# Processes

**Benjamin Brewster**

Adapted from slides by Jon Herlocker, OSU

# The Process

- Process Management is a necessary component of an operating system

- Process
  - An instance of an executing program
  - A collection of execution resources associated with an executing program

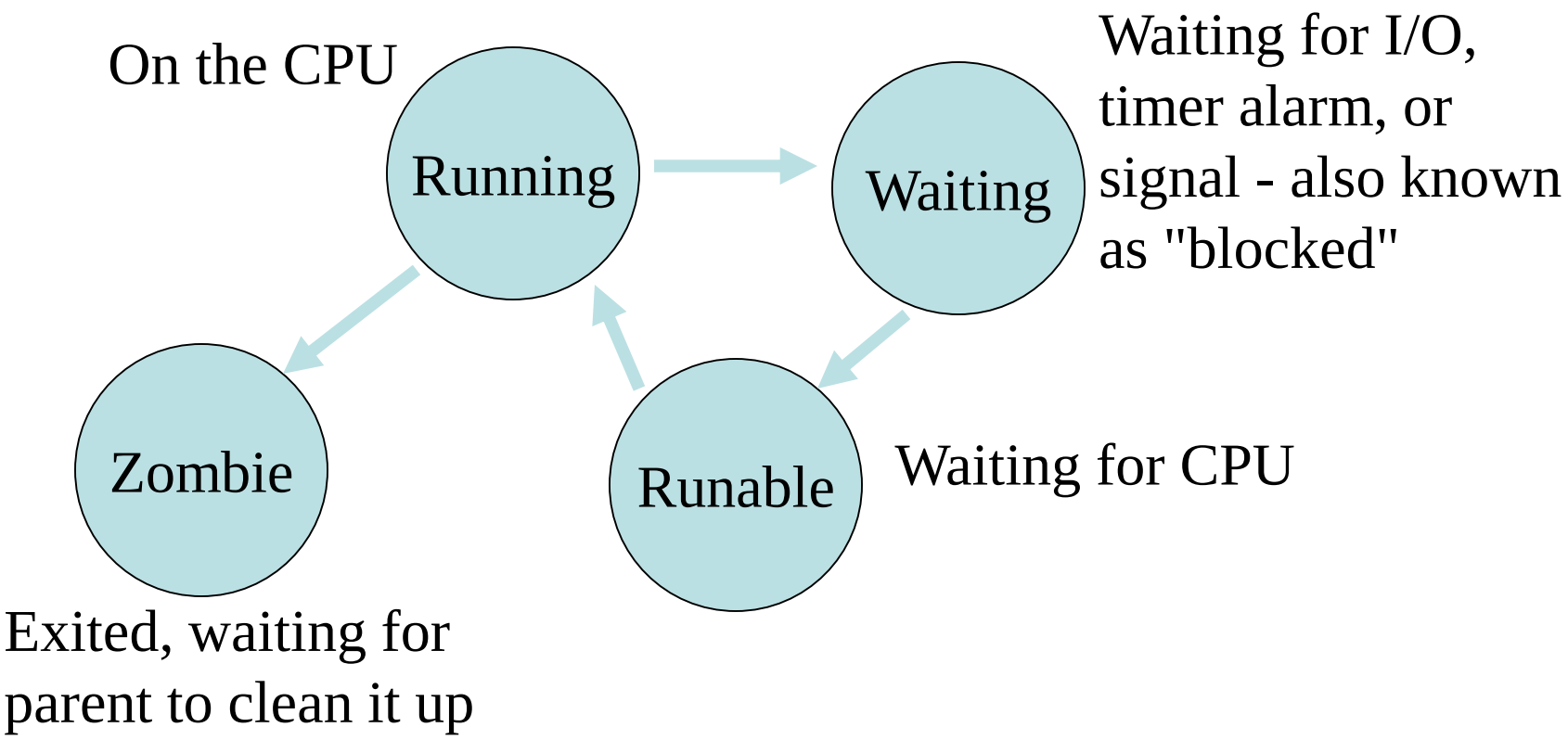
# Programs vs processes

- 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
  - They will have separate execution resources
    - Different virtual address spaces, process id, etc.
  - On a modern OS - some execution resources will be shared if two processes are executing the same program code

# A UNIX process consists of...

- A unique identify (process id aka pid)
- A virtual address space
- Program code and data (variables) in memory
- user/group identity, umask value
- An execution environment
  - 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

