Process Management III

Science

Computer Science

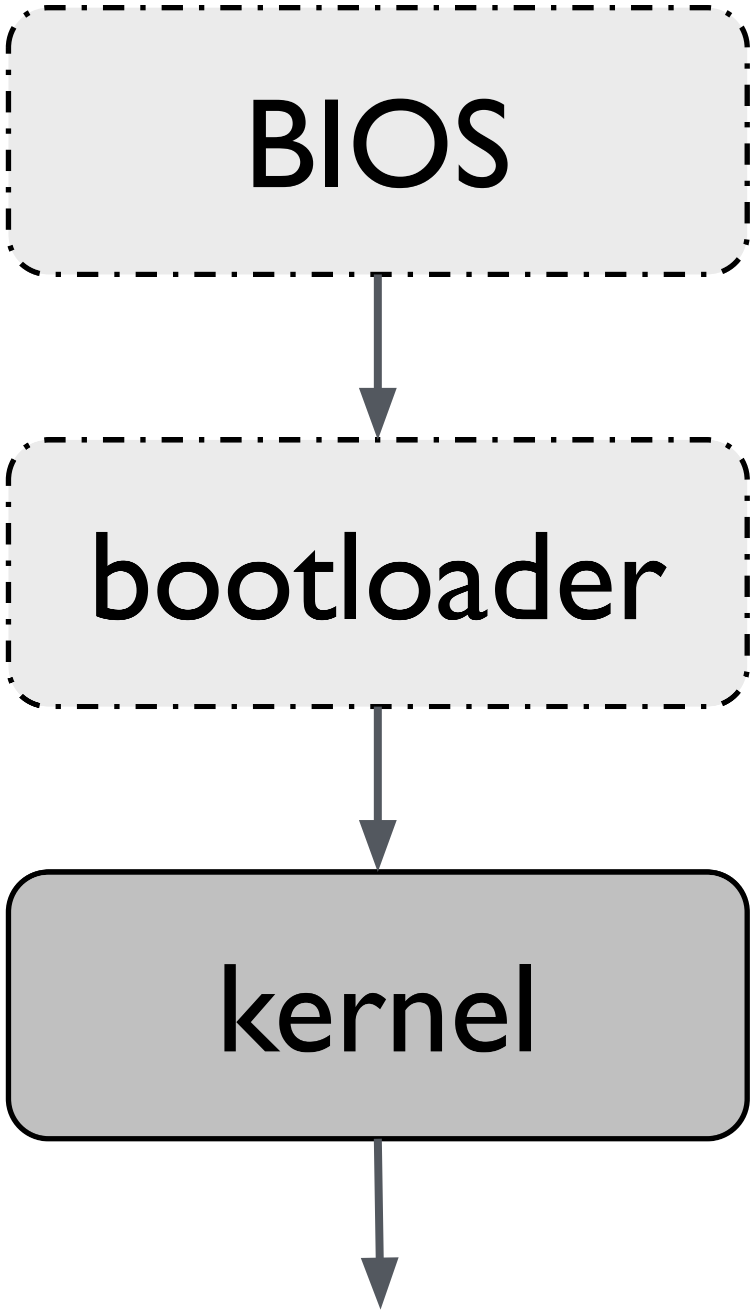
CS 351: Systems Programming

Melanie Cornelius

Slides and course content obtained with permission
from Prof. Michael Lee, <lee@iit.edu>

IIT College of Science
ILLINOIS INSTITUTE OF TECHNOLOGY

§ The Unix Family Tree

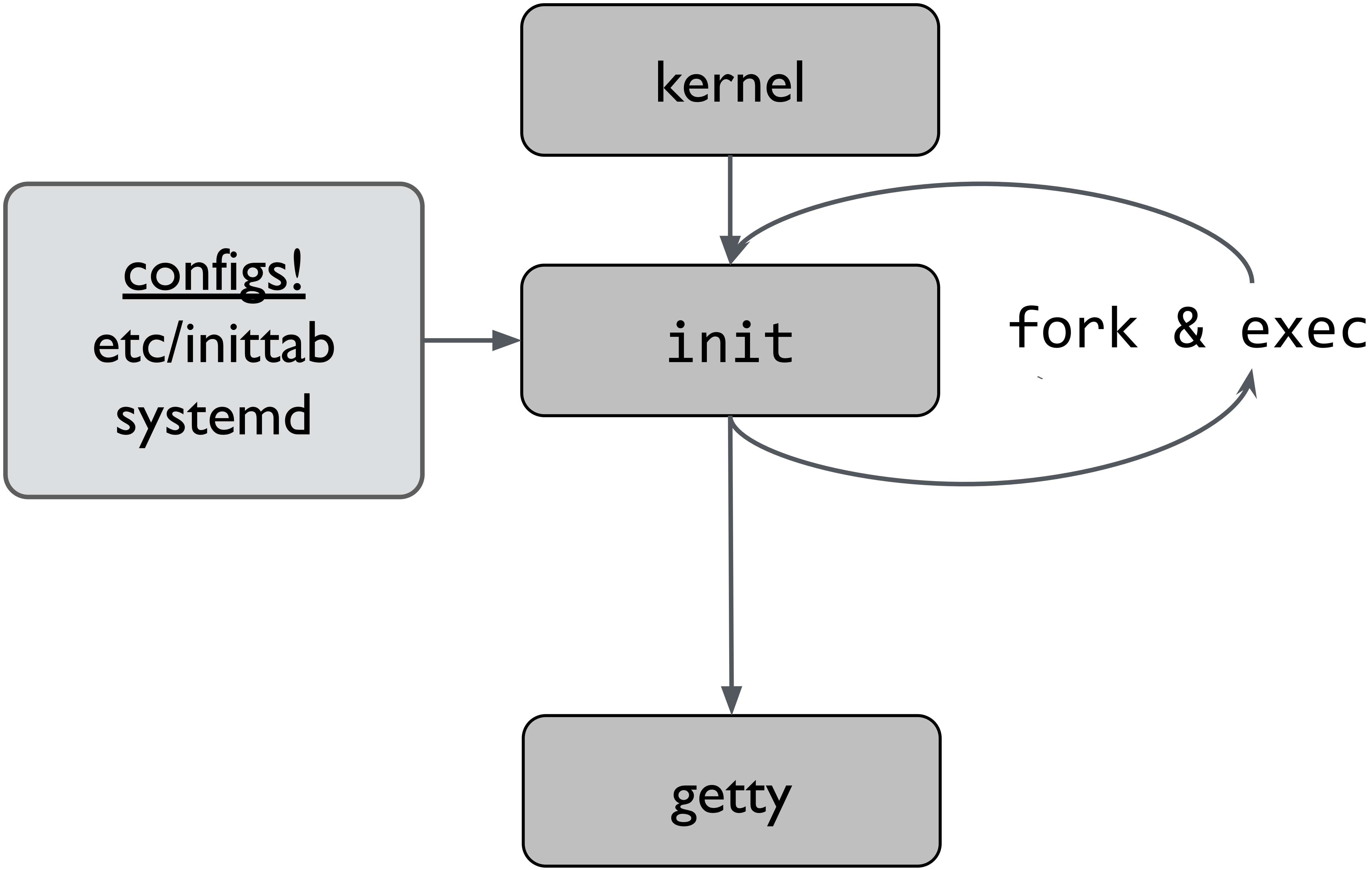


BIOS

bootloader

kernel

“handcrafted”
process -- pid = 1



kernel

configs!

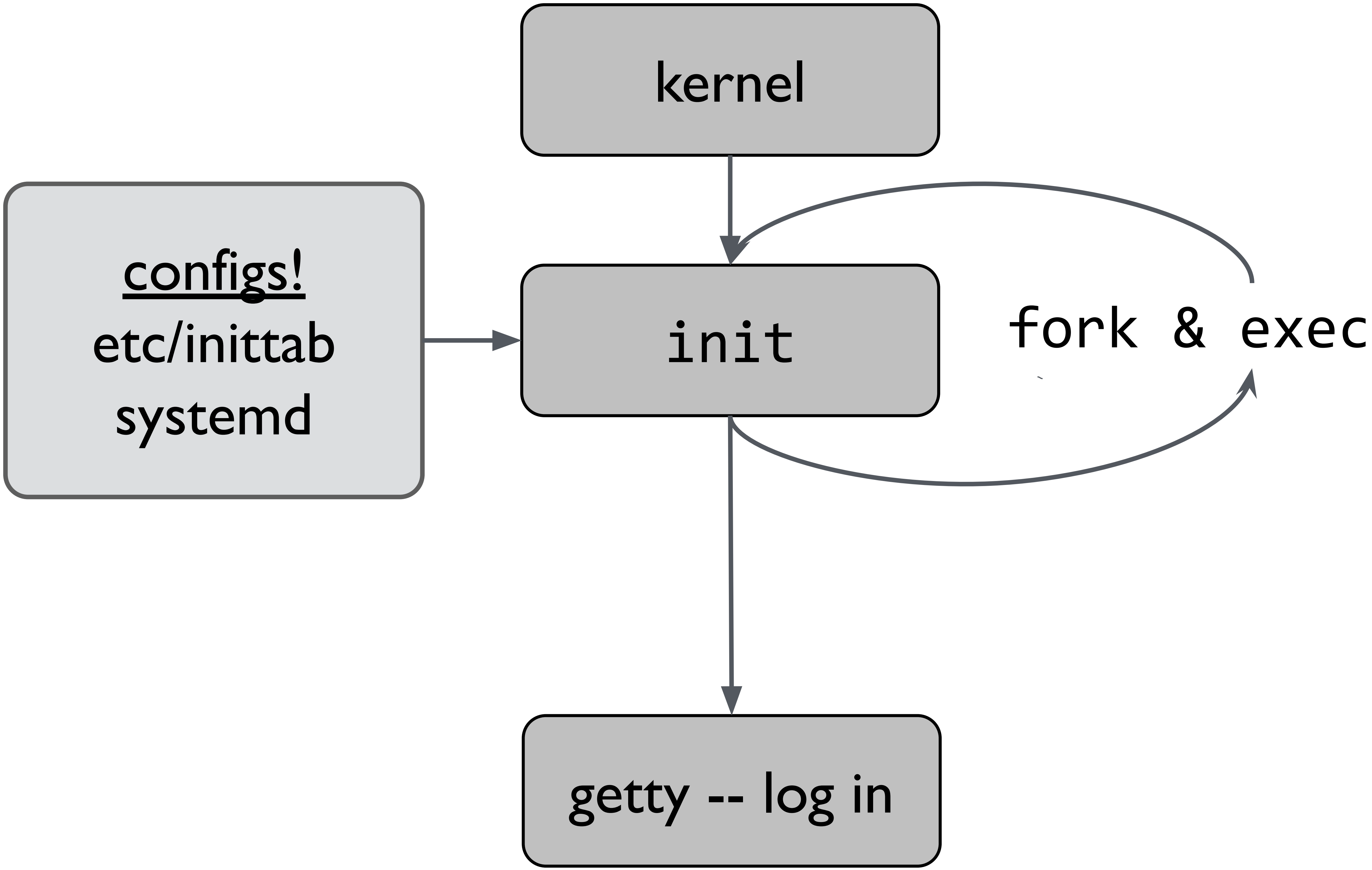
etc/inittab

systemd

init

fork & exec

getty



kernel

configs!

