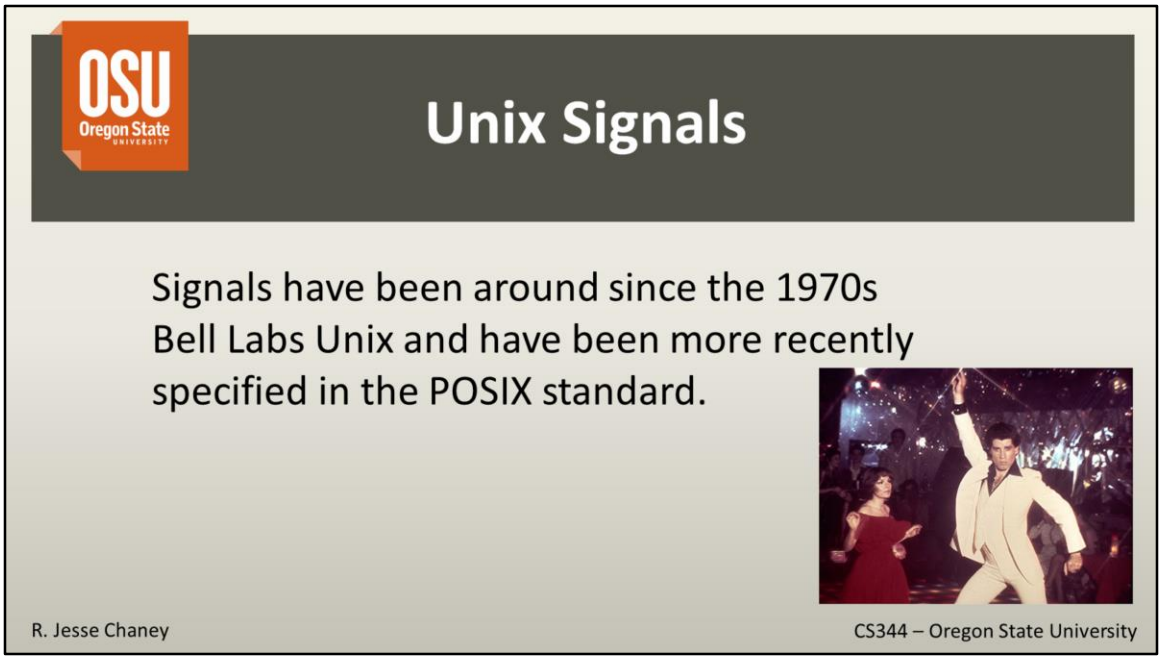


We are now FINALLY going to get into some inter-process communication. We will start with a simple and limited form of IPC called signals.

A signal is an asynchronous notification sent to a process or to a specific thread within the same process in order to notify it that some sort of event has occurred. Signals have been around since the 1970s Bell Labs Unix and have been more recently specified in the POSIX standard.

When a signal is sent, the operating system interrupts the target process's normal flow of execution to deliver the signal. Execution can be interrupted during any non-atomic instruction (or function). If the process has previously registered a **signal handler**, that routine is executed. Otherwise, the default signal handler is executed.

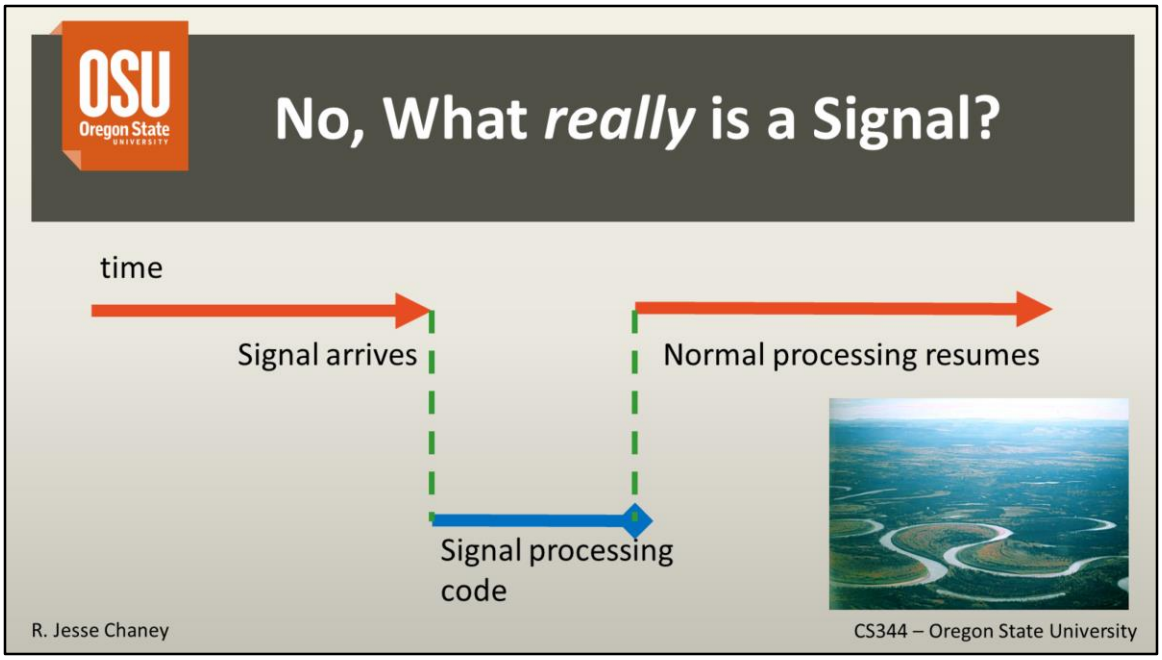
We know/remember what atomic operations are, they complete without interruption (adding deposits of $200 and $300 to your bank account).

