CMPSCI 230
Computer Systems Principles

More on Processes

Objectives

- To learn how to control child processes
- To learn about other process-related functions
- To learn how to load and execute programs
- To learn about signals
- To learn about pipes

Create & Terminate Processes

We use the fork() function to create a new process.

```
pid_t pid = fork();
if (pid == 0) {
   printf("hello from child\n");
} else {
   printf("hello from parent\n");
}
```

Create & Terminate Processes

What happens here?

```
void main()
{
    if (fork() == 0) {
        /* Child */
        printf("Terminating Child, PID = %d\n",
                getpid());
        exit(0);
    } else {
        printf("Running Parent, PID = %d\n",
                getpid());
        while (1)
            ; /* Infinite loop */
    }
}
```

01_zombie.c

Zombie!

01_zombie.c

Linux

```
student@osboxes:code$ ./01_zombie
Running Parent, PID = 1176
Terminating Child, PID = 1177
^Z
[1]+  Stopped                 ./01_zombie
student@osboxes:code$ ps
  PID TTY          TIME CMD
 1136 pts/0    00:00:00 bash
 1176 pts/0    00:00:03 01_zombie
 1177 pts/0    00:00:00 01_zombie <defunct>
 1179 pts/0    00:00:00 ps
student@osboxescode$
```

01_zombie.c

Mac OS X

```
ridgway@swift:code$ ./01_zombie
Running Parent, PID = 25768
Terminating Child, PID = 25769
^Z
[1]+  Stopped                 ./01_zombie
ridgway@swift:code$ ps
  PID TTY           TIME CMD
  444 ttys000    0:00.92 -bash
25768 ttys000    0:02.44 ./01_zombie
25769 ttys000    0:00.00 (01_zombie)
ridgway@swift:code$
```

Create & Terminate Processes

What about this one?

```
void main()
{
    if (fork() == 0) {
    /* Child */
    printf("Running Child, PID = %d\n",
           getpid());
    while (1)
        ; /* Infinite loop */
    } else {
    printf("Terminating Parent, PID = %d\n",
           getpid());
    exit(0);
    }
}
```

02_zombie.c

Zombie?

02_zombie.c

```
ridgway@swift:code$ ./02_zombie
Terminating Parent, PID = 25820
Running Child, PID = 25821
ridgway@swift:code$ ps -f
  UID   PID  PPID   C STIME   TTY           TIME CMD
  501   444   443   0  5:47PM ttys000    0:00.95 -bash
  501 25821     1   0 10:31AM ttys000    0:02.05 ./02_zombie
ridgway@swift:code$
```

NOTE!

How do we control processes?

Need a way to get rid of (**kill**) the zombies!

How do we control processes?

Need a way to get rid of **(kill)** the zombies!

This is called **reaping**!

Create & Terminate Processes

So, how do we “reap” a child process programmatically?

Zombie?

Create & Terminate Processes

So, how do we "reap" a child process programmatically?

`wait()`

`waitpid()`

Zombie?

Create & Terminate Processes

`int wait(int *child_status)`

```
void main()
{
   pid_t pid[N];
   int i;
   int child_status;
   for (i = 0; i < N; i++) {
      if ((pid[i] = fork()) == 0) { // Child
         exit(100+i);
   }  }
   for (i = 0; i < N; i++) {
      pid_t wpid = wait(&child_status);
      if (WIFEXITED(child_status)) {
         printf("Child %d terminated with exit status %d\n",
                wpid, WEXITSTATUS(child_status));
      } else {
         printf("Child %d terminated abnormally\n", wpid);
}  }  }
```

fork a child processes

Create & Terminate Processes

`int wait(int *child_status)`

```
void main()
{
   pid_t pid[N];
   int i;
   int child_status;
   for (i = 0; i < N; i++) {
      if ((pid[i] = fork()) == 0) { // Child
         exit(100+i);
   }  }
   for (i = 0; i < N; i++) {
      pid_t wpid = wait(&child_status);
      if (WIFEXITED(child_status)) {
         printf("Child %d terminated with exit status %d\n",
                wpid, WEXITSTATUS(child_status));
      } else {
         printf("Child %d terminated abnormally\n", wpid);
}  }  }
```

wait for each to terminate

Create & Terminate Processes

`int wait(int *child_status)`

```
void main()
{
   pid_t pid[N];
   int i;
   int child_status;
   for (i = 0; i < N; i++) {
      if ((pid[i] = fork()) == 0) { // Child
         exit(100+i);
   }  }
   for (i = 0; i < N; i++) {
      pid_t wpid = wait(&child_status);
      if (WIFEXITED(child_status)) {
         printf("Child %d terminated with exit status %d\n",
                wpid, WEXITSTATUS(child_status));
      } else {
         printf("Child %d terminated abnormally\n", wpid);
}  }  }
```

wait for each to terminate

Get Info on Status

Create & Terminate Processes

`int waitpid(pid, &status, options)`

```
void main()
{
   pid_t pid[N];
   int i;
   int child_status;
   for (i = 0; i < N; i++) {
      if ((pid[i] = fork()) == 0) { // Child
         exit(100+i);
   }  }
   for (i = N-1; i >= 0; i--) {
      pid_t wpid = waitpid(pid[i], &child_status, 0);
      if (WIFEXITED(child_status)) {
         printf("Child %d terminated with exit status %d\n",
                wpid, WEXITSTATUS(child_status));
      } else {
         printf("Child %d terminated abnormally\n", wpid);
}  }  }
```

wait for specific child to terminate

Create & Terminate Processes

```
int waitpid(-1, &status, 0)
```

is the same as…

```
int wait(&status)
```

Activity

```
void main()
{
    if (fork() == 0) {
      printf("a");
    }
    else {
      printf("b");
      waitpid(-1, NULL, 0);
    }
    printf("c");
    exit(0);
}
```

What is the output of the program on the left?

A. acbc
B. bcac
C. abcc
D. bacc
E. A or C or D

Activity

```
void main()
{
    if (fork() == 0) {
      printf("a");
    }
    else {
      printf("b");
      waitpid(-1, NULL, 0);
    }
    printf("c");
    exit(0);
}
```

What is the output of the program on the left?

A. acbc
B. bcac
C. abcc
D. bacc
E. A or C or D

Objectives

- To learn how to control child processes
- To learn about **`sleep`** and **`wait`**
- To learn how to load and execute programs
- To learn about signals
- To learn about pipes

Putting a process to sleep: zzzzz…

```
unsigned int sleep(unsigned int secs);
```

```
#include <unistd.h>
void main()
{
  int amt = sleep(5);
}
```

Returns 0 if time has elapsed.

Returns non-0 if a signal woke up the process.

Putting a process to sleep: zzzzz…

```
int pause(void);
```

```
#include <unistd.h>
void main()
{
  int amt = pause();
}
```

Returns the amount of time the process was paused.

Objectives

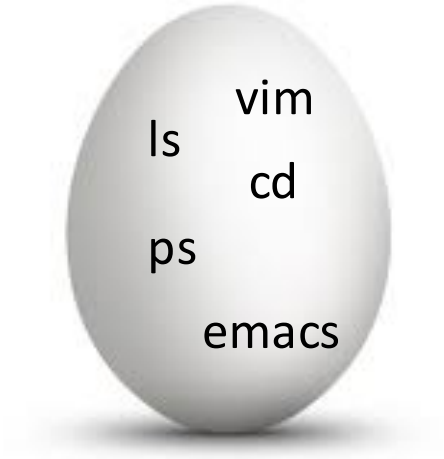
- To learn how to control child processes
- To learn about **`sleep`** and **`wait`**
- To learn how to load and execute programs
- To learn about signals
- To learn about pipes

Loading and Running Programs

What does a shell program do?

Loading and Running Programs

What does a shell program do?



A shell program is a special kind of program whose primary responsibility is to run other programs.

Loading and Running Programs

```
int execve(const char* filename,
           const char* argv[],
           const char* envp[]);
```

Loading and Running Programs

```
int execve(const char* filename,
           const char* argv[],
           const char* envp[]);
```

We know what these are…

Loading and Running Programs

```
int execve(const char* filename,
           const char* argv[],
           const char* envp[]);
```

We know what these are…

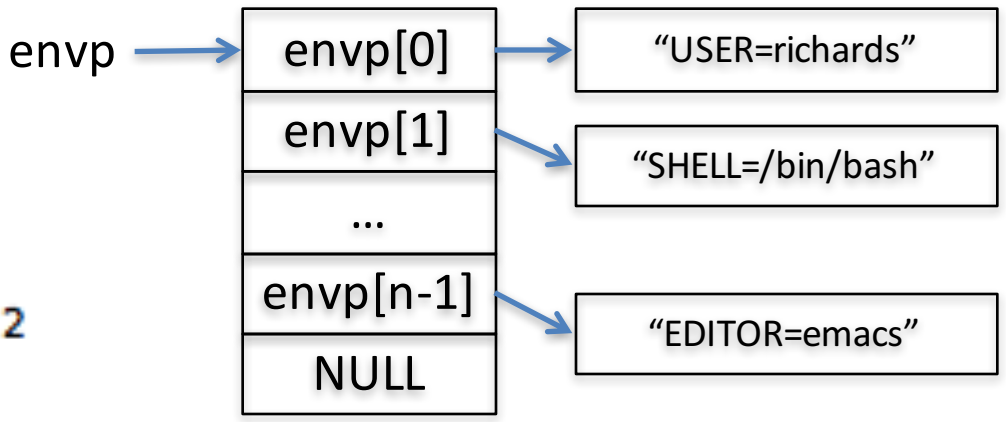
But, what is this?

Loading and Running Programs

```
int execve(const char* filename,
           const char* argv[],
           const char* envp[]);
```

Every program runs in an "environment". The environment is customized using *environment variables*: PATH, EDITOR, …

```
[elnux1:~] printenv
XDG_SESSION_ID=7763
HOSTNAME=elnux1.cs.umass.edu
TERM=xterm-256color
SHELL=/bin/bash
HISTSIZE=1000
SSH_CLIENT=128.119.40.196 58373 22
SSH_TTY=/dev/pts/10
USER=richards
```



Loading and Running Programs

```
int execve(const char* filename,
           const char* argv[],
           const char* envp[]);
```

It turns out that the more general form for main is:

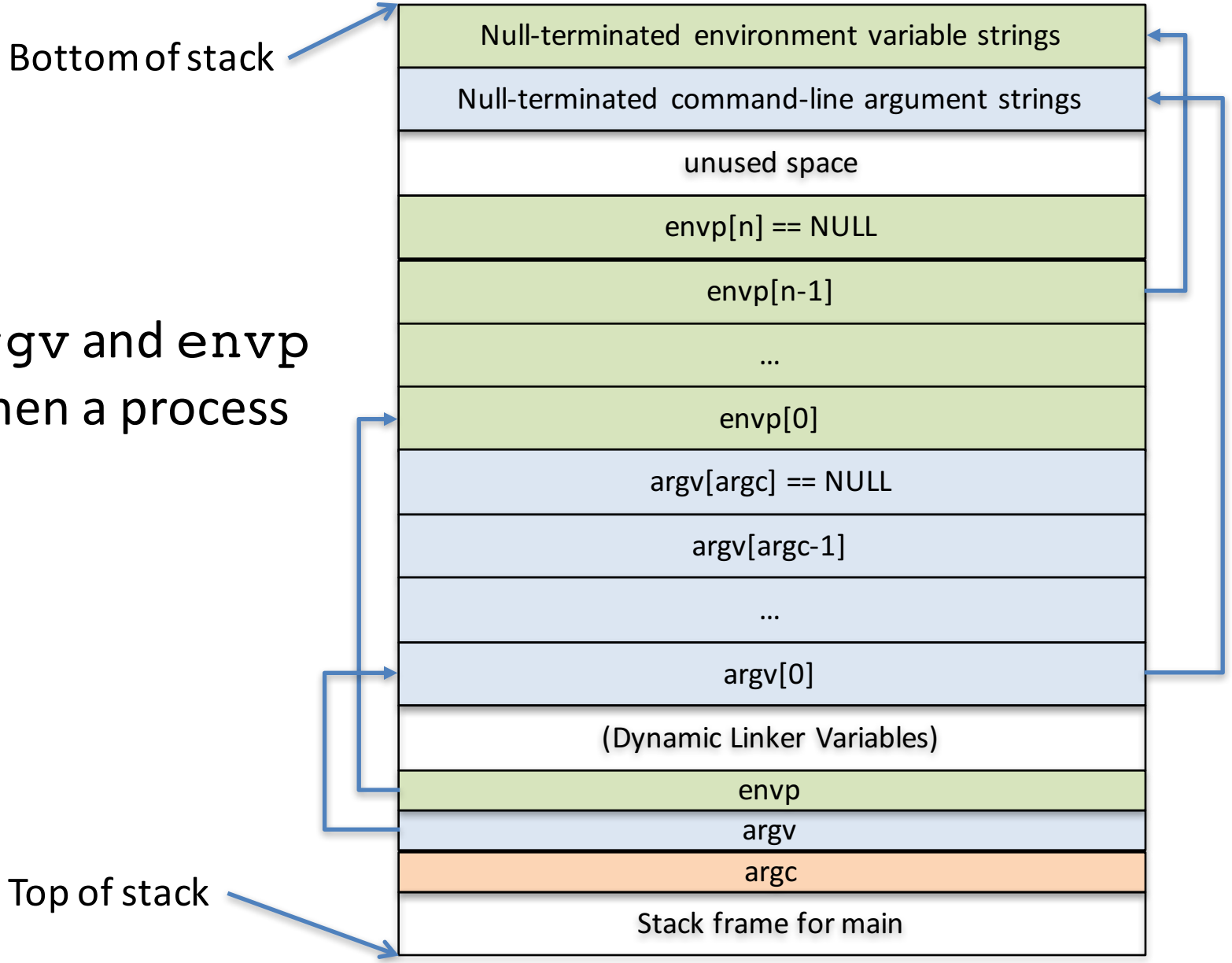
```
int main(int argc,
         const char* argv[],
         const char* envp[]);
```

Loading and Running Programs

Where are `argv` and `envp` in memory when a process executes?

