Processes

Benjamin Brewster

The Process

- Process Management is a necessary component of a multiprogrammable operating system

- Process:

  An instance of an executing program, with a collection of execution resources associated with it

UNIX Process Components

- A unique identity (process id *aka* pid) :: `pid_t pid = getpid();`
- A virtual address space (from 0 to memory limit)
- Program code and data (variables) in memory
- User/group identity (controls what you can access), umask value
- An execution environment all to itself
  - Environment variables
  - Current working directory
  - List of open files
  - A description of actions to take on receiving signals
- Resource limits, scheduling priority
- and more… see the exec() man page

More on this later

Programs vs Processes

- A program is the executable code:

- A process is a running instance of a program:

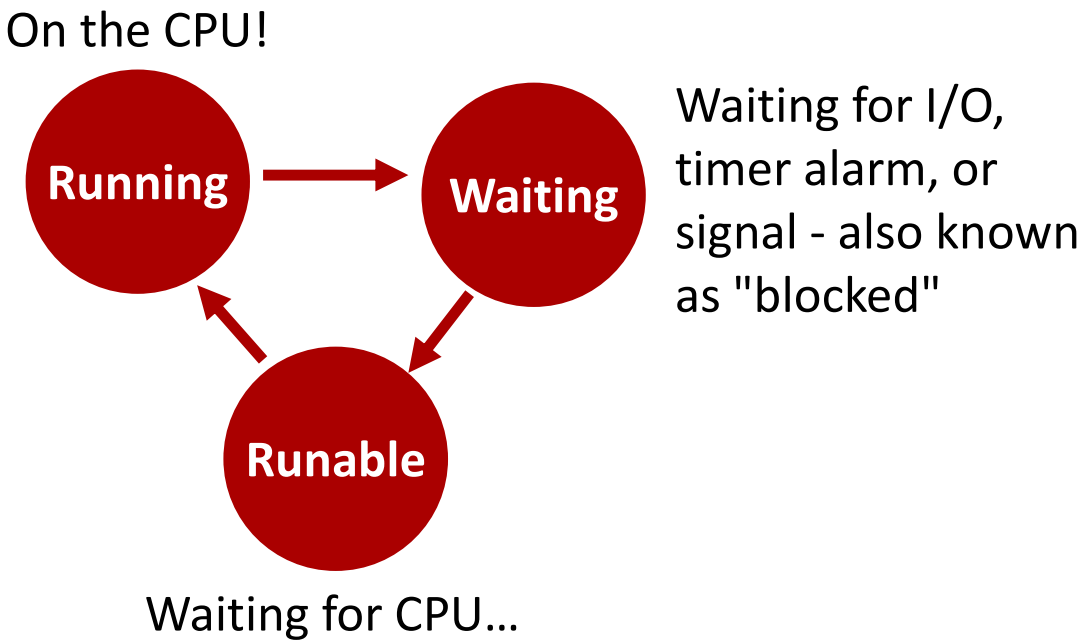
- More than one process can be concurrently executing the same program code, with separate process resources:

Important Process States in UNIX

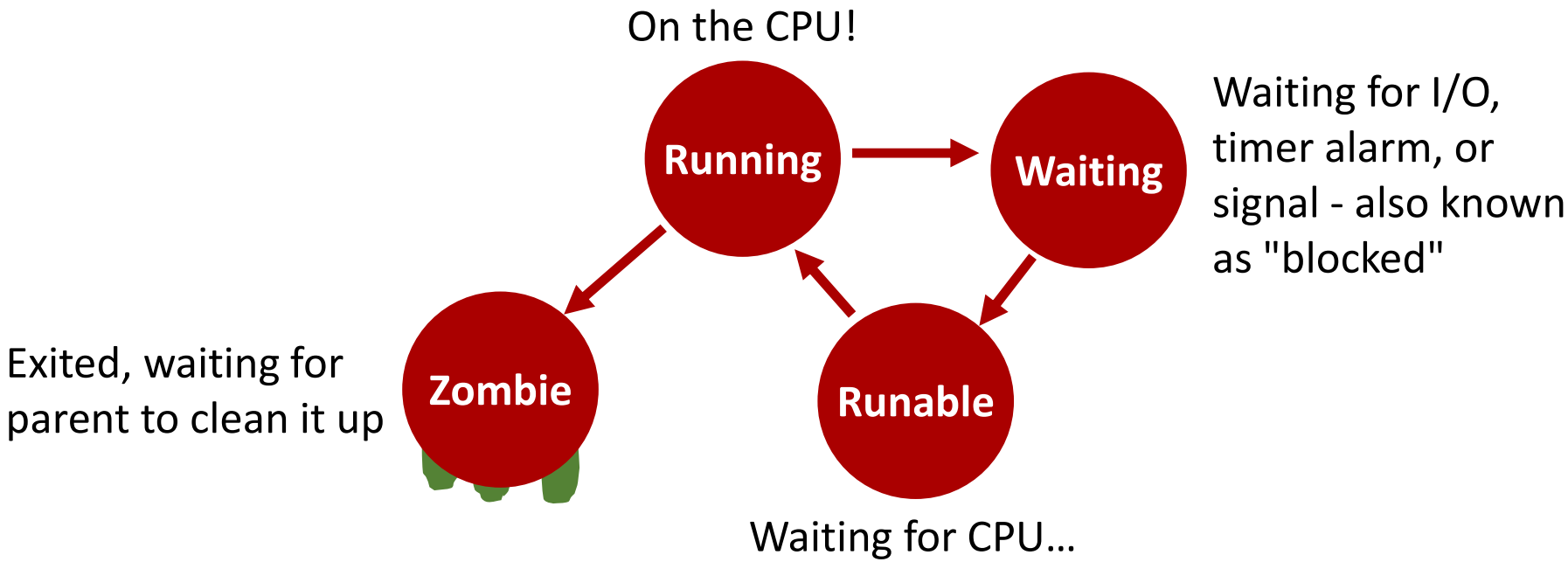
On the CPU!

Important Process States in UNIX

Important Process States in UNIX



Important Process States in UNIX



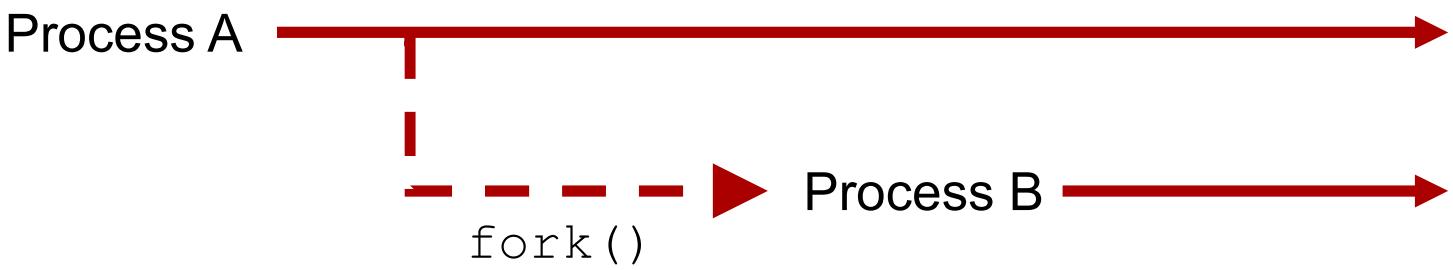
How Do You Create a Process?

- Let the shell do it for you!
  - When you execute a program, the shell creates the process for you
- In some cases, you'll want to do it yourself
  - Our shell-writing assignment
- Unix provides a C API for creating and managing processes explicitly, as the following material shows

Managing Processes

- Functions we'll be covering:
  - `fork()`
  - The exec() family:
    - `execl(),execlp(),execv(),execvp()`
  - `exit()`
  - `wait(),waitpid()`
  - `getpid()`
  - `getenv(),putenv()`

How to Start a New Process



- Processes A and B are nearly identical copies, both running the same code, and continuing on from where the fork() call occurred

Process A == Process B ??

- The two processes have different pids
- Each process returns a different value from `fork()`
- Process B gets copies of all the open file descriptors of Process A
- Process B has all of the same variables set to the same values as Process A, but they are now separately managed!
- More to come in a bit

fork()

- A sample program using `fork()`

If something went wrong, `fork()` returns -1 to the parent process and sets the global variable `errno`; no child process was created

In the child process, `fork()` returns 0

In the parent process, `fork()` returns the process id of the child process that was just created

```
$ cat forktest.c
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>

void main()
{
    pid_t spawnpid = -5;
    int ten = 10;

    spawnpid = fork();
    switch (spawnpid)
    {
        case -1:
            perror("Hull Breach!");
            exit(1);
            break;
        case 0:
            ten = ten + 1;
            printf("I am the child! ten = %d\n", ten);
            break;
        default:
            ten = ten - 1;
            printf("I am the parent! ten = %d\n", ten);
            break;
    }
    printf("This will be executed by both of us!\n");
}
```

Results

```
$ forktest
I am the child! ten = 11
This will be executed by both of us!
I am the parent! ten = 9
This will be executed by both of us!
```

The order of whether the parent or child reports its text first is up to the OS and its scheduler

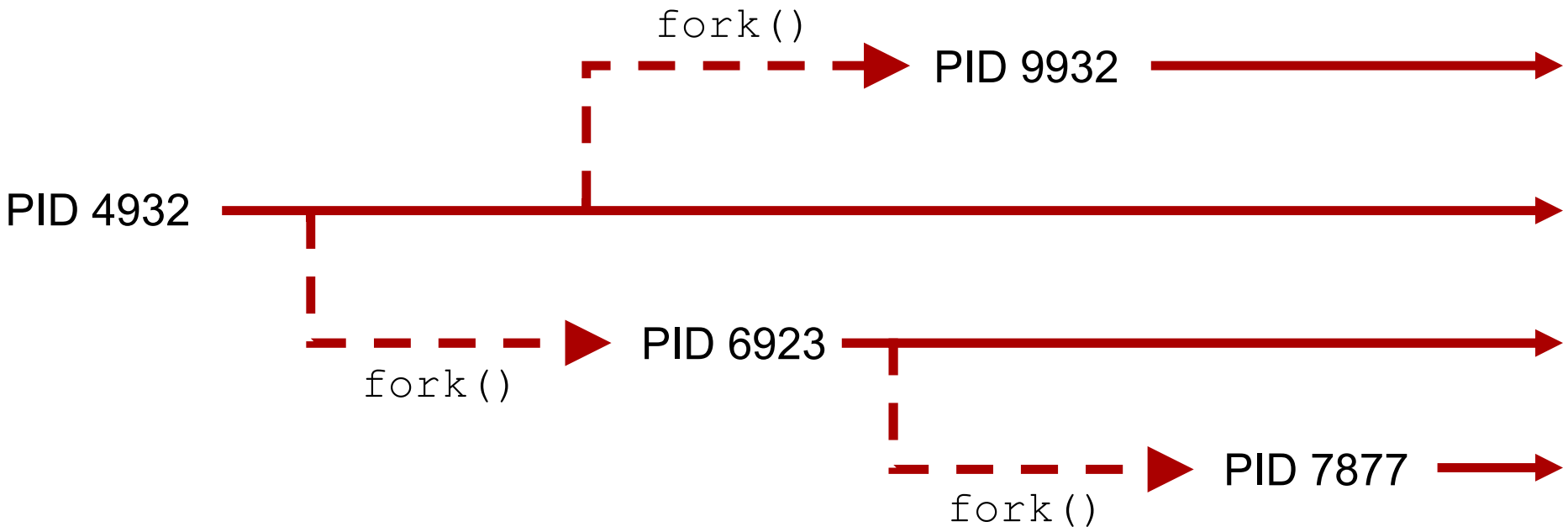
Key Items Inherited

- Inherited by the child from the parent:
  - Program code
  - Process credentials (real/effective/saved UIDs and GIDs)
  - Virtual memory contents, including stack and heap
  - Open file descriptors
  - Close-on-exec flags
  - Signal handling settings
  - process group ID
  - current working directory (CWD)
  - controlling terminal
  - …

Key Items Unique to the Child Process

- Unique to the child:
  - Process ID
  - Parent process ID is different (it's the parent that just spawned it)
  - Own copy of file descriptors
  - Process, text, data and other memory locks are NOT inherited
  - Pending signals initialized to the empty set
  - …

`fork()` Forms a Family Tree



Child Process Termination

- A child process can exit for two reasons
  - It completes execution and exits normally
    - Case 1: The child process completed what it was supposed to do and exited with a successful exit status (ie 0)
    - Case 2: The child process encountered an error condition, recognized it, and exited with a non-successful exit status (ie non-zero)
  - It was killed by a signal
    - The child process was sent a signal that by default terminates a process, and the child process *did not catch it*

- How do parents check to see if child processes have terminated?

