

Lecture 10.2

Signals the kernel's doorbell

Md Mahbub E Noor

Assistant Professor, Department of CSE
University of Barishal

```
● ● ● ./catch, with real Ctrl-C (real)  
PID 533: press Ctrl-C three times  
^C caught SIGINT  
^C caught SIGINT  
^C caught SIGINT  
three times: goodbye  
# Ctrl-C did not kill it
```

What you will learn today



Say what a signal is, where signals come from, and what each one does by default.



Catch, ignore and block signals with `sigaction` and `sigprocmask`; write a safe handler.



Follow a signal from `kill()` to your handler and back: the frame and `rt_sigreturn`.



Handle `EINTR`, `SIGCHLD` and timeouts in real programs.

Today's plan (60 min)

0-5

Recap of 10.1

5-15

What a signal is

15-27

Catching signals

27-40

Inside the kernel

40-47

Blocking, queueing

47-56

`EINTR`, `SIGCHLD`, `exec`

56-60

Checkpoint

Lab 10.2: mytimeout.c.

Quick recap of Lecture 10.1

Q1

When does `read()` on a pipe return 0?

When the pipe is empty AND no process holds a write end any more (fs/pipe.c 362).

Q2

In `yes` | `head -2`, who stops `yes`, and how?

`head` exits; `yes` writes into a pipe with no reader: the kernel sends `SIGPIPE` (13). Status 141. Today: signals.

Q3

Two processes did `counter++` 10 million times each and got 11.7 million. Why?

Load, add, store interleaved on two CPUs: updates were lost. Fix: an atomic instruction or a semaphore.

PART 1

What a signal is

A small number, delivered at any moment: "something happened".

A signal: a number the kernel delivers to a process

No.	Name	Typical source	Default action
2	SIGINT	Ctrl-C in the terminal	terminate
3	SIGQUIT	Ctrl-\ in the terminal	terminate + core dump
9	SIGKILL	kill -9: cannot be caught	terminate
11	SIGSEGV	the CPU: a bad memory access	terminate + core dump
13	SIGPIPE	writing to a pipe with no reader	terminate
14	SIGALRM	a timer set with alarm()	terminate
15	SIGTERM	kill PID: "please stop"	terminate
17	SIGCHLD	a child ended or stopped	ignore
19 / 20	SIGSTOP / SIGTSTP	kill -STOP / Ctrl-Z	stop

DIFFERENT FROM C

A function call you did not make

A handler is an ordinary C function, but nobody in your code calls it: the kernel makes your program jump there between any two instructions.

From the kernel: something your program did: SIGSEGV, SIGPIPE, SIGFPE; or an event: SIGCHLD, SIGALRM.

From the terminal: keys typed by the user: Ctrl-C, Ctrl-\, Ctrl-Z.

From a process: kill(pid, sig), the kill command, raise(sig).

Analogy: a doorbell. It says only "someone rang" (and which bell), not what they want.

`kill -l` lists all 64 numbers

Book: CS:APP §8.5 (Figure 8.26); Kerrisk, TLPI Ch. 20

Default actions: tested, and in the kernel source

```
./deflt (real; ulimit -c unlimited)
```

```
2 INT      -> killed
3 QUIT     -> killed (core dumped)
11 SEGV    -> killed (core dumped)
15 TERM    -> killed
9 KILL     -> killed
10 USR1    -> killed
13 PIPE    -> killed
14 ALRM    -> killed
17 CHLD    -> ignored (still running; we killed it)
19 STOP    -> stopped
20 TSTP    -> stopped
```

```
include/linux/signal.h, Linux 6.8 (real lines)
```

```
419 #define SIG_KERNEL_ONLY_MASK (
      rt_sigmask(SIGKILL) | rt_sigmask(SIGSTOP))
422 #define SIG_KERNEL_STOP_MASK (
      SIGSTOP | SIGTSTP | SIGTTIN | SIGTTOU
426 #define SIG_KERNEL_COREDUMP_MASK (
      SIGQUIT | SIGILL | SIGTRAP | SIGABRT |
      SIGFPE | SIGSEGV | SIGBUS | SIGSYS | ...
434 #define SIG_KERNEL_IGNORE_MASK (
      SIGCONT | SIGCHLD | SIGWINCH | SIGURG
      /* every other signal: terminate */
```

Five defaults: terminate, terminate with a core dump (a file with the memory, for gdb), ignore, stop, continue. A program can change all of them, except for SIGKILL and SIGSTOP: those always work, so the administrator can always stop a program.