Loading and Running Programs

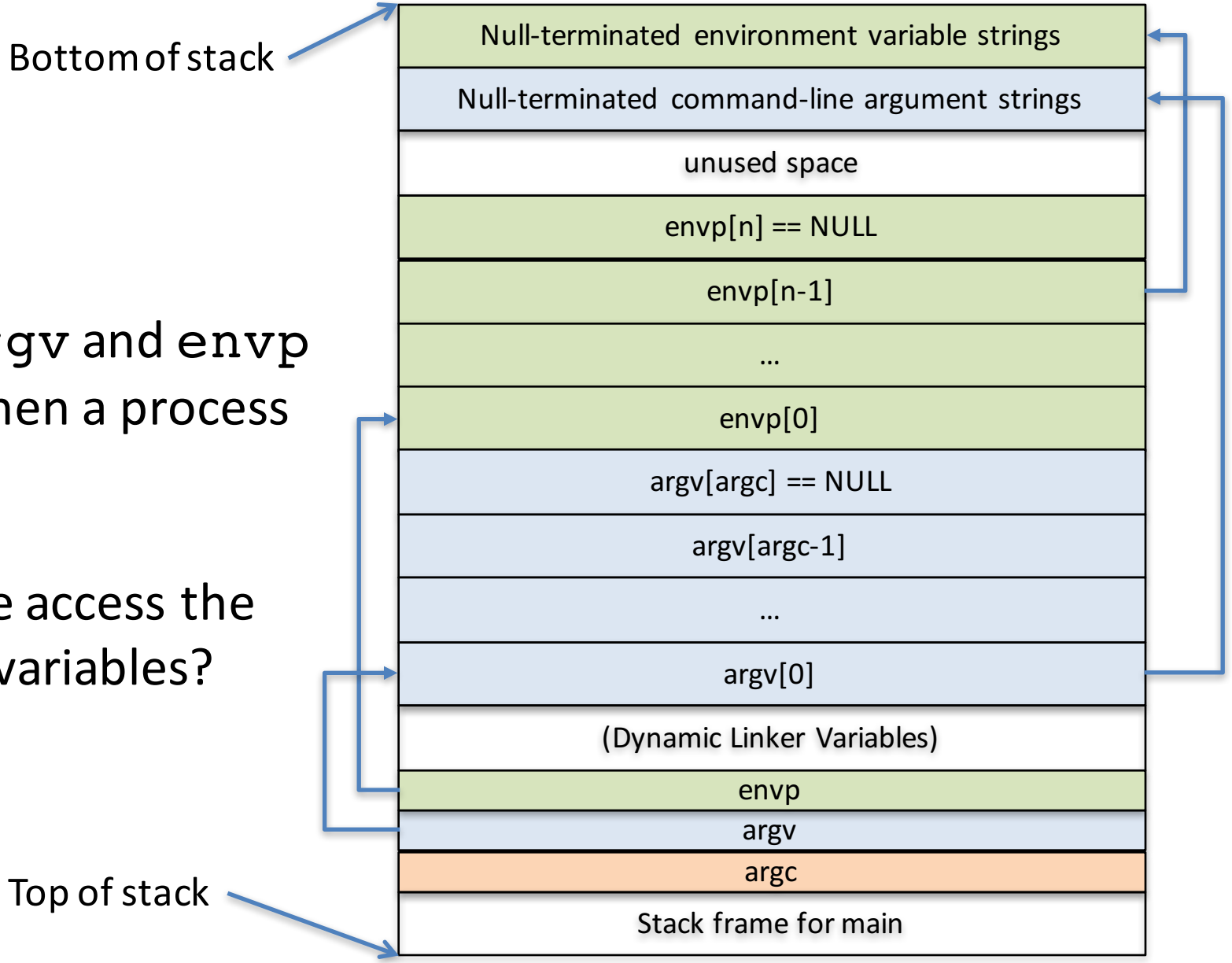
Where are `argv` and `envp` In memory when a process executes?



Loading and Running Programs

Where are `argv` and `envp` In memory when a process executes?

So, how do we access the environment variables?



Loading and Running Programs

```
int* getenv(const char* name);
```

Returns *pointer* to value if there is an entry of the form "`name=value`", and NULL if there is not

```
int setenv(const char* name,
           const char* newval,
           int overwrite);
```

Returns 0 on success, -1 on error.

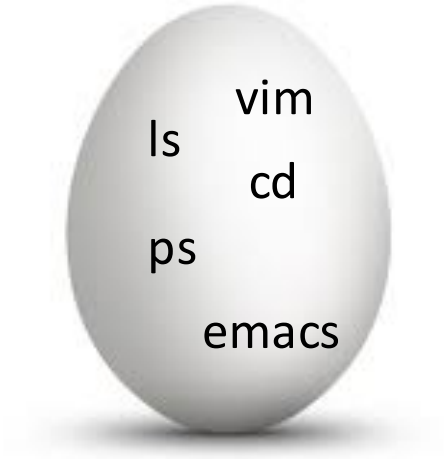
```
void unsetenv(const char* name);
```

Returns nothing.

`03_setenv.c`

Loading and Running Programs

What does a shell program do?

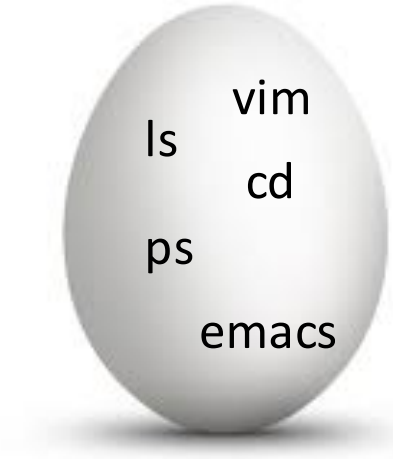


A shell program is a special kind of program whose primary responsibility is to run other programs.

Loading and Running Programs

What does a shell program do?

fork + execve



`04_shell.c`

A shell program is a special kind of program whose primary responsibility is to run other programs.

Objectives

- To learn how to control child processes
- To learn about **`sleep`** and **`wait`**
- To learn how to load and execute programs
- To learn about signals
- To learn about pipes

Signals

How do we communicate to processes?

We send them messages called *signals*

Signal	Value	Action	Comment
SIGHUP	1	Term	Hangup detected on controlling terminal or death of controlling process
SIGINT	2	Term	Interrupt from keyboard
SIGQUIT	3	Core	Quit from keyboard
SIGILL	~~4~~	Core	Illegal Instruction
SIGABRT	6	Core	Abort signal from **abort**(3)
SIGFPE	~~8~~	Core	Floating point exception
SIGKILL	9	Term	Kill signal
SIGSEGV	~~11~~	Core	Invalid memory reference
SIGPIPE	~~13~~	Term	Broken pipe: write to pipe with no readers
SIGALRM	14	Term	Timer signal from **alarm**(2)
SIGTERM	15	Term	Termination signal
SIGUSR1	30,10,16	Term	User-defined signal 1
SIGUSR2	31,12,17	Term	User-defined signal 2
SIGCHLD	20,17,18	Ign	Child stopped or terminated
SIGCONT	19,18,25	Cont	Continue if stopped
SIGSTOP	17,19,23	Stop	Stop process
SIGTSTP	18,20,24	Stop	Stop typed at tty
SIGTTIN	21,21,26	Stop	tty input for background process
SIGTTOU	22,22,27	Stop	tty output for background process

A **signal** is an *event* of some type that has occurred in the system.

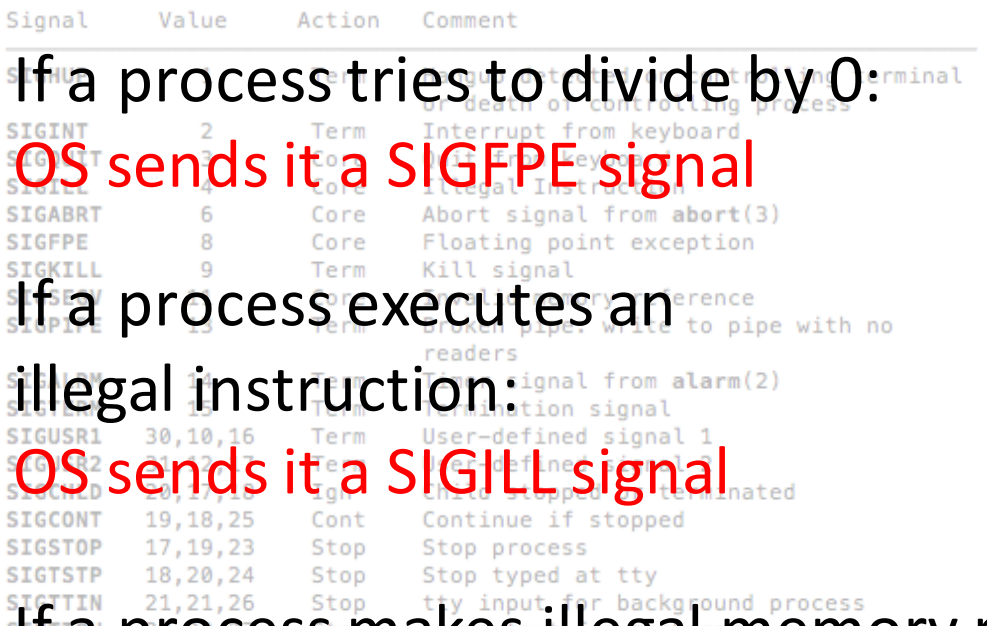
It allows the OS to communicate to a process
AND
user processes to communicate to each other

Signals

How do we communicate to processes?

We send them messages called *signals*

A **signal** is an *event* of some type that has occurred in the system.



