Signals

CISC 3350 — CUNY Brooklyn College
Lecture Notes

Signals: Motivation

How does Linux know what to do with a running process when:

- A number in the program is divided by 0?
- You press `Ctrl+C`?
- The process attempts to access a memory address that is hasn't allocated?
- Its child process terminates or stops?

The behavior of the OS in these cases is controlled through **signals** and the way *you*, the system programmer, configure them!

Signals

Taken from Maslosoft.com

Signals are software interrupts (= **traps**) that provide a way for handling asynchronous events, such as those listed on the previous slide.

Not only do the events occur asynchronously (e.g., a user can press `Ctrl+C` during the execution of any statement,) but signals are also handled asynchronously: code keeps executing until the OS addresses the signal at a convenient moment for it.

- Example: Sometimes, when you press `Ctrl+C`, the program doesn't quit right away: it might take a few seconds for the OS to process that request.

POSIX has standardized signal handling, which Linux implementations follow.

Most serious applications interact with signals, and you too may be required to work with them at some point in your career (e.g., handling program termination signals.)

Signal Concepts

- When a signal is issued, we say that it is **raised** (or **sent** or **generated**.)
- The kernel then **stores** the signal until it can **deliver** it to the target process.
- When it's free to do so, the kernel **handles** the signal.

The kernel will take one of the following 3 possible actions when handling a signal:

1. **Ignore the signal.** However, there are 2 signals that cannot be ignored: `SIGKILL` and `SIGSTOP`, since the system administrator must have the power to kill/stop processes (Think of a cyber attack that the sysadmin must stop no matter what.)
2. **Catch and handle the signal.** The kernel pauses the process's current execution and jumps to a registered function (called a **handler**) and runs it. Once the process returns from the handler, it jumps back to the instruction that was executing when the signal was caught. `SIGKILL` and `SIGSTOP` cannot be caught.
3. **Perform the default action.** Each signal has its own default action, often being to terminate (exit) the process, as is the case with `SIGKILL`.

Signal Concepts

- Every signal has a symbolic name that starts with the prefix `SIG`. Examples:
  - `SIGINT`: raised when the user presses `Ctrl+C`.
  - `SIGTERM`: raised when a process calls the `exit()` function.
  - `SIGABRT`: raised when a process calls the `abort()` function.
- These signals are all defined in `<signal.h>` using preprocessor definitions that represent positive integers, e.g., `#define SIGHUP 1`.
- However, the name-to-integer mapping for signals depends on the implementation of every Linux machine (though some are fixed, such as `SIGKILL`, which is always `9`). Therefore, a portable program should not use integer values (that is, use: `SIGHUP` instead of `1`, for example.)
- There are (at least) 31 signals supported by all Linux distributions, with integer values `1` to `31`, but only a handful of them are used by most programs. There is no signal with the value `0`, which is known generically as the "null signal".
- The command below will display a list of signals supported by the Linux system you use:

```
$ kill -l
```

Signal Supported by Linux & Their Default Action: A-Z List

Table 10-1. Signals

Signal	Description	Default action
SIGABRT	Sent by `abort()`	Terminate with core dump
SIGALRM	Sent by `alarm()`	Terminate
SIGBUS	Hardware or alignment error	Terminate with core dump
SIGCHLD	Child has terminated	Ignored
SIGCONT	Process has continued after being stopped	Ignored
SIGFPE	Arithmetic exception	Terminate with core dump
SIGHUP	Process's controlling terminal was closed (most frequently, the user logged out)	Terminate
SIGILL	Process tried to execute an illegal instruction	Terminate with core dump
SIGINT	User generated the interrupt character (Ctrl-C)	Terminate
SIGIO	Asynchronous I/O event	Terminate[a]
SIGKILL	Uncatchable process termination	Terminate
SIGPIPE	Process wrote to a pipe but there are no readers	Terminate
SIGPROF	Profiling timer expired	Terminate
SIGPWR	Power failure	Terminate
SIGQUIT	User generated the quit character (Ctrl-\)	Terminate with core dump
SIGSEGV	Memory access violation	Terminate with core dump
SIGSTKFLT	Coprocessor stack fault	Terminate[b]
SIGSTOP	Suspends execution of the process	Stop
SIGSYS	Process tried to execute an invalid system call	Terminate with core dump
SIGTERM	Catchable process termination	Terminate
SIGTRAP	Break point encountered	Terminate with core dump
SIGTSTP	User generated the suspend character (Ctrl-Z)	Stop
SIGTTIN	Background process read from controlling terminal	Stop
SIGTTOU	Background process wrote to controlling terminal	Stop
SIGURG	Urgent I/O pending	Ignored
SIGUSR1	Process-defined signal	Terminate
SIGUSR2	Process-defined signal	Terminate
SIGVTALRM	Generated by `setitimer()` when called with the `ITIMER_VIRTUAL` flag	Terminate
SIGWINCH	Size of controlling terminal window changed	Ignored
SIGXCPU	Processor resource limits were exceeded	Terminate with core dump
SIGXFSZ	File resource limits were exceeded	Terminate with core dump

[a] The behavior on other Unix systems, such as BSD, is to ignore this signal.

[b] The Linux kernel no longer generates this signal; it remains only for backward compatibility.

All the 31 signals and their default actions. "Table 10-1" (pages 335-336), Love, Robert. *Linux System Programming: Talking Directly to the Kernel and C Library*. 2013.

Explanations of All 31 Signals

Note: Signals below that are underlined can be sent to more than one process.