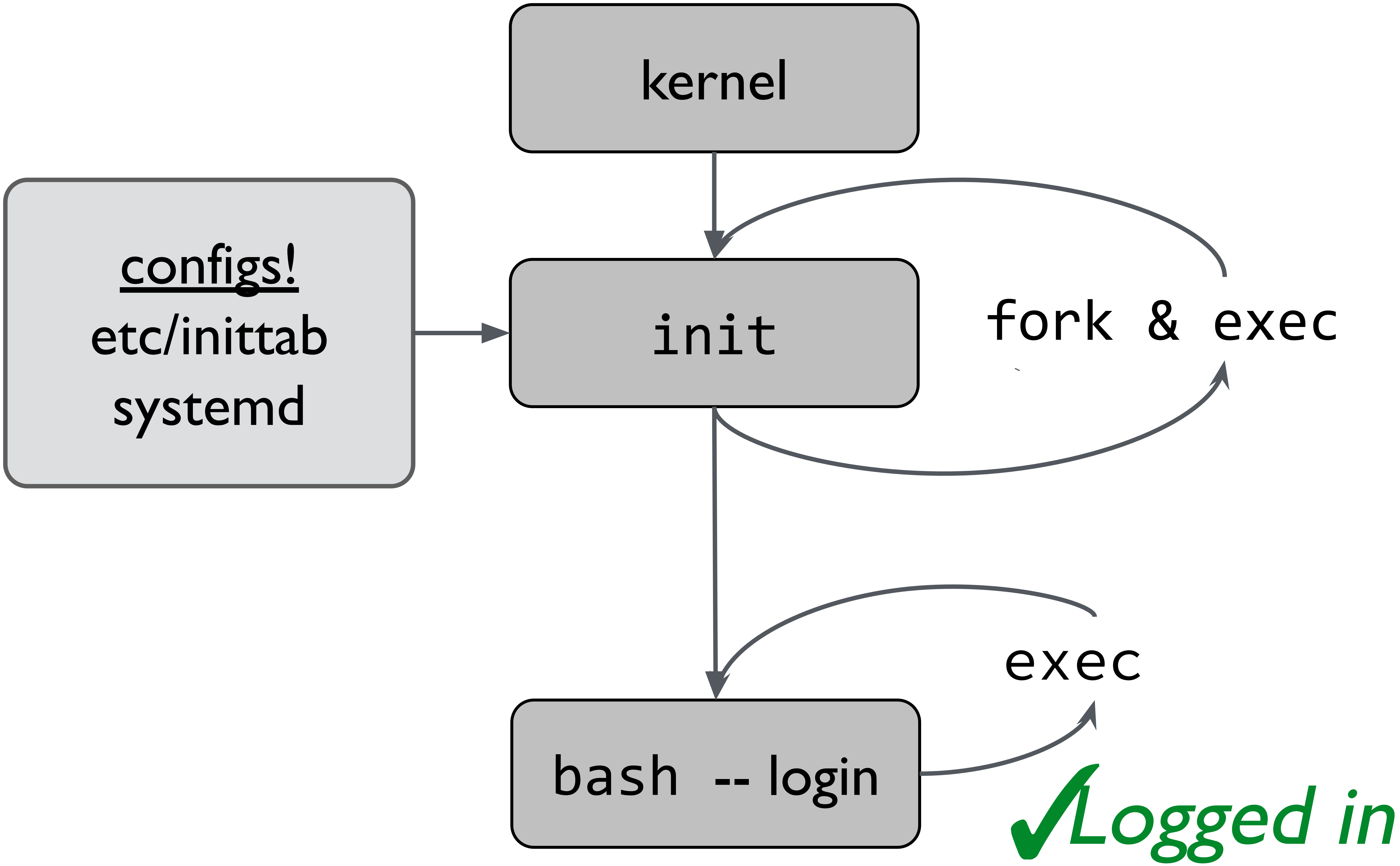
etc/inittab

systemd

init

fork & exec

getty -- log in



kernel

init

fork & exec

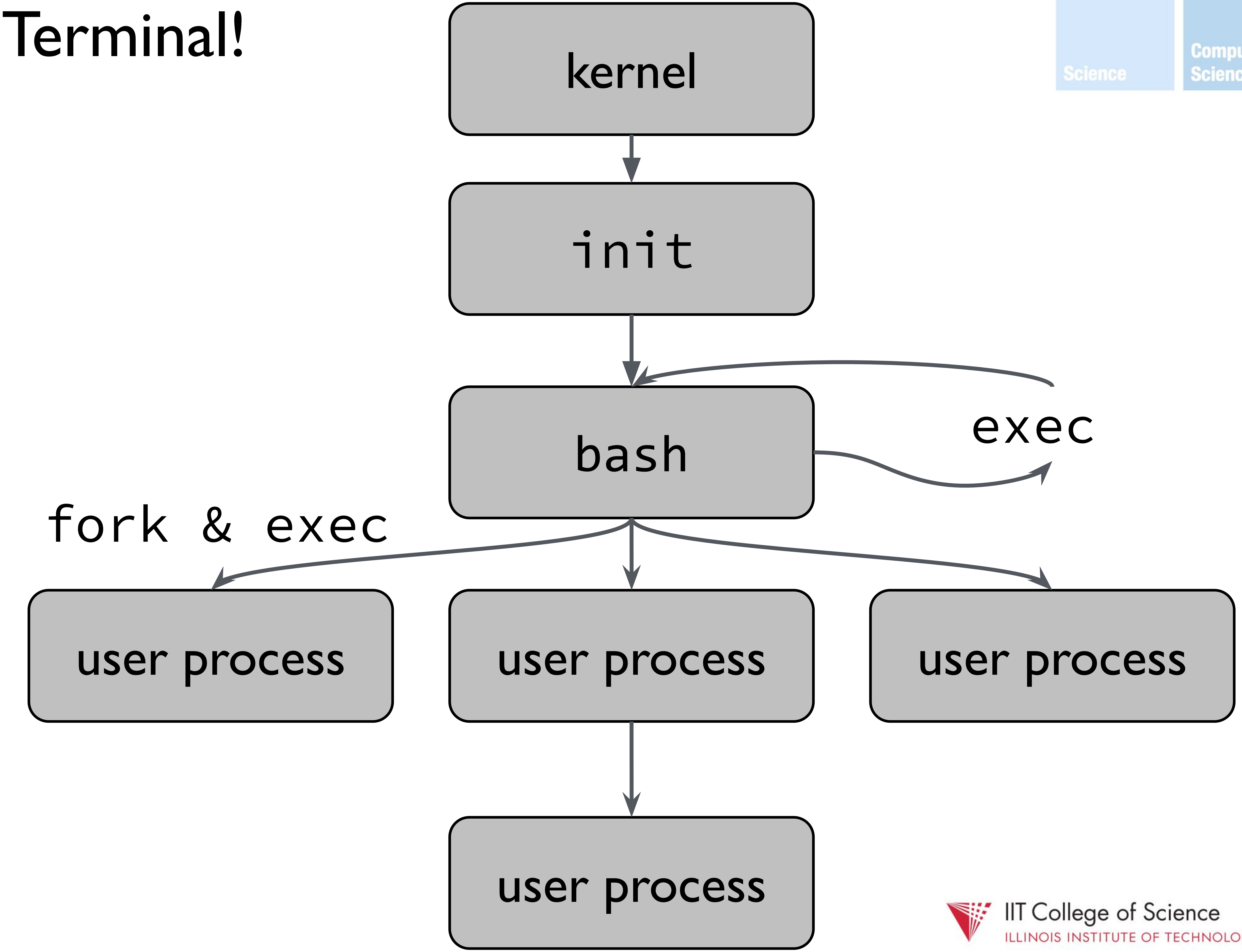
configs!
etc/inittab
systemd

exec

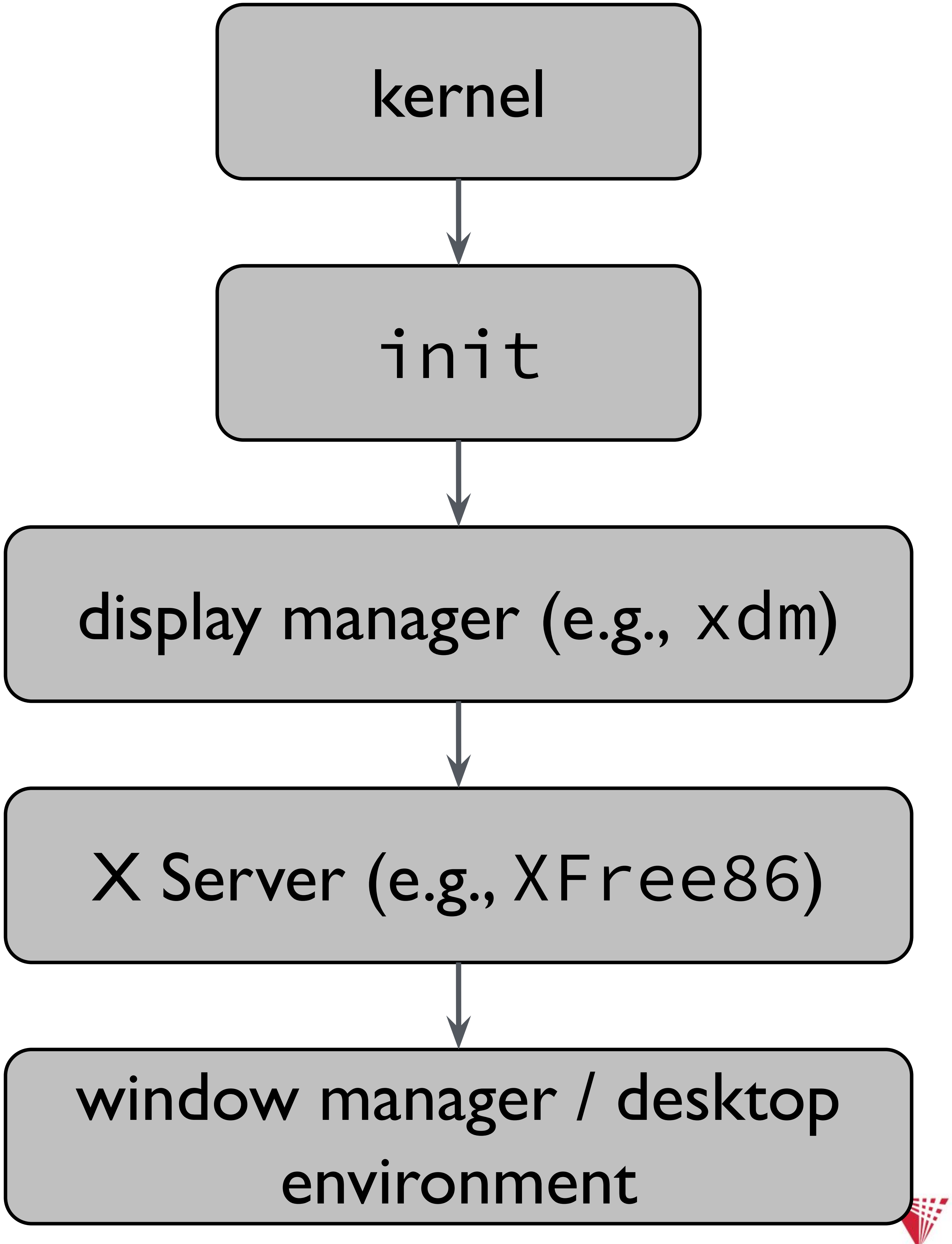
bash -- login

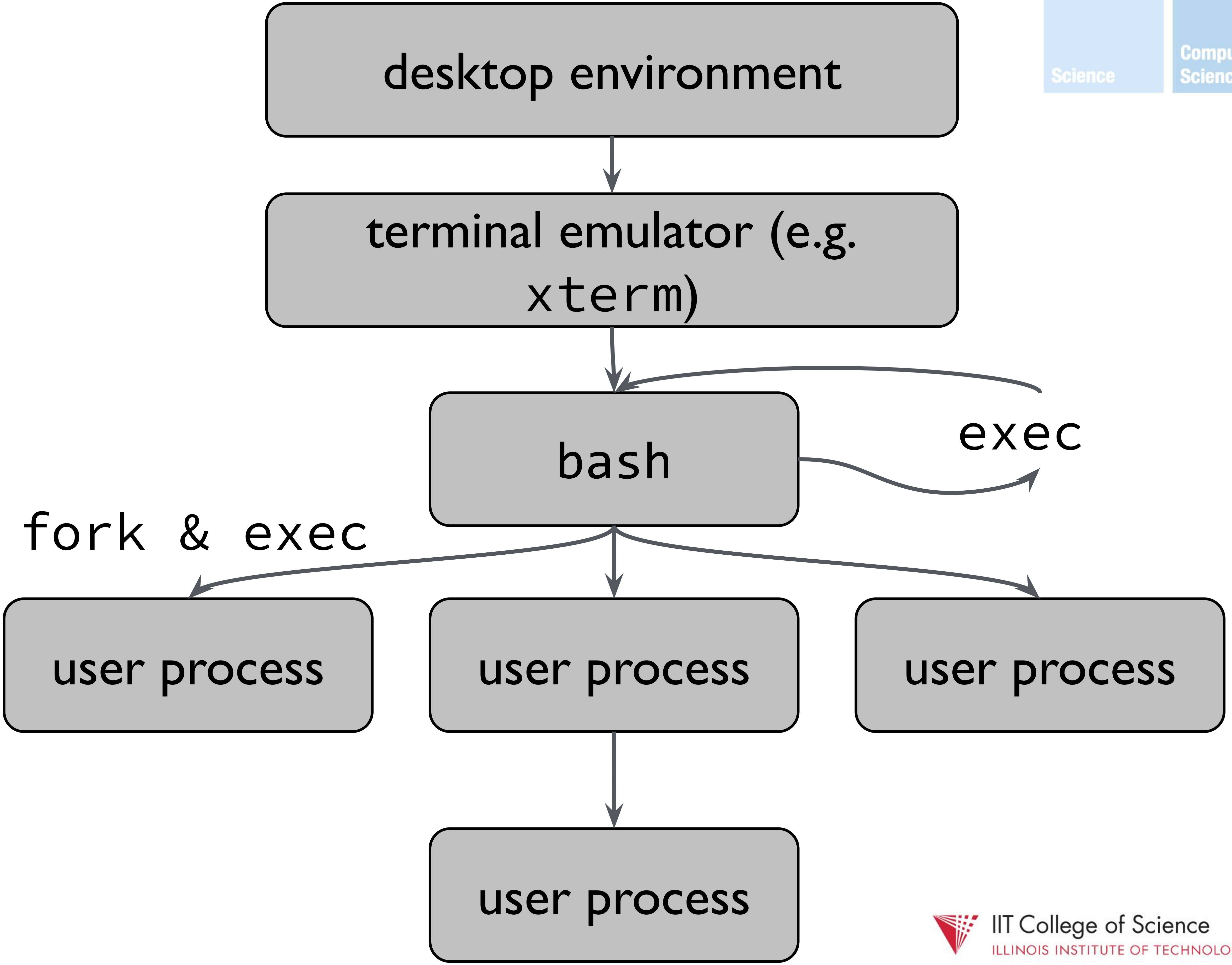
✓Logged in

Terminal!


kernel
init
bash
exec
fork & exec
user process
user process
user process
user process

GUI!


kernel
init
display manager (e.g., xdm)
X Server (e.g., XFree86)
window manager / desktop environment



desktop environment

terminal emulator (e.g. `xterm`)

`bash`

`exec`

`fork & exec`

user process

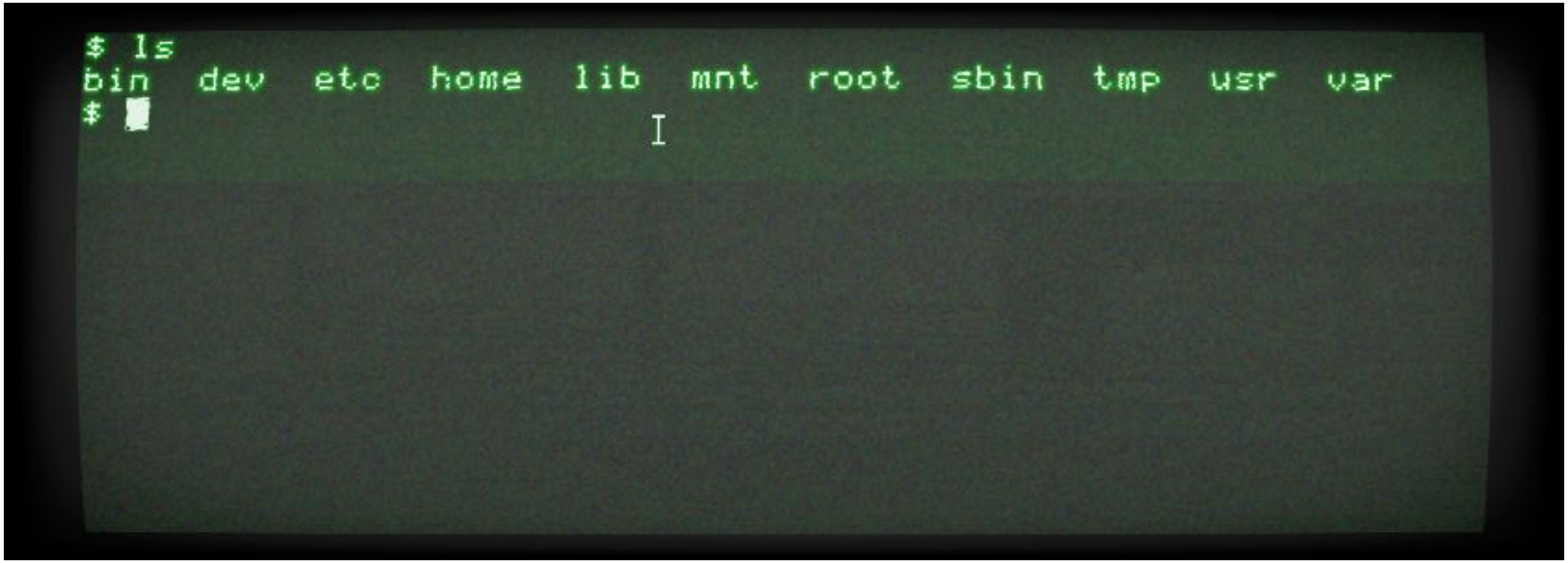
user process

user process

user process

§ The Shell (*aka* the CLI)

the original operating system user interface



```
$ ls
bin  dev  etc  home  lib  mnt  root  sbin  tmp  usr  var
$ 
```

Let the user issue requests to the operating system

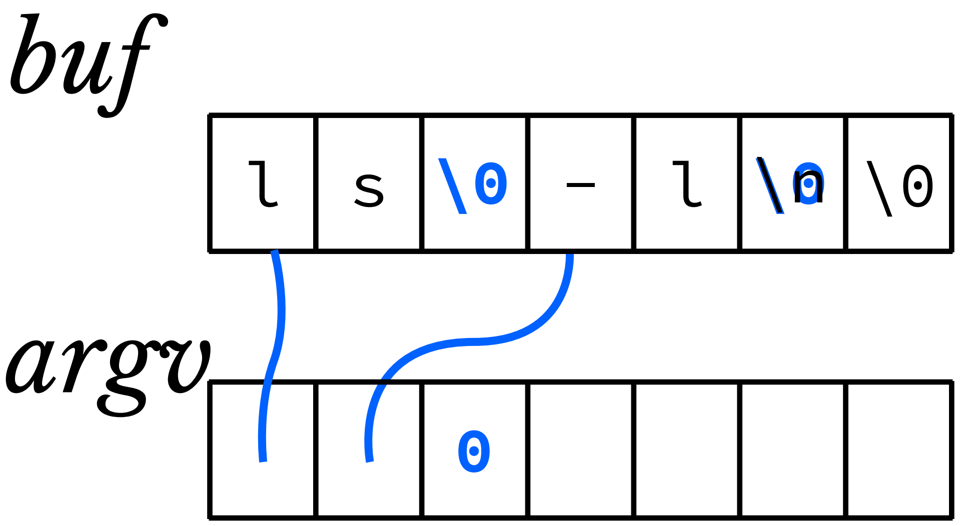
e.g.,

fork/exec a program,

manage processes (list/stop/term),

browse/manipulate the file system

```
pid_t pid;
char buf[80], *argv[10];

while (1) {
    /* print prompt */
    printf("$ ");

    /* read command and build argv */
    fgets(buf, 80, stdin);
    for (i=0, argv[0] = strtok(buf, " \n");
         argv[i];
         argv[++i] = strtok(NULL, " \n"));

    /* fork and run command in child */
    if ((pid = fork()) == 0)
        if (execvp(argv[0], argv) < 0) {
            printf("Command not found\n");
            exit(0);
        }

    /* wait for completion in parent */
    waitpid(pid, NULL, 0);
}
```



… but we are *far* from done :-)

all shells provide *task management* features

i.e., to run, track and manage *multiple* processes at a time distinguish between *foreground* (fg) and *background* (bg) processes

- fg process “blocks” additional commands from being run
- can have multiple bg processes at once

Some shell conventions:

- start bg process: `prog_name` **&**
- `fg`/`bg`: move a process into fg/bg

```
fgets(buf, 80, stdin);

/* check if bg job requested */
if (buf[strlen(buf)-2] == '&') {
    bg = 1;
    buf[strlen(buf)-2] = 0;
} else
    bg = 0;


for (i=0, argv[0] = strtok(buf, " \n");
     argv[i];
     argv[++i] = strtok(NULL, " \n"));

/* fork and run command in child */
if ((pid = fork()) == 0)
    if (execvp(argv[0], argv) < 0) {
        printf("Command not found\n");
        exit(0);
    }

/* wait for completion only if bg */
if (!bg) {
    waitpid(pid, NULL, 0);
}
```

background zombies!!!

```
/* background zombie reaping? */

if (!bg) {
    /* wait for fg job completion */
    waitpid(pid, NULL, 0);
}

/* ... and reap all bg zombies at once */
while (waitpid(-1, NULL, WNOHANG) > 0) ;
```

