Lecture 05: `fork` and Understanding `execvp`

Principles of Computer Systems

Winter 2020

Stanford University

Computer Science Department

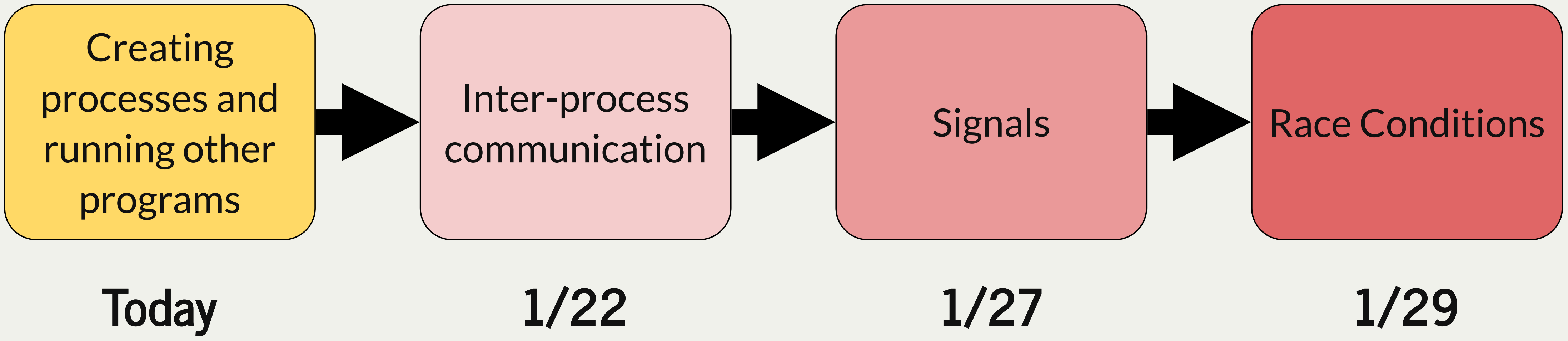
Instructors: Chris Gregg and
Nick Troccoli

Reading: Bryant & O'Hallaron,
Chapters 10 and 8

PDF of this presentation

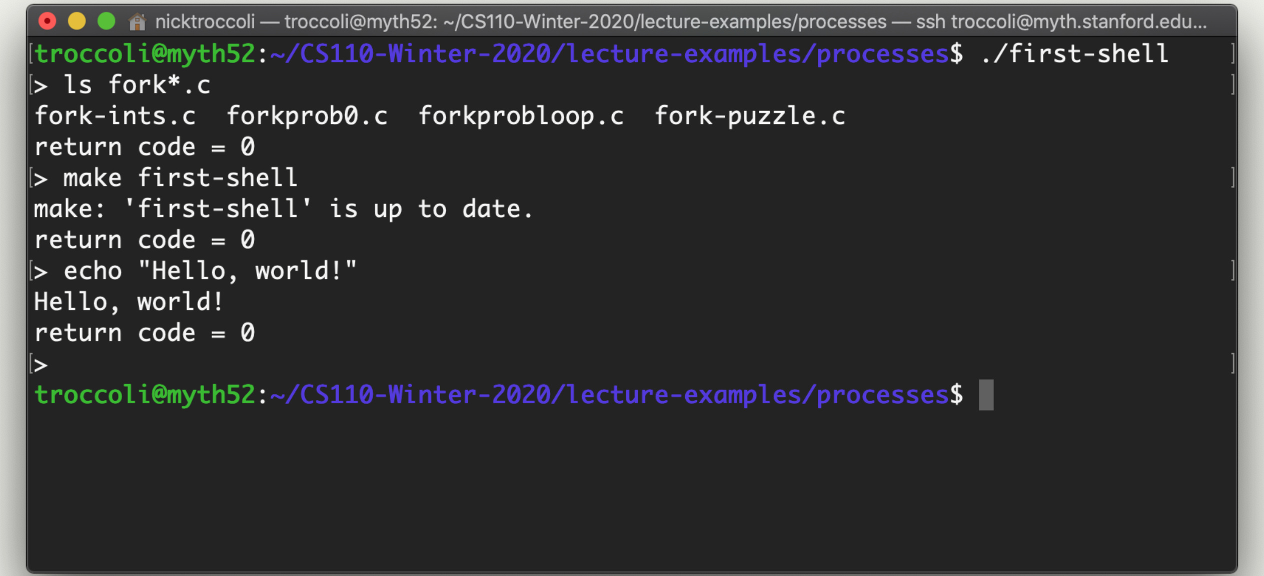
CS110 Topic 2: How can our programs create and interact with other programs?

