System-Level Programming

29 Signals

Peter Wägemann

Lehrstuhl für Informatik 4
Systemsoftware

Friedrich-Alexander-Universität
Erlangen-Nürnberg (FAU)

Summer Term 2025

http://sys.cs.fau.de/lehre/ss25

Signals

- Microcontrollers react to concurrent events (interrupts) with so-called interrupt service routines
- A similar concept exists on the level of processes: **signals**

Signals (2)

Interrupt: **asynchronous** signal triggered by an “external” event

- CTRL-C pressed on the keyboard
- timer expired
- child process terminated
- ...

Exception: **synchronous** signal triggered by an activity of a process

- access to invalid memory address
- illegal machine instruction
- division by 0
- write operation to a closed communication connection
- ...

Communication: a process sends an event to another process

Signals (3)

CTRL-C:

```
int main(void)
{
    while (1) {
    }
}
```

```
~> ./test
^C
~>
```

Illegal memory access:

```
int main(void)
{
    *(int *) NULL = 0;
    return 0;
}
```

```
~> ./test
Segmentation fault
~>
```

Inter-process communication:

```
int main(void)
{
    while (1) {
    }
}
```

```
~> ./test
Terminated
~>
```

```
~> killall test
~>
```

Reaction to Signals

abort:

creates a *core dump* (contents from memory and registers are saved in the file `./core`) and terminates the process
default setting for all exceptions (to make debugging easier)

exit:

terminates the process (without core dump)
default setting for e. g., CTRL-C, kill signal

ignore:

signal gets ignored
default setting for all "unimportant" signals (e. g., child process terminated, size of the terminal window has changed)

...

Reaction to Signals (2)

...

handler:

call to a signal handler function, continuation afterwards

no default setting possible since handling requires program-dependent behavior

stop:

stops the process

default setting for stop signal

continue:

continues a process

default setting for continue signal

reaction can be changed with system call (`sigaction`)

Programming Interface

- Configuration of the signal handler function
(equivalent to configuring a function as ISR)

```
#include <signal.h>

int sigaction(int sig, struct sigaction *new, struct sigaction *old);
```

`struct sigaction` contains:

```
void (*sa_handler)(int sig); /* handler function
                                or SIG_DFL or SIG_IGN */
sigset_t sa_mask;            /* list of blocked signals while
                                handler is executed */
int sa_flags;                /* 0 or SA_RESTART ... */
```

- ...

Programming Interface

- ...
- For solving concurrency problems: blocking/unblocking of signals necessary, equivalent to µController's `cli()`, `sei()`

```
#include <signal.h>

int sigprocmask(int how, sigset_t *nmask, sigset_t *omask);
```

  - `SIG_BLOCK`: block all given signals
  - `SIG_UNBLOCK`: unblock all given signals
  - `SIG_SETMASK`: set a signal mask
- Unblocking + passive waiting for signal + blocking again, equivalent to µController's `sei(); sleep_cpu(); cli();`

```
#include <signal.h>

int sigsuspend(sigset_t *mask);
```

- ...

Programming Interface

- ...
- Creating an empty signal list

```
#include <signal.h>

int sigemptyset(sigset_t *mask);
```

- Creating a full signal list

```
int sigfillset(sigset_t *mask);
```

- Adding one signal to an existing signal list

```
int sigaddset(sigset_t *mask, int sig);
```

- Removing one signal from an existing signal list

```
int sigdelset(sigset_t *mask, int sig);
```


Programming Interface

- Typical signals:

SIGSEGV: “segmentation fault” (invalid access to memory)
SIGINT: “interrupt” (CTRL-C)
SIGALRM: “alarm” (timer expired)
SIGCHLD: “child” (child process terminated)
SIGTERM: “terminate” (termination of the process; possible to handle in program)
SIGKILL: “kill” (termination of the process; impossible to handle in program)

Signal Example 1

```c
#include <signal.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>

int main(void)
{
  // Call handler when
  // CTRL-C signal is received.
  struct sigaction sa;
  sa.sa_handler = handler;
  sigfillset(&sa.sa_mask);
  sa.sa_flags = 0;
  sigaction(SIGINT, &sa, NULL);

  for (int i = 0; ; i++) {
    printf("%d\n", i);
  }
  return 0;
}
```

```c
static void handler(int sig)
{
    char s[] = "CTRL-C!\n";
    write(STDOUT_FILENO,
          s, strlen(s));
}
```

(error handling omitted...)

```
~> ./test
...
146431
146432
146433
14^CCTRL-C!
6434
146435
146436
...
~>
```