Signal	Value	Action	Comment
SIGINT	2	Term	Interrupt from keyboard
SIGILL	4	Core	Illegal Instruction
SIGABRT	6	Core	Abort signal from **abort**(3)
SIGFPE	8	Core	Floating point exception
SIGKILL	9	Term	Kill signal
SIGUSR1	30,10,16	Term	User-defined signal 1
SIGCONT	19,18,25	Cont	Continue if stopped
SIGSTOP	17,19,23	Stop	Stop process
SIGTSTP	18,20,24	Stop	Stop typed at tty

If a process tries to divide by 0:

OS sends it a SIGFPE signal

If a process executes an illegal instruction:

OS sends it a SIGILL signal

It allows the OS to communicate to a process

AND

user processes to communicate to each other

If a process makes illegal memory reference:

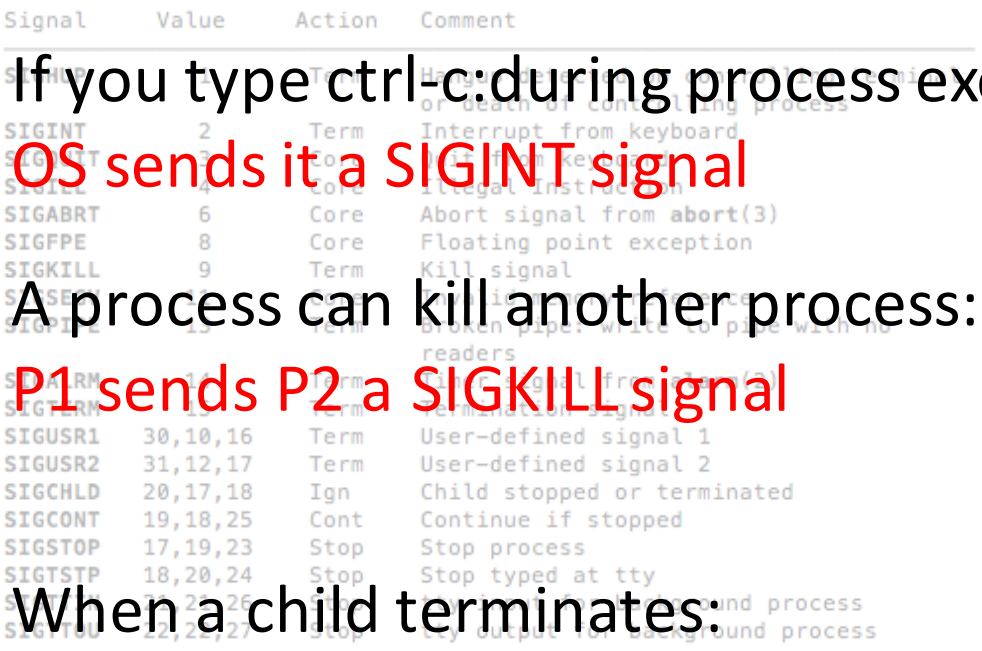
OS sends it a SIGSEGV signal

Signals

How do we communicate to processes?

We send them messages called *signals*

A **signal** is an *event* of some type that has occurred in the system.



Signal	Value	Action	Comment
SIGINT	2	Term	Interrupt from keyboard
SIGILL	4	Core	Illegal instruction
SIGABRT	6	Core	Abort signal from **abort**(3)
SIGFPE	8	Core	Floating point exception
SIGKILL	9	Term	Kill signal
SIGUSR1	30,10,16	Term	User-defined signal 1
SIGUSR2	31,12,17	Term	User-defined signal 2
SIGCHLD	20,17,18	Ign	Child stopped or terminated
SIGCONT	19,18,25	Cont	Continue if stopped
SIGSTOP	17,19,23	Stop	Stop process
SIGTSTP	18,20,24	Stop	Stop typed at tty

If you type ctrl-c:during process exec

OS sends it a SIGINT signal

A process can kill another process:

P1 sends P2 a SIGKILL signal

When a child terminates:

OS sends the parent a SIGCHLD signal

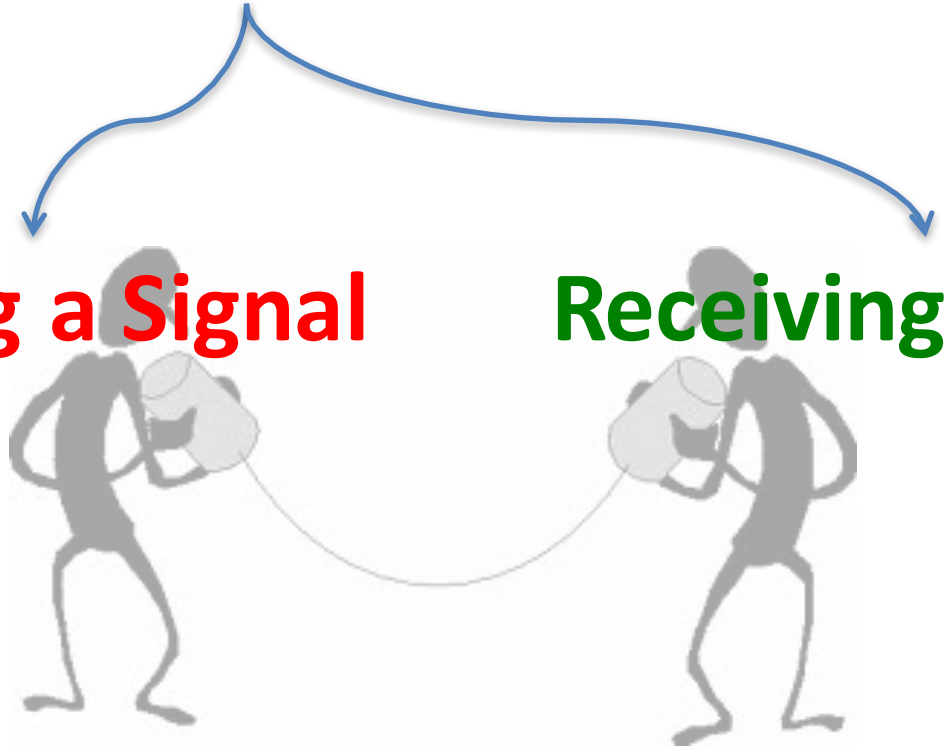
It allows the OS to communicate to a process AND

user processes to communicate to each other

Signals

How do we communicate to processes?

The transfer of a signal to a destination occurs in 2 steps



Sending a Signal

Receiving a Signal

Signals

How do we communicate to processes?

Sending a Signal

The kernel *sends* a signal to a destination process by **updating** some state in the context of that process.

This occurs when the kernel detects a system event (div by 0) or when a process invokes the **`kill`** system call to explicitly request the kernel to send a signal to the destination process.

Signals

How do we communicate to processes?

Receiving a Signal

A destination process *receives* a signal when it is forced by the kernel to react in some way.

The process can either ignore the signal, terminate, or *catch* the signal by executing user-level functions called *signal handlers.*

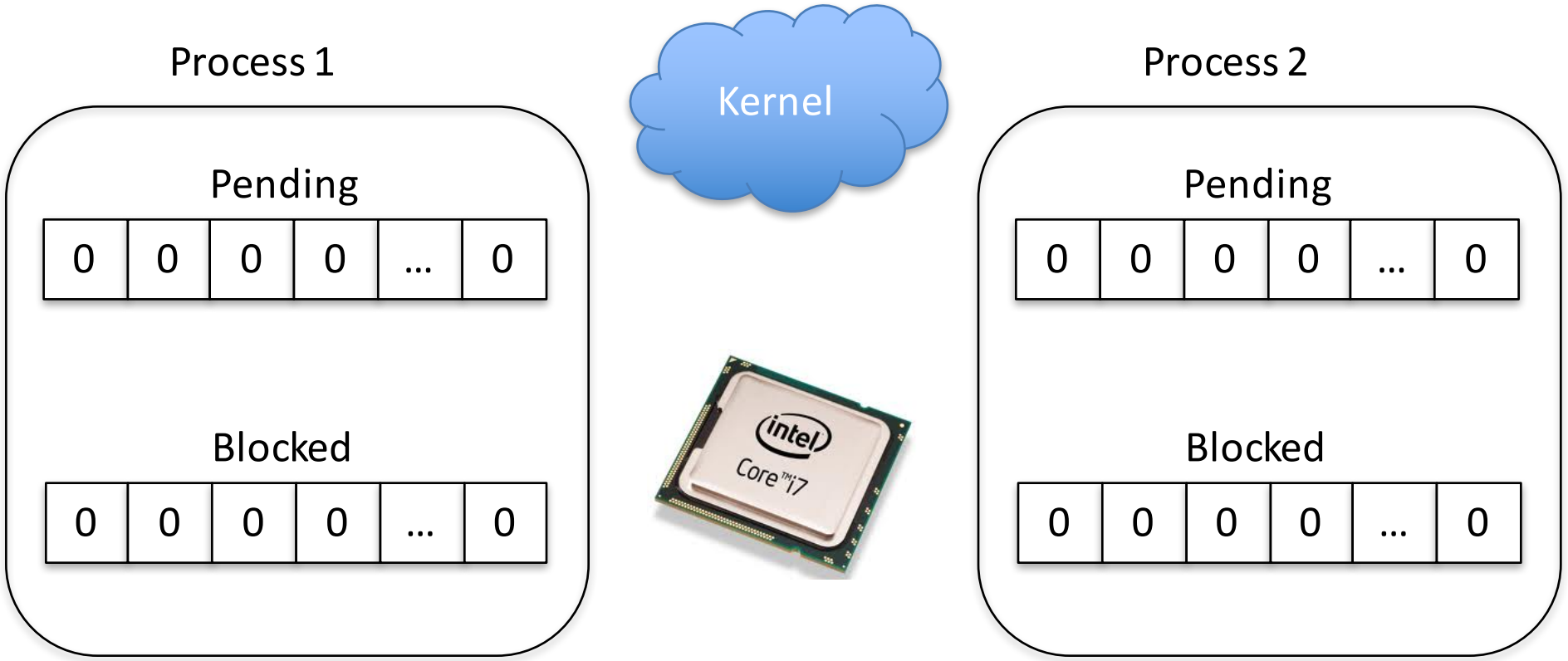
Signals

How do we communicate to processes?

- A **signal** is *pending* if sent but not yet received
  - There can be at most 1 pending signal for a type
  - **Important:** signals are *not queued*
    - If a process has a pending signal of type *k*, then subsequent signals of type *k* that are sent to that process are discarded.
- A process can **block** the receipt of certain signals
  - Blocked signals can be delivered, but will not be received until the signal is *unblocked*.

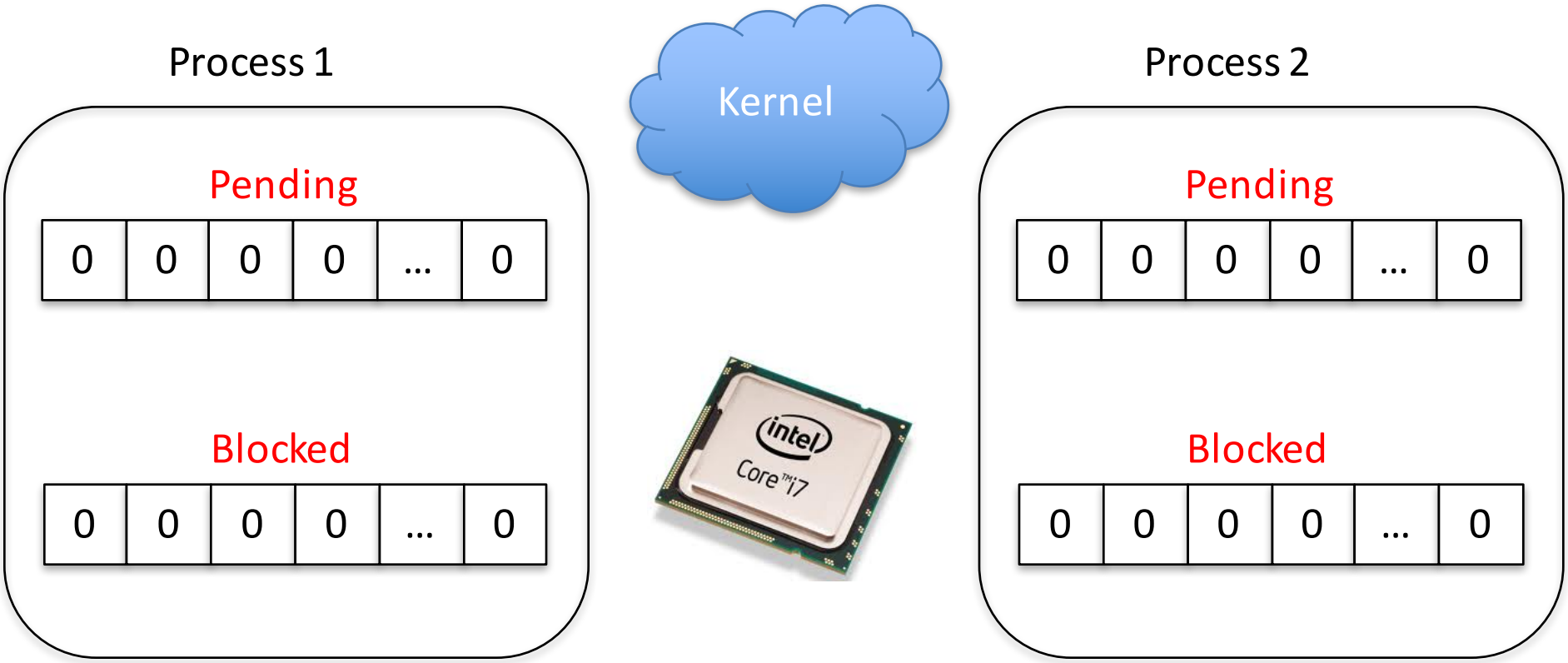
Signals

So, how are they implemented?