A signal is an asynchronous notification sent to a process or to a specific thread within the same process in order to notify it that some sort of event has occurred. Blaa blaa blaa.

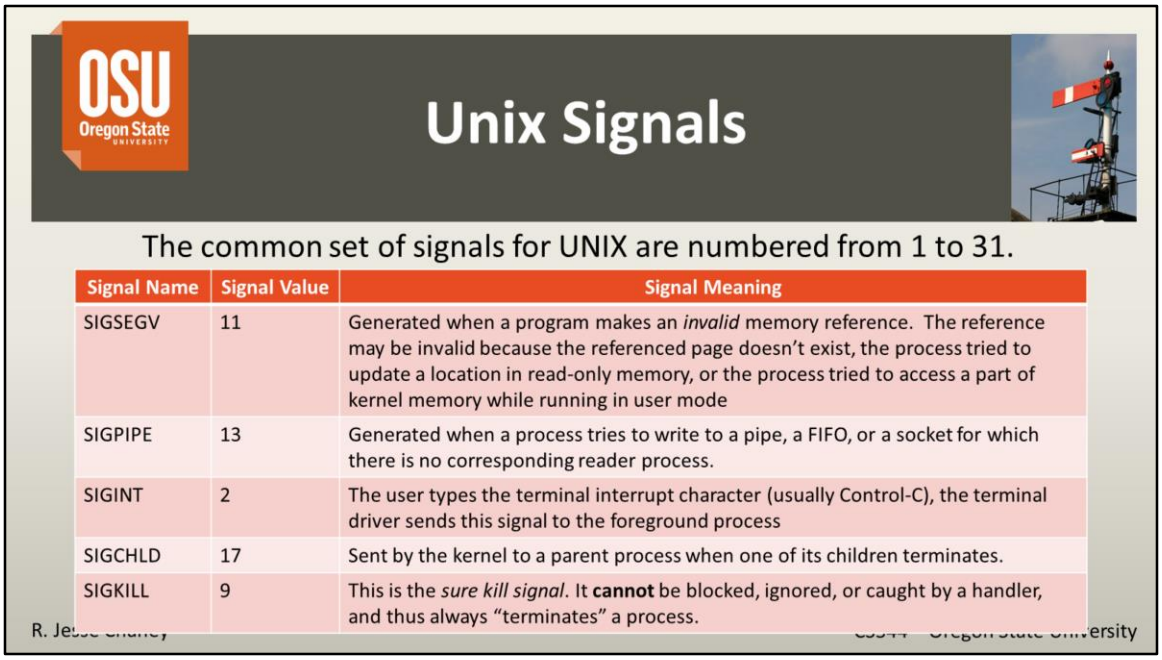
Imagine you are a process (easier for some of us to imagine that). You are clipping along getting stuff done. On either side of you sits a segmentation fault or worse. Then, out of the blue, someone comes along a punches you in the shoulder. You really don't know by whom you were punched, just that you were punched. If you've setup some special kinds of handlers, you can recover from the punch (or most punches) and continue on your merry way. When your process receives the signal, it must alter it path of execution to service the signal. It can then (optionally) return to what it was doing before the signal arrived (or at least try to).

In the case of our process, the punch could have been a hardware exception (divide by zero (always bad), or trying to access out-of-bounds memory). The signal could have been sent either internally (by your self, sort of a personal slap in the face) or from another process. There are also some software events that can generate signals: timers, file descriptor becomes ready, …

A signal is also often called a **software interrupt**.



Here is what your process execution path over time may look like in your code.



