Announcements

- **Assignment 1's hard deadline is tonight at 11:59 p.m.**
  - We expect to crank on Assignment 1 submissions over the course of the week and provide full grade reports by Saturday.
- **Assignment 2 goes out today as well, due a week from Wednesday.**
  - It's an opportunity to leverage your CS107 optimization skills and a few new Linux tools to speed up an existing system—your Assignment 1 filesystem — to run as quickly as possible. In particular, you'll augment your Assignment 1 implementation and the application that uses it to minimize the number of I/O operations, using main memory and its caches as a **file** cache. Very neat assignment, in my opinion!).
- **Readings:**
  - Continue skimming Sections 6.1, 6.2, and 6.3 of the Salzer and Kaashoek textbook, which discuss performance, caching, and multilevel memory hierarchies. The material is a primer for the optimization work you're currently managing for Assignment 2.
  - Finish reading B&O Chapter 2 (Chapter 10 of full textbook) so you can confirm that you know most of the material there, as I covered much of it during the first two weeks of lecture.
  - Finish reading B&O Chapter 1 (Chapter 8 of the full textbook), focusing on Section 5, which covers process groups, signals, and signal handlers, all three of which will contribute to your Assignment 3 submission, which goes out the day Assignment 2 is due. (You're going to love that one as well, I promise!)

Today

- **Today's topics are all kinds of awesome.**
  - We have the majority of the lecture examples from Lecture 06's slides to work through. In particular, we want to implement a collection of miniature shells to illustrate how **`fork`**, **`waitpid`**, and **`execvp`** work together to do what we need to do.
  - I want to introduce the notion of a pipe, the **`pipe`** and **`dup2`** functions, and how they can be used to foster more sophisticated communication between the parent process and any forked child processes. In particular, we'll learn how the parent process can print to one end of a pipe such that the text flows across the parent/child process boundary and can be consumed via the child process's **`stdin`**.
  - Later this week and early next week, we'll begin our duscussion of signals, signal handlers, and how to temporarily suspend the delivery of a signal category.

Introduction to Signals

- **Signals**
  - A signal is a small message that notifies a process that an event of some type occurred. Signals are often sent by the kernel, but they can be sent from other processes as well.
  - A signal handler is a function that executes in response to the arrival and consumption of a signal.
  - You're already familiar with some signals, even if you've not referred to them by that name before:
    - You haven't really programmed in C before unless you've dereferenced a **`NULL`** pointer. When that happens, the kernel delivers a signal of type **`SIGSEGV`**, informally known as a segmentation fault (or a SEGmentation Violation, or **`SIGSEGV`** for short). Unless you install a custom signal handler to manage the signal differently, a **`SIGSEGV`** terminates the program and generates a core dump.
    - Whenever a process commits an integer-divide-by-zero (and, in some cases, a floating-point divide by zero), the kernel hollers and issues a **`SIGFPE`** signal to the offending process. (By default, the program terminates with a message of the **`SIGFPE`** and a core dump is produced).
    - When you type ctrl-c, the kernel issues a **`SIGINT`** signal to the foreground process (and by default, the program is terminated).
    - When you type ctrl-z, the kernel issues a **`SIGTSTP`** signal to the foreground process (and by default, the process is suspended until a subsequent **`SIGCONT`** signal instructs it to resume).
    - Whenever a child process exits (either normally or abnormally), the kernel protentially delivers a **`SIGCHLD`** signal to the parent process.
      By default, the signal is ignored (and in fact, by default the kernel doesn't even deliver it unless the parent process registers an interest in receiving them). This particular signal type is instrumental to allowing forked child processes to run in the background while the parent process moves on to do its own work without blocking on a **`waitpid`** call. The parent process, however, is not relieved of its obligation to reap child process zombies, so the parent process will typically register code to be invoked whenever a child process terminates. Doing so prompts the kernel to begin issuing **`SIGCHLD`** signals so that the registered **`SIGCHLD`** handler can reap the zombies via **`waitpid`**.

