int argv, char *argv[]) {
07     struct sigaction action;
08     action.sa_handler = my_handler;
09     sigemptyset(&action.sa_mask);
10     action.sa_flags = SA_RESTART;
11     sigaction(SIGUSR1, &action, NULL);
12     // [...]
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!

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

  1. `sigsuspend(2)` setzt temporäre Signalmaske
  2. Prozess blockiert bis zum Eintreffen eines Signals
  3. Signalhandler wird ausgeführt
  4. `sigsuspend(2)` stellt ursprüngliche Signalmaske wieder her
- AVR-Analogie: Schlafschleife, `sleep_cpu()`

sigsuspend – Beispiel

- SIGUSR1 im kritischen Abschnitt sperren
- Auf Signal warten

```
01 sigset_t sync_mask, old_mask;
02 sigemptyset(&sync_mask);
03 sigaddset(&sync_mask, SIGUSR1);
04
05 sigprocmask(SIG_BLOCK, &sync_mask, &old_mask);
06 while(!event) {
07     sigsuspend(&old_mask);
08 }
09 sigprocmask(SIG_SETMASK, &old_mask, NULL);
```

POSIX-Signale vs. AVR-Interrupts

Beschreibung	**Interrupts**	**Signale**
Behandlung installieren	`ISR()`-Makro	`sigaction(2)`
Auslöser	Hardware	Prozesse mit `kill(2)` oder Betriebssystem
Synchronisation	`cli()`, `sei()`	`sigprocmask(2)`
Warten auf Signale	`sei(); sleep_cpu()`	`sigsuspend(2)`

- Signale und Interrupts sind sehr **ähnliche Konzepte**
- Synchronisation ist oft konzeptionell identisch zu lösen

Aufgabe: mish - Teil b)

Signalbehandlung von SIGINT

- Anpassen der Signalbehandlungen für CTRL+C
- SIGINT wird allen Prozessen des Terminals zugestellt

```
01 $> ./mish
02 mish> sleep 2
03 Exit status [5321] = 0
04 mish> sleep 10000
05 ^C                          # CTRL+C
06 $>
```

⇒ Bei CTRL+C stirbt `sleep` und `mish`

- Anpassen der Signalbehandlung:
  - Elternteil: Signal ignorieren (SIG_IGN)
  - Kind: Default-Behandlung (SIG_DFL)

Aufgabe: mish - Teil b)

Aufsammeln von Zombieprozessen

- Bisher: Aufsammeln durch `waitpid(2)` (blockierend)
- Signal `SIGCHLD` zeigt Statusänderung von Kindprozessen an
  - Kindprozess wurde gestoppt
  - Kindprozess ist terminiert
- Jetzt: Aufsammeln durch `waitpid(2)` (nicht-blockierend)
- Warten auf Statusveränderungen mit `sigsuspend(2)`

Aufgabe: mish - Teil c)

Unterstützung von Hintergrundprozessen

- Kommandos mit abschließenden '&'

  ⇒ Hintergrundprozess
- Beispiel: `./sleep 10 &`
- Ausgabe der Prozess-ID und des Prompts
- Anschließend sofort Entgegennahme neuer Befehle

```
01 # Starten eines Hintergrundprozesses mit &
02 mish> sleep 10 &
03 Started [2110]
04 mish> ls
05 Makefile mish mish.c
06 Exit Status [2115] = 0
07 ...
08 Exit status [2110] = 0
```

Aufgabe: mish - Teil c)

Unterstützung von Hintergrundprozessen

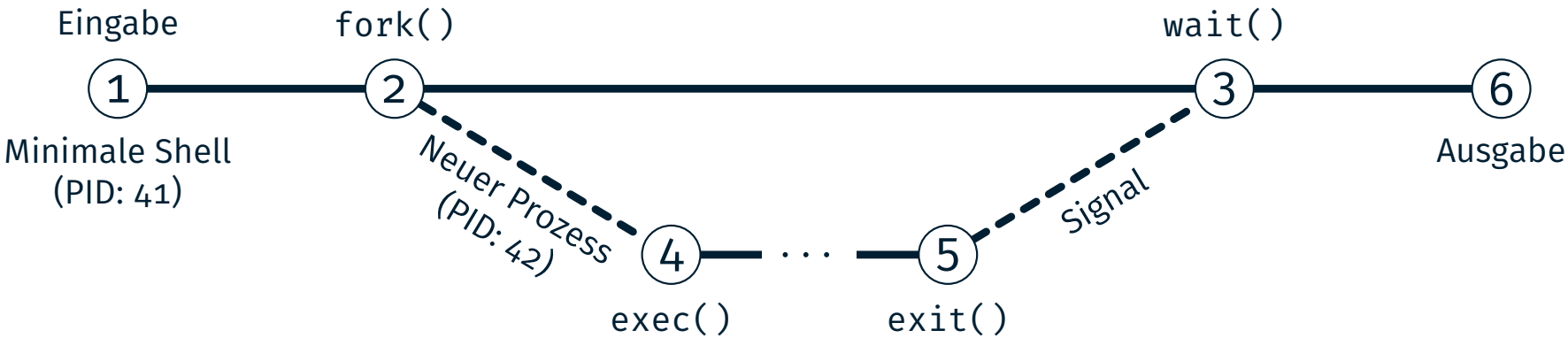
- Beim Warten auf Vordergrundprozesse sollen terminierende Hintergrundprozesse sofort eingesammelt werden

```
01 # Starten mehrerer Hintergrundprozesse
02 mish> sleep 3  &
03 Started [2110]
04 mish> sleep 5  &
05 Started [2115]
06 mish> sleep 10 &
07 Started [2118]
08
09 # Starten eines Vordergrundprozesses
10 mish> sleep 20
11 Exit Status [2110] = 0    # sleep 3  &
12 Exit Status [2115] = 0    # sleep 5  &
13 Exit Status [2118] = 0    # sleep 10 &
14 Exit Status [2121] = 0    # sleep 20
15 mish>
```