Signals

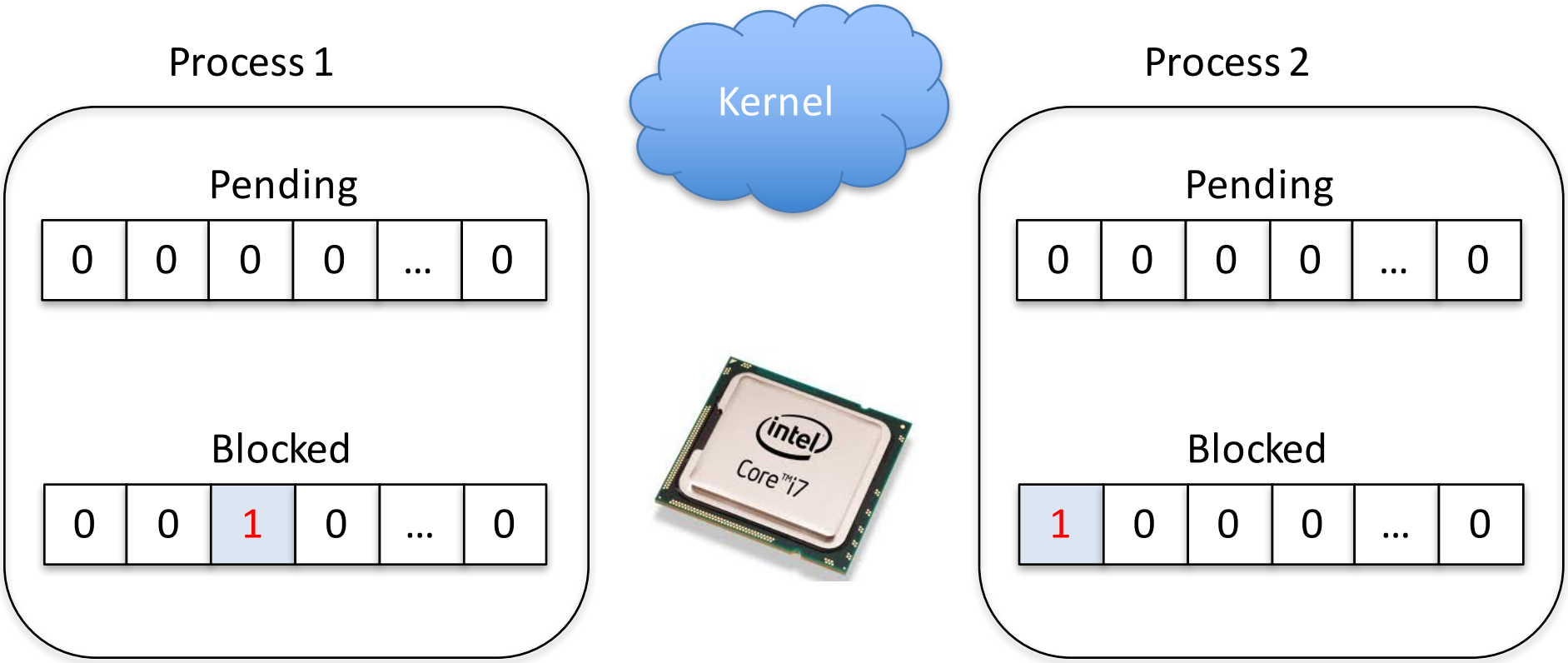
So, how are they implemented?



Each process structure contains a **pending** and **blocked** bit vector.
Each entry corresponds to a specific *signal*.

Signals

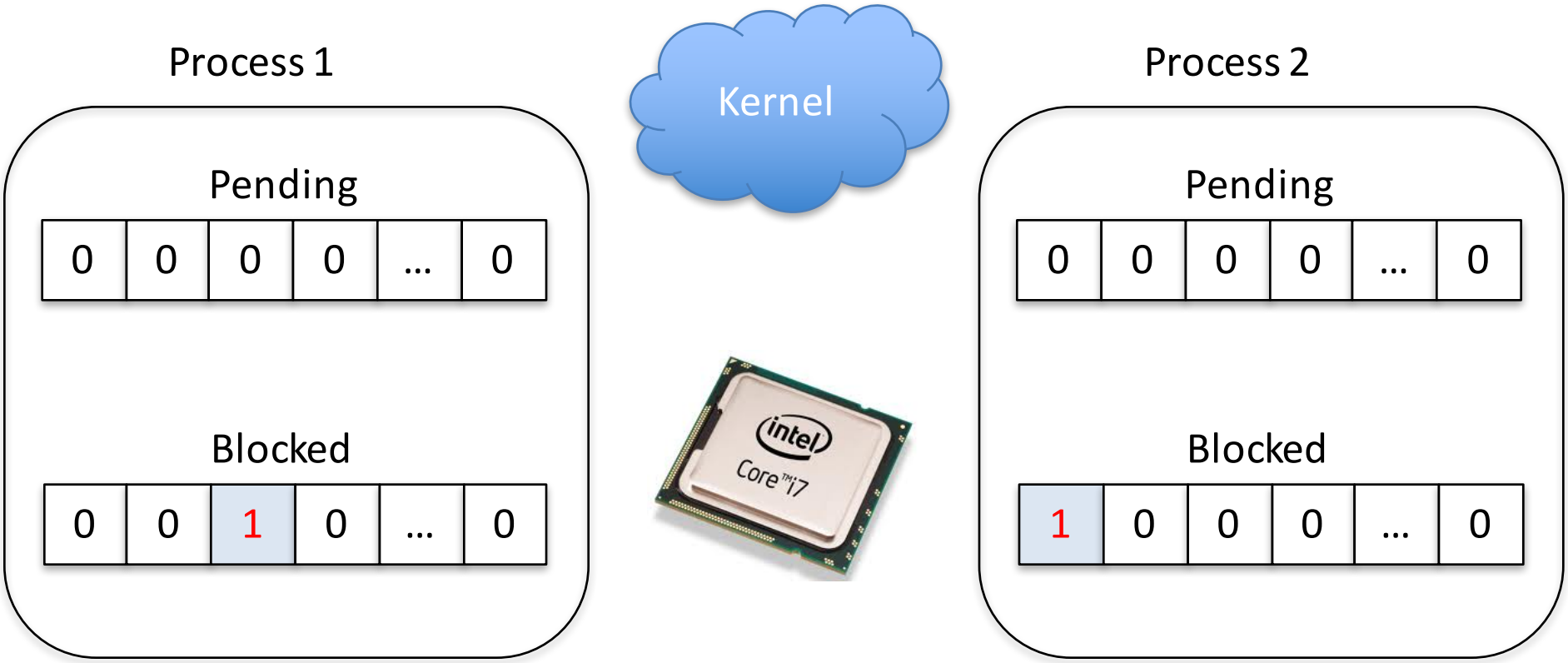
So, how are they implemented?



A process can decide to **block** a *signal* by setting the corresponding bit in the *blocked* bit vector.

Signals

So, how are they implemented?

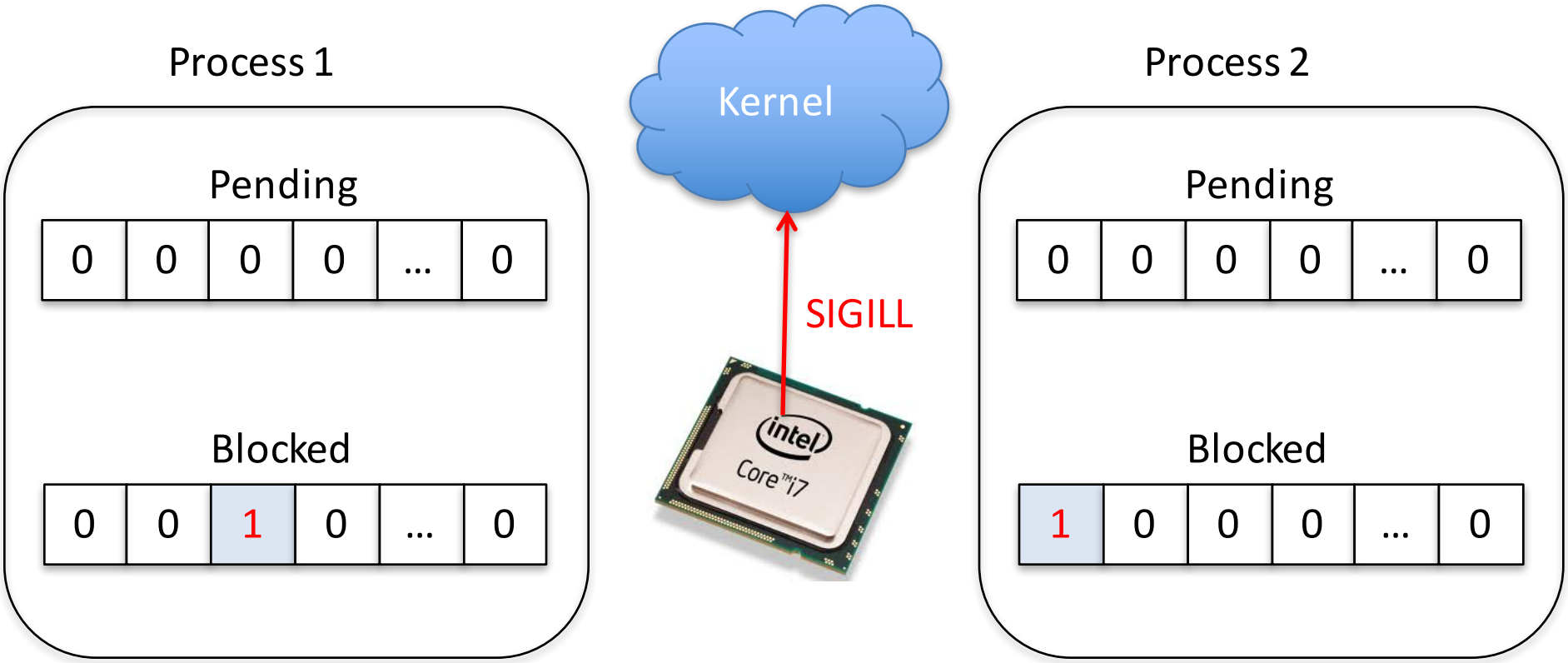


Not all signals can be blocked:

SIGKILL can't be blocked or handled by the process.