What is wrong with the previous solution?

A. background jobs aren't reaped

B. the loop won't reap all zombie children

C. we may reap the same child twice

D. reaping may be delayed for a long time

```
/* background zombie reaping? */

if (!bg) {
    /* wait for fg job completion */
    waitpid(pid, NULL, 0);
}

/* ... and reap all bg zombies at once */
while (waitpid(-1, NULL, WNOHANG) > 0) ;
```

- inefficient & ugly

- no guarantee when reaping will occur

what we really want is a way to be *notified* when a child turns into a zombie

… so that we can run our reaping code

“notification” → exceptional control flow

§ Signals

Signals are messages delivered by the kernel to user processes

- in response to OS events (e.g., segfault)
- or at the request of other processes
- “delivered” by executing a handler function in the receiving process

aspects of signal processing:

1. *sending* a signal to a process
2. *registering* a handler for a given signal
3. *delivering* a signal (kernel mechanism)
4. *designing* a signal handler

1. *sending* a signal to a process

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

No	Name	Default Action	Description
1	SIGHUP	terminate process	terminal line hangup
2	SIGINT	terminate process	interrupt program
3	SIGQUIT	create core image	quit program
6	SIGABRT	create core image	abort program (formerly SIGIOT)
9	SIGKILL	terminate process	kill program
10	SIGBUS	create core image	bus error
11	SIGSEGV	create core image	segmentation violation
12	SIGSYS	create core image	non-existent system call invoked
13	SIGPIPE	terminate process	write on a pipe with no reader
14	SIGALRM	terminate process	real-time timer expired
17	SIGSTOP	stop process	stop (cannot be caught or ignored)
18	SIGTSTP	stop process	stop signal generated from keyboard
19	SIGCONT	discard signal	continue after stop
20	SIGCHLD	discard signal	child status has changed
30	SIGUSR1	terminate process	User defined signal 1
31	SIGUSR2	terminate process	User defined signal 2

```c
int main () {
    int stat;
    pid_t pid;
    if ((pid = fork()) == 0)
        while(1) ;
    else {
        kill(pid, SIGINT);
        wait(&stat);
        if (WIFSIGNALED(stat))
            psignal(WTERMSIG(stat),
                    "Child term due to");
    }
}
```

```
Child term due to: Interrupt
```

sometimes it’s convenient to be able to send a signal to *multiple* processes at once mechanism: *process groups*

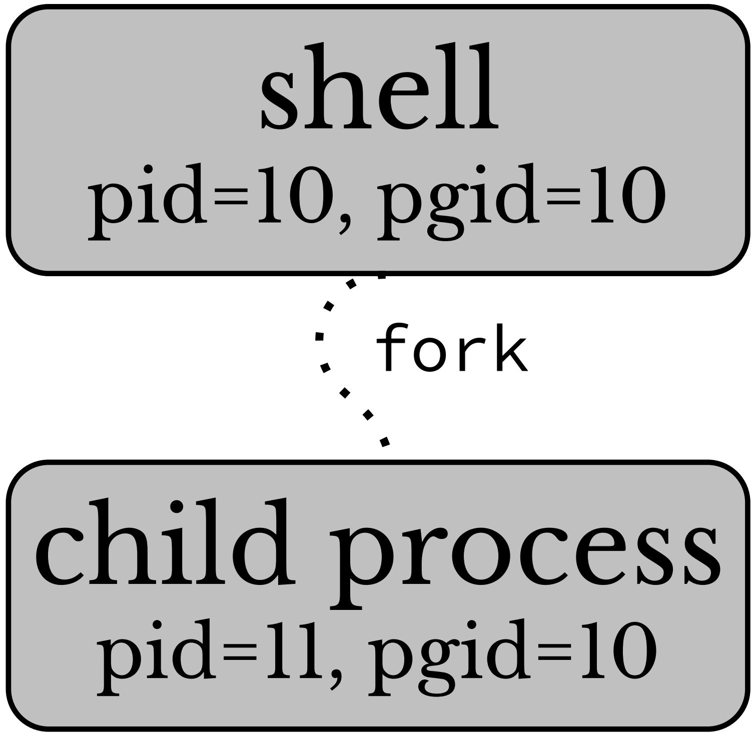
- each process belongs to a *process group*, identified by group id (PGID)
  - PGIDs are positive integers, and in a separate namespace from PIDs
  - processes inherit their parents' PGIDs

```c
/* set pid's group to given pgid */
int setpgid(pid_t pid, pid_t pgid);
```