First Signal Handler Example

- **Source code can install a signal handler to *catch* and handle a certain type of signal in a way that's different than the default.**
  - Here's a carefully coded example that illustrates how to implement and install a **`SIGCHLD`** handler (over two slides). The premise is that dad takes his five kids out to play. Each of the five children plays for a different length of time. When all five children are tired of playing, the six of them (five kids and their dad) all go home.
  - Understand that the parent process is modeling dad, and that the child processes are modeling his children. (Code can be found right here).

```
static const size_t kNumChildren = 5;
static size_t numChildrenDonePlaying = 0;

static void reapChild(int sig) {
  exitIf(waitpid(-1, NULL, 0) == -1, kWaitFailed,
         stderr, "waitpid failed within reapChild sighandler.\n");
  numChildrenDonePlaying++;
  sleep(1); // represents time spent doing other useful work
}

int main(int argc, char *argv[]) {
  printf("Let my five children play while I take a nap.\n");
  exitIf(signal(SIGCHLD, reapChild) == SIG_ERR, kSignalFailed,
         stderr, "Failed to install SIGCHLD handler.\n");
  for (size_t kid = 1; kid <= 5; kid++) {
    pid_t pid = fork();
    exitIf(pid == -1, kForkFailed, stderr, "Child #%zu doesn't want to play.\n", kid);
    if (pid == 0) {
      sleep(3 * kid); // sleep emulates "play" time
      printf("Child #%zu tired... returns to dad.\n", kid);
      return 0;
    }
  }
```

First Signal Handler Example (Continued)

- **Here's the second half of the program from the last slide:**
  - Notice the use of the **`snooze`** function, which is my own implementation of an uninterruptable **`sleep`**:

```
  while (numChildrenDonePlaying < kNumChildren) {
    printf("At least one child still playing, so dad nods off.\n");
    snooze(5); // implementation in /usr/class/cs110/local/include/sleep-utils.h
    printf("Dad wakes up! ");
  }

  printf("All children accounted for.  Good job, dad!\n");
  return 0;
}
```

  - The above code is specifically crafted so each child process exits about 3 seconds apart. **`reapChild`**, of course, catches and handles each of the **`SIGCHLD`** signals delivered by the kernel as each forked child process exits. **`reapChild`** is contrived to take about a second to execute to completion, so each **`reapChild`** finishes about two seconds before the next child process exits.
  - The **`signal`** function doesn't allow for state to be shared with the signal handler, so we have no choice but to use globals.
  - Here's the fairly predictable output of the above program.

```
myth22> ./independent-children
Let my five children play while I take a nap.
At least one child still playing, so dad nods off.
Child #1 tired... returns to dad.
Child #2 tired... returns to dad.
Dad wakes up! At least one child still playing, so dad nods off.
Child #3 tired... returns to dad.
Child #4 tired... returns to dad.
Dad wakes up! At least one child still playing, so dad nods off.
Child #5 tired... returns to dad.
Dad wakes up! All children accounted for.  Good job, dad!
myth22>
```

Variation on a Theme

- **Consider this next program:**
  - The program is largely the same, except that each child process takes *the same amount of time to complete* (program is over two slides)
  - The code for this example can be found right here.

```
static const size_t kNumChildren = 5;
static size_t numChildrenDonePlaying = 0;

static void reapChild(int sig) {
  exitIf(waitpid(-1, NULL, 0) == -1, kWaitFailed,
         stderr, "waitpid failed within reapChild sighandler.\n");
  numChildrenDonePlaying++;
  sleep(1); // represents time that useful work might be done
}

int main(int argc, char *argv[]) {
  printf("Let my five children play while I take a nap.\n");
  exitIf(signal(SIGCHLD, reapChild) == SIG_ERR, kSignalFailed,
         stderr, "Failed to install SIGCHLD handler.\n");
  for (size_t kid = 1; kid <= 5; kid++) {
    pid_t pid = fork();
    exitIf(pid == -1, kForkFailed, stderr, "Child #%zu doesn't want to play.\n", kid);
    if (pid == 0) {
      sleep(3); // all kids play together for three seconds
      printf("Kid #%zu done playing... runs back to dad.\n", kid);
      return 0;
    }
  }
```

Variation on a Theme (Continued)

- **Here's the second part of the program:**
  - (This part is precisely the same as it was before).

```
  while (numChildrenDonePlaying < kNumChildren) {
    printf("At least one child still playing, so dad nods off.\n");
    snooze(5);
    printf("Dad wakes up! ");
  }

  printf("All children accounted for.  Good job, dad!\n");
  return 0;
}
```