# Important Process States in UNIX



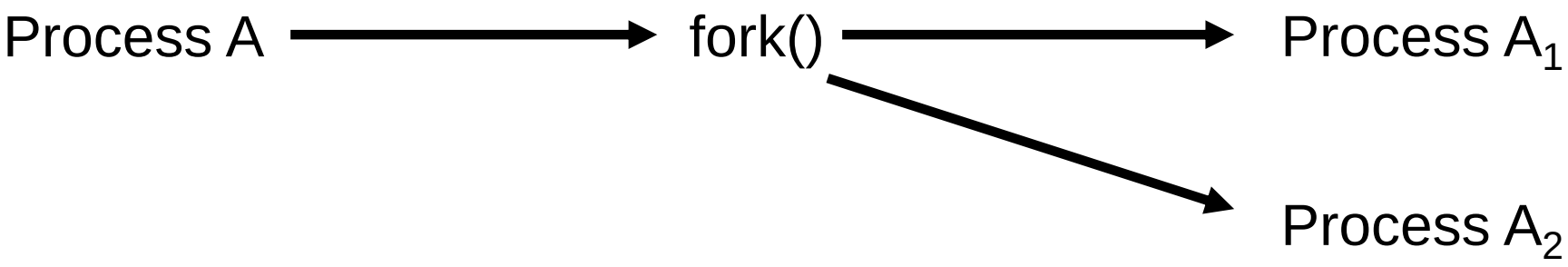
# How do you create a process

- Let the shell do it for you
  - When you run a program, the shell creates the process for you
- In some cases, you want to do it yourself
  - EX: Web Server (Prog5)
  - Unix provides a C API for creating and managing processes explicitly

# Managing processes

- Functions you will need to understand
  - fork()
  - The exec() family
    - execl, execlp, execv, execvp
  - exit()
  - wait(), waitpid()
  - getpid(), getpgrp(), setpgid(), setsid(), getsid(), getenv(), putenv(), nice()

# How to start a new process




Processes $A_1$ and $A_2$ are *almost* identical copies…

# Process $A_1$ == Process $A_2$ ??

- The two processes have different pids
- Each process returns a different value from `fork()`
- Process $A_2$ gets copies of all the open file descriptors of Process $A_1$
- Process $A_2$ has all of the same variables set to the same values as Process $A_1$, but they are now separately managed!
- More to come in a bit

# Return values of fork()

- In the child process, fork() returns 0
- In the parent process, fork() returns the process-id of the child process that was just created
- If something went wrong, fork() returns -1 to the parent process and sets the global variable errno
  - If -1 is returned to the parent process, then no child process was created

