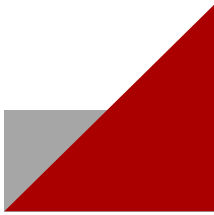
Signals

Benjamin Brewster

Inter-Process Communication (IPC)

- How can we connect our processes together? How can they communicate? Are there simple ways to do it?
- When a user process wants to contact the kernel, it uses a system call
- There are certain events that occur for which the kernel needs to notify a user process directly
- But how does the kernel or another process initiate contact with a user process?

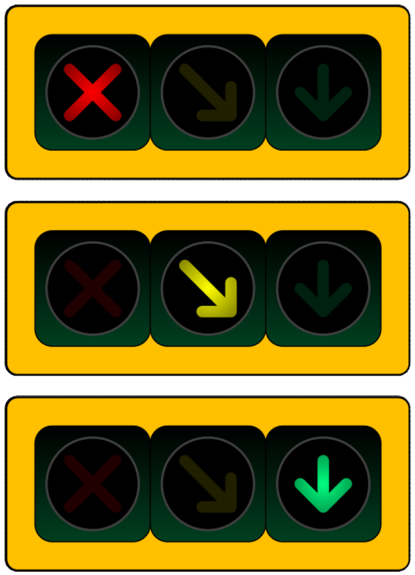


Signals

- Signals are the answer: they interrupt the flow of control (the order that individual instructions are executed) by stopping execution and jumping to an explicitly specified or default signal handler function

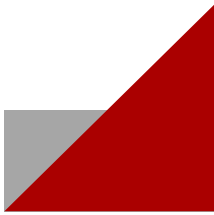
- Critical point: signals tell a process to DO something - to take an action because of a user command or an event

- There are a fixed set of signals:
  - You cannot create your own signals, though the programmatic *response to* and *meaning of* most signals is up to you
  - There are two signals with no inherent meaning at all - you assign meaning to them by catching them and running code



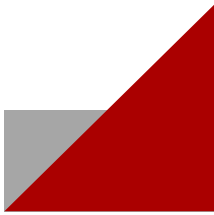
Uses for Signals: Kernel to Process

- Notifications from the Kernel
  - A process has done something wrong
  - A timer has expired
  - A child process has completed executing
  - An event associated with the terminal has occurred
  - The process on the other end of a communication link has gone away



Uses for Signals: Process to Process

- User process to user process notifications, perhaps to:
  - Suspend or resume execution of process
  - Terminate
  - Change modes
  - Change communication methods

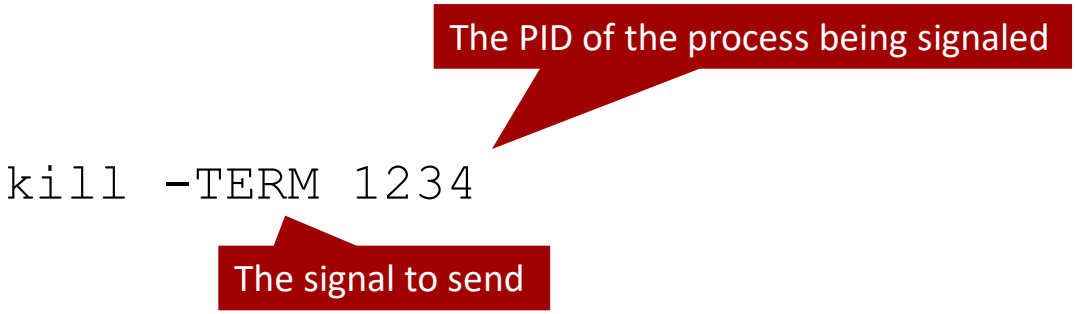


Signal Dictionary - Termination

Processes that output a core dump do not clean anything up - they just die

Signal	#	Easy Name	Catchable	Meaning of Signal sent to Process	Default Action If Not Caught	Core Dump
SIGABRT	6	Abort	Yes	Terminate; sent *by process itself* during `abort()` call which performs no cleanup, unlike `exit()`.	Terminate	Yes
SIGQUIT	3	Quit	Yes	Terminate; sent by user.	Terminate	Yes
SIGINT	2	Interrupt	Yes	The process is requested to terminate; performs cleanup; CTRL-C sends this to process and all its children.	Terminate	No
SIGTERM	15	Terminate	Yes	The process is requested to terminate; performs cleanup.	Terminate	No
SIGKILL	9	Kill	No	Terminate instantly, no cleanup; handled entirely by the kernel; nuke from orbit.	Terminate, not catchable	No

kill



- The given PID affects who the signal is sent to:
  - If PID > 0, then the signal will be sent to the process PID given
  - If pid == 0, then the signal is sent to all processes in the same process group as the sender (from an interactive command line, this means the foreground process group, i.e. your shell)
  - More trickiness for pid < 0

- Let's test it out!

