

CSE 4105 • SYSTEM PROGRAMMING • SHORT COURSE

Class 8

System Calls, Pipes and Signals

how a program asks the kernel, and how processes talk

Md Mahbub E Noor

Assistant Professor, Department of CSE

University of Barishal

CLASS

8 / 10

19 slides, taken from the full course:
Lecture 9, Lecture 10.1, Lecture 10.2

A system call is a request; a library function is just code

your code

```
printf("n=%d\n", n);
```

C library (libc)

formats the text into a buffer; later calls write()

the door

the syscall instruction, number 1 in rax

the kernel

sys_write → the terminal driver

user mode, ring 3

kernel mode, ring 0

man section 3: library	man section 2: system call
printf, fopen, fread	write, open, read
malloc, free	brk, mmap, munmap
strlen, qsort, sin	(none: pure computation)
time, gettimeofday	(none: the vDSO, Part 5)
system, popen	fork, execve, wait4

SAME AS C

Looks like a C function

write(1, buf, n) is a normal C function in libc: same call, same return value. Inside, it is three or four instructions around `syscall`.

DIFFERENT FROM C

But it leaves your program

A C function runs your code with your rights. A system call runs **kernel** code with **all** rights, so the kernel checks every argument.

Book: Kerrisk, TLPI §3.1; Love, LKD Ch. 5

strace: watching the door

```
strace -c -S calls ls -l /usr/bin (real, top lines)
```

% time	seconds	usecs/call	calls	errors	syscall
28.59	0.015423	10	1468	1468	lgetxattr
40.93	0.022080	15	1468		statx
16.13	0.008703	9	912		listxattr
13.21	0.007127	12	556		readlink
0.50	0.000272	10	26		write
...					
100.00	0.053948	11	4552	1477	total

1467 files, 4552 system calls: about 3 per file (statx for the size and date, lgetxattr and listxattr for security labels). Every one crossed the door.

Option	What it does
-C	count calls and time per call
-f	follow children (fork, clone)
-e trace=write,openat	only these calls
-T / -tt	time in each call / time stamps
-o file	write the trace to a file

How strace works: it uses another system call, `ptrace`. The kernel stops the traced process at every system call entry and exit and lets strace read its registers: the number in rax, the arguments, the result.

The register contract (x86-64 Linux)

	system call (syscall)	C function call (call)
what to run	rax = the number (1 = write)	the address of the function
argument 1, 2, 3	rdi, rsi, rdx	rdi, rsi, rdx
argument 4	r10	rcx
argument 5, 6	r8, r9	r8, r9 (more on the stack)
result	rax (-4095 ... -1 = error)	rax
destroyed	rcx, r11 (by the CPU)	rax, rcx, rdx, rsi, rdi, r8-r11

Why r10, not rcx? The syscall instruction itself writes the return address into rcx and the flags into r11. So argument 4 must travel in another register, and the kernel moves it back to rcx before calling C code.

● ● ● entry_64.S, Linux 6.8 (real)

```
68 * Registers on entry:
69 * rax  system call number
70 * rcx  return address
71 * r11  saved rflags
72 * rdi  arg0
73 * rsi  arg1
74 * rdx  arg2
75 * r10  arg3 (needs to be moved
           to rcx to conform to C ABI)
```

C link: a system call is a function call by **number**, not by address. You cannot jump into the kernel's code; you can only ask for entry number 1.

Book: Bryant & O'Hallaron, CS:APP §8.1.3 (Linux/x86-64 system calls)

Hello world with two system calls and nothing else

```
● ● ● rawhello.s (gcc -nostdlib -static -o rawhello rawhello.s)
```

```
_start:
    mov    $1, %rax          # number 1 = write
    mov    $1, %rdi          # fd 1: the screen
    lea    msg(%rip), %rsi   # address of the bytes
    mov    $len, %rdx        # how many bytes
    syscall                 # enter the kernel
    mov    $60, %rax         # number 60 = exit
    mov    $0, %rdi          # exit status 0
    syscall                 # never returns
msg: .ascii "hello from the kernel, no libc\n"
```

```
● ● ● ./rawhello ; strace ./rawhello (real)
```

```
hello from the kernel, no libc
execve("./rawhello", ...) = 0
write(1, "hello from the ...", 31) = 31
exit(0)                        = ?
```

