Announcements

- **Still riding this ECF train!**
  - I need to finish the second version of our shell—this one called **`simplesh`**—to illustrate how a real shell parses the command line, leverages **`fork`** and **`execvp`** to spawn off new executables, and exactly what it means for a process to run in the background as opposed to the foreground.
  - I also need to work through the client program using and the implementation of the **`subprocess`**, which demonstrates how different processes can communicates via **pipes**. #lovethisexample
  - I also need to work through the examples from this past Monday's slide deck. We haven't gotten to that slide deck yet. #sadface
  - And I need to sneak in an example where **processes** signal other processes. So much to do!
    - Most of our examples just rely on the kernel firing signals at processes that need to be informed about some (possibly bad, possibly not bad) event.
    - However, there are certain times when a process (e.g. your Assignment 3 **`tsh`** shell, for instance, which you'll get next Friday) needs to signal a child process (e.g. one of the shell's jobs) that something happened. In particular, your **`tsh`** shell will be the first to hear about **`SIGINT`** and **`SIGTSTP`** signals when the user types ctrl-c and ctrl-z. Your **`tsh`** implementation will want to handle each of those siganls by effectively forwarding the same signal to the foreground process group.
  - Come later next week, I'll present a straightforward, high-level view of how two (or 200) processes can seemingly run in the same address space by talking about virtual-to-physical memory mapping (our first real foray into virtualization).

Interprocess communications

- **Processes can message other processes using signals via the `kill` system call.**
  - Prototype:

```
int kill(pid_t pid, int signum);
```

  - Analogous to the **`/bin/kill`** shell command.
    - Unfortunately named, since kill implies **`SIGKILL`**, which implies death.
    - So named because the default action of most signals in early UNIX implementations was to just terminate the target process (identified by **`pid`**).
  - Returns -1 if the call fails (generally because the calling process doesn't have permission to fire signals at target process), 0 if all is swell.
  - **`pid`** parameter is overloaded to provide many signalling strategies:
    - When **`pid`** is a positive number, the target is a process with that **`pid`**.
    - When **`pid`** is a negative number less than -1, the targets are all processes within the process group **`abs(pid)`**.
    - **`pid`** can also be 0 or -1, but we don't need to worry about those (though it's documented in the B&O textbook if you're curious).
  - Your Assignment 3 **`tsh`** shell will use **`kill`** within your **`SIGINT`** and **`SIGTSTP`** handlers to forward those same signals to the foreground process group.

One final example

- **One final puzzle to ensure you follow the ideas:**
  - Meaningful example illustrating **`kill`** function would be too large.
  - Instead, we'll work through a small puzzle to confirm you all understand the workflow of the processes and understand how **`kill`** triggers various signal handlers to be executed. (Error checking is omitted from this example, since it's such a small example, and we'll assume, for simplicity, that nothing ever fails).
  - Putting it all together. What're the possible outputs (plural!) of the following program?

```
static pid_t pid;
static int counter = 0;

static void parentHandler(int unused) {       static void childHandler(int unused) {
  counter++;                                    counter += 3;
  printf("counter = %d\n", counter);            printf("counter = %d\n", counter);
  kill(pid, SIGUSR1);                         }
}

int main(int argc, char *argv[]) {
  signal(SIGUSR1, parentHandler);
  if ((pid = fork()) == 0) {
    signal(SIGUSR1, childHandler);
    kill(getppid(), SIGUSR1);
    return 0;
  }

  waitpid(pid, NULL, 0);
  counter += 7;
  printf("counter = %d\n", counter);
  return 0;
}
```

One final example (continued)

- **What's the expected output?**
  - Make sure you understand why the program on the previous slide is capable of producing two different outputs, depending on processor scheduling. (Hint: the child process may or may not exit before **`parentHandler`** has a chance to kill it!)

```
myth22> ./kill-puzzle
counter = 1
counter = 8
myth22> ./kill-puzzle
counter = 1
counter = 3
counter = 8
myth22>
```