Lectures 17-18: Signals

kill

What does **kill** do? Now, by now you should be familiar with this useful little command. As mentioned before it has some variants:

```
kill -TERM
kill -STOP
kill -CONT
```

Now the obvious answer to this question, the 'functional' answer is that these functions either end or pause a process. However, what is the 'mechanical' answer to this question. What is **kill** actually doing? How does it work? By what mechanism does **kill** end a process.

The **kill** command works by sending a **signal** to the process. Depending on how the process is designed, it may react in a variety of different ways. By default, most of the times these signals will do what they say they will do, but not always.

Ctrl-C (aka SIGINT)

By now you should probably know that aside from **copy-paste**, Ctrl-C has a very important function within the linux terminal. If you have ever written code with an infinite loop, some helpful person probably showed you how to reclaim your terminal via the Ctrl-C shortcut. Again, what is Ctrl-C actually doing?

Most terminals implement special shortcuts to send signals, in this case Ctrl-C is SIGINT. It stands for Signal-Interrupt which makes some intuitive sense. It interrupts (aka ends) the currently running process in the terminal. Other shortcuts exist such as pausing or sending EOF characters. Not all of these are signals (sending EOF is hardly a signal), but many of them are.

Signals, How they Work

So while I have been dancing around the real question here, I haven't really given a substantive answer to 'how do signals do what they do'. First things first, is to understand where signals are generated and how they are sent to a process. Signals are not generated by the terminal (even though you can invoke them from the terminal). Instead, **signals** are generated by the CPU (or CPU related software). The OS keeps track of two 32-bit integers. Each bit of these integers represents a signal. The fact there are two such integers is because each integer represents a different thing:

- Pending Signals
- Blocked Signals

One note before we continue is that all of this is highly OS dependent (i.e. YMMV).

Pending Signals, Blocked Signals

So let us assume that the user has somehow requested that the OS generate a signal and send it to a process. What happens now? Well, the OS will update the **pending** signals 'list' (it is really just one integer). The OS is in charge of delegating processor resources to processes so now it has the opportunity to check the pending and blocked signals to decide what it will do.

If both the **pending** and **blocked** list are checked for a signal, nothing happens. So, the **blocked** list kind of amounts to a 'ignore signals list'. However, if the signal is **pending** but not **blocked**, it executes the **signal handler** in the code. Now, up until now, y'all haven't been writing any **signal handlers**. So does that mean your code does nothing? Well, there are default handlers for many signals. For example, when you write a C program, SIGINT… kills the program, by default.

Standard Signals

Here are some common ones:

SIGINT	(ctrl-c) Interrupt (i.e. exit)
SIGHUP	Signal 'Hang Up'. Received after losing a remote connection
SIGKILL	A more aggressive version of SIGINT
SIGPIPE	Sent when attempting to write to a pipe which has been closed by the reader (i.e. this character has nowhere to go).

Command Line Signals

(ctrl-c)	SIGINT
(ctrl-z)	SIGSTP: Analogous to pause on a music/video player (rather than stop, despite being equivalent to kill -STOP)
(ctrl-\)	SIGQUIT: Sends a quit signal which causes the process to terminate and dump core

Signal Handling

Signal handling in C is very straight forward and consists of two main parts (actually, quite similar to threads). There is the 'setup' part where you specify when in the code you want the signal handler to become active. Then there is the function which is called by the signal.

```
void handler(int num)
{
      write(STDOUT_FILENO, "Thwarted!\n", 13);
}
int main()
{
      signal(SIGINT, handler);
      while (1)
      {
            printf("Do nothing\n");
      }
}
```

The above is perhaps one of the easiest signal examples. Basically, it is an infinite while loop that will be immune to Ctrl-C. If you ever want to write an annoying program, this is the way to do it. One thing to note is the use of the function **write**, rather than **printf** which you should be more familiar with. You might be wondering why this is the case. Well, it turns out that when you are running a signal handler, you aren't exactly in the same 'processing environment' as you are when you are running normal code (we'll cover this later).

Just some basic rules of Signal Handlers. They have to return **void** and take an **int**. Now given you never write any code that calls a Signal Handler (i.e. there won't be a my_signal_handler(27) anywhere in your code… or shouldn't be) you might be wondering what the **int** is for? How does this bit of code, get its **int**? Well, this **int** will be a reference to the type of **signal** that happened. If you used Ctrl-C it

will be **SIGINT** but it doesn't necessarily have to be the same. One can set up a single signal handler (say that ten times fast) for as many signals as one wants:

```
signal(SIGINT, handler);
signal(SIGFPE, handler);
signal(SIGPIPE, handler);
signal(SIGSTOP, handler);

void handler(int in_signal)
{
      if (in_signal == SIGINT) {}
      if (in_signal == SIGPIPE) {}
      ...
}
```

But why would you?

