

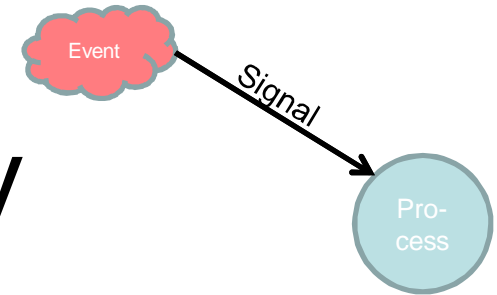


Real-Time Programming TI00AA55

Lecture 7 - 04.03.2015

Jarkko.Vuori@metropolia.fi

Signal terminology



- **Event** can happen when a program is running
- **Signal** is a method that is used to inform a process about the event
 - A signal is a short message sent to a process, or group of processes, containing the number identifying the signal
 - No data is delivered with traditional signals
 - POSIX defines interface for queueing & ordering RT signals with arguments
- It is the kernel that first knows about an event
- Kernel sends (**delivers**) a signal to the process
- The process can catch the signal
- The signal is not necessarily delivered immediately after the event (**pending** signal)
- The event (or the delivery of signal) is **asynchronous** to the program
- It is not possible to know in advance where the process is executing the code, when a signal is delivered
- The delivery of a signal interrupts the execution of the code
- The program starts to execute **signal handling** function, if it is installed
- The programmer can write his own signal handling functions for a process
- Signal handling function is never called explicitly
- Kernel calls it when it delivers the signal
- Installing a signal handler function means that we tell to the kernel, what function it should call when a signal is delivered
 - This can be done in most simple way with a function `signal`

Events that can generate signals

- The following categories of events can generate signal:
 - A certain key presses (Control-C) from the keyboard (SIGINT)
 - Command line command kill (whatever signal)
 - An error in program execution (hardware related), Division by zero or incorrect memory reference (SIGFPE or SIGSEGV)
 - A certain software condition (SIGALRM)
 - Process can send a signal to another process (kill)
 - A process can send a signal to itself (raise)
- Signals can be used to make a process to inform another process that a certain kind of event has happened (so called user defined signals)
- POSIX standard defines signals (so called reliable signals)
- Each signal has its own signal number
 - To make it easy to use them, symbolic names are defined for each signal number according the event they signal

- Examples of signals (symbolic names are used):

SIGALRM	A given delay has elapsed
SIGABRT	Program has asked the kernel to abort itself with this signal
SIGFPE	Floating point arithmetic error (for example division by zero)
SIGCHLD	Child process has terminated
SIGINT	Interrupt a program (Ctrl-C)
SIGUSR1	User signal 1
SIGUSR2	User signal 2

Signal handling methods

- There are three options:

1. Use the default handling for the signal

- The default handling of most signals is to "terminate process"

2. Ignore the signal

- Nothing is done, the process continues
- It is not possible to ignore signal SIGKILL

3. Catch the signal and handle it

- This means that a programmer writes a signal handling function, and "installs" it

- Install (set) signal handler

- The simplest way to install signal handler is to use function signal:

`<signal.h>`

```
typedef void (*sighdlr_t)(int);  
sighdlr_t signal(int signum,  
sighdlr_t handler);
```

This function takes two parameters: 1) signo that indicates the signal we want to catch and 2) func that is the function we want to execute when signal is delivered. Function returns the current handler or SIG_ERR in error situation.