There are some signals outside that range and I'll mention them briefly, but they are not covered in this class.

It is very rare that you use a signal by its number. More often you'll use a symbolic name such as SIGTERM, SIGKILL, and SIGCHLD to indicate the signals. Just to make things easier, the actual numbers used for each signal vary across implementations, it is these symbolic names that are **always** used in programs. On many (most) UNIX implementations, there are overlapping and have duplicate names.

The set of signals that are outside the 1-31 are the realtime signals. The realtime signals have some significant differences from the regular set of signals. The realtime signals are numbers from 34 to 64. Unless I specifically point out that were are talking about the realtime signals, we will be talking about the regular set of signals 1-31.

You can send the 0 signal, but it has a very different meaning, which we'll get to next week.

The signal SIGSEGV is a very popular one. It often visits students at the worst

possible time, just before an assignment is due and the student waited longer than he/she should have to start the assignment.

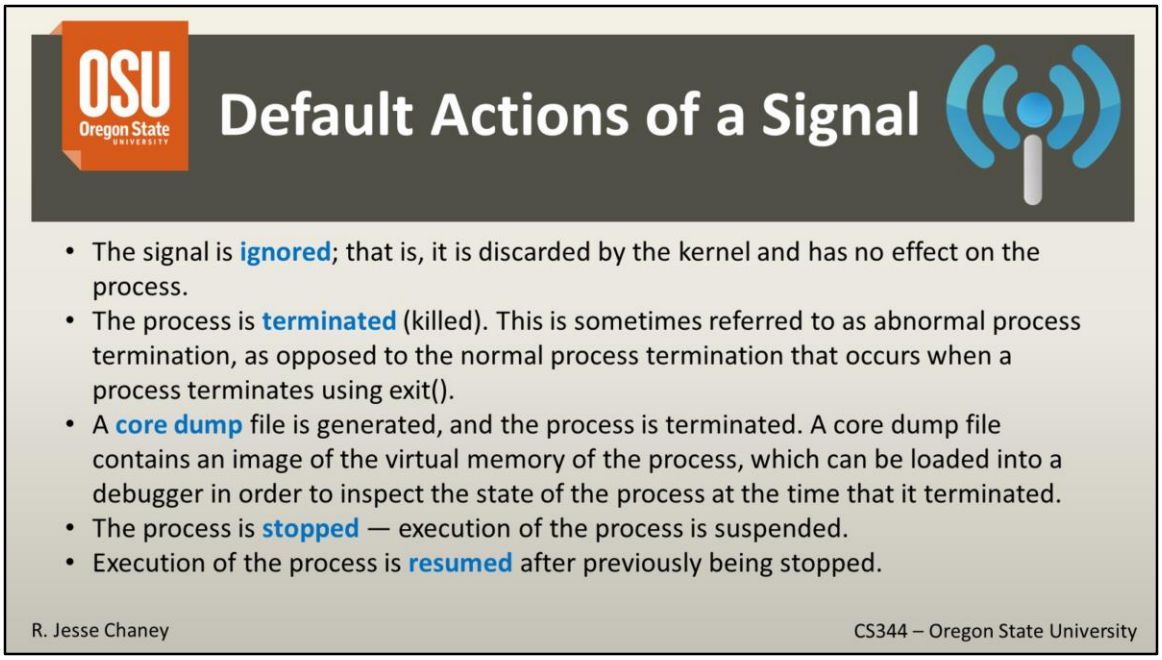
You may start seeing SIGPIPE next week and the week after.

As mighty as the SIGKILL signal may be, there are some processes that even it cannot destroy.

There are also a couple of signals for which there is no designated meaning. They are for specifically set aside for inter-process communication: SIGUSR1 and SIGUSR2. I have an example that makes use of these signals. We'll look at it later in the week.

You can get a complete list of the available signals by typing the `kill –l` command line into your shell.

Even though I show the value in the table above and `kill –l` shows the value, you **REALLY** don't want to use the numeric value in your code. Use the symbolic name.



- The signal is ignored; that is, it is discarded by the kernel and has no effect on the process. (**The process never even knows that it occurred**.)
- The process is terminated (killed). This is sometimes referred to as abnormal process termination, as opposed to the normal process termination that occurs when a process terminates using exit().
- A core dump file is generated, and the process is terminated. A core dump file contains an image of the virtual memory of the process, which can be loaded into a debugger in order to inspect the state of the process at the time that it terminated.
- The process is stopped—execution of the process is suspended. This is like a **control-Z**
- Execution of the process is resumed after previously being stopped. This is like the **fg** command.