Program	System calls	Who makes them
rawhello	2 (+ execve)	our 9 instructions
hello.c, gcc -static	16	libc's start-up code
hello.c, normal	34	ld.so loads libc first

Where do 1 and 60 come from? /usr/include/x86_64-linux-gnu/asm/unistd_64.h: `#define __NR_write 1, __NR_exit 60`. The same numbers are in the kernel's table, arch/x86/entry/syscalls/syscall_64.tbl. This is the whole interface: a number and six registers.

Watch one write() go in and come back

One system call: write(1, msg, 31)

SET UP SYSCALL DISPATCH WRITE RETURN

user space: rawhello (ring 3)

```
401000  mov  $1,%rax
401007  mov  $1,%rdi
40100e  lea  msg,%rsi
401015  mov  $31,%rdx
40101c  syscall
40101e  mov  $60,%rax
401025  mov  $0,%rdi
40102c  syscall
```

msg at 0x402000: "hello from the kernel, no libc\n"

CPU registers

rax 1

rdi 1

rsi 0x402000

rdx 0x1f

rcx 0x40101e

r11 0x202

rip kernel

rsp kernel stack

kernel mode, ring 0

kernel space (Linux 6.8)

entry_SYSCALL_64
entry_64.S 87

kernel stack: pt_regs (part)

ss	user
sp	...8af0
flags	0x202
cs	user
ip	0x40101e
orig_ax	1
di	1
si	0x402000
dx	31
ax	-ENOSYS

RETURN

Watch for

1. **rax** holds the number, not a name.
2. The CPU fills **rcx** and **r11** itself.
3. The kernel saves everything in **pt_regs**.
4. A **switch** picks the function.
5. The answer comes back in **rax**: 31.

Step 4: entry_SYSCALL_64 switches to this task's own kernel stack (the user stack cannot be trusted) and pushes every user register there: struct pt_regs. The user's registers are now safe.

Click the video to play or pause. About 1.5 minutes, no sound.

The kernel returns `-errno`; libc returns `-1` and sets `errno`

```
./errno_demo (real)
raw syscall: write(42, "x", 1) returns -9
libc: write(42, "x", 1) returns -1 errno = 9 Bad file descriptor
libc: syscall(1000) returns -1 errno = 38 Function not implemented
raw syscall: syscall(1000) returns -38
libc: write(1, "ok\n", 3) returns 3
$ strace ./errno_demo
write(42, "x", 1) = -1 EBADF (Bad file descriptor)
syscall_0x3e8(...) = -1 ENOSYS (Function not implemented)
```

-errno	the result
-4095 ... -1: error	0 or more (or an address): success

```
include/linux/err.h 18: #define MAX_ERRNO 4095
```

Why -4095 ... -1 only? `mmap` returns an address, and an address can look like a big "negative" number. The top 4095 values are reserved for errors.

1 Check the return value first: `errno` is only meaningful after a call **failed**. A successful call may leave any old value in it.

2 Save `errno` at once (`int e = errno;`): `printf` and other calls can change it before you print it.

C link: `perror("open")` and `strerror(errno)` turn the number into the text you saw in Lecture 8.1's mysh: "No such file or directory" is `errno 2` (`ENOENT`).

Separate memories, so the kernel must carry the message

Way	What moves	Who can use it	Lecture
pipe	a stream of bytes, one way	parent and child (related)	today
FIFO (named pipe)	a stream of bytes, one way	any process that can open the name	today
shared memory	nothing moves: both see the same page	related (mmap) or by name (shm_open)	today
Unix domain socket	bytes or messages, both ways	any process, by a file name	CN course
message queue	whole messages, with priorities	any process, by a name	TLPI Ch. 52
signal	only a number: "something happened"	any process you may signal	10.2

Pipe: a tube between two houses: what goes in one end comes out the other, in order

Shared memory: a whiteboard both houses can see: fastest, but two people writing at once make a mess

Signal: a doorbell: it says only that someone rang

SAME AS C

Like two functions

Two C functions talk through arguments and return values: the caller hands data to the callee. A pipe hands bytes from one process to another.