Definition: `#define SIG_ERR (void (*)(int)) -1`

- It is possible to set the following handler "functions":

SIG_IGN	Ignore signal
SIG_DFL	Use default handler

Installing signal handler

- The following example illustrates the phases when a process installs it's own signal handler for a certain part of the program and restores the original handler for the rest of the program

```
static void signal_handler(int); // forward declaration
int main(void) {
    void (*old_handler) (int);

    // Set your own handler and save the old handler
    old_handler = signal(SIGINT, signal_handler);
    if (old_handler == SIG_ERR)
        fprintf(stderr, "\nCannot set signal handler");
    // Now INT signals are handled by our own handler
    ...
    // Reset the original handler
    signal(SIGINT, old_handler);
    // Now INT signals are handled by original (default)
    // handler
    // It is possible to explicitly set default handler:
    // signal(SIGINT, SIG_DFL);
    ...
    return 0;
}

static void signal_handler(int sig_no) {
    ...
}
```

Example

```
/* Ctrl-C (interrupt) signal handler */
#include <stdio.h>
#include <unistd.h>
#include <signal.h>

static void ctrl_handler(int);
int stop = 0;


int main(void) {
    if (signal(SIGINT, ctrl_handler) == SIG_ERR)
        fprintf(stderr,
            "Cannot set signal handler\n");

    while (!stop) {
        printf("Program continues\n");
    }

    return 0;
}
```

```
static void ctrl_handler(int sig_no) {
    char answer;
    if(sig_no == SIGINT) {
        printf("Do you really want to stop?\n");
        printf("Press Y and Enter stop,
            Enter to cont. \n");
        answer = getchar();
        if (answer == 'Y') stop = 1;
    }
    return;
}
```

Consider the
difference, if we had
used exit (0);



Remark. The same handler can handle several signals

ALARM signal

- Signal SIGALRM is used for an example to
 - start some task after a certain time period
 - implement sleep
 - unblock blocked function calls (implement timeout)
- The default handler terminates the process
- The request for kernel to generate a signal after a time period can be done with a function alarm
- Only one timer can be set for a process
- Function alarm

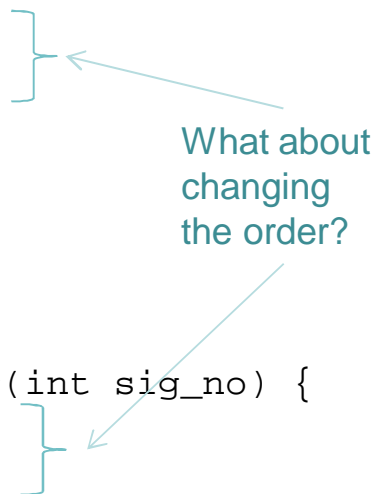
```
<signal.h>  <sys/types.h>
int alarm(int seconds);
```

 - Set the time in seconds after which the signal SIGALRM is generated
 - Function returns the time left from the previous alarm clock
 - The possible previous alarm request is cancelled
- Remark. It is possible to cancel the alarm clock (not yet expired) without setting a new alarm by passing time parameter 0
 - The function returns the time left to alarm

Example of periodic timing

```
// Do the thing once in 5 seconds
void alarm_handler(int sig_no);
int main(void) {
    signal(SIGALRM, alarm_handler);
    alarm(5);
    do_something();
    while (1) {
        ...
        pause();
    }
}

void alarm_handler(int sig_no) {
    alarm(5);
    do_something();
}
```



What about changing the order?

- Remark. Consider the option where function `do_something` would be called in the main function inside the while loop
 - What would be the difference in the functionality of the program?
- Remark. Later on we will see examples of using ALARM-signal for the implementation of the sleep function and timeout
 - The purpose of these examples is to demonstrate the so called race condition

Sending a signal

- A process can send a signal to another process (or to itself) with the function kill

```
<signal.h>    <sys/types.h>
int kill(pid_t pid, int signo);
```

- The parameter pid indicates what process will receive the signal
- Although the name of the function is kill, it can be used to send whatever signal
 - Actually this function asks kernel to deliver a signal to the process pid
- Example. A child process sends signal SIGUSR1 to it's parent:
`kill(getppid(), SIGUSR1);`
 - Special cases:
 - If pid = 0, the signal is sent to processes that are in the same process group than the sending process
 - If pid < 0, signal is send to all processes in the process group |pid|