Ctrl-C: from the keyboard to a whole process group

● ● ● interactive bash, in a terminal (real)

```
$ sleep 100 | sleep 100 &
$ ps -o pid,ppid,pgid,sid,tpgid,stat,comm
  PID  PPID  PGID   SID TPGID  STAT  COMMAND
   682   679   682   682   685  Ss    bash
   683   682   683   682   685  S     sleep
   684   682   683   682   685  S     sleep
   685   682   685   682   685  R+    ps
$ fg
sleep 100 | sleep 100
^C
$ echo "status $?"
status 130
```

The path: the key Ctrl-C is the byte 0x03. The terminal driver (not the shell!) sees it, and sends SIGINT to the foreground group. Both sleeps die: status 130 = 128 + 2.

● ● ● drivers/tty/n_tty.c, Linux 6.8 (real)

```
1363 if (c == INTR_CHAR(tty)) {
1364     n_tty_receive_signal_char(tty,
                               SIGINT, c);
...
1067 static void __isig(int sig, ...)
1071     kill_pgrp(tty_pgrp, sig, 1);
```

Process group: A pipeline is one group: both sleeps have PGID 683. The shell puts each job in its own group.

Foreground group: The terminal knows one group: TPGID. Ctrl-C goes to every process in that group, not to the shell.

PART 2

Catching signals

sigaction: your function instead of the default action.

sigaction: install a handler

● ● ● catch.c

```
static volatile sig_atomic_t hits = 0;
static void on_int(int sig)
{
    hits++;
    write(1, " caught SIGINT\n", 16); /* safe */
}
int main(void) {
    struct sigaction sa;
    memset(&sa, 0, sizeof sa);
    sa.sa_handler = on_int; /* or SIG_IGN, SIG_DFL */
    sigemptyset(&sa.sa_mask); /* block nothing extra */
    sigaction(SIGINT, &sa, NULL);
    while (hits < 3)
        pause(); /* sleep until a signal */
}
```

SAME AS C

A function pointer

sa_handler holds a void (*)(int), like qsort's comparator: you give the kernel a function to call later.

Field	Meaning
sa_handler	your function, SIG_IGN or SIG_DFL
sa_mask	signals blocked while the handler runs
sa_flags	SA_RESTART, SA_SIGINFO, ...

● ● ● QEMU guest, Ubuntu 6.8 (real)

```
PID 127: press Ctrl-C three times
catch waits in: __do_sys_pause
caught SIGINT      # kill -INT 127
caught SIGINT
caught SIGINT
three times: goodbye
```

sig_atomic_t + volatile: the only safe way for a handler to tell main something.

Book: Kerrisk, TLPI §20.13 (sigaction); CS:APP §8.5.3

What strace and gdb see

```
● ● ● strace ./catch (sent kill -INT three times; real)

rt_sigaction(SIGINT, {sa_handler=0x562616afb320, sa_mask=[],
  sa_flags=SA_RESTORER, sa_restorer=0x7f7819845330}, NULL, 8) = 0
write(1, "PID 542: press Ctrl-C three time...", 34) = 34
pause() = ? ERESTARTNOHAND (To be restarted if no handler)
--- SIGINT {si_signo=SIGINT, si_code=SI_USER, si_pid=536, si_uid=0} ---
write(1, "  caught SIGINT\n", 16) = 16
rt_sigreturn({mask=[]}) = -1 EINTR (Interrupted system call)
pause() = ? ERESTARTNOHAND ... (twice more)
write(1, "three times: goodbye\n", 21) = 21
```

```
● ● ● gdb: break on_int, then kill -INT
```

```
Breakpoint 1, ... in on_int ()
$ (gdb) bt
#0  on_int ()
#1  <signal handler called>
#2  __libc_pause () at
    .../linux/pause.c:29
#3  main ()
```

sa_restorer

a libc function, `__restore_rt`: 3 instructions that call `rt_sigreturn` (system call 15). The handler "returns" there.

