Lab-1. Signals

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

In this way, the signal is similar to the interrupt mechanism at the processor level: the processor executes some action that is interrupted by the interrupt of some device. The processor then temporarily interrupts the execution of the program, saves its context, and jumps to device interrupt processing. At the end of interrupt processing, it returns and continues with the thread (restores its context). Similarly, signals interrupt the thread, the signal processing function (assumed or default in the program) is called, and after its completion, it returns to the thread and continues its work.

Let's consider examples of signalsSIGINT (*signal interrupt*) andSIGTERM (*terminate*). They are being processed SIGINT sends when you want to stop it from working. Most often it is a "forced" termination due to some error in the execution of the process. On the other side,SIGTERMit also serves to stop the process, but most often for other reasons, not because of program errors, for example when shutting down the system when all processes need to be shut down, especially those "in the background" (services).

In the terminal, we will send SIGINT to the active process with a key combinationCtrl+Cand the process will be terminated (usual behavior). The signal can also be sent by special shell commands or other programs through the OS interface. By commandkillwe can send the signal to the process if we know its identification number (PID) with:

$ kill -<signal_id> <PID>

Let the process have PID 2351 then muSIGTERMwe can send with:

$ kill -SIGTERM 2351

The $ sign represents a prefix on the shell command line, it is not part of the commands.

For most signals, the program can define itself what to do in case of receipt. If this is not done, the "normal" behavior will be used. For most signals, this will mean interrupting program execution, such as forSIGINTandSIGTERM.

A program defines behavior for a signal by telling the OS which program-defined function to call for a particular signal. There are several interfaces for defining program behavior for a signal. Oldersignalandsixsetare trying to be replaced by newer onessigaction,so it will be used in the example below.

In the example, three signals are maskedSIGUSR1, SIGTERMandSIGINT.SignalSIGUSR1is a "user" signal, with no special purpose. Here it is used as a simulation of the occurrence of an event that needs to be acted upon, which is simulated here.SIGTERMandSIGINTwill terminate the execution of the process after printing the message, with the fact that in this example withSIGTERMit does not come out immediately but only via a variableit's not the endmark that the work should be finished.

A description of the lines of code is provided within the program itself.

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

/* signal processing functions, listed below main */ voidprocess_event(int
voidprocess_sigterm(int voidprocess_sigint(int
                                        sig);
                                       sig);

intit's not the end=1;

intmain()
{
        structaction act;

        /* 1st SIGUSR1 signal masking */
        act.sa_handler=process_event;/*with which function the signal is processed */ sigemptyset(&act.
        with_mask);
        sigaddset(&act.with_mask,SIGTERM);/*also block SIGTERM during processing */ act.with_flags=0;/*more
        advanced features skipped */
        sigaction(SIGUSR1, &act,NULL);/*signal masking via the OS interface */

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


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

        printf("Program with PID=%ld started running\n", (long)getpid()); /* some work that
        the program does; only simulation here */ intand=1;

        while(it's not the end) {
                printf("Program: iteration %d\n",and++); sleep(1);

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

        return0;
}
voidprocess_event(int {                 sig)

        intand;
        printf("Beginning of signal processing %d\n",sig); for(and
        =1;and<=5;and++) {
                printf("Processing signal %d: %d/5\n",sig,and); sleep(1);

        }
        printf("End of signal processing %d\n",sig);
}
voidprocess_sigterm(intsig) {

        printf("Received SIGTERM signal, cleaning up before exiting the program\n"); it's not the end=0;

}
voidprocess_sigint(intsig) {

        printf("Received SIGINT signal, aborting\n"); exit(1);