Signals

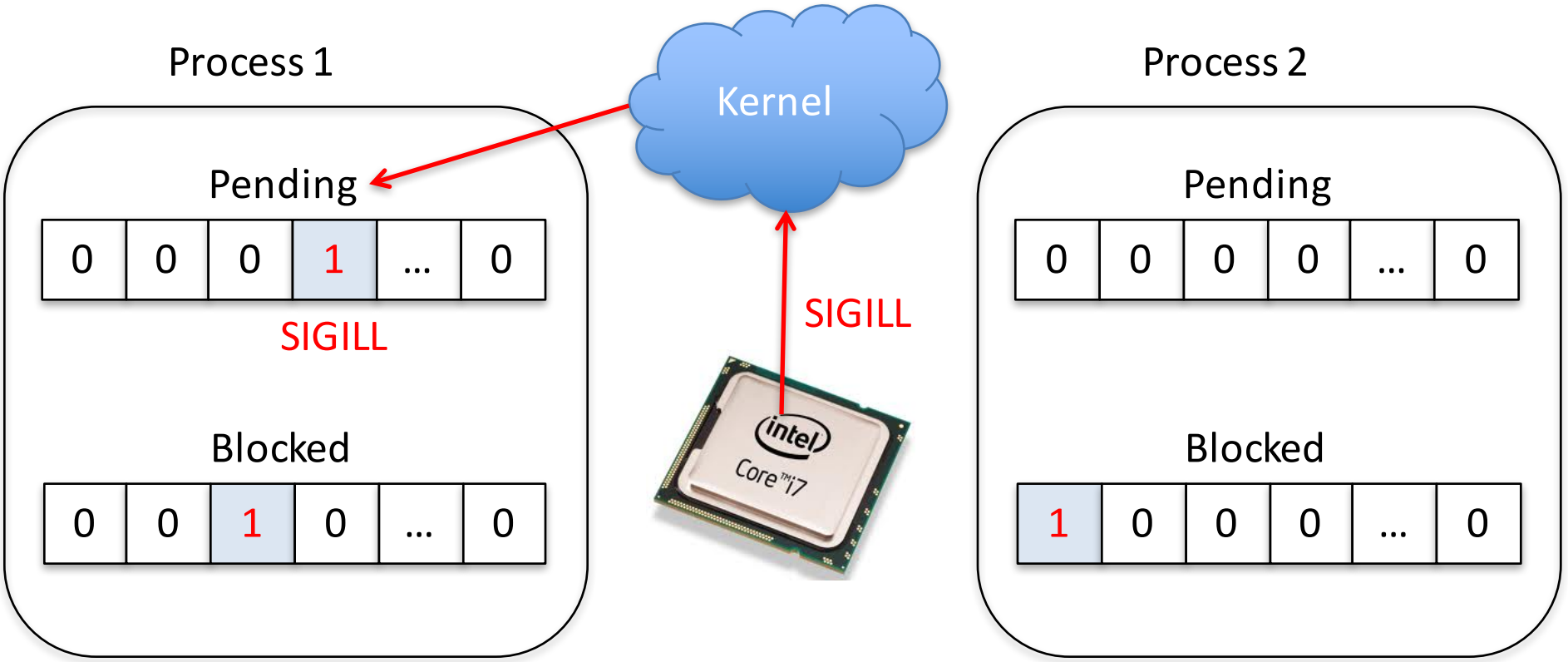
So, how are they implemented?



Imagine the processor encounters an illegal instruction
during the execution of Process 1

Signals

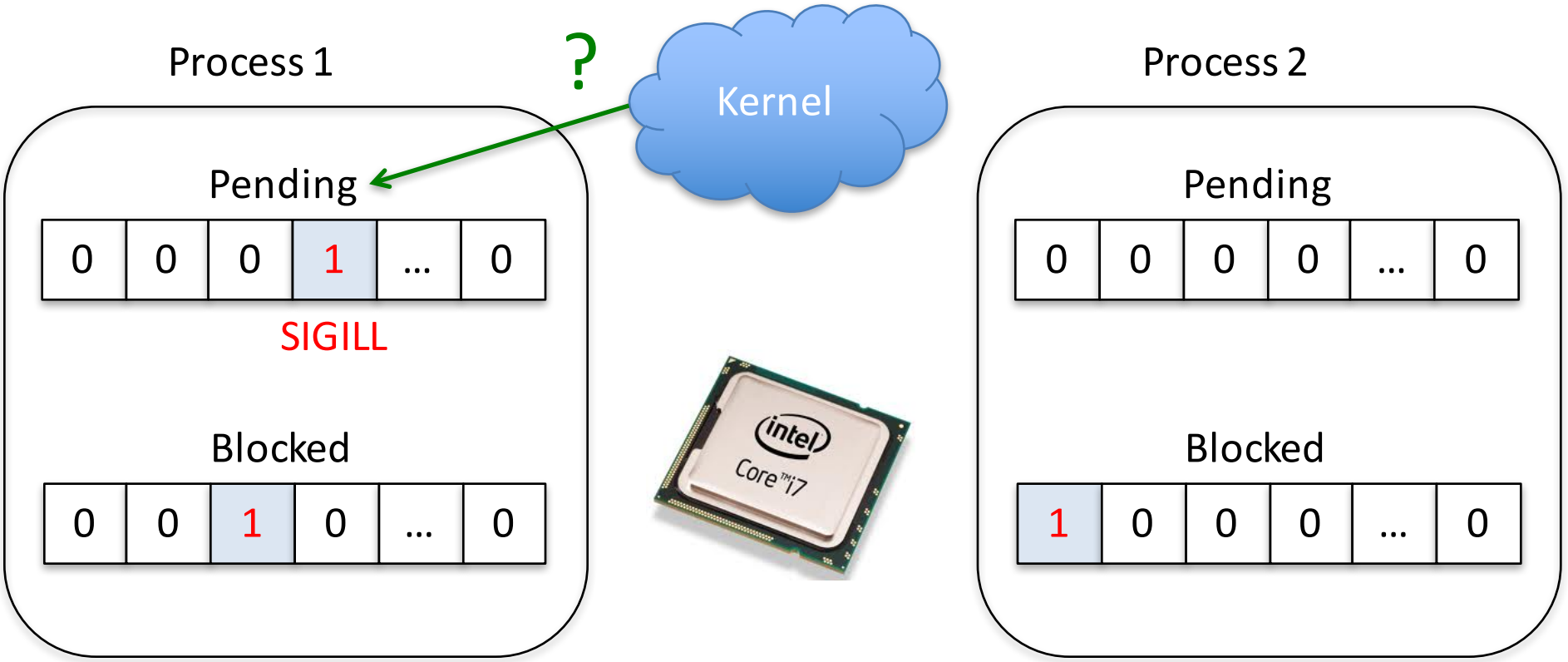
So, how are they implemented?



The kernel will set the corresponding bit in the Pending bit vector

Signals

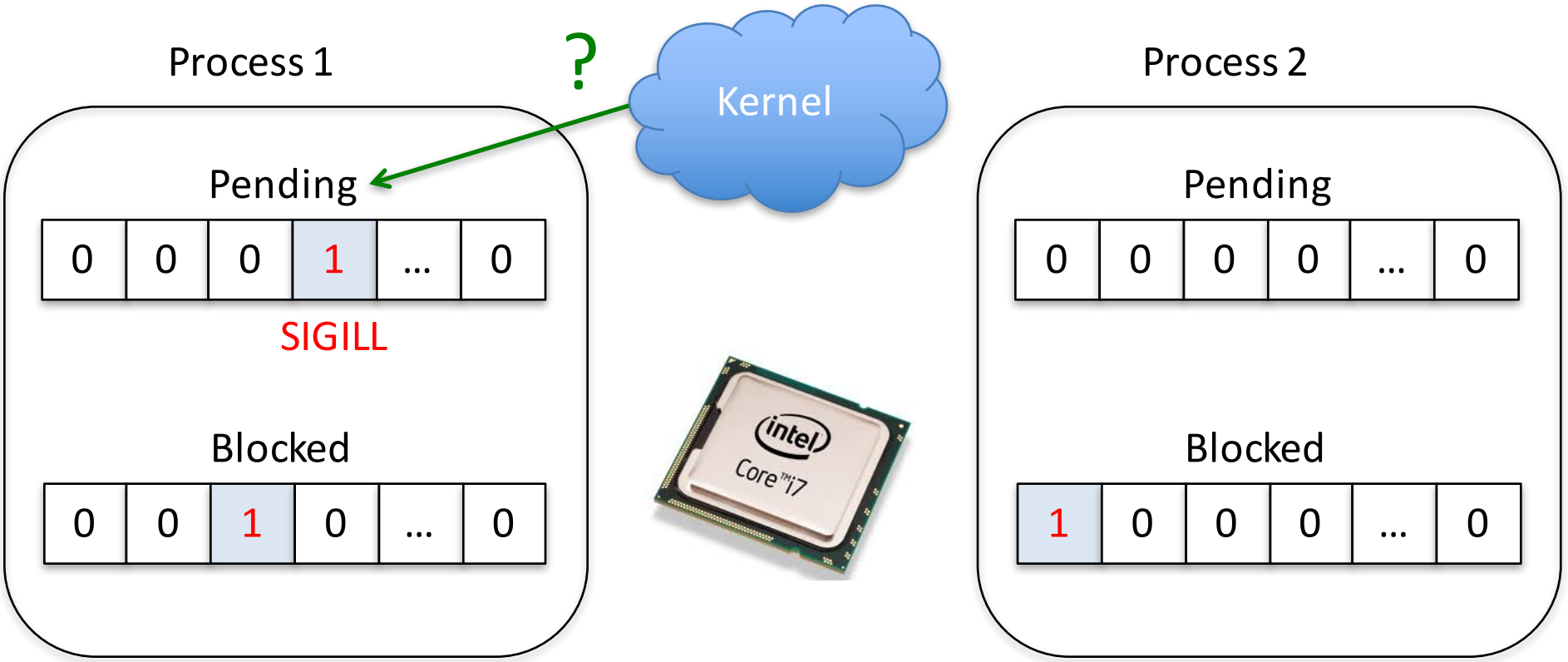
So, how are they implemented?



Just before Process 1 *resumes execution* the kernel will check to see if the process has any pending signals that are **not** blocked

Signals

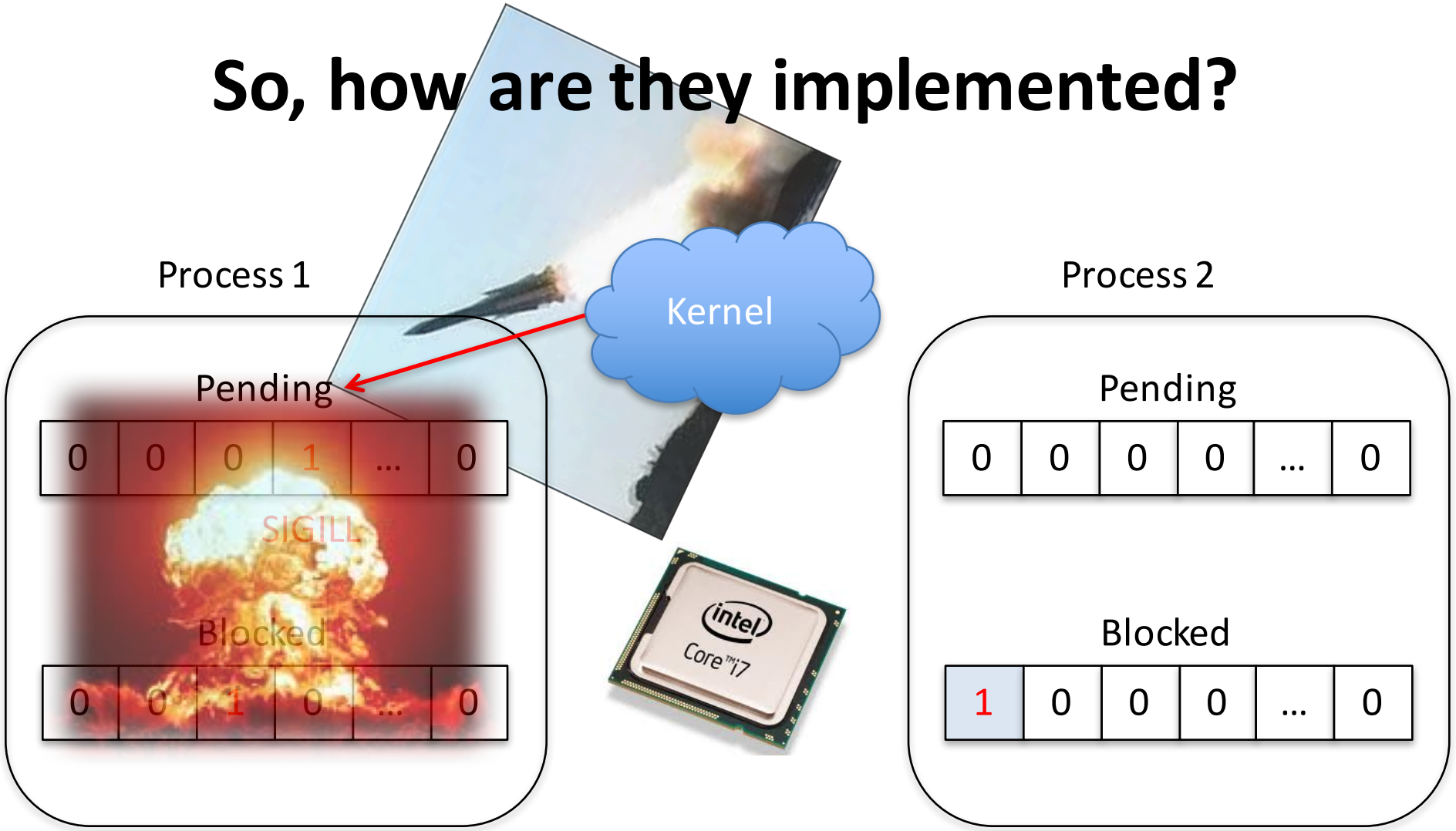
So, how are they implemented?