- A process can send a signals to another process only if they have same real user id or effective user id
- Process can test whether another process is "alive" sending a signal 0 to another process (signal 0 has no effect on the receiving process):

```
if (kill(pid, 0) == -1 &&
    errno == ESRCH))
    printf("The process %d is not
           alive\n", pid);
```
- Remark. A process can send a signal to itself using the function raise
`int raise(int signo);`

Signal handling and parent/child relationship

- After fork:
 - Child inherits the signal handlers of parent if they are installed before fork
 - Child inherits the process signal mask of the parent
 - Pending alarms are not inherited
 - Pending signals are not inherited (set of pending signals of child is empty after the fork)
- After exec:
 - The child must set its own signal handler functions
 - The signal mask of the process remains unchanged
 - Pending alarm requests are “inherited” by the new program
 - Also pending signals are “inherited” by the new program

Reliable signals

- The old fashion signal handling in UNIX-systems was often unreliable
 - The POSIX-standard requires so called “reliable signals”
- Originally the function `signal` did not offer reliable signal handling. In addition the implementation were different
 - POSIX defined a new function `sigaction` for signal handling
 - Nowadays the function `signal` is often implemented using `sigaction`
 - It is better to use `sigaction` however
- POSIX reliable signals mean the following things:
 1. The signal handling function remains installed when it is once installed. The first signal does not restore the default handler
 2. If the same event occurs again, when the signal is currently under processing, the new signal is not delivered so that signal handling is not interrupted. Instead, a new signal is delivered, when the previous has been processed (signal handler has returned) (it is not guaranteed however, that if event happens many times, when signal handler is running, that signals are queued)
 3. The thing explained above means, that signal is blocked while it is handled
 - However, notice that signal S1 can interrupt the signal handling of signal S2

Special things to observe with signals

- There are four special things you have to observe when writing programs with signal handlers:
 1. Signal delivery (return from the signal handler) can interrupt system call. This means that blocked system call can return, without completing the task requested
 2. Global variable `errno` can cause problems, because signal delivery is asynchronous and function calls in signal handler can change `errno`
 3. Non-reentrant functions can cause problems if they are called from main code and from signal handler. The main reason again is that signal delivery is asynchronous to main code
 4. Signals can cause a so called race condition
- These four things are handled in more detailed way in the following pages

Interrupted system calls 1

- So called slow system calls can be interrupted by signals
- These kind of slow system calls are for example:
 - Input from terminal (when no input is entered)
 - Input from a pipe (when data is not available)
 - Waiting for a child
 - Sleeping (and so on)
- The program can check if the system call was interrupted by a signal, because the return value in this case is -1 (error) and errno is set to EINTR
- The example on the right shows how we can always explicitly restart system call that is interrupted by signal. The loop terminates when another error happens or when the system call succeeds:

- Example 1 (How to explicitly restart a read)

```
char chr;  
while(read(STDIN_FILENO, &chr, 1) == -1 &&  
      errno == EINTR);
```

See more complete separate example program (interrupted_read_1.c, interrupted_read_2.c)

- Example 2 (How to wait for all children)

```
while((r = wait(NULL)) > 0 ||  
      (r = -1 && errno == EINTR));
```

- Example 3. (How to guarantee that we sleep a certain time)

```
time_left = n;  
while((time_left = sleep(time_left)) > 0);
```

Interrupted system calls 2

- POSIX does not specify should system calls be interrupted by signals or should they be restarted (continue to execute them)
- In many systems, if you use function `signal` to set signal handler, system calls are restarted for all other signals than SIGALRM. This means that in this case only SIGALRM interrupts system calls
- In Linux, if you set your own signal handler for signal S using function `signal`, signal S does not interrupt system calls (the system calls are restarted). This is true for all signals. The behaviour depends actually also on certain flags
- If you use function `sigaction` when setting the signal handler, you can explicitly specify yourself, whether you want this particular signal to interrupt system calls or not. This is done using SA_RESTART flag in a structure member `sa_flags`. See function `sigaction` a bit later
- System call `pause` is always interrupted by any signal