Checking the Exit Status

- Both of these commands check for child process termination:
  - `wait()`
  - `waitpid()`

- For both functions, you pass in a pointer to which the OS writes an int, which identifies how the child exited
  - We examine this int with various macros to learn what happened

`wait` vs `waitpid`

- `wait()` will block - until *any* one child process terminates; returns the process id of the terminated child

- `waitpid()` will block - until the child process with the *specified* process ID terminates (or has already terminated); returns the process id of the terminated child
  - If you pass it a special flag, it will check if the specified child process has terminated, then immediately return even if the specified child process hasn't terminated yet

`wait()` and `waitpid()` Syntax

```
pid_t wait(int *childExitMethod);

pid_t waitpid(pid_t pid,
              int *childExitMethod,
              int options);
```

- Block this parent until any child process terminates:

```
childPID = wait(&childExitMethod);
```

- Block this parent until the specified child process terminates:

```
childPID_actual = waitpid(childPID_intent, &childExitMethod, 0);
```

- Check if any process has completed, return immediately with 0 if none have:

```
childPID = waitpid(-1, &childExitMethod, WNOHANG);
```

- Check if the process specified has completed, return immediately with 0 if it hasn't:

```
childPID_actual = waitpid(childPID_intent, &childExitMethod, WNOHANG);
```

You can use the same variable here, if you like

Proper `waitpid()` Placement

$ cat forkwaittest.c

```
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>

void main()
{
        pid_t spawnPid = -5;
        int childExitMethod = -5;

        spawnPid = fork();
        if (spawnPid == -1) //
        {
                perror("Hull Breach!\n");
                exit(1);
        }
        else if (spawnPid == 0) // Terminate the child process immediately
        {
                printf("CHILD: PID: %d, exiting!\n", spawnPid);
                exit(0);
        }

        printf("PARENT: PID: %d, waiting...\n", spawnPid);
        waitpid(spawnPid, &childExitMethod, 0);
        printf("PARENT: Child process terminated, exiting!\n");
        exit(0);
}
```

$ forkwaittest

```
PARENT: PID: 3311, waiting...
CHILD: PID: 0, exiting!
PARENT: Child process terminated, exiting!
```

Blocks the parent until the child process with specified PID terminates

Checking the Exit Status - Normal Termination

- `wait(&childExitMethod)` and `waitpid(…, &childExitMethod, …)` can identify two ways a process can terminate:

- If the process terminates normally, then the WIFEXITED macro returns non-zero:

```
if (WIFEXITED(childExitMethod) != 0)
    printf("The process exited normally\n");
```

- We can get the actual exit status with the WEXITSTATUS macro:

```
int exitStatus = WEXITSTATUS(childExitMethod);
```

Checking the Exit Status - Signal Termination

- `wait(&childExitMethod)` and `waitpid(…, &childExitMethod, …)` can identify two ways a process can terminate:

- If the process was terminated by a signal, then the WIFSIGNALED macro returns non-zero:

```
if (WIFSIGNALED(childExitMethod) != 0)
    printf("The process was terminated by a signal\n");
```

- We can get the terminating signal with the WTERMSIG macro:

```
int termSignal = WTERMSIG(childExitMethod);
```

Checking the Exit Status - Exclusivity

- Barring the use of the non-standard `WCONTINUED` and `WUNTRACED` flags in `waitpid()`, only one of the `WIFEXITED()` and `WIFSIGNALED()` macros will be non-zero!
- Thus, if you want to know how a child process died, you need to use both WIFEXITED and WIFSIGNALED!
- If the child process has terminated normally, do not run `WTERMSIG()` on it, as there is *no signal number* that killed it!
- If the child process was terminated by a signal, do not run `WEXITSTATUS()` on it, as it has *no exit status* (i.e., no `exit()` or `return()` functions were executed)!

Checking the Exit Status

```
int childExitMethod;
pid_t childPID = wait(&childExitMethod);

if (childPID == -1)
{
   perror("wait failed");
   exit(1);
}

if (WIFEXITED(childExitMethod))
{
   printf("The process exited normally\n");
   int exitStatus = WEXITSTATUS(childExitMethod);
   printf("exit status was %d\n", exitStatus);
}
else
   printf("Child terminated by a signal\n");
```

Non-zero evaluates to true in C

This statement is true, but it never hurts to examine `WIFSIGNALED()`, also, to make sure!

How to Run a Completely Different Program

- `fork()` always makes a copy of your *current* program
- What if you want to start a process that is running a completely different program?
- For this we use the `exec…()` family

`exec…()` - Execute

- `exec…()` replaces the currently running program with a *new* program that you specify
- The `exec…()` functions do not return - they destroy the currently running program
  - No line after a successful `exec…()` call will run
- You can specify arguments to `exec…()`: these become the command line arguments that show up as `argc`/`argv` in C, and as the `$1`, `$2`, etc positional parameters in a bash shell

A møøse once bit my sister

Two Types of Execution

```
int execl(char *path, char *arg1, …, char *argn);
```

- Executes the program specified by *path*, and gives it the command line arguments specified by strings *arg1* through *argn*

```
int execv(char *path, char *argv[]);
```

- Executes the program specified by *path*, and gives it the command line arguments indicated by the pointers in *argv*

Current Working Directory

- `exexl()` and `execv()` do not examine the PATH variable - they only look in the current working directory (but see the next slide)

- If you don't specify a fully qualified path name, then your programs will not be executed, even if they are in a directory listed in PATH, and `execl()` and `execv()` will return with an error

- To move around the directory structure in C, use the following:
  - `getcwd()` :: Gets the current working directory
  - `chdir()` :: Sets the current working directory

Exec…() and the PATH variable

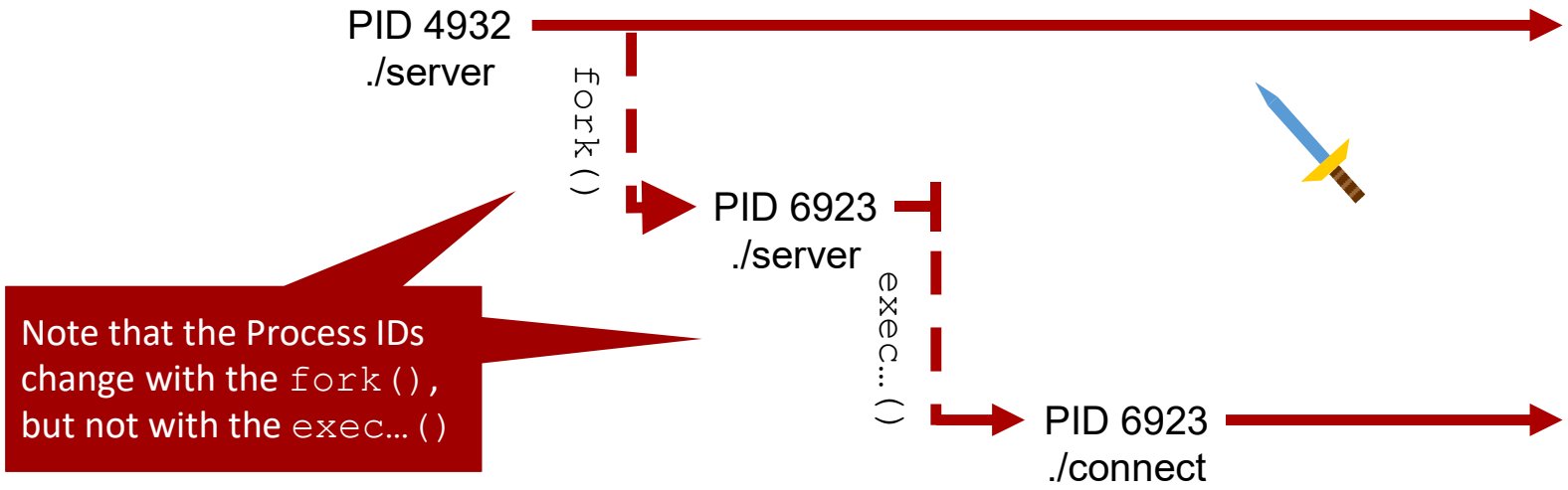
```
int execl(char *path, char *arg1, …, char *argn);
int execlp(char *path, char *arg1, …, char *argn);

int execv(char *path, char *argv[]);
int execvp(char *path, char *argv[]);
```

- The versions ending with *p* will search your PATH environment variable for the executable given in *path*
- In general, you'll want to use the versions with *p* - `execlp()` or `execvp()` - as they are much more convenient

Execute a New Process

- `exec…()` *replaces* the program it is called from - it does not create a new process!
- Using `fork()` and `exec…()`, we can keep our original program going, and spawn a brand-new process!



Passing parameters to `execlp()`

- `int execlp(char *path, char *arg1, …, char *argn);`
- First parameter to `execlp()` is the pathname of the new program
- Remaining parameters are "command line arguments"
- First argument should be the same as the first parameter (the command itself)
- Last argument must always be NULL, which indicates that there are no more parameters
- Do not pass any shell-specific operators into any member of the `exec…()` family, like `<`, `>`, `|`, `&`, or `!`, because the shell is not being invoked - only the OS is!
- Example:

```
execlp("ls", "ls", "-a", NULL);
```

fork() + execlp() Example

```
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
void main() {
        pid_t spawnPid = -5;
        int childExitStatus = -5;

        spawnPid = fork();
        switch (spawnPid) {
                case -1: { perror("Hull Breach!\n"); exit(1); break; }
                case 0: {
                        printf("CHILD(%d): Sleeping for 1 second\n", getpid());
                        sleep(1);
                        printf("CHILD(%d): Converting into \'ls -a\'\n", getpid());
                        execlp("ls", "ls", "-a", NULL);
                        perror("CHILD: exec failure!\n");
                        exit(2); break;
                }
                default: {
                        printf("PARENT(%d): Sleeping for 2 seconds\n", getpid());
                        sleep(2);
                        printf("PARENT(%d): Wait()ing for child(%d) to terminate\n", getpid(), spawnPid);
                        pid_t actualPid = waitpid(spawnPid, &childExitStatus, 0);
                        printf("PARENT(%d): Child(%d) terminated, Exiting!\n", getpid(), actualPid);
                        exit(0); break;
                }
        }
}
```

`fork()` + `execlp()` Output

```
$ gcc -o forkexec forkexec.c
$ forkexec
PARENT(8201): Sleeping for 2 seconds
CHILD(8204): Sleeping for 1 second
CHILD(8204): Converting into 'ls -a'
.                   cAd                   Ctests          forkyouzombie    leaky2            python-billion
..                  catsAndDogs           dollars         forkyouzombie.c  leaky2.c          python-billion-fast
addsix-bash         c-billion             doubleparen     forloop          leaky3            pythonmath
addsix-c            c-billion.c           error.txt       greptests        leaky3.c          pythonstring
addsix-c.c          cstring-array         exiter          hardlink1        leaky.c           pythontest
array-of-pointers   cstring-array.c       forkexec        havoc            malloctest        readerror
array-of-pointers.c cstring-array-unint   forkexec.c      hw               malloctest.c      readpipetest
arraytest           cstring-array-unint.c forktest        hw.c             memerrors         readtest
arraytest.c         cstring-inlinearray   forktest.c      inodetest        paramtest         rowfile
arraytest.c.backup  cstring-inlinearray.c forkwaittest    killthesis       perlcamel         rowfile2
billion             cstring-segfault.c    forkwaittest.c  leak2.c_backup   permissionstests  sortdata
PARENT(8201): Wait()ing for child(8204) to terminate
PARENT(8201): Child(8204) terminated, Exiting!
```

Passing parameters to `execvp()`

- `int execvp(char *path, char *argv[]);`
- First parameter to `execvp()` is the pathname of the new program
- Second parameter is an array of pointers to strings
- First string should be the same as the first parameter (the command itself)
- Last string must always be NULL, which indicates that there are no more parameters
- Do not pass any shell-specific operators into any member of the `exec…()` family, like `<`, `>`, `|`, `&`, or `!`, because the shell is not being invoked - only the OS is!
- Example:

```
char* args[3] = {"ls", "-a", NULL};
execvp(args[0], args);
```

execvp() Example

```
$ cat execvptest.c
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>

void execute(char** argv)
{
        if (execvp(*argv, argv) < 0)
        {
                perror("Exec failure!");
                exit(1);
        }
}
void main()
{
        char* args[3] = {"ls", "-a", NULL};
        printf("Replacing process with: %s %s\n", args[0], args[1]);
        execute(args);
}
$ gcc -o execvptest execvptest.c
$ execvptest
Replacing process with: ls -a
.         ..        execvptest  execvptest.c
```

`exit()`

- `atexit()`
  - Arranges for a function to be called before `exit()`
- `exit()` does the following:
  - Calls all functions registered by `atexit()`
  - Flushes all stdio output streams
  - Removes files created by `tmpfile()`
  - Then calls `_exit()`
- `_exit()` does the following:
  - Closes all files
  - Cleans up everything - see the man page for `wait()` for a complete list of what happens on exit
- `return()` from `main()` does exactly the same thing as `exit()`

Environment Variables

- A set of text variables, often used to pass information between the shell and a C program
- May be useful if:
  - You need to specify a configuration for a program that you call frequently (LESS, MORE)
  - You need to specify a configuration that will affect many different commands that you execute (TERM, PAGER, PRINTER)
- You can view/edit the environment from bash by using the `printenv` and `export` commands, and assignment (=) operator
- The environment can be edited in C with `setenv()` and `getenv()`

printenv

$ printenv

```
MANPATH=/usr/local/man:/usr/man:/usr/share/man
HOSTNAME=eos-class.engr.oregonstate.edu
SELINUX_ROLE_REQUESTED=
TERM=xterm
SHELL=/bin/bash
HISTSIZE=1000
SSH_CLIENT=128.193.54.168 52413 22
MORE=-c
QTDIR=/usr/lib64/qt-3.3
QTINC=/usr/lib64/qt-3.3/include
SSH_TTY=/dev/pts/17
USER=brewsteb
PAGER=less
MAIL=/var/spool/mail/brewsteb
PATH=/bin:/sbin:/usr/local/bin:/usr/bin:/usr/local/apps/bin:/usr/bin/X11:/nfs/stak/faculty/b/brewsteb/bin:.
PWD=/nfs/stak/faculty/b/brewsteb/tempdir
LANG=en_US.UTF-8
MODULEPATH=/usr/share/Modules/modulefiles:/etc/modulefiles
KDEDIRS=/usr
SSH_ASKPASS=/usr/libexec/openssh/gnome-ssh-askpass
HISTCONTROL=ignoredups
SHLVL=1
HOME=/nfs/stak/faculty/b/brewsteb
LESS=QMcde
LOGNAME=brewsteb
SSH_CONNECTION=128.193.54.168 52413 128.193.37.0 22
LESSOPEN=||/usr/bin/lesspipe.sh %s
G_BROKEN_FILENAMES=1
BASH_FUNC_module()=() {  eval `/usr/bin/modulecmd bash $*`
}
_=/usr/bin/printenv
```