  - The primary difference here is that all child processes exit, more or less, at the same time. And while it is the case that the kernel will issue five **`SIGCHLD`** signals, not all of them prompt a dedicated execution of **`reapChild`**.
  - Don't believe me? Check out the reproducible test run of this second version (where all five children finish playing pretty much much simultaneously).

```
myth22> ./close-children
Let my five children play while I take a nap.
At least one child still playing, so dad nods off.
Kid #1 done playing... runs back to dad.
Kid #3 done playing... runs back to dad.
Kid #4 done playing... runs back to dad.
Kid #2 done playing... runs back to dad.
Kid #5 done playing... runs back to dad.
Dad wakes up! At least one child still playing, so dad nods off.
Dad wakes up! At least one child still playing, so dad nods off.
Dad wakes up! At least one child still playing, so dad nods off.
Dad wakes up! At least one child still playing, so dad nods off.
Dad wakes up! At least one child still playing, so dad nods off.
Dad wakes up! At least one child still playing, so dad nods off.
<ctrl-c>
myth22>
```

Variation on a Theme (Continued)

- **Here's what happens:**
  - One of the child processes finishes before the other four, and the kernel sends a **`SIGCHLD`** signal to the parent on its behalf.
  - The **`SIGCHLD`** signal is caught, and **`reapChild`** executes (over the course of one second) to handle it.
  - During that one second, the second of the five child processes exits, and the corresponding **`SIGCHLD`** signal is recorded but blocked until the call to first call to **`reapChild`** exits. The third, fourth, and fifth child processes all exit while the first **`reapChild`** is still running, but their **`SIGCHLD`** signals are discarded, as the kernel maintains not a **count** of pending **`SIGCHLD`** signals, but rather a single bit that records whether **one or more** signals have arrived.
  - When the first call to **`reapChild`** exits, the block on pending **`SIGCHLD`** signals is lifted. The kernel soon detects the high **`SIGCHLD`** bit, unaware of how many **`SIGCHLD`** signals were actually fired. **`reapChild`** is invoked to handle all the outstanding **`SIGCHLD`** signals, so that **`reapChild`** executes only one more time.
  - Redux: **`numChildrenDonePlaying`** global variable is only incremented twice, and the parent process (the process modeling daddy) repeatedly takes five second naps for all of eternity.

The Solution...

- **... is simple, provided you understand how pending signals are recorded as bools and not counts.**
  - **`reapChild`** must be implemented to account for the possibility that many **`SIGCHLD`** signals were fired, not just one (e.g. that many child processes finished during the prior **`reapChild`** call, during the window when **`SIGCHLD`** signals were received but blocked).
  - Each call to **`waitpid`** should include a third argument of **`WNOHANG`**, which is a flag instructing waitpid to not block on children that haven't exited yet. When **`WNOHANG`** is used, we want to break on two different return values:
    - 0, which means that there are other child processes, but that none of them have transitioned to zombie status, and
    - -1, which means what it's meant before: that there are no children (which is confirmed by the fact that **`errno`** is set to **`ECHILD`**).

The Solution

- **The solution makes use of `WNOHANG`.**
  - Here is the **correct** implementation of **`reapChild`** (look here for the code).

```
static void reapChild(int sig) {
  pid_t pid;
  while (true) {
    pid = waitpid(-1, NULL, WNOHANG);
    if (pid <= 0) break;
    numChildrenDonePlaying++;
  }
  exitUnless(pid == 0 || errno == ECHILD, kWaitFailed,
             stderr, "waitpid failed within reapChild sighandler.\n");
  sleep(1); // represents time that useful work might be done
}
```

  - Here's the output of the repaired program:

```
myth22> ./close-children-improved
Let my five children play while I take a nap.
At least one child still playing, so dad nods off.
Kid #1 done playing... runs back to dad.
Kid #2 done playing... runs back to dad.
Kid #4 done playing... runs back to dad.
Kid #3 done playing... runs back to dad.
Kid #5 done playing... runs back to dad.
Dad wakes up! All children accounted for.  Good job, dad!
myth22>
```