Signaling a Script

```
$ cat sigtermtest
#!/bin/bash
trap "echo 'SIGTERM Received! Exiting with 0!'; exit 0" SIGTERM
while [ 1 -eq 1 ]
do
        echo "nothing" > /dev/null
done
```

```
$ sigtermtest &
[1] 1708
```

```
$ alias psme
alias psme='ps -o ppid,pid,euser,stat,%cpu,rss,args | head -n 1;
ps -eH -o ppid,pid,euser,stat,%cpu,rss,args | grep brewsteb'
```

```
$ psme
 PPID   PID EUSER    STAT %CPU   RSS COMMAND
 4533   751 root     Ss    0.0  4284     sshd: brewsteb [priv]
  751   767 brewsteb S     0.0  2116       sshd: brewsteb@pts/9
  767   768 brewsteb Ss+   0.0  2176         -bash
 4533  1508 root     Ss    0.0  4284     sshd: brewsteb [priv]
 1508  1510 brewsteb S     0.0  2112       sshd: brewsteb@pts/12
 1510  1511 brewsteb Ss    0.0  2064         -bash
 1511  1708 brewsteb R    97.5  1220           /bin/bash ./sigtermtest
 1511  1731 brewsteb R+    0.0  1716           ps -eH -o ppid,pid,euser,stat,%cpu,rss,args
 1511  1732 brewsteb S+    0.0   816           grep brewsteb
```

```
$ kill -SIGTERM 1708
SIGTERM Received! Exiting with 0!
[1]+  Done                    sigtermtest
```

Signal Dictionary - Notification of Wrongdoing

Signal	#	Easy Name	Catchable	Meaning of Signal sent to Process	Default Action If Not Caught	Core Dump
SIGSEGV	-	Segmentation Fault	Yes	Invalid memory reference; terminate, no cleanup.	Terminate	Yes
SIGBUS	-	Bus Error	Yes	Non-existent physical address.	Terminate	Yes
SIGFPE	-	Floating Point Error	Yes	Sent when a process executes an erroneous floating point *or* integer operation, such as divide by zero.	Terminate	Yes
SIGILL	-	Illegal Instruction	Yes	Sent when a process attempts a CPU instruction it cannot issue (malformed, unknown, wrong permissions).	Terminate	Yes
SIGSYS	-	System Call	Yes	Sent when a process passes an incompatible argument to a system call (rare, we use libraries to do this right for us).	Terminate	Yes
SIGPIPE	-	Pipe	Yes	Sent when a process tries to write to a pipe without another process attached to the other end of the pipe.	Terminate	Yes

Why Notify on Events? Branching Logic!

- Gives the process a chance to clean up and finish any important tasks:
  - Perform final file writes
  - `free()` data
  - Write to log files
  - Send signals itself

- A process catching a signal and handling it will do all, some, or none of the above, and then either terminate itself or continue executing!

Signal Dictionary - Control

Signal	#	Easy Name	Catchable	Meaning of Signal sent to Process	Default Action If Not Caught	Core Dump
SIGALRM	14	Alarm	Yes	Sent by alarm() function, normally sent & caught to execute actions at a specific time; performs cleanup	Terminate	No
SIGSTOP	-	Stop	No	Stop execution (but stay alive).	Stop, not catchable	-
SIGTSTP	-	Terminal Stop	Yes	Stop execution (but stay alive).	Stop	-
SIGCONT	-	Continue	Yes	Continue (resume) execution if stopped.	Continue	-
SIGHUP	1	Hang Up	Yes	Sent to a process when its terminal terminates	Terminate	No
SIGTRAP	-	Trap	Yes	Sent when a trap occurs for debugging, i.e. var value change, function start, etc.; terminate, no cleanup.	Terminate	Yes

Timers!

- If you want to wait a specified period of time…
  - You *can* do a busy wait which will consume the CPU continuously while accomplishing nothing
  - Or you can tell the kernel that you want to be notified after a certain amount of time passes

- To set a timer in UNIX
  - Call the `alarm()` or `ualarm()` functions
  - After the time you specify has passed, the kernel will send your process a SIGALRM signal

- This is how `sleep()` works:
  - `sleep()` calls `alarm()`
  - `sleep()` then calls `pause()`, which puts process into waiting state
  - when SIGALARM is received, `sleep()` finally returns

Signal Dictionary: Child Process Has Terminated

Signal	#	Easy Name	Catchable	Meaning of Signal sent to Process	Default Action If Not Caught	Core Dump
SIGCHLD	-	Child Terminated	Yes	A foreground or background child process of this process has terminated, stopped, or continued.	None	-

