Announcements

- **Assignment 1 due Monday, Assignment 2 out on Monday as well.**
  - Due just before midnight. If you're new to mercurial and/or the CS110 submissions process (which you are unless you took CS107 and followed the same script), go through the submission process now to confirm everything works sans drama.
  - Assignment 2 will be an opportunity to revisit your Assignment 1 implementation and optimize it to run as quickly as possible.
- **Topics for today:**
  - Continue playing with **`fork`**, introduce **`waitpid`**, and maybe even get to the awesomeness that is **`execvp`**.
  - Lots of examples from previous slide decks to work through, and I have a few more here as well.
  - You can download Truman's lecture slides from today right here

Enter the `execvp` command

- **`execvp` effectively reboots a process to run a different program from scratch.**
  - **`execvp`** has many variants (**`execle`**, **`execlp`**, and so forth. Type **`man execvp`** to see all of them).
  - Here is the prototype:

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

  - Here's what the arguments and the return type mean:
    - **`path`** identifies the name of the executable that should be invoked.
    - **`argv`** is the argument vector that should be funneled through to the new executable's **`main`** function.
    - Generally, at least for the purposes of CS110, **`path`** and **`argv[0]`** end up being the same exact string.
    - If **`execvp`** fails to consume the process and install a new process image within it, **`execvp`** will return -1.

Using execvp

- **Core implementation of `mysystem` (to emulate the `system` builtin)**
  - Here we present our own implementation of the **`mysystem`** function, which executes the provided **`command`** (guaranteed to be a **`'\0'`**-terminated C string) by calling **`"/bin/sh -c command"`** and ultimately returning once the surrogate **`command`** has finished.
  - If the execution of **`command`** exits normally (either via an **`exit`** system call, or via a normal return statement from **`main`**), then our **`mysystem`** implementation should return that exact same exit status.
  - If the execution exits abnormally (e.g. it segfaults), then we'll assume it aborted because some signal was ignored, and we'll return that signal number (e.g. 11 for **`SIGSEGV`**).
  - Here's the implementation (online right here)

```
static int mysystem(const char *command) {
  pid_t pid = fork();
  if (pid == 0) {
    char *arguments[] = {"/bin/sh", "-c", (char *) command, NULL};
    execvp("/bin/sh", arguments);
    exitIf(true, kExecFailed, stderr, "execvp failed to invoke this: %s.\n", command);
  }

  int status;
  waitpid(pid, &status, 0);
  if (WIFEXITED(status))
    return WEXITSTATUS(status);
  else
    return WTERMSIG(status);
}
```

  - Here's a trivial unit test that I'll run in lecture to prove this thing really works:

```
static const size_t kMaxLine = 2048;
int main(int argc, char *argv[]) {
  char buf[kMaxLine];
  while (true) {
    printf("> ");
    fgets(buf, kMaxLine, stdin);
    if (feof(stdin)) break;
    buf[strlen(buf) - 1] = '\0'; // overwrite '\n'
    printf("retcode = %d\n", mysystem(buf));
  }

  printf("\n");
  return 0;
}
```

Core implementation of `simplesh`, version 1.0

- **This is the best example of `execvp` imaginable: a minimum version of a shell not unlike those you've been using since the day you learned UNIX.**
  - Relies on **`fork`**, **`waitpid`**, and **`execvp`**.
  - This first version operates as a read-eval-print loop (keyword: repl), responding to many things we type in by forking off child processes.
    - Each child process is initially a deep clone of the shell.
    - Each child proceeds to replace its own process image with the new one we specify (e.g. **`ls`**, **`cp`**, our own CS110 **`search`** (which we wrote the second day of class), or even **`emacs`**.
    - A trailing ampersand (&) (e.g. **`emacs &`**) is an instruction to execute in the background, without blocking.
  - Implementation of **`simplesh`** is presented over the next three slides. Where helper functions don't rely on CS110 concepts, I omit their implementations (but will describe them in lecture).

Core of `simplesh` implementation

- **Implementation of `main`:**
  - Here's the first half (full implementation right here)

```
int main(int argc, char *argv[]) {
  while (true) {
    char command[kMaxCommandLength + 1];
    readCommand(command, sizeof(command) - 1);
    if (feof(stdin)) break;
    char *arguments[kMaxArgumentCount + 1];
    int count = parseCommandLine(command, arguments, sizeof(arguments)/sizeof(arguments[0]));
    if (count == 0) continue;
    bool builtin = handleBuiltin(arguments);
    if (builtin) continue; // it's been handled, move on
    bool isBackgroundProcess = strcmp(arguments[count - 1], "&") == 0;
    if (isBackgroundProcess) arguments[--count] = NULL; // overwrite "&"
    pid_t pid = forkProcess();
```

