

Elements of Operating Systems

Chapter 7

Signals

J. Fornés M. Farreras Y. Becerra

Barcelona School of Informatics
Universitat Politècnica de Catalunya

Bachelor's Degree in Informatics Engineering — Operating Systems

Preview for readers · October 8, 2026



UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
Facultat d'Informàtica de Barcelona

© Universitat Politècnica de Catalunya

This work is licensed under the Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International licence (CC BY-NC-SA 4.0):

«By Universitat Politècnica de Catalunya — BarcelonaTech (UPC), Any 2026»



<https://creativecommons.org/licenses/by-nc-sa/4.0/deed.en>

This text was written with the help of Claude, Anthropic's language model. Any errors that remain are the authors'.

7 Signals

Up to now, when a process had to wait for something, it waited for it: it blocked in a `waitpid()` or in a `read()` and the kernel woke it up when the thing arrived. Everything went through a place the process had chosen.

Signals are the other way. There is nowhere to wait for them, because they can arrive at any moment and at any point in the code.

7.1 A software interrupt

You already know what a hardware interrupt does: the processor is executing something, an electrical signal arrives from a device, and the processor drops what it was doing, runs a service routine and returns to where it was. A signal is exactly that, one floor further up.

A **signal** is a **software interrupt** that the kernel sends to a process to tell it that something has happened. The process interrupts whatever it was doing, runs the routine that signal has associated with it and — if it survives — carries on where it was.

The word that matters here is *asynchronous*. A synchronous event happens every time a particular instruction is executed; a signal can fall between any two instructions, and the program has no way of knowing which. That is the source of almost all the difficulty in this chapter, and also of everything you will be able to do with them.

7.2 Where they come from

Signals have been in Unix since the earliest versions, but they were not put there to do what we do with them now. They were for exceptional situations — a user trying to kill another user's process, the hardware failing —, and at no point were they designed as a mechanism for communication between processes [2].

It shows. The first signals were easy to lose and hard to handle. Both BSD and AT&T's System V Release 3 changed the model to get *reliable* signals, but they did it in two incompatible ways, and for a good few years programming signals meant programming them twice. The one that sorted it out was POSIX.1*, and it is POSIX's model that we will explain here.

Careful

You will still find code written with the old schemes, and it still works. The `signal()` call, without the `sig` prefix, is its centrepiece, and you will find it in many books and in a lot of old code. In this book we will never use it: it does different things on different systems and it has a race condition of its own that nobody wants. What we will use is `sigaction()`, which is its replacement and the one POSIX fixes. Mixing the two models, on the other hand, is never a good idea.

7.3 Sending one

A signal can reach you from three places.

From another process, which executes the system call `kill()`. The name is unfortunate: the call kills nothing, it sends the signal you tell it to, and whether that kills the recipient or not depends on which signal it is and on what the recipient has set up for it.

*Also known as IEEE Std 1003.1-1988.

System call `kill`

```
int kill(pid_t pid, int signum);
```

To send one you have to know the recipient's PID, and you have to be entitled to: you can only send signals to processes belonging to your user, by the rule in Chapter 5.

Listing 7.1. Sending a signal is two lines

```
1 int main(int argc, char *argv[])
2 {
3     pid_t target;
4
5     if (argc != 2)
6         Usage(argv[0]);
7
8     target = (pid_t)atoi(argv[1]);
9     if (kill(target, SIGUSR1) < 0)
10         error_and_exit("kill", 1);
11     return 0;
12 }
```

- ❶ The program's only system call. It returns `-1` if the process does not exist or is not yours.

From the terminal, which is the same thing by another route. When you press `Ctrl-C`, the terminal sends a `SIGINT` to every process you have in the foreground there; `Ctrl-Z` sends a `SIGTSTP`, which stops them. And from the shell you have the `kill` command, which does exactly the same as Listing 7.1:

```
$ kill -USR1 1234      # sends a SIGUSR1 to process 1234
$ kill -9 1234        # the number works too: 9 is SIGKILL
$ kill -1             # which signals this machine has
$ man 7 signal        # and what each one means
```

And from the kernel itself, which sends them without anybody asking for them: when a child of yours finishes, when a timer of yours runs out, or when your program does something the hardware will not let it do — touching an address that is not its own, dividing by zero.