- **SIGABRT**: the `abort()` function sends this signal to the process that calls it. The default action is for the process to terminate and generate a core dump file.
- **SIGALRM**: the `alarm()` and `setitimer()` functions send this signal to the process that calls them after a specified amount of time elapses. The default action is to terminate the process.
- **SIGBUS**: the kernel raises this signal when the process incurs a hardware fault other than memory protection (= `SIGSEGV`). The default action is for the process to terminate and generate a core dump file.
- **SIGCHLD**: whenever a process terminates or stops, the kernel sends this signal to its parent process. It is ignored by default, but you can catch it by calling `wait()`.
- **SIGCONT**: the kernel sends this signal to a process when it is resumed after being stopped. It is ignored by default but is commonly used by terminals and editors to refresh a screen (e.g., a terminal) that wasn't used for a while.

Explanations of All 31 Signals

- **`SIGFPE`**: represents any arithmetic exception/error, not only related to floating-point operation errors (e.g., division by 0.) The default action is to terminate the process.
- **`SIGHUP`**: the kernel sends this signal to the session leader process whenever the session's terminal disconnects. The kernel also sends it to each process in the foreground process group when the session leader terminates. The default action is to terminate, as this signal means that the user has logged out.
- **`SIGILL`**: the kernel sends the signal when a process attempts to execute an illegal machine instruction. The default action is to terminate the process.
- **`SIGINT`**: it is sent to all the foreground group's processes when the user enters the interrupt character (`Ctrl+C`). The default behavior is to terminate.
- **`SIGIO`**: this signal is sent when a BSD-style async I/O event is generated. This style of I/O is rarely used on Linux. By default, it terminates the process.

Explanations of All 31 Signals

- **`SIGKILL`**: it is sent by calling the `kill()` syscall; it exists to let the sysadmin kill a process unconditionally. This signal cannot be caught/ignored, and it always results in terminating the process.
- **`SIGPIPE`**: the kernel raises it when a process writes to a pipe, but the reader has terminated. The default action is to terminate.
- **`SIGPROF`**: it is generated when a timer using `setitimer()` expires. The default action is to terminate the process.
- **`SIGPWR`**: it is system-dependent. On Linux, it represents a low-battery condition. A power daemon sends this signal to `init`, which cleans up and shuts the system down right before the battery is completely empty.
- **`SIGQUIT`**: the kernel raises this signal for all the foreground group's processes when the user enters the terminal quit character (usually `Ctrl+\`). The default action is termination and creating a core dump.
- **`SIGSEGV`**: segmentation-violation is set to a process when it attempts an invalid memory access. The default action is to terminate the process.

Explanations of All 31 Signals

- **SIGSTOP**: it is sent only by `kill()`; It unconditionally stops a process and cannot be caught or ignored.
- **SIGSYS**: the kernel sends this signal to a process when it attempts to invoke an invalid system call, such as when a binary created on a newer system executes on an older system that doesn't support a syscall.
- **SIGTERM**: it is sent only by `kill()`; it allows a user to gracefully terminate a process (the default action.)
- **SIGTRAP**: the kernel sends this to a process when it reaches a breakpoint. Generally, debuggers catch this signal, and other processes ignore it.
- **SIGTSTP**: the kernel sends this to all the foreground group's processes when the user enters the suspend character (`Ctrl+Z`).
- **SIGTTIN**: it is sent to a process in the background when it attempts to read from its controlling terminal. The default action is to stop the process.

Explanations of All 31 Signals

- **SIGTTOU**: it is sent to a process in the background when it attempts to write to its controlling terminal. The default action is to stop the process.
- **SIGURG**: the kernel sends this to a process when out-of-band (OOB) data has arrived on a socket. OOB data is a topic beyond the scope of this book. It is ignored by default.
- **SIGUSR1 and SIGUSR2**: user-defined signals: the kernel never raises them, but you can configure them as you wish. Processes may use them for any purpose. The default action is termination.
- **SIGVTALRM**: `setitimer()` sends this when a timer created with the `ITIMER_VIRTUAL` flag expires (decrements only when the process is executing.)
- **SIGWINCH**: the kernel raises this for all the foreground group's processes when the size of the terminal window changes. By default, processes ignore it.

Explanations of All 31 Signals

- **`SIGXCPU`**: the kernel raises this when a process exceeds its soft processor limit. The kernel will continue to raise this once per sec until the process exits or exceeds its hard processor limit, upon which the kernel sends `SIGKILL`.
- **`SIGXFSZ`**: the kernel raises this when a process exceeds its file size limit. The default action is termination.

Now, let's see how to choose which action (from those on slide 4) should be taken when a certain signal is issued!

Two functions that we study in this chapter, `signal()` and `sigaction()`, are used to change the behavior of Linux when a signal is received: we first cover `signal()`, the simplest of the two.

In short, `signal()` receives 2 arguments: the signal code (e.g., `SIGINT`) and the action that should be applied whenever this signal is caught. The action could be a function that you write inside your program, which Linux will run whenever the signal is received.