Problem with errno-variable

- A process has one global variable `errno`, where kernel stores error code of the latest system call
- It is possible that system call is just done, it has set the `errno`, but `errno` is not tested yet and signal is delivered
 - This means that program starts to execute signal handler
 - It is possible that signal handler makes a system call (same or different), that sets error code in `errno`
 - When signal handler returns, the next task in the main code is to test `errno`, but now we don't test the `errno` from the system call of main code, but the `errno` of system call from signal handler, what is not what we want!
- The solution is to store `errno` in the beginning of signal handler and restore it at the end of the signal handler. The following example illustrates the problem and the solution (imagine that a signal is delivered immediately after `read`):

```
result = read(fd, &chr, 1);
if (result < 0 && errno == SOMETHING)
    doSomething();
```

Signal handler:

```
void sighandler(int sig) {
    int saved_errno = errno;
    //call something that can change errno
    errno = saved_errno;
}
```

Solution

Non reentrant functions

- As we already know, signals are asynchronous to the main code
- This means that it is possible, that the main code has called a function that is not fully completed, when kernel delivers the signal and the execution of signal handler is started (this uncompleted function can be for example `malloc` or `getpwnam`)
- What happens, if the same function is called from the signal handler?
- This can cause problems, if the called function is not designed to tolerate that situation so that it can provide service for two callers simultaneously
 - Function that can manage this situation is called reentrant function (otherwise it is called non reentrant)
 - The problem in the `malloc` function for example, is the maintenance of links between memory blocks
 - The problem of function `getpwnam` on the other hand is the static array, where the function stores the result value
- POSIX requires that certain functions must be reentrant
- It is interesting to notice that this list does not contain functions `malloc` or standard i/o-library functions

How to fix the problem of non re-entrant function

- The problem with non reentrant function can be avoided in two ways:
 1. No problem can arise, if we do not call the same non reentrant function from the ordinary code and from the signal handler
 2. Sometimes it is not possible to follow the advice given above. Then we need to block the signal (whose signal handler calls the same function) just before we call the non reentrant function `f` in ordinary function. The signal is unblocked immediately, after the function returns. In other words we prevent the signal from being delivered to the process as long as the function `f` is executed for the main code.

- Example.

Main code of the process

```
//block signal S
f();
//unblock signal S
```

This is only principle. The final solution on the page 25

Signal handler

```
void sighandler_for_S(int sig) {
    f();
}
```

- This technique of blocking signals is more general and it is used in many situations where signal can cause (often timing) problems

Race condition 1 (process to sleep)

- This is needed for the demonstration of a **race** condition
- The simplest way to wait for a signal is calling the function `pause`. Function `pause` suspends the execution of the calling process (puts it to sleep) until a signal (whatever) is delivered.

`<unistd.h>`

```
int pause(void);
```

- Function returns only when signal is caught and the signal handler returns (return but not exit)
- In this case the `pause` returns -1 and `errno` is `EINTR`

- Remark. This means that we have our own signal handler installed
- Remark. It is also possible to wait for a specific signal

`<signal.h>`

```
int sigwait(const sigset_t *sigmask,  
            int *signo);
```

- Let's discuss about this on the page 30
- Signal `SIGALRM` and `pause` can be used together to put a process to sleep for a certain period of time (`sleep`)
 - A simple incomplete solution and the problems it involves are presented on the next page. The purpose of this example is to illustrate a so called **race condition**

Race condition 2 (implementation of sleep)

```
static void alarm_handler(int signo) {  
    return;  
}  
unsigned int sleep(unsigned int sec) {  
    signal(SIGALRM, alarm_handler);  
    alarm(sec);  
    pause();  
    return(alarm(0));  
}
```

There is a race which happens first.