7.3.1 Which signals there are

Every event has a signal assigned to it, and the list is fixed by the kernel: you cannot invent new ones. They are numbers, but they have names and they are always written by name. How many there are depends on the system^{*}; these are the ones that will come up in this chapter:

Name	Default	When it arrives
SIGCHLD	ignore	a child has finished or stopped
SIGCONT	continue	carry on, if it was stopped
SIGSTOP	stop	stop the process
SIGINT	terminate	keyboard interrupt (C trl- C)
SIGALRM	terminate	the alarm() has run out
SIGKILL	terminate	kill the process
SIGSEGV	terminate and <i>core</i>	invalid memory reference
SIGUSR1	terminate	whatever you want
SIGUSR2	terminate	whatever you want

The last two are Unix’s gift to the programmer: they mean nothing, nobody ever sends them unless you send them, and they are there precisely so that you can use them for whatever suits you. In this chapter we will use them to synchronise processes.

Careful

Look at the first row. The parent gets a **SIGCHLD** every time a child of its dies — what everybody calls “the parent is notified” —, but the default action is to **ignore it**. If you do nothing, you will notice nothing.

^{*}POSIX.1-2018 requires 29. A **kill -1** on Linux gives 64; on macOS, 31.

7.4 The three dispositions

When a signal reaches a process, the kernel looks at what the process has associated with it, and there are only three possibilities:

- **SIG_DFL**: the default action, which is the one in the table above. Usually terminating the process, but some are ignored, one stops it and one makes it carry on.
- **SIG_IGN**: ignore it. The signal arrives and nothing happens, as if it had never been sent.
- **A function of your own**, which we will call the **handler** — the *signal handler* of the documentation. The kernel calls it when the signal arrives, and on returning from it the process carries on where it was.

When a program starts, every signal has **SIG_DFL** or **SIG_IGN**. Associating a function with one is what we will call **catching** the signal, and it is the business of the next section.

Careful

Two signals can be neither caught nor ignored nor blocked: **SIGKILL** and **SIGSTOP**. And that is no oversight. If they could be caught, a process could write an empty handler for both and keep the machine for ever: not even the administrator could stop it. `kill -9` is the system's last resort because the kernel guarantees that nobody can resist it.

7.5 Catching it: `sigaction()`

System call `sigaction`

```
int sigaction(int signum, const struct sigaction *act,
              struct sigaction *oldact);
```

The call queries or changes the disposition of *one* signal, **signum**. If **act** is not **NULL**, it sets the disposition you give it; if **oldact** is not **NULL**, it leaves in it the one that was there before, which is useful if you want to restore it later. Once made, the disposition stays as you left it until you call **sigaction()** again: this, which looks obvious, is one of the things the old implementations did not guarantee.

The structure is not the same everywhere*, but the three fields we care about are always there:

```
struct sigaction {
    void      (*sa_handler)(int); /* what to do          */
    sigset_t   sa_mask;           /* what is blocked while */
    int        sa_flags;          /* variants              */
};
```

sa_handler can be **SIG_DFL**, **SIG_IGN** or the address of a function of your own with the header **void name(int)**[†]. The integer the function receives is the signal that arrived, so you can associate the same function with several signals and tell them apart inside it.

sa_mask is a set of signals that will be blocked *while* the handler runs, and unblocked on leaving it. It is there to protect the handler from itself and from its neighbours.

sa_flags changes details of the behaviour. With **SA_RESETHAND**, once the signal has been handled the disposition goes back to the default one; with **SA_RESTART**, a blocked system call that the signal has interrupted is resumed instead of returning an error. With 0 there is no variant, and that is what we will use. It does mean something you have to keep in mind, though: without **SA_RESTART**, a blocked call that the signal interrupts is **not**

*On some systems the two possible dispositions are a **union**, and on others two separate fields. Either way only one of them is filled in.

[†]There is a second form, **sa_sigaction**, with three parameters, which gives the handler a great deal more information about who sent it the signal and why. We will not use it in this book.

resumed; it returns `-1` with `errno` set to `EINTR`. The library's `signal` function, which you will find in many examples, does just the opposite without saying so; Section 9.12 will show it with a program.

i Careful

`act` is a local variable: if you do not fill in the three fields, the ones you leave carry whatever rubbish is on the stack. The program will work — until the day it doesn't. **Always fill in all three**, even if two of them are empty.