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

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!
```

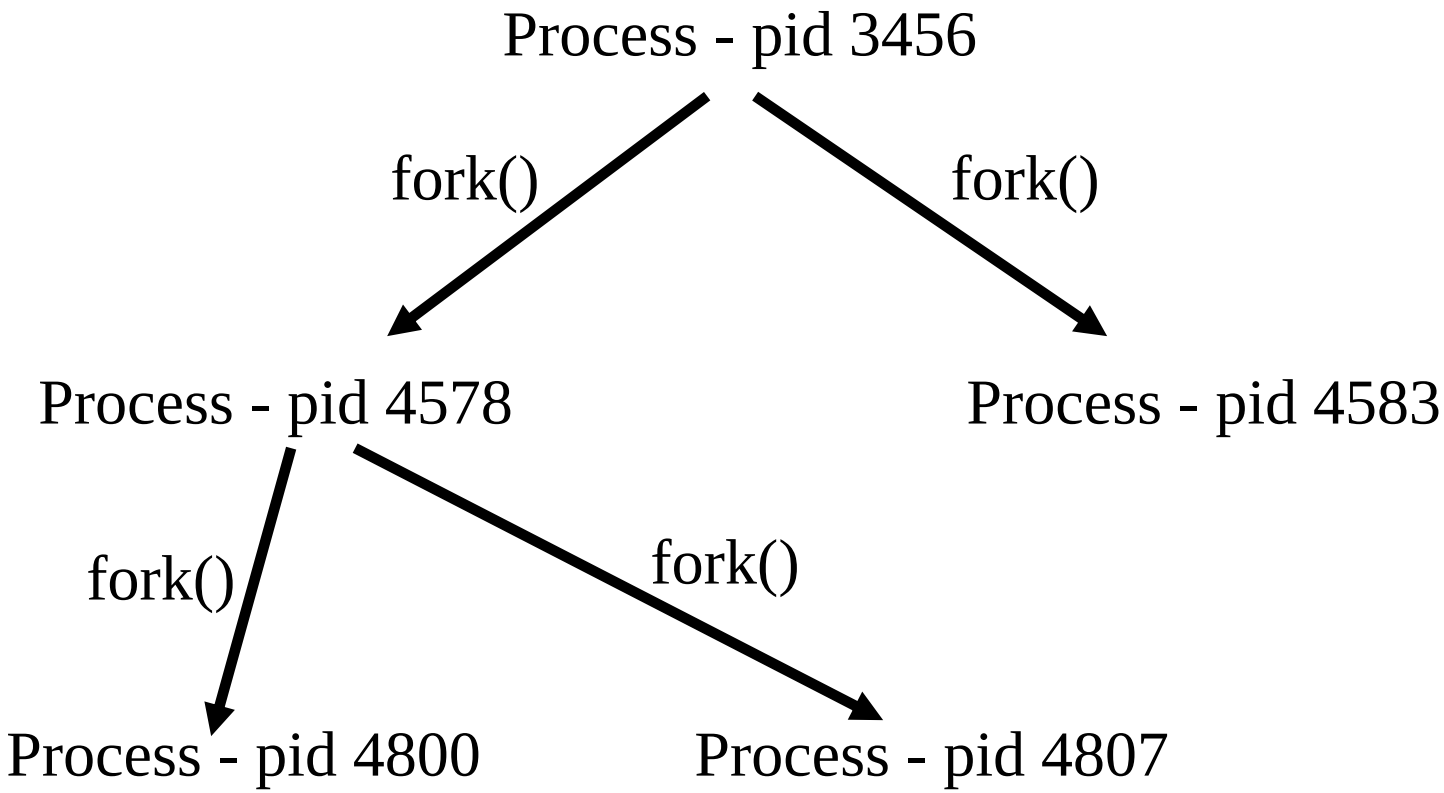
# In Total…

- Inherited by the child from the parent:
  - process credentials (real/effective/saved UIDs and GIDs)
  - environment
  - stack
  - memory
  - open file descriptors (note that the underlying file positions are shared between the parent and child, which can be confusing)
  - close-on-exec flags
  - signal handling settings
  - nice value
  - scheduler class
  - process group ID
  - session ID
  - current working directory
  - root directory
  - file mode creation mask (umask)
  - resource limits
  - controlling terminal

# In Total…

- Unique to the child:
  - process ID
  - different parent process ID
  - Own copy of file descriptors and directory streams.
  - process, text, data and other memory locks are NOT inherited.
  - process times, in the tms struct
  - resource utilizations are set to 0
  - pending signals initialized to the empty set
  - timers created by timer_create not inherited
  - asynchronous input or output operations not inherited

# Fork() forms a family tree



# Checking up on your kids

- 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 process was sent a signal, and the process did not catch the signal, so the OS killed it
- How do parents check up on their children?

# Checking the exit status

- Use the `wait()`, or `waitpid()`
- Both of these commands wait for a child process to end (normally, or abnormally)
- For both functions, you pass them a pointer to which the OS writes an integer which identifies how the child exited

# `wait` **VS** `waitpid`

- `wait()` will wait until any one child process completes. When one completes, wait returns the process-id
- `waitpid()` waits for the child process with the specified process ID to complete. If you pass it a special flag, it will also check if a process has exited yet, without waiting for it

```
int exitMethod;
pid_t spawnpid;

spawnpid = fork();  // create duplicate process

if (spawnpid == 0)
{
   printf(“:child: sleeping\n");
   sleep(5);
}
else if (spawnpid > 0)
{
   printf("parent: waiting\n");
   pid_t exitpid = wait(&exitMethod);
   printf("parent: child exited [%d]\n", exitMethod);
}
else
{
   printf("fork failed!\n");
   perror("fork()\n");
}
```

# Checking the exit status

```
int status;
cpid = wait(&status);

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

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

# waitpid

- This has exactly the same effect as wait()

```
cpid = waitpid(-1, &status, 0);
```

- This ignores all child processes except for the one identified by cpid

```
cpid = waitpid(cpid, &status, 0);
```

- This checks to see if any process has completed, but returns immediately with 0 if none have

```
cpid = waitpid(-1, &status, WNOHANG);
```