Manipulating the Environment

- Bash:

```
MYVAR="Some text string 1234"
export MYVAR
echo $MYVAR
MYVAR="New text"
```

More on `export` in a bit

- C:

```
setenv("MYVAR", "Some text string 1234", 1);
printf("%s\n", getenv("MYVAR"));
```

`1` means overwrite the value, if it exists

Manipulating the Environment… for Just You

```
$ cat bashAndCEnvironment.c
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <string.h>
int main(int argc, char* argv[])
{
  char array[1000];
  printf("Variable %s has value: %s\n", argv[1], getenv(argv[1]));
  printf("Doubling it!\n");
  strcpy(array, getenv(argv[1]));
  strcat(array, getenv(argv[1]));
  printf("New value of %s will be: %s\n", argv[1], array);
  setenv(argv[1], array, 1);
  printf("Variable %s has value: %s\n", argv[1], getenv(argv[1]));
}
```

```
$ MYVAR="TEXT."
$ export MYVAR
$ echo $MYVAR
TEXT.
$ gcc -g -o bashAndCEnvironment bashAndCEnvironment.c
$ bashAndCEnvironment MYVAR
Variable MYVAR has value: TEXT.
Doubling it!
New value of MYVAR will be: TEXT.TEXT.
Variable MYVAR has value: TEXT.TEXT.
$ echo MYVAR
TEXT.
```

All that work for nothing! A processes execution environment belongs to only that process, which gets its initial values from the parent shell - but a process cannot edit the environment variables of it's parent shell!

Modifications, thus, will only be useful for your current process.

Exporting Environment Variables

```
$ MYTESTVAR="testtext"
$ echo $MYTESTVAR
testtext
$ bashAndCEnvironment MYTESTVAR
Variable MYTESTVAR has value: (null)
Doubling it!
Segmentation fault (core dumped)
$ export MYTESTVAR
$ bashAndCEnvironment MYTESTVAR
Variable MYTESTVAR has value: testtext
Doubling it!
New value of MYTESTVAR will be: testtexttesttext
Variable MYTESTVAR has value: testtexttesttext
$ echo $MYTESTVAR
testtext
```

`export` makes the variable available for all child processes of the shell

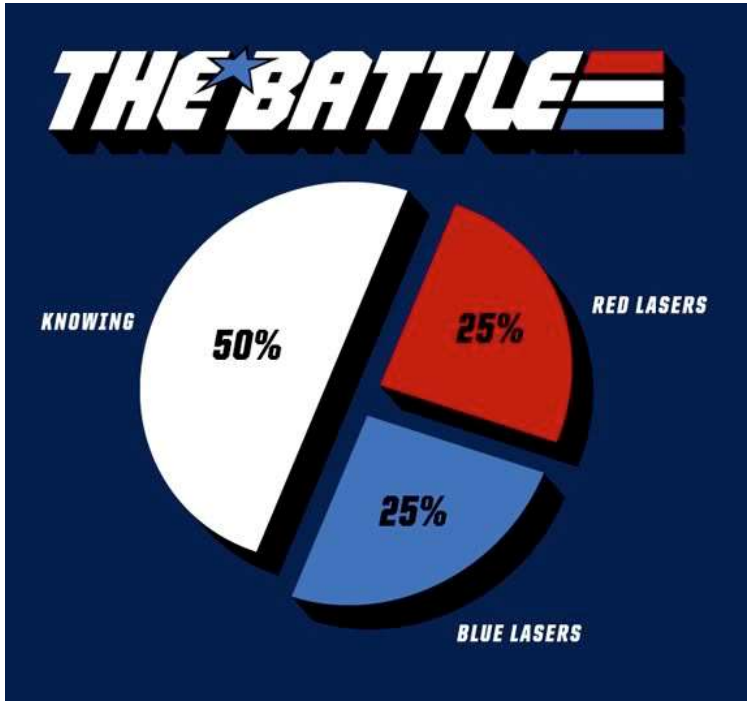
But again, remember this environment variable change is only valid for this script - it doesn't affect the shell's environment

Fork Bombs - Notes and Avoidance Techniques

- Yes, they're hilarious
- Under no circumstances should you be running systems development code on any non-OS class server!
- Consider the following warning signs that you might be about to do something dangerous, where if something goes wrong, your program might consume all of the system resources available and lock you and everyone else out:
  - You've written a loop that calls fork()
  - You've written code in which your child process creates another child process (a fork() within a forked process; these are usually not what you want)
  - You've written code in which your child process is starting up a loop

Fork Bombs - Notes and Avoidance Techniques

- Remember that you need to be really extra sure that you have termination methods built-in to your loops
- Consider having a variable set a flag called `forkNow` in your loop. Then, have a separate function call `fork()` because the flag value was set, with this function *also* resetting the flag value at the end
- Consider during testing, for example, adding an extra condition to a loop with a counting variable: if you hit 50 forks, say, then `abort()`



Process Management & Zombies

Benjamin Brewster

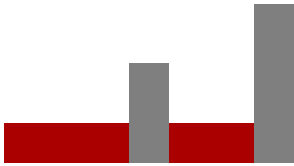
Running Processes

- How can we tell which processes are running? Use the `ps` command to get information about currently running processes

- Ps by itself is really boring, and not all that useful:

```
$ ps
  PID TTY          TIME CMD
18779 pts/8    00:00:00 bash
18934 pts/8    00:00:00 ps
```

- I've put together my two favorite ways to run it on the next few slides



ps For Me

Reminder: these aliases go into your ~/.bashrc file

```
$ alias
…
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 18776 root     Ss    0.2  4284     sshd: brewsteb [priv]
18776 18778 brewsteb S     0.0  2112       sshd: brewsteb@pts/8
18778 18779 brewsteb Ss    0.0  2044         -bash
18779 18911 brewsteb R+    4.0  1840           ps -eH -o ppid,pid,euser,stat,%cpu,rss,args
18779 18912 brewsteb S+    0.0   820           grep brewsteb
```

- PPID Parent Process ID
- PID Process ID
- EUSER Effective User ID
- STAT Execution State
- %CPU Percentage of CPU time this process occupies
- RSS Real Set Size - kilobytes of RAM in use by this process
- Command The actual command the user entered

ps For Me

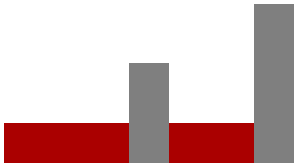
```
$ psme
 PPID   PID EUSER    STAT %CPU   RSS COMMAND
 4533 18776 root     Ss    0.2  4284     sshd: brewsteb [priv]
18776 18778 brewsteb S     0.0  2112       sshd: brewsteb@pts/8
18778 18779 brewsteb Ss    0.0  2044         -bash
18779 18911 brewsteb R+    4.0  1840           ps -eH -o ppid,pid,euser,stat,%cpu,rss,args
18779 18912 brewsteb S+    0.0   820           grep brewsteb
```

First State Character:

- `D` Uninterruptible sleep (usually IO)
- `R` Running or runnable (on run queue)
- `S` Interruptible sleep (waiting for an event to complete)
- `T` Stopped, either by a job control signal or because it is being traced
- `Z` Defunct ("zombie") process, terminated but not reaped by its parent

Second State Character (Optional):

- `<` High-priority (not nice to other users)
- `N` Low-priority (nice to other users)
- `L` Has pages locked into memory (for real-time and custom IO)
- `s` Is a session leader (closes all child processes on termination)
- `L` Is multi-threaded (Uses `pthread`)
- `+` Is in the foreground process group



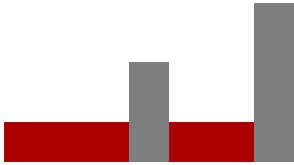
ps For All

$ alias

```
…
alias psall='ps -eH -o ppid,pid,euser,stat,%cpu,rss,args | awk '\''$1!=0'\'' | awk '\''$1!=1'\'' | awk
'\''$1!=2'\'' | more'
…
```

$ psall

```
 PPID   PID EUSER    STAT %CPU   RSS COMMAND
 …
4533 21922 root     Ss    0.0  4288     sshd: meadosc [priv]
21922 21936 meadosc  S     0.0  2128        sshd: meadosc@pts/11
21936 21937 meadosc  Ss    0.0  2024          -tcsh
21937 21962 meadosc  S+    0.0  1900            bash
4533 25083 root     Ss    0.4  4284     sshd: brewsteb [priv]
25083 25104 brewsteb S     0.0  2112        sshd: brewsteb@pts/8
25104 25105 brewsteb Ss    0.0  2040          -bash
25105 25761 brewsteb R+    8.0  1852            ps -eH -o ppid,pid,euser,stat,%cpu,rss,args
25105 25762 brewsteb S+    0.0   908            awk $1!=0
25105 25763 brewsteb S+    0.0   908            awk $1!=1
25105 25764 brewsteb S+    0.0   908            awk $1!=2
25105 25765 brewsteb S+    0.0   708            more
 4982  5194 root     Ss    0.0  5136     /opt/dell/srvadmin/sbin/dsm_sa_datamgrd
 5339  5340 root     Sl    0.1 244404     /opt/dell/srvadmin/sbin/dsm_om_connsvcd -run
 5461 25756 root     Zs    0.0     0     [check_nfs.sh] <defunct>
23087 23088 groveed  Ss+   0.0  1784     -bin/tcsh
…
```



Zombie?

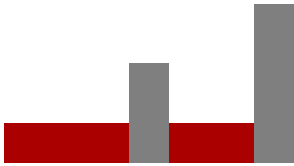
- When a child process terminates, but its parent does not wait for it, the process becomes known as a zombie (aka *defunct*)

Zombies!?!

- Child processes must report to their parents before their resources will be released by the OS

- If the parents aren't waiting for their children, the processes become the *living undead – forever consuming, forever enslaved to a non-life of waiting and watching.*

- The purpose of a zombie process is to retain the state that `wait()` can retrieve; they *want* to be harvested



Makin' Zombies

```
$ cat forkyouzombie.c
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>

void main()
{
        pid_t spawnPid = -5;
        int childExitStatus = -5;

        spawnPid = fork();
        switch (spawnPid)
        {
                case -1:
                        perror("Hull Breach!\n");
                        exit(1);
                        break;
                case 0:
                        printf("CHILD: Terminating!\n");
                        break;
                default:
                        printf("PARENT: making child a zombie for ten seconds;\n");
                        printf("PARENT: Type \"ps -elf | grep \'username\'\" to see the defunct child\n");
                        printf("PARENT: Sleeping...\n");
                        fflush(stdout);
                        sleep(10);
                        waitpid(spawnPid, &childExitStatus, 0);
                        break;
        }
        printf("This will be executed by both of us!\n");
        exit(0);
}
```

Make sure all text is outputted before sleeping

Output - Makin' Zombies

```
$ gcc -o forkyouzombie forkyouzombie.c
$ forkyouzombie
PARENT: making child a zombie for ten seconds;
PARENT: Type "ps -elf | grep 'username'" to see the defunct child
PARENT: Sleeping...
CHILD: Terminating!
This will be executed by both of us!
This will be executed by both of us!
```

Dramatic ten-second pause right here…

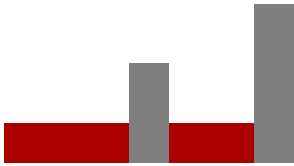
```
// In a second terminal…
$ ps -elf | grep 'brewsteb'
```

Nice, reasonable, short way to do a useful `ps`

```
4 S root     15296  4443  0  80   0 - 32719 unix_s 10:55 ?        00:00:00 sshd: brewsteb [priv]
5 S brewsteb 15298 15296  0  80   0 - 32719 poll_s 10:55 ?        00:00:00 sshd: brewsteb@pts/40
0 S brewsteb 15299 15298  0  80   0 - 30233 wait   10:55 pts/40   00:00:00 -bash
0 S brewsteb 17053 27991  0  80   0 -   981 hrtime 11:15 pts/9    00:00:00 forkyouzombie
1 Z brewsteb 17054 17053  0  80   0 -     0 exit   11:15 pts/9    00:00:00 [forkyouzombie] <defunct>
0 R brewsteb 17057 15299 12  80   0 - 30674 -      11:15 pts/40   00:00:00 ps -elf
0 S brewsteb 17058 15299  0  80   0 - 25829 pipe_w 11:15 pts/40   00:00:00 grep brewsteb
4 S root     27987  4443  0  80   0 - 32719 unix_s 08:51 ?        00:00:00 sshd: brewsteb [priv]
5 S brewsteb 27990 27987  0  80   0 - 32719 poll_s 08:51 ?        00:00:00 sshd: brewsteb@pts/9
0 S brewsteb 27991 27990  0  80   0 - 30234 wait   08:51 pts/9    00:00:00 -bash
```

How to Deal With Zombies

Zombies stay in the system until they are waited for



How to deal with Zombies

Zombies stay in the system until they are waited for

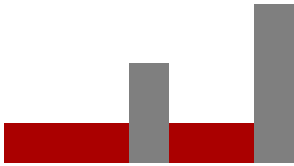
Orphan Zombies!

