Tutorial 3 - Unix signals and threads

Gidon Rosalki

2025-04-09

Notice: If you find any mistakes, please open an issue at `https://github.com/robomarvin1501/notes_operating_systems`

1 Reminder

Kernel mode and user modes are 2 different modes for the CPU. When the CPU is in kernel mode, it is assumed to be executing *trusted* software, and can execute any instructions, and access any memory address. The kernel is the core of the OS, and it has complete control over everything that occurs in the system. The kernel is trusted software, and all other programs are considered untrusted software. These programs might call operations that need to occur in kernel mode, such as reading/writing to the disk, at which point it makes a system call, and the kernel does this for it in kernel mode.
A **process** is an executing instance of a program. It is called an active process when it is currently advancing on the CPU, as opposed to waiting in memory for its turn to have some CPU time. The execution of a process can be interrupted by an **interrupt**. This is a notification to the OS that an event has occurred, which results in a change in the sequence of instructions that is executed by the CPU. Interrupts are of the type of hardware/external interrupts, which originate from a hardware device, such as a keyboard, mouse, or system clock. Software / internal interrupts, include exceptions and traps. Exceptions are caused during program execution. These are of two different types, errors (such as division by 0), which require special handling, and not errors, which are things such as access to paged memory, which just requires a few extra instructions to be inserted, and then the program may continue as though nothing happened. Traps occur in the usual run of the program, but unlike exceptions are not the product of an error. The execution of an instruction that is intended for user programs but transfers control to the OS is called a system call.

2 Signals

Signals are notifications sent to / by processes or by the kernel to a process. Having them enables a primitive communication mechanism between processes. Typically they are used to notify the occurrence of events. They are primitive because there is a predefined list of signals that may be sent. Signals cause the process to stop (after the current CPU cycle), and force the process to handle them immediately. The process may configure how it handles a signal (except for some that it cannot configure).
Signals are different from interrupts, since they are generated by the OS and received by a process, as opposed to interrupts, which are produced by the hardware or software, and handled by the OS. Signals in Unix have names and numbers, read `man kill` to find out more.
Signals may be triggered from asynchronous user input, such as `^C` (SIGKILL). The system or another process may also trigger signals, for instance if an alarm set by the process has timed out (SIGALARM). Software interrupts caused by an illegal instruction may also cause signals, for example illegal instructions cause software interrupts, which are then received by the OS, which then generates a signal and sends it to the process.
The most common way of sending signals to processes is using the keyboard:

- Ctrl-C causes the system to send an INT signal (SIGINT) to the running process, generally used to interrupt / stop the process.
- Ctrl-\ causes the system to send a QUIT signal (SIGQUIT) to the running process, which has the default behaviour of telling the application to quit as soon as possible without saving anything.
- Ctrl-Z causes the system to send a TSTP signal (SIGTSTP) to the running process, which generally pauses the currently running process, and it can be resumed later.

As implied earlier, the `kill` command sends the specified signal to the specified process. Use the -l flag to see the list of all the signals you can send. The `fg` command resumes execution of a process that was suspended with Ctrl-Z, by sending it a CONT signal.
There is a C function `int kill(pid_t pid, int sig)` which can be used to send signals from one process to another. We cannot send any data, since all signals are predefined.

2.1 Handling signals

There are several ways of handling signals. The process can terminate, ignore, stop, or continue execution. The process can specify which action to take through a signal handler. There are some signals the process cannot catch, for example KILL and STOP. If you install no signal handlers of your own, the runtime environment sets up a set of default handlers. For example the default handlers for TERM calls `exit()`, and the default handler for ABRT is to dump the process's memory image into a file, and then exit.

```
int sigaction (int sig, struct sigaction *new_act, struct sigaction *old_act);
```

This is a functions that allows the calling process to examine and / or specify the action associated with a specific signal. `action = signal handler + signal mask + flags`. The signal mask is calculated and installed **only for the duration of the signal handler**. By default, the signal itself is also blocked when the signal occurs. Once an action is installed for a specific signal using `sigaction` it remains installed until another action is explicitly installed. If `old_act` is not NULL, then the previous action is saved as `old_act`.

2.1.1 Example of signals

This code will wait for a signal in an infinite loop, however the signal Ctrl-C has been overwritten to simply print to stdout "Don't do that\n", and then continue execution.

