Exercises in System Level Programming (SLP) – Summer Term 2024

Exercise 12

Maximilian Ott

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

Chair in Distributed Systems and Operating Systems

FRIEDRICH-ALEXANDER UNIVERSITÄT ERLANGEN-NÜRNBERG
FACULTY OF ENGINEERING

Presentation Assignment 7

Signals

Signals

- Usage of signals
  - Signaling kernel events to a process
  - Signaling events between processes
- Similar to interrupts on AVR
- Two types of signals
  - Synchronous signals: Triggered by process activity (trap)
    ⇒ Access to invalid memory, invalid instruction
  - Asynchronous signals: Triggered “from outside” (interrupt)
    ⇒ Timer, keyboard input
- Default signal handlers already defined

Selected POSIX-Signals

- The standard behavior for most signals is the termination of the process, some signals additionally create a core dump.
  - `SIGALRM` (`Term`): Alarm clock (`alarm(2)`, `setitimer(2)`)
  - `SIGCHLD` (`Ign`): Child process terminated, stopped, or continued
  - `SIGINT` (`Term`): Terminal interrupt signal (Shell: `CTRL-C`)
  - `SIGQUIT` (`Core`): Terminal quit signal (Shell: `CTRL-\`)
  - `SIGKILL` (cannot be caught or ignored): Kill
  - `SIGTERM` (`Term`): Termination signal; standard signal of `kill(1)`
  - `SIGSEGV` (`Core`): Invalid memory reference
  - `SIGUSR1`, `SIGUSR2` (`Term`): User-defined signal `1/2`
- Refer to `signal(7)`

Sending Signals

- Shell command `kill(1)`

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

  - Parameter: Signal number or signal without “SIG” prefix
- System call `kill(2)`

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


Setting a Process Wide Signal Mask

- Configuration with the help of a variable of the type `sigset_t`
- Helper functions configure the signal mask
  - `sigemptyset(3)`: Remove all signals from a mask
  - `sigfillset(3)`: Add all signals to a mask
  - `sigaddset(3)`: Add one signal to a mask
  - `sigdelset(3)`: Remove one signal from a mask
  - `sigismember(3)`: Query, whether a signal is included in a mask
- Set signals are blocked
- AVR analogue: `EIMSK`-register

Setting a Process Wide Signal Mask

- Setting the mask with

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

- how: Operation
  - `SIG_SETMASK`: Sets an absolute signal mask
  - `SIG_BLOCK`: Blocks signals relative to the current mask
  - `SIG_UNBLOCK`: Unblocks signals relative to the current mask
- `oset`: Stores copy of old signal mask (optional)
- The signal mask is inherited when using `fork(2)`/`exec(3)`

Examples

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

- AVR analogue: Blocking critical sections (`cli()`, `sei()`)

sigaction – Signal Handler

- Configuration using the struct `sigaction`

```
01 struct sigaction {
02   void (*sa_handler)(int); // Handler function
03   sigset_t sa_mask;        // Additionally blocked signals
04   int sa_flags;            // More settings
05 }
```

- Signal handler can be configured with `sa_handler`:
  - `SIG_IGN`: Ignore signal
  - `SIG_DFL`: Set to default signal handler
  - Function pointer
- `SIG_IGN` and `SIG_DFL` can be inherited with `exec(3)`, function pointers can't. Why?
- AVR analogue: `ISR(..)`, alarm handler

sigaction – Mask

- Configuration with the help of the struct `sigaction`

```
01 struct sigaction {
02   void (*sa_handler)(int); // Handler function
03   sigset_t sa_mask;        // Additionally blocked signals
04   int sa_flags;            // More settings
05 }
```

- During the handling of a signal, following signals are disabled:
  - Signal mask upon the signal occurred
  - Additionally: Triggered signal
  - Additionally: Signals in `sa_mask`

⇒ Synchronization of multiple signal handlers with `sa_mask`

sigaction – Flags

- Configuration with the help of the struct `sigaction`

```
01 struct sigaction {
02   void (*sa_handler)(int); // Handler function
03   sigset_t sa_mask;        // Additionally blocked signals
04   int sa_flags;            // More settings
05 }
```

- `sa_flags` influence the behavior when the signal is received
- For SLP: `sa_flags=SA_RESTART`

Setting the Signal Handler

- Configuration with the help of the struct `sigaction`

```
01 struct sigaction {
02   void (*sa_handler)(int); // Handler function
03   sigset_t sa_mask;        // Additionally blocked signals
04   int sa_flags;            // More settings
05 }
```

- Applying the configuration

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

sigaction – Example

```c
struct sigaction {
  void (*sa_handler)(int); // Handler function
  sigset_t sa_mask;        // Additionally blocked signals
  int sa_flags;            // More settings
}
```

- Installing a handler for SIGUSR1

```c
#include <signal.h>
static 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);
    // [...]
}
```


Waiting for Signals

- Problem: Waiting for a signal inside a critical section
  1. Unblock the signal
  2. *Passively* wait for the signal (go to *sleep mode*)
  3. Block signal
  4. Execute critical section
- Operations have to be executed atomically as one!

