Signals and processes

* Signal fundamentals
* Signal blocking, ignoring, handling
* Creating a new process
* Terminating a process

Signals

Signals are mechanisms for communicating with, and manipulating processes.

A signal is a notification sent to a process that an event has occurred.

Think of them as software interrupts. They are similar to hardware interrupts in the sense that they interrupt the normal flow of execution.

Software generates this interrupt, you stop execution, do the necessary functions that signal requires.

It is impossible to predict exactly when a signal is going to arrive.

Examples: dividing by zero, referencing an inaccessible memory location, pressing CTRL-C...all lead to signal generation.

Signals are asynchronous; when a process receives a signal, it processes the signal immediately, without finishing the current function or even the current line of code.

When a signal is delivered to a process, the process can:

* Ignore it (some signals cannot be ignored)
* Take the default action associated with that signal: termination, suspension of execution, core dump file generation, etc.
* Execute a signal handler; i.e. a custom function written by you, that takes appropriate action; e.g. in case of CTRL-C, make sure all data/settings are saved properly.
  + After signal handling happens, you have to be careful not to use some of the basic calls of C library.

ignore it – block it – or signal handler

If a signal handler is used, the currently executing program is paused, the signal handler is executed, and, when the signal handler returns, the program resumes from where it left off.

Every signal is associated with a unique integer number (starting from 1) and is referenced by a symbolic name (defined in signal.h) that starts with SIG; SIGSEGV, SIGTERM, SIGINT, etc.

Make sure you include signal.h in your source files when working with signals.

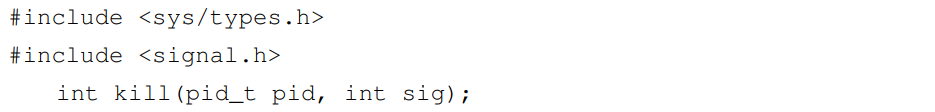
SIGUSR1 and SIGUSR2 are reserved for user use.

Table

Description automatically generated

You can do your own handler to not to do default action.

Signals can be sent by the kernel, by other processes, or even by the user using the kill system call (also available as a shell command):



Returns 0 on success, and -1 on error and sets errno.

You are asking kernel of the OS to do operation. When OS decides it is OK to send signal to that process, it will send the process.

You can use kill in terminal with just pid. It send terminate request to that process.

echo $? 🡪 will return the exit status of last command.

The name kill derives from the fact that historically, many signals have the default action of terminating the process.

pid > 0 -----> signal sent to the process of that pid

pid == -1 -----> sent to all processes for which it has permission to send

If no process matches the specified pid, kill() fails and sets errno to ESRCH (“No such process”).

A process needs appropriate permissions to be able to send a signal to another process; you cannot go around killing other users’ processes.

● The init process (pid 1) is special; it can only be sent signals, for which it has a handler installed; this prevents accidental kills.

● For all others, a process can send another process a signal if their user ids match (there are some intricacies involved).

● SIGCONT is an exception; (can be sent to any process in the same session). Kernel will decide if process this signal is sent will continue or not.

Tip: you can use the kill system call to send the null signal (0) to test for the existence of a specific pid.

It will either send no signal (successfully since the null signal does not exist) or return with ESRCH (no such process).

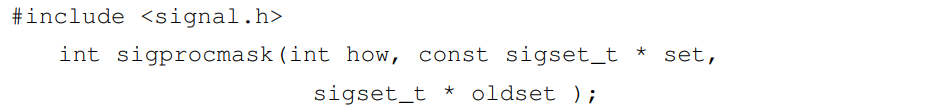
A process can also send a signal to itself, either through

or through the raise system call



Signal mask

For each process (in fact for each thread – kernel sends the signal to a random thread if none block it), the kernel maintains a signal mask - a set of signals whose delivery to the process is currently blocked. **If a signal that is blocked is sent to a process, delivery of that signal is delayed until it is unblocked by being removed from the process signal mask.**

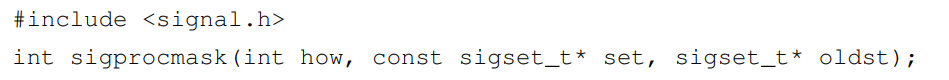


Returns 0 on success, or -1 on error

The sigprocmask() system call can be used at any time to explicitly add signals to, and remove signals from the signal mask.

set is set of signals that you want to manipulate. You want to take action when those signals are received.

how is what you gonna do with signals.



how: how should the mask be changed

● SIG\_BLOCK: the signals in set are added into the signal mask

● SIG\_UNBLOCK: the signals in set are removed from the signal mask

● SIG\_SETMASK: set becomes the signal mask

Some signals, such as SIGSTOP and SIGKILL, cannot be blocked. If an attempt is made to block these signals, the system ignores the request without reporting an error.