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

void
catch_int (int sig_num)
{
    printf ("Don't do that\n");
    fflush (stdout);
}

int
main (int argc, char *argv[])
{
    // Install catch_int as the
    // signal handler for SIGINT.
    struct sigaction sa;
    sa.sa_handler = &catch_int;
    sigaction (SIGINT, &sa, NULL);
    for (;;)
        // wait until receives a signal
        pause ();
}
```

There are 2 pre-defined signal handler functions that we can use instead of writing our own:

1. `SIG_IGN` Causes the process to ignore the specified signal
2. `SIG_DFL`: Causes the system to set the default signal handler for the given signal.

2.2 Masking / blocking

Some processes need to perform cleanup when stopping, deleting old data, saving data (word processes), and so on. If during the cleanup the program exits abruptly, some old files will remain, and the data will be inconsistent / corrupted. In order to avoid this, signals that can cause us to exit (such as SIGINT) should be blocked during cleanup, but ONLY during cleanup. Masking / blocking is intended only for specific parts of the code.
It is also useful for avoiding signal races. Since signals are handled asynchronously, race conditions can occur. A signal may be received and handled in the middle of an operation that should not be interrupted. A second signal may occur before the current signal handler has finished, and the second signal may be of a different type, or the same type, as the first one. Therefore, we need to block signals from being processed when they are harmful. The blocked signal will be processed after the block is removed. Some signals cannot be blocked. We do this through `sigprocmask`.

```
int sigprocmask(int how, const sigset_t *restrict set, sigset_t *restrict oset);
```

Allows us to specify a set of signal to block, and / or get the list of signals that were previously blocked. `how` takes something like SIG_BLOCK, SIG_UNBLOCK, and SIG_SETMASK, which block, unblock, or only block, the following signals. `*set` is the set of signals, and `*oldset` will return the previous mask if it is not `NULL`.

So, we have seen two system calls for handling signals

1. `sigaction` - which specifies the action to take when a signal is received. It also has a lot of features, we will focus on the `sa_handler`. It can also block signals while the handler is running.
2. `sigprocmask` defines which signals to block

You might find `signal` as an alternative to `sigaction`. **DO NOT USE IT**. It resets the signal handler to the default behaviour every time it is called, and is deprecated, so should not be used at all.

2.3 Summary

Signals are notifications that are sent to a process. The OS causes the process to handle a signal immediately the next time it runs. There are default signal handlers for processes, and these handlers can be changed using `sigaction`. To avoid race conditions, one usually needs to block signals for some of the time using `sigprocmask` and / or `sigaction`.

3 Threads

So as discussed there are many processes on a given machine, but beneath this we have a further separation, into threads. There are 2 types, kernel and user. We want threads to solve the following problem. Given a machine with one CPU, how can we efficiently execute a number of tasks? Each task lives in its own world. These tasks are all given time shared time on the CPU. Let us define this all a little more formally:

A **process** is an instance of an application execution. It is defined by its

- Registers: (PC, SP, etc)
- Memory: (data, heap, stack, and text)
- Environments (files, etc.)

The OS stores all this in a data structure called the **Process Control Block (PCB)**. It saves the process data, and also some OS data:

- Priority and relevant data
- The user - for the access rights
- State (is it now running, etc).

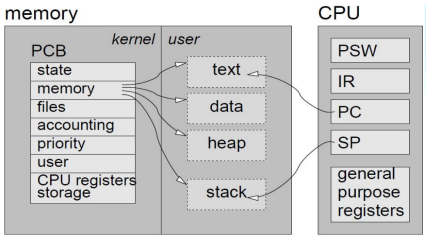
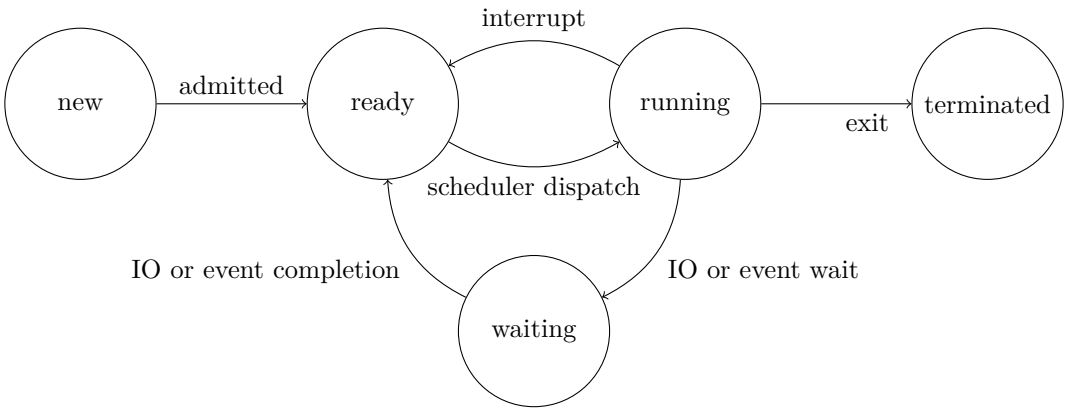


Figure 1: PCB data structure

A process has the following life cycle:



We switch between processes using context switching. When there is an interrupt or system call, the state of this process is stored in the PCB. Once this is done, another state is loaded from its PCB, and runs for a bit until an interrupt or system call. The processes are idle while they are stored in their PCBs, and the other process is being actively run on the CPU.

So, now that we have fully defined what is a process, we should define what is a thread. A **thread** lives within a process, but a process can have *many* threads. A thread possesses an independent flow of control, and can be scheduled to run separately from other threads, because it maintains its own stack and registers. The other resources of the process (such as code, memory, open files, and so on) are shared by all its threads.
There are 2 types of threads. There are the user level threads, where the kernel is unaware of the existence of threads, and the handling is managed by the programmer, and kernel level threads (lightweight processes), where the thread management is handled by the kernel.

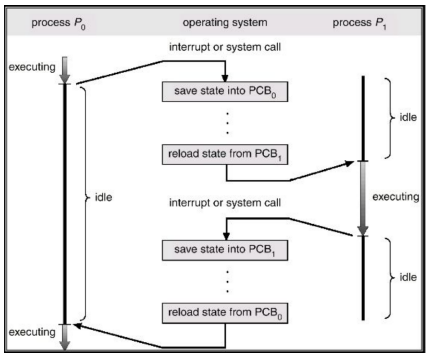


Figure 2: Context switching

3.1 User level threads

These are implemented as a thread library, which contains the code for thread creation, termination scheduling, and switching. The kernel sees one process, and is unaware of the thread activity. Switching between threads is done in user space, so the penalty for a context switch is much lower than an OS context switch, since trap was not called. However, if one thread is blocked by the kernel (from calling `read` for example), then the entire process is blocked. Additionally, if one thread crashes, then the entire process will crash.

To implement a thread library, there are a few things to consider. Only one thread can modify a shared resource at a time (if implemented correctly), so some of the locks required for threads may not be needed. To handle this all correctly, one must maintain a thread descriptor, a subset of the things in a PCB, for each thread. To switch between threads one must

1. Stop running the current thread
2. Save the current state of the thread
3. Jump to another thread, and continue where it had stopped by using its saved state.

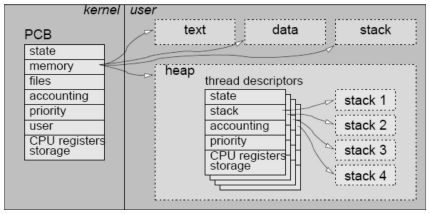


Figure 3: TCB data structure

This requires special functions:

- `sigsetjmp` - saves the current location, CPU state, and signal mask
- `siglongjmp` - goes to the saved location, restoring CPU state and signal mask