If it does it will check if the process has defined a *signal handler* for the signal or execute *default behavior*

Signals

So, how are they implemented?

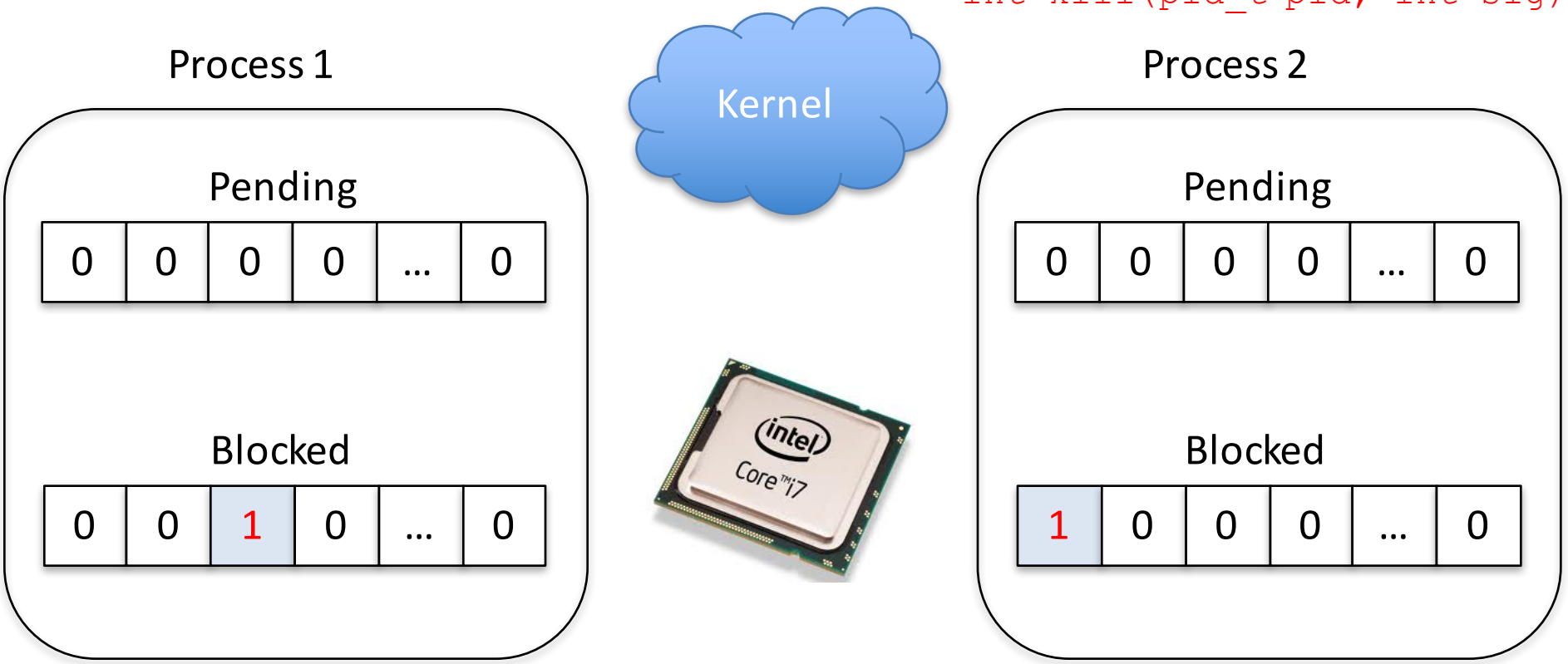


The default behavior for SIGILL is to *terminate* the process

Signals

So, how are they implemented?

```
#include <signal.h>
int kill(pid_t pid, int sig);
```

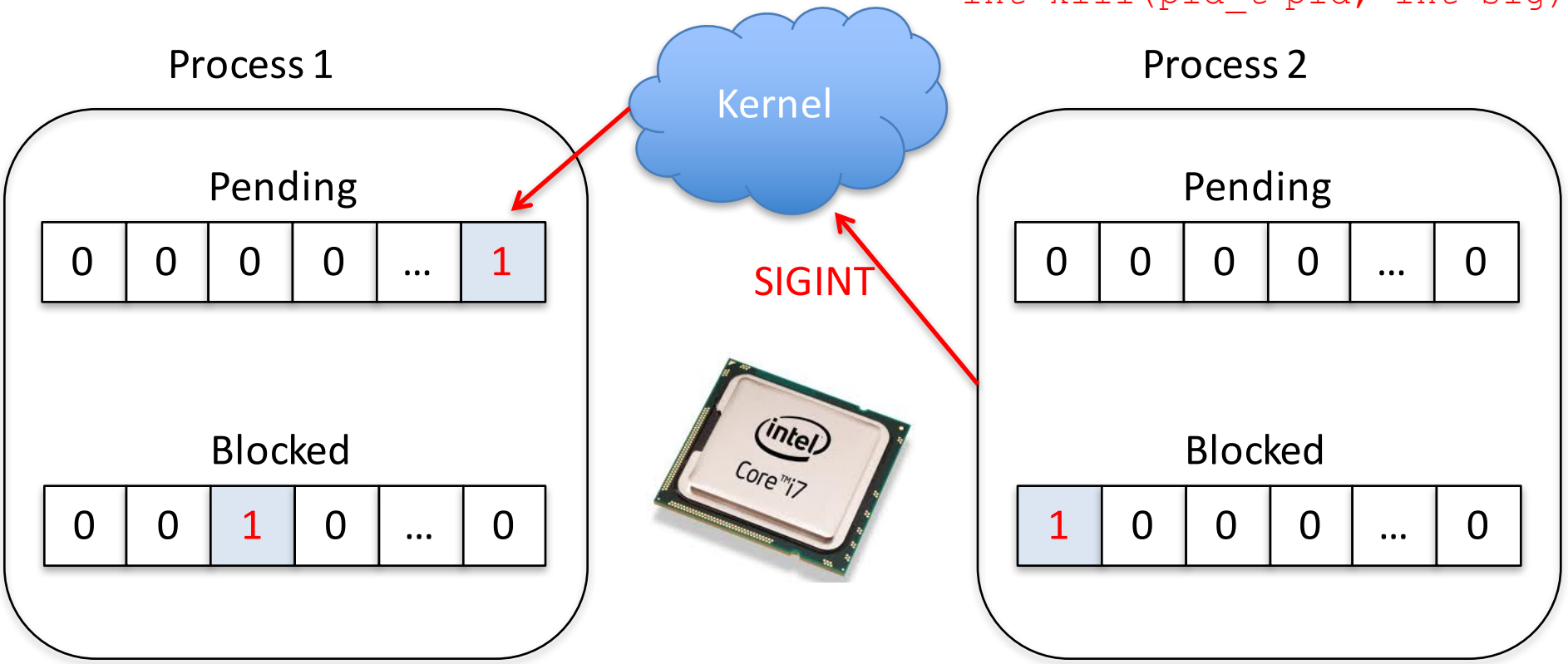


A process can send another process
a signal using the **kill** system call

Signals

So, how are they implemented?

```
#include <signal.h>
int kill(pid_t pid, int sig);
```

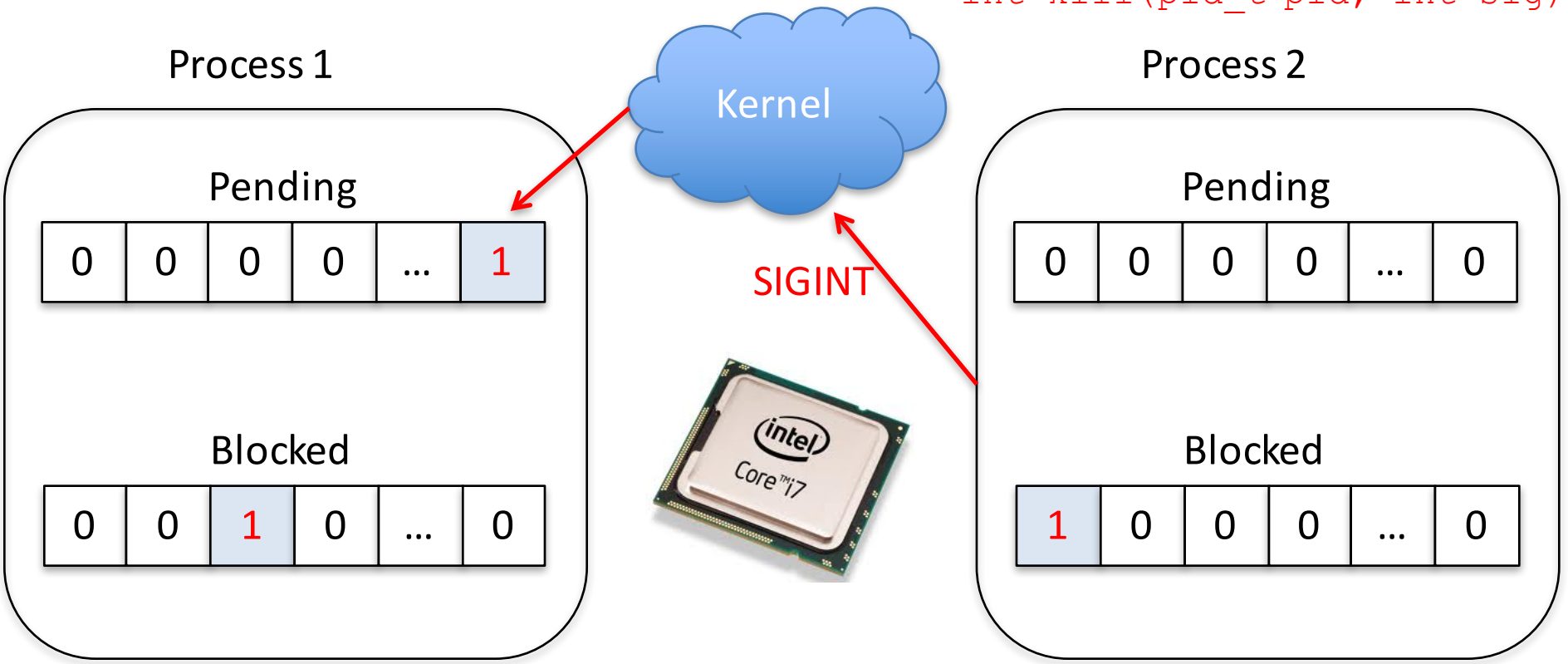


A process can send another process
a signal using the **kill** system call

Signals

So, how are they implemented?

```
#include <signal.h>
int kill(pid_t pid, int sig);
```