1. Delivery of the signal
2. Reaching the pause

- The **problems in this solution** are:
 1. The **race condition** can cause the blocking of the program. This situation arises if the signal SIGALRM is delivered before the program has reached the waiting state (executed the pause)
 2. Any other signal can cause the pause to return too early. The caller of the sleep function can test this situation by checking the return value of the sleep function (alarm returns the remaining time, if any)
 3. If the programmer has used the signal SIGALRM for some other purpose, calling sleep above spoils this another use of the signal. This happens because sleep function does not restore the signal handler or the time left from the timer set before calling the sleep
- See the principle on the next page, how the race condition problem can be solved

Fixing principle of race condition problem

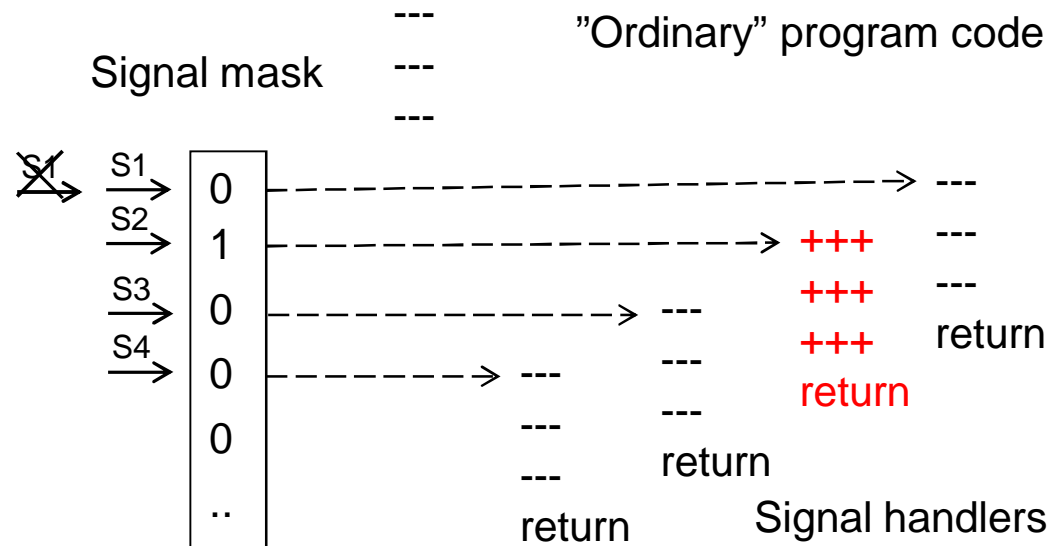
- We again need to prevent the signal delivery (block the signal)
- In addition to that we need an operation that does two things atomically: it goes to the wait state and unblocks the signal delivery
- The sleep function does this in the following way:
 - Block the signal SIGALRM
 - Call the function alarm
 - Go to the wait state and atomically unblock the signal
 - The program returns from the wait state when signal is delivered
- Same thing as a program (containing pseudo code)

```
unsigned int sleep (unsigned int sec) {  
    //block signal SIGALRM  
    signal(SIGALRM, alarm_handler);  
    alarm(sec);  
    go_to_pause_and_open_sigalrm();  
    return(alarm(0));  
}
```
- In the reality the function sigsuspend corresponds the instruction go_to_pause_and_open_sigalrm. It is handled on the page 28
- This is only a principle. The final version (POSIX implementation) that also fixes the problem 3 on the page 18 is represented in the book (Stevens & Rago) in the program

Signal mask of the process

- We have seen that signal blocking is needed in solving the problem caused by non reentrant functions as well as in solving the race condition problem
- Now we will see how and why signal blocking is possible in practice
- The blocking of signal delivery is possible, because each process has a so called signal mask, that indicates what signals can be delivered and what signals are blocked
- If signal delivery is blocked it means that if event occurs the corresponding signal remains pending
 - This signal is delivered later when we unblock the signal
 - Unblocking signal means that we change the signal mask of the process so that it allows a signal to be delivered again
- Blocking and unblocking the signal both means that the signal mask is changed accordingly
 - This is done with a function `sigprocmask`
- The figure on the next page illustrates the functioning of the signal mask of the process