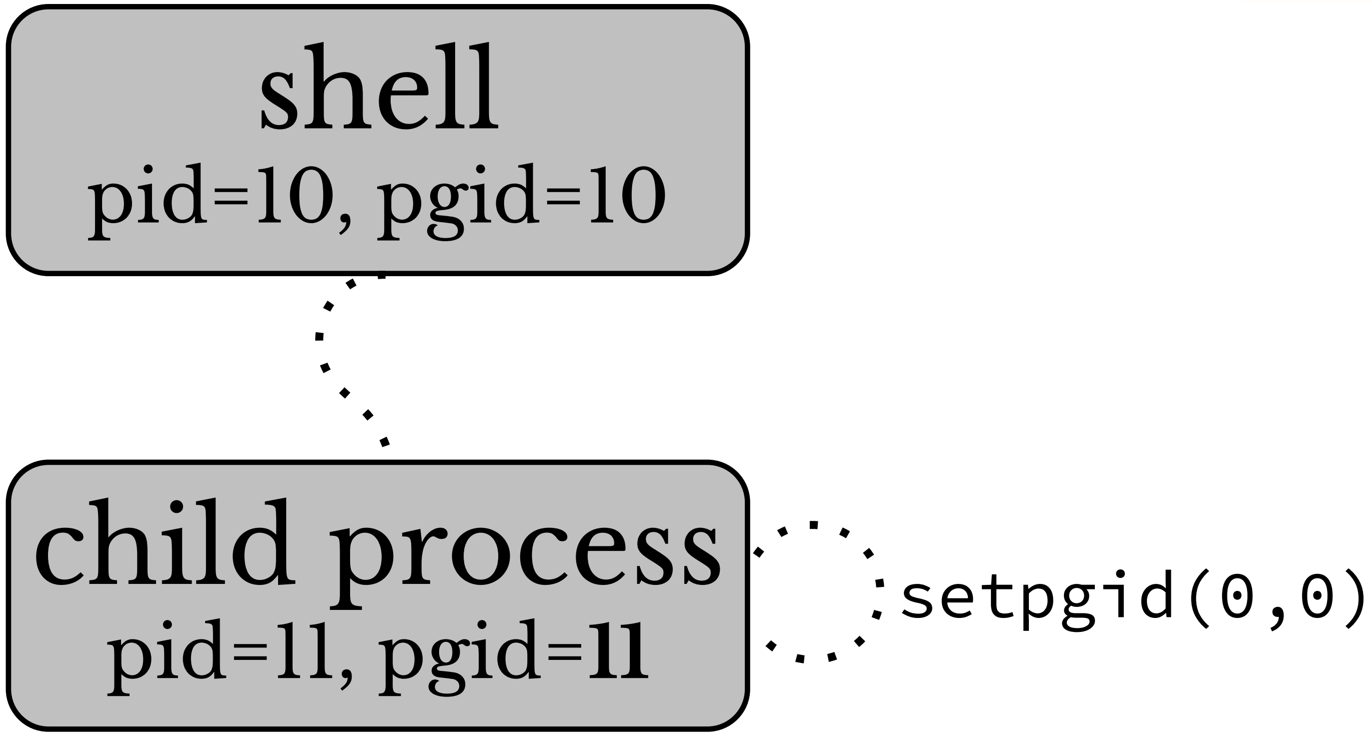
- if `pid=`0, alter the calling process
- if `pgid=`0, set the process's PGID equal to its PID

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

- if `kill` is given a *negative* `pid`, signal is sent to *all processes* with PGID=*abs*(`pid`)



shell
pid=10, pgid=10
fork
child process
pid=11, pgid=10



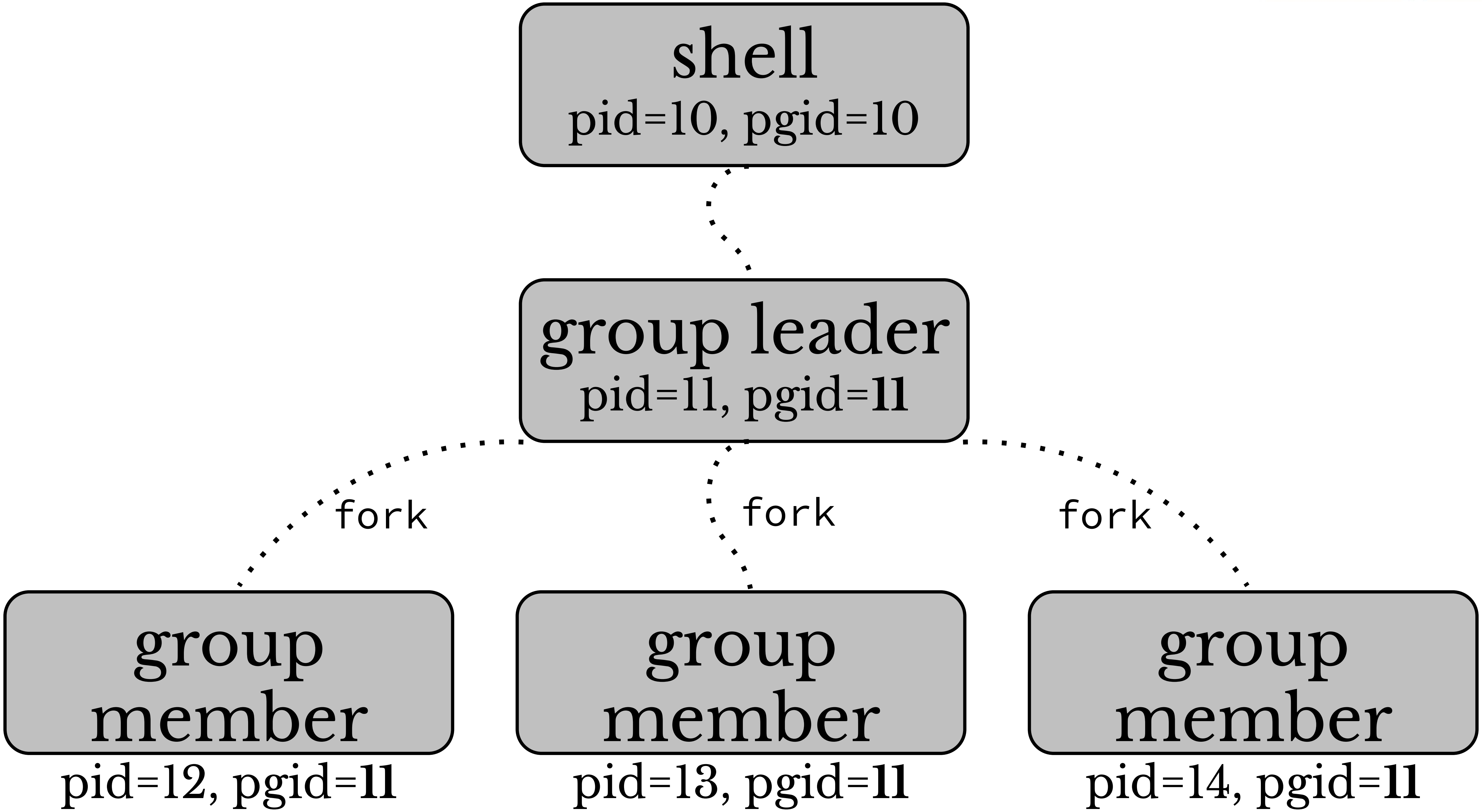
shell

pid=10, pgid=10

child process

pid=11, pgid=**11**

setpgid(0,0)



shell
pid=10, pgid=10

group leader
pid=11, pgid=**11**

fork

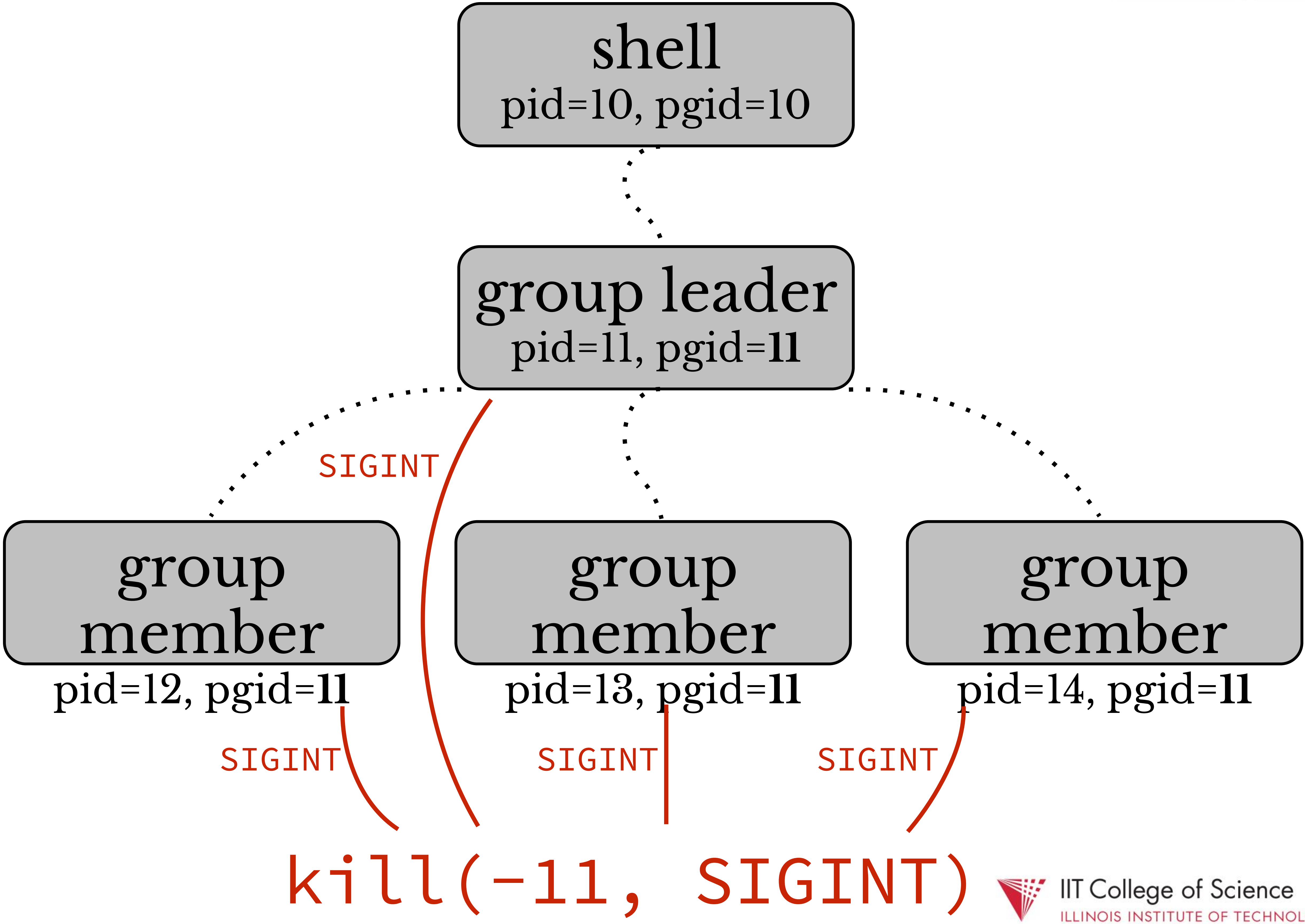
fork

fork

group member
pid=12, pgid=**11**

group member
pid=13, pgid=**11**

group member
pid=14, pgid=**11**



shell
pid=10, pgid=10

group leader
pid=11, pgid=**11**

group member
pid=12, pgid=**11**

group member
pid=13, pgid=**11**

group member
pid=14, pgid=**11**

SIGINT

SIGINT

SIGINT

SIGINT

```
kill(-11, SIGINT)
```

What is the purpose of the “kill” syscall?

