Lab-1. Signals

The signaling mechanism at the operating system level enables the processing of events that occur in parallel with the normal operation of the program, i.e. the process, i.e. its threads.

In this respect, the signal is similar to the processor-level interrupt mechanism: the processor executes a thread that can be interrupted by a device interrupt. The processor then suspends execution of the thread, saves its context, and jumps to the interrupt handler. After the interrupt handler completes, it returns and resumes the thread (restores its context). Similarly, the signals interrupt the execution of a thread, the signal processing function is called (the default function or a function defined in the program) and after its completion the processor returns to the thread and resumes its operation.

Let us consider the signals SIGINT (signal interrupt) and SIGTERM (terminate). The usual use of the signal SIGINT is to stop a process. Usually this is a "forced" interrupt due to an error in the execution of the process. On the other hand, SIGTERM is also used to interrupt the process, but for other reasons and not because of program errors. For example, when the system is shutting down, all processes must be stopped, but in a pleasant way. They are notified to stop with this signal. Then a corresponding function (SIGTERM handler) is called within a process, which can perform additional "housekeeping" before the process stops voluntarily.

In the terminal, we send SIGINT to the active process by pressing Ctrl + C and the process is terminated (default behavior). The signal can also be sent with special shell commands or other programs through the OS interface. With the kill command we can send a signal to a process whose identification number (PID) we know with:

```
$ kill -<signal id> <PID>
```

Signal SIGTERM can be sent to process with PID 2351 with command:

```
$ kill -SIGTERM 2351
```

Character `$` is command shell prefix, not part of the command.

For most signals, the program can specify what to do with them. If the program does not do this, the default behavior is used. In many cases this means that the process is stopped, like with the SIGINT and SIGTERM signals.

The program defines its behavior for signals through OS interface - it masks a signal, usually with a signal handler function (function defined in the program). There are several interfaces for this, such as the older *signal* and *sigset* functions and the newer *sigaction*, which is used in this lab.

In the next example, three signals are masked, SIGUSR1, SIGTERM, and SIGINT. The SIGUSR1 signal is a "user" signal that serves no particular purpose. Here SIGUSR1 is used to simulate an event where an action must be performed. The signal handler for SIGTERM and SIGINT, on the other hand, prints a message and stops the process. However, in the handler for SIGTERM, we only announce that programs need to be stopped by using a global variable run. When the process returns from this signal handler, it will recognize this change and exit the infinite loop.

Then next program is explained with comments within source code.

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

/* signal handlers declarations */
void proces_event(int sig);
void process_sigterm(int sig);
void process_sigint(int sig);

int run = 1;

int main()
{
    struct sigaction act;

    /* 1. masking signal SIGUSR1 */

    /* signal handler function */
    act.sa_handler = proces_event;

    /* additionally block SIGTERM in handler function */
    sigemptyset(&act.sa_mask);
    sigaddset(&act.sa_mask, SIGTERM);

    act.sa_flags = 0; /* advanced features not used */

    /* mask the signal SIGUSR1 as described above */
    sigaction(SIGUSR1, &act, NULL);

    /* 2. masking signal SIGTERM */
    act.sa_handler = process_sigterm;
    sigemptyset(&act.sa_mask);
    sigaction(SIGTERM, &act, NULL);

    /* 3. masking signal SIGINT */
    act.sa_handler = process_sigint;
    sigaction(SIGINT, &act, NULL);

    printf("Process with PID=%ld started\n", (long) getpid());

    /* processing simulation */
    int i = 1;
    while(run) {
        printf("Process: iteration %d\n", i++);
        sleep(1);
    }

    printf("Process with PID=%ld finished\n", (long) getpid());

    return 0;
}

void proces_event(int sig)
{
    int i;
    printf("Event processing started for signal %d (SIGINT)\n", sig);
    for (i = 1; i <= 5; i++) {
        printf("Processing signal %d: %d/5\n", sig, i);
        sleep(1);
    }
```