- If a parent process terminates *without* cleaning up its zombies, the zombies become orphan zombies
- Orphans are adopted by the `init` process (usually pid = 1) which periodically (in practice, very quickly) waits() for orphans
- Thus eventually, the orphan zombies die

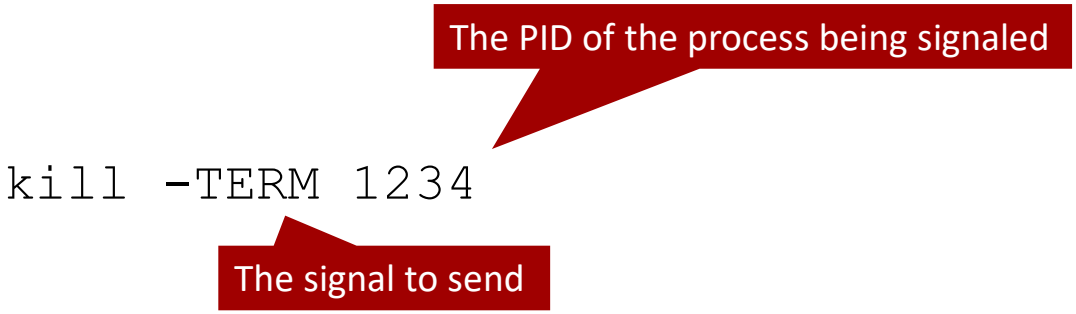
`kill`

- This UNIX command is used to kill programs
  - another old version is called `kfork`

- "kill" is really a misnomer – it really *just sends signals*



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

- We'll discuss more signals later, but you can use the signal `KILL` to tell a process to immediately terminate with no clean-up

top

- `top` allows you to view the processes running on the machine in real time - one of the few animated built-in programs

```
$ top
top - 14:14:34 up 34 days,  5:15,  9 users,  load average: 0.03, 0.18, 0.22
Tasks: 703 total,   1 running, 697 sleeping,   4 stopped,   1 zombie
Cpu(s):  0.1%us,  0.1%sy,  0.0%ni, 99.8%id,  0.0%wa,  0.0%hi,  0.0%si,  0.0%st
Mem:  65922540k total,  7876576k used, 58045964k free,   663988k buffers
Swap:  2588668k total,        0k used,  2588668k free,  5258716k cached

  PID USER      PR  NI  VIRT  RES  SHR S %CPU %MEM    TIME+  COMMAND
27609 brewsteb  20   0 27884 1796  996 R  3.4  0.0   0:00.10 top
    1 root      20   0 33656 1624 1292 S  0.0  0.0   1:41.47 init
    2 root      20   0     0    0    0 S  0.0  0.0   0:09.18 kthreadd
    3 root      RT   0     0    0    0 S  0.0  0.0   0:21.23 migration/0
    4 root      20   0     0    0    0 S  0.0  0.0   0:14.47 ksoftirqd/0
    5 root      RT   0     0    0    0 S  0.0  0.0   0:00.00 stopper/0
…
```

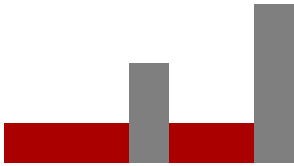
Diagnosing a Slow CPU

- The *uptime* command shows the average number of runnable processes over several different periods of time (the same info top displays)

```
$ uptime
1:23pm up 25 day(s), 5:59, 72 users, load average: 0.18, 0.19, 0.20
```

- This shows the average number of runnable (the current running process plus the queue of processes waiting to be run) or uninterruptable (waiting for IO) processes over the last 1, 5 and 15 minutes

- If uptime is showing that your runnable queue is consistently *larger than the number of cores*, your CPU is a bottleneck and is causing slow-down



Diagnosing a Slow CPU - Number of Cores?

```
$ cat /proc/cpuinfo
processor       : 0
vendor_id       : GenuineIntel
cpu family      : 6
model           : 45
model name      : Intel(R) Xeon(R) CPU E5-2665 0 @ 2.40GHz
stepping        : 7
microcode       : 1808
cpu MHz         : 2399.993
cache size      : 20480 KB
physical id     : 0
siblings        : 16
core id         : 0
cpu cores       : 8
apicid          : 0
initial apicid  : 0
fpu             : yes
fpu_exception   : yes
…
```

Diagnosing a Slow CPU - Example - Single Core

```
$ uptime
14:33:04 up 34 days,  5:34, 10 users,  load average: 0.05, 0.15, 0.20
```

This CPU is the champ… or it's not being given anything to do

```
$ uptime
14:33:04 up 34 days,  5:34, 10 users,  load average: 0.88, 1.03, 0.96
```

This CPU is at max - time to upgrade!

```
$ uptime
14:33:04 up 34 days,  5:34, 10 users,  load average: 4.79, 7.23, 6.44
```

It's 3am, and your server is borked; start paging everyone!

Diagnosing a Slow CPU - Example - Octo Core

```
$ uptime
14:33:04 up 34 days,  5:34, 10 users,  load average: 0.05, 0.15, 0.20
```

Your CPU is bored, and you wasted all the money; but hey, headroom for games!

```
$ uptime
14:33:04 up 34 days,  5:34, 10 users,  load average: 7.99, 8.10, 7.94
```

This CPU is handling processes exactly as fast as it gets them - time to make it more betterer

```
$ uptime
14:33:04 up 34 days,  5:34, 10 users,  load average: 39.90, 41.54, 40.72
```

You're so fired

Job Control

- How do we start a program, and *still retain access* to the command line for the next program we want to run?
- Can we run multiple processes at once?
- This is called Job Control in UNIX-speak

Foreground/Background

- There can be only one shell ***foreground*** process – it's the one you're currently interacting with
- If you're at the command prompt, then your foreground program is the shell itself!

- Processes in the ***background*** can still be executing, but they can also be in any number of stopped states:

First State Character:

- `D` Uninterruptible sleep (usually IO)
- `R` Running or runnable (on run queue)
- `S` Interruptible sleep (waiting for an event to complete)
- `T` Stopped, either by a job control signal or because it is being traced
- `Z` Defunct ("zombie") process, terminated but not reaped by its parent

Foreground/Background in Reality

- There really isn't any difference between processes in these two states; its merely shell nomenclature used to distinguish between them
- When a user enters a command that is intended to run in the foreground (i.e. a normal command), the process started runs to completion *before* the user is prompted again
- When a user enters a command that is intended to run in the background (see later slides), the user is *immediately* prompted again after the process is executed
- In other words, control input to the terminal is not interrupted by a background process

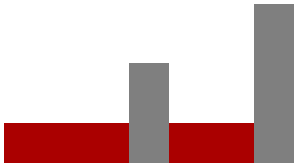
Start Backgrounded

- Here's how to start a program in the background in the first place:

```
$ ping www.oregonstate.edu &
```

- The ampersand means to start in the background, and must be the last character

- Note that stdout and stderr are still going to the terminal for that process, and stdin might be too if the shell is badly programmed



Stopping a Process

- Sending the `TSTP` signal stops (not terminates) a process, and puts it into the background
  - Control-z also sends this signal

`ping` is now the foreground process, and I can't enter commands anymore

```
$ ping www.oregonstate.edu
PING www.orst.edu (128.193.4.112) 56(84) bytes of data.
64 bytes from www.orst.edu (128.193.4.112): icmp_seq=1 ttl=250 time=0.362 ms
64 bytes from www.orst.edu (128.193.4.112): icmp_seq=2 ttl=250 time=0.321 ms
64 bytes from www.orst.edu (128.193.4.112): icmp_seq=3 ttl=250 time=0.324 ms
64 bytes from www.orst.edu (128.193.4.112): icmp_seq=4 ttl=250 time=0.328 ms
^Z
[1]+  Stopped                 ping www.oregonstate.edu
$
```

Our shell is once again the foreground process

jobs

- Use the `jobs` command to see what you're running:

```
$ ping www.oregonstate.edu
PING www.orst.edu (128.193.4.112) 56(84) bytes of data.
64 bytes from www.orst.edu (128.193.4.112): …
64 bytes from www.orst.edu (128.193.4.112): …
^Z
[3]+  Stopped                 ping www.oregonstate.edu
$ jobs -l
[1]   31314 Stopped                 ping www.oregonstate.edu
[2]-  31317 Stopped                 ping www.oregonstate.edu
[3]+  31327 Stopped                 ping www.oregonstate.edu
$ kill -KILL 31327
[3]+  Killed                  ping www.oregonstate.edu
$ kill -KILL %1
[1]-  Killed                  ping www.oregonstate.edu
```

The `-l` switch adds the PID

The – symbol means it was the second-to-last process put in the background

The + symbol means it was the last process put in the background

"Job 1"

`fg`

- Use the job numbers provided by `jobs` to manipulate processes

- Bring job 1 from the background to the foreground, and start it running again:

  ```
  fg %1
  ```

- Bring most recent backgrounded job to the foreground, and start it running again:

  ```
  fg
  ```

bg

- Start a specific stopped program that is currently in the background (and keep it in the background):

  ```
  bg %1
  ```

- Start the most recently stopped program in the background (and keep it in the background):

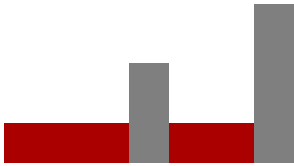
  ```
  bg
  ```

Who's Got Control of stdout?

- Be advised – background processes can still write to any file including stdout & stderr!

- Jobs demo:

```
1. ping www.oregonstate.edu
2. CTRL-Z
3. jobs
4. fg %1
5. CTRL-Z
6. jobs
7. bg %1
8. ps
9. CTRL-Z CTRL-Z CTRL-Z (doesn't do anything)
10. fg %1
11. CTRL-C
```



You're Suspended

- Suspend a process that is currently running in the background when you're at the shell

Send stderr and stdout somewhere other than the terminal

Background this command!

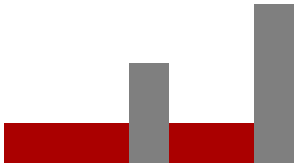
Send the "stop" signal

```
$ ping www.oregonstate.edu 2>/dev/null 1>logfile &
[1] 1660
$ jobs
[1]+  Running                 ping www.oregonstate.edu 2> /dev/null > /dev/null &
$ kill -TSTP %1
[1]+  Stopped                 ping www.oregonstate.edu 2> /dev/null > /dev/null
$ jobs
[1]+  Stopped                 ping www.oregonstate.edu 2> /dev/null > /dev/null
```

`history` - a Command Visibility Utility

- The history command provides a listing of your previous commands:

```
$ history 5
 1012  jobs
 1013  psme
 1014  top
 1015  jobs
 1016  history 5
```



Execute a Previous Command

```
$ history 3
 1030  jobs
 1031  psme
 1032  history 3
$ !1030
jobs
$ history 3
 1032  history 3
 1033  jobs
 1034  history 3
$ !-2
jobs
$ !!
jobs
$ history 3
 1034  history 3
 1035  jobs
 1036  history 3
```

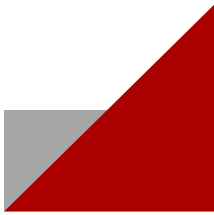
Note that no exclamation marks are in the history list - only the actual commands, even when repeated with the ! operator

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?

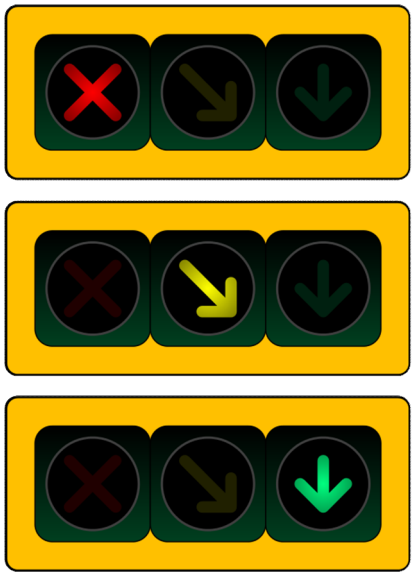


Signals

- 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

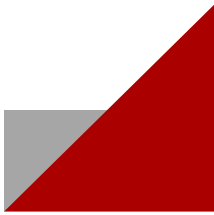
- Critical point: signals tell a process to DO something - to take an action because of a user command or an event