Basic Signal Management: `signal()`

<table>
<tr><td colspan="2">Name: <code>signal()</code></td><td>System call or function: System Call</td><td>Links: Online Manual | Course Packet</td></tr>
<tr><td colspan="4">What it does: sets Linux's action when the signal <code>signo</code> is received to <code>handler</code>.</td></tr>
<tr><td colspan="4">What libraries you must include: <code>#include <signal.h> // Defines signal().</code></td></tr>
<tr><td colspan="4">Syntax: <code>sighandler_t signal (int signo, sighandler_t handler);</code></td></tr>
<tr><td colspan="4">Description of arguments: <code>signo</code>: a signal code such as <code>SIGINT</code>, <code>SIGTERM</code>, etc.
<code>handler</code>: one of the following:
1. <code>SIG_DFL</code>: sets the behavior of the signal <code>signo</code> to its default.
2. <code>SIG_IGN</code>: ignores the signal given by <code>signo</code>.
3. The name of a function of the form: <code>void my_handler (int signo);</code></td></tr>
<tr><td colspan="2" rowspan="3">What type it returns:
<code>sighandler_t</code></td><td colspan="2">On success: the previous behavior of the signal, which is either a pointer to a handler, <code>SIG_DFL</code>, or <code>SIG_IGN</code>.</td></tr>
<tr><td colspan="2">On failure: <code>SIG_ERR</code></td></tr>
<tr><td colspan="2">If failure, does it set <code>errno</code>? No</td></tr>
<tr><td colspan="4">Quick example: <code>if (signal (SIGINT, my_function) == SIG_ERR) /* error */</code></td></tr>
<tr><td colspan="4">Other variations: N/A</td></tr>
</table>

Waiting for (Any) Signals

Name: `pause()`	**System call or function:** System Call	**Links:** Online Manual | Course Packet
What it does: puts a process to sleep until it receives a signal that either causes the invocation of a signal-catching function or terminates the process. If the kernel raises an ignored signal, the syscall won't return, and the process will keep sleeping.		
What libraries you must include: `#include <unistd.h> // Defines pause().`		
Syntax: `int pause (void);`		
Description of arguments: – This system call doesn't accept any argument –		
What type it returns: `int`	**On success:** returns `-1` when a signal that is either handled or terminates the process is caught, and sets `errno` to `EINTR`.	
	On failure: – This system call doesn't fail –	
	If failure, does it set `errno`? Yes	
Quick example: `pause();`		
Other variations: N/A		

This function is useful for debugging and writing demonstrative code.

division_by_0.c: Full C Program


```c
/* A program that raises SIGFPE when we divide a
 *    number by 0.
 *
 *    Miriam Briskman, 4/26/2023
 *    CISC 3350, Brooklyn College
 *    Licensed under CC BY-NC 4.0
 */

#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <signal.h>

// Handler for the SIGFPE signal::
```

handling_sigint.c: Another Full C Program


```c
// A program that handles SIGINT by printing a
//    message, and then terminating the program.

// This program is taken from Linux System Programming:
//    Talking Directly to the Kernel and C Library,
//    2nd Edition, by Love. ISNB: 978-1-44933953-1,
//    page 342.

#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <signal.h>

// Handler function for SIGINT:
```

more_signals.c: Another Full C Program


```
// A program that registers the same handler for
//    SIGTERM and SIGINT. It also resets the
//    behavior for SIGPROF to the default (which
//    is to terminate the process) and ignores
//    SIGHUP (which would otherwise terminate
//    the process.)

// This program is taken from Linux System Programming:
//    Talking Directly to the Kernel and C Library,
//    2nd Edition, by Love. ISNB: 978-1-44933953-1,
//    pages 343-344.

#include <stdlib.h>
#include <stdio.h>
```

Signals

Any questions so far?

Signal Inheritance and `fork()`/`exec()`

- On `fork()`, a child process inherits the signal actions of its parent.
- The child copies all the registered actions (ignore, default, handles) for each signal from its parent, but pending signals (= signals that were issued but haven't yet been handled by the OS) are not inherited.
- Afterwards, when one of the `exec()` functions is called to create a new process image, all signals are set to their default actions unless the parent is ignoring them, which will make the child ignore them as well.
- In short, any signal caught (handled) by the process before `exec()` was called is reset to the default action after `exec()` runs, and all other signals remain the same (default or ignored.)

Signal Behavior	After `fork()`	After `exec()`
Ignored	Inherited	Inherited
Default	Inherited	Inherited
Handled	Inherited	Not Inherited
Pending Signals	Not Inherited	Inherited

Getting a Text Description (String) of a Signal

Sometimes, we need to use a text description of a signal number.

- For example, when we want to print to the screen what the signal `SIGHUP` exactly means.

We can do this by using an array called `sys_siglist`, which holds the names of all signals supported by the system, indexed by signal number:

```
printf ("Caught %s\n", sys_siglist[signo]);
```

`sys_siglist` is available once you include `<string.h>`in your program.

There are also 2 functions that you can use to get or print this description out: `psignal()` and `strsignal()`, which we present on the next slides. It is recommended that you use these functions instead of `sys_siglist` since `sys_siglist` is deprecated in recent C library versions.

Printing a Text Description (String) of a Signal

<table>
<tr><td>Name: <code>psignal()</code></td><td>System call or function: Function</td><td>Links: Online Manual | Course Packet</td></tr>
<tr><td colspan="3">What it does: directly prints to <code>stderr</code> the signal description of <code>signo</code>, preceded by the message <code>msg</code> that you indicate.</td></tr>
<tr><td colspan="3">What libraries you must include: <code>#include <signal.h> // Defines psignal().</code></td></tr>
<tr><td colspan="3">Syntax: <code>void psignal (int signo, const char *msg);</code></td></tr>
<tr><td colspan="3">Description of arguments: <code>signo</code>: a signal code such as <code>SIGINT</code>, <code>SIGTERM</code>, etc.
<code>msg</code>: a char array containing the message you want to print before the description is printed.</td></tr>
<tr><td rowspan="3">What type it returns: <code>void</code></td><td colspan="2">On success: – This function doesn't return anything –</td></tr>
<tr><td colspan="2">On failure: – This function doesn't fail –</td></tr>
<tr><td colspan="2">If failure, does it set <code>errno</code>? No</td></tr>
<tr><td colspan="3">Quick example:
<code>psignal (SIGHUP, "Here is the description");</code>
<code>// Will print the following to stderr:</code>
<code>// Here is the description: Hangup</code></td></tr>
<tr><td colspan="3">Other variations: <code>strsignal()</code></td></tr>
</table>