Change Actions of a Signal

A program can set one of the following dispositions for a signal:

- The **default action should occur**. This is useful to undo an earlier change of the disposition of the signal to something other than its default.
- The **signal is ignored**. This is useful for a signal whose default action would be to terminate the process.
- A **signal handler is executed**.

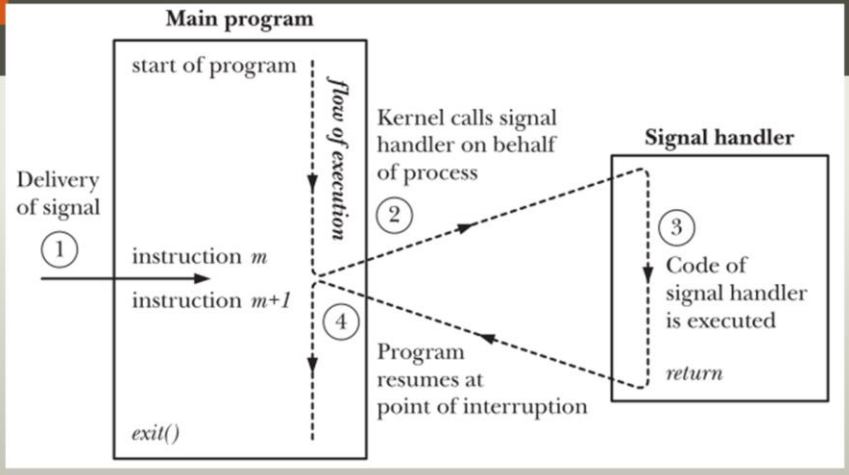
R. Jesse Chaney CS344 – Oregon State University

A signal handler is a function, written by the programmer, that performs appropriate tasks in response to the delivery of a signal.

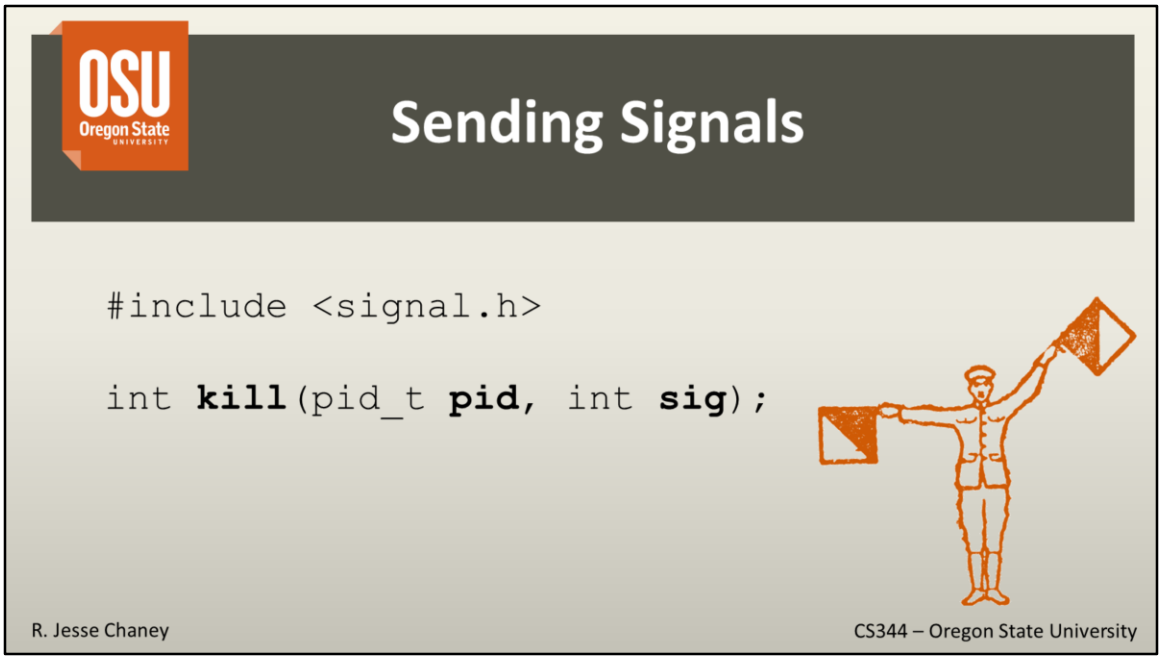
For example, the shell has a handler for the SIGINT signal (generated by the interrupt character, Control-C) that causes it to stop what it is currently doing and return control to the main input loop, so that the user is once more presented with the shell prompt.

Notifying the kernel that a handler function should be invoked is usually referred to as installing or establishing a signal handler. When a signal handler is invoked in response to the delivery of a signal, we say that the **signal has been handled or, synonymously, caught**.

