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)

- kill -USR1 <pid>

Parameter: Signal number or signal without "SIG" prefix

System call: kill(2)

- int kill(pid_t pid, int signo);

Setting a Process Wide Signal Mask

Setting the mask with:

- 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

- sigset_t set;
- sigemptyset(&set);
- sigaddset(&set, SIGUSR1);
- sigprocmask(SIG_BLOCK, &set, NULL); /* Blocks SIGUSR1 */

- AVR analogue: Blocking critical sections (cli(), sei())
Configuration using the struct `sigaction`

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

- 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

Configuration with the help of the struct `sigaction`

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

- 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`

Configuration with the help of the struct `sigaction`

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

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

Applying the configuration

```c
#include <signal.h>
int sigaction(int sig, const struct sigaction *act, 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
};
```

**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>
static void my_handler(int sig) {
    // [...]
}
```

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

**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**

<table>
<thead>
<tr>
<th>Description</th>
<th>Interrupts</th>
<th>Signals</th>
</tr>
</thead>
<tbody>
<tr>
<td>Install handler</td>
<td>ISR() macro</td>
<td>sigaction(2)</td>
</tr>
<tr>
<td>Trigger</td>
<td>Hardware</td>
<td>Processes with kill(2) or operating system</td>
</tr>
<tr>
<td>Synchronization</td>
<td>cli(), sei()</td>
<td>sigprocmask(2)</td>
</tr>
<tr>
<td>Waiting for signals</td>
<td>sei(); sleep_cpu()</td>
<td>sigsuspend(2)</td>
</tr>
</tbody>
</table>

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

### Part b)

**Handling the signal SIGINT**

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

```bash
$> ./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)

---

**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)`

---

**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

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

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

```
# Starting multiple background processes
mish> sleep 3 &
Started [2110]
mish> sleep 5 &
Started [2115]
mish> sleep 10 &
Started [2118]
# Starting a foreground process
mish> sleep 20
```

```
Exit Status [2110] = 0 # sleep 3 &
Exit Status [2115] = 0 # sleep 5 &
Exit Status [2118] = 0 # sleep 10 &
mish>
```

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

Recap: Signals

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

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

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

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

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

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

Hands-on: Stopwatch

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

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
Alarms with `setitimer(1)`

Configure timer with `setitimer(2)`

1. #include <sys/time.h>
2. int setitimer(int which, const struct itimerval *new_value, 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

```c
struct timeval {
    time_t tv_sec; /* seconds */
    suseconds_t tv_usec; /* microseconds */
};
```

Describes time interval with `tv_sec` s and `tv_usec` µs

```c
struct itimerval {
    struct timeval it_interval; /* Interval for periodic timer */
    struct timeval it_value; /* Time until next expiration */
};
```

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