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

  1. `sigsuspend(2)` sets a temporary signal mask
  2. Process is blocked until a signal is received
  3. Signal handler is executed
  4. `sigsuspend(2)` restores the original signal mask
- AVR analogue: Sleep loop, `sleep_cpu()`

sigsuspend – Example

- Block SIGUSR1 inside the critical section
- Wait for the signal

```c
sigset_t sync_mask, old_mask;
sigemptyset(&sync_mask);
sigaddset(&sync_mask, SIGUSR1);

sigprocmask(SIG_BLOCK, &sync_mask, &old_mask);
while(!event) {
    sigsuspend(&old_mask);
}
sigprocmask(SIG_SETMASK, &old_mask, NULL);
```


POSIX-Signals vs. AVR-Interrupts

Description	Interrupts	Signals
Install handler	`ISR()` macro	`sigaction(2)`
Trigger	Hardware	Processes with `kill(2)` or operating system
Synchronization	`cli()`, `sei()`	`sigprocmask(2)`
Waiting for signals	`sei(); sleep_cpu()`	`sigsuspend(2)`

- Signals and interrupts are **similar concepts**
- Synchronization can usually be implemented identical

Task: `mish`

Task: mish - Part b)

Handling the signal `SIGINT`

- Configuring the signal handler for `CTRL+C`
- `SIGINT` is send to all processes in the terminal

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

⇒ On `CTRL+C` both `sleep` and `mish` get terminated

- Changing the signal handler:
  - Parent: ignore the signal (`SIG_IGN`)
  - Child: default behaviour (`SIG_DFL`)

Task: mish - Part b)

Collection of zombie processes

- Until now: collection with `waitpid(2)` (blocking)
- Signal `SIGCHLD` indicates that a child process changed its state
  - child process got stopped
  - child process terminated
- Now: collection with `waitpid(2)` (not blocking)
- Waiting for the change of state with `sigsuspend(2)`

Task: mish - Part c)

Support for background processes

- Commands with trailing '&'
  ⇒ background process
- Example: `./sleep 10 &`
- Output of the process ID and the prompt
- Afterwards new commands should be receivable

```
# Starting a background process with &
mish> sleep 10 &
Started [2110]
mish> ls
Makefile mish mish.c
Exit Status [2115] = 0
...
Exit status [2110] = 0
```

Task: mish - Part c)

Support for background processes

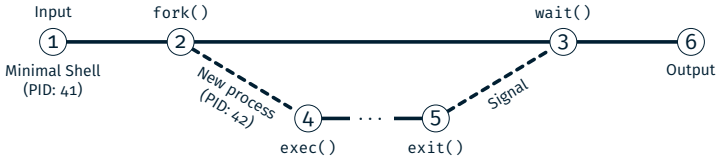
- While waiting for the termination of foreground processes, all terminating background processes should be collected immediately

```
01 # Starting multiple background processes
02 mish> sleep 3  &
03 Started [2110]
04 mish> sleep 5  &
05 Started [2115]
06 mish> sleep 10 &
07 Started [2118]
08
09 # Starting a foreground process
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>
```


Task: mish - Part c)

- Extension of the basic cycle

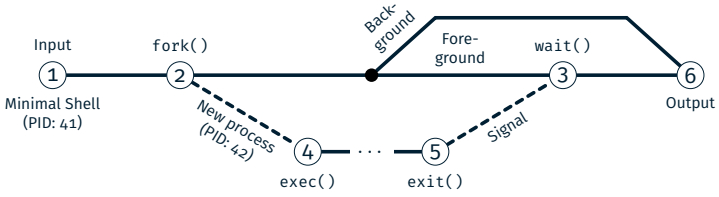
1. Waiting for input from the user
2. Creating a new process
3. Parent: Waiting for the termination of the child
4. Child: Starting program
5. Child: Program terminates
6. Parent: Outputting the state of the child



Task: mish - Part c)

- Extension of the basic cycle

1. Waiting for input from the user
2. Creating a new process
3. Parent: Waiting for the termination of the child *(only foreground)*
4. Child: Starting program
5. Child: Program terminates
6. Parent: Outputting the state of the child



Exam Preparation

Next Week: Mock Exam (link to PDF will be on the website)

Hands-on: Stopwatch

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

Hands-on: Stopwatch

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

- Procedure:
  - Stopwatch is started by signal `SIGINT`
    - → Each second, the current duration is printed (format: "3 sec")
  - Stopwatch is stopped again by the next occurrence of `SIGINT`
    - → Prints duration incl. milliseconds (format: "4 sec 132 msec")
    - → Terminates afterwards
- Internally, `SIGALRM` and `setitimer(2)` are used
- Remember to protect critical sections

Recap: Signals

1. Install signal handler: `sigaction(2)`

```
struct sigaction act;
act.sa_handler = SIG_DFL; // Signature of the handler: void f(int
    ↪  signum)
act.sa_flags = SA_RESTART;
sigemptyset(&act.sa_mask);
sigaction(SIGINT, &act, NULL);
```

2. Blocking/Unblocking of signals: `sigprocmask(2)`

```
sigset_t set;
sigemptyset(&set);
sigaddset(&set, SIGUSR1);
sigprocmask(SIG_BLOCK, &set, NULL); /* Blocks SIGUSR1 */
// critical section
sigprocmask(SIG_UNBLOCK, &set, NULL); /* Unblocks SIGUSR1 */
```


Recap: Signals

3. Waiting for signals: `sigsuspend(2)`

```
sigprocmask(SIG_BLOCK, &set, &old); /* Blocks signals */
while(event == 0){
  sigsuspend(&old); /* Waits for signals */
}
sigprocmask(SIG_SETMASK, &old, NULL); /* Unblocks signals */
```

Alarms with `setitimer` (1)

- Configure timer with `setitimer(2)`

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

- Parameters:

  `which` Here: `ITIMER_REAL` (physical time)
  `new_value` Setting the new Configuration
  `old_value` Reading the old configuration

- `SIGALRM`: Timer is expired or alarm occurred
  → Default handling: terminate program
  → Install custom signal handler

Alarms with `setitimer` (2)

- Structures for configuration

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

Describes time interval with `tv_sec` s and `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 };
```

First alarm after interval `it_value`
afterwards periodic alarm with interval `it_interval`

- Special values

  `it_interval = {0, 0}` Single shot alarm
  `it_value = {0, 0}` Cancel alarm