Getting a Text Description (String) of a Signal

<table>
<tr><td>Name: <code>strsignal()</code></td><td>System call or function: Function</td><td>Links: Online Manual | Course Packet</td></tr>
<tr><td colspan="3">What it does: returns a <code>char</code> pointer to a string containing the signal description.</td></tr>
<tr><td colspan="3">What libraries you must include:
<code>#define _GNU_SOURCE</code>
<code>#include <string.h> // Defines strsignal().</code></td></tr>
<tr><td colspan="3">Syntax: <code>char * strsignal (int signo);</code></td></tr>
<tr><td colspan="3">Description of arguments: <code>signo</code>: a signal code such as <code>SIGINT</code>, <code>SIGTERM</code>, etc.</td></tr>
<tr><td rowspan="3">What type it returns: <code>char *</code></td><td colspan="2">On success: A string (char array) containing the description of a signal.</td></tr>
<tr><td colspan="2">On failure: an unknown signal message if the signal number is invalid.</td></tr>
<tr><td colspan="2">If failure, does it set <code>errno</code>? No</td></tr>
<tr><td colspan="3">Quick example:
<code>printf ("SIGHUP means: %s\n", strsignal(SIGHUP));</code>
<code>// Will print the following to stderr:</code>
<code>// SIGHUP means: Hangup</code></td></tr>
<tr><td colspan="3">Other variations: <code>psignal()</code></td></tr>
</table>

Sending a Signal

Name: `kill()`	**System call or function:** System Call	**Links:** Online Manual | Course Packet

What it does: sends a signal `signo` from the current process to another process with the ID `pid`. If you pass `0` for `pid`, `signo` is sent to every process in the caller's process group. If you pass `-1`, it is sent to every process the caller is permitted to send a signal to, except itself and `init`. Finally, if you pass a number less than `-1`, it is sent to the process group with ID `|pid|`.

What libraries you must include:
```
#include <signal.h> // Defines kill().
```

Syntax:
```
int kill (pid_t pid, int signo);
```

Description of arguments: `pid`: the ID of the process to which you want to send a signal.
`signo`: a signal code such as `SIGINT`, `SIGTERM`, etc., of the signal you want to send.

What type it returns: `int`

- **On success:** `0`
- **On failure:** `-1`
- **If failure, does it set `errno`?** Yes

Quick example:
```
if ((result = kill (1722, SIGHUP)) == -1) // Same as the command: $ kill -HUP 1722
    perror ("kill");
```

Other variations: `raise()`, `killpg()`

Permissions to Send Signals

- A process needs proper permissions to send a signal to another process.
- A process with the `CAP_KILL` capability can send a signal to any process. Usually, such a process is run by the `root` user.
- Otherwise, if a user doesn't have the `CAP_KILL` capability, this user's processes can send a signal only to a process that this user owns.
- If you pass `0` (the null signal) for `signo`, the call doesn't send a signal, but it still performs error checking that `kill()` usually performs. This is useful to test whether a process has suitable permissions to send a signal to another process or not.
  - Example (checking if we can send a signal to process `1722` without actually sending a signal:)

```c
int ret = kill (1722, 0);
if (ret == -1)
    /* We lack permissions: perhaps print an error message. */
else
    /* We have permission: we can proceed to actually send a signal. */
```

Sending a Signal to Itself

<table>
<tr><td colspan="2">Name: <code>raise()</code></td><td>System call or function: Function</td><td>Links: Online Manual | Course Packet</td></tr>
<tr><td colspan="4">What it does: sends a signal <code>signo</code> from the current process to itself.
Question: How does a process call the <code>kill()</code> system call to send a signal to itself? Fill in the blank:
<code>int result = kill (_____, SIGHUP);</code>
[Hint: What function does a process call to find its own <code>pid</code>?]</td></tr>
<tr><td colspan="4">What libraries you must include: <code>#include <signal.h> // Defines raise().</code></td></tr>
<tr><td colspan="4">Syntax: <code>int raise (int signo);</code></td></tr>
<tr><td colspan="4">Description of arguments: <code>signo</code>: a signal code such as <code>SIGINT</code>, <code>SIGTERM</code>, etc., of the signal you want to send.</td></tr>
<tr><td rowspan="3" colspan="2">What type it returns: <code>int</code></td><td colspan="2">On success: <code>0</code></td></tr>
<tr><td colspan="2">On failure: a non-zero integer.</td></tr>
<tr><td colspan="2">If failure, does it set <code>errno</code>? No</td></tr>
<tr><td colspan="4">Quick example:
<pre>if ((result = raise (SIGHUP))) // Checks if result != 0
    perror ("raise");</pre></td></tr>
<tr><td colspan="4">Other variations: <code>kill()</code>, <code>killpg()</code></td></tr>
</table>

Another Way to Send a Signal to a Process Group

Name: `killpg()`	**System call or function:** System Call	**Links:** Online Manual | Course Packet

What it does: sends a signal `signo` from the current process to all the processes in the process group with ID `pgrp`.
Question: How does a process call the `kill()` system call to send a signal to the process group with ID `pgrp`? Fill in the blank: `int result = kill (_______, SIGHUP);`
[***Hint***: What should the 1st argument be equal to when sending signals to process groups?]