A. to terminate a process

B. to send a signal

C. to receive a signal

D. to create a process group

What is a primary reason for creating a process group?

A. so parents can reap their children

B. so children can reap their parents

C. so we can send a signal to multiple processes at once

D. so signals can be sent from children to their parents

2. *registering* a handler for a given signal

```
typedef void (*sig_t) (int);
sig_t signal(int sig, sig_t func);
sig_t signal(int sig, sig_t func);
```

- `func` is typically a pointer to a signal handler function — "callback" API

- some signals *cannot* be caught! (e.g., SIGKILL)

```
sig_t signal(int sig, sig_t func);
```

- `func` can also take special values:
    - `SIG_IGN`: ignore signal
    - `SIG_DFL`: use default action

```c
void handler(int sig) {
   printf("And I still live!!!\n");
}

int main() {
    signal(SIGINT, handler);

    while(1) {
        sleep(1);
    }
    return 0;
}
```

```
^CAnd I still live!!!
^CAnd I still live!!!
^CAnd I still live!!!
^CAnd I still live!!!
^CAnd I still live!!!
```

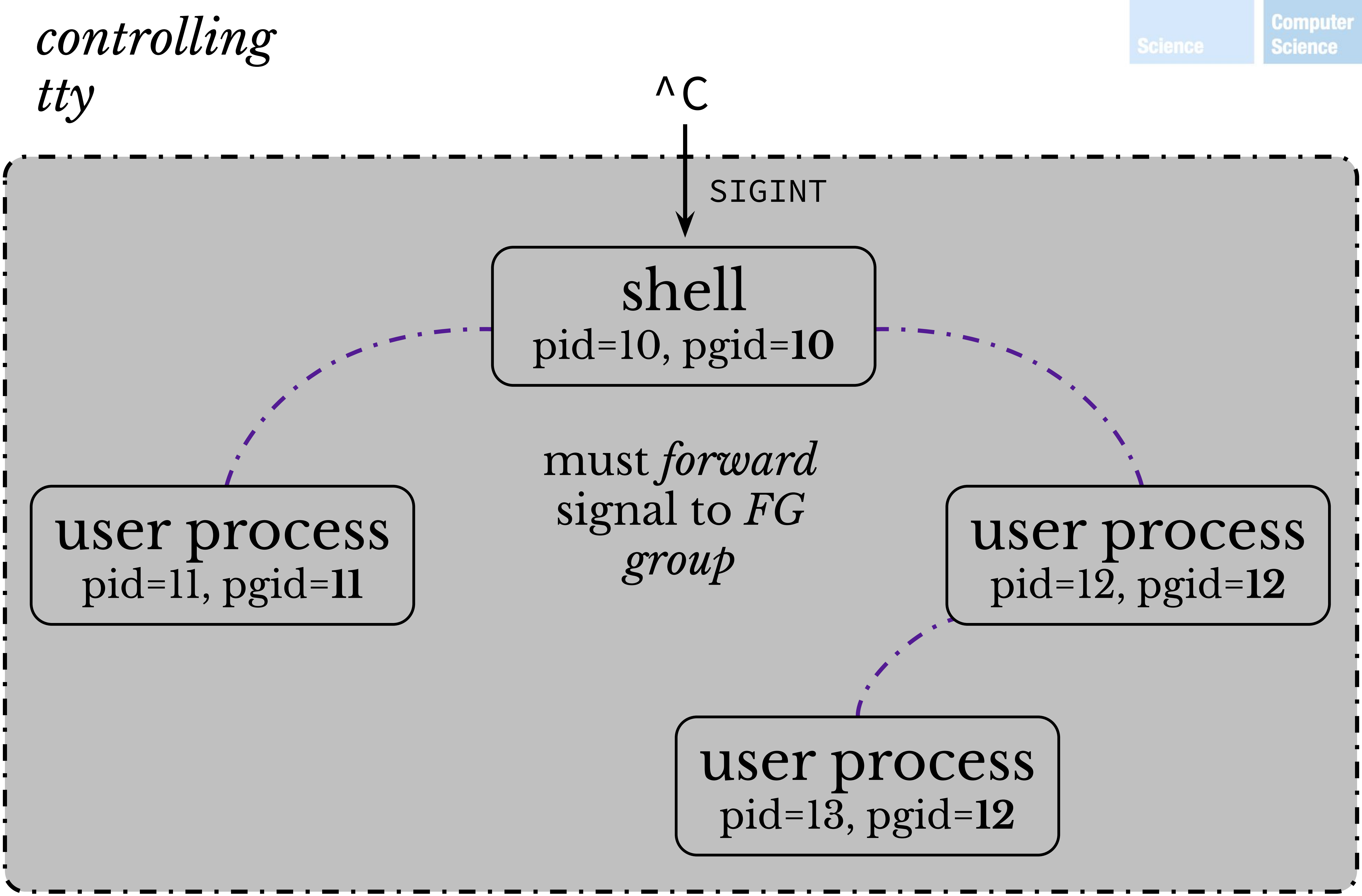
```c
int main () {
    signal(SIGINT, SIG_IGN);

    kill(getpid(), SIGINT);

    while(1) {
        sleep(1);
        printf("And I still live!!!\n");
    }
    return 0;
}
```

```
^CAnd I still live!!!
And I still live!!!
^CAnd I still live!!!
And I still live!!!
^CAnd I still live!!!
^C^C^CAnd I still live!!!
```

Q: how does ^C → SIGINT ?

A: the terminal emulator (tty device) maps keystrokes to signals, which are sent to the *session leader’s* process group

(typically, login shell)


controlling tty
^C
SIGINT
shell
pid=10, pgid=**10**
must *forward* signal to *FG group*
user process
pid=11, pgid=**11**
user process
pid=12, pgid=**12**
user process
pid=13, pgid=**12**

Child processes inherit their parent's signal handlers!

…but lose them when exec-ing a program

After the following system call is invoked, when does the kernel call `handler`?

```
    signal(SIGINT, handler);
```

A. immediately

B. when the process terminates due to ^C

C. when the SIGINT signal is delivered to the process

D. when a child process is reaped

```c
void handler (int sig) {
    printf("Signal %d received\n", sig);
    sleep(1);
}

int main () {
    signal(SIGINT, handler);
    while (1) {
        pause(); /* pauses until signal */
        printf("Back in main\n");
    }
}
```

```
^CSignal SIGINT received
Back in main
^CSignal SIGINT received
Back in main
^CSignal SIGINT received
Back in main
```

3. *delivering* a signal (kernel mechanism)

per-process kernel structures: 2 *bit vectors*

- “pending” – 1 bit per pending signal
- “blocked” – 1 bit per blocked signal

adjusting blocked signals (*signal mask*):

```
int sigprocmask(int how, /* SIG_BLOCK, SIG_UNBLOCK, or SIG_SETMASK */
                const sigset_t *set, /* specified signals  */
                sigset_t *oset);     /* gets previous mask */
```

(`SIGKILL` & `SIGTSTP` can’t be blocked!)

note: a newly forked child will inherit its parent’s `blocked` vector, but its `pending` vector will start out empty!

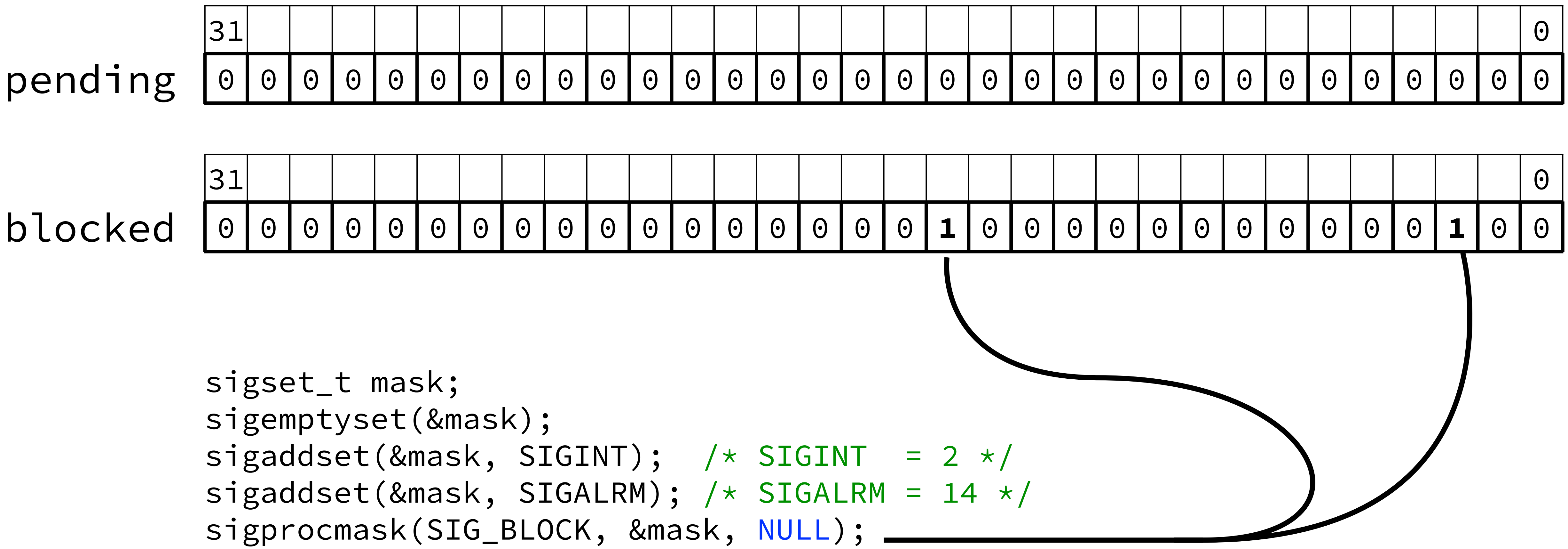
	31																															0
pending	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

	31																															0
blocked	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

	31																															0
pending	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

	31																															0
blocked	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

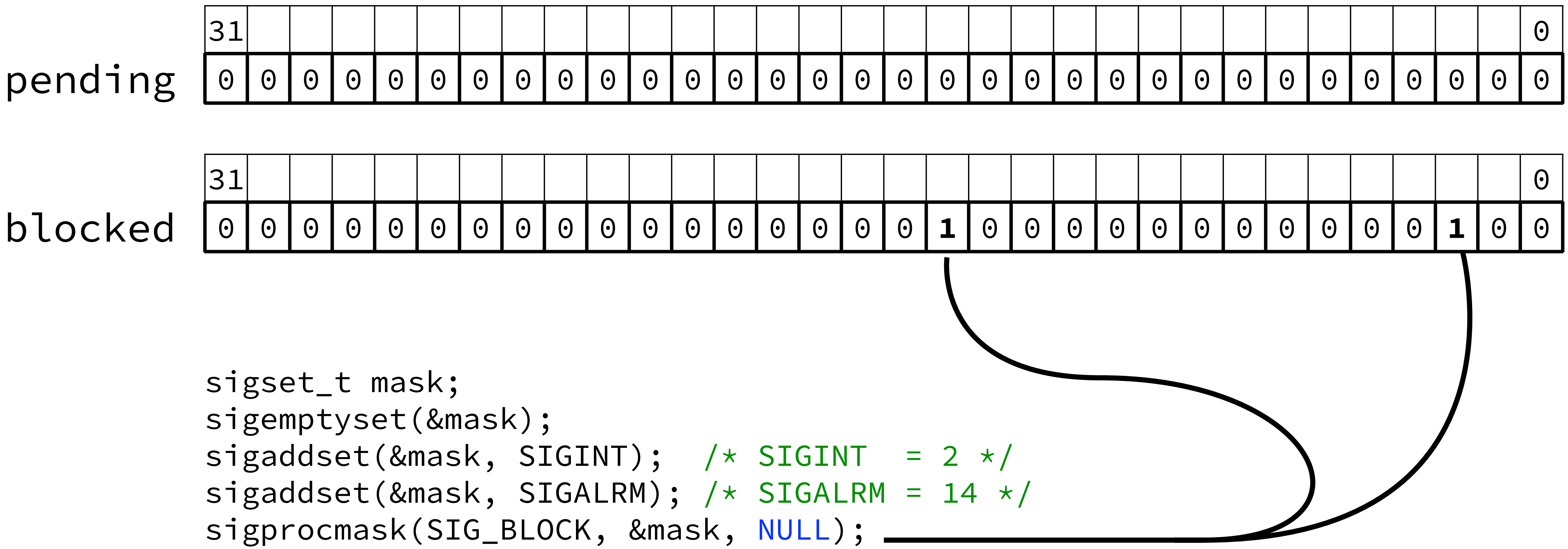
```
sigset_t mask;
sigemptyset(&mask);
sigaddset(&mask, SIGINT);  /* SIGINT  = 2 */
sigaddset(&mask, SIGALRM); /* SIGALRM = 14 */
sigprocmask(SIG_BLOCK, &mask, NULL);
```