```
int sigsetjmp(sigjmp_buf env, int savesigs);
```

This saves the stack context, and CPU state in env for later use. If `savesigs` is non zero, then it also saves the current signal mask in env, so blocked signals will remain blocked when restoring the context. We can jump back to this code location using `siglongjmp`, and it returns 0 if returning directly (ie, first time it is called), and a user defined value when it is returning from `siglongjmp`.

We save in env the program counter (PC), which describes the location in the code, the stack pointer (SP), which is the locations of local variables, and the return addresses of called functions, the signal mask (if specified), and the rest of the environment (CPU state) to ensure calculations can resume from where they stopped. We do not save global variables, dynamically allocated variables, values of local variables, or any other global resource. Since these are shared between the threads of the process, it is not necessary.

```
void siglongjmp(sigjmp_buf env, int val);
```

This jumps to the code location, and restores the CPU state as defined by `env`, ie the code location where `sigsetjmp` was called. If the signal mask was saved in `sigsetjmp`, then it too will be restored. The return value of `sigsetjmp` after arriving from a `siglongjmp` call will be the value defined by the user in `val`.
An example of thread switching is available in exercise 2
When jumping between threads, you need to be aware that you have called `sigsetjmp` before one calls the corresponding `siglongjmp`, since otherwise your code will not know to where to jump.

3.2 Kernel level threads

The kernel has a threads table that keeps track of all the threads. Scheduling is done by the OS's scheduler. If one thread is blocked, then the rest can keep on running. Switching between kernel level threads is typically more expensive than user level threads, since it includes more steps than just saving some registers. These are **not** to be confused with **kernel**

threads or **daemons**. Kernel threads are threads of the kernel running kernel code in kernel mode. They are used to perform some tasks of the kernel in parallel to other processes (and in parallel to the kernel itself).

Advantages	
User level threads	Kernel level threads
Switching between threads is cheaper	Blocking is done on a thread level
Often don't need to worry about concurrent access to data structures	Multiple threads can possibly be executed on different processors
Scheduling can be application specific	The scheduler can make intelligent decisions amongst threads and processes
Disadvantages	
Cannot enjoy the benefits of a multi core machine	Greater cost for switching between threads
One blocked thread blocks the entire process	Need to pay more attention to shared resources
Can suffer from poor OS scheduling	

Table 1: Thread level comparison

So which is the best? Well, there's no such thing, just whichever is most suitable to the job at hand. When choosing one must consider the specifications, and needs of the application. For an application that switches between threads frequently, user level threads may be better. However, for an application with many threads, or I/O bound threads, then kernel level threads may be better.
But why not both? Set a constant set of kernel level threads, and each has its own small set of user level threads. When each thread runs, it quickly switches between the tasks for which it is responsible using the user level threads. A real life example of this is Golang's goroutines.