}
```

For launch and demonstration, it is necessary to open two consoles/terminals, in one the program will be launched, and in the other with a commandkillsend signals, except for signalsSIGINTwhich can be sent directly fromCtrl+C.Examples of work are shown in two columns: on the left is a printout of the program, and on the right are the commands that were run in the other console at a given time.

Example 1. SendingSIGINT**with**Ctrl+C

Terminal 1	Terminal 2
$ gcc sig-example-1.c -o sig1 $ ./sig1 Program with PID=14284 started working Program: iteration 1 Program: iteration 2 Program: iteration 3 ^CReceived SIGINT signal, aborting $	

The first example shows sending a signalSIGINTvia abbreviationCtrl+C (the other terminal se in this example he did not use). When the signal was received, the default function was called, which printed out the corresponding function and finished working. The same signal can be sent by commandkilljust remember the PID of the process first.

Example 2. SendingSIGINT**with**kill

Terminal 1	Terminal 2
$ ./sig1 Program with PID=14296 started working Program: iteration 1 Program: iteration 2 Program: iteration 3 Received the SIGINT signal, I am terminating work $	$ kill -SIGINT 14296

It is similar with sending other signals.

Example 3. SendingSIGTERM

Terminal 1	Terminal 2
$ ./sig1 Program with PID=14299 started working Program: iteration 1 Program: iteration 2 Program: iteration 3 Received SIGTERM signal, clean up before exiting the program Program with PID=14299 finished working $	$ kill -SIGTERM 14299

Example with expected processing forSIGUSR1is below.

Example 4. SendingSIGUSR1

Terminal 1	Terminal 2
$ ./sig1 Program with PID=14425 started working Program: iteration 1 Program: iteration 2 Start of signal processing 10 Signal processing 10: 1/5 Signal processing 10: 2/5 Signal processing 10: 3/5	$ kill -SIGUSR1 14425

Signal processing 10: 4/5 Signal processing 10: 5/5 End of signal processing 10 Program: iteration 4 Program: iteration 5 ^CReceived SIGINT signal, aborting $	

Upon receiving the signal, it jumps into the processing function, and upon completion of processing, it returns to where it left off (in the loop inmain).When an interrupt is accepted, the context of the tree is automatically stored in that volume the moment before it starts processing the signal. This context is later restored and the company continues to work normally.

In signal processingSIGUSR1new signals are temporarily not acceptedSIGUSR1 –It is normal behavior for each signal, it is temporarily blocked while the previous one of the same type is processed. However, when masking that signal, it is defined in the program that iSIGTERMnot interrupts signal processingSIGUSR1.These signals are processed afterwards. This behavior is demonstrated in the following example.

Example 5. Sending multiple signals

Terminal 1	Terminal 2

$ ./sig1
Program with PID=14492 started working
Program: iteration 1
Program: iteration 2
Start of signal processing 10
Signal processing 10: 1/5 Signal
processing 10: 2/5 Signal
processing 10: 3/5 Signal
processing 10: 4/5 Signal
processing 10: 5/5 End of signal
processing 10 Start of signal
processing 10 Signal processing
10 : 1/5 Signal processing 10: 2/5
Signal processing 10: 3/5 Signal
processing 10: 4/5 Signal
processing 10: 5/5 End of signal
processing 10
 	
Received SIGTERM signal, clean up before exiting the program Program with PID=14492 finished working
$
	$ kill -SIGUSR1 14492
	
	$ kill -SIGUSR1 14492
	
	$ kill -SIGTERM 14492

Second signalSIGUSR1was accepted only after the first one was processed, i.e. it was "on hold" until then. It's the same with the signalSIGTERMwhich is accepted only after the completion of the second processing.

The behavior of the process on the signal can be:

1. processing in the usual way (presumed), when the program does not define otherwise,
2. processing with a given function (e.g. withsigaction),
3. Temporarily do not accept the signal - block the signal
4. ignore the signal.

Modes 1, 2 and 4 can be set via the interfacesigaction,with a constantSIG_DFLfor 1, function address for 2 andSIG_IGNto ignore the signal. The behavior for 3 is automatically set at

signal reception. The interfacesixsetand a constantSIG_HOLDcan set behavior 3 as well as by calling a functionsigh.

Signals are sent to the process by the operating system for its own reasons or at the request of a process. In the above examples, signals were sent by command (program)killor directly throughCtrl+C which the shell interpreted as a request to send a signal and sent it to the process it started.

Many mechanisms of the operating system (UNIX) are based on the use of signals. Periodic operations can be done by asking the OS to periodically send a signal with which the function is then connected (e.g.setitimer).A simple operationsleep(x)it is realized through signals: first, the device asks the OS to send a signal afterxseconds (alarm(x)),and then pauses its performance (pause()).Therefore, it may happen that such a delay withsleep(x) it doesn't lastxseconds already less. In simulation it is desirablesleep(x)replace with loop sxthe iteration in which it is used sleeping(1)to reduce delay error.

Task

Simulate the operation of an interrupting system with an interrupt acceptance circuit with three interrupt levels (three priorities). Simulate interruptions with arbitrarily chosen signals. For example signalSIGINTsimulates the highest level interrupt (3), signalSIGUSR1simulates an intermediate level interruption (2), and the signalSIGTERM simulates the lowest level interrupt (1). Interrupts can be generated either via the keyboard (eg Ctrl+C) or by some other program (your own or commandkill)which runs in a separate terminal. Except simulations of circuit registers for receiving interrupts, with an additional data structure to simulate what is on the stack.

When starting, the program should print the initial state of the system when no interruption has yet occurred. When an interrupt occurs, the interrupt level is printed. Interrupt processing is simulated so that the program prints out that interrupt processing has started, the state of the system during interrupt processing (control flags, current priority and state on the stack) and after sleeping for a few seconds, prints out that the processing has ended.

Put the current time at the beginning of each printout. An example of how to get the time and sleep for a certain time can be viewed (and used) from the example lab1-example_sleep.c.

An example of a printout (the printout doesn't have to look like this, but it must contain this information!):

```
$ ./lab1
000,000:        Program with PID=14497 started working
000,000:        K_Z=000, T_P=0, stack: -

002.041:        ASSEMBLY: A level 2 interrupt has occurred and is being passed to processor
002.041:        K_Z=010, T_P=0, stack: -

002.041:        Level 2 processing started K_Z=000, T_P=2,
002.041:        stack: 0, reg[0]

003.716:        ASSEMBLY: A level 1 interrupt has occurred, but it is remembered and not passed to the processor
003.716:        K_Z=100, T_P=2, stack: 0, reg[0]

 Ĉ005.416:      ASSEMBLY: A level 3 interrupt has occurred and is being passed to processor
005.416:        K_Z=101, T_P=2, stack: 0, reg[0]

005.416:        Started interrupt level 3 processing K_Z=100, T_P=3,
005.416:        stack: 2, reg[2]; 0, reg[0]

010.416:        Interrupt processing level 3 finished Processing
010.416:        interrupt level 2 continues K_Z=100, T_P=2, stack:
010.416:        0, reg[0]

012.041:        Finished interrupt level 2 processing
```

012.041: The execution of the main program continues
012.041: K_Z=100, T_P=0, stack: -

012.041: ASSEMBLY: T_P changed, forwards level 1 interrupt to processor Level 1 processing
012.041: started
012.041: K_Z=000, T_P=1, stack: 0, reg[0]

017.041: Finished interrupt level 1 processing
017.041: K_Z=000, T_P=0, stack: -
. . .