Learning About Processes


Creating processes and running other programs
Inter-process communication
Signals
Race Conditions
Today
1/22
1/27
1/29

Today's Learning Goals

- Get more practice with using **fork()** to create new processes
- Understand how to use **waitpid()** to coordinate between processes
- Learn how **execvp()** lets us execute another program within a process
- **End Goal:** write our first implementation of a shell!



nicktroccoli — troccoli@myth52: ~/CS110-Winter-2020/lecture-examples/processes — ssh troccoli@myth.stanford.edu...

```
troccoli@myth52:~/CS110-Winter-2020/lecture-examples/processes$ ./first-shell
> ls fork*.c
fork-ints.c  forkprob0.c  forkprobloop.c  fork-puzzle.c
return code = 0
> make first-shell
make: 'first-shell' is up to date.
return code = 0
> echo "Hello, world!"
Hello, world!
return code = 0
>
troccoli@myth52:~/CS110-Winter-2020/lecture-examples/processes$
```

Plan For Today

- Reintroducing **fork()**
- **Practice:** Seeing....Quadruple?
- **waitpid()** and waiting for child processes
- **Break:** Announcements
- **Demo**: Waiting For Children
- **Putting it all together:** `first-shell`

`fork()`

- A system call that creates a new *child process*
- The "parent" is the process that creates the other "child" process
- From then on, both processes are running the code after the fork
- The child process is *identical* to the parent, except:
  - it has a new Process ID (PID)
  - for the parent, fork() returns the PID of the child; for the child, fork() returns 0
  - fork() is **called once**, but **returns twice**

```c
pid_t pidOrZero = fork();
// both parent and child run code here onwards
printf("This is printed by two processes.\n");
```

fork()

What happens to variables and addresses?

```c
// fork-copy.c
int main(int argc, char *argv[]) {
    char str[128];
    strcpy(str, "Hello");
    printf("str's address is %p\n", str);

    pid_t pid = fork();

    if (pid == 0) {
        // The child should modify str
        printf("I am the child. str's address is %p\n", str);
        strcpy(str, "Howdy");
        printf("I am the child and I changed str to %s. str's address is still %p\n", str, str);
    } else {
        // The parent should sleep and print out str
        printf("I am the parent. str's address is %p\n", str);
        printf("I am the parent, and I'm going to sleep for 2 seconds.\n");
        sleep(2);
        printf("I am the parent. I just woke up. str's address is %p, and its value is %s\n", str, str);
    }

    return 0;
}
```

fork()

```
$ ./fork-copy
str's address is 0x7ffc8cfa9990
I am the parent. str's address is 0x7ffc8cfa9990
I am the parent, and I'm going to sleep for 2 seconds.
I am the child. str's address is 0x7ffc8cfa9990
I am the child and I changed str to Howdy. str's address is still 0x7ffc8cfa9990
I am the parent. I just woke up. str's address is 0x7ffc8cfa9990, and its value is Hello
```

How can the parent and child use the same address to store different data?

- Each program thinks it is given all memory addresses to use
- The operating system maps these *virtual* addresses to *physical* addresses
- When a process forks, its virtual address space stays the same
- The operating system will map the child's virtual addresses to different physical addresses than for the parent

`fork()`

```
$ ./fork-copy
str's address is 0x7ffc8cfa9990
I am the parent. str's address is 0x7ffc8cfa9990
I am the parent, and I'm going to sleep for 2 seconds.
I am the child. str's address is 0x7ffc8cfa9990
I am the child and I changed str to Howdy. str's address is still 0x7ffc8cfa9990
I am the parent. I just woke up. str's address is 0x7ffc8cfa9990, and its value is Hello
```

Isn't it expensive to make copies of all memory when forking?

- The operating system only *lazily* makes copies.
- It will have them share physical addresses until one of them changes its memory contents to be different than the other.
- This is called *copy on write* (only make copies when they are written to).