- Normally, `wait()` and `waitpid()` will suspend a process until one of its child processes has terminated
- Using the signal SIGCHLD allows a parent process to do other work instead of going to sleep and be *notified via signal* when a child terminates
- Then, when SIGCHLD is received, the process can (immediately or later) call `wait()` or `waitpid()` when ready, perhaps leaving the child a zombie for just a little while

Signal Dictionary: User-Defined Signals

Signal	#	Easy Name	Catchable	Meaning of Signal sent to Process	Default Action If Not Caught	Core Dump
SIGUSR1	-	User 1	Yes	Has no particular meaning, performs cleanup.	Terminate	No
SIGUSR2	-	User 2	Yes	Has no particular meaning, performs cleanup.	Terminate	No

- SIGUSR1 and SIGUSR2 have no special meaning to the kernel
- The author of both the sending and receiving processes must agree on the interpretation of the meaning of SIGUSR1 and SIGUSR2

SIGUSR1 Put Through its Paces

```
$ cat sigchldtest
#!/bin/bash
set -m
trap "echo 'Triggering a child process termination with a silent ls'; ls > /dev/null" USR1
trap "echo 'SIGCHLD Received! Exiting!'; exit 0" CHLD
while [ 1 -eq 1 ]
do
        echo "nothing" > /dev/null
done
```

```
$ sigchldtest &
[1] 19141
```

```
$ psme
 PPID   PID EUSER    STAT %CPU   RSS COMMAND
 4533 18174 root     Ss    0.0  4280     sshd: brewsteb [priv]
18174 18187 brewsteb S     0.0  2108       sshd: brewsteb@pts/9
18187 18188 brewsteb Ss    0.0  2104         -bash
18188 19141 brewsteb R     102  1224           /bin/bash ./sigchldtest
18188 19159 brewsteb R+    0.0  1844           ps -eH -o ppid,pid,euser,stat,%cpu,rss,args
18188 19160 brewsteb S+    0.0   816           grep brewsteb
```

```
$ kill -SIGUSR1 19141
Triggering a child process termination with a silent ls
SIGCHLD Received! Exiting!
[1]+  Done                    sigchldtest
```

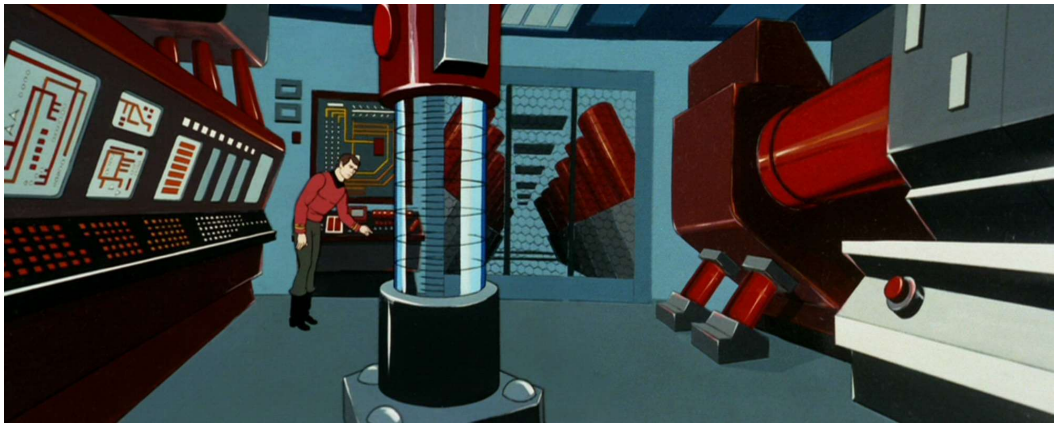
```
$ psme
 PPID   PID EUSER    STAT %CPU   RSS COMMAND
4533 18174 root     Ss    0.0  4280     sshd: brewsteb [priv]
18174 18187 brewsteb S     0.0  2108       sshd: brewsteb@pts/9
18187 18188 brewsteb Ss    0.0  2104         -bash
18188 19200 brewsteb R+    1.0  1844           ps -eH -o ppid,pid,euser,stat,%cpu,rss,args
18188 19201 brewsteb S+    0.0   820           grep brewsteb
```

Receives SIGUSR1, and then generates SIGCHLD: by causing a `fork()` => `exec(ls)` from the script shell, `ls` causes a SIGCHLD to be sent to the parent shell when it terminates

Abnormal Termination: Core Dumps

- Some signals received *cause* an "abnormal termination"
- This also occurs during runtime if the *process* crashes due to a segmentation fault, bus error, etc.
- When this happens, a memory core dump is created which contains:
  - Contents of all variables, hardware registers, & kernel process info at the time the termination occurred
- This core file can be used after the fact to identify what went wrong
- Depending on configuration, core dump files can be difficult to locate on your machine

“Core” Etymology



“Magnetic-core memory was the predominant form of random-access computer memory for 20 years between about 1955 and 1975. Such memory is often just called core memory, or, informally, core.