What libraries you must include: `#include <signal.h> // Defines killpg().`

Syntax: `int killpg (int pgrp, int signo);`

Description of arguments: `pgrp`: the ID of the process group to which you want to send a signal.
`signo`: a signal code such as `SIGINT`, `SIGTERM`, etc., of the signal you want to send.

What type it returns: `int`	**On success:** `0`
	On failure: `-1`
	If failure, does it set `errno`? Yes

Quick example: `if ((result = killpg (532, SIGHUP)) == -1)  perror ("killpg");`

Other variations: `kill()`, `raise()`

issuing_signals.c: C Program That Uses `kill()`


```c
/* A program that creates a child process and
 *    sends it the SIGINT signal via kill().
 *    The child handles the SIGINT signal by
 *    calling the write() system call, which is
 *    reentrant. Finally, the parent sends a
 *    signal to itself using raise().
 *
 *    Miriam Briskman, 4/30/2023
 *    CISC 3350, Brooklyn College
 *    Licensed under CC BY-NC 4.0
 */

#include <unistd.h>   // write().
#include <sys/wait.h> // waitpid().
```

Another Way to Send a Signal to a Process Group

Any questions so far?

Reentrancy

- When the kernel raises a signal, a process could be executing code anywhere, which may not be interrupted. E.g., it could be in the middle of handling another signal.
- Thus, the signal handler must be careful about what data it touches/changes.
  - A signal handler should never modify global variables.
  - Sometimes, it's necessary to block the delivery of signals to keep data safe.
- What do we do if the process is calling syscalls, like writing to files or allocating memory, and a signal handler tries to do the same? → We must discuss this idea of **reentrancy**.
- A **reentrant function** is one that is safe to call from within itself. Basically, such a function can only manipulate stack-allocated data or data provided to it by the calling function. It must not manipulate static data or call any non-reentrant function.
- Some functions are not reentrant, and a handler calling the same function will lead to chaos.
  - A function calling itself forever might lead to infinite recursion.
- Signal handlers, therefore, must call only those functions that are reentrant.

Reentrant-Guaranteed Functions

POSIX has a list of functions that are guaranteed to be reentrant and signal-safe on all compliant platforms.

The functions shown here are guaranteed by Linux to be reentrant:

abort()	accept()	access()	read()	readlink()	recv()
aio_error()	aio_return()	aio_suspend()	recvfrom()	recvmsg()	rename()
alarm()	bind()	cfgetispeed()	rmdir()	select()	sem_post()
cfgetospeed()	cfsetispeed()	cfsetospeed()	send()	sendmsg()	sendto()
chdir()	chmod()	chown()	setgid()	setpgid()	setsid()
clock_gettime()	close()	connect()	setsockopt()	setuid()	shutdown()
creat()	dup()	dup2()	sigaction()	sigaddset()	sigdelset()
execle()	execve()	_Exit()	sigemptyset()	sigfillset()	sigismember()
_exit()	fchmod()	fchown()	signal()	sigpause()	sigpending()
fcntl()	fdatasync()	fork()	sigprocmask()	sigqueue()	sigset()
fpathconf()	fstat()	fsync()	sigsuspend()	sleep()	socket()
ftruncate()	getegid()	geteuid()	socketpair()	stat()	symlink()
getgid()	getgroups()	getpeername()	sysconf()	tcdrain()	tcflow()
getpgrp()	getpid()	getppid()	tcflush()	tcgetattr()	tcgetpgrp()
getsockname()	getsockopt()	getuid()	tcsendbreak()	tcsetattr()	tcsetpgrp()
kill()	link()	listen()	time()	timer_getoverrun()	timer_gettime()
lseek()	lstat()	mkdir()	timer_settime()	times()	umask()
mkfifo()	open()	pathconf()	uname()	unlink()	utime()
pause()	pipe()	poll()	wait()	waitpid()	write()
posix_trace_event()	pselect()	raise()			

Functions guaranteed by Linux to be reentrant. "Table 10-3" (pages 349-350), Love, Robert. *Linux System Programming: Talking Directly to the Kernel and C Library*. 2013.

Reentrant-Guaranteed Functions

Any questions?

Signal Sets

- Some signal configurations can be applied to sets of signals, all at once.
- We, therefore, show below how to work with sets of signals. We first create a set variable: `sigset_t set;` and then call any of the following functions on it by passing `&set` as an argument:

```
int sigemptyset (sigset_t *set);                 // Empties the signal set 'set'.
int sigfillset (sigset_t *set);                  // Includes all the signals in 'set'.
int sigaddset (sigset_t *set, int signo);        // Adds 'signo' to 'set'.
int sigdelset (sigset_t *set, int signo);        // Removes 'signo' from 'set'.
int sigismember (const sigset_t *set, int signo); // Checks if 'signo' is in 'set'.
```

- You can use any of these functions if you `#include <signal.h>` in your program.
- The first 4 functions return `0` on success and `-1` on failure. On failure, they set `errno`.
- `sigismember()` returns `1` if `signo` is in the signal set given by `set`, `0` if it's not, and `-1` on failure (and sets `errno`).

More Signal Set Functions

- Linux adds several nonstandard functions:

```
int sigisemptyset (sigset_t *set);
int sigorset (sigset_t *dest, sigset_t *left, sigset_t *right);
int sigandset (sigset_t *dest, sigset_t *left, sigset_t *right);
```

- You can use any of these functions if you `#include <signal.h>` and `#define _GNU_SOURCE` in your program.
- Descriptions:
  - `sigisemptyset()` returns `1` if the `set` is empty and `0` otherwise.
  - `sigorset()` sets `dest` to the union (binary `OR`) of sets `left` and `right`.
  - `sigandset()` sets `dest` to the intersection (binary `AND`) of sets `left` and `right`. Both return `0` on success and `-1` on error.
- Avoid using these if you want your program/application to comply with the POSIX standard (since these 3 functions aren't considered standard.)