Why is fork useful?

- Running a program in a shell
- Your operating system kernel spawns different processes for system services
- Network servers span processes when they receive connections
- and more...

Practice: fork()

```c
int main(int argc, char *argv[]) {
    // Initialize the random number with a "seed value"
    // this seed state is used to generate future random numbers
    srandom(time(NULL));

    printf("I'm unique and just get printed once.\n");
    pid_t pidOrZero = fork();
    exitIf(pidOrZero == -1, kForkFailure, stderr, "Call to fork failed... aborting.\n");

    // force exactly one of the two to sleep (why exactly one?)
    bool isParent = pidOrZero != 0;
    int result = random() % 2;
    if ((result == 0) == isParent) {
        sleep(1);
    }

    printf("I get printed twice (this one is being printed from the %s).\n", isParent  ? "parent" : "child");
    return 0;
}
```

Practice: fork()

```c
int main(int argc, char *argv[]) {
    // Initialize the random number with a "seed value"
    // this seed state is used to generate future random numbers
    srandom(time(NULL));

    printf("I'm unique and just get printed once.\n");
    pid_t pidOrZero = fork();
    exitIf(pidOrZero == -1, kForkFailure, stderr, "Call to fork failed... aborting.\n");

    // force exactly one of the two to sleep (why exactly one?)
    bool isParent = pidOrZero != 0;
    int result = random() % 2;
    if ((result == 0) == isParent) {
        sleep(1);
    }

    printf("I get printed twice (this one is being printed from the %s).\n", isParent  ? "parent" : "child");
    return 0;
}
```

Key Idea: all state is copied from the parent to the child, even the random number generator seed! *Both the parent and child will get the same return value from random().*

Practice: fork()

```c
int main(int argc, char *argv[]) {
    printf("Starting the program\n");
    pid_t pidOrZero1 = fork();
    pid_t pidOrZero2 = fork();

    if (pidOrZero1 != 0 && pidOrZero2 != 0) {
        printf("Hello\n");
    }

    if (pidOrZero2 != 0) {
        printf("Hi there\n");
    }

    return 0;
}
```

How many processes run in total?

a) 1 b) 2 c) 3 d) 4

How many times is "Hello" printed?

a) 1 b) 2 c) 3 d) 4

How many times is "Hi there" printed?

a) 1 b) 2 c) 3 d) 4

Plan For Today

- Reintroducing **fork()**
- **Practice:** Seeing....Quadruple?
- **waitpid()** and waiting for child processes
- **Break:** Announcements
- **Demo**: Waiting For Children
- **Putting it all together:** `first-shell`

waitpid()

A function that a parent can call to wait for its child to exit:

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

- **pid:** the PID of the child to wait on (we'll see other options later)
- **status:** where to put info about the child's termination (or NULL)
- **options:** optional flags to customize behavior (always 0 for now)
- the function returns when the specified **child process** exits
- the return value is the PID of the child that exited, or -1 on error (e.g. no child to wait on)
- If the child process has already exited, this returns immediately - otherwise, it blocks

waitpid()

```c
// waitpid.c
int main(int argc, char *argv[]) {
  printf("Before.\n");
  pid_t pidOrZero = fork();
  printf("After.\n");
  if (pidOrZero == 0) {
    printf("I'm the child, and the parent will wait up for me.\n");
    return 110;
  } else {
    int status;
    int result = waitpid(pidOrZero, &status, 0);

    if (WIFEXITED(status)) {
      printf("Child exited with status %d.\n", WEXITSTATUS(status));
    } else {
      printf("Child terminated abnormally.\n");
    }
    return 0;
  }
}
```

We can use **WIFEXITED** and **WEXITSTATUS** (among others) to extract info from the status.

(full program, with error checking, is right here)

waitpid()

```
$ ./waitpid
Before.
After.
After.
I am the child, and the parent will wait up for me.
Child exited with status 110.
$
```

The output will be the same every time this program runs

- The parent will always wait for the child to finish before continuing

`waitpid()`

A parent process should always wait on its children processes.

