Lecture 07: Signals

"The barman asks what the first one wants, two race conditions walk into a bar."

Principles of Computer Systems

Spring 2019

Stanford University

Computer Science Department

Lecturer: Chris Gregg

PDF of this presentation

Lecture 07: Signals

- 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 that executes in response to the arrival and consumption of a signal. The signal handler *runs in the process that receives the 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). In fact, C **`#define`**s **`SIGSEGV`** to be the number 11.

Lecture 07: Signals

- 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 07: Signals

- One signal type most 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 right 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 culls the zombie the relevant child process has become.

Lecture 07: Signals

- Our first signal handler example: Disneyland
  - Here's a carefully coded 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.)

```c
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 07: Signals

- 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.
  - 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");
    snooze(5); // our implementation -- does not wake up upon signal
    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 07: Signals

- 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.

```
cgregg@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!
cgregg@myth60$
```

Lecture 07: Signals

- 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 makes it clear dad never detects all five kids are present and accounted for, and the program runs forever because dad keeps going back to sleep.

```
cgregg*@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.
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!
cgregg@myth60$
```

Lecture 07: Signals

- 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 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 07: Signals

- 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);
    if (pid <= 0) break; // note the < is now a <=
    numDone++;
  }
}
```

Lecture 07: Signals

- 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.

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 will 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 handle signals.
  - 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 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.

Lecture 07: Masking Signals and Deferring Handlers

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

```c
// 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);
  }
}

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;
}
```

```
myth60$ ./job-list-broken
Sun Jan 27 03:57:30 PDT 2019
Job 27981 removed from job list.
Job 27981 added to job list.
Sun Jan 27 03:57:31 PDT 2019
Job 27982 removed from job list.
Job 27982 added to job list.
Sun Jan 27 03:57:32 PDT 2019
Job 27985 removed from job list.
Job 27985 added to job list.
myth60$ ./job-list-broken
Sun Jan 27 03:59:33 PDT 2019
Job 28380 removed from job list.
Job 28380 added to job list.
Sun Jan 27 03:59:34 PDT 2019
Job 28381 removed from job list.
Job 28381 added to job list.
Sun Jan 27 03:59:35 PDT 2019
Job 28382 removed from job list.
Job 28382 added to job list.
myth60$
```

Lecture 07: Masking Signals and Deferring Handlers

- Even with a program this simple, there are implementation issues that need 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 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.

```
myth60$ ./job-list-broken
Sun Jan 27 03:57:30 PDT 2019
Job 27981 removed from job list.
Job 27981 added to job list.
Sun Jan 27 03:57:31 PDT 2019
Job 27982 removed from job list.
Job 27982 added to job list.
Sun Jan 27 03:57:32 PDT 2019
Job 27985 removed from job list.
Job 27985 added to job list.
myth60$ ./job-list-broken
Sun Jan 27 03:59:33 PDT 2019
Job 28380 removed from job list.
Job 28380 added to job list.
Sun Jan 27 03:59:34 PDT 2019
Job 28381 removed from job list.
Job 28381 added to job list.
Sun Jan 27 03:59:35 PDT 2019
Job 28382 removed from job list.
Job 28382 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—that's 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 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 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 I 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.)

```c
// job-list-fixed.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-fixed
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-fixed
Sun Jan 27 05:17:15 PDT 2018
Job 4677 added to job list.
Job 4677 removed from job list.
Sun Jan 27 05:17:16 PDT 2018
Job 4691 added to job list.
Job 4691 removed from job list.
Sun Jan 27 05:17:17 PDT 2018
Job 4692 added to job list.
Job 4692 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 narrow as possible, since you're overriding default signal handling behavior and want to do that as infrequently as possible.

```
myth60$ ./job-list-fixed
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-fixed
Sun Jan 27 05:17:15 PDT 2018
Job 4677 added to job list.
Job 4677 removed from job list.
Sun Jan 27 05:17:16 PDT 2018
Job 4691 added to job list.
Job 4691 removed from job list.
Sun Jan 27 05:17:17 PDT 2018
Job 4692 added to job list.
Job 4692 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)`**. We'll rely on this in Assignment 4.
    - **`pid`** can also be 0 or -1, but we don't need to worry about those. See the man page for **`kill`** if you're curious.