Blocking Signals

<table>
<tr><td colspan="2">Name: <code>sigprocmask()</code></td><td>System call or function: System Call</td><td>Links: Online Manual | Course Packet</td></tr>
<tr><td colspan="4">What it does: blocks (= masks) or unblocks the signals in <code>set</code> from being delivered to the process.</td></tr>
<tr><td colspan="4">What libraries you must include: <code>#include <signal.h> // Defines sigprocmask().</code></td></tr>
<tr><td colspan="4">Syntax: <code>int sigprocmask (int how, const sigset_t *set, sigset_t *oldset);</code></td></tr>
<tr><td colspan="4">Description of arguments: <code>how</code>: must be one of the following constants:
1. <code>SIG_SETMASK</code>: the signals in <code>set</code> become blocked, while all the signal types not in <code>set</code> become unblocked.
2. <code>SIG_BLOCK</code>: the signals in <code>set</code> are added to the process's existing blocked signals (no signal gets unblocked.)
3. <code>SIG_UNBLOCK</code>: the signals in <code>set</code> are removed from the process's blocked signals.
<code>set</code>: a set of signals.
<code>oldset</code>: an empty set of signals that, if not <code>NULL</code>, will contain the old signal mask after <code>sigprocmask()</code> finishes running.</td></tr>
<tr><td colspan="2" rowspan="3">What type it returns: <code>int</code></td><td colspan="2">On success: <code>0</code></td></tr>
<tr><td colspan="2">On failure: <code>-1</code></td></tr>
<tr><td colspan="2">If failure, does it set <code>errno</code>? Yes</td></tr>
<tr><td colspan="4">Quick example: <code>if (sigprocmask (SIG_SETMASK, &my_set, &oldset) == -1) /* error */</code></td></tr>
<tr><td colspan="4">Other variations: N/A</td></tr>
</table>

Blocking Signals

A few notes on `sigprocmask()`:

- Using `sigprocmask()` is important for parts of the program that you want to prevent from being interrupted, such as critical regions, which can be protected by suspending the delivery of signals (blocking them).
- Any signals that are raised while being blocked are not handled until unblocked (by calling `sigprocmask()` and passing `SIG_UNBLOCK` to it.)
- A process may block any number of signals, as you choose.
- The set of signals blocked by a process is called its **signal mask**. So, instead of saying "the set of signals blocked by the process", we can say "the signal mask of the process".
- If `oldset` is not `NULL`, it will contain the previous signal set after `sigprocmask()` returns.
- If `set` is `NULL`, the function ignores `how` and doesn't change the signal mask, but it does place the current mask in `oldset` .
- Blocking `SIGKILL` or `SIGSTOP` is not allowed.

Getting the Pending Signals

Name: `sigpending()`	**System call or function:** System Call	**Links:** Online Manual | Course Packet
What it does: places the set of pending signals in `set`. A **pending signal** (in our current discussion) is a signal that the kernel wants to send to the process, but the process currently blocks this signal type. When the signal is later unblocked, the kernel will then deliver this signal to the process.		
What libraries you must include: `#include <signal.h> // Defines sigpending().`		
Syntax: `int sigpending (sigset_t *set);`		
Description of arguments: `set`: a set of signals that will be filled-up with the pending signals after `sigpending()` finishes running.		
What type it returns: `int`	**On success:** `0`	
	On failure: `-1`	
	If failure, does it set `errno`? Yes	
Quick example: `sigset_t pending_signals_set;` `if (sigpending (&pending_signals_set) == -1) /* error */`		
Other variations: N/A		

Waiting for Pending Signals

<table>
<tr><td colspan="2">Name: <code>sigsuspend()</code></td><td>System call or function: System Call</td><td>Links: Online Manual | Course Packet</td></tr>
<tr><td colspan="4">What it does: temporarily replaces the signal mask of the process with the signal set given by <code>set</code> and then pauses the process's execution until delivery of a signal whose action is to invoke a signal handler or to terminate a process.</td></tr>
<tr><td colspan="4">What libraries you must include: <code>#include <signal.h> // Defines sigsuspend().</code></td></tr>
<tr><td colspan="4">Syntax: <code>int sigsuspend (const sigset_t *set);</code></td></tr>
<tr><td colspan="4">Description of arguments: <code>set</code>: a set of signals that will become the termporarily blocked signals.</td></tr>
<tr><td rowspan="3">What type it returns: <code>int</code></td><td colspan="3">On success: If a handled signal is raised, it returns <code>-1</code>, and sets and sets <code>errno</code> to <code>EINTR</code>.</td></tr>
<tr><td colspan="3">On failure: <code>-1</code> and sets <code>errno</code> to <code>EFAULT</code>.</td></tr>
<tr><td colspan="3">If failure, does it set <code>errno</code>? Yes</td></tr>
<tr><td colspan="4">Quick example:
<pre>errno = 0;
sigsuspend (&pending_signals_set);
if (errno == EFAULT) /* Error: set is an invalid pointer. */
else if (errno == EINTR) /* Not an error: a handled signal was received! */</pre></td></tr>
<tr><td colspan="4">Other variations: N/A</td></tr>
</table>

Waiting for Pending Signals

A few notes on `sigsuspend()`:

- If a signal that is set to terminate the process is delivered, `sigsuspend()` doesn't return.
- If a signal that is set to be handled is delivered, `sigsuspend()` returns `-1` after the signal handler returns and sets `errno` to `EINTR`.
- If set is an invalid pointer, `sigsuspend()` returns `-1`, too, but `errno` is set to `EFAULT`.
- The `sigsuspend()` system call is usually used in the following scenario to prevent the delivery of a signal during the execution of a critical region:
  - The process first uses `sigprocmask()` to block a set of signals since it enters a critical region. It saves the old mask in `oldset`.
  - After exiting the critical region, during which there might be signals that arrived but were previously blocked, the process calls `sigsuspend()` using `oldset` to wait for and finally receive those pending signals.