- 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



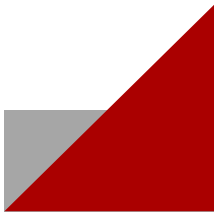
Uses for Signals: Kernel to Process

- 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

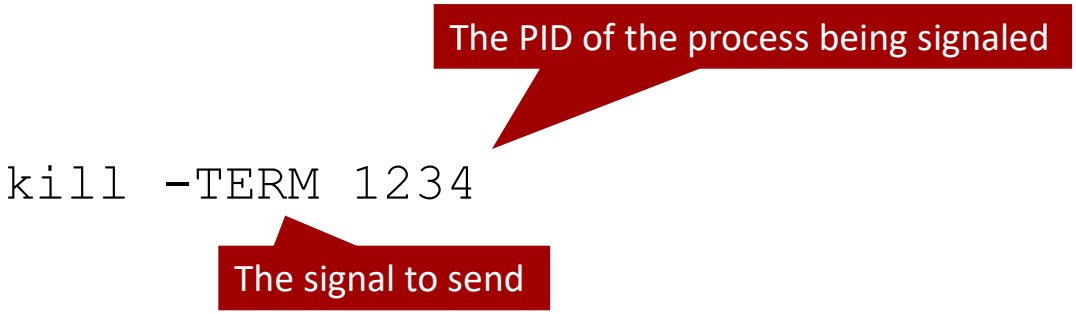


Signal Dictionary - Termination

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
```

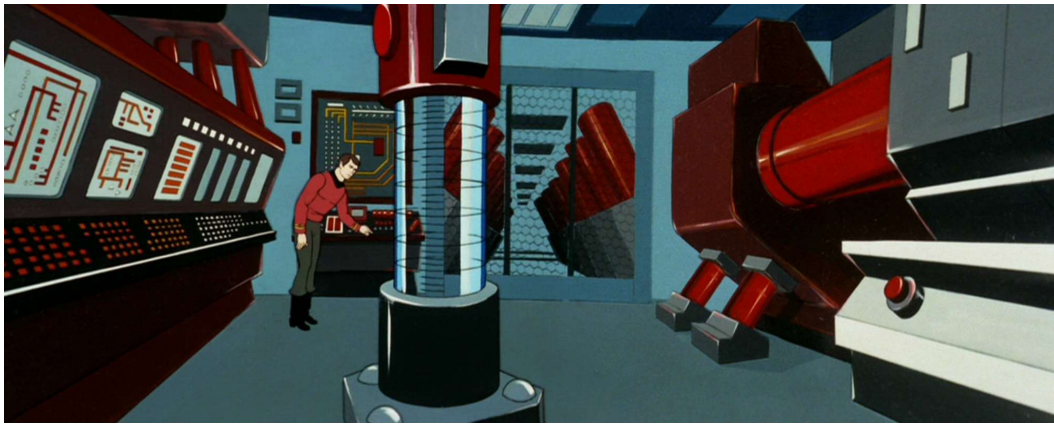
```
$ psme
 PPID   PID EUSER    STAT %CPU   RSS COMMAND
4533 18174 root     Ss    0.0  4280     sshd: brewsteb [priv]
18174 18187 brewsteb S     0.0  2108       sshd: brewsteb@pts/9
18187 18188 brewsteb Ss    0.0  2104         -bash
18188 19200 brewsteb R+    1.0  1844           ps -eH -o ppid,pid,euser,stat,%cpu,rss,args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

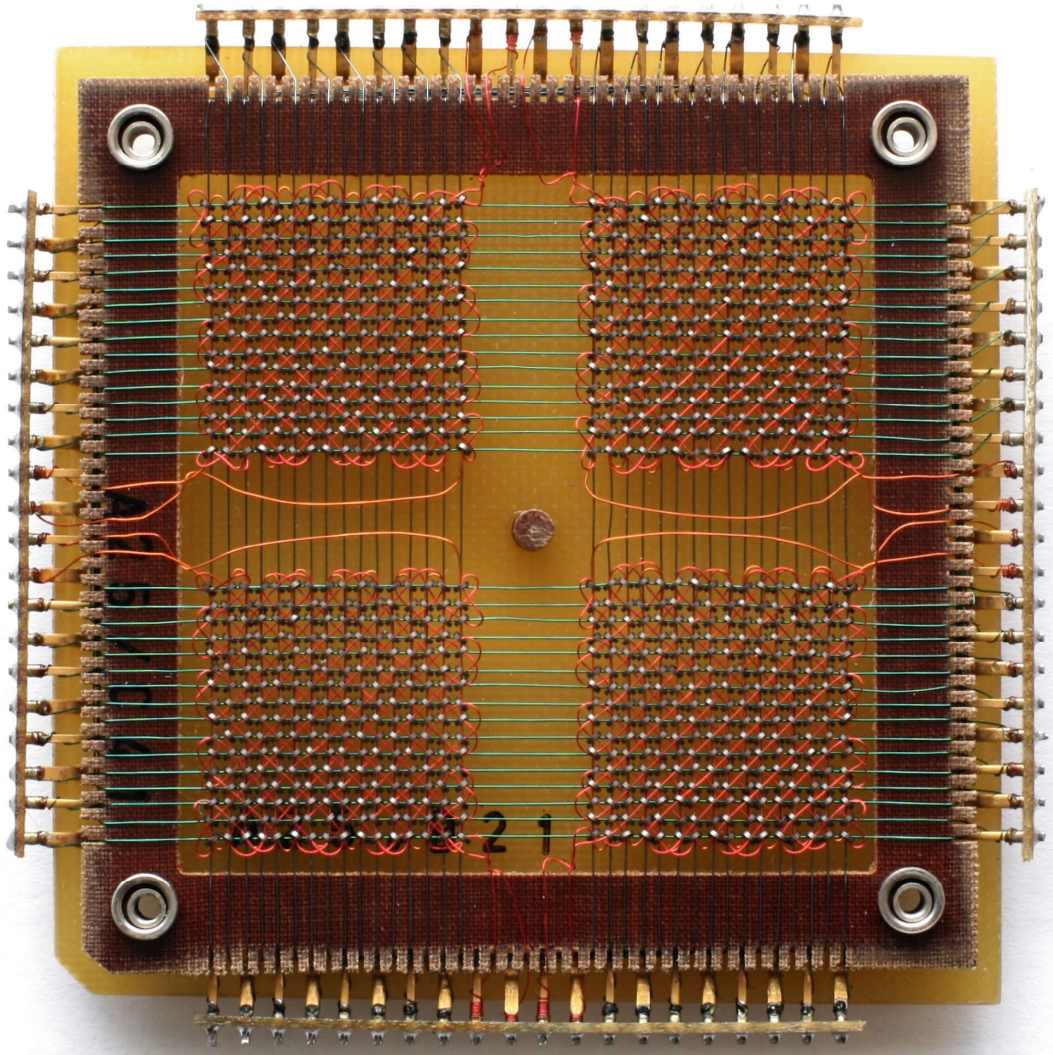
Core uses tiny magnetic toroids (rings), the cores, through which wires are threaded to write and read information. Each core represents one bit of information. The cores can be magnetized in two different ways (clockwise or counterclockwise) and the bit stored in a core is zero or one depending on that 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

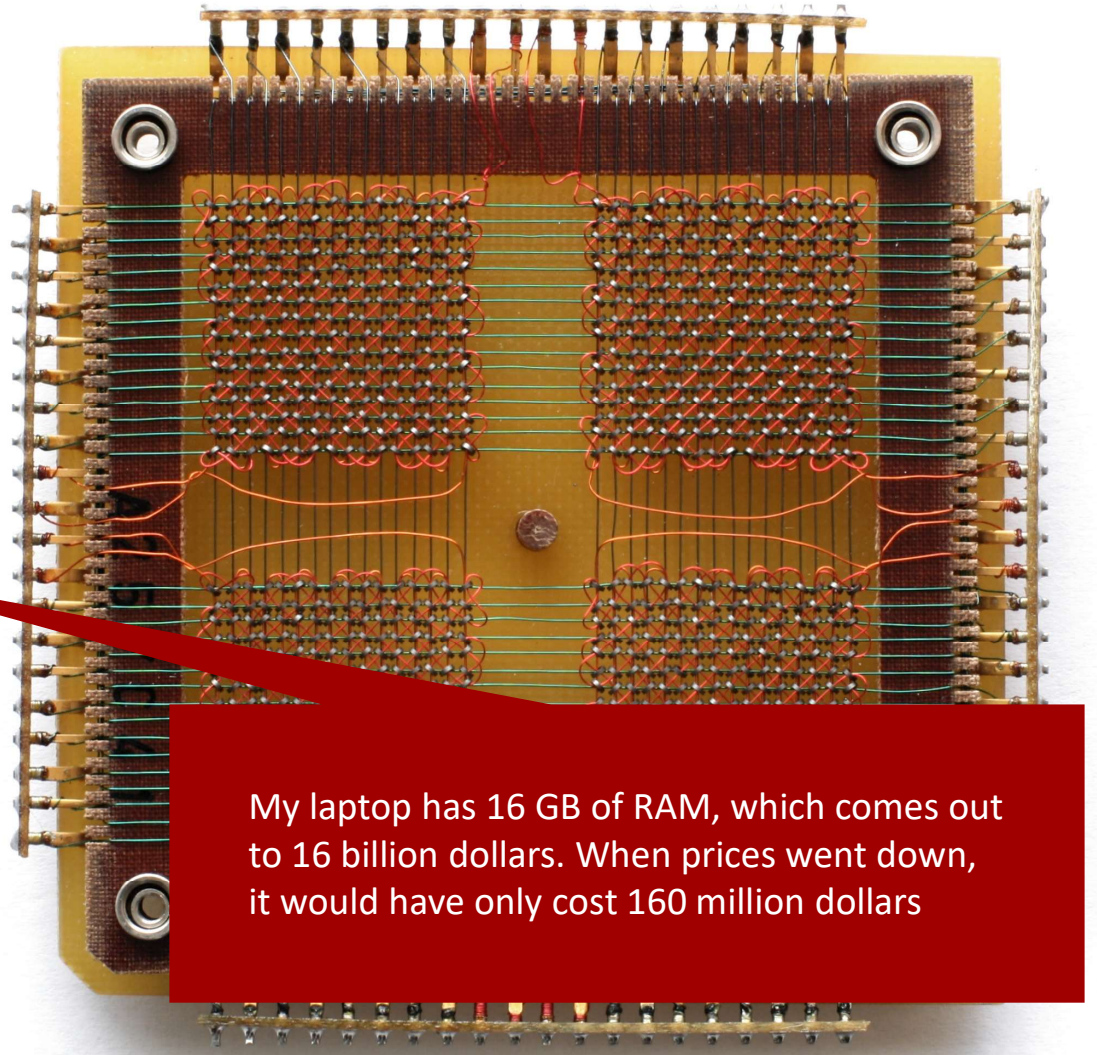


By Konstantin Lanzet - received per EMailCamera: Canon EOS 400D, CC BY-SA 3.0, https://commons.wikimedia.org/w/index.php?curid=7025

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

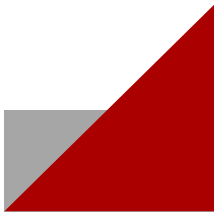
By Konstantin Lanzet - received per EMailCamera: Canon EOS 400D, CC BY-SA 3.0, https://commons.wikimedia.org/w/index.php?curid=7025

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

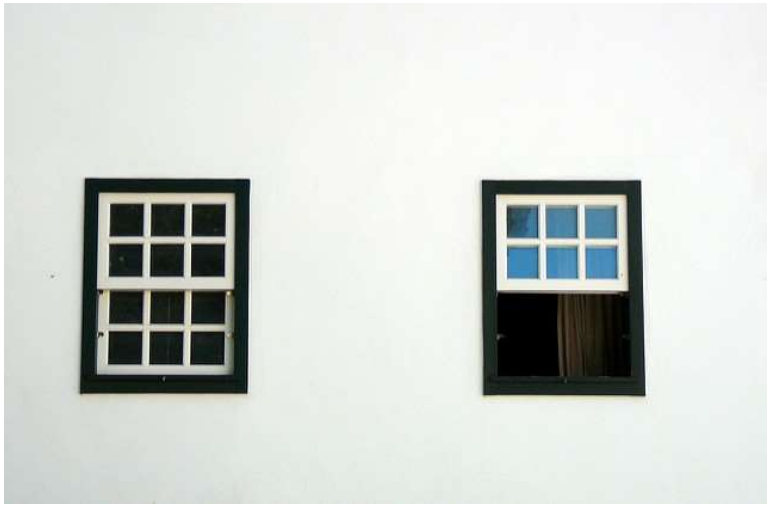
- The next bunch of slides all discuss these signal handling functions, but first a few utility functions…



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

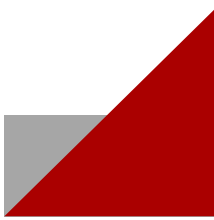
- Sometimes a process has nothing to do, so you consider calling `sleep()`, but you want it to be able to respond to signals, which it can't do in `sleep()`

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



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

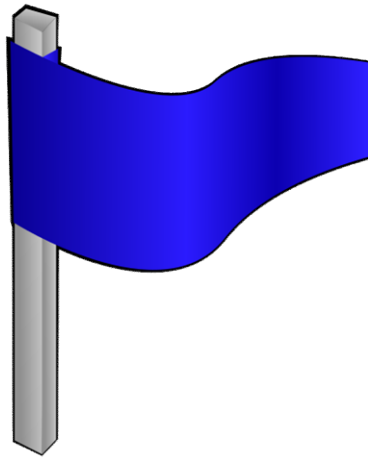
- If a signal is set to be ignored, then the `pause()` continues to be in effect
- If a signal causes a termination, `pause()` does nothing (because the process dies)
- If a signal is caught, the appropriate signal handler function will be called. After the signal handler function is done, `pause()` 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()`

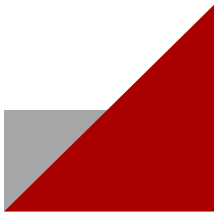
- You can send yourself a specified signal immediately with `raise()`:
  - `int raise(int signal);`

