Übungen zu Systemnahe Programmierung in C

Abschnitt 13.1: Signale

20.07.2020

Tim Rheinfels
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

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

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;        // 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 }
```

- `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     if(sigaction(SIGUSR1, &action, NULL) != 0) {
12         // [...] Fehlerbehandlung
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()` 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