With that you can already reprogram Ctrl-C:

Listing 7.2. Catching a signal

```

1  /* The signal handler. The kernel calls it, not us. */
2  static void handler(int sig)                                ❶
3  {
4      static const char bye[] = "\nSIGINT received; I'm off\n";
5
6      (void)sig;
7
8      write(1, bye, sizeof(bye) - 1);                          ❷
9      _exit(0);                                                ❸
10 }
11
12 int main(void)
13 {
14     struct sigaction act;
15     char buf[80];
16
17     sigemptyset(&act.sa_mask);                                ❹
18     act.sa_flags = 0;
19     act.sa_handler = handler;
20
21     if (sigaction(SIGINT, &act, NULL) < 0)                    ❺
22         error_and_exit("sigaction", 1);
23
24     for (;;) {

```

```

25     sprintf(buf, "doing my work...\n");
26     write(1, buf, strlen(buf));
27     sleep(1);
28 }
29 return 0;
30 }

```

- ❶ The handler. Notice that nobody in the program calls it: the kernel does.
- ❷ `write()` and not `sprintf()`: inside a handler you cannot call just anything. We talk about it in the last section but one.
- ❸ `_exit()` and not `exit()`, for the same reason.
- ❹ All three fields, always.
- ❺ From here on, a `SIGINT` no longer kills the process.

```

alumni@labS0:~/S4$ ./sig-catch
doing my work...
doing my work...
^C
SIGINT received; I'm off

```

7.6 Blocking it: sets and `sigprocmask()`

Catching a signal says *what* has to be done when it arrives. Blocking it says *when* you want it to arrive, and that is quite a different thing.

Every process has a set of blocked signals — the *signal mask*. While a signal is in it, if somebody sends it to you it does not reach you: the kernel keeps it as **pending** and will give it to you when you unblock it. It serves to protect the pieces of code where receiving a signal would hurt.

There are five functions for manipulating sets:

```

int sigemptyset(sigset_t *set);    /* empty set */
int sigfillset(sigset_t *set);    /* set with all of them */
int sigaddset(sigset_t *set, int signum);
int sigdelset(sigset_t *set, int signum);

```

```
int sigismember(const sigset_t *set, int signum);
```

These five **are not system calls**: they are C library functions, and you will find them in section 3 of the manual, not in section 2. They do not leave your process; they only touch a variable of yours. The one that is a system call is the one that comes next, which is the only one that changes anything inside the kernel.

System call `sigprocmask`

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

The first parameter says what has to be done with the set:

SIG_BLOCK adds the signals in `set` to the blocked ones.

SIG_UNBLOCK takes them out.

SIG_SETMASK replaces the whole mask with `set`.

If `set` is `NULL` nothing is changed, and with `oldset` you can read the mask that was there: that is how to ask what you have blocked at a given moment.

Blocking a signal, then, is four lines that you will always see together:

```
sigset_t block;  
  
sigemptyset(&block);  
sigaddset(&block, SIGUSR1);  
sigprocmask(SIG_BLOCK, &block, NULL);
```

7.6.1 The life of a signal

With blocking on the table, we can now say the words the documentation uses and that must not be confused.

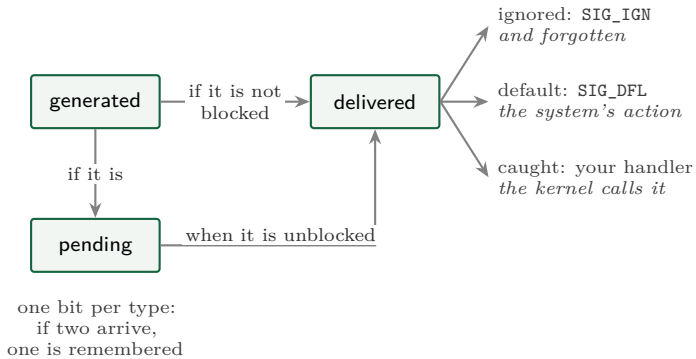


Figure 7.1. A signal is *generated*, may wait a while *pending*, and is *delivered* when the process finally receives it.

A signal is **generated** when the thing that causes it happens, and it is **delivered** when the process really receives it and its disposition runs. There can be a while between the two: if the signal is blocked, it stays **pending** until you unblock it. The time between being generated and being delivered is its lifetime.

i Careful

Pending signals are kept in one bit per signal type, not in a counter. If, while you have **SIGUSR1** blocked, somebody sends you five of them, on unblocking it you will run the handler **once**. This is not an implementation bug: signals are not a mechanism for counting things. If you need to count them, signals will serve to warn you and you will have to do the counting yourselves.