DIFFERENT FROM C

But no shared variables

After fork, x in the child is a copy (Homework 8.1: same &x, different values). A global variable cannot carry a message between processes.

Book: Kerrisk, TLPI Ch. 43 (IPC overview); Bovet & Cesati Ch. 19

pipe(): two file descriptors and a buffer in the kernel

● ● ● pipe1.c (shortened)

```
int fd[2]; pipe(fd);          /* fd[0] read, fd[1] write */
if (fork() == 0) {            /* child: the reader */
    close(fd[1]);
    n = read(fd[0], buf, sizeof buf - 1);
    ...
    n = read(fd[0], buf, sizeof buf); /* again */
    return 0;
}
close(fd[0]);                  /* parent: the writer */
write(fd[1], msg, strlen(msg));
close(fd[1]);                  /* gives the child EOF */
wait(NULL);
```

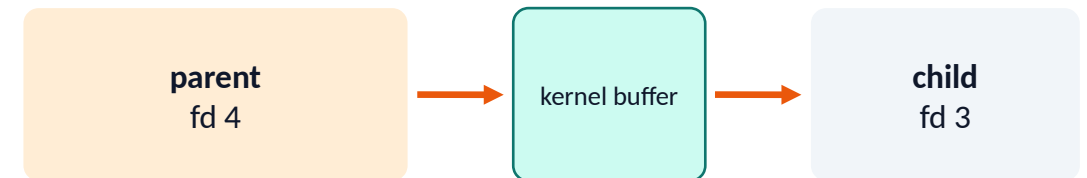
SAME AS C

Like a file

read and write work as on any open file (Lecture 12). fds 0, 1, 2 are taken: the pipe gets 3 and 4.

● ● ● ./pipe1 (real)

```
pipe() gave fd[0] = 3 (read end), fd[1] = 4
parent 246 wrote 20 bytes
child 247 read 20 bytes: "hello through a pipe"
child 247 read again: 0 (0 = end of file)
```



DIFFERENT FROM C

But no name, no seeking

A pipe has no file name and no position: bytes come out in the order they went in, and each byte can be read only once. fork makes the child inherit both ends.

Book: Kerrisk, TLPI §44.2 (creating and using pipes)

Three rules every pipe user must know

● ● ● block.c (QEMU guest, real)

```
# write(100000) into a 64 KiB pipe,  
# nobody reads for 1 s:  
writer 87: state S,  
  waiting in pipe_write  
writer: write returned 100000  
reader got 100000 bytes
```

● ● ● eof.c (real)

```
$ ./eof  
parent read 14 bytes  
parent: read returned 0: end of file  
$ timeout 3 ./eof forget  
parent read 14 bytes  
exit 124: killed after 3 s, still  
waiting
```

● ● ● sigpipe.c ; bash (real)

```
default: child killed by signal 13  
SIGPIPE ignored: write returned -1,  
  errno 32 (Broken pipe)  
$ yes | head -2; echo ${PIPESTATUS[*]}  
141 0      # 141 = 128 + 13
```

1. Full pipe: the writer waits

A pipe holds 64 KiB. A faster writer sleeps (state S) until the reader makes room. Nobody loses data; the writer is just slowed down.

2. End of file needs ALL write ends closed

read() returns 0 only when no process holds a write end any more, including the reader itself and the shell. Forget one close, and the reader waits for ever.

3. Nobody reads: SIGPIPE

Writing into a pipe with no reader kills the writer with SIGPIPE (13), or returns EPIPE if the signal is ignored. That is how `yes | head` stops `yes`.