	31																														0
pending	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
blocked	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	**1**	0	0	0	0	0	0	0	0	0	0	0	**1**	0	0

```
sigset_t mask;
sigemptyset(&mask);
sigaddset(&mask, SIGINT);  /* SIGINT  = 2 */
sigaddset(&mask, SIGALRM); /* SIGALRM = 14 */
sigprocmask(SIG_BLOCK, &mask, NULL);
```

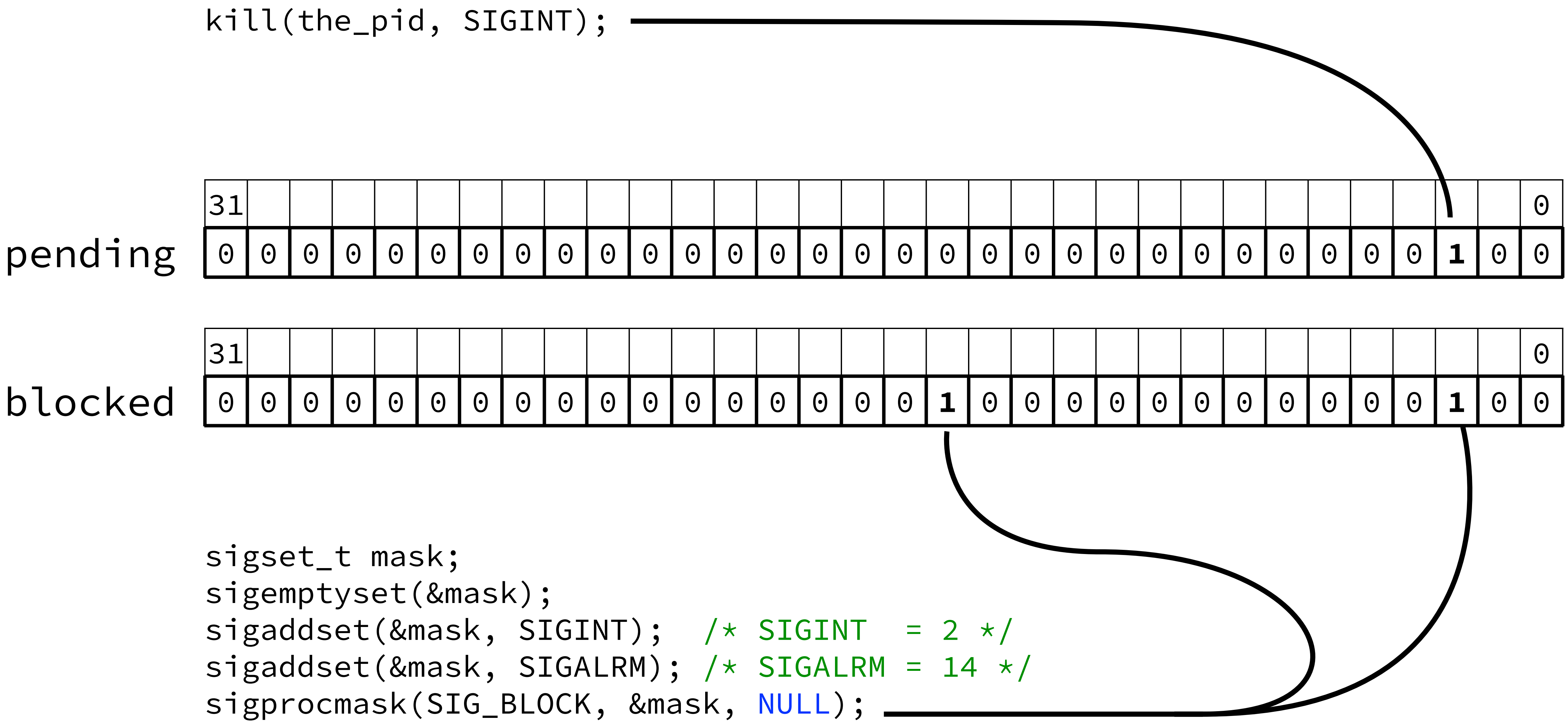
```
kill(the_pid, SIGINT);
```



	31																															0
pending	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

	31																															0
blocked	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	**1**	0	0	0	0	0	0	0	0	0	0	0	**1**	0	0

```
sigset_t mask;
sigemptyset(&mask);
sigaddset(&mask, SIGINT);  /* SIGINT  = 2 */
sigaddset(&mask, SIGALRM); /* SIGALRM = 14 */
sigprocmask(SIG_BLOCK, &mask, NULL);
```



```
kill(the_pid, SIGINT);
```

	31																															0
pending	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	**1**	0	0

	31																															0
blocked	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	**1**	0	0	0	0	0	0	0	0	0	0	0	**1**	0	0

```
sigset_t mask;
sigemptyset(&mask);
sigaddset(&mask, SIGINT);  /* SIGINT  = 2 */
sigaddset(&mask, SIGALRM); /* SIGALRM = 14 */
sigprocmask(SIG_BLOCK, &mask, NULL);
```

	31																														0
pending	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	**1**	0	0

	31																														0
blocked	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	**1**	0	0	0	0	0	0	0	0	0	0	0	**1**	0	0

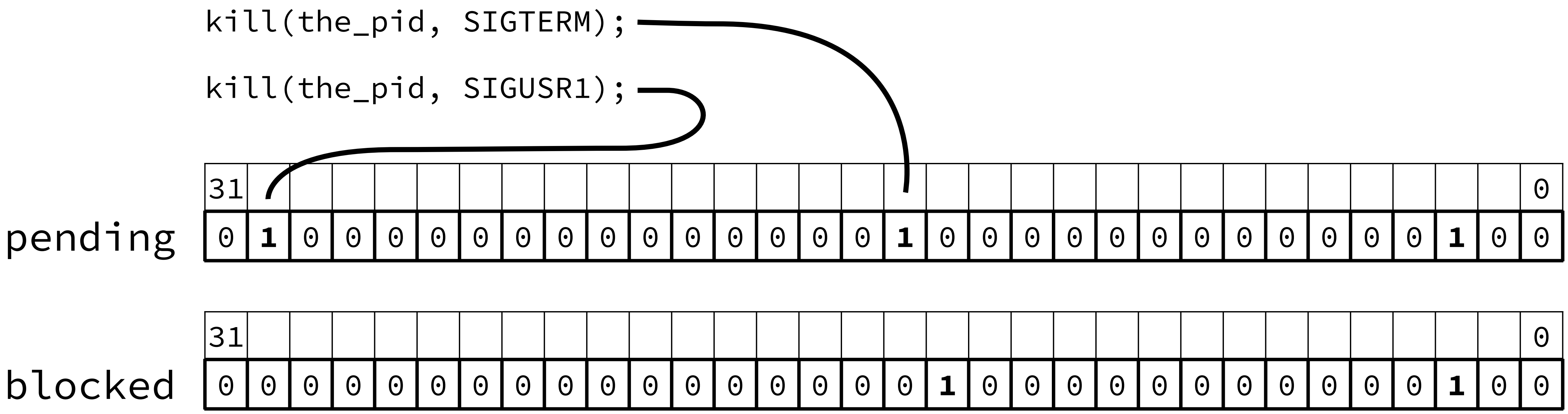
before resuming this process, kernel computes `pending & ~blocked`

	31																														0
pending	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	**1**	0	0
	31																														0
& ~blocked	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	**0**	1	1	1	1	1	1	1	1	1	1	1	**0**	1	1
	31																														0
	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

31																															0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

```
(pending & ~blocked) ⇒ 0
```

i.e., no signals to deliver — resume regular control flow



```
kill(the_pid, SIGTERM);

kill(the_pid, SIGUSR1);
```

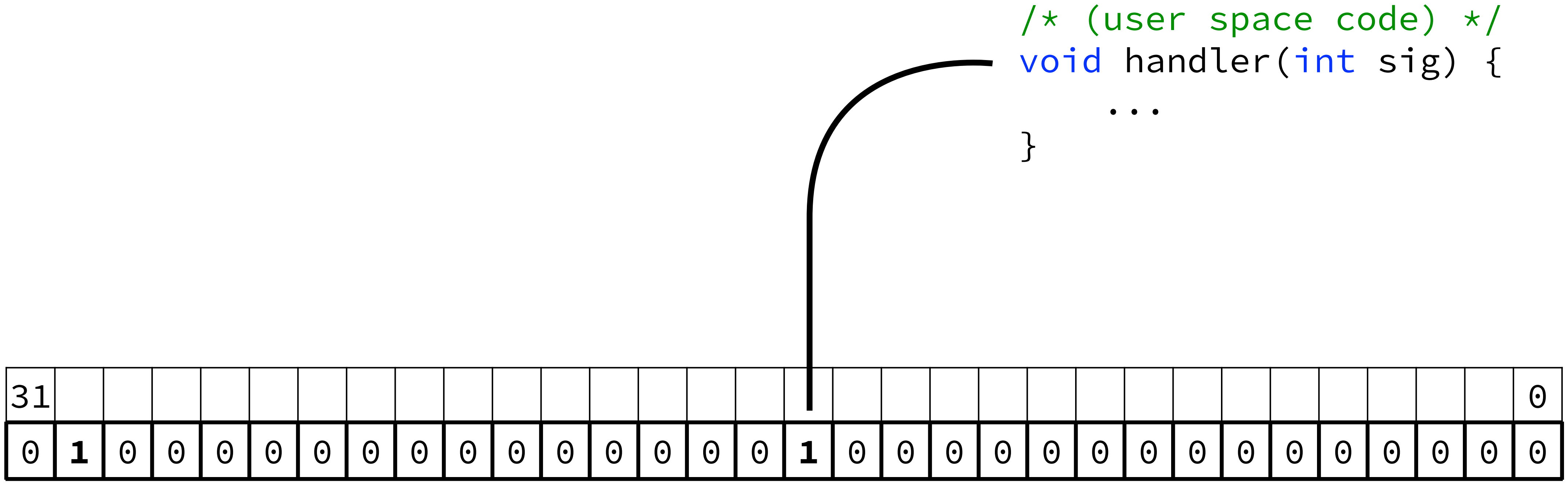
	31																															0
pending	0	**1**	0	0	0	0	0	0	0	0	0	0	0	0	0	0	**1**	0	0	0	0	0	0	0	0	0	0	0	0	**1**	0	0
blocked	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	**1**	0	0	0	0	0	0	0	0	0	0	0	**1**	0	0

	31																															0
pending	0	**1**	0	0	0	0	0	0	0	0	0	0	0	0	0	0	**1**	0	0	0	0	0	0	0	0	0	0	0	0	**1**	0	0
& ~blocked	1	**1**	1	1	1	1	1	1	1	1	1	1	1	1	1	1	**1**	**0**	1	1	1	1	1	1	1	1	1	1	1	**0**	1	1
	0	**1**	0	0	0	0	0	0	0	0	0	0	0	0	0	0	**1**	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

31																															0
0	**1**	0	0	0	0	0	0	0	0	0	0	0	0	0	0	**1**	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

