Lecture 07: Masking Signals and Deferring Handlers

- Synchronization, multi-processing, 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 creep in unless you code very, very 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 execute handlers.
  - Consider the program on the next slide, which is a nod to the type of code you'll write for Assignment 4. The full program, with error checking, is right here):
    - The program spawns off three child processes at one-second intervals.
    - 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.

Lecture 07: Masking Signals and Deferring Handlers

- Here is the program itself on the left, and some test runs on the right.

```
// job-list-broken.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);
  }
}

const char *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;
}
```

```
myth60$ ./job-list-broken
Tue Apr 21 22:16:41 PDT 2020
Job 481 removed from job list.
Job 481 added to job list.
Tue Apr 21 22:16:42 PDT 2020
Job 482 removed from job list.
Job 482 added to job list.
Tue Apr 21 22:16:43 PDT 2020
Job 484 removed from job list.
Job 484 added to job list.
myth60$ ./job-list-broken
Tue Apr 21 22:17:03 PDT 2020
Job 503 removed from job list.
Job 503 added to job list.
Tue Apr 21 22:17:04 PDT 2020
Job 505 removed from job list.
Job 505 added to job list.
Tue Apr 21 22:17:05 PDT 2020
Job 506 removed from job list.
Job 506 added to job list.
$myth60$
```

Lecture 07: Masking Signals and Deferring Handlers

- Even with a program this simple, there are implementation issues to be addressed.
  - 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 being **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 publish the date and time to `stdout`.
  - But even if `sleep(1)` were 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 and sensible to do so. Restated, we'd like to postpone `reapProcesses` from executing until the parent's `printf` has returned.

```
myth60$ ./job-list-broken
Tue Apr 21 22:16:41 PDT 2020
Job 481 removed from job list.
Job 481 added to job list.
Tue Apr 21 22:16:42 PDT 2020
Job 482 removed from job list.
Job 482 added to job list.
Tue Apr 21 22:16:43 PDT 2020
Job 484 removed from job list.
Job 484 added to job list.
myth60$ ./job-list-broken
Tue Apr 21 22:17:03 PDT 2020
Job 503 removed from job list.
Job 503 added to job list.
Tue Apr 21 22:17:04 PDT 2020
Job 505 removed from job list.
Job 505 added to job list.
Tue Apr 21 22:17:05 PDT 2020
Job 506 removed from job list.
Job 506 added to job list.
$myth60$
```

Lecture 07: Masking Signals and Deferring Handlers

- The kernel provides directives that allow a process to temporarily ignore signal delivery.
- The subset of directives that interest us are presented below:

```
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—used as a bit vector of length 32. Since there are just under 32 signal types, the presence or absence of **`signum`**s 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 already being blocked. The third argument is the location of a **`sigset_t`** that should be updated with the set of signals being blocked at the time of the call. Again, we generally ignore the return value.

Lecture 07: Masking Signals and Deferring Handlers

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

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

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

```
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 we don't care to hear about what signals were being blocked before the call.

Lecture 07: Masking Signals and Deferring Handlers

- Here's an improved version of the job list program from earlier. (Full program here.)

```
// job-list-fixed.c
const char *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-fixed
Tue Apr 21 22:22:57 PDT 2020
Job 23574 added to job list.
Job 23574 removed from job list.
Tue Apr 21 22:22:58 PDT 2020
Job 23575 added to job list.
Job 23575 removed from job list.
Tue Apr 21 22:22:59 PDT 2020
Job 23577 added to job list.
Job 23577 removed from job list.
myth60$ ./job-list-fixed
Tue Apr 21 22:23:15 PDT 2020
Job 23594 added to job list.
Job 23594 removed from job list.
Tue Apr 21 22:23:16 PDT 2020
Job 23595 added to job list.
Job 23595 removed from job list.
Tue Apr 21 22:23:17 PDT 2020
Job 23597 added to job list.
Job 23597 removed from job list.
$myth60$
```

Lecture 07: Masking Signals and Deferring Handlers

- The program on the previous page addresses all of our concurrency concerns.
  - The implementation of `reapProcesses` is the same as before, so I didn't reproduce it.
  - 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 `fork`ed 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 short as possible, since you're overriding default signal handling behavior and want to do that as infrequently as possible.

```
myth60$ ./job-list-fixed
Tue Apr 21 22:22:57 PDT 2020
Job 23574 added to job list.
Job 23574 removed from job list.
Tue Apr 21 22:22:58 PDT 2020
Job 23575 added to job list.
Job 23575 removed from job list.
Tue Apr 21 22:22:59 PDT 2020
Job 23577 added to job list.
Job 23577 removed from job list.
myth60$ ./job-list-fixed
Tue Apr 21 22:23:15 PDT 2020
Job 23594 added to job list.
Job 23594 removed from job list.
Tue Apr 21 22:23:16 PDT 2020
Job 23595 added to job list.
Job 23595 removed from job list.
Tue Apr 21 22:23:17 PDT 2020
Job 23597 added to job list.
Job 23597 removed from job list.
$myth60$
```

Lecture 07: Masking Signals and Deferring Handlers

- 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`.

```
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.
    - Unfortunately named, since `kill` implies `SIGKILL` implies death.
    - So named, 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)`. You'll learn about how to group processes as part of Assignment 4.
    - `pid` can also be 0 or -1, but we don't need to worry about those. Type `man 2 kill` at the terminal if you're curious.