How the shell runs `ls / | wc -l`

```
● ● ● strace -f bash -c 'ls / | wc -l' (real, the important lines)
```

```
367 pipe2([3, 4], 0)          = 0
367 clone(...SIGCHLD...)      = 368    # child: ls
367 close(4)                  = 0
367 clone(...SIGCHLD...)      = 369    # child: wc
367 close(3)                  = 0
368 close(3)                  = 0
368 dup2(4, 1)                 = 1      # stdout -> pipe
368 close(4)                  = 0
369 dup2(3, 0)                 = 0      # stdin <- pipe
369 close(3)                  = 0
368 execve("/usr/bin/ls", ["ls", "/"], ...) = 0
369 execve("/usr/bin/wc", ["wc", "-l"], ...) = 0
367 wait4(-1, ...) = 368 ; wait4(-1, ...) = 369
```

	fd 0	fd 1	fd 3	fd 4
ls (368)	terminal	pipe (write)	—	—
wc (369)	pipe (read)	terminal	—	—
bash (367)	terminal	terminal	—	—

```
● ● ● sleep 30 | sleep 30 ; ls -l /proc/PID/fd
```

```
pid 349:
l-wx----- ... 1 -> pipe:[2170]
pid 350:
lr-x----- ... 0 -> pipe:[2170]
```

dup2(old, new) makes descriptor new point to the same open file as old. After `dup2(4, 1)`, every `printf` of `ls` goes into the pipe, and `ls` never knows. Programs need no special code to be used in a pipeline.

Watch 100000 bytes squeeze through a 64 KiB pipe

A pipe: 16 pages in a ring (fs/pipe.c)

WRITE

FULL

READ

EOF

writer (child)

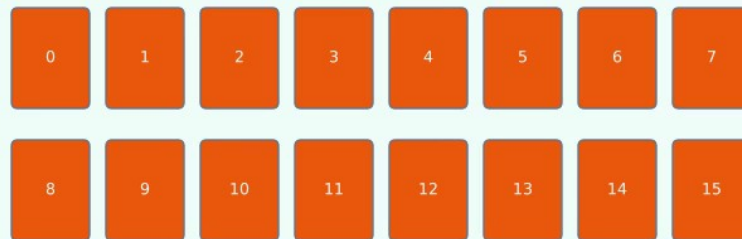
```
write(4, buf, 100000)
```

state S

sent 65,536 B

write() not finished

kernel: struct pipe_inode_info



bufs[16]: one 4 KiB page each = 64 KiB

head = 16 tail = 0 pages in use: 16

writers = 1 readers = 1

wr_wait: writer sleeps here rd_wait: reader sleeps here

reader (parent)

```
read(3, buf, 65536)
```

state S

got 0 B

sleeping 1 s

Watch for

1. **16 pages** fill up: 65536 bytes.
2. The writer **sleeps** inside one write().
3. One read frees **all 16** pages.
4. The ring **wraps**: slots 0 to 8 again.
5. **writers = 0** and empty: read returns 0.

Its write() has not returned yet: it is still busy with the same call. A writer that is faster than its reader is slowed down automatically. This is called back-pressure.

Click the video to play or pause. About 1 minute, no sound.

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.

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, ...