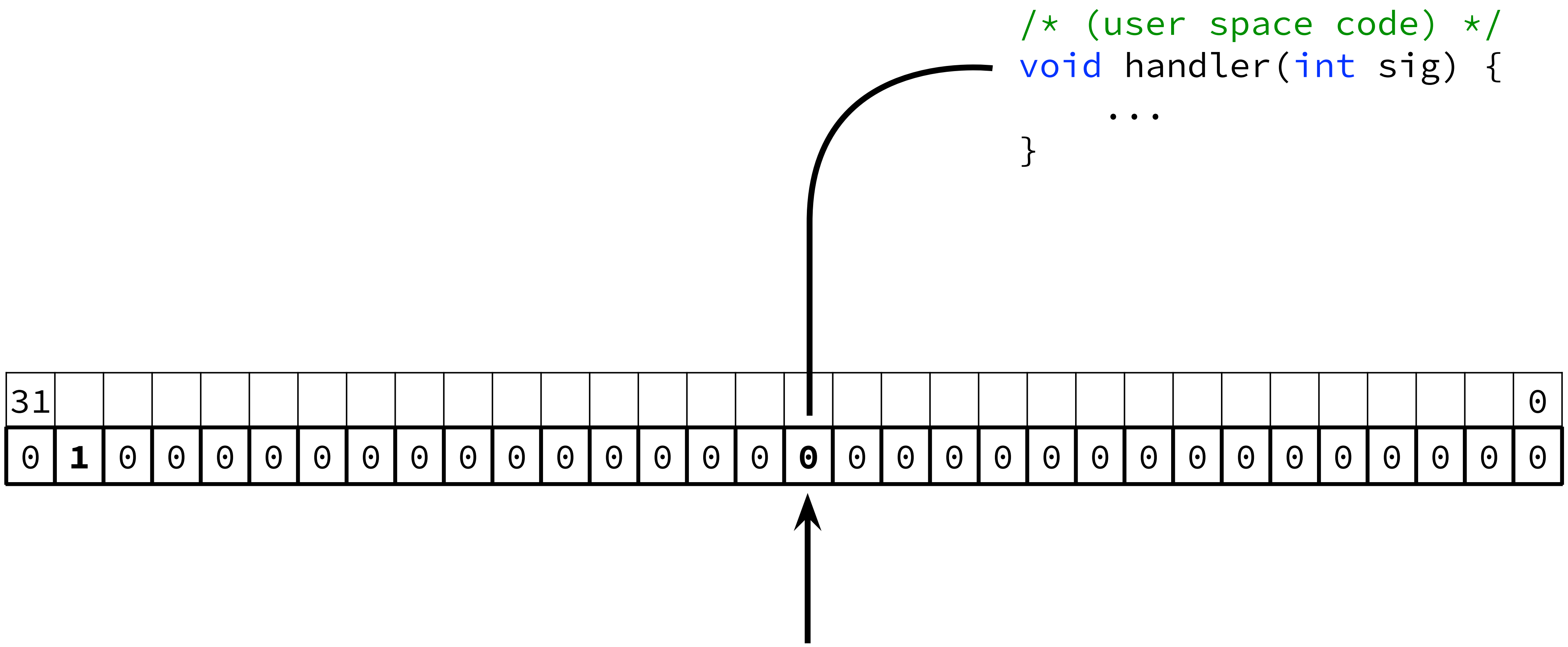
← - - - - - - - - - - - - - - - deliver signals in order

(i.e., ignore, terminate, or run handler)



```
/* (user space code) */
void handler(int sig) {
    ...
}
```

31																															0
0	**1**	0	0	0	0	0	0	0	0	0	0	0	0	0	0	**1**	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

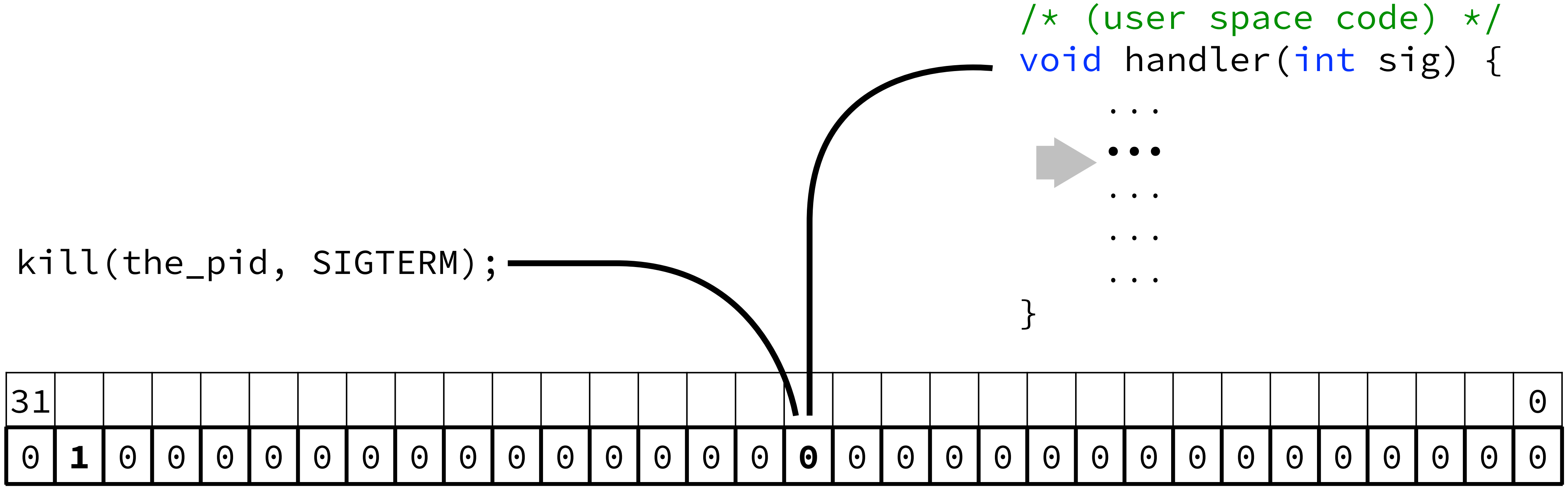


```
/* (user space code) */
void handler(int sig) {
    ...
}
```

31																															0
0	**1**	0	0	0	0	0	0	0	0	0	0	0	0	0	0	**0**	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

mark signal as "delivered"

(and block this signal until the handler returns)

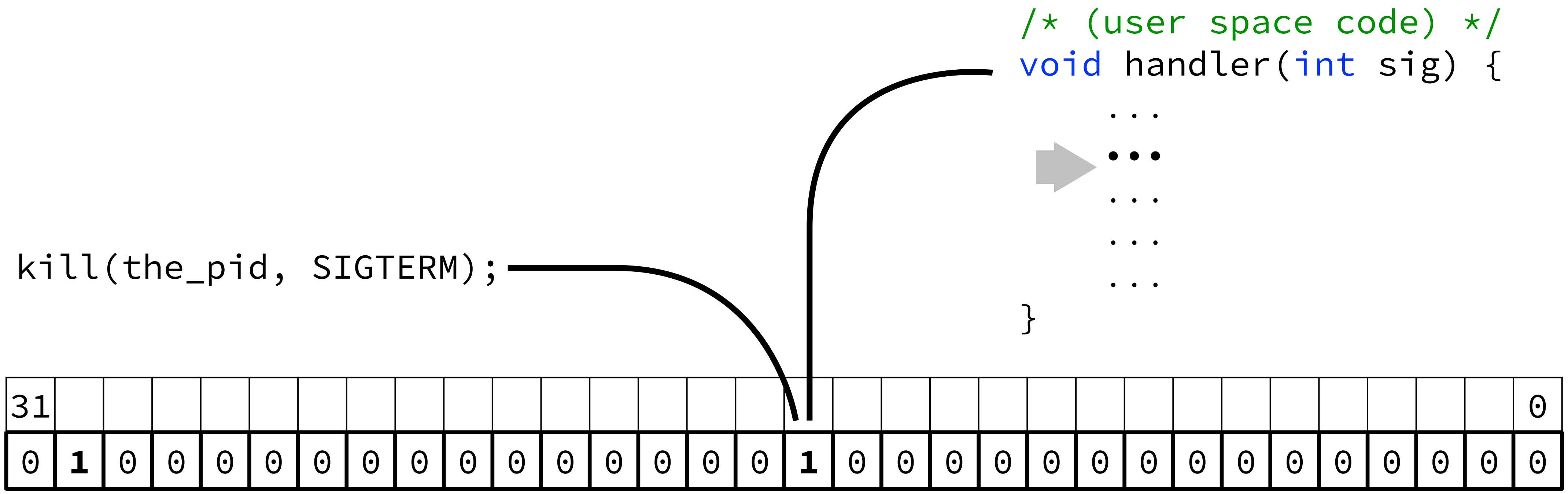


```
/* (user space code) */
void handler(int sig) {
    ...
    ...
    ...
    ...
    ...
}
```

```
kill(the_pid, SIGTERM);
```

31																															0
0	**1**	0	0	0	0	0	0	0	0	0	0	0	0	0	0	**0**	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Q: what happens if a signal is received as its handler is running?

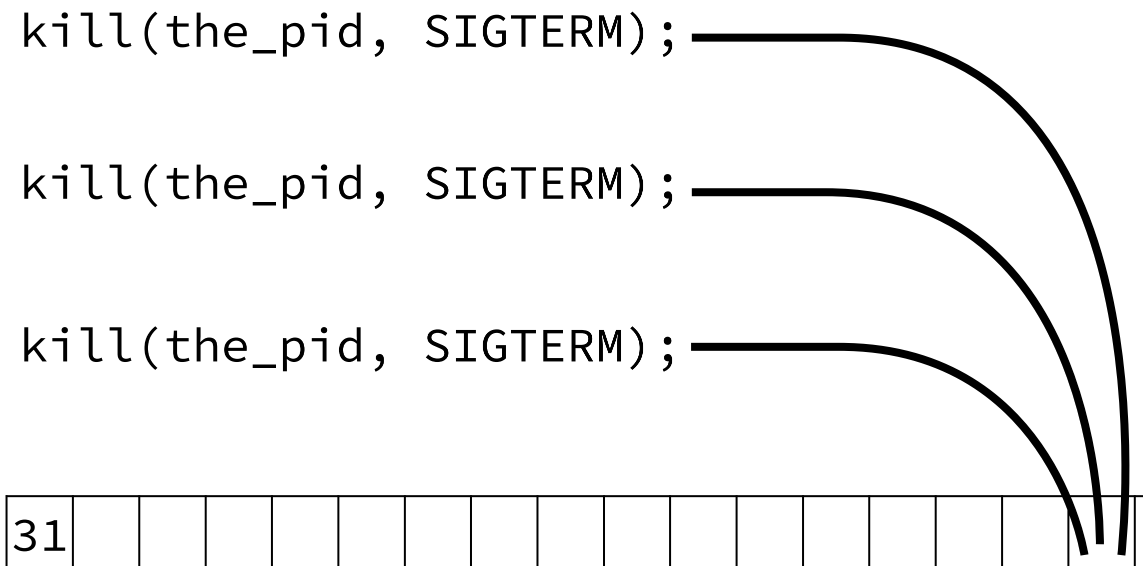


```
/* (user space code) */
void handler(int sig) {
    ...
    ...
    ...
    ...
    ...
}
```

```
kill(the_pid, SIGTERM);
```

31																															0
0	**1**	0	0	0	0	0	0	0	0	0	0	0	0	0	0	**1**	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

A: mark it as pending, but don’t run the handler again! (signal currently blocked)

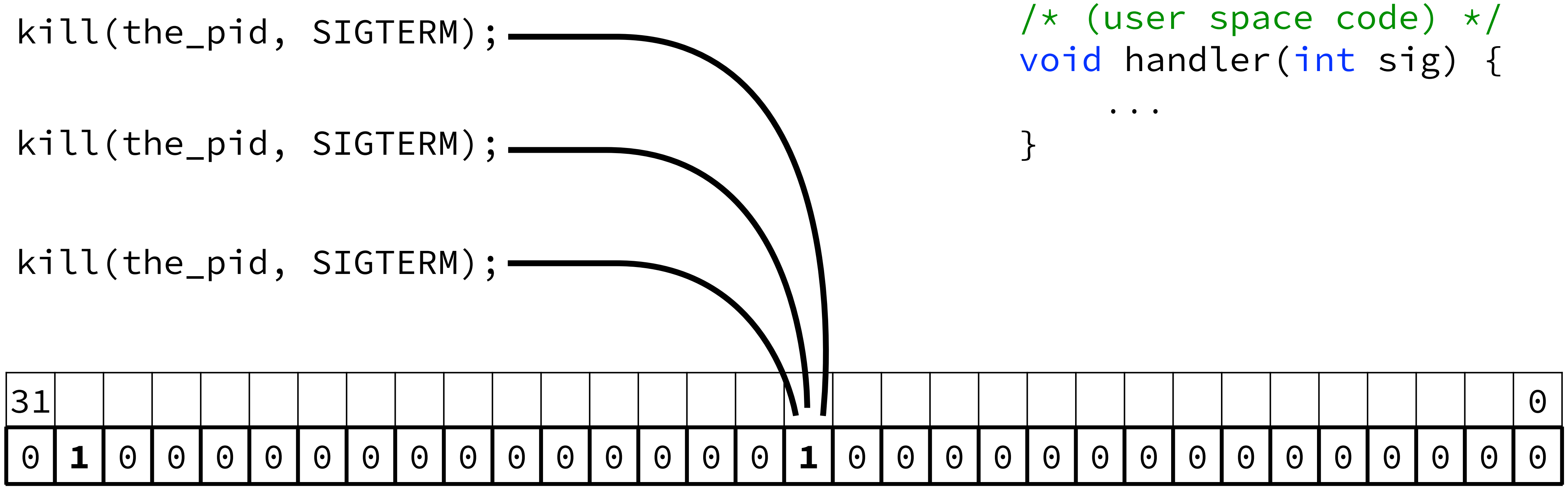


```
kill(the_pid, SIGTERM);

kill(the_pid, SIGTERM);

kill(the_pid, SIGTERM);
```

```
/* (user space code) */
void handler(int sig) {
    ...
}
```

31																															0
0	**1**	0	0	0	0	0	0	0	0	0	0	0	0	0	0	**1**	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Q: what happens if a signal is sent many times before its handler is run?