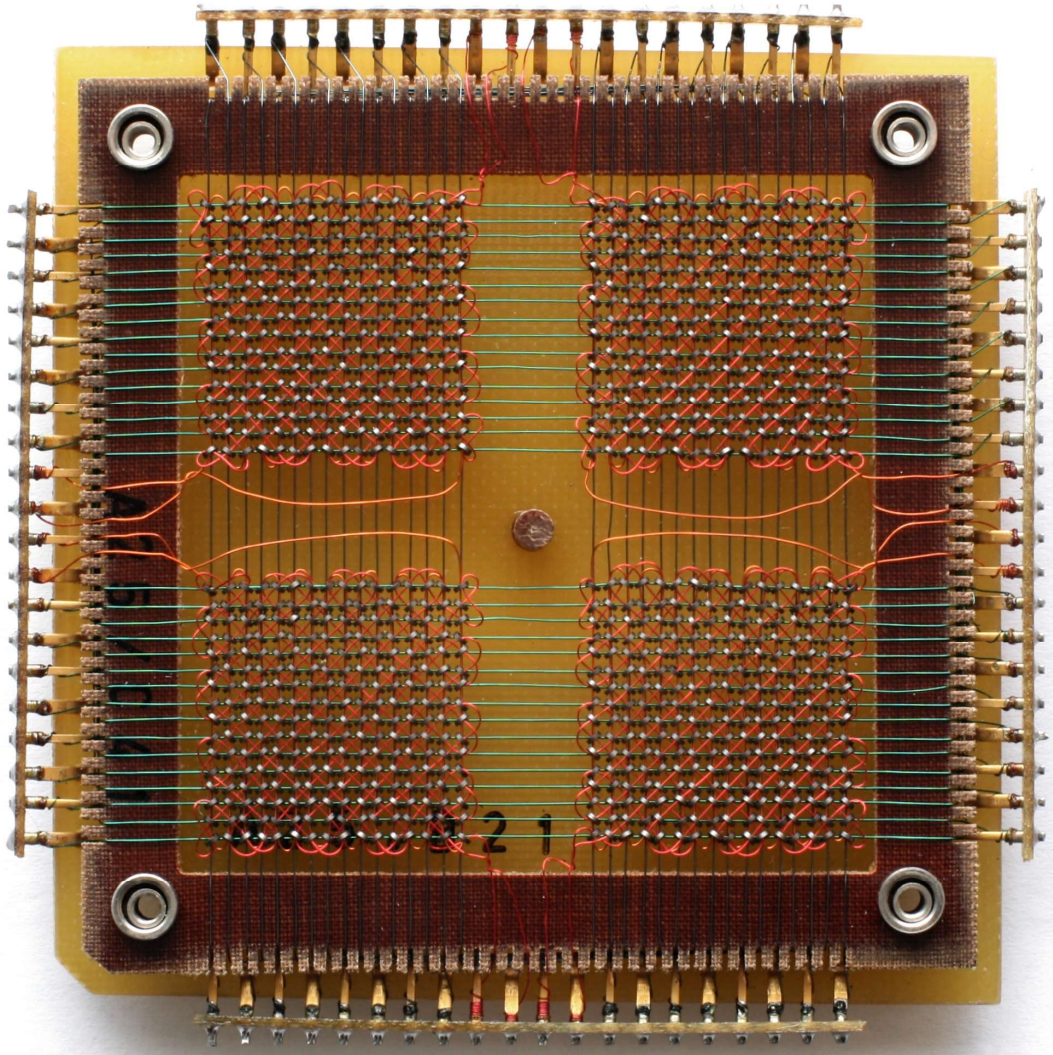
Core uses tiny magnetic toroids (rings), the cores, through which wires are threaded to write and read information. Each core represents one bit of information. The cores can be magnetized in two different ways (clockwise or counterclockwise) and the bit stored in a core is zero or one depending on that core's magnetization direction. The wires are arranged to allow for an individual core to be set to either a one or a zero and for its magnetization to be changed by sending appropriate electric current pulses through selected wires. The process of reading the core *causes the core to be reset to a zero, thus erasing it*. This is called destructive readout. When not being read or written, the cores maintain the last value they had, even when power is turned off. This makes them nonvolatile.”