- 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*);
};
```

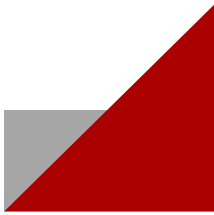
- The first attribute should be set to one of three values:
  - SIG_DFL :: Take the default action for the signal
  - SIG_IGN :: Ignore the signal
  - A pointer to a function that should be called when this signal is received (see next slide)

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

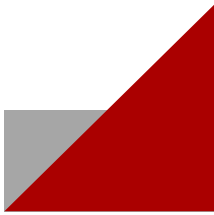
Does nothing, it's disabled!

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

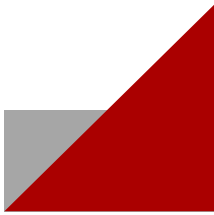


Blocking Signals

- 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



More UNIX I/O

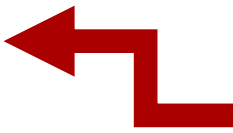
Benjamin Brewster

Open File Inheritance

- When you `exec…()` a process, you replace the current process with a new one, but the files are still open and accessible to the new process

- This may not be what you want!

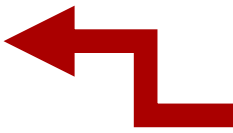
- In this lecture, we discuss other IPC methods and UNIX I/O. We'll cover:
  - Close On Exec
  - File redirection in C
  - Pipes



I/O redirection

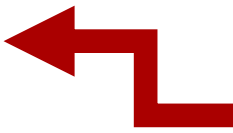
- We saw I/O redirection in the shell:
  - `ls > file`
  - `stats < file1`
  - `cat longfile | more`
  - `find . -name "paper" -print 2> /dev/null`
  - `echo "an error occurred" 1>&2`

- I/O redirection is possible *because* open files are shared across `fork()` and `exec…()`



I/O redirection

- I/O redirection is possible *because* open files are shared across `fork()` and `exec…()`
- Each child process has the same files open as its parent, as the file descriptors are the same file descriptors in both
- Note this important point: both parent and child read & write function calls move the *same* file pointer for the shared file descriptor!
  - To prevent this, have one of the processes close and re-open the file



Sharing File Pointers - Example 1 of 2

$ cat forkFPsharing.c

```
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <fcntl.h>

void main()
{
  pid_t forkPID;
  int childExitMethod;
  int fileDescriptor;
  char *newFilePath = "./newFile.txt";
  char readBuffer[8];
  memset(readBuffer, '\0', sizeof(readBuffer));

  printf("PARENT: Opening file.\n");
  fileDescriptor = open(newFilePath, O_RDWR | O_CREAT | O_TRUNC, S_IRUSR | S_IWUSR);
  if (fileDescriptor == -1) { printf("Hull breach - open() failed on \"%s\"\n", newFilePath); exit(1); }

  printf("PARENT: Writing 01234 to file.\n");
  write(fileDescriptor, "01234", 5);
  printf("PARENT: New FP position (all FP positions are zero-indexed): %d\n", lseek(fileDescriptor, 0, SEEK_CUR));
  fflush(stdout);

  printf("PARENT: Spawning child.\n");
  forkPID = fork();
```

The child process will have `fileDescriptor` open, with the SHARED file pointer!

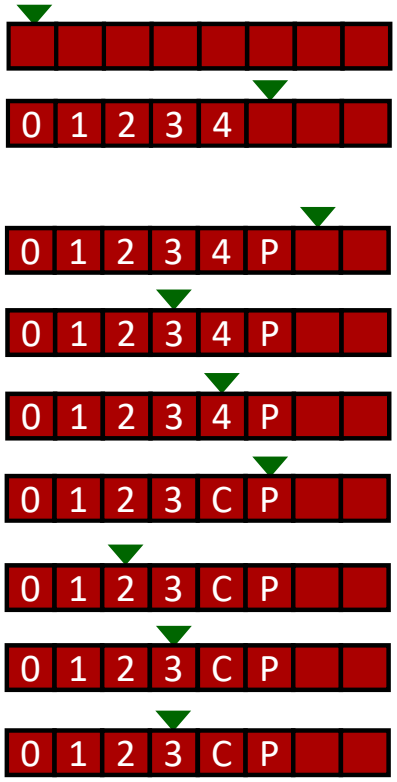
Sharing File Pointers - Example 2 of 2

This code and trace statements just reads and writes to a file open and shared in both the parent and child processes

```
  switch (forkPID)
  {
  case -1: perror("Hull Breach!"); exit(1); break;
  case 0:
    printf("CHILD: Writing C to file.\n"); fflush(stdout);
    write(fileDescriptor, "C", 1);
    printf("CHILD: After write, new FP position: %d\n", lseek(fileDescriptor, 0, SEEK_CUR)); fflush(stdout);
    printf("CHILD: lseek back 3 chars.\n"); fflush(stdout);
    printf("CHILD: New FP position: %d\n", lseek(fileDescriptor, -3, SEEK_CUR)); fflush(stdout);
    printf("CHILD: Reading char.\n"); fflush(stdout);
    read(fileDescriptor, &readBuffer, 1);
    printf("CHILD: After read, new FP position: %d, char read was: %c\n", lseek(fileDescriptor, 0, SEEK_CUR), readBuffer[0]); fflush(stdout);
    break;
  default:
    printf("PARENT: Writing P to file.\n"); fflush(stdout);
    write(fileDescriptor, "P", 1);
    printf("PARENT: After write, new FP position: %d\n", lseek(fileDescriptor, 0, SEEK_CUR)); fflush(stdout);
    printf("PARENT: lseek back 3 chars.\n"); fflush(stdout);
    printf("PARENT: New FP position: %d\n", lseek(fileDescriptor, -3, SEEK_CUR)); fflush(stdout);
    printf("PARENT: Reading char.\n"); fflush(stdout);
    read(fileDescriptor, &readBuffer, 1);
    printf("PARENT: After read, new FP position: %d, char read was: %c\n", lseek(fileDescriptor, 0, SEEK_CUR), readBuffer[0]); fflush(stdout);
    waitpid(forkPID, &childExitMethod, 0);
    lseek(fileDescriptor, 0, SEEK_SET);
    read(fileDescriptor, &readBuffer, 7);
    printf("PARENT: child terminated; file contents: %s\n", readBuffer); fflush(stdout);
    break;
  }
}
```

Sharing File Pointers - Results

```
$ gcc -o forkFPsharing forkFPsharing.c
$ forkFPsharing
PARENT: Opening file.
PARENT: Writing 01234 to file.
PARENT: New FP position (all FP positions are zero-indexed): 5
PARENT: Spawning child.
PARENT: Writing P to file.
PARENT: After write, new FP position: 6
PARENT: lseek back 3 chars.
PARENT: New FP position: 3
PARENT: Reading char.
PARENT: After read, new FP position: 4, char read was: 3
CHILD: Writing C to file.
CHILD: After write, new FP position: 5
CHILD: lseek back 3 chars.
CHILD: New FP position: 2
CHILD: Reading char.
CHILD: After read, new FP position: 3, char read was: 2
PARENT: child terminated; file contents: 0123CP
```

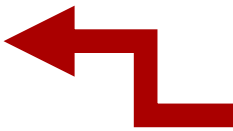


Important Background Review

- The kernel opens stdin, stdout, and stderr automatically for every process created:
  - File descriptor 0 is stdin
  - File descriptor 1 is stdout
  - File descriptor 2 is stderr

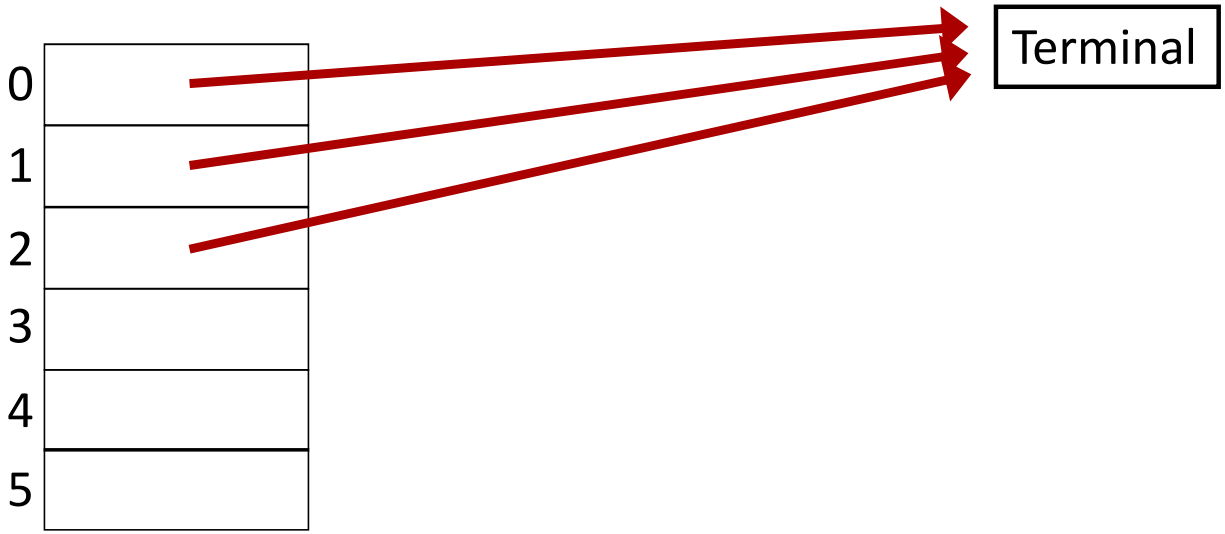
- They default to reading and writing to the terminal

- The trick: you can change where the standard I/O streams are pointing to at any time before the `exec…()` call, including after the `fork()`



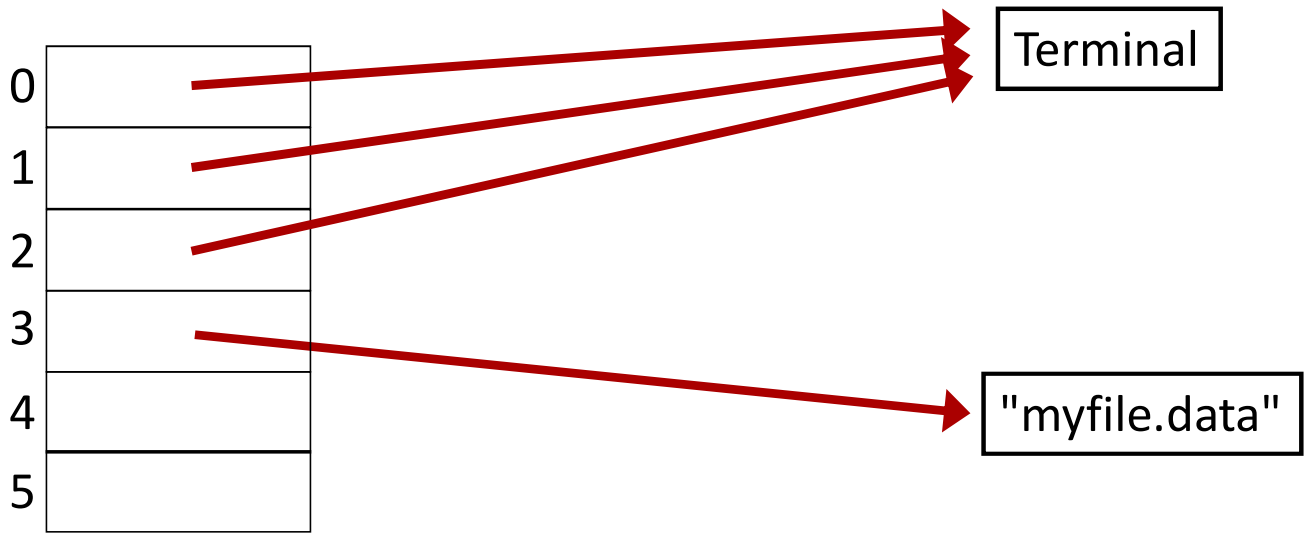
Redirecting stdout

How all processes start:



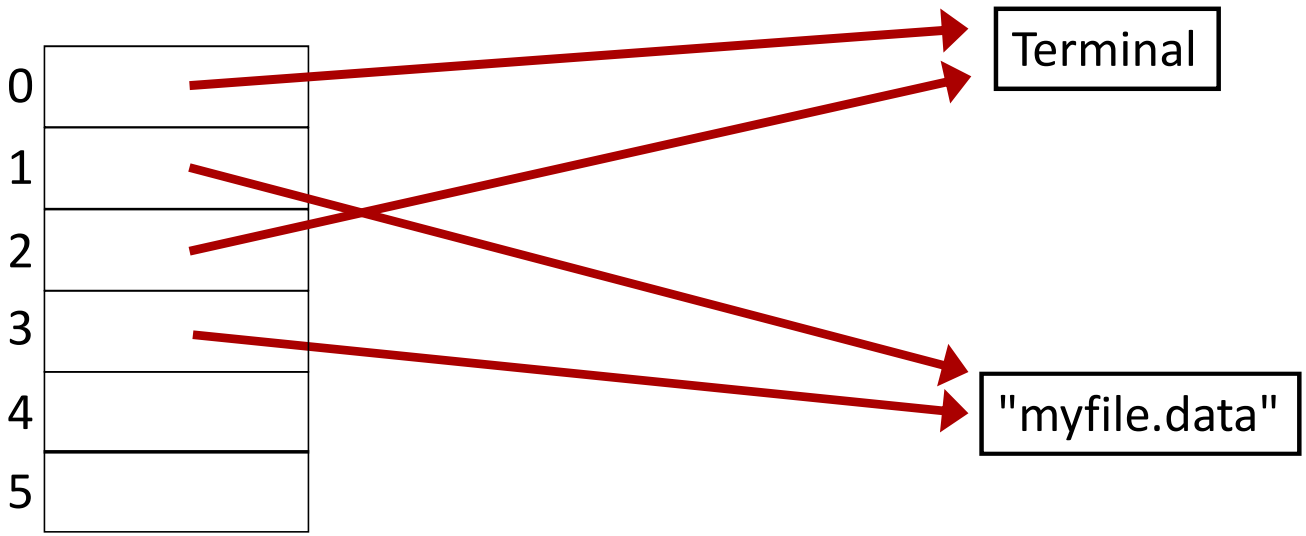
Redirecting stdout

1. First open the new file



Redirecting stdout

2. Call `dup2()` to change fd 1 to point where fd 3 points: `dup2(3, 1);`



stdout, merely another name for fd 1, can now be used to access “myfile.data”

Redirecting stdout

Set FD `1` (stdout) to point to the same place that `targetFD` points.

So, anything written to stdout (like with `printf()`) will go to `targetFD`, which is the passed in filename

