Lecture Overview

- All about signals!

Reading

- Bryant & O'Hallaron: Section 5 of Chapter 1 (reader) or 8 (full textbook)
Accessing Code Examples

- Today's lecture examples reside within:
  
  `/usr/class/cs110/lecture-examples/processes`.
  
  - First `ssh` into a myth machine (`ssh yourusername@myth.stanford.edu`). When prompted for your password, it is normal for the text not to appear as you enter your password. Once logged onto a myth machine, `cd` into the above directory.
  
  - To get started, type:
    
    `git clone /usr/class/cs110/lecture-examples cs110-lecture-examples`
    
    at the command prompt to create a local copy of the master.
  
  - Each time I mention there are new examples (or whenever you think to), descend into your local copy and type `git pull`. Doing so will update your local copy to match whatever the master has become.
Signals

- **A signal** is a small message that notifies a process that an event of some type occurred. Signals are a higher-level software form of exceptional control flow that allows processes and the kernel to interrupt other processes (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 types of signals, even if you've not referred to them by that name before.
  - You haven't truly programmed in C before unless you've unintentionally dereferenced a **NULL** pointer.
  - When that happens, the kernel delivers a signal of type **SIGSEGV**, informally known as a segmentation fault (or a **SEG**mentation **V**iolation, 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.

- Each signal category (e.g. **SIGSEGV**) is represented internally by some number (e.g. 11). For example, C **#defines** **SIGSEGV** to be the number 11.
Signals (continued)

- Low-level hardware exceptions are processed by the kernel’s exception handlers and would not normally be visible to user processes.
  - Signals provide a way to expose these types of exceptions to user processes.

Other signal types:

- Whenever a process commits an integer-divide-by-zero (and, in some cases, a floating-point divide by zero on older architectures), the kernel hollers and issues a **SIGFPE** signal to the offending process. By default, the program handles the **SIGFPE** by printing an error message announcing the zero denominator and generating a core dump.
  - When you type CTRL-C, the kernel sends a **SIGINT** to the foreground process (and by default, that foreground is terminated).
  - When you type CTRL-Z, the kernel issues a **SIGTSTP** to the foreground process (and by default, the foreground process is halted until a subsequent **SIGCONT** signal instructs it to continue).
  - When a process attempts to publish data to the write end of a pipe after the read end has been closed, the kernel delivers a **SIGPIPE** to the offending process. The default **SIGPIPE** handler terminates the program.
## Signals (continued)

Some common signals (30 types are supported on Linux systems):

<table>
<thead>
<tr>
<th>#</th>
<th>Name</th>
<th>Default Action</th>
<th>Corresponding Event</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>SIGINT</td>
<td>Terminate</td>
<td>Interrupt from keyboard (CTRL-C)</td>
</tr>
<tr>
<td>9</td>
<td>SIGKILL</td>
<td>Terminate</td>
<td>Kill program. This signal can’t be caught or ignored. Doesn’t allow children to be reaped</td>
</tr>
<tr>
<td>11</td>
<td>SIGSEGV</td>
<td>Terminate and dump core</td>
<td>Invalid memory reference (seg fault)</td>
</tr>
<tr>
<td>13</td>
<td>SIGPIPE</td>
<td>Terminate</td>
<td>Wrote to a pipe that has no reader</td>
</tr>
<tr>
<td>15</td>
<td>SIGTERM</td>
<td>Terminate</td>
<td>Software termination signal (allows children to possibly be reaped)</td>
</tr>
<tr>
<td>17</td>
<td>SIGCHLD</td>
<td>Ignore</td>
<td>A child process has stopped or terminated</td>
</tr>
<tr>
<td>18</td>
<td>SIGCONT</td>
<td>Ignore</td>
<td>Continue process if stopped</td>
</tr>
<tr>
<td>19</td>
<td>SIGSTOP</td>
<td>Stop until next SIGCONT</td>
<td>Stop signal not from terminal. This signal can’t be caught or ignored.</td>
</tr>
<tr>
<td>20</td>
<td>SIGTSTP</td>
<td>Stop until next SIGCONT</td>
<td>Stop signal from terminal (CTRL-Z).</td>
</tr>
</tbody>
</table>
Signals (continued)

One signal type that is important to multiprocessing:

- Whenever a child process **changes state**—that is, it exits, crashes, stops, or resumes from a stopped state—the kernel sends a **SIGCHLD** signal to the process' parent.
  - By default, the signal is ignored. In fact, we've ignored it until now and gotten away with it.
  - 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 still required to reap child processes, so the parent will typically register a custom **SIGCHLD** handler to be asynchronously invoked whenever a child process changes state.
  - These custom **SIGCHLD** handlers almost always include calls to **waitpid**, which can be used to surface the pids of child processes that've changed state. If the child process of interest actually terminated, either normally or abnormally, the **waitpid** also reaps the now-zombie child process.
Signals (continued)