Things not to do in Signal Handlers

There are a variety of things that signal handlers should definitely not do. So there are a few obvious ones. Do not use **malloc**. If you think about it, a signal handler is a pretty independent bit of code. It does not return anything and barely interacts with the rest of your program. If you **malloc**, you are almost certainly going to cause a memory leak. Worse than this however is what happens when you interrupt an existing **malloc** call to run a signal handler that contains a new **malloc** call (note… it ends badly).

Don't use **printf**. Now this one is a bit more complicated. It amounts to when your signal is called. What if, for example you call **printf** but before it has resolved, you call a signal handler which also calls **printf**. In this case, what is in the **printf** buffer (what is going to be written) and what is in the **printf** settings (how much to write) can go out of sync because you append to the buffer but only update the size. This can cause the wrong thing to be displayed but can also cause the whole program to crash. These functions are not **async-signal-safe**.

There are lists for functions which do meet this criteria (async-signal-safe is a fun google search term). If it isn't on this list, don't use it.

There are other things not to do, things that are kind of 'crazy things'. Things like… altering the signal mask (within a signal handler… why???), changing what a signal does (i.e. reassigning a signal handler inside a signal handler), altering errno…

When Signal Handlers Fail

There are a few signals that can't be handled and attempts to do so will be denied. These are:

- SIGKILL (i.e. kill -kill)
- SIGSTOP (i.e. Ctr-Z)

It may not be immediately obvious why this is the case but intuitively it makes sense that the OS should have final control over what processes can run on it. Otherwise you could have a process that is completely 'unkillable' (what is dead may never die).

User Defined Signals

There is also the possibility of user-defined signals:

- SIGUSR1
- SIGUSR2

They have no predefined behaviour. A good use might be to help two processes 'dock' with one another by telling each other via a signal that they are ready to interact.

Predefined Signal Redirects

Sometimes, you might want to 'go back to what you had before'. So for example, having redirected SIGINT, you want to go back to normal behaviour again. You can!

```
signal(<some_signal>, SIG_DFL)
```

This is akin to a 'factory reset' restoring the signal to whatever it used to do.

There is also the glorious:

```
signal(<some_signal>, SIG_IGN)
```

SIGIGN stands for signal-ignore. It is a bit like writing a signal handler with no code in it. Again, SIGKILL and SIGSTOP cannot be redirected to SIGIGN.

Signalling, more broadly

Now, the whole point of this signal escapade, other than pausing and killing processes is really to address the idea of synchronous and asynchronous behaviour. This is especially important for tasks such as connecting hardware to a computer. When I have a device which is slow to function but I want it to have low latency, asynchronous behaviour will usually benefit me more.

Polling vs Interrupts

In a synchronous system, a CPU will be forced to continually check whether something is connected or not. This is almost always a waste of CPU time. Instead, it is much more efficient if the CPU ticks along doing its own thing when it receives a signal that it can respond to tell it that something has happened.

Hardware Signals

Hardware falls into this category of signals. A hardware interrupt (i.e. the printer is online, this other hardware is connected, someone typed on a key on the keyboard) is a kind of **exception**, a disruption to the normal process flow. Unlike **exceptions** (something went wrong), **interrupts** are usually intentional functionality with the system both sending them and being prepared to receive them in a logical way.

In Unix, a hardware interrupt (i.e. divide by zero) automatically changes the processor context to start executing a kernel exception handler. Some of these events are handled by the kernel. In some exceptions, such as a page fault, the kernel has sufficient information to fully handle the event itself and resume the process's execution.

In some cases, the kernel cannot process intelligently and it must instead **defer** the exception handling operation to the faulting process.

This deferral is achieved via the **signal mechanism**, wherein the kernel sends to the process a signal corresponding to the current exception.

Divide by Zero Example

What happens when a process demands the result of 1/0 from the CPU. Well, usually the program will crash because CPUs think this is a bad idea. Whenever the CPU receives a request for this instruction, it sends a **SIGFPE** (floating point error) to the offending process. By default in C, this results in a core dump (though you could easily overwrite the default behaviour if you… wanted to?). A similar thing happens when a segfault is found.