Signal Example 2

```
int main(void)
{
  // Call handler when
  // timer signal is received.
  struct sigaction sa;
  sa.sa_handler = handler;
  sigfillset(&sa.sa_mask);
  sa.sa_flags = 0;
  sigaction(SIGALRM, &sa, NULL);

  // Send timer signal every sec.
  struct itimerval it;
  it.it_value.tv_sec  = 1;
  it.it_value.tv_usec = 0;
  it.it_interval.tv_sec  = 1;
  it.it_interval.tv_usec = 0;
  setitimer(ITIMER_REAL, &it, NULL);

  // Wait for timer ticks.
  sigset_t mask;
  sigemptyset(&mask);
  while (1) sigsuspend(&mask);
}
```

```
static void handler(int sig)
{
  write(STDOUT_FILENO,
        "Tick\n", 5);
}
```

(error handling omitted...)

```
~> ./test
Tick
Tick
Tick
^C
~>
```

Signal Example 3

```c
#include <signal.h>
#include <unistd.h>

int main(void)
{
  // Call handler when
  // I/O signal is received.
  struct sigaction sa;
  sa.sa_handler = handler;
  sigfillset(&sa.sa_mask);
  sa.sa_flags = 0;
  sigaction(SIGIO, &sa, NULL);

  // Send I/O signal when
  // STDIN can be read.
  int flags = fcntl(STDIN_FILENO,
      F_GETFL);
  flags |= O_ASYNC;
  fcntl(STDIN_FILENO, F_SETFL,
      flags);

  while (1) sleep(1);
}
```

```c
static void handler(int sig)
{
  char buf[256];
  int len;

  // Read chars from STDIN.
  len = read(STDIN_FILENO, buf,
           sizeof(buf));

  // Handle chars in buf.
  ...
}
```

(error handling omitted...)

Signals and Concurrency

- Signals create **concurrency** inside processes
- Resulting problems are **analogous to concurrency of interrupts** on a microcontroller platform
- For example: lost update, lost wakeup, ...

Example of Concurrency

```c
int main(void) {
  struct sigaction sa;
  struct itimerval it;
  /* Setup timer tick handler. */
  sa.sa_handler = tick;
  sa.sa_flags = 0;
  sigfillset(&sa.sa_mask);
  sigaction(SIGALRM, &sa, NULL);
  /* Setup timer. */
  it.it_value.tv_sec = 1;
  it.it_value.tv_usec = 0;
  it.it_interval.tv_sec = 1;
  it.it_interval.tv_usec = 0;
  setitimer(ITIMER_REAL, &it, NULL);
  /* Print time while working. */
  while (1) {
    int s = sec, m = min, h = hour;
    printf("%02d:%02d:%02d\n", h, m, s);
    do_work();
  }
}
```

↓ signal here

```c
volatile int hour = 0;
volatile int min = 0;
volatile int sec = 0;

static void tick(int sig) {
  sec++;
  if (60 <= sec) {
    sec = 0; min++;
  }
  if (60 <= min) {
    min = 0; hour++;
  }
  if (24 <= hour) {
    hour = 0;
  }
}
```

```
~> ./test
...
23:59:59
00:59:59
00:00:00
...
```

← problem here!

(error handling omitted...)

Solution for the Concurrency Example

1. Solution

```
sigset_t nmask, omask;

/* Block SIGALRM. */
sigemptyset(&nmask);
sigaddset(&nmask, SIGALRM);
sigprocmask(SIG_BLOCK,
            &nmask, &omask);

/* Get current time. */
int s = sec, m = min, h = hour;

/* Restore signal mask. */
sigprocmask(SIG_SETMASK,
            &omask, NULL);

/* Print current time. */
printf("%02d:%02d:%02d\n",
       h, m, s);
```

2. Solution

```
/* Get current time. */
int s, m, h;
do {
  s = sec;
  m = min;
  h = hour;
} while (s != sec
      || m != min
      || h != hour);

/* Print current time. */
printf("%02d:%02d:%02d\n",
       h, m, s);
```

More solutions exist...

Problems with Concurrency

- Additional problem:
  internal functionality of library functions unknown in general
- Example 1:
  `printf` inserts character into a buffer
  ⇒ use of `printf` in the main program *and* in the signal handler is dangerous
- Example 2:
  `malloc` searches list for free memory area; `free` inserts a block into the list
  ⇒ use of `malloc`/`free` in the main program *and* in the signal handler is dangerous
- Solution:
  - block signals during the execution of **critical sections** or
  - no unknown library functions is called from inside a signal handler function