Our first signal handler example: Disneyland

- This example that shows how to implement and install a `SIGCHLD` handler.
- The premise: dad takes his five kids out to play. Each child plays for a different length of time. When all five kids are done playing, the six of them all go home. The parent process models dad and the five child processes model his children. (Full program is [here](#).)
- Each child process exits at three-second intervals. `reapChild` handles each of the `SIGCHLD` signals delivered as each child process exits (or rather, when each child has had enough of playing).
- The `signal` prototype doesn't allow for state to be shared via parameters, so we have no choice but to use global variables.

```c
static const size_t kNumChildren = 5;
static size_t numDone = 0;

static void reapChild(int unused) {
    waitpid(-1, NULL, 0);
    numDone++;
}

int main(int argc, char *argv[]) {
    printf("Let my five children play while I take a nap.\n\n");
    signal(SIGCHLD, reapChild);
    for (size_t kid = 1; kid <= 5; kid++) {
        if (fork() == 0) {
            sleep(3 * kid); // sleep emulates "play" time
            printf("Child \%zu tired... returns to dad.\n", kid);
            return 0;
        }
    }
    while (numDone < kNumChildren) {
        printf("At least one child still playing, so dad nods off.\n\n");
        snooze(5); // our version of sleep that allows the full 5
                // seconds to pass, even if signals arrive
        printf("Dad wakes up! ");
    }
    printf("All children accounted for. Good job, dad!\n\n");
    return 0;
}
```
Here's the output of the previous program.

- Dad's wake-up times (at t = 5 sec, t = 10 sec, etc.) interleave the various finish times (3 sec, 6 sec, etc.) of the children, and the output published below reflects that.
- The **SIGCHLD** handler is invoked 5 times, each in response to some child process finishing up.

```bash
myth60$ ./five-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.
Dad wakes up! At least one child still playing, so dad nods off.
Child #2 tired... returns to dad.
Child #3 tired... returns to dad.
Dad wakes up! At least one child still playing, so dad nods off.
Child #4 tired... returns to dad.
Child #5 tired... returns to dad.
Dad wakes up! All children accounted for.  Good job, dad!
myth60$
```
Now consider the scenario where the five kids are the same age and run about Disneyland for the same amount of time. Restated, \( \text{sleep}(3 * \text{kid}) \) is now \( \text{sleep}(3) \) so all five children flashmob dad when they're all done.

The output presented below suggests dad never figured out that all five kids are done, so dad keeps napping and the program runs forever. Why? Because the \( \text{numDone} \) global never gets big enough to match \( k\text{NumChildren} \). Why is that? The next slide explains why!
Signals (continued)

Advancing our understanding of signal delivery and handling:

- The five children all return to dad at the same time, but dad can't tell.
- Why? Because if multiple signals come in at the same time, the signal handler is only run once.
  - If three SIGCHLD signals are delivered while dad is off the processor, the operating system only records the fact that one or more SIGCHLDs came in!
  - When the parent is forced to execute its SIGCHLD handler, it must do so on behalf of the one or more signals that may have been delivered since the last time it was on the processor.
- That means our SIGCHLD handler needs to call waitpid in a loop, as with:

```c
static void reapChild(int unused) {
    while (true) {
        pid_t pid = waitpid(-1, NULL, 0);
        if (pid < 0) break;
        numDone++;
    }
}
```
The improved `reapChild` implementation seemingly fixes the `pentuplets` program, but it changes the behavior of the first `five-children` program.

- When the first child in the original program has exited, the other children are still out playing.
- The `SIGCHLD` handler will call `waitpid` once, and it will return the pid of the first child.
- The `SIGCHLD` handler will then loop around and call `waitpid` a second time.
- This second call will block until the second child exits three seconds later, preventing dad from returning to his nap.

We need to instruct `waitpid` to only reap children that have exited but to return without blocking, even if there are more children still running. We use `WNOHANG` for this.

Remember: If the calling process has no more children, then `waitpid` returns -1 and sets `errno` to `ECHILD`. (And if `waitpid` was interrupted by a signal, then it returns -1 and sets `errno` to `EINTR`.)

```c
static void reapChild(int unused) {
    while (true) {
        pid_t pid = waitpid(-1, NULL, WNOHANG); // introducing WNOHANG
        if (pid <= 0) break; // note the < is now a <=
        numDone++;
    } assert(pid == 0 || errno == ECHILD); // pid could be 0 because of WNOHANG
}
```
Signals (continued)

- All **SIGCHLD** handlers generally have this **while** loop structure.
  - Note we changed the `if (pid < 0)` test to `if (pid <= 0)`.
  - A return value of -1 typically means that there are no child processes left.
  - A return value of 0—that's a new possible return value for us—means there are other child processes, and we would have normally waited for them to exit, but we're returning instead because of the **WNOHANG** being passed in as the third argument.