```
    printf("Event processing completed for signal %d (SIGINT)\n", sig);
}

void process_sigterm(int sig)
{
    printf("Received SIGTERM, saving data before exit\n");
    run = 0;
}

void process_sigint(int sig)
{
    printf("Received SIGINT, canceling process\n");
    exit(1);
}
```

For the demonstration shown below, two terminals must be open. On the first one a program is started and on the second one signals are sent with the command kill. The signal SIGINT can be sent directly with Ctrl+C. In the examples, the commands and printouts from both terminals are shown in separate columns.

Example 1. Sending signal `SIGINT` with `Ctrl+C`

Terminal 1	Terminal 2

`$ gcc lab1_example-signals.c -o sig1`
`$ ./sig1`
`Process with PID=14284 started`
`Process: iteration 1`
`Process: iteration 2`
`Process: iteration 3`
`^CReceived SIGINT, canceling process`
`$`	

In this first example, the key combination Ctrl+C is used to send a SIGINT signal. When the signal is received, the signal handler function is called, which simply prints a message and stops the process with a call to the exit function. The SIGINT signal can also be sent with the kill command, just like any other signal.

Example 2. Sending signal `SIGINT` with command `kill`

Terminal 1	Terminal 2

`$ ./sig1`
`Process with PID=14296 started`
`Process: iteration 1`
`Process: iteration 2`
`Process: iteration 3`
`Received SIGINT, canceling process`
`$`
	`$ kill -SIGINT 14296`

Similarly, other signal handlers are called for other signals.

Example 3. Sending signal `SIGTERM`

Terminal 1	Terminal 2

`$ ./sig1`
`Process with PID=14299 started`
`Process: iteration 1`
`Process: iteration 2`
`Process: iteration 3`
	`$ kill -SIGTERM 14299`

Received SIGTERM, saving data before exit Process with PID=14299 finished $	

Example 4. Sending signal `SIGUSR1`

Terminal 1	Terminal 2
$ ./sig1 Process with PID=14425 started Process: iteration 1 Process: iteration 2 Event processing started for signal 10 Processing signal 10: 1/5 Processing signal 10: 2/5 Processing signal 10: 3/5 Processing signal 10: 4/5 Processing signal 10: 5/5 Event processing completed for signal 10 Process: iteration 4 Process: iteration 5 ^CReceived SIGINT, canceling process $	$ kill -SIGUSR1 14425

When the SIGUSR1 signal is received, the execution of the main program is temporary suspended, and the signal handler function is called. After its completion, the program resumes its execution. If a new signal SIGUSR1 arrives while the program is in the signal handler function for this signal, this new signal is "paused" until the previous handling is completed. Only then is this new signal released and the signal handler called again. Since we have defined this in the program, SIGTERM will also be put on hold if it arrives while SIGUSR1 is being processed.

Example 5. Sending signals while in signal handler

Terminal 1	Terminal 2

$ ./sig1
Process with PID=14492 started
Process: iteration 1
Process: iteration 2
Event processing started for signal 10
Processing signal 10: 1/5
Processing signal 10: 2/5
Processing signal 10: 3/5
Processing signal 10: 4/5
Processing signal 10: 5/5
Event processing completed for signal 10
Event processing started for signal 10
Processing signal 10: 1/5
Processing signal 10: 2/5
Processing signal 10: 3/5
Processing signal 10: 4/5
Processing signal 10: 5/5
Event processing completed for signal 10
Received SIGTERM, saving data before exit
Process with PID=14492 finished
$
	$ kill -SIGUSR1 14492
	$ kill -SIGUSR1 14492
	$ kill -SIGTERM 14492

The second signal SIGUSR1 is processed only after the first has been processed. Similarly, the signal SIGTERM must wait until the second SIGURS1 handler has finished. However, if SIGINT would arrive it will immediately be handled since it is not on hold, even if SIGUSR1 is being processed. This behavior with SIGINT is not shown in the example.

The process can respond to a signal in several ways:

1. call the default handler if the program has not defined another,
2. call a handler function provided by the program (e.g., with *sigaction*),
3. hold signals (they will not interrupt, but are stored for later processing)
4. ignore (they will not interrupt nor are stored).

Behaviors 1, 2, and 4 can be set with *sigaction*: with the constant SIG _DFL for 1, the handler function for 2, and SIG _IGN for 4. Behavior 3 is set automatically when a signal is accepted with the handler function and is reset when the handler is finished. Additionally, behavior 3 can be set with *sighod* and reset with *sigrelse*.

Signals are sent to process by the operating system for its own reasons or at the request of another process. In the above examples, the signals were sent with the kill command (which is a program), or directly via Ctrl+C, which the shell interpreted as a request to send the signal SIGINT to the process shell started.

Many mechanisms in UNIX are based on signals. For example, periodic operations can be implemented by alarms - signals that are sent periodically (e.g., with *setitimer*). The delay operation *sleep(x)* is implemented by signals: the process asks OS to send a signal after x seconds (*alarm*), and then the program suspends itself (with *pause*). However, such a *sleep* can also be interrupted by other signals, so that the program is not delayed for a given number of seconds.

Lab assignment

Simulate interrupt handling on a system which have interrupt controller (IC) with three interrupt levels (three priorities: 1-3). Use signals to simulate interrupts (arbitrarily chose three signals). For example, the SIGINT signal simulates the highest level interrupt (3), the SIGUSR1 signal simulates the middle level interrupt (2), and the SIGTERM signal simulates the lowest level interrupt (1). Interrupts can be generated either via the keyboard (e.g., Ctrl+C) or by some other program (your own or with the command *kill*) that is started in a separate terminal. In addition to simulating the registers of the interrupt controller, an additional data structure should be used to simulate what is on the stack.

On start, the program must print the initial state before a single interrupt arrived. When an interrupt arrives, the interrupt level must be printed. When interrupt processing starts program must print the state of interrupt subsystem, including interrupt controller registers and context on the stack. After completing interrupt processing (simulate with a few seconds sleep), print new state.

Each print must be prefixed with current simulation time. An example of using time interface is shown in the example program *lab1_example-sleep.c*.

Example output (doesn't need to look like this, but must have all those information):