- This checks to see if the process identified by cpid has completed. If not, it returns immediately with 0

```
cpid = waitpid(cpid, &status, WNOHANG);
```

# 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 complete different program?
- For this we use the `exec...()` family

# exec

- Exec - short for "execute"
- exec replaces the currently running program with a new program that you specify
- The exec() function does not return - it destroys the currently running program
  - No line after a succesful exec call will run
- You can specify arguments to exec() - these become the command line arguments that show up in argc/argv in C, and $1, $2, etc in Bourne shell

# two types of exec

```
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-argn

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

- Executes the program specified by "path", and gives it the command line arguments specified by the array of pointers to strings

# 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[]);
```

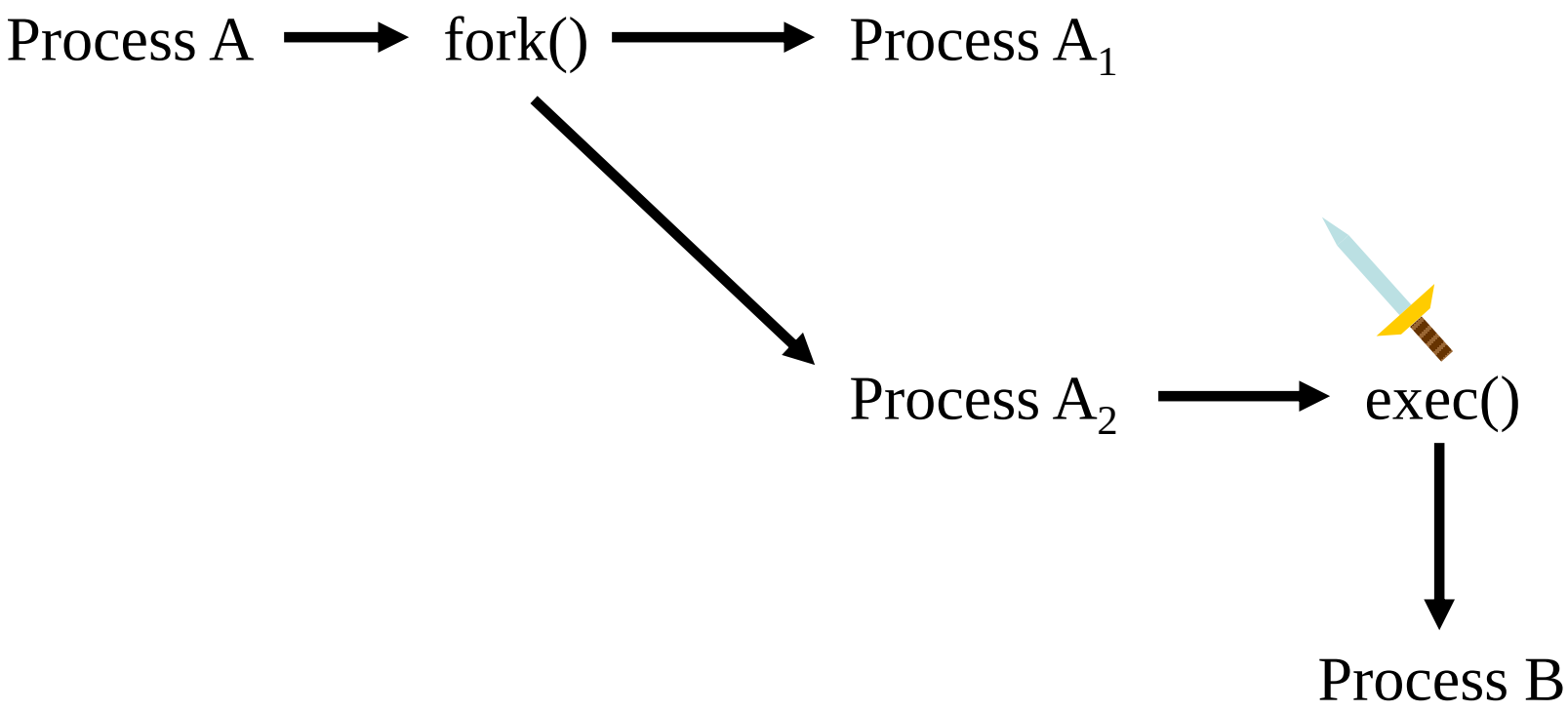
- The versions ending with 'p' will search your PATH environment variable for the executable if you specify a file in your current directory
  - Otherwise they are identical

