Übungen zu Systemnahe Programmierung in C (SPiC) – Sommersemester 2019

Übung 11

Benedict Herzog
Bernhard Heinloth

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

Lehrstuhl für Verteilte Systeme und Betriebssysteme

FRIEDRICH-ALEXANDER UNIVERSITÄT ERLANGEN-NÜRNBERG
TECHNISCHE FAKULTÄT

Vorstellung Aufgabe 7

Signale

Signale

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

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 Signal-Maske

- Konfiguration mit Hilfe einer Variablen vom Typ `sigset_t`
- Hilfsfunktionen konfigurieren das Signalset
  - `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
- AVR-Analogie: `EIMSK`-Register

Setzen der prozessweiten Signal-Maske

- 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

Beispiel

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

- AVR-Analogie: Sperren kritische 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;        // Signalmaske während der Behandlung
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;        // Signalmaske während der Behandlung
04   int sa_flags;            // Diverse Einstellungen
05 }
```


sigaction - Flags

- Konfiguration mit Hilfe der Struktur `sigaction`

```
01 struct sigaction {
02   void (*sa_handler)(int); // Behandlungsfunktion
03   sigset_t sa_mask;        // Signalmaske während der Behandlung
04   int sa_flags;            // Diverse Einstellungen
05 }
```

- `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;        // Signalmaske während der Behandlung
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

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

- Installieren eines Handlers für `SIGUSR1`

```c
#include <signal.h>

void my_handler(int sig) {
    ...
}

int main(int argv, char * argv[]){
    struct sigaction action;
    action.sa_handler = my_handler;
    sigemptyset(&action.sa_mask);
    action.sa_flags = SA_RESTART;
    sigaction(SIGUSR1, &action, NULL);
    ....
```


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!

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

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

POSIX-Signale vs. AVR-Interrupts

Beschreibung	Interrupts	Signale
Behandlung installieren	`ISR()`-Makro	`sigaction(2)`
Auslöser	Hardware	Prozesse mit `kill()` 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`

Aufgabe: mish - Teil b)

Signalbehandlung von SIGINT

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

```
$> ./mish
mish> sleep 2
Exit status [5321] = 0
mish> sleep 10000
^C                          # CTRL+C
$>
```

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

- Anpassen der Signalbehandlung:
  - Vater: 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

```
# Starten eines Hintergrundprozesses mit &
mish> sleep 10 &
Started [2110]
mish> ls
Makefile mish mish.c
Exit Status [2115] = 0
...
Exit status [2110] = 0
```


Aufgabe: mish - Teil c)

Unterstützung von Hintergrundprozessen

- Beim Warten auf Vordergrundprozesse sollen terminierende Hintergrundprozesse sofort eingesammelt werden

```
# Starten mehrerer Hintergrundprozesse
mish> sleep 3  &
Started [2110]
mish> sleep 5  &
Started [2115]
mish> sleep 10 &
Started [2118]

# Starten eines Vordergrundprozesses
mish> sleep 20
Exit Status [2110] = 0    # sleep 3  &
Exit Status [2115] = 0    # sleep 5  &
Exit Status [2118] = 0    # sleep 10 &
Exit Status [2121] = 0    # sleep 20
mish>
```

Klausurvorbereitung

Welche Klausur wollen wir nächste Woche besprechen?