Signal set

And what about the sigset\_t type? Easy to manipulate:

Text

Description automatically generated

Signal (un)blocking example

Text

Description automatically generated

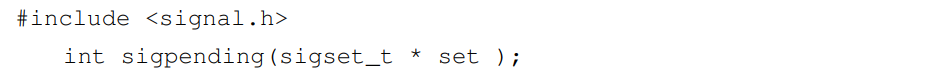
Interrupt sinyali gelince blokluyorsun. Birkaç işlem yapıyorsun. Sonra unblockluyorsun.

If you wanna do some operation and you don’t want it to be stopeed by SIGINT (interrupt) signal, you can use this.

Signal can happen anytime. If it happens before you set a block to it somewhere in execution, for portion of the code will not be executed. You are now in the asynchroneous world of the programming.

Pending signals

If a process receives a signal that it is currently blocking, that signal is added to the process’s set of pending signals. When (and if) the signal is later unblocked, it is then delivered to the process. To determine which signals are pending for a process, we can call



Returns 0 on success, or –1 on error

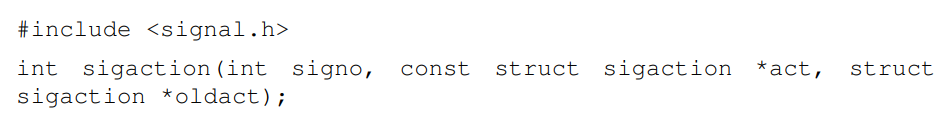
Pending signals are blocked signals. They blocked until they are unblocked.

We can then examine set using sigismember()

The set of pending signals is only a mask; it indicates whether or not a signal has occurred, but not how many times it has occurred. Pending signals DO NOT queue.

Signal handlers

Handling signals either by catching or ignoring them, is done through the sigaction call:



* signo: signal number for action
* act: the action to take
* oldact: receives the previous action towards signo

Graphical user interface, text, application, email

Description automatically generated

sa\_handler can be these other than function:

SIG\_DFL: default action

SIG\_IGN: ignore signal

The handler cannot return anything, and receives only signo as param.

sa\_handler is the function that we are working on.

execution of the function is defined by sa\_mask.

Signal handling example

Text

Description automatically generated

We say 🡪 for the signal generated by the user (SIGUSR1), use sa structure.

Whenever we receive SIGUSR1 signal, sa will execute handler function.

Don’t do sth big in handler.

Signal handling

The following code segment sets the signal handler for SIGINT to mysighand

Graphical user interface, text, application

Description automatically generated

The following code segment causes the process to ignore SIGINT if the default action is in effect for this signal.

Text

Description automatically generated

Invocation of a signal handler may interrupt the main program flow at any time; the kernel calls the handler on the process’s behalf, and when the handler returns, execution of the program resumes at the point where the handler interrupted it.

Diagram

Description automatically generated

From the perspective of the OS, you have 2 executions going on at the same time: main program and signal handler. Two executions going on the same time.

Things you do in signal handler can affect the main program such as changing some global variables.

After execution of signal handler, you have to restart what is going on before the raise of that signal happened. You have to at least restart the execution of last command. C is not compatible with this.

The sa\_flags field is a bit mask specifying various options controlling how the signal is handled. The following bits can be OR’ed (|)

SA\_RESTART: automatically restart system calls interrupted by this signal handler.

SA\_SIGINFO: invoke the signal handler with additional arguments providing further information about the signal.

And more: SA\_RESETHAND, SA\_NOCLDSTOP, etc.

Waiting for signals

The UNIX signaling mechanism also serves as a way of waiting for an event without busy waiting.

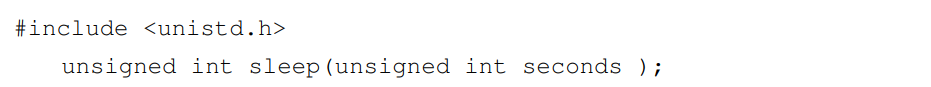
Busy waiting: testing continuously (within a loop) whether a certain event occurred by using CPU cycles.

With busy waiting, I am doing nothing when my turn comes for using CPU. I say to OS don’t make me use the resources of the hardware.

The alternative is to suspend the process until a signal arrives notifying it that the event occurred (hence the CPU is free).

POSIX system calls for suspending processes until a signal occurs: sleep, nanosleep, pause, sigsuspend (suspend the signal before execution of that part of your program then releasing those signals after that execution is finished)...

The sleep() function suspends execution of the calling process for the number of seconds specified in the seconds argument or until a signal is caught (thus interrupting the call).