Illustration of the signal mask



- Remark 1. +++ means that the signal handler function is under execution
- You can think that there is one bit per signal number in the signal mask
 - If the bit is set, the signal delivery is prohibited
 - If the bit is off, the signal delivery is allowed
- Remark 2. Remark. If a signal is blocked and an event occurs many times, it is not guaranteed by POSIX, that signals are queued (that all signals are delivered). Only one signal can be waiting behind the mask.
- Remark 3. When a signal is delivered (the signal handler is started to execute), this signal is blocked in the mask automatically. It is opened when signal handler returns.
- Signal masks are represented in the program with the data type `sigset_t` (signal set)

Signal sets (sigset_t)

- We saw that signal sets are needed and used when
 1. Signal mask of the process is manipulated (when signals are blocked and unblocked)
 2. Signal handler is installed with the function `sigaction`. In addition, signal set is also needed when using functions
 3. `Sigsuspend`
 4. `sigpending` and
 5. `sigwait`
- Data type `sigset_t` is used to represent signal sets
- The following functions can be used to manipulate the signal set:

```
int sigemptyset(sigset_t *set);
int sigfillset(sigset_t *set);
int sigaddset(sigset_t *set, int signo);
int sigdelset(sigset_t *set, int signo);
```

 - Returns 0 (OK) or -1 (ERROR)
- `int sigismember(const sigset_t *set, int signo);`
 - Returns 1, if `signo` is in signals set, otherwise 0.
- Data type `sigset_t` can be (but is not necessarily) long int, where bit positions correspond signals (bit position 0 is signal 1, etc.)
- Function `sigaddset` could be like

```
int sigaddset(sigset_t *set, int signo) {
    if (signo <= 0 || signo >= NSIG) {
        errno = EINVAL;
        return -1;
    }
    *set = *set | (1 << (signo-1));
    return 0;
}
```

Set and get the signal mask

- Function `sigprocmask`
`<signal.h>`
`int sigprocmask(int how, const sigset_t *set, sigset_t *oldset);`
 - Returns 0, if OK, -1, if error
- Parameter `how` indicates, how the signal set is used when modifying current signal mask of the process
- The options are:
 - `SIG_BLOCK` The signals in the signal set are or'ed (added) to the current signal mask of the process
 - `SIG_UNBLOCK` The signals in the signal set are cleared (removed) from the current signals mask of the process
 - `SIG_SETMASK` The signal set is used as such as a new signals mask. The old signal mask has no effect
- The current (old) signals mask can be stored passing an address of `sigset_t` variable as a third parameter to the `sigprocmask`
- Function `sigprocmask` can be used to ask the current signal mask without modifying it by passing `NULL` as the second parameter:
`sigset_t current_mask;`
`sigprocmask(0, NULL, ¤t_mask);`

Example how to block and unblock a signal

Compare to page 17

Main code of the process

```
sigset_t new_mask, old_mask;
...
//Block signal S
sigemptyset(&new_mask);
sigaddset(&new_mask, S);
sigprocmask(SIG_BLOCK, &new_mask, &old_mask);

//Do something, for example call
//non re-entrant function f
f();

//Unblock signal S
sigprocmask(SIG_SETMASK, &old_mask, NULL);
...

//Signal handler
void sighandler_for_S(int sig) {
    f();
}
```