Signal Handlers



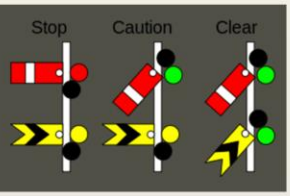
This is figure 20-1 from the book. I think it does a very nice job of showing how a signal interrupts the flow in a process, jumps to the side to handle the signal, and then returns to the previous flow.



This is the common way to send a signal to a process. There are other ways, but this is the most common. This is the same as the kill command you use from the shell.

The term kill was chosen because the default action of most of the signals that were available on early UNIX implementations was to terminate the process.

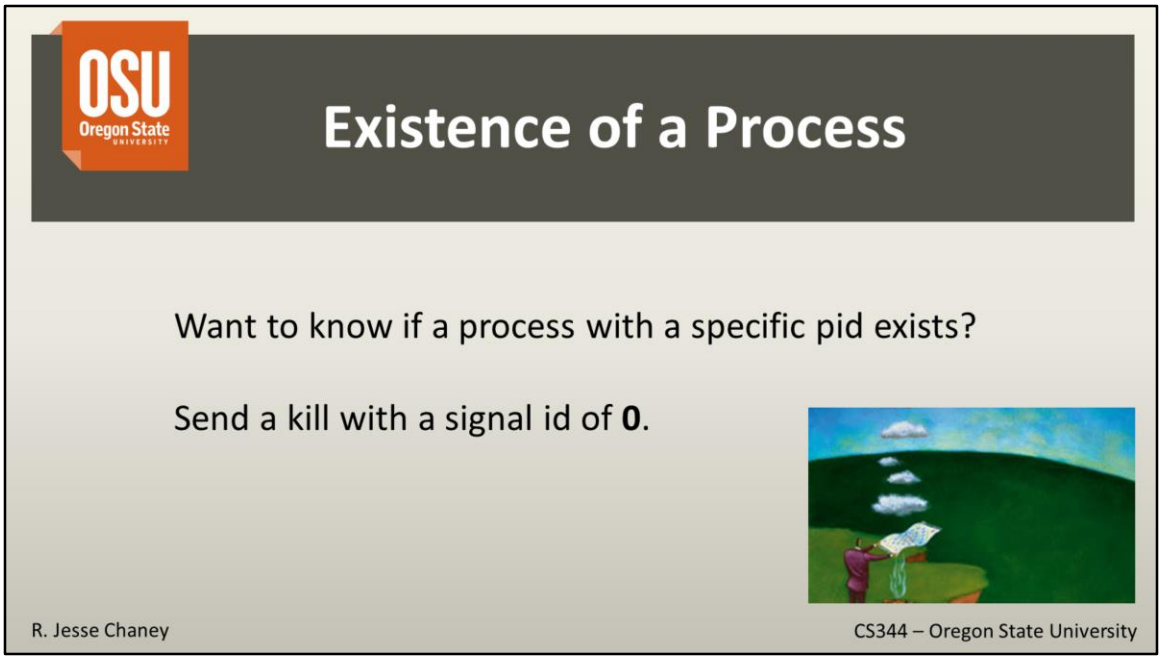
Kill and pids



- If **pid is greater than 0**, the signal is sent to the process with the process ID specified by pid.
- If **pid equals 0**, the signal is sent to every process in the same process group as the calling process, including the calling process itself.
- If **pid is less than –1**, the signal is sent to all of the processes in the process group whose ID equals the absolute value of pid.
- If **pid equals –1**, the signal is sent to every process for which the calling process has permission to send a signal, except init (pid = 1) and the calling process.