7.7 Waiting for it properly

Now we can pose the real problem: process A sends a signal to process B at some point, and B has to wait for it.

The first idea is busy waiting: a global variable that the handler sets to 1 and a loop in the main program that keeps looking at it.

```
while (!arrived)
    ;                               /* and meanwhile, burn CPU */
```

It works, and it is not always a bad idea: if the wait is going to be very short, the cost of blocking the process, scheduling another one and coming back (the context switch of Chapter 6) can be greater than that of going round a few times. But for long waits it is squandering the machine: while you go round, the other processes — and the one you are waiting for, which may well be one of yours — make no progress.

The second idea is to block, and that is where the problem starts.

7.7.1 The `pause()` race condition

There is a call, `pause()`, which puts the process to sleep until any signal arrives. It looks as if that should be enough:

Listing 7.3. How it must not be done

```
1  static volatile sig_atomic_t arrived = 0;           ❶
2
3  static void handler(int sig)
4  {
5      (void)sig;
6      arrived = 1;
7  }
8
9  int main(void)
10 {
11     struct sigaction act;
12     char buf[80];
13
14     sigemptyset(&act.sa_mask);
15     act.sa_flags = 0;
```

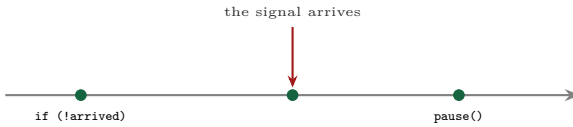
```

16     act.sa_handler = handler;
17     if (sigaction(SIGUSR1, &act, NULL) < 0)
18         error_and_exit("sigaction", 1);
19
20     sprintf(buf, "waiting for a SIGUSR1...\n");
21     write(1, buf, strlen(buf));
22
23     if (!arrived)                                ❷
24         pause();                                  ❸
25
26     sprintf(buf, "arrived\n");
27     write(1, buf, strlen(buf));
28     return 0;
29 }

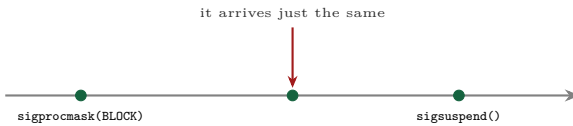
```

- ❶ `volatile sig_atomic_t` and not `int`: two things touch the variable at once. We explain it in the section on the handler.
- ❷ We check whether it has already arrived...
- ❸ ...and if not, we go to sleep.

Between balloon ❷ and balloon ❸ there is a gap. They are two lines of C, and therefore a good few machine instructions. If the signal arrives exactly there, the handler runs, sets `arrived` to 1, returns... and the program carries on where it was, which is doing the `pause()`. The signal we were waiting for has already gone by, and the `pause()` waits for the next one, which will never come.

With pause()

The handler runs and sets `arrived` to 1, but the condition had already been tested. The `pause()` waits for the *next* signal: the process stays there for ever.

With sigsuspend()

Since it is blocked, it is not delivered: it stays *pending*. The `sigsuspend()` unblocks it and goes to sleep in a single operation, so it is received straight afterwards.

Figure 7.2. The same signal, arriving at the same moment. At the top it is lost; at the bottom it stays pending and is delivered straight afterwards.

The program in Listing 7.3 will work nine times out of every ten you try it, and that is exactly why it is dangerous.

7.7.2 sigsuspend()

The solution has to do the unblocking and the going to sleep in a single operation, with no gap in between. That is what `sigsuspend()` is:

System call `sigsuspend`

```
int sigsuspend(const sigset_t *mask);
```

The call changes the process's mask to `mask` and puts the process to sleep, all at once, indivisibly. The process wakes up when a signal

that `mask` does not block arrives; on returning from the handler, the mask goes back to what it was.

The scheme is always the same: **block the signal before preparing anything**, and pass `sigsuspend()` a set that lets it through. That way, if the signal arrives before you are ready, it is not lost: it stays pending, and the `sigsuspend()` receives it immediately.

