Lecture 07: Process Control, Interprocess Communication

- The `mysystem` function is the first example I've provided where `fork`, `execvp`, and `waitpid` tap multiprocessing to do something useful.
  - The test harness we used to exercise `mysystem` is operationally a miniature terminal.
  - We need to continue implementing a few more miniature terminals (aka shells) to fully demonstrate how `fork`, `waitpid`, and `execvp` work.
  - All of this is paying it forward to your fourth assignment, where you'll implement your own shell—we call it `stsh`, for the Stanford shell—to imitate the functionality of the shell (`csh`, `bash`, `zsh`, `tcsh`, etc.) you've been using since you started using Unix.
- I'll also introduce the notion of a pipe, the `pipe` and `dup2` system calls, and how they can be used to introduce communication channels between the different processes.
Lecture 07: Process Control, Interprocess Communication

- Let's work through the implementation of a more sophisticated shell: the `simplesh`.
  - This is the best example of `fork`, `waitpid`, and `execvp` I can think of: a miniature shell not unlike those you've been using since the day you first logged into a `myth` machine.
  - `simplesh` operates as a read-eval-print loop—often called a repl—which itself responds to the many things we type in by forking off child processes.
    - Each child process is initially a deep clone of the `simplesh` process.
    - 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 during our second lecture), or even `emacs`.
    - As with traditional shells, a trailing ampersand—e.g. as with `emacs &`—is an instruction to execute the new process in the background without blocking the shell itself.
  - Implementation of `simplesh` is presented on the next slide. Where helper functions don't rely on CS110 concepts, I omit their implementations (but describe them in lecture).
Here's the core implementation of simplesh (full implementation is right here):

```c
int main(int argc, char *argv[]) {
    while (true) {
        char command[kMaxCommandLength + 1];
        readCommand(command, kMaxCommandLength);
        char *arguments[kMaxArgumentCount + 1];
        int count = parseCommandLine(command, arguments, kMaxArgumentCount);
        if (count == 0) continue;
        if (strcmp(arguments[0], "quit") ==) break; // hardcoded builtin to exit shell
        bool isbg = strcmp(arguments[count - 1], ";") == 0;
        if (isbg) arguments[--count] = NULL; // overwrite ";"
        pid_t pid = fork();
        if (pid == 0) execvp(arguments[0], arguments);
        if (isbg) { // background process, don't wait for child to finish
            printf("%d %s\n", pid, command);
        } else { // otherwise block until child process is complete
            waitpid(pid, NULL, 0);
        }
    }
    printf("\n");
    return 0;
}
```
Lecture 07: Process Control, Interprocess Communication

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

```c
int pipe(int fds[]);
```

- `pipe` is particularly useful for allowing parent processes to communicate with spawned child processes.
  - That's because the file descriptor table of the parent is cloned, and that clone is installed in the child.
  - That means the open file table entries references by the parent's pipe endpoints are also referenced by the child's copy of them.
Lecture 07: Process Control, Interprocess Communication

- How does pipe work?
  - To illustrate how pipe works and how arbitrary data can be passed over from one process to a second, let's consider the following program:

```c
int main(int argc, char *argv[]) {
    int fds[2];
    pipe(fds);
    pid_t pid = fork();
    if (pid == 0) {
        close(fds[1]);
        char buffer[6];
        read(fds[0], buffer, sizeof(buffer));
        printf("Read from pipe bridging processes: %s.\n", buffer);
        close(fds[0]);
        return 0;
    }
    close(fds[0]);
    write(fds[1], "hello", 6);
    waitpid(pid, NULL, 0);
    close(fds[1]);
    return 0;
}
```
Lecture 07: Process Control, Interprocess Communication

- How do **pipe** and **fork** work together in this example?
  - The base address of a small integer array called **fds** is shared with the call to **pipe**.
  - **pipe** allocates two descriptors, setting the first to draw from a resource and the second to publish to that same resource.
  - **pipe** then plants copies of those two descriptors into indices 0 and 1 of the supplied array before it returns.
  - The **fork** call creates a child process, which itself inherits a shallow copy of the parents fds array.
    - The reference counts in each of the two open file entries is promoted from 1 to 2 to reflect the fact that two descriptors—one in the parent, and a second in the child—reference each one of them.
    - Immediately after the **fork** call, anything printed to **fds[1]** is readable from the parent's **fds[0]** and the child's **fds[0]**.
    - Similarly, both the parent and child are capable of publishing text to the same resource via their copies of **fds[1]**.
Lecture 07: Process Control, Interprocess Communication

- How do **pipe** and **fork** work together in this example?
  - While not strictly necessary in the example, the parent closes `fds[0]` before it writes to anything to `fds[1]` to emphasize the fact that the parent has no interest in reading anything from the pipe.
  - Similarly, the child closes `fds[1]` before it reads from `fds[0]` to emphasize the fact that it has zero interest in publishing anything to the pipe.
  - For simplicity, I assume the one call to `write` in the parent pressed all six bytes of "hello" (including the newline) in a single call. Similarly, I assume the one call to `read` pulls in those same six bytes into its local buffer with just the one call.
  - As is the case with all programs, I make the concerted effort to donate all resources back to the system before I exit. That's why I include as many `close` calls as I do in both the child and the parent before allowing them to exit.
Here's a more sophisticated example:

○ 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 (full implementation of everything is right here):

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

○ The child process created by `subprocess` executes the provided `command` (assumed to be a `\0`-terminated C string) by calling `"/bin/sh -c <command>"` as we did in our `mysystem` implementation.

- Rather than waiting for `command` to finish, `subprocess` returns a `subprocess_t` with the `command` process’s `pid` and a single descriptor called `supplyfd`.
- By design, arbitrary text can be published to the return value’s `supplyfd` field with the understanding that that same data can be ingested verbatim by the child's `stdin`.
Lecture 07: Process Control, Interprocess Communication

- Let's first implement a test harness to illustrate how `subprocess` should work.
  - By understanding how `subprocess` works for us, we'll have an easier time understanding the details of its implementation.
  - Here's the program, which spawns a child process that reads from `stdin` and publishes everything it reads to its `stdout` in sorted order:

```c
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.supplyfd, "%s\n", words[i]);
    }
    close(sp.supplyfd);
    int status;
    pid_t pid = waitpid(sp.pid, &status, 0);
    return pid == sp.pid && WIFEXITED(status) ? WEXITSTATUS(status) : -127;
}
```
Lecture 07: Process Control, Interprocess Communication

- Key features of the test harness:
  - The program creates a `subprocess_t` running `sort` and publishes eight fancy words to the `supplyfd`, knowing those words flow through the pipe to the child's `stdin`.
  - The parent shuts the `supplyfd` down by passing it to `close`. The reference count of the open file entry referenced by `supplyfd` is demoted from 1 to 0 with that `close` call. That sends an EOF to the process that tries to ingest data from the other end of the pipe.
  - The parent then blocks within a `waitpid` call until the child exits. When the child exits, the parent assumes all of the words have been printed in sorted order to `stdout`.

```
poohbear@myth60$ ./subprocess
ablution
felicity
halcyon
indefatigable
pulchritude
somnolent
susurration
umbrage
poohbear@myth60$
```
Lecture 07: Process Control, Interprocess Communication

- Implementation of subprocess (error checking intentionally omitted for brevity):

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

- The write end of the pipe is embedded into the **subprocess_t**. That way, the parent knows where to publish text so it flows to the read end of the pipe, across the parent process/child process boundary. This is bonafide interprocess communication!
- The child process uses **dup2** to bind the read end of the pipe to its own standard input. Once the reassociation is complete, **fds[0]** can be closed.