- The third argument supplied to **waitpid** can include several flags bitwise-OR'ed together.
  - **WUNTRACED** informs **waitpid** to block until some child process has either ended or been stopped.
  - **WCONTINUED** informs **waitpid** to block until some child process has either ended or resumed from a stopped state.
  - **WUNTRACED | WCONTINUED | WNOHANG** asks that **waitpid** return information about a child process that has changed state (i.e. exited, crashed, stopped, or continued) but to do so without blocking.
Masking Signals, Deferring Handlers

Synchronization, multiprocessing, parallelism, and concurrency.

- All of the above are central themes of the course, and all are difficult to master.
- When you introduce multiprocessing (as you do with `fork`) and asynchronous signal handling (as you do with `signal`), concurrency issues and race conditions will creep in unless you code carefully.
- Signal handlers and the asynchronous interrupts that come with them mean that your normal execution flow can, in general, be interrupted at any time to handle signals.
- Consider the program on the next slide, which is a nod to the type of code you'll write for Assignment 3. The full program, with error checking, is right here.
  - The program spawns off three child processes at one-second internals.
  - Each child process prints the date and time it was spawned.
  - The parent also maintains a pretend job list. It's pretend, because rather than maintaining a data structure with active process ids, we just inline `printf` statements stating where pids would be added to and removed from the job list data structure instead of actually doing it.
Masking Signals, Deferring Handlers

The program is on the left and some test runs are on the right.

```c
// job-list-synchronization.c
static void reapProcesses(int sig) {
    while (true) {
        pid_t pid = waitpid(-1, NULL, WNOHANG);
        if (pid <= 0) break;
        printf("Job %d removed from job list.\n", pid);
    }
}

char *const kArguments[] = {"date", NULL};

int main(int argc, char *argv[]) {
    signal(SIGCHLD, reapProcesses);
    for (size_t i = 0; i < 3; i++) {
        pid_t pid = fork();
        if (pid == 0) execvp(kArguments[0], kArguments);
        sleep(1); // force parent off CPU
        printf("Job %d added to job list.\n", pid);
    }
    return 0;
}
```

Reading: B&O's Exceptional Control
Flow chapter, section 5
The most troubling part of the output on the right is the fact that process ids are being removed from the job list before they're added!

It's true that we're artificially pushing the parent off the CPU with that `sleep(1)` call, which allows the child process to churn through its `date` program and print the date and time to `stdout`.

Even if the `sleep(1)` is removed, it's possible that the child executes `date`, exits, and forces the parent to execute its `SIGCHLD` handler before the parent gets to its own `printf`. The fact that it's possible means we have a concurrency issue.

We need some way to block `reapProcesses` from running until it's safe or sensible to do so. Restated, we'd like to postpone `reapProcesses` from executing until the parent's `printf` has returned.
The kernel provides directives that allow a process to temporarily ignore signal delivery.

The subset of directives that interest us are presented below:

```c
int sigemptyset(sigset_t *set);
int sigaddset(sigset_t *additions, int signum);
int sigprocmask(int op, const sigset_t *delta, sigset_t *existing);
```

- The `sigset_t` type is a small primitive—usually a 32-bit, unsigned integer—that's used as a bit vector of length 32. Since there are just under 32 signal types, the presence or absence of `signums` can be captured via an ordered collection of 0's and 1's.
- `sigemptyset` is used to initialize the `sigset_t` at the supplied address to be the empty set of signals. We generally ignore the return value.
- `sigaddset` is used to ensure the supplied signal number, if not already present, gets added to the set addressed by `additions`. Again, we generally ignore the return value.
- `sigprocmask` adds (if `op` is set to `SIG_BLOCK`) or removes (if `op` is set to `SIG_UNBLOCK`) the signals reachable from `delta` to/from the set of signals being ignored at the moment. The third argument is the location of a `sigset_t` that can be updated with the set of signals being blocked right before the call. Again, we generally ignore the return value.
Masking Signals, Deferring Handlers

- Here's a function that imposes a block on `SIGCHLDs`:

```c
static void imposeSIGCHLDBlock() {
    sigset_t set;
    sigemptyset(&set);
    sigaddset(&set, SIGCHLD);
    sigprocmask(SIG_BLOCK, &set, NULL);
}
```