--Wikipedia, Magnetic-core memory, https://en.wikipedia.org/wiki/Magnetic-core_memory

Handcrafted Memory

"Using smaller cores and wires, the memory density of core slowly increased, and by the late 1960s a density of about 32 kilobits per *cubic foot* was typical. However, reaching this density required extremely careful manufacture, *almost always carried out by hand* in spite of repeated major efforts to automate the process. The cost declined over this period from about *$1 per bit to about 1 cent per bit*. The introduction of the first semiconductor memory SRAM chips in the late 1960s began to erode the core market. The first successful DRAM, the Intel 1103 which arrived in quantity in 1972 at 1 cent per bit, marked the beginning of the end of core. Improvements in semiconductor manufacturing led to rapid increases in storage and decreases in price that drove core from the market by around 1974."

--Wikipedia, Magnetic-core memory, https://en.wikipedia.org/wiki/Magnetic-core_memory

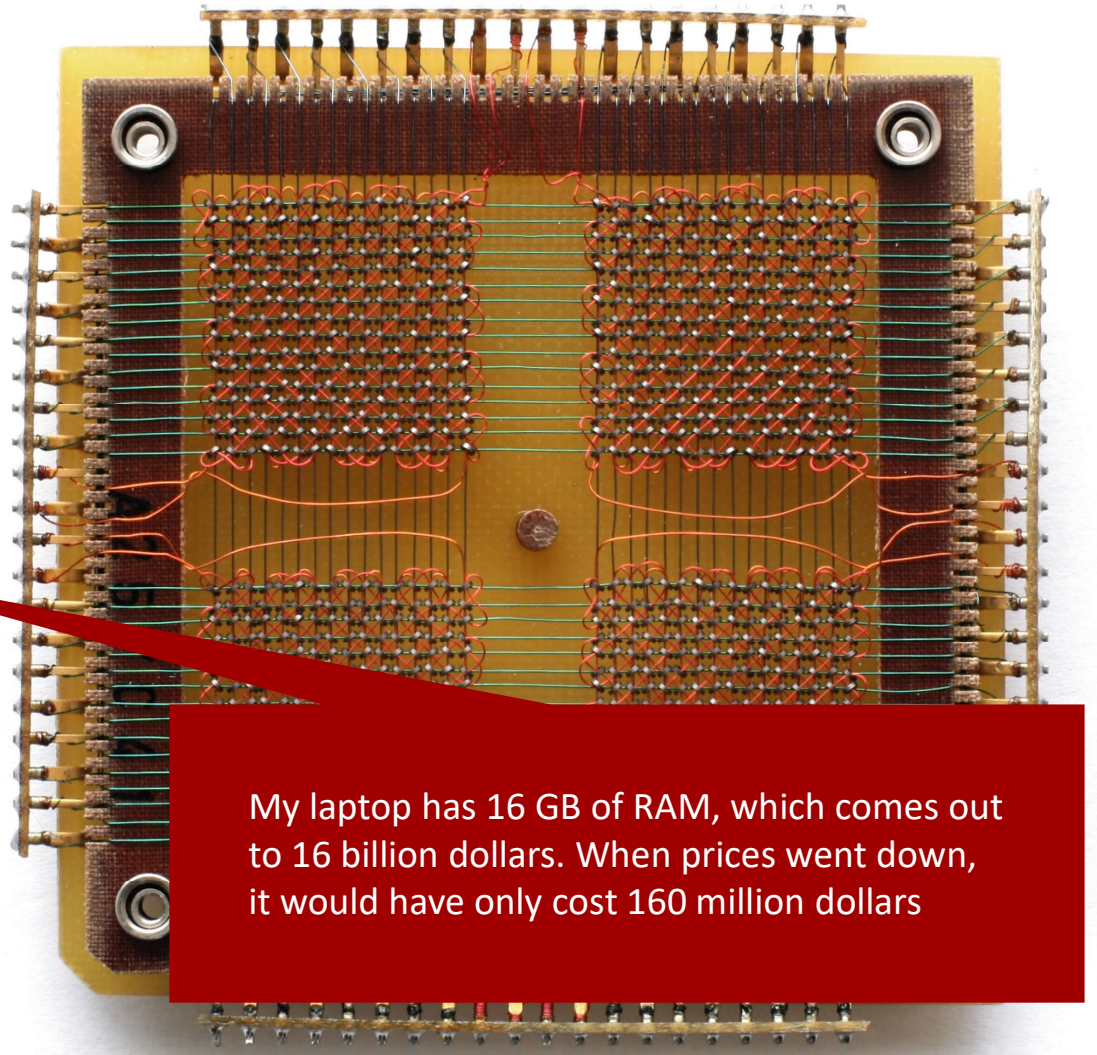


By Konstantin Lanzet - received per EMailCamera: Canon EOS 400D, CC BY-SA 3.0, https://commons.wikimedia.org/w/index.php?curid=7025

Handcrafted Memory

"Using smaller cores and wires, the memory density of core slowly increased, and by the late 1960s a density of about 32 kilobits per *cubic foot* was typical. However, reaching this density required extremely careful manufacture, *almost always carried out by hand* in spite of repeated major efforts to automate the process. The cost declined over this period from about *$1 per bit to about 1 cent per bit*. The introduction of the first semiconductor memory SRAM chips in the late 1960s began to erode the core market. The first successful DRAM, the Intel 1103 which arrived in quantity in 1972 at 1 cent per bit, marked the beginning of the end of core. Improvements in semiconductor manufacturing led to rapid increases in storage and decreases in price that drove core from the market by around 1974."