Listing 7.4. How it must be done

```

1  int main(void)
2  {
3      struct sigaction act;
4      sigset_t block, waitmask;
5      char buf[80];
6
7      sigemptyset(&block);
8      sigaddset(&block, SIGUSR1);
9      if (sigprocmask(SIG_BLOCK, &block, NULL) < 0)
10         error_and_exit("sigprocmask", 1);
11
12     sigemptyset(&act.sa_mask);
13     act.sa_flags = 0;
14     act.sa_handler = handler;
15     if (sigaction(SIGUSR1, &act, NULL) < 0)
16         error_and_exit("sigaction", 1);
17
18     sigfillset(&waitmask);
19     sigdelset(&waitmask, SIGUSR1);
20
21     sprintf(buf, "waiting for a SIGUSR1...\n");
22     write(1, buf, strlen(buf));
23     while (!arrived)
24         sigsuspend(&waitmask);
25
26     if (sigprocmask(SIG_UNBLOCK, &block, NULL) < 0)
27         error_and_exit("sigprocmask", 1);
28     sprintf(buf, "arrived\n");
29     write(1, buf, strlen(buf));
30     return 0;
31 }
```

- ❶ Block first. From here on, no `SIGUSR1` is lost.
- ❷ The waiting set: everything blocked except the one we are waiting for. That way we know exactly what can wake us up.
- ❸ The loop is not decorative: we could be woken by another signal, and then we have to go back to sleep.
- ❹ The error is not checked because `sigsuspend()` *always* returns `-1`, with `errno` set to `EINTR`. It is the only call in the book that does so.
- ❺ And the mask is left as it was.

```
alumni@labS0:~/S4$ ./sig-wait &
[1] 2402
waiting for a SIGUSR1...
alumni@labS0:~/S4$ ./sig-send 2402
arrived
[1]+  Done                  ./sig-wait
```

⚠ Question 1

In Listing 7.4, the set in balloon ❷ is built with `sigfillset` and a `sigdelset`. Some people, instead of that, save the old mask with the third parameter of `sigprocmask()` and pass it to `sigsuspend()`. Do the two do the same thing?

💡 Answer

Almost, and both are correct, but they do not say the same thing. Passing the old mask means “let in again everything that got in before”; `sigfillset` plus `sigdelset` means “let nothing in except this”. The second is more restrictive and leaves the program easier to reason about: there is a single thing that can wake you up. In this book we will always do the second.

7.8 The clock: `alarm()`

System call `alarm`

```
unsigned alarm(unsigned seconds);
```

It asks the kernel to send you a `SIGALRM` so many seconds from now. It returns the seconds that were left of the previous alarm, and with `alarm(0)` you cancel it.

Careful

Every process has **one** timer and no more. If you set an alarm when there already was one, the first disappears; they are not queued.

With that and `sigsuspend()` you can already do something periodic without burning a single CPU cycle waiting. This program launches a `ps` every two seconds:

Listing 7.5. Doing something every two seconds

```
1  #define ROUNDS 3
2
3  /* It has nothing to do: all we want is for it to wake us up. */
4  static void handler(int sig) ❗
5  {
6      (void)sig;
7  }
8
9  static void launch_ps(void)
10 {
11     pid_t ret = fork();
12
13     if (ret < 0)
14         error_and_exit("fork", 1);
15     if (ret == 0) {
```

```

16         execlp("ps", "ps", (char *)NULL);
17         error_and_exit("execlp", 1);
18     }
19     waitpid(ret, NULL, 0);
20 }
21
22 int main(void)
23 {
24     struct sigaction act;
25     sigset_t block, waitmask;
26     int i;
27
28     sigemptyset(&block);
29     sigaddset(&block, SIGALRM);
30     sigprocmask(SIG_BLOCK, &block, NULL);
31
32     sigemptyset(&act.sa_mask);
33     act.sa_flags = 0;
34     act.sa_handler = handler;
35     if (sigaction(SIGALRM, &act, NULL) < 0)
36         error_and_exit("sigaction", 1);
37
38     sigfillset(&waitmask);
39     sigdelset(&waitmask, SIGALRM);
40
41     for (i = 0; i < ROUNDS; i++) {
42         alarm(2);
43         sigsuspend(&waitmask);
44         launch_ps();
45     }
46     return 0;
47 }

```

- ❶ The handler is empty on purpose. It does not have to do anything: all we want is for the signal to get us out of the `sigsuspend()`.
- ❷ The alarm is set...
- ❸ ...and we wait. Here the process is in the BLOCKED state: have a look at the graph in Chapter 5.
- ❹ And the work is done after waking up, not inside the handler.