Waiting for Pending Signals

Any questions so far?

Advanced Signal Management: `sigaction()`

The `signal()` function is very basic. The POSIX standard also mentions `sigaction()`, which provides much greater abilities:

<table>
<tr><td colspan="2">Name: <code>sigaction()</code></td><td>System call or function: System Call</td><td>Links: Online Manual | Course Packet</td></tr>
<tr><td colspan="4">What it does: changes the behavior of the signal <code>signo</code> to <code>act</code>, and stores the previous behavior in <code>oldact</code>.</td></tr>
<tr><td colspan="4">What libraries you must include: <code>#include <signal.h> // Defines sigaction().</code></td></tr>
<tr><td colspan="4">Syntax: <code>int sigaction (int signo, const struct sigaction *act, struct sigaction *oldact);</code></td></tr>
<tr><td colspan="4">Description of arguments: <code>signo</code>: the signal type whose behavior we want to change.
<code>act</code>: the new behavior we want to set (see more on the <code>sigaction</code> structure on the next slide.)
<code>oldact</code>: will keep the previous behavior for the <code>signo</code> signal.</td></tr>
<tr><td rowspan="3">What type it returns: <code>int</code></td><td colspan="3">On success: <code>0</code></td></tr>
<tr><td colspan="3">On failure: <code>-1</code></td></tr>
<tr><td colspan="3">If failure, does it set <code>errno</code>? Yes</td></tr>
<tr><td colspan="4">Quick example: <code>if (sigaction (SIGTERM, &action, &oldaction) == -1) /* error */</code></td></tr>
<tr><td colspan="4">Other variations: N/A</td></tr>
</table>

More Info on `sigaction()`

- If you pass `NULL` for `act`, the behavior for `signo` won't change, but the `oldact` structure will then store the current behavior for `signo`.
- You can pass `NULL` for `oldact` if you don't want to store/get or don't care about the previous behavior for `signo`.
- The `sigaction` structure allows fine-grained control over signals and is defined in `<sys/signal.h>`, which is already included in `<signal.h>` (so, there's no need to `#include <sys/signal.h>` in your code.) Here are its data fields:

```
struct sigaction {
    void (*sa_handler)(int);                          // 1. signal handler or action
    void (*sa_sigaction)(int, siginfo_t *, void *); // 2. alternative handler or action
    sigset_t sa_mask;                                 // 3. set of signals to block
    int sa_flags;                                     // 4. flags
    void (*sa_restorer)(void);                        // 5. obsolete and non-POSIX: don't use.
};
```

More Info on the `sigaction` Structure

1. The `sa_handler` field specifies the function to run on receiving the signal, or `SIG_DFL` (default), or `SIG_IGN` (ignore).
2. If you set `sa_flags` to `SA_SIGINFO`, then `sa_sigaction` is used instead of `sa_handler`. The function you assign to `sa_sigaction` has the prototype:

```
void my_handler (int signo, siginfo_t *si, void *ucontext);
```

   - More information on the **`siginfo_t`** structure is on the next slide.
3. The `sa_mask` field provides a set of signals the system should block while the signal handler is executing. The signal currently being handled is also blocked (if it's in `sa_mask`,) unless the `SA_NODEFER` flag is set in `sa_flags`. However, you cannot block `SIGKILL` or `SIGSTOP`, as we already mentioned.
4. The `sa_flags` field is a bitwise `OR`ing of zero, one, or more flags that change the handling of signal `signo` (see the full list of flags at this link.)
   - `SA_SIGINFO`: uses `sa_sigaction` instead of `sa_sighandler`.
5. The `sa_restorer` field is obsolete and no longer used in Linux: don't use it in your programs.

More Info on the `siginfo_t` Structure

The `siginfo_t` structure provides a lot of information to the signal handler, `sa_sigaction`:

```
typedef struct siginfo_t {
    int      si_signo;  // Signal number ------------------> Same as the first argument to sa_sigaction.
    int      si_errno;  // Error value --------------------> If nonzero, contains the error code associated with this signal.
    int      si_code;   // Signal code --------------------> An explanation of why/where the process received the signal.
    pid_t    si_pid;    // Sending process's PID ----------> For SIGCHLD, the pid of the process that terminated.
    uid_t    si_uid;    // Sending process's real UID -----> For SIGCHLD, the owner uid of the process that terminated.
    int      si_status; // Exit value or signal -----------> For SIGCHLD, the exit status of the process that terminated.
    clock_t  si_utime;  // User time consumed -------------> For SIGCHLD, the user time consumed of the process that terminated.
    clock_t  si_stime;  // System time consumed -----------> For SIGCHLD, the system time consumed of the process that terminated.
    sigval   si_value;  // Signal's payload value ---------> A union of si_int and si_ptr (see fields below.)
    int      si_int;    // POSIX.1b signal ----------------> Payload (data you wish to pass) in a form of an integer.
    void     *si_ptr;   // POSIX.1b signal ----------------> Payload (data you wish to pass) in a form of a void pointer.
    void     *si_addr;  // Memory location causeing fault --> For SIGBUS, SIGFPE, SIGILL, SIGSEGV, and SIGTRAP only.
    long     si_band;   // Band event ---------------------> For SIGPOLL (pollable events), out-of-band/priority info for si_fd.
    int      si_fd;     // File descriptor ----------------> For SIGPOLL, the fd for the file whose operation completed.
};
```

What Does `si_code` Contain?