Core of `simplesh` implementation (continued)

- **Implementation of `main`:**
  - Here's the second half

```
    if (pid == 0) {
      if (execvp(arguments[0], arguments) < 0) {
        printf("%s: Command not found\n", arguments[0]);
        exit(0);
      }
    }

    if (!isBackgroundProcess) {
      waitForChildProcess(pid);
    } else {
      printf("%d %s\n", pid, command);
    }
  }

  printf("\n");
  return 0;
}
```

Core of `simplesh` implementation (continued)

- **Helper routines**
  - Here are a few helper routines that do rely on CS110 material:

```
static bool handleBuiltin(char *arguments[]) {
  if (strcasecmp(arguments[0], "quit") == 0) exit(0);
  return strcmp(arguments[0], "&") == 0;
}

static pid_t forkProcess() {
  pid_t pid = fork();
  exitIf(pid == -1, kForkFailed, stderr, "fork function failed.\n");
  return pid;
}

static void waitForChildProcess(pid_t pid) {
  exitUnless(waitpid(pid, NULL, 0) == pid, kWaitFailed,
             stderr, "Error waiting in foreground for process %d to exit", pid);
}
```

Implementing `subprocess`

- **Introducing `pipe`:**
  - The **`pipe`** system call takes an uninitialized array of two integers (let's call this array **`fds`**) and populates it with two file descriptors such that everything *written* to **`fds[1]`** can be *read* from **`fds[0]`**.

```
int pipe(int fds[]); // fds array should be of length 2, return -1 on error, 0 otherwise
```

  - **`pipe`** is particularly useful for allowing parent processes to communicate with forked child processes. (Recall that **`fork`** clones the caller's virtual address space **and** duplicates all open file descriptors as well).
  - Using **`pipe`**, **`fork`**, **`dup2`**, **`execvp`**, **`close`**, and **`waitpid`**, we can implement the **`subprocess`** function, which relies on the following record definition and is implemented to the following prototype:

```
typedef struct {
  pid_t pid;
  int infd;
} subprocess_t;
subprocess_t subprocess(const char *command);
```

  - The subprocess created by **`subprocess`** executes the provided command (guaranteed to be a **`'\0'`**-terminated C string) by calling **`"/bin/sh -c command"`**. Rather than waiting for **`command`** to finish, the implementation returns a **`subprocess_t`** with the **`command`** process's pid and a single file descriptor. In particular, arbitrary data can be published to the return value's **`infd`** with the understanding that it'll be read by **`command`**'s standard input.

Implementing **`subprocess`** (continued)

- **Sample client application, with output**
  - The following client program and test run illustrate precisely how **`subprocess`** should work:

```
int main(int argc, char *argv[]) {
  subprocess_t sp = subprocess("/usr/bin/sort");
  const char *words[] = {
    "felicity", "umbrage", "susurration", "halcyon",
    "pulchritude", "ablution", "somnolent", "indefatigable"
  };
  for (size_t i = 0; i < sizeof(words)/sizeof(words[0]); i++) {
    dprintf(sp.infd, "%s\n", words[i]);
  }
  close(sp.infd); // effectively sends cntl-D to child process
  int status;
  pid_t pid = waitpid(sp.pid, &status, 0);
  return pid == sp.pid && WIFEXITED(status) ? WEXITSTATUS(status) : -1;
}
```

  - The output of the above program, given a properly implemented **`subprocess`** routine, should look like so:

```
myth22> ./subprocess-test
ablution
felicity
halcyon
indefatigable
pulchritude
somnolent
susurration
umbrage
```

Implementing `subprocess` (continued)

- **Implementation is right here (dense):**
  - Here is the implementation of **`subprocess`** (where I omit error checking so that the meat of the implementation is clear):

```
subprocess_t subprocess(const char *command) {
  int fds[2];
  pipe(fds);
  subprocess_t process = { fork(), fds[1] };
  if (process.pid == 0) {
    dup2(fds[0], STDIN_FILENO);
    close(fds[0]);
    close(fds[1]);
    char *argv[] = {"/bin/sh", "-c", (char *) command, NULL};
    execvp(argv[0], argv);
  }
  close(fds[0]);
  return process;
}
```

  - Note that the write end of the pipe is embedded into the **`subprocess_t`**. That way, the parent knows where to publish text so that it flows to the other end of the pipe, across the parent process/child process boundary.
  - Further note that the child process reassociates the read end of the pipe with its own standard input (using **`dup2`**).
  - Once the **`dup2`** reassociation has been made, the child process can close both ends of its copy of the pipe.
  - The parent doesn't need the read end of the pipe, so it too can close its own copy of **`fds[0]`**.