Returns 0 on normal completion, or number of unslept seconds if prematurely terminated

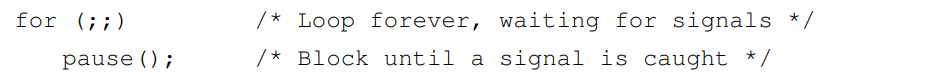
nanosleep is a more powerful version of sleep that operates at higher resolution.

Calling pause() suspends execution of the process until the call is interrupted by a signal handler (or until an unhandled signal terminates the process).

We don’t use any resources of hardware when we say pause.



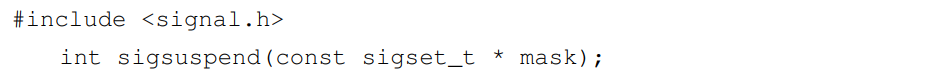
Always returns –1 with errno set to EINTR



Text

Description automatically generated

The solution:



Returns –1 with errno set to EINTR

Atomically unblocks a signal and suspends the process. Equivalent to (atomic):

Text

Description automatically generated with medium confidence

Block, do whatever you want to do, then restore the old mask.

During the execution of the code, that portion of code will not be evoked.

Dealing with signals

There are 3 delicate issues when dealing with signals.

*1) Handling errors that use errno*

* If error happens wile you are handling the signal, error number changes. When you go back to execution of your program from handler and you wanna track whats going on with errno,
  + If error happens on execution of handler, it will be same in execution of your program
  + If you check error that happened before handler, you lose that information.
* STORE ERROR BEFORE HANDLE AND LOAD IT BACK
* If errno is set within the signal handler, that could mean that its previous eventually unprocessed value from the main program is lost!
* Solution: in the signal handler, save and restore errno.
* Graphical user interface, text

  Description automatically generated with medium confidence

*2) Whether POSIX system calls that are interrupted by signals should be restarted.*

* A signal arrives in the middle of a system call. It is handled, and now we return to the call. Should it continue, restart, cancel?
* By default, the system call will fail with the error EINTR (“Interrupted function”). You can use this feature for manual restarts:
* Graphical user interface, text, application

  Description automatically generated
* If this becomes a major concern you can turn it into a macro too.
* Graphical user interface, text

  Description automatically generated
* Unfortunately the SA\_RESTART flag we saw earlier, is not supported by all system calls, for historical reasons.
* This is request to OS, you don’t know it will be fulfilled or not.

*3) The last issue is about which system calls to call in a signal handler.*

* Due to initial design of the C library, most of the C functions in glib were not designed to be asynchronously friendly. If you wanna use those functions inside a handler you will get into problem.
* System calls that rely on global/static variables and data structures are not safe (the majority of stdio calls: printf, scanf, etc).
* By not safe we mean that if you call them in the signal handler, you no longer have the guarantee that they will behave as expected when you return to the main program.
* Unfortunately the list of async-safe functions is relatively short. Almost none of the functions in the standard C library are on it.

Signal handlers

Because signals are asynchronous, the main program may be in a very **fragile** state when a signal handler function executes. Therefore, you should avoid performing any I/O operations or calling most library and system functions from signal handlers.

A signal handler should perform the minimum work necessary to respond to a signal, and then return control to the main program (or terminate the program).

In most cases, this consists simply of recording the fact that a signal occurred. Signals may arrive even while handling them...this is very hard to debug.

Process state

The state of a process in a simple operating system

Diagram

Description automatically generated

But how exactly are processes created?

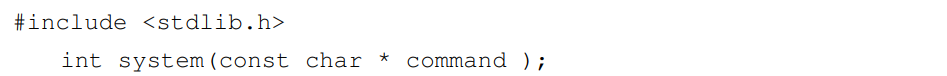
Process creation

Two common techniques are used for creating a new process.

* The first is relatively simple but should be used sparingly because it is inefficient and has considerably security risks. CALLING FUNCTION FROM TERMINAL
* The second technique is more complex but provides greater flexibility, speed, and security. CREATING A COPY OF THE PROCESS THAT YOU ARE USING

The system function in the standard C library provides an easy way to execute a command from within a program, much as if the command had been typed into a shell.

In fact, system creates a subprocess running the standard Bourne shell ( /bin/sh ) and hands the command to that shell for execution.



It returns the exit status of the shell command. If the shell itself cannot be run, it returns 127; if another error occurs, it returns -1.

Kernel generates a terminal for you when you call system. Inside that terminal, it will run whatever the command you give with system.

There are 3 executions 🡪 your program, system terminal, your command

If error happens in your command, it is terminal program that will handle it. You don’t have to worry about that.

Bad thing is you generate terminal process. There are more than 1 kind of terminal in most UNIX system. You cannot know while writing the program which kind of terminal emulator that the system.