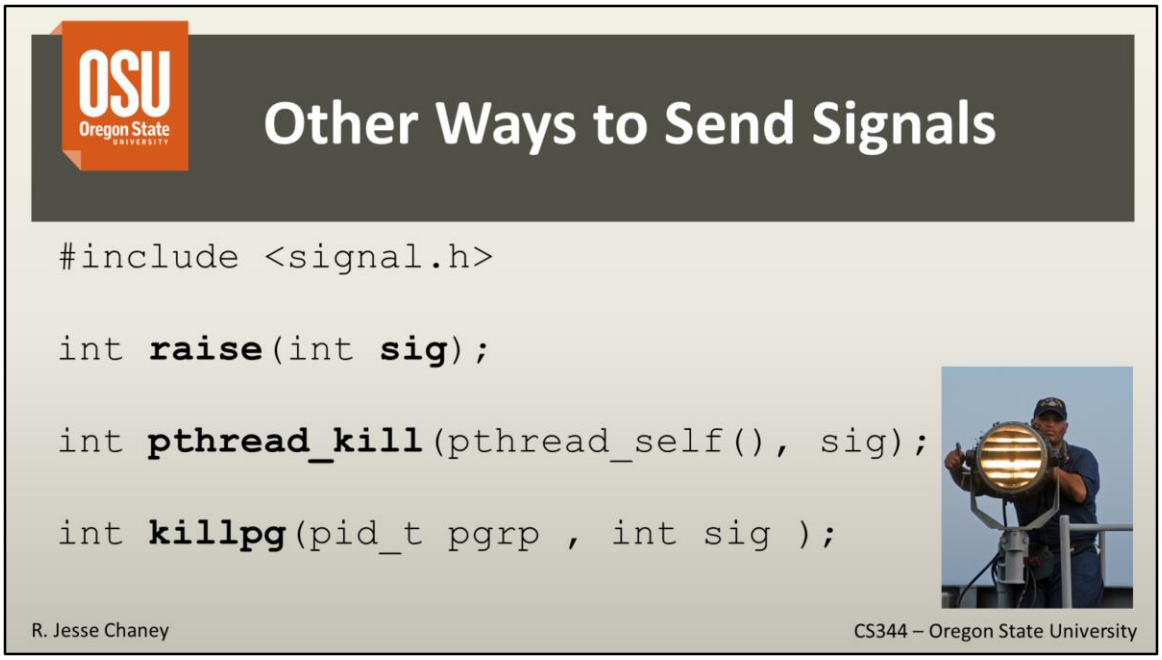
R. Jesse Chaney CS344 – Oregon State University

There are a few special cases for using kill. The most common is to specify the pid. But here are some special cases.



If the sig argument is specified as 0 (**the so-called null signal**), then no signal is sent. Instead, kill() merely performs error checking to see if the process can be signaled. Read another way, this means we can use the null signal to test if a process with a specific process ID exists.

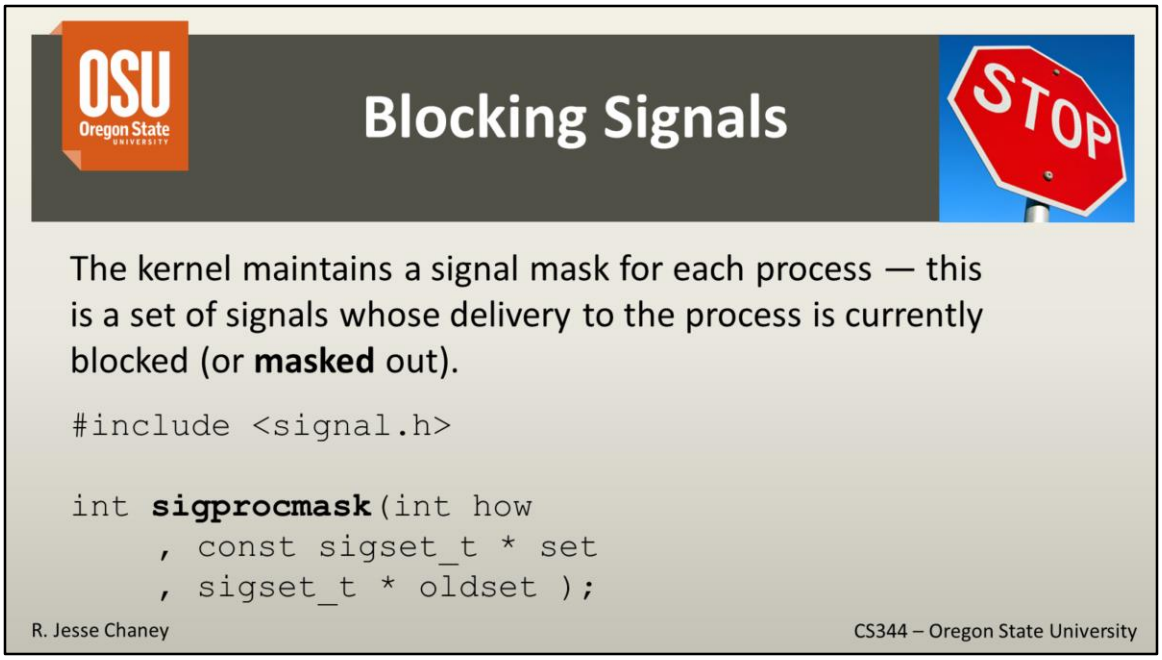
Remember that the list of normal signals is 1 to 31.



The raise() call sends a signal to the process itself. As we'll see from the sample code, this is not really the same as kill(getpid(), sig).

The signal will be delivered to the specific thread. We'll cover threads in a couple weeks.