--- SIGINT ---

the moment of delivery. `si_pid = 536`: the kernel tells you who sent it (with `SA_SIGINFO`).

<signal handler called>

`gdb` sees the handler on top of `pause()`, as if `pause` had called it. Part 3 shows how.

Rules for a handler that never breaks your program

- 1 **Call only async-signal-safe functions:** write, _exit, kill, waitpid ... Not printf, malloc, free.
- 2 **Do little:** set a volatile sig_atomic_t flag, and let main do the work.
- 3 **Save and restore errno:** a call in the handler may change the errno main is about to read.
- 4 **Use sigaction, not signal():** it says exactly what you get (flags, mask).
- 5 **SIGKILL and SIGSTOP cannot be caught or ignored:** sigaction returns EINVAL.

DIFFERENT FROM C

Like a second thread, without locks

A handler can interrupt main between any two instructions, but it cannot wait for main to finish: no lock can protect you here.

```
./nokill (real)
sigaction(SIGKILL, handler) = -1 Invalid argument
sigaction(SIGSTOP, handler) = -1 Invalid argument
sigaction(SIGTERM, handler) = 0 (allowed)
```

```
kernel/signal.c 4129: if (... (act && sig_kernel_only(sig)))
return -EINVAL;
```

Why not printf? The signal may arrive while main is inside printf, halfway through updating stdout's buffer (and holding its lock). If the handler calls printf too, the buffer is corrupted, or the program waits for a lock it holds itself: a deadlock.

Book: Kerrisk, TLPI §21.1 (safe functions); man 7 signal-safety

PART 3

How the kernel delivers a signal

A bit in a set, a flag, and a fake function call on your own stack.

Sending: set a bit, wake the task

```
kernel/signal.c, Linux 6.8 (real lines)
3807 SYSCALL_DEFINE2(kill, pid_t, pid, int, sig)
3813     return kill_something_info(sig, &info, pid);
...
1068 static inline bool legacy_queue(..., int sig)
1070     return (sig < SIGRTMIN) &&
        sigismember(&signals->signal, sig);
1094 if (legacy_queue(pending, sig))
1095     goto ret; /* already pending: drop */
1168 sigaddset(&pending->signal, sig);
... complete_signal() → signal_wake_up()
763 set_tsk_thread_flag(t, TIF_SIGPENDING);
...
88 if (is_global_init(t) && sig_kernel_only(sig))
```

A set of bits

Each task has a pending set: one bit per signal (task_struct → pending, signal → shared_pending). Sending = setting a bit.

Wake it up

TIF_SIGPENDING marks the task; if it sleeps interruptibly (S), it is woken up. Nothing else happens yet.

Already pending?

A standard signal that is already pending is simply dropped (line 1094): signals are not counted.

PID 1 is protected (line 88): init gets no signal it has not asked for, not even SIGKILL. In the guest, our init shell ran `kill -9 1`: "returned 0; still alive". Lecture 7.2: if init dies, the kernel panics.

Book: Bovet & Cesati Ch. 11 (generating a signal)

Delivering: on the way back to user mode

1

Leaving the kernel (after a system call, an interrupt, or a wake-up):
exit_to_user_mode_loop checks TIF_SIGPENDING (entry/common.c 104).

2

get_signal (signal.c 2661) takes the lowest pending, unblocked signal out of the set.

3

SIG_IGN: nothing (2791). **Handler**: break out, run it (2793).

4

Default: ignore (2806), stop (2823), core dump (2879), or **do_group_exit** (2893).