```
$ cat redirectToFile.c
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>

int main(int argc, char* argv[])
{
        if (argc == 1)
        {
                printf("Usage: redirectToFile <filename to redirect stdout to>\n");
                exit(1);
        }

        int targetFD = open(argv[1], O_WRONLY | O_CREAT | O_TRUNC, 0644);
        if (targetFD == -1) { perror("open()"); exit(1); }
        printf("targetFD == %d\n", targetFD); // Written to terminal

        int result = dup2(targetFD, 1);
        if (result == -1) { perror("dup2"); exit(2); }
        printf("targetFD == %d, result == %d\n", targetFD, result); // Written to file

        return(0);
}
$ gcc -o redirectToFile redirectToFile.c
$ redirectToFile
Usage: redirectToFile <filename to redirect stdout to>
$ redirectToFile test.junk
targetFD == 3
$ cat test.junk
targetFD == 3, result == 1
```

Redirecting stdout & stdin with execlp()

```
$ cat sortViaFiles.c
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <unistd.h>

int main(int argc, char* argv[])
{
        int sourceFD, targetFD, result;

        if (argc != 3)
        {
                printf("Usage: sortViaFiles <input filename> <output filename>\n");
                exit(1);
        }

        sourceFD = open(argv[1], O_RDONLY);
        if (sourceFD == -1) { perror("source open()"); exit(1); }
        printf("sourceFD == %d\n", sourceFD); // Written to terminal

        targetFD = open(argv[2], O_WRONLY | O_CREAT | O_TRUNC, 0644);
        if (targetFD == -1) { perror("target open()"); exit(1); }
        printf("targetFD == %d\n", targetFD); // Written to terminal

        result = dup2(sourceFD, 0);
        if (result == -1) { perror("source dup2()"); exit(2); }
        result = dup2(targetFD, 1);
        if (result == -1) { perror("target dup2()"); exit(2); }

        execlp("sort", "sort", NULL);
        return(3);
}
```

Set FD `0` (stdin) to point to the same place that `sourceFD` points.

So, anything in stdin (like the contents of the input file) will be used by this program

Set FD `1` (stdout) to point to the same place that `targetFD` points.

So, anything written to stdout (like the result of sort) will go to `targetFD`, which is the passed in output filename

`execlp()` now starts with stdin and stdout pointing to files, which are used by `sort`

Redirecting stdout & stdin with execlp()

```
$ cat sortViaFiles.c
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <unistd.h>

int r
{
```

Results:

```
$ gcc -o sortViaFiles sortViaFiles.c
$ echo -e "3\n1\n2" > junkinput
$ cat junkinput
3
1
2
$ sortViaFiles junkinput junkoutput
sourceFD == 3
targetFD == 4
$ cat junkoutput
1
2
3
```

```
name> <output filename>\n");


exit(1); }
n to terminal

_TRUNC, 0644);
exit(1); }
n to terminal


it(2); }

it(2); }

    execlp("sort", "sort", NULL);
    return(3);
}
```

Set FD `0` (stdin) to point to the same place that `sourceFD` points.

So, anything in stdin (like the contents of the input file) will be used by this program

Set FD `1` (stdout) to point to the same place that `targetFD` points.

So, anything written to stdout (like the result of sort) will go to `targetFD`, which is the passed in output filename

`execlp()` now starts with stdin and stdout pointing to files, which are used by `sort`

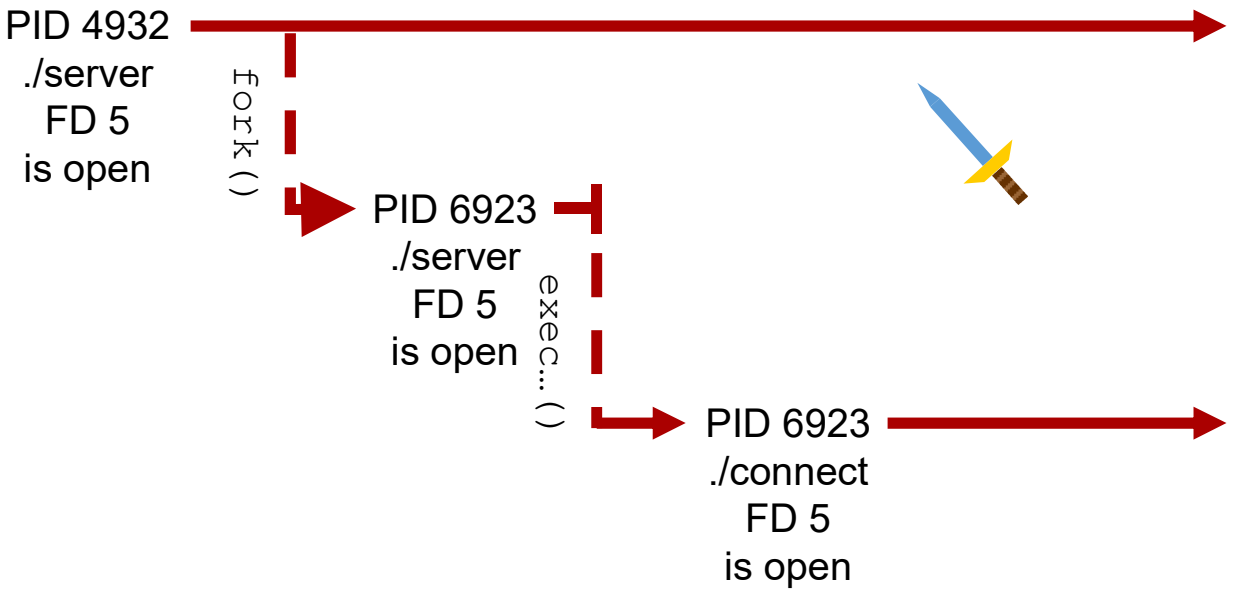
Close On Exec Details

- A flag specified in an open() call that tells the kernel to close any open files when/if the process gets `exec..()`'d
- Why do we care? Because open files are inherited by child processes
  - Thus the file pointer is shared (where the file is currently being read and written to)
  - Secure and/or sensitive data is shared with the new process
- Specific to only that file you added the flag to
- This is inherited through fork, so if the parent specifies it, a child that later calls exec…() will trigger the close on exec flag

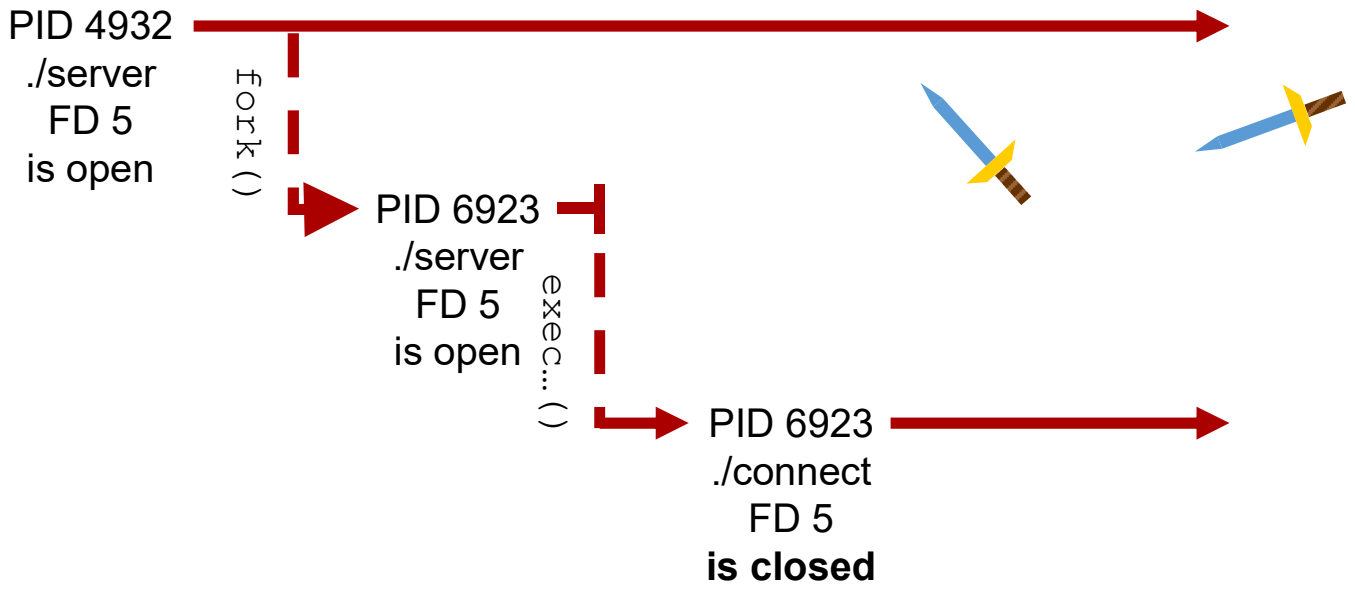
Close on exec example

```
#include <fcntl.h>
...
int fd;
fd = open("file", O_RDONLY);
…
fcntl(fd, F_SETFD, FD_CLOEXEC);
…
// exec...
```

Normally…

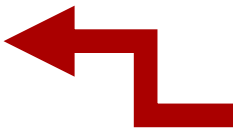

PID 4932
./server
FD 5
is open
fork()
PID 6923
./server
FD 5
is open
exec...()
PID 6923
./connect
FD 5
is open

With Close On Exec



Real Inter-Process Communication (IPC)

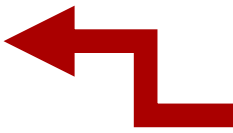
- IPC methods in UNIX
  - Intermediate/temporary files :: Often used together with I/O redirection
  - Pipes :: IPC between two processes forked by a common ancestor process
  - FIFOs (named pipes) :: communication between any two processes on the same machine
  - Message queues :: communication between any two processes on the same machine
    - Not common; older System V libraries were replaced with POSIX libraries
    - Not a simple byte stream: In Linux, the POSIX queues can be mounted as a filesystem, with each message appearing as a file; can then use `ls`, `rm`, etc.
    - Supports message categories - often used for priorities
  - Sockets :: communication between any two processes, potentially separated by a network



Between-Process IPC - Intro to Pipes

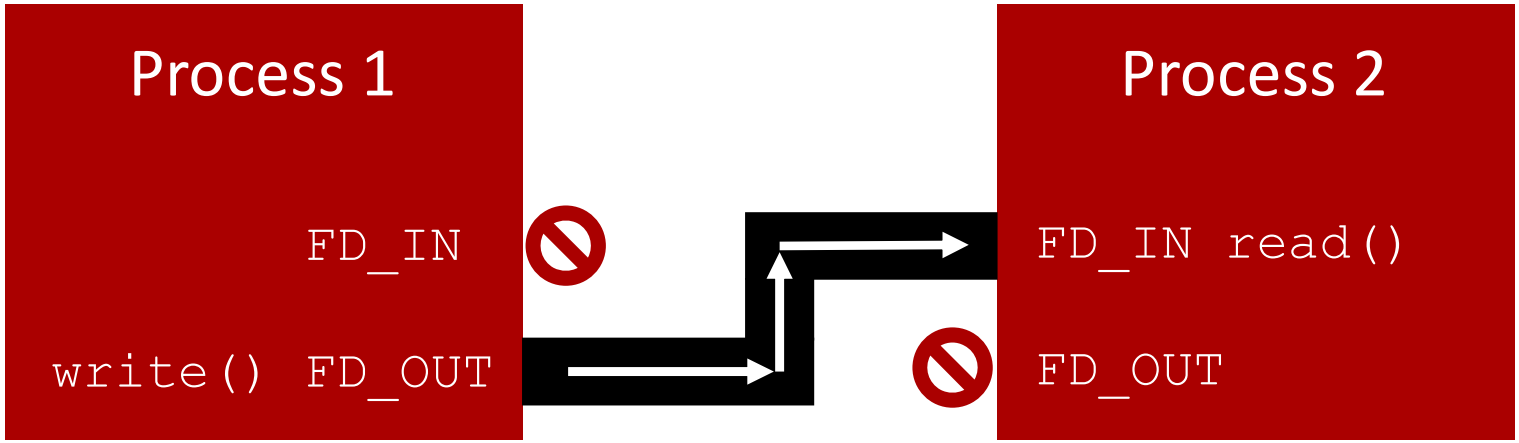
- I/O redirection with `dup2()` allows you to redirect input and output between processes and files…

- But how do we *redirect input* between processes and other processes on the same machine?
  - We could use temporary/intermediate files, but:
    - Writes to disk are slow
    - No fast & efficient way to track when the other process is ready to receive or send new data other than some sort of semaphore library
  - Better answer: use pipes!



Pipes

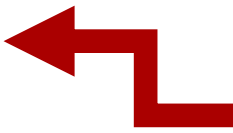
- Pipes provide a way to connect a write-only file descriptor in one process to an read-only file descriptor in another process



- `write()` puts bytes in the pipe, `read()` takes them out

Creating a Pipe

- Pipes are possible because file descriptors are shared across `fork()` and `exec…()`
- A parent process creates a pipe
  - Results in two new open file descriptors, one for input and one for output
- The parent process calls `fork()` and possibly `exec…()`
  - Parent and child have the file descriptors created with the pipe
- The child process now reads from the input file descriptor, and the parent process writes to the output file descriptor
  - or vice-versa



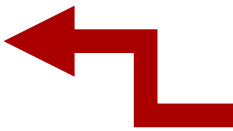
The `pipe()` Function

- You pass `pipe()` an array of two integers, where it stores the two new open file descriptors that it creates

- The first is the input file descriptor, and the second is the output file descriptor

- One of the descriptors should be used by the parent process and the other should be used by the child process

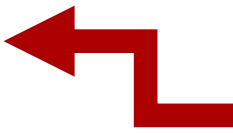
We'll talk about how to use a pipe to communicate between two non-decendent processes later



Flow Control with `read()`

- `read()` succeeds if data is available
  - Receives the data and returns immediately
  - The return value of `read()` tells you how many bytes were read, which *may be less than you requested*

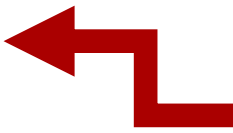
- If data is not available, read() will *block* waiting for data (your process execution is suspended until data arrives)
  - `read()` is a system call



Flow Control with `write()`

- Similarly, write will not return until all the data has been written
  - `write()` is a system call

- Pipes have a certain size
  - Only so much data will fit in a pipe (typically 64K, but can be changed)
  - If the pipe fills up, and there is no more room, write() will *block* until space becomes available (ie somebody `read`s the data from the pipe)



pipe() Example