# How to exec a new process?

- exec replaces the program it is called from
  - exec does not create a new process

- How do we create a new process running a different program, but keep our existing program?
  - Use fork() and exec() together

# Exec a new process



```
cpid = fork();  // create new process

if (cpid == 0)
{ // child
   printf("execing %s\n", argv[1]);
   execl(argv[1], argv[1], NULL);

   perror("exec");  // will never run unless error
   return(1);
}
else if (cpid == -1)
{
   printf("fork failed!");
   perror("fork()");
}

// parent process will continue executing here
```

# Passing parameters to execl

- `execl()`
- First parameter to `execl()` 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 name itself)
  - Some times the first argument will be the command name stripped of the path
- Last argument must always be NULL.
  - Indicates that there are no more parameters

# Passing parameters to execv

- `execv()`
- First parameter to `execv()` is the pathname of the new program
- Second parameters is an array of pointers to strings
- First string should be the same as the first parameter (the command name itself)
  - Some times the first argument will be the command name stripped of the path
- Last string must always be NULL.
  - Indicates that there are no more parameters

# `exec` Comparison

**`execvp()`**

```
i = 0;
myargv[i++] = strdup("ls");
myargv[i++] = strdup("-a");
myargv[i++] = strdup("-f");
myargv[i++] = strdup("-b");
myargv[i++] = NULL;
execvp(myargv[0], myargv);
```

**`execlp()`**

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

# 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()`

# Process Attributes

- Process ID
- Process group ID or process session ID
- Environment
- Current directory
- Real and effective user ids
- Real and effective group ids
- Process scheduling priority

# Process ID

- Each process has a unique process ID
  - pid
  - Fixed by the kernel - cannot be changed

- getpid() will return the process id

```
pid_t pid;
pid = getpid();
```

# Process environment

- 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 csh or tcsh by using the setenv/unsetenv commands

# setenv

```
% setenv
USER=brewstbe
HOME=/nfs/rack/u2/b/brewstbe
PATH=/usr/ccs/bin:/usr/ucb:/bin:/usr/local/bin:/usr/bin:/usr/local/bin/mh:/usr/bin/X
    11:/usr/local/X11R6/bin:/usr/local/apps/bin:/nfs/rack/u2/b/brewstbe/bin:.
MAIL=/var/spool/mail/brewstbe
SHELL=/bin/csh
SSH_CLIENT=::ffff:128.193.138.169 1072 22
SSH_CONNECTION=::ffff:128.193.138.169 1072 ::ffff:128.193.54.5 22
SSH_TTY=/dev/pts/1
TERM=vt100
HOSTTYPE=i386-linux
OSTYPE=linux
PWD=/nfs/rack/u2/b/brewstbe/public_html/CS311
HOST=flip.engr.oregonstate.edu
REMOTEHOST=128-193-138-169.public.oregonstate.edu
LANG=en_US.UTF-8
MANPATH=/usr/local/man:/usr/man:/usr/local/X11R6/man:/usr/dt/man:/opt/fortran/share/
    man:/opt/CC/share/man:/opt/ansic/share/man:/opt/fortran90/share/man
PAGER=less
EDITOR=pico
CVSROOT=/nfs/jungle/u4/transfer/CVS
```

# Accessing the env

- Csh, tcsh:

```
setenv MY_VAR "Some text string 1234"
echo $MY_VAR
```

- sh, bash

```
MY_VAR="Some text string 1234"
export MY_VAR
echo $MY_VAR
```

- C

```
putenv("MY_VAR= Some text string 1234");
printf("%s\n", getenv("MY_VAR"));
```

# Current working directory

- If you don't specify a fully qualified path name, then files specified are searched for in the current directory
- getcwd()
  - Gets the current working directory
  - In shell, use "pwd" (print working directory)
- chdir()
  - Sets the current working directory

# User and Group IDs

- This is how UNIX implements security
- The process user id and group id are used to determine what resources your process has rights to access or modify
- This affects
  - access to files and directories
  - signals (you can only kill processes you own)
  - access to privileged system calls (`nice`, `chroot`, etc)
  - resource limits, quotas, etc.
  - and more

# Which groups?

```
flip 134 CS311% id
uid=22026(brewsteb) gid=6009(upg22026)
  groups=6009(upg22026),12028(transfer)
```

- These groups are the same groups referred to when using the `chmod` command