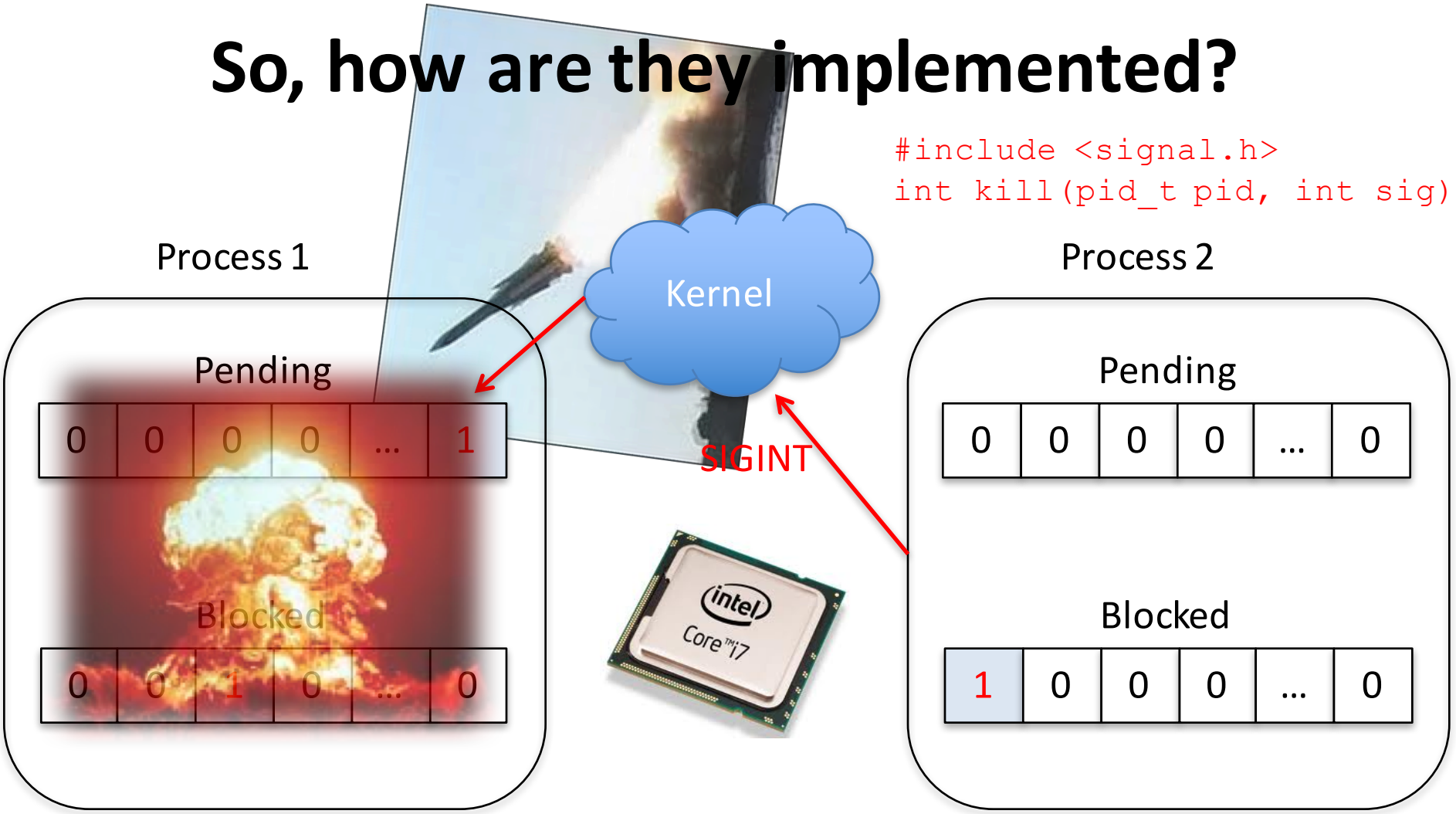


Again, the kernel will check for pending signals,
check for a handler or execute default behavior

Signals

So, how are they implemented?

```
#include <signal.h>
int kill(pid_t pid, int sig);
```



Again, the kernel will check for pending signals,
check for a handler or execute default behavior

Signals

We can send signals from the keyboard

`ctrl-c`

Typing ctrl-c from the keyboard sends a SIGINT signal to the shell.

The shell catches the signal and then sends a SIGINT to every process in the **foreground** process group.

`ctrl-z`

Typing ctrl-z from the keyboard sends a SIGTSTP signal to the shell.

The shell catches the signal and then sends a SIGTSTP to every process in the **foreground** process group.

Signals

Ok, so how do we send signals programmatically?

```
#include <signal.h>
int kill(pid_t pid, int sig);
```

05_kill_prog.c

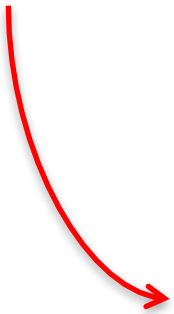
06_handler.c

Signals

And how do we catch a signal?

```
#include <unistd.h>
unsigned int alarm(unsigned int secs);
```

Generates a SIGALRM signal to the calling process after `secs` seconds.



Need a handler to "catch" the signal and do something interesting.

`07_alarm.c` **`08_sigint.c`**

Objectives

- To learn how to control child processes
- To learn about **`sleep`** and **`wait`**
- To learn how to load and execute programs
- To learn about signals
- To learn about pipes

How else can processes communicate?

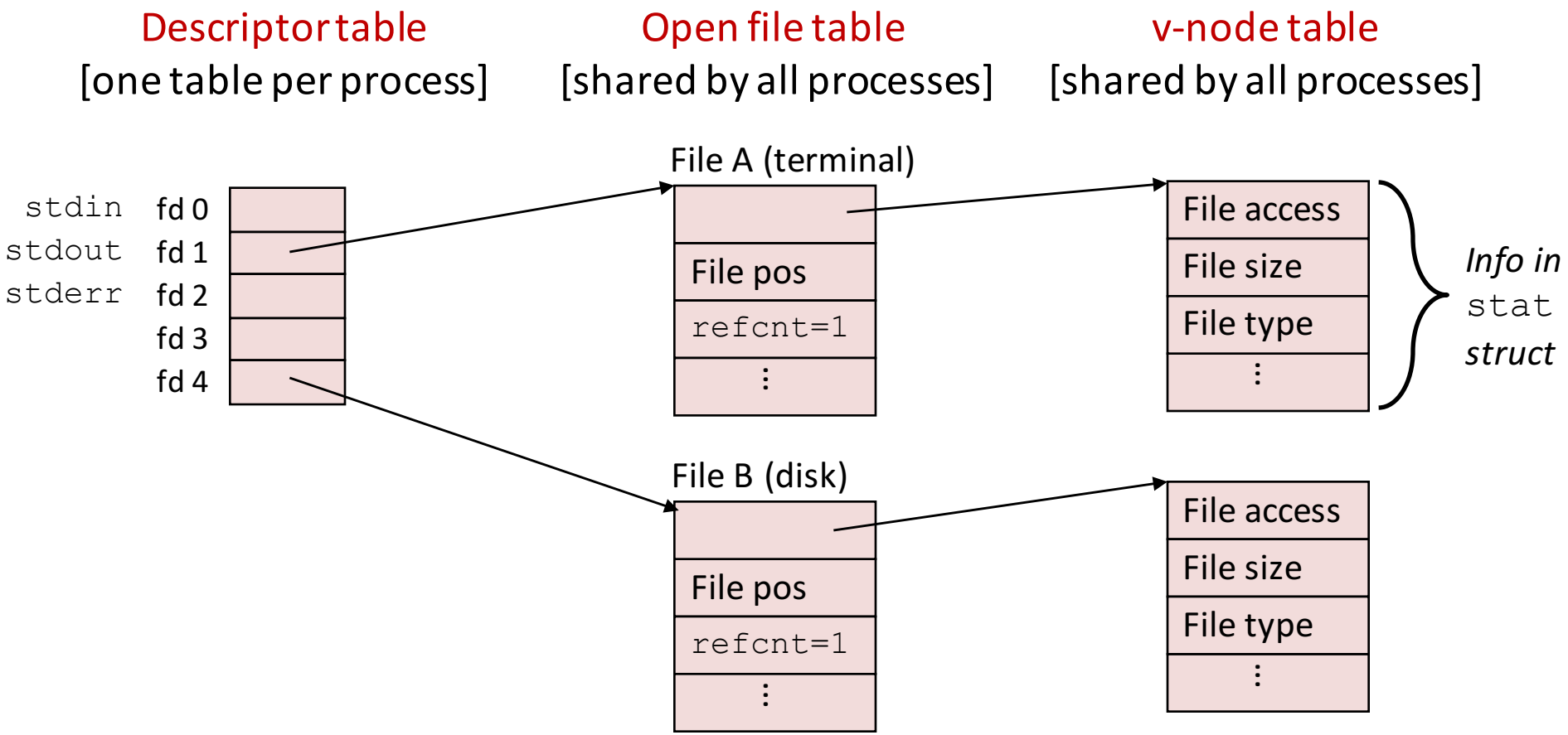
- **Different Address Space**
  - Great for protection
  - Hard for communication
  - Can’t share data structures
- **Is there another way to communicate?**
  - Yes, through files…
  - But, we need to understand how files work at a lower level…

Low-Level Files

- Two system calls
  - int open(char *path, int flags)
  - int close(int fd)
- At this level we work with *file descriptors*
- What is a *file descriptor*?
  - An integer that is used to reference an open file inside the operating system
  - Processes share open files!

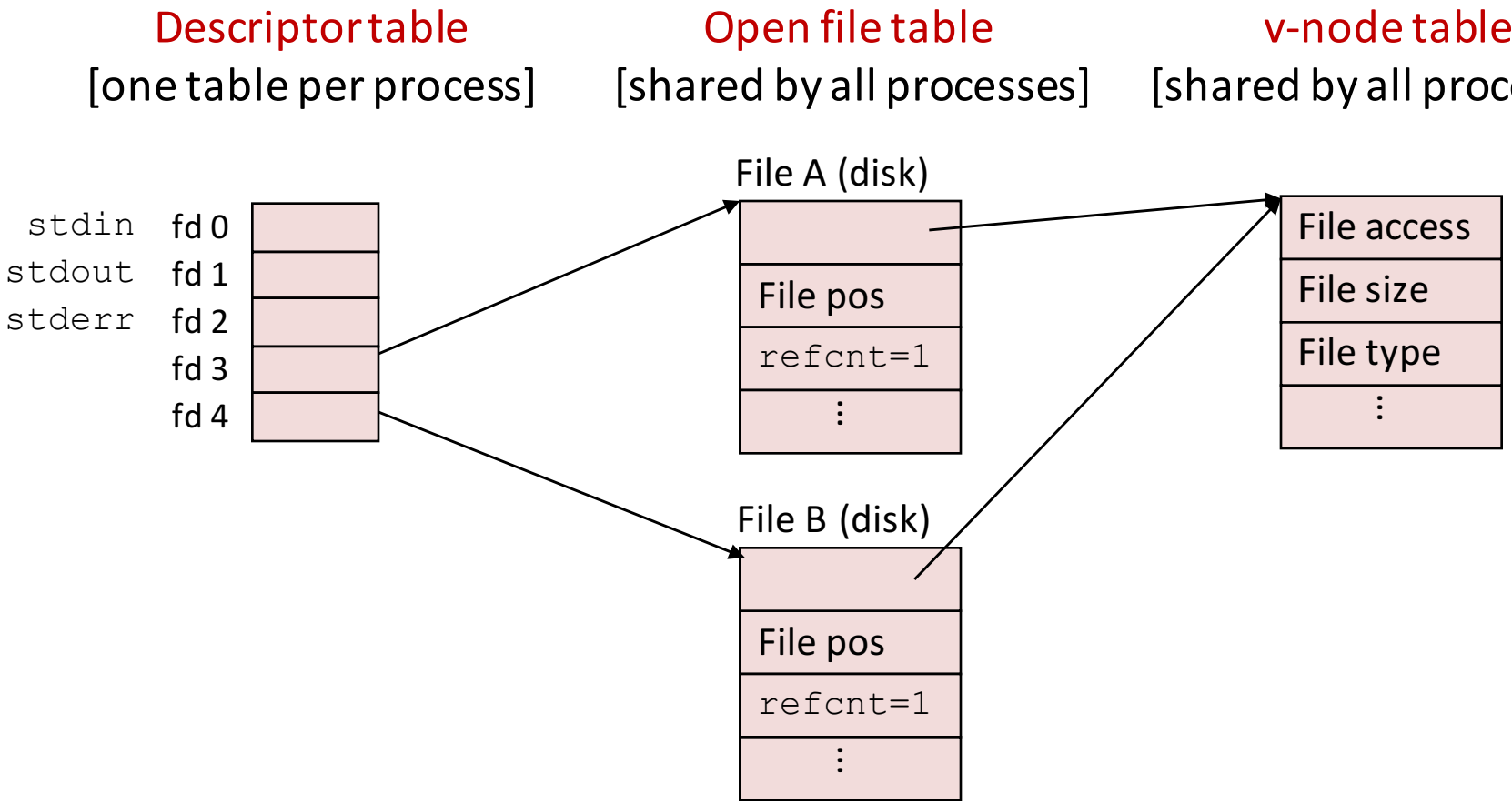
How the Unix Kernel Represents Open Files

- Two descriptors referencing two distinct open disk files. Descriptor 1 (stdout) points to terminal, and descriptor 4 points to open disk file