- Here's a C++ function that lifts the block on the signals packaged within the supplied vector:

```cpp
static void liftSignalBlocks(const vector<int>& signums) {
    sigset_t set;
    sigemptyset(&set);
    for (int signum: signums) sigaddset(&set, signum);
    sigprocmask(SIG_UNBLOCK, &set, NULL);
}
```

- Note that `NULL` is passed as the third argument to both `sigprocmask` calls. That just means that I don't care to hear about what signals were being blocked before the call.
Masking Signals, Deferring Handlers

Here’s an improved version of the earlier example.

```c
// job-list-synchronization-improved.c
c char * const kArguments[] = {"date", NULL};
int main(int argc, char *argv[]) {
  signal(SIGCHLD, reapProcesses);
  sigset_t set;
  sigemptyset(&set);
  sigaddset(&set, SIGCHLD);
  for (size_t i = 0; i < 3; i++) {
    sigprocmask(SIG_BLOCK, &set, NULL);
    pid_t pid = fork();
    if (pid == 0) {
      sigprocmask(SIG_UNBLOCK, &set, NULL);
      execvp(kArguments[0], kArguments);
    }
    sleep(1); // force parent off CPU
    printf("Job %d added to job list.\n", pid);
    sigprocmask(SIG_UNBLOCK, &set, NULL);
  }
  return 0;
}
```

myth60$ ./job-list-synchronization-improved
Sun Jan 27 05:16:54 PDT 2019
Job 3522 added to job list.
Job 3522 removed from job list.
Sun Jan 27 05:16:55 PDT 2019
Job 3524 added to job list.
Job 3524 removed from job list.
Sun Jan 27 05:16:56 PDT 2019
Job 3527 added to job list.
Job 3527 removed from job list.
myth60$ ./job-list-synchronization-improved
Sun Jan 27 05:17:15 PDT 2019
Job 4677 added to job list.
Job 4677 removed from job list.
Sun Jan 27 05:17:16 PDT 2019
Job 4691 added to job list.
Job 4691 removed from job list.
Sun Jan 27 05:17:17 PDT 2019
Job 4692 added to job list.
Job 4692 removed from job list.
myth60$
The program on the previous page addresses all of our concurrency concerns.

- **reapProcesses** is the same as before so it isn’t shown.
- The updated parent programmatically defers its obligation to handle signals until it returns from its `printf`—that is, it's added the pid to the job list.
- As it turns out, a **forked** process inherits blocked signal sets, so it needs to lift the block via its own call to `sigprocmask(SIG_UNBLOCK, ...)`). While it doesn't matter for this example (`date` almost certainly doesn't spawn its own children or rely on `SIGCHLD` signals), other executables may very well rely on `SIGCHLD`, as signal blocks are retained even across `execvp` boundaries.
- In general, you want the stretch of time that signals are blocked to be as narrow as possible since you're overriding default signal handling behavior and want to do that as infrequently as possible.
The job list examples highlight a key issue that comes with the introduction of signals and signal handling.

- Neither `job-list-synchronization` nor `job-list-synchronization-improved` can anticipate when a child process will finish up. That means it has no control over when `SIGCHLD` signals arrive.
- Processes do, however, have some control over how they respond to `SIGCHLD` signals.
  - They install custom `SIGCHLD` handlers to surface information about what process exited. We've seen a lot of that already.
  - When a process *elects* to use signal handling, it shouldn't be penalized by having to live with the concurrency issue that come with it. That would only encourage programmers to avoid signals and signal handling, even when it's the best thing to do.
  - That's why the kernel provides the option to defer a signal handler to run only when it can't cause problems. That's what our `job-list-fixed` program does.
  - It's true that the program could abuse the power to block signals for longer than necessary, but we have no choice but to assume the program wants to use signal handlers properly, else they wouldn't be installing them in the first place.
The kill Signal

Signal extras: **kill** and **raise**

- Processes can message other processes using signals via the **kill** system call.
- And processes can even send themselves signals using **raise**.

```c
int kill(pid_t pid, int signum);
int raise(int signum); // equivalent to kill(getpid(), signum);
```

- The **kill** system call is analogous to the `/bin/kill` shell command.
  - It is unfortunately named, since **kill** implies **SIGKILL** implies death. It got its name because the default action of most signals in early UNIX implementations was to just terminate the target process.
- We generally ignore the return value of **kill** and **raise**. Just make sure you call it properly.
- The **pid** parameter is overloaded to provide more flexible signaling.
  - When **pid** is a positive number, the target is the process with that pid.
  - When **pid** is a negative number less than -1, the targets are all processes within the process group `abs(pid)`. We'll rely on this in Assignment 3.
  - **pid** can also be 0 or -1, but you don't need to know this.
End of Lecture 7

WHAT IF I TOLD YOU
WE CAN KILL THE PID