Introduction to Synchronization Issues

- **Synchronization, multi-processing, parallelism, and concurrency**
  - All central themes of the course, and all very powerful features.
  - Very difficult to understand and get right, and all kinds of concurrency issues and race conditions can appear unless you code very carefully.
  - Consider the following program, which is a gesture to where your Assignment 3 shell will end up when it's all done (code for entire program can be found right here):

```
static void reapChild(int sig) {
  pid_t pid;
  while (true) {
    pid = waitpid(-1, NULL, WNOHANG);
    if (pid <= 0) break;
    printf("Job %d removed from job list.\n", pid);
  }
  exitUnless(pid == 0 || errno == ECHILD, kWaitFailed,
             stderr, "waitpid failed within reapChild sighandler.\n");
}

int main(int argc, char *argv[]) {
  exitIf(signal(SIGCHLD, reapChild) == SIG_ERR, kSignalFailed,
         stderr, "signal function failed.\n");
  for (size_t i = 0; i < 3; i++) {
    pid_t pid = fork();
    exitIf(pid == -1, kForkFailed,
           stderr, "fork function failed.\n");
    if (pid == 0) {
      char *listArguments[] = {"date", NULL};
      exitIf(execvp(listArguments[0], listArguments) == -1,
             kExecFailed, stderr, "execvp function failed.\n");
    }
    snooze(1); // represents meaningful time spent
    printf("Job %d added to job list.\n", pid);
  }

  return 0;
}
```

Baby's First Concurrency Issue!

- **Look at a test run of the previous program:**
  - How wrong is this?!!

```
myth22> ./job-list-synchronization
Fri Jan 24 12:49:00 PST 2014
Job 11129 removed from job list.
Job 11129 added to job list.
Fri Jan 24 12:49:01 PST 2014
Job 11130 removed from job list.
Job 11130 added to job list.
Fri Jan 24 12:49:02 PST 2014
Job 11131 removed from job list.
Job 11131 added to job list.
myth22>
```

  - The huge issue here is that each of the child processes (which quickly publish the date to the console via a **`fork`**/**`execvp`** pair) exits and prompts the kernel to fire a **`SIGCHLD`** signal at the parent before the parent makes it through its one second nap. The **`reapChild`** function executes in full (and "removes" the job from some job list) before the parent advances to the point where it "adds" the job to the same job list.
  - That's messed up! Welcome to the land of concurrent contexts operating on shared data structures.
  - The solution:
    - Informally, we need to programmatically block **`SIGCHLD`** signals from being handled until the parent manages to add the process to the job list.
    - Formally, we need to use **`sigset_t`** masks to temporarily block **`SIGCHLD`** signals from being handled until the parent has gotten through the code that adds the job to the job list.

Baby's First Concurrency Solution

- **The concurrency issue is unacceptable.**
  - We must take steps to programmatically suspend the handling of **`SIGCHLD`** siganls until the parent is prepared to handle them.
  - Enter the **`sigset_t`** type and the collection of functions that can be used to temporarily block the receipt of one or more signal types.
  - Here's the working **`main`** function (code for entire program can be found right here):

```
int main(int argc, char *argv[]) {
  exitIf(signal(SIGCHLD, reapChild) == SIG_ERR, kSignalFailed,
         stderr, "signal function failed.\n");
  sigset_t mask;
  sigemptyset(&mask);
  sigaddset(&mask, SIGCHLD);
  for (size_t i = 0; i < 3; i++) {
    sigprocmask(SIG_BLOCK, &mask, NULL);
    pid_t pid = fork();
    exitIf(pid == -1, kForkFailed,
           stderr, "fork function failed.\n");
    if (pid == 0) {
      sigprocmask(SIG_UNBLOCK, &mask, NULL);
      char *listArguments[] = {"date", NULL};
      exitIf(execvp(listArguments[0], listArguments) == -1,
             kExecFailed, stderr, "execvp function failed.\n");
    }

    snooze(1);
    printf("Job %d added to job list.\n", pid);
    sigprocmask(SIG_UNBLOCK, &mask, NULL); // begin handling SIGCHLD signals again
  }

  return 0;
}
```

  - The **`sigset_t`** figure manages the set of signal types to be blocked and unblocked via calls to **`sigprocmask`**.
    - Note the call to **`sigprocmask(SIG_BLOCK, &mask, NULL);`**, which informs the kernel that the calling process isn't interested in handling any **`SIGCHLD`** events until after the spawned child process is added to the job list. Note, also, the call to **`sigprocmask(SIG_UNBLOCK, &mask, NULL);`** appears after the child process pid has been added to the job list (gestured via **`printf`**).
    - As it turns out, the forked process inherits the blocked signals vector, so it too needs to lift the block via its own call to **`sigprocmask(SIG_UNBLOCK, &mask, NULL);`**. In this example, it doesn't matter, but it would matter if the forked child process itself forked off child processes of its own.