File Sharing

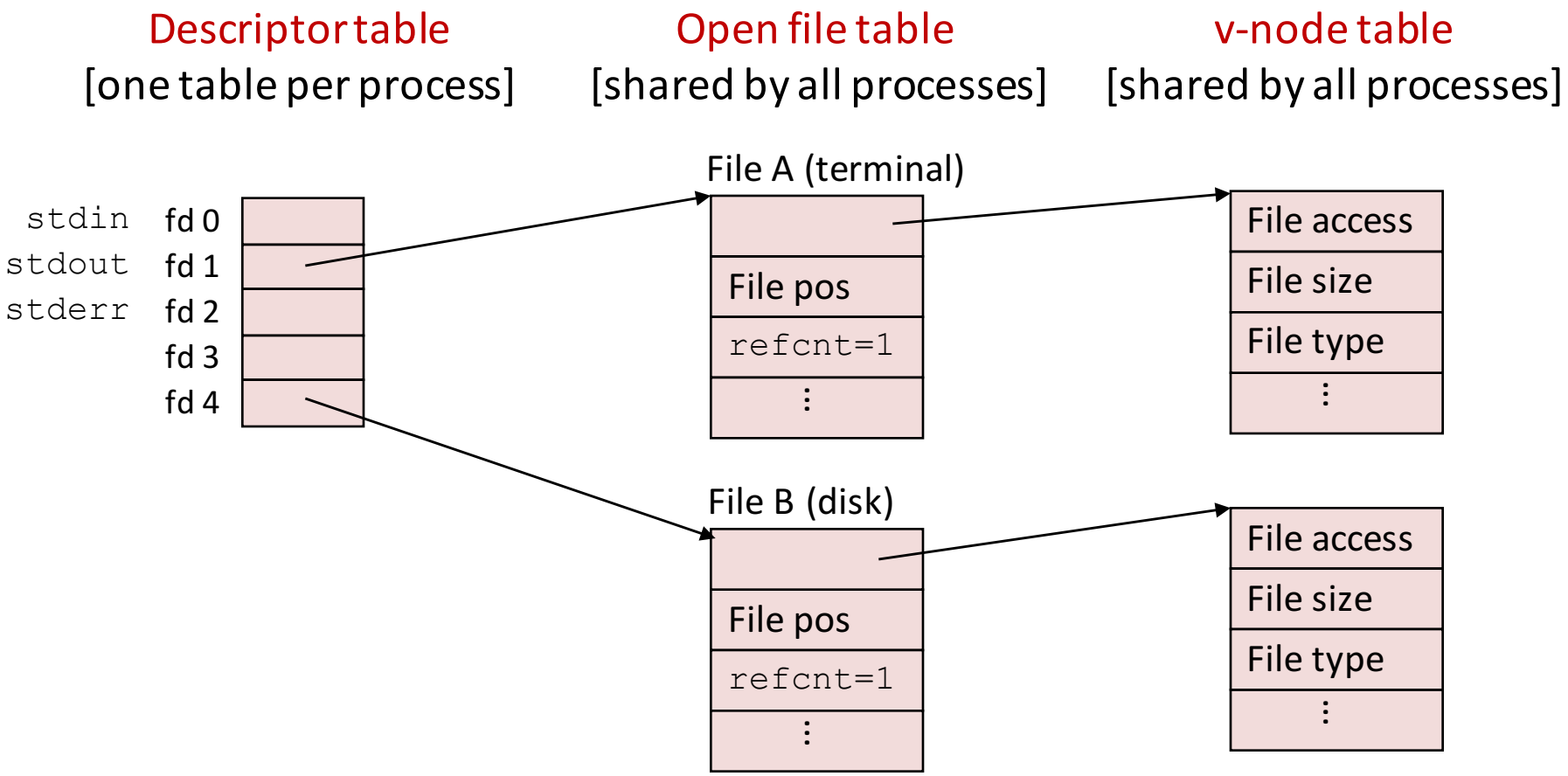
- Two distinct descriptors sharing the same disk file through two distinct open file table entries
  - E.g., Calling **`open`** twice with the same **`filename`** argument



How Processes Share Files: Fork()

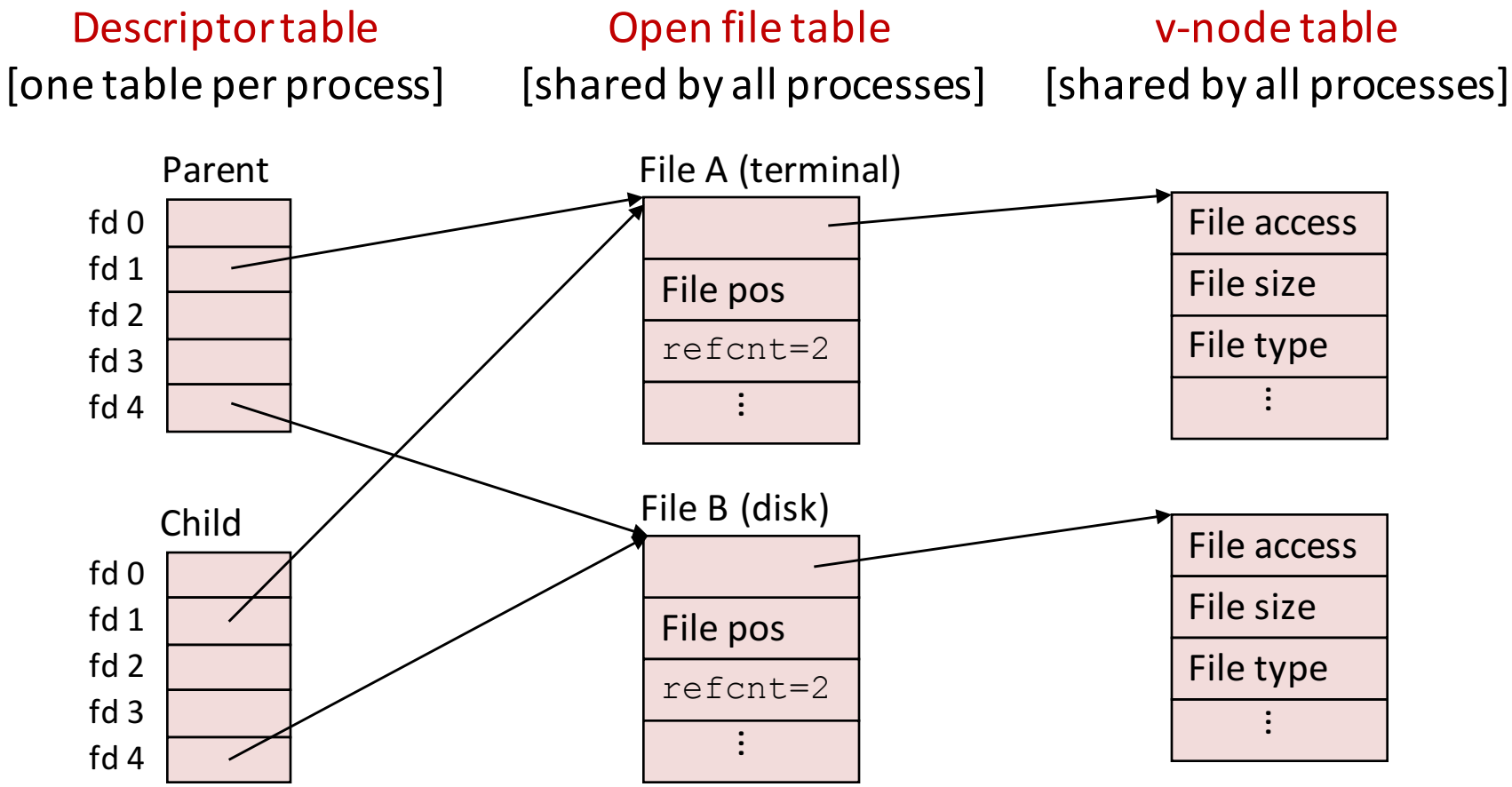
- A child process inherits its parent's open files

- *Before* fork() call:



How Processes Share Files: Fork()

- A child process inherits its parent's open files
- *After* fork():
  - Child's table same as parent's, and +1 to each refcnt



Pipes

- From the command line:

```
$ cat hello.txt | program
```

- We use the pipe operator | to communicate between two running processes. So, how do we do this between processes?

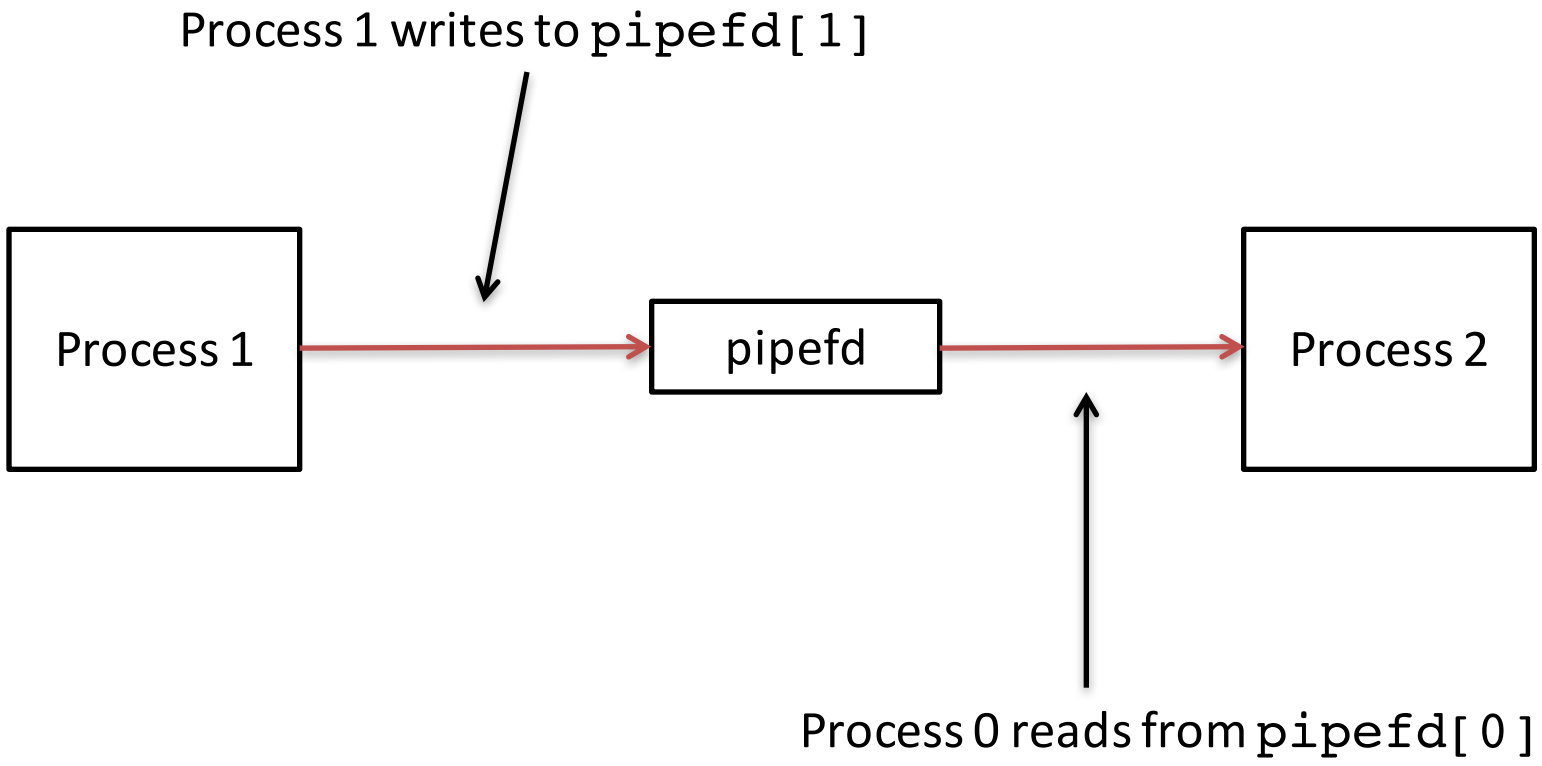
pipe system call

- `int pipe(int pipefd[2])`

  The **pipe** system call will create a “pipe” between two running processes.

  The pipe system call will open two “files”:
  - `pipefd[0]` for reading
  - `pipefd[1]` for writing

Pipes in Action



09_pipes.c