--Wikipedia, Magnetic-core memory, https://en.wikipedia.org/wiki/Magnetic-core_memory



My laptop has 16 GB of RAM, which comes out to 16 billion dollars. When prices went down, it would have only cost 160 million dollars

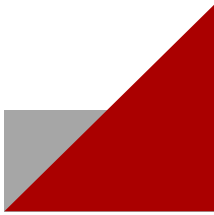
By Konstantin Lanzet - received per EMailCamera: Canon EOS 400D, CC BY-SA 3.0, https://commons.wikimedia.org/w/index.php?curid=7025

Signal Handling API

- Signals that hit your process will cause the default action to occur (see the Signal Dictionaries above)

- To change this, organize signals into sets, then assign your own custom defined “signal handler” functions to these sets, to override the default actions and do whatever you want

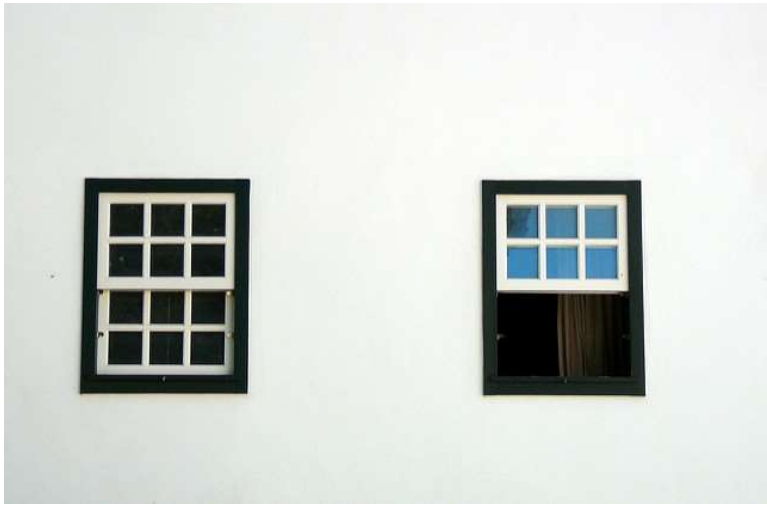
- The next bunch of slides all discuss these signal handling functions, but first a few utility functions…



Sleeping With One Eye Open
Utility: `pause()`

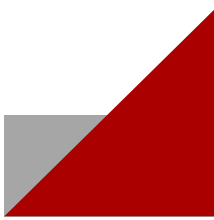
- Sometimes a process has nothing to do, so you consider calling `sleep()`, but you want it to be able to respond to signals, which it can't do in `sleep()`

- To handle this, use the `pause()` function



Sleeping With One Eye Open
Utility: `pause()`

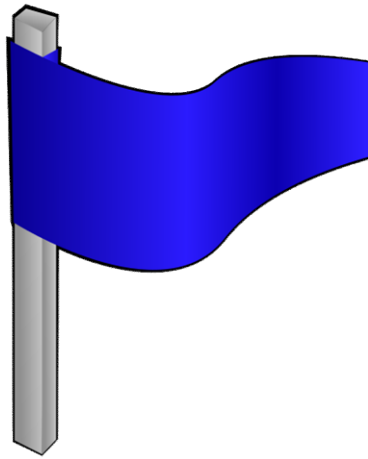
- If a signal is set to be ignored, then the `pause()` continues to be in effect
- If a signal causes a termination, `pause()` does nothing (because the process dies)
- If a signal is caught, the appropriate signal handler function will be called. After the signal handler function is done, `pause()` returns -1 and sets `errno` to EINTR, and the process resumes execution
- You could then issue another `pause()`, for example, or continue on



Sending Signals to Yourself Utilities: `raise()` and `alarm()`

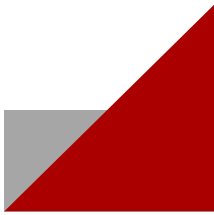
- You can send yourself a specified signal immediately with `raise()`:
  - `int raise(int signal);`

- The `alarm()` function sends your process a SIGALRM signal at a later time
  - `unsigned int alarm(unsigned int seconds);`
  - Note that `alarm()` will return immediately, unlike `sleep()`
  - You can only have one alarm active at any time



