Announcements

- **Assignment 1 due Tuesday night.**
  - 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 recipe), go through the submission process now to confirm everything works sans drama.
  - No office hours this Sunday, but office hours on Monday and Tuesday as usual.
- **Topics for today:**
  - Finish up our discussion of filesystems and system calls.
    - Learn how **`open`**, **`read`**, **`write`**, and **`close`** work to grant us the raw ability to create, read, and manipulate files.
    - Discuss what data structures are maintained by the OS (file descriptor tables, file entry tables, and vnode tables) to help support low-level system I/O.
  - Begin our discussion of **`fork`**, how to use it, and how it works.
    - Refer to Lecture 04's slides for most of today's introductory examples, and hopefully everything goes swimmingly well with those and we can advance on to those contained in this slide deck.
    - With any luck, you'll understand how a standard UNIX shell works by the end of next week. Great stuff, this **`fork`**.
    - Sync up against **`/usr/class/cs110/lecture-examples`** to get the examples from both slide decks 4 and 5.

Parent and child processes normally diverge:

- **The following program is a step toward how `fork` gets used in practice:**
  - Check this out (code is in lecture examples folder **`exceptional-control-flow/separate.c`** and can also be viewed right here)

```
int main(int argc, char *argv[]) {
  printf("Before.\n");
  pid_t pid = fork();
  exitIf(pid == -1, kForkFailed, stderr, "Fork function failed.\n");
  printf("After.\n");
  if (pid == 0) {
    printf("I'm the child, and the parent will wait up for me.\n");
    return 110; // contrived exit status
  } else {
    int status;
    exitUnless(waitpid(pid, &status, 0) == pid, kWaitFailed, stderr,
               "Parent's wait for child process with pid %d failed.\n", pid);
    if (WIFEXITED(status)) {
      printf("Child exited with status %d.\n", WEXITSTATUS(status));
    } else {
      printf("Child terminated abnormally.\n");
    }
    return 0;
  }
}
```

  - The above example directs the child process one way, the parent another.
  - The parent process correctly waits for the child to complete, and this example (as opposed to the last example from the last slide deck) actually does the right thing by asserting **`waitpid`**'s return value matches the process id of the child that exited.
  - The parent also lifts exit status information about the child process out of the **`waitpid`** call, and uses the **`WIFEXITED`** macro to examine some high-order bits of this **`status`** argument to confirm the process exited normally, and it also uses the **`WEXITSTATUS`** macro to extract the lower eight bits from its argument to produce the child process's return value.
  - Check out the man page for **`waitpid`** for the good word on all of the different macros. (Your textbook covers them all really nicely as well).

Spawning multiple processes

- **Reaping multiple child processes:**
  - Of course, the parent is allowed to call **`fork`** multiple times, provided it eventually reaps the child processes after they exit.
  - If we want to reap child processes as they exit without concern for the ordering in which they fork, then this does the trick:
  - Check this out (code is in lecture examples folder **`exceptional-control-flow/reap-as-they-exit.c`** and can also be viewed right here)

```
int main(int argc, char *argv[]) {
  for (size_t i = 0; i < kNumChildren; i++) {
    pid_t pid = fork();
    exitIf(pid == -1, kForkFail, stderr, "Fork function failed.\n");
    if (pid == 0) exit(110 + i);
  }

  while (true) {
    int status;
    pid_t pid = waitpid(-1, &status, 0);
    if (pid == -1) break;
    if (WIFEXITED(status)) {
      printf("Child %d exited: status %d\n", pid, WEXITSTATUS(status));
    } else {
      printf("Child %d exited abnormally.\n", pid);
    }
  }

  exitUnless(errno == ECHILD, kWaitFail, stderr, "waitpid failed.\n");
  return 0;
}
```

  - Note we feed a -1 as the first argument to **`waitpid`**. That -1 states we want to hear about **any** child as it exits.
  - Eventually, all children exit (normally or not) and waitpid **correctly** returns -1 to signal that all child processes have ended.
    When **`waitpid`** returns -1, it sets a global variable called **`errno`** to the constant **`ECHILD`** as a signal that -1 was returned because all child processes terminated. Interestingly enough, that's the "error" we want.

Spawning multiple processes, take II

- **We can do the same thing, but reap in the order they are forked.**
  - Look at this lovely program of octuplets (code is in lecture examples folder **`exceptional-control-flow/reap-in-fork-order.c`** and can also be viewed right here)

```
int main(int argc, char *argv[]) {
  pid_t children[kNumChildren];
  for (size_t i = 0; i < kNumChildren; i++) {
    children[i] = fork();
    exitIf(children[i] == -1, kForkFail, stderr, "Fork function failed.\n");
    if (children[i] == 0) exit(110 + i);
  }

  for (size_t i = 0; i < kNumChildren; i++) {
    int status;
    exitUnless(waitpid(children[i], &status, 0) == children[i],
               kWaitFail, stderr, "Intentional wait on child %d failed.\n", children[i]);
    exitUnless(WIFEXITED(status) && WEXITSTATUS(status) == 110 + i,
               kExitFail, stderr, "Correct child %d exited abnormally.\n");
  }

  return 0;
}
```

  - This version spawns and reaps child processes in some first-spawned-first-reaped (let's invent an acronym: FSFR) manner.
  - Understand, of course, that the child processes aren't required to exit or otherwise terminate 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 actually finishes. But the process zombies (as they're called) are certainly reaped in the order they were forked.