Graphical user interface, application

Description automatically generated

Pros: it’s simple, error and signal handling are taken care of.

Cons:

* inefficient since it creates 2 processes, one for the shell, and one more more for the commands
* it’s subject to the features, limitations, and security flaws of the system’s shell. Since you can’t rely on the availability of any particular shell, it’s not portable either.

Create a process inside a process.

The windows API has the spawn family of functions to create new processes by being given just the name of the program to run.

Line chart

Description automatically generated with low confidenceIn the UNIX world the creation of a new process is done in 2 steps!

1. fork: that makes a child process that is an exact copy of its parent process
2. exec: causes a particular process to cease being an instance of one program and to instead become an instance of another program

Main process starts waiting for it when we call fork. Child do what it needs to do and comes back to same point.

Diagram

Description automatically generated

Why create multiple instances of the same process? In many applications, it can be a useful way of dividing up a task.

For web server, for each connection, you can fork out a version of yourself and wait for child to finish whatever they are doing.

For example, a network server process may listen for incoming client requests and create a new child process to handle each request; meanwhile, the server process continues to listen for further client connections.

Dividing tasks up in this way often makes application design simpler. It also permits greater concurrency (i.e., more tasks or requests can be handled simultaneously).

This is accomplished through fork

fork creates exact copy of the process that executes that fork.



In parent: returns process id of child on success, or –1 on error;   
in successfully created child: always returns 0

The key point to understanding fork() is to realize that after it has completed its work, two processes exist, and, in each process, execution continues from the point where fork() returns.

Because no process ever has a process id of zero, this makes it easy for the program to know whether it is now running as the parent or the child process.

Text, letter

Description automatically generated

The child process is created as an exact copy of its parent, except for its process id. This means a copy of everything: data, heap and stack. They have 2 different portions in the memory and kernel space. OS knows they are 2 separate executions.

What about open files?

* The child receives duplicates of all of the parent’s file descriptors. The descriptors in the parent and the child refer to the same open files. The open file description contains the current file offset (as modified by read(), write(), and lseek()) and the open file status flags (set by open()).
  + read, write, lseek operations are done either by child or parent, will affect both of them
  + You open file, read it. You think you read it once but child will also read it. It is hard to keep track of the file offset from one process.
* Consequently, these attributes of an open file are shared between the parent and child. For example, if the child updates the file offset, this change is visible through the corresponding descriptor in the parent.

Map

Description automatically generatedText, letter

Description automatically generated

8 processes have been formed. If you write any code after last line, all of them will execute that. Order is not known.

Process scheduling

After a fork(), it is indeterminate which process - the parent or the child - next has access to the CPU. (On a multiprocessor system, they may both simultaneously get access to a CPU.)

Applications that implicitly or explicitly rely on a particular sequence of execution in order to achieve correct results are open to failure due to race conditions.

Such bugs can be hard to find, as their occurrence depends on scheduling decisions that the kernel makes according to system load.

Use the nice command to control the “priority” of processes.

Process termination

Normally, a process terminates in one of two ways.

Either the executing program calls the exit function, or the program’s main function returns. Each process has an exit code: a number that the process returns to its parent. The exit code is the argument passed to the exit function, or the value returned from main.



By convention

* 0 status: successful completion
* < 0 status: failure
* > 0 status: their meaning is application specific

There are different kinds of exit functions in the system. Some of them will leave its resources to init and init will take care of them after that. All the file descriptors and all the stuff when you exit in some portion will still be opened and init function will come and remove them which we do not recommend.

On some exits, all those requests for the init to get rid of the rest of the resources are done instantaneously.

With most shells, it’s possible to obtain the exit code of the most recently executed program using the special $? variable. Example:

Graphical user interface, text

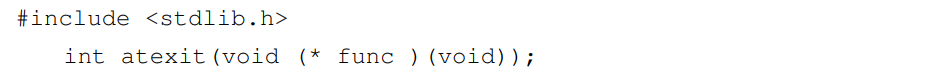
Description automatically generated

shows the latest error code

Calling exit leads to

* execution of exit handlers
* flushing of all stdio buffers

An exit handler is a function to be executed when the process terminates. It is registered through atexit:



More generally, at process termination:

* All open files and streams are closed
* All file locks held by the process are released

The second way of termination is the abnormal way, in response to a signal such as SIGSEGV, SIGBUS, SIGINT, SIGTERM and SIGABRT.

Their default disposition is to terminate the target process.

The most powerful termination signal is SIGKILL which ends a process immediately and cannot be blocked or handled by a program.

*Example*

Graphical user interface, text, application

Description automatically generated

Text

Description automatically generated

Graphical user interface, text, application, email

Description automatically generated

Graphical user interface, application

Description automatically generated

Why are there sleep calls in the child process? What could happen if we remove them?