- A process that finished but was not waited on by its parent is called a *zombie*.
- Zombies take up system resources (until they are ultimately cleaned up later by the OS)
- Calling waitpid in the parent "reaps" the child process (cleans it up)
  - If a child is still running, waitpid in the parent will block until it finishes, and then clean it up
  - If a child process is a zombie, waitpid will return immediately and clean it up
- Orphaned child processes get "adopted" by the **init** process (PID 1)

Make sure to reap your zombie children.

(Wait what?)

Plan For Today

- Reintroducing **fork()**
- **Practice:** Seeing....Quadruple?
- **waitpid()** and waiting for child processes
- **Break:** Announcements
- **Demo**: Waiting For Children
- **Putting it all together:** `first-shell`

Announcements

- assign1 due tonight at 11:59PM PST
- Sections start *tomorrow* (handout will be posted on website this afternoon)
- Reminder: please send us your OAE letters if you haven't already
- assign2 (filesystems) goes out later today, due *next Thurs. at 11:59PM PST*

Assignment 2: The Unix v6 Filesystem

- Your task is to write a C program that can read from a 1970s-era Unix version 6 filesystem.
  - The test data are bit-for-bit representations of a Unix v6 disk.
  - You will leverage all of the information covered in the file system lectures, and for more detailed information, see Section 2.5 of the Salzer and Kaashoek textbook.
- You will primarily be writing code in four different files (and we suggest you tackle them in this order):
  - **`inode.c`**
  - **`file.c`**
  - **`directory.c`**
  - **`pathname.c`**
- Because the program is in C, you will have to rely on arrays of structs, and low-level data manipulation, as you don't have access to any C++ standard template library classes.

Assignment 2: The Unix v6 Filesystem

- You will be implementing 4 layers in the filesystem, each on top of one another. Ultimately your program will be able to locate and read files in the filesystem.
- The lowest level (reading a sector from disk) is provided for you:

```
/**
 * Reads the specified sector (e.g. block) from the disk.  Returns the number of bytes read,
 * or -1 on error.
 */
int diskimg_readsector(int fd, int sectorNum, void *buf);
```

- **Key Idea:** sometimes, **`buf`** will be an array of **`inode`**s, sometimes it will be a buffer that holds actual file data. The function will *always* read **`DISKIMG_SECTOR_SIZE`** bytes, and you must determine the relevance of those bytes.
- **Carefully read through the header files for this assignment.** There are key constants (e.g., **`ROOT_INUMBER`**, **`struct direntv6`**, etc.) that are defined for you to use, and function headers, and reading them will orient you for the assignment.

Assignment 2: The Unix v6 Filesystem

One function that can be tricky to write is the following:

```
/**
 * Gets the location of the specified file block of the specified inode.
 * Returns the disk block number on success, -1 on error.
 */
int inode_indexlookup(struct unixfilesystem *fs, struct inode *inp, int blockNum);
```

- The **`unixfilesystem`** struct is defined and initialized for you.
- The **`inode`** struct will be populated already
- The **`blockNum`** is the number, in linear order, of the *data* block you are looking for in the file.
- Let's say the **`inode`** indicates that the file it refers to has a size of 180,000 bytes. And let's assume that **`blockNum`** is **`302`**.
    - This means that we are looking for the 302nd block of data in the file referred to by **`inode`**.
    - Recall that blocks are 512 bytes long.
    - How would you find the block index (i.e., sector index) of the 302nd block in the file?

Assignment 2: The Unix v6 Filesystem

- For example:
  - Let's say the **`inode`** indicates that the file it refers to has a size of 180,000 bytes. And let's assume that **`blockNum`** is **`302`**.
  - This means that we are looking for the 302nd block of data in the file referred to by **`inode`**.
  - Recall that blocks are 512 bytes long
  - How would you find the block index (i.e., sector index) of the 302nd block in the file?

a. Determine if the file is large or not

b. If it isn't large, you know you only have direct addressing.

c. If it is large (this file is), then you have indirect addressing.

d. The 302nd block is going to fall into the *second* indirect block, because each block has 256 block numbers (each block number is an **`unsigned short`**, or a **`uint16_t`**).

e. You, therefore, need to use **`diskimg_readsector`** to read the sector listed in the 2nd block number (which is in the **`inode`** struct), then extract the (302 % 256)th short from that block, and return the value you find there.

f. If the block number you were looking for happened to fall into the 8th inode block, then you would have a further level of indirection for a doubly-indirect lookup.