Aufgabe: mish - Teil c)

- Erweiterung des Basisablaufs

1. Auf Eingaben vom Benutzer warten
2. Neuen Prozess erzeugen
3. Elternteil: Wartet auf die Beendigung des Kindes *(Nur Vordergrund)*
4. Kind: Startet Programm
5. Kind: Programm terminiert
6. Elternteil: Ausgabe der Kindzustands



Klausurvorbereitung

Welche Klausur wollen wir nächste Woche besprechen?

Hands-on: Stoppuhr

Screencast: `https://www.video.uni-erlangen.de/clip/id/19835`

Hands-on: Stoppuhr

```
$ ./stoppuhr
Press Ctrl+C (SIGINT) to start and stop
^CStarted...
1 sec
2 sec
3 sec
4 sec
^CStopped.
Duration: 4 sec 132 msec
```

- Ablauf:
  - Stoppuhr startet durch `SIGINT` Signal
    - → Gibt jede Sekunde die bisherige Dauer aus (Format: “3 sec”)
  - Stoppuhr stoppt bei weiterem `SIGINT` und gibt Dauer aus
    - → Gibt Gesamtdauer inkl. Millisekunden aus (Format: “4 sec 132 msec”)
    - → Beendet sich anschließend
- Verwendet intern `SIGALRM` und `setitimer(2)`
- Schutz kritischer Abschnitte beachten

Wiederholung Signale

1. Signalhandler installieren: `sigaction(2)`

```
01 struct sigaction act;
02 act.sa_handler = SIG_DFL; // Handlersignatur: void f(int signum)
03 act.sa_flags = SA_RESTART;
04 sigemptyset(&act.sa_mask);
05 sigaction(SIGINT, &act, NULL);
```

2. Signale blockieren/deblockieren: `sigprocmask(2)`

```
01 sigset_t set;
02 sigemptyset(&set);
03 sigaddset(&set, SIGUSR1);
04 sigprocmask(SIG_BLOCK, &set, NULL); /* Blockiert SIGUSR1 */
05 // kritischer Abschnitt
06 sigprocmask(SIG_UNBLOCK, &set, NULL); /* Deblockiert SIGUSR1 */
```

Wiederholung Signale

3. Auf Signale warten: `sigsuspend(2)`

```
01 sigprocmask(SIG_BLOCK, &set, &old); /* Blockiert Signale */
02 while(event == 0){
03   sigsuspend(&old); /* Wartet auf Signale */
04 }
05 sigprocmask(SIG_SETMASK, &old, NULL); /* Deblockiert Signale */
```

Alarme mit `setitimer` (1)

- Zeitgeber mittels `setitimer(2)` konfigurieren

```
01 #include <sys/time.h>
02
03 int setitimer(int which, const struct itimerval *new_value,
04               struct itimerval *old_value);
```

- Parameter:

  `which` Hier: `ITIMER_REAL` (Physikalische Zeit)
  `new_value` Zu setzende Konfiguration
  `old_value` Zum Auslesen der vorherigen Konfiguration

- `SIGALRM`: Timer ist abgelaufen bzw. Alarm eingetreten
  → Standardbehandlung: Programm beenden
  → Eigenen Signalhandler installieren

Alarme mit `setitimer` (2)

- Strukturen zur Konfiguration

```
01 struct timeval {
02     time_t      tv_sec;         /* seconds */
03     suseconds_t tv_usec;        /* microseconds */
04 };
```

Beschreibt Zeitintervall mit `tv_sec` s und `tv_usec` μs

```
01 struct itimerval {
02     struct timeval it_interval; /* Interval for periodic timer */
03     struct timeval it_value;    /* Time until next expiration */
04 };
```

Erster Alarm nach Intervall `it_value`
danach periodischer Alarm mit Intervall `it_interval`

- Besondere Werte

`it_interval = {0, 0}` Singleshot Alarm
`it_value = {0, 0}` Alarm abbrechen