New way to set signal handler

- We have used function `signal` for installing signal handler
 - It is simple to use, but it also has restrictions
- Now we will learn another function `sigaction`. Some of the essential features of `sigaction` are:
 1. When signal handler is called, the signal is automatically added to the signal mask of the process (and it is removed when handler returns). This is also for `signal f`
 2. We can specify, what other signals we want to be added to the signal mask of the process when signal handler is called (and to be removed when signal handler returns)
 3. We can explicitly determine, do we want system calls to be restarted or not (this is not required by POSIX)
 4. This function can also be used to ask the current handler without modifying it
- Function `sigaction` takes two parameters of the type `struct sigaction*`
- Prototype of the function:
`<signal.h>`

```
int sigaction(int signo,  
              const struct sigaction *act,  
              struct sigaction *oldaction  
              );
```

Function `signal` can be implemented with `sigaction`. See the program 10.18 in the book (Stevens & Rago).
- The definition of structure `sigaction`:

```
struct sigaction {  
    void (*sa_handler) (int);  
    sigset_t sa_mask; //during signal handling  
    int sa_flags; //SA_RESTART (not in POSIX)  
    void (*sa_sigaction) (int, siginfo_t *,  
                          void *); //rth  
};
```

Options

Examples

- Next examples illustrates how to use function `sigaction` and how the programmer can decide whether he wants the signal to interrupt the system call or not
- 1. The first example (`interrupted_read_1.c`) shows how we let the signal `SIGINT` interrupt the system call
 - However it is restarted explicitly in the program
- 2. In the second example (`interrupted_read_2.c`) we do not allow the signal `SIGINT` to interrupt the system call
 - The kernel restarts it automatically
- 3. The third example (`no_signal_queue_demo.c`) illustrates how signals can be blocked
 - It can be used to demonstrate that there is no signal queue behind the mask
 - Signals are lost if the event happens many times while signal is blocked
- 4. The fourth example program (`sleep_alarm.c`) is used to examine how the `sleep` function and `SIGALRM` signal can be used together in the program

Function sigsuspend

- On the page 20 we saw the principle of the solution of the race condition problem
 - We used pseudo language in the form `go_to_pause_and_open_sigalrm()`
 - Now we have learned to understand signal masks and are able to study the real counterpart of this pseudo instruction, which is `sigsuspend`
- Function `sigsuspend` solves the race condition because it suspends the process and modifies the signal mask of the process atomically
 - The new signal mask that comes into effect, when the process is already suspended, is passed as a parameter
- Sigsuspend

```
<signal.h>
int sigsuspend(const sigset_t *newmask);
```

 - Function returns only when a signal is caught and signal handler returns
 - In this case the function returns -1 and `errno` is `EINTR`
 - When the function returns, the **signal mask of the process is restored** automatically to what it was before the call of `sigsuspend`
- Example 1. Final solution to the race condition problem in sleep. See the program 10.29 in the book (Stevens & Rago)
- Example 2. Solving the problem when using signals and pause for synchronization (lab exercise)

Function sigpending

- Function `sigpending` can be used to ask what signals are currently pending (signals that are waiting behind the mask)

`<signal.h>`

`int sigpending(sigset_t *set);`

- Returns 0, if OK, -1, is error

```
sigset_t pending;
sigpending(&pending);
if (sigismember(&pending, SIGINT))
    printf("Signal INT is pending\n");
else
    printf("Signal INT is not pending\n");
```

Function sigwait

- Sigwait is used to wait for specific signals
<signal.h>

`int sigwait(const sigset_t *sigmask, int *signo);`

- Returns 0, if OK, -1, if error
- Sigwait makes signal handling simpler in certain situations, especially in multithreaded applications

- General usage principles:

1. Signals are initially blocked
2. The function blocks and waits for the signals that are specified in the parameter sigmask. It also opens the masks for these signals
3. When one of these signals is delivered, sigwait returns
4. The programmer can find out what was the signal that caused sigwait to return. The signal number is stored in variable whose pointer was as a parameter signo
5. This signal is automatically removed from the set of pending signals
6. The signal **handler function is not called or even needed** in this case

```
int main(void) {  
    sigset_t mask; int signo;  
  
    sigemptyset(&mask);  
    sigaddset(&mask, SIGINT);  
    sigprocmask(SIG_BLOCK, &mask, NULL);  
    sleep(5); // Press CTRL_C  
    sigwait(&mask, &signo)  
    if (signo == SIGINT)  
        printf("Signal SIGINT is delivered\n");  
    ...  
}
```