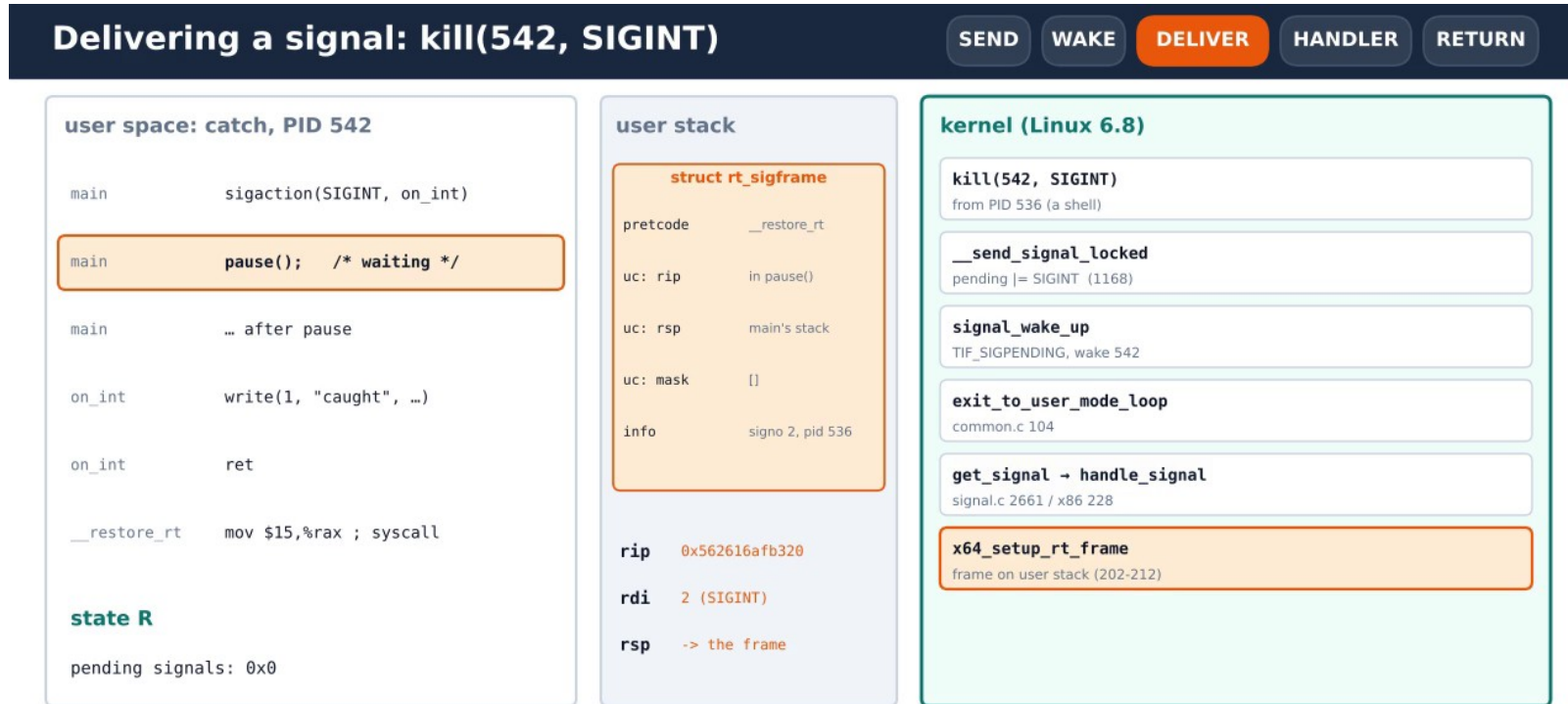
● ● ● QEMU guest, Ubuntu 6.8 (real)

```
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

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.

Key ideas from Class 8

- 1 A system call is a request to the kernel through the syscall instruction: number in rax, arguments in rdi, rsi, rdx ...
- 2 The kernel returns `-errno` on failure; the C library turns it into `-1` and sets `errno`.
- 3 `pipe()` gives two file descriptors: one to write, one to read. A full pipe blocks the writer; an empty one blocks the reader.
- 4 The shell runs `ls | wc -l` with `pipe`, `fork`, `dup2` and `exec`.
- 5 A signal is a number the kernel delivers to a process; `sigaction` installs a handler. `SIGKILL` cannot be caught.

Homework and practice

Homework 8 (bring it to the next class)

1. Which of these enter the kernel: printf, strlen, write, malloc, open? For each one, say yes or no, and why.
2. When does read() on a pipe return 0? Why must the shell close the pipe ends it does not use?
3. What happens when you press Ctrl-C while a program runs? Name the signal and its default action.

Try it in the browser

strace line → registers

[labs/lab09.html](#)

On the course website: open the lab page and choose "Try it here".

Next class: Memory and Files