The killpg() function sends a signal to all of the members of a process group.



All these signals arriving asynchronously can be mighty pesky.

For each process, the kernel maintains a signal mask — a set of signals whose delivery to the process is currently blocked. If a signal that is blocked is sent to a process, delivery of that signal is delayed until it is unblocked by being removed from the process signal mask.

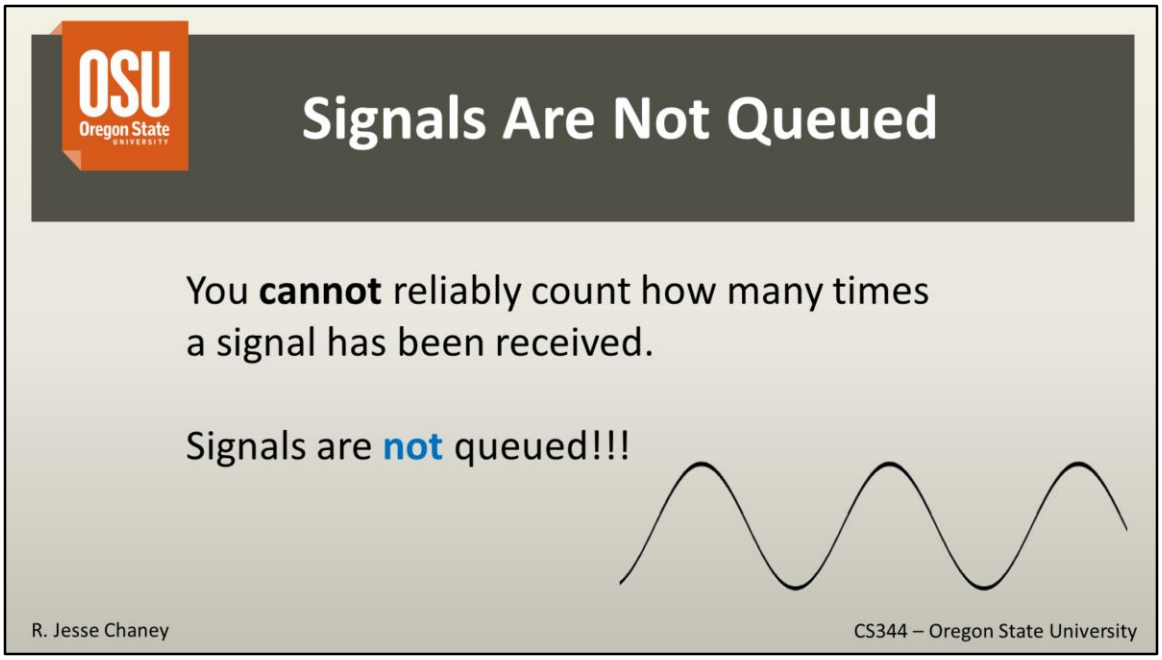
Pending Signals

If a process receives a signal that it is currently blocked, that signal is added to the process' set of **pending** signals.

```
#include <signal.h>

int sigpending(sigset_t * set );
```

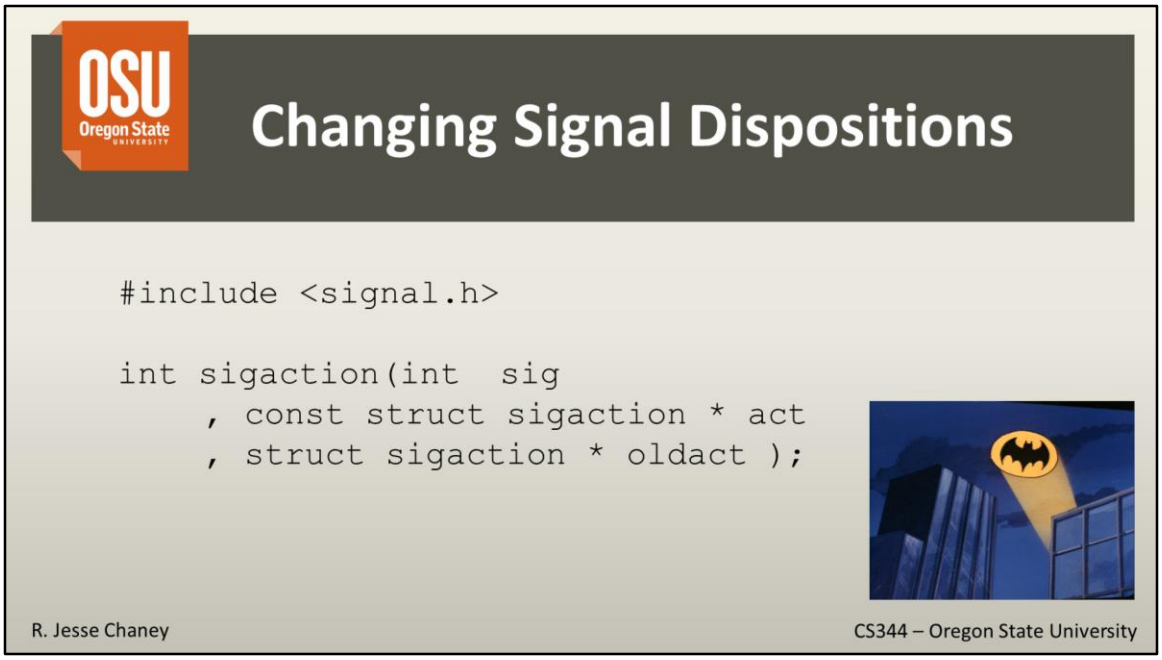
If a process receives a signal that it is currently blocking, that signal is added to the process's set of pending signals. When (and if) the signal is later unblocked, it is then delivered to the process. To determine which signals are pending for a process, we can call sigpending().