Notice a consequence of the blocking that will save you headaches. While the program is at balloon ❹ launching the `ps` — and waiting for it to finish — the `SIGALRM` is still blocked. If the alarm goes off in

the middle of that work, it is not lost: it stays pending, and the next time round the `sigsuspend()` picks it up straight away.

7.9 The children that die

In Chapter 5 the parent blocked in a `waitpid()` until the children finished one by one, and we said it was the thing to do so as not to leave zombies. But a parent that has work to do cannot spend its life waiting.

`SIGCHLD` is the way out. The kernel sends it to the parent every time a child finishes; if you catch it, you can reap them from the handler and never block.

Listing 7.6. Reaping the children without waiting for them

```

1  #define NCHILDREN 3
2
3  static volatile sig_atomic_t alive = NCHILDREN;
4
5  static void handler(int sig)
6  {
7      (void)sig;
8      while (waitpid(-1, NULL, WNOHANG) > 0)      ❶
9          alive--;
10 }
11
12 int main(void)
13 {
14     struct sigaction act;
15     sigset_t block, waitmask;
16     pid_t ret;
17     char buf[80];
18     int i;
19
20     sigemptyset(&block);
21     sigaddset(&block, SIGCHLD);
22     sigprocmask(SIG_BLOCK, &block, NULL);      ❷
23
24     sigemptyset(&act.sa_mask);
25     act.sa_flags = 0;

```

```

26     act.sa_handler = handler;
27     if (sigaction(SIGCHLD, &act, NULL) < 0)
28         error_and_exit("sigaction", 1);
29
30     for (i = 0; i < NCHILDREN; i++) {
31         if ((ret = fork()) < 0)
32             error_and_exit("fork", 1);
33         if (ret == 0) {
34             sleep(i + 1);
35             exit(0);
36         }
37     }
38
39     sigfillset(&waitmask);
40     sigdelset(&waitmask, SIGCHLD);
41     while (alive > 0)
42         sigsuspend(&waitmask);
43
44     sprintf(buf, "all reaped\n");
45     write(1, buf, strlen(buf));
46     return 0;
47 }

```

- ❶ WNOHANG means “if there is none to reap, don’t make me wait”. Inside a handler, blocking would be a sin.
- ❷ Blocked from the start, as always: the children can die before we are ready.
- ❸ They finish in order, but the program does not depend on that at all.
- ❹ And here the parent does not wait for any particular child: it waits for the counter to reach zero.

Question 2

In balloon ❶ there is a loop, and not a single call to `waitpid()`. Why? If a `SIGCHLD` arrives for every child that dies, shouldn’t reaping one per handler run be enough?

Answer

No, because pending signals are kept in a bit and not in a counter. If two children die while `SIGCHLD` is blocked — or while the handler

runs, which blocks it implicitly — only one of them is recorded. The handler will run a single time and there would be two children to reap. The loop with `WNOHANG` reaps all the ones there are and leaves when none is left.

7.10 What crosses a `fork()` and an `exec()`

Signals are the clearest example of Chapter 5’s distinction between what belongs to the program and what belongs to the process.

With a `fork()`, the child **inherits** the table of dispositions — it has a copy of the parent’s code, so the handlers exist in its memory — and it also inherits the mask of blocked signals. What it does not inherit is the pending signals or the timer: those were addressed to the parent, and the child is a new process with a new PID.

With an `exec()` almost the opposite happens. The dispositions you had set are lost and become the default ones again, which is unavoidable: the handlers’ code is no longer there*. But the process does not change, so it *does* keep the mask of blocked signals and any signals it had pending.