Assignment 2: The Unix v6 Filesystem

For the assignment, you will also have to search through directories to locate a particular file.

- You *do not* have to follow symbolic links (you can ignore them completely)
- You do need to consider directories that are longer than 32 files long (because they will take up more than two blocks on the disk), *but* this is not a special case! You are building generic functions to read files, so you can rely on them to do the work for you, even for directory file reading.
- Don't forget that a filename is limited to 14 characters, and if it is exactly 14 characters, there is *not* a trailing `'\0'` at the end of the name (this to conserve that one byte of data!) So...you might want to be careful about using **`strcmp`** for files (maybe use **`strncmp`**, instead?)
- This is a relatively advanced assignment, with a lot of moving parts.
- Start early!
- Come to office hours or ask Piazza questions.
- Remember: CAs *won't* look at your code, so you must formulate your questions to be conceptual enough that they can be answered.

Break / Mid-Lecture Checkin

We now know the answers to the following questions:

- When fork() returns, what does it return to the parent? To the child?
- How can 2 processes report that the same address contains different values?
- What function do we use to have a parent wait for its child?

Plan For Today

- Reintroducing **fork()**
- **Practice:** Seeing....Quadruple?
- **waitpid()** and waiting for child processes
- **Break:** Announcements
- **Demo: Waiting For Children**
- **Putting it all together:** `first-shell`

Waiting On Multiple Children

A parent can call **`fork`** multiple times, but must reap all the child processes.

- A parent can use **waitpid** to wait on *any of its children* by passing in **-1** as the PID.
- **Key Idea:** The children may terminate in *any* order!
- If **waitpid** returns -1 and sets **errno** to **ECHILD**, this means there are no more children.

Demo: Let's see how we might use this (**reap-as-they-exit.c**)

Waiting On Multiple Children

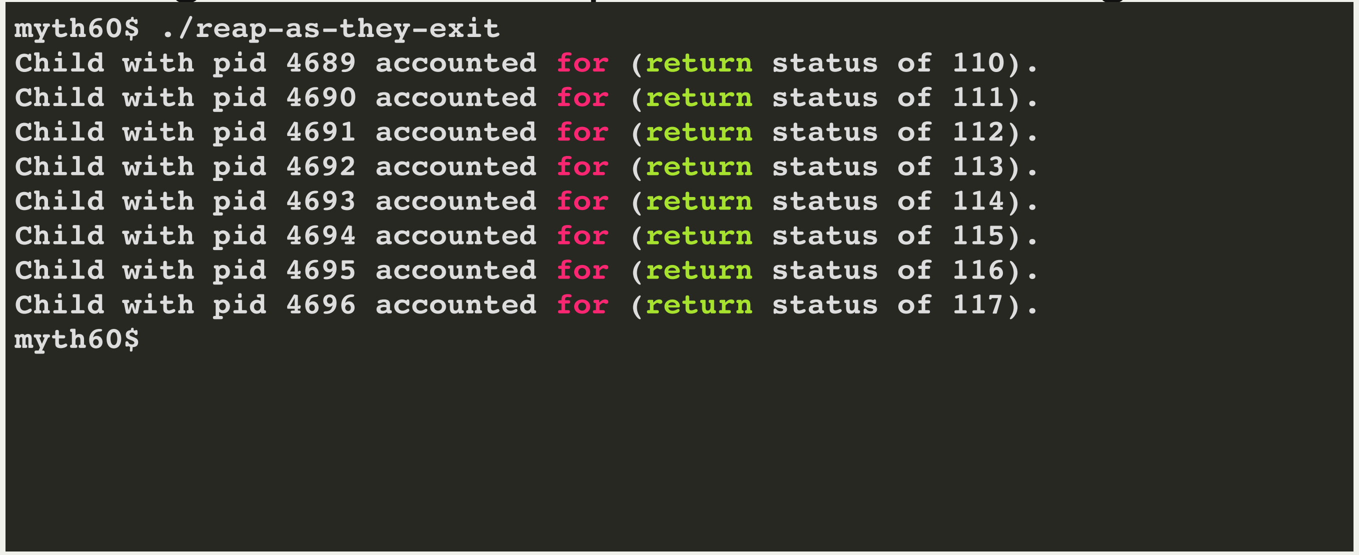
What if we want to wait for children in the order in which they were created?