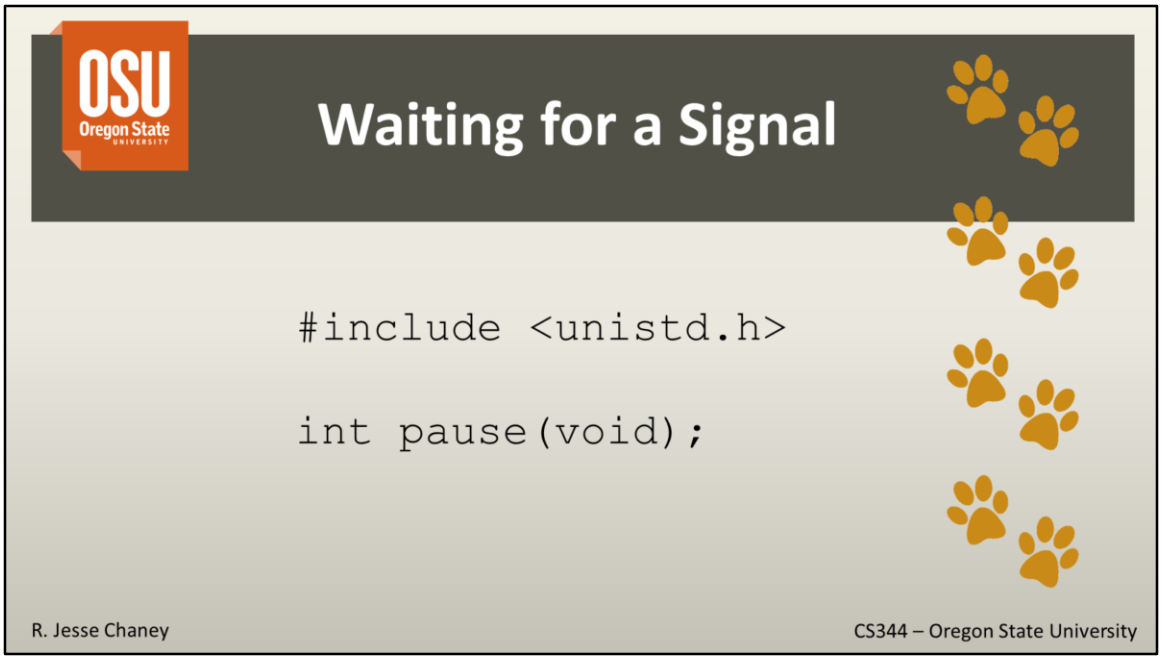
The set of pending signals is only a mask; it indicates whether or not a signal has occurred, but not how many times it has occurred. In other words, if the same signal is generated multiple times while it is blocked, then it is recorded in the set of pending signals, and later delivered, just once.

There is an exception to this (realtime signals), but that is not a topic we cover for this class.

I have some code that will demonstrate this.



The **sigaction**() system call is an alternative to **signal**() for setting the disposition of a signal. Although sigaction() is somewhat more complex to use than signal(), in return it provides greater flexibility. In particular, sigaction() allows us to retrieve the disposition of a signal without changing it, and to set various attributes controlling precisely what happens when a signal handler is invoked. Additionally, as we'll elaborate in Section 22.7, sigaction() is more portable than signal() when establishing a signal handler.



Calling pause() suspends execution of the process until the call is interrupted by a signal handler (or until an unhandled signal terminates the process).

The pause command is like our 4 pawed friends, patient.

Now, let's go through some code!