Listing 7.7. The countdown is not inherited; the handler is

```

1  int main(void)
2  {
3      struct sigaction act;
4      sigset_t block, waitmask;
5      char buf[96];
6      const char *who;
7      unsigned left;
8      pid_t ret;
9
10     sigemptyset(&block);
11     sigaddset(&block, SIGALRM);
12     sigprocmask(SIG_BLOCK, &block, NULL);
13     sigemptyset(&act.sa_mask);

```

*The ones that were at `SIG_IGN` go on being ignored: ignoring a signal takes no code.

```

14     act.sa_flags = 0;
15     act.sa_handler = handler;
16     if (sigaction(SIGALRM, &act, NULL) < 0)
17         error_and_exit("sigaction", 1);
18     sigfillset(&waitmask);
19     sigdelset(&waitmask, SIGALRM);
20
21     alarm(5);                                ❶
22     if ((ret = fork()) < 0)                  ❷
23         error_and_exit("fork", 1);
24     who = (ret == 0) ? " the child" : "the parent";
25
26     sleep(1);
27     left = alarm(0);                        ❸
28     sprintf(buf, "%s: I had %u seconds left\n", who, left);
29     write(1, buf, strlen(buf));
30
31     alarm(1);                                ❹
32     sigsuspend(&waitmask);
33     sprintf(buf, "%s: I have run my handler\n", who);
34     write(1, buf, strlen(buf));
35
36     if (ret > 0)
37         waitpid(ret, NULL, 0);
38     return 0;
39 }

```

- ❶ The parent sets the alarm, before cloning itself.
- ❷ And here there are two of them.
- ❸ `alarm(0)` cancels the alarm and returns the seconds that were left. That is how to ask without waiting.
- ❹ Now each one sets its own, and both run the handler the parent installed before the `fork()`.

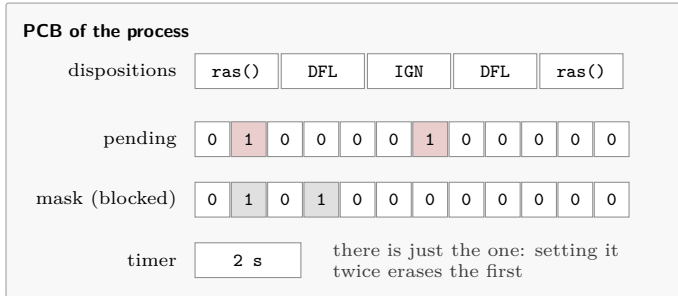
```

alumni@labS0:~/S4$ ./sig-inherit
the parent: I had 4 seconds left
the child: I had 0 seconds left
the child: I have run my handler
the parent: I have run my handler

```

7.11 What the kernel keeps of it

Everything we have been saying lives in the PCB, the structure from Chapter 5: signal management is per process, and every process has its own.



Signal 1 is pending *and* blocked: it will not be delivered until its bit is cleared from the mask. Signal 6 is pending and not blocked: it is delivered straight away.

Figure 7.3. Signals, as the kernel sees them: a table, two strips of bits and a countdown.

Four things. A **table of dispositions** with one entry per signal, which says whether it is `SIG_DFL`, `SIG_IGN` or the address of a handler. A **bitmap of pending signals**, one bit per signal — which is where their not being countable comes from. A **mask** of blocked ones, the same size. And a **timer**, just the one, which is `alarm()`'s.

The kernel handles signals inside the context of the process that receives them, and that means the process has to be running to receive any. A signal sent to a process that is in the `READY` queue is not delivered until the scheduler gives it the CPU back. That is why, when you look at the output of a program with signals, you cannot deduce the exact order of anything from it.

7.12 A handler that won't betray you

A handler runs between any two instructions of your program. That means it can catch the main program halfway through anything —

halfway through updating a structure, halfway through writing into a library buffer —, and that if the handler touches that same thing, what is left will be neither one thing nor the other.

There are few rules, and all of them have to be followed [1]:

Short. The shorter it is, the fewer chances of doing harm. The real work is done in the main program, as in Listing 7.5.

Safe functions only. Inside a handler you can only call *async-signal-safe* functions: the ones that can be interrupted and started again without harm. `write()` is one; `printf`, `sprintf` and `malloc` are not. The complete list is in `man 7 signal-safety`, and there is a long discussion of it in Stevens [3].

Save `errno`. The handler can make a system call that changes `errno` in the middle of a check in the main program. Save it at the start and restore it at the end.

Protect the shared variables. If the main program and the handler touch the same variable, you have a critical region. Block the signal while you touch it.

Declare them `volatile`. A variable the handler changes changes behind the compiler's back, and the compiler is otherwise free to keep it in a register and never read it again. The type for this is `volatile sig_atomic_t`.

Careful

The `sprintf` and the `write()` we use throughout the book will not do inside a handler: `write()` will, `sprintf` won't. That is why this chapter's handlers either write nothing, or write a constant string with `write()` directly.