- The `si_code` field indicates the cause of the signal.
  - For user-sent signals, it indicates *how* the signal was sent.
  - For kernel-sent signals, it indicates *why* the signal was sent.
- Choices of `si_code` that are valid for any signal (see this man page):
  - `SI_ASYNCIO` (async I/O finished), `SI_KERNEL` (signal raised by kernel), `SI_QUEUE`, `SI_TIMER`, `SI_SIGIO`, `SI_USER` (signal was sent by `kill()` or `raise()`.)
- Choices of `si_code` that are valid only for `SIGBUS` (bad memory access), indicating what hardware error type occurred:
  - `BUS_ADRALN` (alignment error), `BUS_ADDRERR` (invalid addr), `BUS_OBJERR` (object-specific hardware error)
- Choices of `si_code` that are valid only for `SIGCHLD`, identifying what the child did:
  - `CLD_CONTINUED`, `CLD_DUMPED`, `CLD_EXITED`, `CLD_KILLED`, `CLD_STOPPED`, `CLD_TRAPPED`
- Choices of `si_code` that are valid only for `SIGFPE` (arithmetic errors):
  - `FPE_FLTDIV` (division by 0), `FPE_FLTOVF` (number overflow), `FPE_FLTINV`, `FPE_FLTRES`, `FPE_FLTSUB`, etc.
- This page also describes choices of `si_code` that are valid only for the signals `SIGILL`, `SIGPOLL`, `SIGSEGV`, and `SIGTRAP`.

sigaction_example.c: Full C Program


```c
/* A C program that ignores the SIGINT signal using
 *    the sigaction() function.
 *
 *    Miriam Briskman, 4/30/2023
 *    CISC 3350, Brooklyn College
 *    Licensed under CC BY-NC 4.0
 */

#include <signal.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main()
```

sigaction_example.c: Full C Program

Any questions so far?

Sending a Signal with a Payload

The following function allows a process to send a signal with a payload (= attached value):

<table>
<tr><td>Name: <code>sigqueue()</code></td><td colspan="2">System call or function: System Call</td><td>Links: Online Manual | Course Packet</td></tr>
<tr><td colspan="4">What it does: It works similarly to kill(): the signal <code>signo</code> is queued to the process or process group identified by <code>pid</code>, and the value <code>value</code> is sent along with the signal.</td></tr>
<tr><td colspan="4">What libraries you must include: <code>#include <signal.h> // Defines sigqueue().</code></td></tr>
<tr><td colspan="4">Syntax: <code>int sigqueue (pid_t pid, int signo, const union sigval value);</code></td></tr>
<tr><td colspan="4">Description of arguments:<code>pid</code>: the process ID or process group ID to whom we send the signal.
<code>signo</code>: the signal that we want to send. You can pass <code>NULL</code> to test signal-sending permissions.
<code>value</code>: a union structure (see the next slide) containing the value we want to pass.</td></tr>
<tr><td rowspan="3">What type it returns: <code>int</code></td><td colspan="3">On success: <code>0</code></td></tr>
<tr><td colspan="3">On failure: <code>-1</code></td></tr>
<tr><td colspan="3">If failure, does it set <code>errno</code>? Yes</td></tr>
<tr><td colspan="4">Quick example: <code>if (sigaction (SIGTERM, &action, &oldaction) == -1) /* error */</code></td></tr>
<tr><td colspan="4">Other variations: N/A</td></tr>
</table>

Sending a Signal with a Payload

- With `sa_sigaction` as the handler, the `si_value` field is an optional payload (data) passed from the signal generator to the signal receiver. This is why we can pass values.
- The `value` variable is a union (see more on unions at this tutorial page.)

```
union sigval {
    int  sival_int;
    void *sival_ptr;
};
```

- The example below sends process `1722` with a `SIGUSR2` (user-defined) signal with an integer payload of `404`:

```
sigval value;
value.sival_int = 404;
if (sigqueue (1722, SIGUSR2, value)) /* Error */
```

- If process `1722` handles `SIGUSR2` with an `SA_SIGINFO` handler, it will set `signo` to `SIGUSR2`, `si->si_int` to `404`, and `si->si_code` to `SI_QUEUE`.

payload_sending.c: Full C Program


```c
/* A program that creates a child process and
 *    sends it the SIGUSR2 signal with a payload
 *    that the user types via the terminal.
 *
 *    Miriam Briskman, 4/30/2023
 *    CISC 3350, Brooklyn College
 *    Licensed under CC BY-NC 4.0
 */

#include <unistd.h>   // write().
#include <sys/wait.h> // waitpid().
#include <stdio.h>    // perror(), printf(), fprintf().
#include <stdlib.h>   // EXIT_SUCCESS, EXIT_FAILURE.
#include <string.h>   // strlen().
```

payload_sending.c: Full C Program

Any questions?

Signals: Sources

Topic 10: Signals. Briskman, Miriam. *CISC 3350 OER Course Packet.* 2024, CC BY-NC 4.0.
URL: https://www.sci.brooklyn.cuny.edu/~briskman/cisc/3350/CISC_3350_OER_Course_Packet.pdf#topic+10:+signals.

Partly based on the teaching of Professor **Yongqing Xiang** of Brooklyn College.