5

handle_signal → **x64_setup_rt_frame**: build the frame for the handler (next slide).

kernel/signal.c get_signal, Linux 6.8 (real)

```
2786 ka = &sigband->action[signr-1];
2791 if (ka->sa.sa_handler == SIG_IGN)
2792     continue;
2793 if (ka->sa.sa_handler != SIG_DFL) {
2795     ksig->ka = *ka; ... break;
2796 }
2806 if (sig_kernel_ignore(signr)) continue;
2823 if (sig_kernel_stop(signr)) { ... }
2879 do_coredump(&ksig->info);
2893 do_group_exit(ksig->info.si_signo);
```

Why only on the way out? The kernel can safely change a process's registers only when it is about to return to user mode. So a signal takes effect at the next system call return, interrupt or timer tick: microseconds, not instantly.

Book: Bovet & Cesati Ch. 11 (delivering a signal)

The trick: a fake function call on your own stack

```
arch/x86/kernel/signal_64.c, Linux 6.8 (real)

164 int x64_setup_rt_frame(struct ksignal *ksig,
                        struct pt_regs *regs)
175     frame = get_sigframe(ksig, regs, ...);
    /* save registers + mask into frame->uc */
188     unsafe_put_user(ksig->ka.sa.sa_restorer,
                    &frame->pretcode, ...);
202     regs->di = ksig->sig;
210     regs->ip = (unsigned long) ksig->ka.sa.sa_handler;
212     regs->sp = (unsigned long)frame;
246 SYSCALL_DEFINE0(rt_sigreturn)
263     ... restore_sigcontext(regs, &frame->uc...)
```

struct rt_sigframe, on the user stack

pretcode = sa_restorer

the return address of the handler

uc: saved rip, rsp, all registers

where main was: inside pause()