If the signal was delivered before the program has reached the sigwait, sigwait never returns

Process group and SIGINT

- • Parent process and child process belong to the same process group
- Parent process is the leader of process group (process group leader)
- Each process group has process group id (type is pid_t)
- The process can ask it's process group id with the function `pid_t getpgrp(void);`
- The process group leader has process id that is same than process group id
- • If Ctrl-C is pressed, signal SIGINT is sent to all members of the foreground process group

Global variables

- Recall the disadvantages of global variables!
- Global variables can be used to pass information from signal handler to main program (and vice versa) or between threads
- The disadvantages of global variables can be removed by encapsulating them in their own file
- The example starting on the right illustrates this (making the program to run a loop for a certain time period)

```
//Header file for delay settings
void set_delay(int seconds);
int delay_elapsed(void);
int timeLeft(void);

//Application
#include <stdio.h>
#include "delay.h"
int main(void) {
    set_delay(5); //execute the loop 5 seconds
    while (!delay_elapsed()) {
        printf("This program is working\n");
        printf("Time left %d\n", timeLeft());
    }

    return 0;
}
```


Global variables

```
//Implementation file for delay settings
#include <signal.h>
#include "delay.h"
static int seconds_left;
static int elapsed;
void (*old_handler)(int);
void alarm_handler(int signo);
void set_delay(int seconds) {
    elapsed = 0;
    seconds_left = seconds;
    old_handler = signal(SIGALRM, alarm_handler);
    alarm(1);
    printf("Delay in module is %d \n", seconds_left);
    printf("Elapsed time in module is %d \n", elapsed);
    getchar();
}
```

```
int timeLeft(void) {
    return seconds_left;
}
int delay_elapsed(void) {
    return elapsed;
}
void alarm_handler(int sig_no) {
    alarm(1);
    seconds_left--;
    if (seconds_left <= 0) {
        elapsed = 1;
        alarm(0);
        signal(SIGALRM, old_handler);
    }
}
```

The principle of implementing timeout 1

- One method to implement timeout for a slow system call is to use signal SIGALRM and the feature we have learned earlier: interrupted system calls
- The implementation principle is now different from the implementation of the sleep function we saw earlier
 - This is because there is no versions for slow system calls that could go to blocked state and atomically open the signal mask (pause had that kind of “version” called sigsuspend)
- Let’s use as an example the program that reads one character from the keyboard
 - If the user does not enter anything, we want the read function to stop waiting after 5 seconds

- This is the first uncomplete version of the program:

```
static void alarm_handler(int signo) {
    return;
}

int main (void) {
    int r; char chr;
    // Reading one character with timeout
    signal(SIGALRM, alarm_handler);
    alarm(5);
    r = read(0, &chr, 1);
    // Test r and errno
    alarm(0);
    return 0;
}
```

The principle of implementing timeout 2

- This example only demonstrates the principle. You could write a function `read_timeout`, that incorporates all things needed to handle time out. This function could simply return -1 and set `errno` to `ETIMEDOUT` in the case of time out.
- Problems of this solution:
 1. Does not work in the **race condition** (cmp. sleep)
 2. Other signals can make the waiting time shorter
 3. Does not work in the system where signal `SIGALRM` does not interrupt a system call
 4. If signal `SIGALRM` is used in the program for the other purpose, using timeout “spoils” this other usage, because the program does not restore signal handler nor does it reset the alarm again after the time that possibly was left until previous alarm
- The race condition problem can be solved using functions `sigsetjmp` and `siglongjmp`
- The program that solves the race condition problem is `read_timeout.c`
- Remark. Another method to implement timeout is to use `select` (studied later)
 - It only can be used for i/o-operations to file descriptors