Lecture 06: Signals and Signal Handlers

- Introduction to 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 set to execute in response to the arrival and consumption of a signal.
  - You're already familiar with several 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, **`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). In fact, C **`#define`**s **`SIGSEGV`** to be the number 11.

Lecture 06: Signals and Signal Handlers

- 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 prints a message identifying the pipe error and terminates the program.

Lecture 06: Signals and Signal Handlers

- There's one more signal category that's 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's **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, including exiting or crashing.
    - 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 culls the zombie process the relevant child has become.

Lecture 06: Signals and Signal Handlers

- Our first signal handler example: Disneyland
  - Here's an example that illustrates how to implement and install a **`SIGCHLD`** handler.
  - The premise? Dad takes his five kids out to play. Each of the five children plays for a different length of time. When all five kids are done playing, the six of them all go home.
  - The parent process is modeling dad, and the five child processes are modeling his children. (Full program, with error checking, is right here.)

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

int main(int argc, char *argv[]) {
  printf("Let my five children play while I take a nap.\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;
    }
  }
```

Lecture 06: Signals and Signal Handlers

- Our first signal handler example: Disneyland
  - The program is crafted so each child process exits at three-second intervals. **`reapChild`**, of course, 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.

```
  // code below is a continuation of that presented on the previous slide
  while (numDone < kNumChildren) {
    printf("At least one child still playing, so dad nods off.\n");
    sleep(5);
    printf("Dad wakes up! ");
  }
  printf("All children accounted for.  Good job, dad!\n");
  return 0;
}

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

Lecture 06: Signals and Signal Handlers

- Here's the output of the above program.
  - Dad's wakeup 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.
  - Understand that the `SIGCHLD` handler is invoked 5 times, each in response to some child process finishing up.

```
poohbear@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!
poohbear@myth60$
```

Lecture 06: Signals and Signal Handlers

- Advancing our understanding of signal delivery and handling.
    - Now consider the scenario where the five kids are the same age and run about Disneyland for the same amount of time. Restated, **`sleep(3 * kid)`** is now **`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 **`numDone`** global never gets big enough to match **`kNumChildren`**. Why is that? The next slide explains why!

```
poohbear@myth60$ ./broken-pentuplets
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 #3 done playing... runs back to dad.
Kid #4 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.
^C # I needed to hit ctrl-c to kill the program that loops forever!
poohbear@myth60$
```

Lecture 06: Signals and Signal Handlers

- 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 arrive 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 at one or more `SIGCHLD`s 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:

```
static void reapChild(int unused) {
  while (true) {
    pid_t pid = waitpid(-1, NULL, 0);
    if (pid < 0) break;
    numDone++;
  }
}
```

Lecture 06: Signals and Signal Handlers

- Advancing our understanding of signal delivery and handling.
  - 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, as with:

```
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++;
  }
}
```

Lecture 06: Signals and Signal Handlers

- 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 surface information about a child process that has either ended or been stopped.
  - `WCONTINUED` informs `waitpid` to surface information about a child process that 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 in any way (i.e. exited, crashed, stopped, or continued) but to do so without blocking.