uc: saved signal mask

restored afterwards

info: siginfo

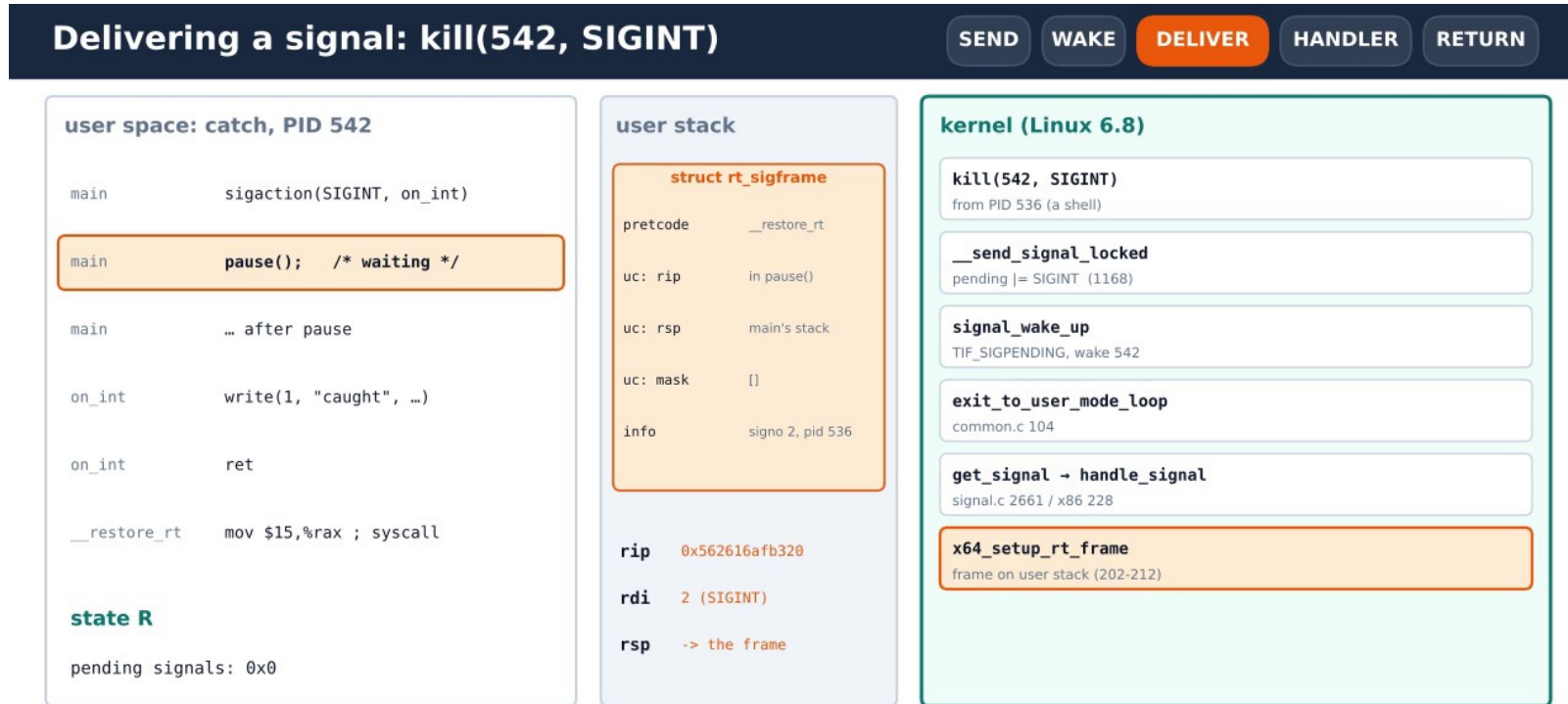
signo 2, si_pid 536, si_code SI_USER

1 The kernel returns to user mode with **rip = on_int**, **rdi = 2**: the CPU runs the handler as if it had been called. Its **ret** jumps to **pretcode**: libc's **__restore_rt**.

2 **__restore_rt** is `mov $15,%rax ; syscall: rt_sigreturn` copies the saved registers back. **main** continues inside `pause()`, which returns `-1, EINTR`.

Book: Bovet & Cesati Ch. 11 (catching the signal)

Watch one SIGINT travel to its handler and back



x64_setup_rt_frame saves the user's registers and signal mask in a struct rt_sigframe ON THE USER STACK, then edits the saved registers: rip = the handler, rdi = 2, rsp = the frame.

Watch for

1. Sending only **sets a bit**.
2. The sleeper is **woken**: S → R.
3. Delivery happens **on the way out** of the kernel.
4. The **frame** goes on the user stack.
5. **rt_sigreturn** puts everything back.

Click the video to play or pause. About 1.3 minutes, no sound.

PART 4

Blocking and queueing

Hold a signal back for a while; and find out how many arrive.

Blocked signals wait; standard signals are not counted

mask.c (core lines)

```
sigemptyset(&set); sigaddset(&set, sig);
sigprocmask(SIG_BLOCK, &set, &old); /* hold back */
for (i = 0; i < 3; i++) kill(getpid(), sig);
... print ShdPnd and SigBlk from /proc/self/status
sigprocmask(SIG_SETMASK, &old, NULL); /* let them in */
```

./mask (real; the guest prints the same)

```
SIGUSR1 (10) sent 3 times while blocked: handler ran 0 times
  ShdPnd: 00000000000000200
  SigBlk: 00000000000000200
  after unblocking: handler ran 1 times
SIGRTMIN (34) sent 3 times while blocked: handler ran 0 times
  ShdPnd: 00000002000000000
  SigBlk: 00000002000000000
  after unblocking: handler ran 3 times
```

Blocked ≠ ignored: A blocked signal stays pending and is delivered when you unblock it. An ignored signal is thrown away.

0x200 = bit 9: Bit $n - 1$ stands for signal n : 0x200 is SIGUSR1 (10). The pending set is just these 64 bits.

1, not 3: Standard signals (1–31) are one bit each: three sends while pending = one delivery (legacy_queue). Real-time signals (32–64) are queued: three = three.

Use it for: critical sections: block SIGINT while you update a file, unblock afterwards.

Book: Kerrisk, TLPI §20.10–20.12; CS:APP §8.5.4

PART 5

Signals in real programs

Interrupted system calls, collecting children, and what exec keeps.