Utility Type: Signal Sets

- A *signal set* is simply a list of signal types which is used elsewhere
- A signal set is defined using the special type `sigset_t` defined in <signal.h>
- Functions for managing signal sets:
  - `segemptyset(),sigfillset(),sigaddset(),sigdelset()`



Utility Type: Signal Sets

- To declare a signal set:

```
sigset_t my_signal_set;
```

- To initialize or reset the signal set to have *no* signal types:

```
sigemptyset(&my_signal_set);
```

- To initialize or reset the signal set to have *all* signal types:

```
sigfillset(&my_signal_set);
```

- To add a single signal type to the set:

```
sigaddset(&my_signal_set, signal);
```

- To remove a single signal type from the set:

```
sigdelset(&my_signal_set, signal);
```

Like SIGINT

`sigaction()`

- `sigaction()` registers a signal handling function that you've created for a specified set of signals

```
int sigaction(int signo, struct sigaction *newact, struct sigaction *origact);
```

- The first parameter is the signal type of interest (SIGINT, SIGHUP, etc.)
- The second parameter is a pointer to a data-filled `sigaction` struct which describes the action to be taken upon receipt of the signal given in the first parameter
- The third parameter is a pointer to another `sigaction` struct, with which the `sigaction()` function will use to write out what the handling settings for this signal were *before* this change was requested

The sigaction Structure

Shares the same name as the `sigaction()` function, so don't get them confused! :)

```
struct sigaction
{
      void (*sa_handler)(int);
      sigset_t sa_mask;
      int sa_flags;
      void (*sa_sigaction)(int, siginfo_t*, void*);
};
```

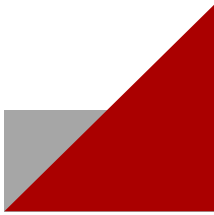
- The first attribute should be set to one of three values:
  - SIG_DFL :: Take the default action for the signal
  - SIG_IGN :: Ignore the signal
  - A pointer to a function that should be called when this signal is received (see next slide)

Pointers to Functions in C

- They look complicated, but they are really simple
- In the sigaction structure, it is defined as

```
void (*sa_handler)(int);
```

- The "*" in front of the name `sa_handler`, and the parentheses around it, indicate that this is a pointer to a function
- The "void" indicates that the function `sa_handler` points to does not return anything
- The "int" indicates that `sa_handler` should point to a function that has one parameter: an integer
- This "int" will hold the signal number when `sa_handler` is called, which is important because multiple signals may be registered with this struct, and the int will be the only way to tell which signal caused the handler to start



Using Pointers to Functions

```
#include <stdio.h>

int AddOne(int inputArg);

void main()
{
        int (*fpArg)(int) = AddOne;
        printf("10 + 1 = %d\n", fpArg(10));
}

int AddOne (int input)
{
        return input + 1;
}
```

Argument

Return

Declares a function pointer variable named **fpArg**, and sets it equal to the same address as where `AddOne()` is declared

The sigaction Struct

```
struct sigaction
{
      void (*sa_handler)(int);
      sigset_t sa_mask;
      int sa_flags;
      void (*sa_sigaction)(int, siginfo_t*, void*);
};
```

- The `sa_mask` attribute indicates what signals should be blocked while the signal handler is executing:
  - *Blocked* means that the signals arriving *during the execution of* `sa_handler` are held until your signal handler is done executing, at which point the signals will then be delivered in order to your process; note that multiple signals of the same type arriving may be combined, so you can't use this to count signals!
- Pass this a `sigset_t` as described above

The sigaction Struct

```
struct sigaction
{
      void (*sa_handler)(int);
      sigset_t sa_mask;
      int sa_flags;
      void (*sa_sigaction)(int, siginfo_t*, void*);
};
```

- The third attribute of the sigaction struct provides additional instructions (flags):
  - SA_RESTHAND :: Resets the signal handler to SIG_DFL (default action) after the first signal has been received and handled
  - SA_SIGINFO :: Tells the kernel to call the function specified in the fourth attribute (`sa_sigaction`), instead of the first attribute (`sa_handler`). More detailed information can be passed to this function, as you can see by the additional arguments
  - Set to 0 if you aren't planning to set any flags

The sigaction Struct

```
struct sigaction
{
        void (*sa_handler)(int);
        sigset_t sa_mask;
        int sa_flags;
        void (*sa_sigaction)(int, siginfo_t*, void*);
};
```