```
$ ./lab1
000.000:     Main program: started, PID=14497
000.000:     C_F=000, C_P=0, stack: -

002.041:     IC: new interrupt, priority: 2, forwarding to processor
002.041:     C_F=010, C_P=0, stack: -

002.041:     Interrupt handling started for priority: 2
```

```
002.041:     C_F=000, C_P=2, stack: 0, reg[0]

003.716:     IC: new interrupt, priority: 1, not forwarding to processor
003.716:     C_F=100, C_P=2, stack: 0, reg[0]

^C005.416:   IC: new interrupt, priority: 3, forwarding to processor
005.416:     C_F=101, C_P=2, stack: 0, reg[0]

005.416:     Interrupt handling started for priority: 3
005.416:     C_F=100, C_P=3, stack: 2, reg[2]; 0, reg[0]

010.416:     Interrupt handling completed for priority: 3
010.416:     Resuming interrupt handling for priority 2
010.416:     C_F=100, C_P=2, stack: 0, reg[0]

012.041:     Interrupt handling completed for priority: 2
012.041:     Resuming main program
012.041:     C_F=100, C_P=0, stack: -

012.041:     IC: C_P changed, forwarding interrupt request 1 to processor
012.041:     Interrupt handling started for priority: 1
012.041:     C_F=000, C_P=1, stack: 0, reg[0]

017.041:     Interrupt handling completed for priority: 1
017.041:     C_F=000, C_P=0, stack: -
...
```