A signal during a blocking call: EINTR, or restart

```
● ● ● ./eintr (read() on an empty pipe, alarm(1); real)
```

```
no flags : read() from an empty pipe, alarm in 1 s
[SIGALRM handler ran]
read returned -1, errno = Interrupted system call
SA_RESTART: read() from an empty pipe, alarm in 1 s
[SIGALRM handler ran]
read returned 1 (the data came at 2 s)
```

```
● ● ● strace -f ./eintr (real)
```

```
read(3, ..., 1) = ? ERESTARTSYS (To be restarted if SA_RESTART is set)
--- SIGALRM {si_signo=SIGALRM, si_code=SI_KERNEL} ---
rt_sigreturn({mask=[]}) = -1 EINTR
read(3, ..., 1) = ? ERESTARTSYS ...
--- SIGALRM ... ---
rt_sigreturn({mask=[]}) = 0
read(3, <unfinished ...> # the same read, again
<... read resumed>"x", 1) = 1
```

```
● ● ● arch/x86/kernel/signal.c handle_signal (real)
```

```
239 switch (syscall_get_error(current, regs)) {
241 case -ERESTARTNOHAND:
242     regs->ax = -EINTR; break;
245 case -ERESTARTSYS:
246     if (!(ksig->ka.sa.sa_flags & SA_RESTART)) {
247         regs->ax = -EINTR; break; }
250     fallthrough;
252     regs->ax = regs->orig_ax;
253     regs->ip -= 2;
```

regs->ip -= 2: the `syscall` instruction is 2 bytes (0f 05, Lecture 9). The kernel moves the saved rip back by 2 and puts the number back in `rax`: after the handler, the CPU simply runs `syscall` again.

SIGCHLD: collect all your children, not one per signal

```
● ● ● sigchld.c (handler)
static void on_chld(int sig)
{
    int saved = errno;          /* rule 3 */
    signals++;
    while (waitpid(-1, NULL, WNOHANG) > 0)
        reaped++;              /* every finished child */
    errno = saved;
}
```

```
● ● ● five children end while SIGCHLD is blocked (real)
while loop: children: 5  SIGCHLD handler calls: 1  reaped: 5
if (once): children: 5  SIGCHLD handler calls: 1  reaped: 1
           # 4 zombies left (Homework 5)
```

One signal, five children

SIGCHLD is a standard signal: five sends while pending = one delivery. The handler must loop until waitpid finds no more.

WNOHANG

"do not wait": returns 0 when no child has ended, so the loop cannot block the program.

Your shell

Lab 8.1's mysh collected background children before each prompt. With a SIGCHLD handler it could do it at once, like bash's "[1]+ Done".

Book: CS:APP §8.5.5 (signals are not queued); TLPI §26.3

Across fork and exec: what survives?

	after fork (child)	after exec (new program)
a handler you installed	kept (same code)	reset to SIG_DFL: the code is gone
SIG_IGN (ignored)	kept	KEPT
blocked mask	kept	kept
pending signals	empty	kept

```
● ● ● bash (real)
$ yes | head -1 ; echo ${PIPESTATUS[*]}
y 141 0
$ trap '' PIPE ; yes | head -1 ; echo ${PIPESTATUS[*]}
y yes: standard output: Broken pipe 1 0
```

This is how nohup works: it ignores SIGHUP, then execs your program.

```
● ● ● kernel/signal.c 531 (real)
531 flush_signal_handlers(t, force_default)
535 for (i = _NSIG; i != 0; i--) {
536     if (force_default ||
        ka->sa.sa_handler != SIG_IGN)
537         ka->sa.sa_handler = SIG_DFL;
        /* exec.c 1385: flush_signal_handlers(me, 0)
*/
```

A real trap. An ignored signal stays ignored in every program you exec. A shell that ignores Ctrl-C must set SIGINT back to SIG_DFL in the child before exec, or the user can no longer stop that program (Homework 1).

LAB 10.2

mytimeout.c

Run a program, but stop it after N seconds: alarm, EINTR, kill, and SIGKILL as a last resort.