```
$ cat pipeNfork.c
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <string.h>
void main()
{
        int r, pipeFDs[2];
        char completeMessage[512], readBuffer[10];
        pid_t spawnpid;

        if (pipe(pipeFDs) == -1) { perror("Hull Breach!"); exit(1); } // Create the pipe with error check

        spawnpid = fork(); // Fork the child, which will write into the pipe
        switch (spawnpid)
        {
        case 0:  // Child
                close(pipeFDs[0]);  // close the input file descriptor
                write(pipeFDs[1], "CHILD: Hi parent!@@", 19); // Write the entire string into the pipe
                exit(0); break; // Terminate the child
        default:  // Parent
                close(pipeFDs[1]);  // close output file descriptor
                memset(completeMessage, '\0', sizeof(completeMessage)); // Clear the buffer

                while (strstr(completeMessage, "@@") == NULL) // As long as we haven't found the terminal...
                {
                        memset(readBuffer, '\0', sizeof(readBuffer)); // Clear the buffer
                        r = read(pipeFDs[0], readBuffer, sizeof(readBuffer) - 1); // Get the next chunk
                        strcat(completeMessage, readBuffer); // Add that chunk to what we have so far
                        printf("PARENT: Message received from child: \"%s\", total: \"%s\"\n", readBuffer, completeMessage);
                        if (r == -1) { printf("r == -1\n"); break; } // Check for errors
                        if (r == 0) { printf("r == 0\n"); break; }
                }
                int terminalLocation = strstr(completeMessage, "@@") - completeMessage; // Where is the terminal
                completeMessage[terminalLocation] = '\0'; // End the string early to wipe out the terminal
                printf("PARENT: Complete string: \"%s\"\n", completeMessage);
                break;
        }
}
$ pipeNfork
PARENT: Message received from child: "CHILD: Hi", total: "CHILD: Hi"
PARENT: Message received from child: " parent!@", total: "CHILD: Hi parent!@"
PARENT: Message received from child: "@", total: "CHILD: Hi parent!@@"
PARENT: Complete string: "CHILD: Hi parent!"
```

```
$ cat pipeNfork.c
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <string.h>
void
{
```

Pointer Arithmetic Example

```
int array[3];
```

Element	Address Calculation	Address
array	1100	1100
array[0]	1100 + (sizeof(int) * 0)	1100
array[1]	1100 + (sizeof(int) * 1)	1104
array[2]	1100 + (sizeof(int) * 2)	1108

Therefore:

```
int* x = &array[0];                                          // x is int pointer
int* y = &array[2];                                          // y is int pointer
int z = y - x == 1108 - 1100 == 8 (bytes) => 2 (ints);       // z is int; z = 2
```

```
                printf("PARENT: Message received from child:                    \n", readBuffer, completeMessage);
                if (r == -1) { printf("r == -1\n"); break; }
                if (r == 0) { printf("r == 0\n"); break; }
            }
            int terminalLocation = strstr(completeMessage, "@@") - completeMessage; // Where is the terminal
            completeMessage[terminalLocation] = '\0'; // End the string early to wipe out the terminal
            printf("PARENT: Complete string: \"%s\"\n", completeMessage);
            break;
        }
}
```

(4068 - 4000) / sizeof(int) = 17

```
$ pipeNfork
PARENT: Message received from child: "CHILD: Hi", total: "CHILD: Hi"
PARENT: Message received from child: " parent!@", total: "CHILD: Hi parent!@"
PARENT: Message received from child: "@", total: "CHILD: Hi parent!@@"
PARENT: Complete string: "CHILD: Hi parent!"
```

$ cat pipeNfork.c

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <string.h>
void
{
```

Could also have just done this:

```
char* t = strstr(completeMessage, "@@"); // Where is the terminal
*t = '\0';
```

```
                    printf("PARENT: Message received from child:                    \n", readBuffer, completeMessage);
                    if (r == -1) { printf("r == -1\n"); break; }
                    if (r == 0) { printf("r == 0\n"); break; }
                }
                int terminalLocation = strstr(completeMessage, "@@") - completeMessage; // Where is the terminal
                completeMessage[terminalLocation] = '\0'; // End the string early to wipe out the terminal
                printf("PARENT: Complete string: \"%s\"\n", completeMessage);
                break;
        }
}
```

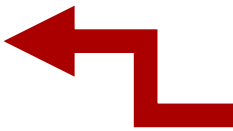
$ pipeNfork

```
PARENT: Message received from child: "CHILD: Hi", total: "CHILD: Hi"
PARENT: Message received from child: " parent!@", total: "CHILD: Hi parent!@"
PARENT: Message received from child: "@", total: "CHILD: Hi parent!@@"
PARENT: Complete string: "CHILD: Hi parent!"
```

Error Checking Reads and Writes

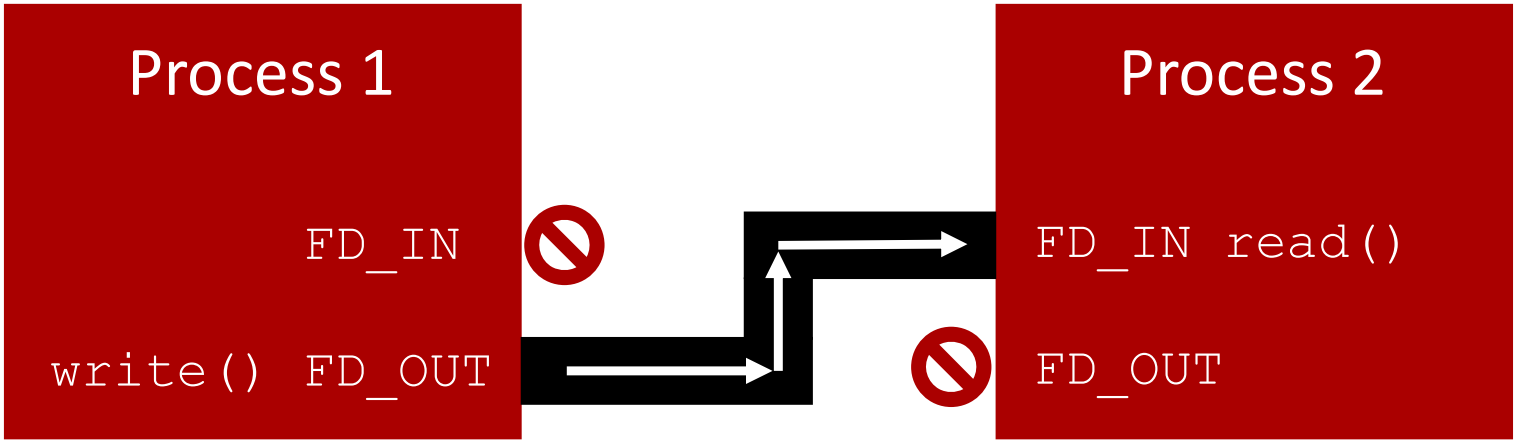
- Checking the return value of `read()` is very important
  - Not just if return value is -1 (an error)
  - The return value will tell you if the desired number of bytes was not read - this can tell you if the pipe didn't have the amount of data you expected it to
  - Our previous example used a terminator @@ instead of tracking byte counts because often you don't know how many bytes there will be

- Same goes for `write()`
  - If the number of bytes returned isn't what you expected (for example, if a signal handler interrupted), you'll need to loop over the write again, *writing only what got missed again* to it



Closing Pipes

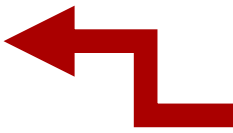
- Process 1 closes output pipe:
  - If process 2 is currently blocked on a `read()`, then process 2's `read()` will return 0
- Process 2 closes input pipe:
  - If process 1 tries to write to the pipe, `write()` will return -1, and errno (in process 1) will be set to EPIPE
  - Process 1 will be sent the SIGPIPE signal



Named Pipe = FIFO

- FIFO = First-In, First-out
- Essentially, a persistent pipe, which is represented by a special file

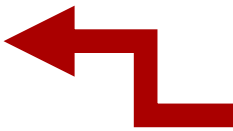
- Create in C with `mkfifo()`, or with `mkfifo` in bash
- Once created, any process can open a FIFO with `open()`
- Once opened, it works just like a pipe (or really: just like any file)



FIFO Use Cases

- You want to build a client-server architecture on a single machine, but you don't want to deal with the complexities of sockets
  - Can be used to transmit data between two non-related processes that didn't use `pipe()` and then `fork()`

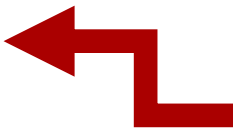
- You want to transmit data with a non-network aware program



FIFO shell example

```
$ mkfifo my_fifo
$ ls -l *my*
prw-rw----. 1 brewsteb upg57541 0 Oct 31 14:46 my_fifo
```

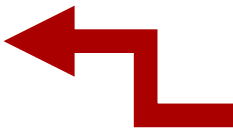
- Since they are files, you can apply most of the common bash shell commands like:
  - read, sort, wc, cut, awk, etc.
- As well as all of the common file input/output system calls in C: `open()`, `read()`, `write()`, etc.



Opening a FIFO

- When opening a FIFO, `open()` called by the first process will block; the first process will unblock once the second process calls open

- Example:
  1. Process A calls `open(…, O_RDONLY)` // Process A blocks
  2. Process B calls `open(…, O_WRONLY)` // Process A & B continue // execution



FIFO Example

```
$ cat pipeNforkFIFO.c
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <string.h>
#include <fcntl.h>
void main()
{
        int r, newfifo, fd;
        char completeMessage[512], readBuffer[10];
        char stringToWrite[20] = "CHILD: Hi parent!@@";
        pid_t spawnpid;
        char* FIFOfilename = "myNewFifo";
        newfifo = mkfifo(FIFOfilename, 0644); // Create the FIFO
        spawnpid = fork(); // Fork the child, which will write into the pipe
        switch (spawnpid) {
        case 0:  // Child
                fd = open(FIFOfilename, O_WRONLY); // Open the FIFO for writing
                if (fd == -1) { perror("CHILD: open()"); exit(1); }
                write(fd, stringToWrite, strlen(stringToWrite)); // Write the entire string into the pipe
                exit(0); break; // Terminate the child
        default:  // Parent
                fd = open(FIFOfilename, O_RDONLY); // Open the FIFO for reading
                if (fd == -1) { perror("PARENT: open()"); exit(1); }
                memset(completeMessage, '\0', sizeof(completeMessage)); // Clear the buffer
                while (strstr(completeMessage, "@@") == NULL) { // As long as we haven't found the terminal...
                        memset(readBuffer, '\0', sizeof(readBuffer)); // Clear the buffer
                        r = read(fd, readBuffer, sizeof(readBuffer) - 1); // Get the next chunk
                        strcat(completeMessage, readBuffer); // Add that chunk to what we have so far
                        printf("PARENT: Message received from child: \"%s\", total: \"%s\"\n", readBuffer, completeMessage);
                        if (r == -1) { printf("PARENT: r == -1, exiting\n"); break; } // Check for errors
                        if (r == 0) { printf("PARENT: r == 0, exiting\n"); break; }
                }
                int terminalLocation = strstr(completeMessage, "@@") - completeMessage; // Where is the terminal
                completeMessage[terminalLocation] = '\0'; // End the string early to wipe out the terminal
                printf("PARENT: Complete string: \"%s\"\n", completeMessage);
                remove(FIFOfilename); // Delete the FIFO
                break;
        }
}
```

```
$ pipeNforkFIFO
PARENT: Message received from child: "CHILD: Hi", total: "CHILD: Hi"
PARENT: Message received from child: " parent!@", total: "CHILD: Hi parent!@"
PARENT: Message received from child: "@", total: "CHILD: Hi parent!@@"
PARENT: Complete string: "CHILD: Hi parent!"
```