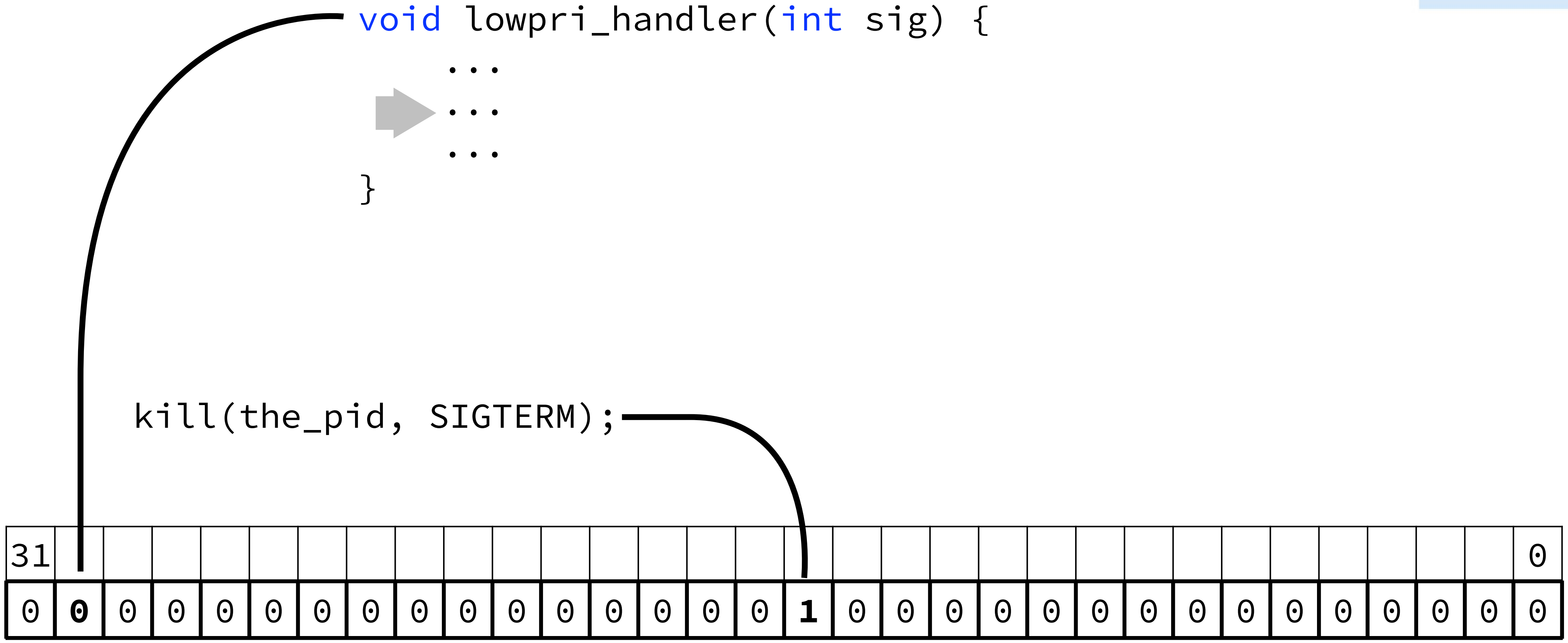
```
kill(the_pid, SIGTERM);

kill(the_pid, SIGTERM);

kill(the_pid, SIGTERM);
```

```
/* (user space code) */
void handler(int sig) {
    ...
}
```

31																															0
0	**1**	0	0	0	0	0	0	0	0	0	0	0	0	0	0	**1**	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Q: …what *can* we do?

A: nothing. (we *can't queue* signals!)

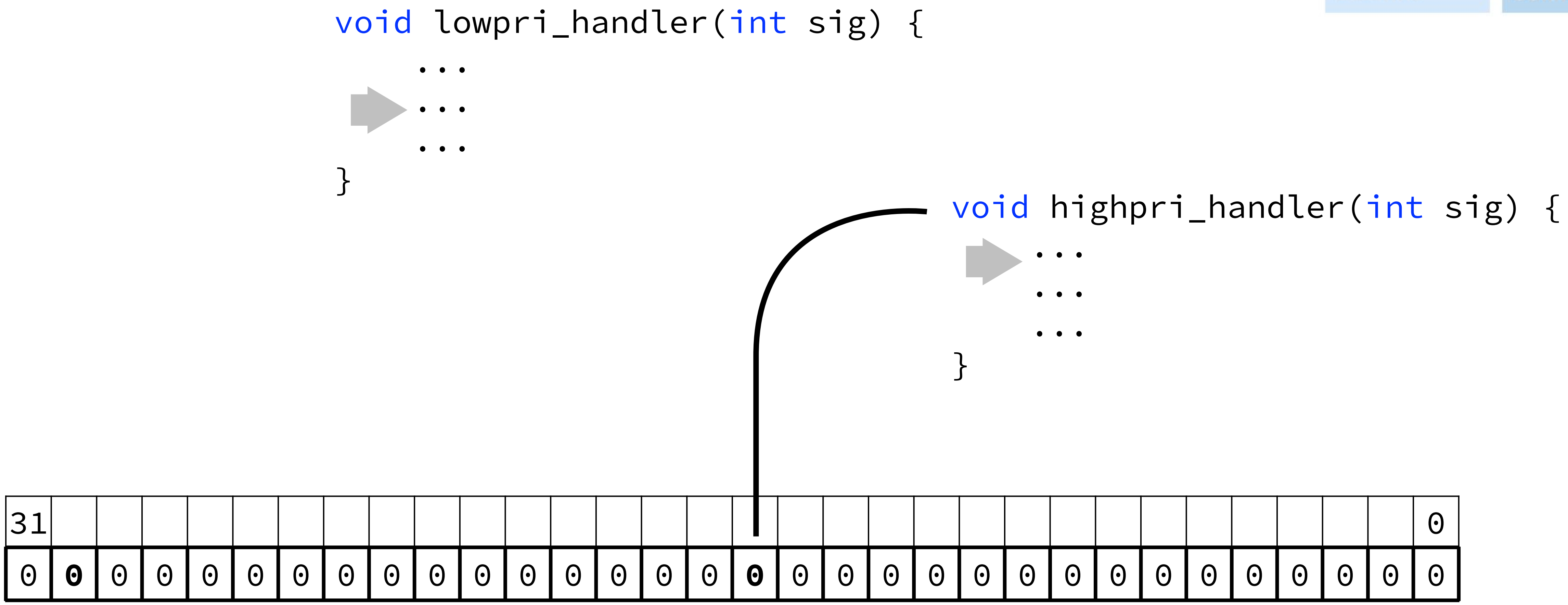


```
void lowpri_handler(int sig) {
        ...
        ...
        ...
}
```

```
kill(the_pid, SIGTERM);
```

31																															0
0	**0**	0	0	0	0	0	0	0	0	0	0	0	0	0	0	**1**	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Q: what happens if a signal is received as a handler for a lower priority one is already running?



```
void lowpri_handler(int sig) {
    ...
    ...
    ...
}
```

```
void highpri_handler(int sig) {
    ...
    ...
    ...
}
```

31																															0
0	**0**	0	0	0	0	0	0	0	0	0	0	0	0	0	0	**0**	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

A: we *preempt* the lower priority handler (and resume it — if possible — later)

Consider a scenario where a single handler function has been registered for two distinct signals. Given that the higher priority signal has just been delivered and the handler is currently executing, what happens if the lower priority signal arrives?

A. the higher priority handler is preempted and the lower priority one is run

B. the signal is marked as pending but is not delivered

C. the lower priority handler is started, then preempted to return to the higher priority one

D. the signal is not marked as pending, as all signals are blocked while the higher priority handler is being run

4. *designing* a signal handler

Q: what can go wrong?

```c
struct foo { int x, y, z; } f;

int main () {
    int i = 1;

    f = (struct foo){ 0, 0, 0 };

    signal(SIGALRM, tick);

    alarm(1); /* send SIGALRM in 1s */


    while(1) {
        f = (struct foo){ i, i, i };
        i = (i + 1) % 100;
    }
}

void tick(int s) {
    printf("%d %d %d\n", f.x, f.y, f.z);
    alarm(1); /* send SIGALRM in 1s */
}
```

```
80 80 80
77 77 77
24 24 24
19 19 19
64 64 64
1 1 0
94 94 94
44 44 44
97 97 97
70 70 70
18 18 18
5 5 5
91 91 91
9 9 9
81 81 80
4 4 4
78 78 78
74 74 74
0 0 0
32 32 32
55 55 55
71 71 71
7 7 7
69 69 69
3 2 2
80 80 80
```

```c
int main () {
    int i;
    signal(SIGUSR1, handler);
    signal(SIGUSR2, handler);
    for (i=0; i<10; i++) {
        if (fork() == 0) {
            while (1) {
                kill(getppid(), SIGUSR1);
                kill(getppid(), SIGUSR2);
            }
        }
    }
    while(1) pause();
}

void handler(int s) {
    static int x = 10, y = 20;
    int tmp = x;
    x = y;
    y = tmp;
    printf("%d %d\n", x, y);
}
```

```
10 20
20 10
10 20
20 10
10 20
20 10
10 20
20 10
10 20
20 10
10 10
10 10
10 10
10 10
...


10 20
20 10
10 20
20 10
10 20
20 10
10 10
10 10
10 10
10 10
```

```
int x = 10, y = 20;


int main () {
    int i;
    signal(SIGUSR1, handler1);
    signal(SIGUSR2, handler2);
    for (i=0; i<10; i++) {
        if (fork() == 0)
            while (1) {
                kill(getppid(), SIGUSR1);
                kill(getppid(), SIGUSR2);
            }
    }
    while(1) pause();
}

void handler1(int s) { swapglobs(); }

void handler2(int s) { swapglobs(); }

void swapglobs() {
    int tmp = x;
    x = y;
    y = tmp;
    printf("%d %d\n", x, y);
}
```

```
10 20
20 10
10 20
20 10
10 20
20 10
10 20
20 10
10 20
20 10
10 20
20 10
10 20
20 10
10 20
20 10
10 20
20 20
20 20
20 20
20 20
20 20
20 20
20 20
20 20
20 20
```

lesson 1: signals can be delivered *at any time*

- may interrupt any *nonatomic* operation
- problematic if using global variables!

design goal 1: minimize use of global variables in sighandlers

- if needed, ideally use data that can be read/written atomically (*most* primitives) *lesson 2*: a sighandler may execute in overlapping fashion (with itself)

- when used to handle multiple signals

design goal 2: prefer separate handlers for different signals

- otherwise, must design handlers to be *reentrant* — i.e., able to be called again (re-entered) when already executing *lesson 3*: execution of sighandlers for separate signals may overlap

- any functions they call may have overlapping execution *design goal 3*: keep sighandlers simple; minimize calls to other functions

- any functions called by sighandlers should be reentrant!