# Announcements

- **Assignment 1 due a week from tomorrow night.**
- **Office hours start tomorrow night.**
  - The full matrix of office hours is posted right here.
  - Office hours are great for asking questions about the lecture material and the assignment specifications that can't be easily managed on Piazza.
  - Office hours are not for debugging your code.
- **Reading for this week:**
  - Skim Sections 6.1, 6.2, and 6.3 of the Salzer and Kaashoek textbook, which discusses performance, caching, and multilevel memory hierarchies. The material is a primer for the optimization work you'll be doing for Assignment 2, which goes out a week from tomorrow.
  - Do a close reading of the Bryant & O'Hallaron reader, Chapter 2 (much of which you know about already: **`open`**, **`read`**, **`write`**, etc) and Chapter 1 (in that order).
- **Expect to finish up filesystems, naming, and layering today, and we'll begin exceptional flow control if we finish a little early.**
  - All exceptional control flow examples can be found in **`/usr/class/cs110/lecture-examples/exceptional-control-flow`**.

# Exceptional Control Flow and Multiprocessing: First Program

- **New system call: `fork`**
  - Here is the simplest of programs that knows how to create other processes. It uses a system call named **`fork`**.
  - Code is in lecture examples folder: **`exceptional-control-flow/basic-fork.c`**. Code is also right here.

```
static const int kForkFailed = 1;
int main(int argc, char *argv[]) {
  printf("Greetings from process %d! (parent %d)\n", getpid(), getppid());
  pid_t pid = fork();
  exitIf(pid == -1, kForkFailed, stderr, "fork function failed.\n");
  printf("Bye-bye from process %d! (parent %d)\n", getpid(), getppid());
  return 0;
}
```

- **`fork` is called once, but it returns twice.**
  - **`fork`** knows how to clone the calling process, create a virtually identical copy of it, and schedule it as if the second copy of the original were running all along. All segments (data, bss, init, stack, heap, text) are faithfully replicated, and all open file descriptors from the first are duplicated and donated to the clone.
  - The only difference: **`fork`**'s return value in the new process (the **child**) is 0, and fork's return value in the spawning process (the **parent**) is the child's process id. The return value can be used to dispatch each of the two processes in a different direction (although in this introductory example, we don't do that).
  - As a result, the output of the above program is really the output of two processes. We should expect to see a single greeting but two separate bye-byes.
  - Key observation: Each of the bye-byes is inserted into the console by two different processes. The order each line executes, in principle, cannot be predicted. The system's scheduler dictates whether the child or the parent gets to print its bye-bye first (although on multiple core machines, child and parent could be running in parallel).

# Explosive `fork` graphs

- **Understand the workflow, beware of `fork` bombing your system.**
  - While you rarely have reason to use fork this way (and you shouldn't, or you'll begin to tax the shared system you're working on and/or hit your limit on the number of concurrent processes), it's instructive to trace through a short program where forked processes themselves call **`fork`**.
  - Code is in lecture examples folder: **`exceptional-control-flow/fork-puzzle.c`**. Code is also right here.
  - Check this out:

```
static const char const *kTrail = "abcd";
static const int kForkFail = 1;
int main(int argc, char *argv[]) {
  size_t trailLength = strlen(kTrail);
  for (size_t i = 0; i < trailLength; i++) {
    printf("%c\n", kTrail[i]);
    pid_t pid = fork();
    exitIf(pid == -1, kForkFail, stderr, "Call to fork failed.");
  }
  return 0;
}
```

  - Reasonably obvious: One a is printed by the soon-to-be-great-grandaddy process.
  - Less obvious: The first child and the parent each return from fork and continue running in mirror processes, each with their own copy of the global "abcd" string, and each advancing to the i++ line within a loop that promotes a 0 to 1. It's hopefully clear now that two b's will be printed.
  - Key questions to answer:
    - How many c's get printed?
    - How many d's get printed?
    - Are both b's necessarily printed one after another?

# Synchronizing execution between parent and child

- **`waitpid` instructs a process to block until another process exits.**
  - The first argument specifies the wait set, which for the moment is just the id of the child process that needs to complete before **`waitpid`** can return. (There are more possibilities, but we'll explore them later).
  - The second argument supplies the address of an integer where child termination status information can be placed (or we can pass in **`NULL`** if we don't care to get that information).
  - The third argument is a collection of bit flags we'll study later. For the time being, we'll just go with 0 as the required parameter value, which means that **`waitpid`** should only return when a child exits.
  - The return value is the id of the child process that exited, or -1 if **`waitpid`** was called and there were no child processes to wait on.
  - Code for next small example is in lecture examples folder: **`exceptional-control-flow/slave-master.c`**. Code is also right here.
  - Here's that example:

```
static const int kForkFailure = 1;
int main(int argc, char *argv[]) {
  printf("I'm unique and just get printed once.\n");
  pid_t pid = fork(); // returns 0 within child, returns pid of child within fork
  exitIf(pid == -1, kForkFailure, stderr, "Call to fork failed... aborting.\n");
  bool parent = pid != 0;
  if ((random() % 2 == 0) == parent) sleep(1); // force exactly one of the two to sleep
  if (parent) waitpid(pid, NULL, 0); // parent shouldn't exit until it knows its child has finished
  printf("I get printed twice (this one is being printed from the %s).\n", parent  ? "parent" : "child");
  return 0;
}
```

  - The above example uses a coin flip to seduce one of the two processes to sleep for a second, which is normally all the time the other process needs to print what it needs to print first.
  - The parent thread elects to wait for the child thread to exit before it allows itself to exit. It's a akin to a parent not being able to go to bed until he or she knows the child has, and it's emblematic of the types of synchronization directives we'll be seeing a lot of this quarter.
  - Understand that the final **`printf`** gets executed twice. The child is always the first to execute it, however, because the parent is blocked in the **`waitpid`** call until the child executes **everything**.