Lab 10.2 tasks, step by step

- 1 **TODO 1** in mytimeout_starter.c: install on_alarm for SIGALRM with sigaction, **without** SA_RESTART
- 2 **TODO 2:** alarm(secs) after starting the program
- 3 **TODO 3:** waitpid returned EINTR: first alarm → SIGTERM and alarm(2); second alarm → SIGKILL
- 4 **TODO 4:** exit status like coreutils: 124 if timed out, else the child's status (or 128 + signal)
- 5 Test: ./mytimeout 2 sleep 10, ./mytimeout 5 sleep 1, ./mytimeout 1 ./stubborn; compare with timeout

Think: why must SA_RESTART be off here? (What would waitpid do after the alarm if it were on?)

```
● ● ● strace ./mytimeout 1 ./stubborn (real)
rt_sigaction(SIGALRM, {..., sa_flags=SA_RESTORER, ...}) = 0
alarm(1)                                = 0
wait4(1857, ...) = ? ERESTARTSYS
--- SIGALRM {si_code=SI_KERNEL} ---
rt_sigreturn({mask=[]}) = -1 EINTR
kill(1857, SIGTERM)         = 0
alarm(2)
# stubborn ignores SIGTERM
wait4(1857, ...) = ? ERESTARTSYS
--- SIGALRM ... ---
rt_sigreturn({mask=[]}) = -1 EINTR
kill(1857, SIGKILL)       = 0
+++ killed by SIGKILL +++ # 1857
wait4 ... WTERMSIG(s) == SIGKILL ... = 1857
+++ exited with 124 +++
```

Checkpoint: test yourself

Q1

You press Ctrl-C. Who sends SIGINT, and to whom?

→ the terminal driver, to every process in the foreground process group

Q2

Which two signals can never be caught, blocked or ignored?

→ SIGKILL and SIGSTOP

Q3

Why must a handler not call printf?

→ it may interrupt main inside printf: corrupted buffer or deadlock

Q4

Where does the kernel put the frame for the handler?

→ on the user stack; rip = handler, return address = __restore_rt

Q5

SIGUSR1 sent 3 times while blocked. How many deliveries?

→ 1: standard signals are not queued (real-time ones are)

Key ideas from today



A signal is a number: from the kernel, the terminal, or kill(). Defaults: terminate, core, ignore, stop.



sigaction installs a handler. Keep it tiny: async-signal-safe calls, a sig_atomic_t flag, save errno.



Sending sets a pending bit and wakes the task; delivery happens on the way back to user mode.



The kernel fakes a call: a frame on the user stack, rip = handler; rt_sigreturn restores everything.



Blocked signals wait; standard signals are not counted, so a SIGCHLD handler must loop.



Blocking calls return EINTR, or restart with SA_RESTART. Ignored signals survive exec.

Homework 10.2 (due before Lecture 11.1)

1. Shell Ctrl-C

Make your `mysh` survive Ctrl-C: the shell ignores SIGINT, its children must not. Test with `sleep 5` and Ctrl-C. What happens if the child does not reset SIGINT?

2. 34 or 32?

`kill -l` says SIGRTMIN is 34; the kernel header `arch/x86/include/uapi/asm/signal.h` says 32. Who is right, and why do they differ?

3. Back again

`segv_loop.c`: a SIGSEGV handler that just returns. Predict what happens; run it. Which instruction runs after the handler, and why?

4. Stubborn

Run `timeout 1 ./stubborn` and `timeout -k 2 1 ./stubborn` (`coreutils`). Explain both results, and compare with your `mytimeout`.

5. Only once

Change `sigchld.c`'s `while` into `if`. How many children are reaped? Where are the others? (ps)

Next lecture and reading

NEXT: LECTURE 11.1

Virtual memory

Why do parent and child see the same address with different values? Address spaces, pages and page tables, mmap, and what really happens at a page fault.

Main text

M. Kerrisk, **TLPI**, Ch. 20–22 (signals) today. R. Love, **LKD**, Ch. 15 (the process address space) for 11.1.

Kernel side

D. P. Bovet & M. Cesati, **Understanding the Linux Kernel**, 3rd ed., Ch. 11 (signals) today; Ch. 2 and 9 for 11.1.

C side

R. E. Bryant & D. R. O'Hallaron, **CS:APP**, 3rd ed., §8.5 (signals) today; Ch. 9 (virtual memory) for 11.1.

Lab files

`lab10_files.zip`: `mytimeout_starter.c`, `stubborn.c` and the signal demos.