For this to work, we need to get our head around what the CPU needs to do. This includes:

- Saving the context (program counter, save registers, stack)
- Consult a table of things to do (which interrupt handler to call)
- Resolve the interrupt's default behaviour
- Business as usual (reload registers)

The **Interrupt Service Handler** also has some responsibilities. Firstly it needs to do stuff (after all a signal was just called). It may also have some responsibilities within the code to do something else, or set some flags or exit.

Signal Don'ts

So, you may have noticed there are SIGFPE for escaping divide by zeros and SIGSEGV for catching segmentation faults. Do not get any smart ideas. Yes, when you segfault you get a SIGSEGV. Yes, you could write a handler to catch your SIGSEGV. If this handler does anything more complex than helping you debug why you got your segfault you are making a big error. The reason this is a bad idea is that SIGSEGV redirections usually results in infinite segfault loops. Just triggering a signal handler doesn't remove the original error (i.e. the bad coding) and thus such code often just loops between the code and the segfault handler. It is just a bad idea. Same thing applies to SIGFPE… just sanitize your input.

Signal Timing

Now, you may be thinking. Signals. Easy. Redirect call, signal handler. Do not **malloc**, **printf** or do anything crazy. But…

It gets worse. There are lots of signals. What happens when you have a lot of different signals all triggering at the same time, or perhaps repeatedly? They are, after all, **asynchronous**. Moreover, signals are not **atomic** (one at a time). It is quite possible to trigger one signal midway through handling another. It is even possible to trigger the same signal, halfway through triggering itself…

The **sigprocmask()** function can be used to **block** and **unblock** signals (remember pending vs blocked). So, this can potentially alleviate some of the problem… but only slightly.

Now, returning to **atomic**, it turns out that 'back in the day', signal designed assumed the existence of one signal at a time (which may have been quite reasonable). Unfortunately, this means that as this problem was emergent, there is no clear nor consistent Unix solution to the problem of having multiple signals arriving simultaneously. What are the choices?

What if, while handling SIGX, SIGY arrives?

What if, while handling SIGX, SIGX arrives?

There are no definitive answers, but there are possibilities.

Option #1: Last in, First Out (stack style).

In this logic, whatever the most recently called signal is, it gets resolve first. Think of it as interrupts of interrupts of interrupts.

Option #2: Just ignore it.

In this logic if two signals of the same kind are received, just pretend the others didn't happen. I already know after all.

Option #3: Block and Queue

In this logic, when you receive an extra signal, you block and queue the new one (this is the safest technique). This way you are ready to receive new signals (i.e. if it has triggered again) but you don't interrupt your current signal to deal with it, you finish what you start and are ready to process the signal again.

The Flaws of Blocking

So what if during a signal handler you block a signal. I.e.

```
handle_sigint()
{
	signal(SIGQUIT, SIG_IGN);
	// Stuff
	signal(SIGQUIT, prev_handler);
}
```

Seems pretty good right? While handling SIGINT you can ignore SIGQUIT. Except…

```
handle_sigint()
{
	// What happens here?
	signal(SIGQUIT, SIG_IGN);
	// Stuff
	signal(SIGQUIT, prev_handler);
}
```

Technically we could get a SIGQUIT after SIGINT is triggered but before the signal redirection occurs… as you can see, signals are a bit messy.

Signal Ambiguity

Another issue with signals is that they don't always have clear meanings. SIGFPE can occur for many reasons (divide by zero isn't the only one). Likewise SIGSEGV is about as useful as saying 'your code sucks'.

Sigaction

So given that signals are such a mess, one way to try to avoid this issue is via sigaction.

```
int sigaction(signalnumber, action, prevaction);
```

- **signalnumber:** The signal to redirect
- **action:** Is a struct (type **sigaction**) containing handler etc.
- **prevaction:** Can be **NULL** but basically used to store the previous action (allowing some control)

sigaction contains:

sa_handler: Another struct

- Can be set to a few options
  - IGN: Ignore
  - DLF: Default
  - handler: Custom function

sa_sigaction:

- You can set this to the name of a handler which will get not only the number (i.e. SIGINT) but some extra context for the signal

sa_flags:

- Used to tell system whether you want to use old-style (no info) or new style (with info) you set a bit in sa_flags

sa_mask:

- Used to define the set of signals to block

Moustrap and Mousetrap Problems

In some original implementations, signals worked as follows. You have a handler for signals. As part of that signal handler, you needed to reinstall (think reset) the handler. In essence, a handler only ever worked once. Each time you trigger the Mousetrap, you need to (at the end) reset the Mousetrap. However, what is the problem here? What if another mouse arrives prior to resetting the mousetrap… for more information see the example code.