7.13 Terminology

The vocabulary of signals is small, and it is the same one you will use all your life: `man`, the kernel source and any book you look at will

give it to you in English, and the lab will give it to you in Catalan. That is why both sides of every word are here. This is not a list of translations: these are the eight words that describe everything that can happen to a signal, and confusing two of them is confusing the mechanism.

English	Catalan	What it means
generate	generar	the event that causes the signal takes place
deliver	dipositar	reach the process and have its disposition run
lifetime	temps de vida	what goes by between being generated and being delivered
pending	pendent	generated and not yet delivered, because it is blocked
catch	capturar	having a handler of your own associated with <code>sigaction()</code>
ignore	ignorar	receive it and do nothing with it; it is a disposition, not the absence of one
block	bloquejar	delay its delivery; a blocked signal is not lost, it waits
signal handler	RAS	rutina d'atenció al signal: the function that runs when it is delivered

7.13.1 Caught does not mean delivered

Of these eight words there are two that get confused systematically, and we have to stop at them, because the confusion is not one of vocabulary: it is one of model.

Catching is a property of the *type* of signal, and it is done a single time. **Delivery** is an event that happens to *one* particular signal, and it happens every time one arrives.

When you do a `sigaction()` on `SIGUSR1`, you have caught `SIGUSR1` — every `SIGUSR1` that reaches you from now on, and none in particular.

None has arrived yet. The program in Listing 7.2 catches `SIGINT` on the third line of `main` and spends hours without a single one being delivered.

And the other way round: a signal can be delivered without being caught. If you have touched nothing, the `SIGINT` that arrives with `Ctrl-C` is delivered all the same — the default action runs and the process dies. The signal arrived, it was handled, it was delivered. You simply had not caught it.

Careful

This is where the most frequent lab mistake of this chapter comes from: “I have caught the signal but nothing happens”. Catching it does not make it arrive. If it does not arrive, either nobody is sending it to you, or you have it blocked, or the recipient is another process.

And two more confusions, easier to undo:

- **Ignoring** and **blocking** are not the same. An ignored signal arrives and is thrown away — it has been delivered, and the disposition was to do nothing. A blocked one has *not yet* arrived, and it will.
- **Generating** and **delivering** are not the same either. There can be a long while between the two, and it is exactly the while that Figure 7.1 lasts.

7.14 Exercises

1. Run `sig-race` in the background and send it the `SIGUSR1` straight away, without waiting. Now do it with a `sleep 1` in between. How many times out of ten does it hang in each case?
2. Modify `sig-wait` so that it waits for a `SIGUSR1` *or* a `SIGUSR2` and says which one it got. A single handler is enough: look at

the parameter it receives.

3. In `sig-catch`, change `SIGINT` to `SIGKILL` and try it. What does `sigaction()` tell you? Why?
4. Write a function that says which signals you have blocked at that moment. (Hint: `sigprocmask()` with the second parameter at `NULL`, and `sigismember` in a loop.) Take care when using it from a handler: what can you call in there and what can't you?
5. In `sig-children`, take the loop out of the handler and leave a single `waitpid()` there. Make the three children finish at the same time — take out their `sleep` — and see whether the program finishes. With `ps` you will be able to see what is left.
6. Write a program that, every second and for ten of them, writes how much time it has left, and stops when it is done. Without burning CPU and without using `sleep`.

Bibliography

- [1] Randal E. Bryant and David O'Hallaron. *Computer systems : a programmer's perspective*. eng. 3rd ed. Essex: Pearson, 2016. ISBN: 9781292101767. URL: https://discovery.upc.edu/permalink/34CSUC_UPC/11q3oqt/alma991004062589706711.
- [2] Marshall Kirk McKusick and George V. Neville-Neil. *The design and implementation of the FreeBSD operating system*. eng. Boston: Addison-Wesley, 2005. ISBN: 0201702452. URL: https://discovery.upc.edu/permalink/34CSUC_UPC/8e3cvcv/alma991003102659706711.
- [3] W. Richard. Stevens and Stephen A. Rago. *Advanced programming in the UNIX environment*. eng. 3a ed. Addison-Wesley professional computing series. Uper Saddle River, NJ: Addison-Wesley, 2013. ISBN: 0321637739. URL: https://discovery.upc.edu/permalink/34CSUC_UPC/11q3oqt/alma991004011309706711.