- The fourth attribute, `sa_sigaction`, specifies an alternative signal handler function to be called. This attribute will only be used if the SA_SIGINFO flag is set in `sa_flags`
- The `siginfo_t` struct pointer you pass in will be written to once `sa_sigaction` is invoked; it will then contain information such as which process sent you the signal
- The “void*” pointer allows you to pass in a context, an obsolete, non-POSIX construct that manages user threads
- Most of the time you will use `sa_handler` and not `sa_sigaction`

Catching & Ignoring Signals - Part 1 of 2

```
$ cat catchingSignals.c
#include <signal.h>
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>

void catchSIGINT(int signo)
{
        char* message = "Caught SIGINT, sleeping for 5 seconds\n";
        write(STDOUT_FILENO, message, 38);
        raise(SIGUSR2);
        sleep(5);
}

void catchSIGUSR2(int signo)
{
        char* message = "Caught SIGUSR2, exiting!\n";
        write(STDOUT_FILENO, message, 25);
        exit(0);
}
```

Can't use `printf()` during a signal handler, as it's non-reentrant; can't use `strlen(message)` in place of 38, as it's also non-entrant!

Send this process SIGUSR2; if this signal is blocked, it'll be delivered when the block is removed

Catching & Ignoring Signals Part 2 of 2

```
main()
{
        struct sigaction SIGINT_action = {0}, SIGUSR2_action = {0}, ignore_action = {0};

        SIGINT_action.sa_handler = catchSIGINT;
        sigfillset(&SIGINT_action.sa_mask);
        SIGINT_action.sa_flags = 0;

        SIGUSR2_action.sa_handler = catchSIGUSR2;
        sigfillset(&SIGUSR2_action.sa_mask);
        SIGUSR2_action.sa_flags = 0;

        ignore_action.sa_handler = SIG_IGN;

        sigaction(SIGINT, &SIGINT_action, NULL);
        sigaction(SIGUSR2, &SIGUSR2_action, NULL);
        sigaction(SIGTERM, &ignore_action, NULL);
        sigaction(SIGHUP, &ignore_action, NULL);
        sigaction(SIGQUIT, &ignore_action, NULL);

        printf("SIGTERM, SIGHUP, and SIGQUIT are disabled.\n");
        printf("Send a SIGUSR2 signal to kill this program.\n");
        printf("Send a SIGINT signal to sleep 5 seconds, then kill this program.\n");

        while(1)
                pause();
}
```

Completely initialize this complicated struct to be empty

Block/delay all signals arriving while this mask is in place

Sleep, but wake up to signals

Catching & Ignoring Signals - Results

```
$ catchingSignals
SIGTERM, SIGHUP, and SIGQUIT are disabled.
Send a SIGUSR2 signal to kill this program.
Send a SIGINT signal to sleep 5 seconds, then kill this program.
^CCaught SIGINT, sleeping for 5 seconds
Caught SIGUSR2, exiting!
$ catchingSignals &
[1] 29443
$ SIGTERM, SIGHUP, and SIGQUIT are disabled.
Send a SIGUSR2 signal to kill this program.
Send a SIGINT signal to sleep 5 seconds, then kill this program.
$
$
$ kill -SIGTERM 29443
$ kill -SIGUSR2 29443
Caught SIGUSR2, exiting!
[1]+  Done                    catchingSignals
```

Send SIGINT

The prompt is written before the output of "catchingSignals &" was written

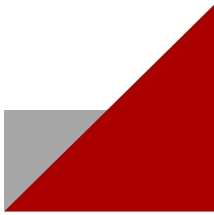
Does nothing, it's disabled!

Child Processes and Inheritance

- When calling `fork()`, child processes *inherit* and get their own instance of the signal handler functions declared in the parent - these are assigned to the same signals as the parent automatically

- However, calling `exec…()` in your process will *remove* any special signal handler function you wrote and then assigned to `sa_handler` or `sa_sigaction` previously!

- Critical note: SIG_DFL and SIG_IGN *are* preserved through an `exec…()`. This is the only way to tell processes you `exec…()`, that you didn't write (like **ls**, other bash commands, etc.), to ignore particular signals. In other words, there is no way to set up an *arbitrary* signal handler for programs you can't change: you can only set them to *ignore* specific signals



Blocking Signals

- It is also possible to block signals from occurring during your program execution *outside* of a signal handler function
- As before, blocking a signal simply means that it is delayed until you unblock the signal
- This could be useful if you have code where it is extremely critical that you don't get interrupted by any signal
- This kind of blocking is done with `sigprocmask()`

Signals and System Calls

- Signals can arrive any time, including in the middle of a system call!
- System calls are savvy about signals and prevent data loss and corruption from occurring
  - They also prevent partial actions from happening
- Normally, system calls will return an error if a signal interrupts them, and set `errno` to EINTR
- You can tell system calls to automatically restart by setting SA_RESTART in the `sa_flags` variable of the `sigaction` struct