Check out the abbreviated program below (full program with error checking right here):

```c
// reap-in-fork-order.c
int main(int argc, char *argv[]) {
  pid_t children[8];
  for (size_t i = 0; i < 8; i++) {
    if ((children[i] = fork()) == 0) exit(110 + i);
  }
  for (size_t i = 0; i < 8; i++) {
    int status;
    pid_t pid = waitpid(children[i], &status, 0);
    assert(pid == children[i]);
    assert(WIFEXITED(status) && (WEXITSTATUS(status) == (110 + i)));
    printf("Child with pid %d accounted for (return status of %d).\n",
           children[i], WEXITSTATUS(status));
  }
  return 0;
}
```

Waiting On Multiple Children

- This version spawns and reaps processes in some first-spawned-first-reaped manner.
- The child processes aren't required to exit in FSFR order.
- In theory, the first child thread could finish last, and the reap loop could be held up on its very first iteration until the first child really is done. But the process zombies are reaped in the order they were forked.
- Below is a sample run of the **`reap-in-fork-order`** executable. The pids change between runs, but even those are guaranteed to be published in increasing order.



```
myth60$ ./reap-as-they-exit
Child with pid 4689 accounted for (return status of 110).
Child with pid 4690 accounted for (return status of 111).
Child with pid 4691 accounted for (return status of 112).
Child with pid 4692 accounted for (return status of 113).
Child with pid 4693 accounted for (return status of 114).
Child with pid 4694 accounted for (return status of 115).
Child with pid 4695 accounted for (return status of 116).
Child with pid 4696 accounted for (return status of 117).
myth60$
```

execvp()

The most common use for **fork** is not to spawn multiple processes to split up work, but instead to run a *completely separate program* under your control and communicate with it.

- This is what a **shell** is; it is a program that prompts you for commands, and it executes those commands in separate processes. Let's take a look.

execvp()

`execvp` is a function that lets us run *another program* in the current process.

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

It runs the specified program executable, *completely cannibalizing the current process.*

- **`path`** identifies the name of the executable to be invoked.
- **`argv`** is the argument vector that should be passed to the new executable's **`main`** function.
- For the purposes of CS110, **`path`** and **`argv[0]`** end up being the same exact string.
- If **`execvp`** fails to cannibalize the process and install a new executable image within it, it returns -1 to express failure.
- If **`execvp`** succeeds, it **never returns** in the calling process.
  - **`execvp`** has many variants (**`execle`**, **`execlp`**, and so forth. Type **`man execvp`** to see all of them). We generally rely on **`execvp`** in this course.

Plan For Today

- Reintroducing **fork()**
- **Practice:** Seeing....Quadruple?
- **waitpid()** and waiting for child processes
- **Break:** Announcements
- **Demo**: Waiting For Children
- **Putting it all together:** `first-shell`

execvp()

We are going to use **execvp** to implement our own shell program!

Demo: first-shell.c

mysystem()

- Here's the implementation, with minimal error checking (the full version is right here):

```c
static int mysystem(const char *command) {
  pid_t pid = fork();
  if (pid == 0) {
    char *arguments[] = {"/bin/sh", "-c", (char *) command, NULL};
    execvp(arguments[0], arguments);
    printf("Failed to invoke /bin/sh to execute the supplied command.");
    exit(0);
  }
  int status;
  waitpid(pid, &status, 0);
  return WIFEXITED(status) ? WEXITSTATUS(status) : -WTERMSIG(status);
}
```

- Instead of calling a subroutine to perform some task and waiting for it to complete, `mysystem` spawns a **child process** to perform some task and waits for it to complete.
- We don't bother checking the return value of `execvp`, because we know that if it returns at all, it returns a -1. If that happens, we need to handle the error and make sure the child process terminates, via an exposed `exit(0)` call.
- Why not call `execvp` inside parent and forgo the child process altogether? Because `execvp` would consume the calling process, and that's not what we want.

Lecture Recap

- Reintroducing **fork()**
- **Practice:** Seeing....Quadruple?
- **waitpid()** and waiting for child processes
- **Break:** Announcements
- **Demo**: Waiting For Children
- **Putting it all together:** `first-shell`

Next time: inter-process communication