CS110: Principles of Computer Systems

Autumn 2021
Jerry Cain
PDF

Lecture 10: Introduction to 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 generally sent by the kernel, but they can be sent from other processes as well).
- 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 (or intentionally, if that's who you are) dereferenced a **NULL** pointer.
  - When that happens, the kernel delivers a **SIGSEGV** signal of type **SIGSEGV**, informally known as a segfault.
- Each signal category (e.g. **SIGSEGV**) is represented internally by some number (e.g. 11). For example, C # **define**s **SIGSEGV** to be the number 11.

- 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.

Lecture 10: Introduction to 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 fires a **SIGFPE** signal at the offending process. By default, the program responds to the **SIGFPE** by immediately terminating the program.
  - 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 simply terminates the program.
  - When a process tries to execute an illegal assembly code instruction, the kernel issues a **SIGILL** instruction to that process (and the program terminates).

process control related

  - When you type CTRL-C from the terminal, the kernel sends a **SIGINT** to the foreground process. By default, that foreground process is terminated.
  - When you type CTRL-Z, the kernel send a **SIGTSTP** to the foreground process (and by default, the foreground process is halted until a subsequent **SIGCONT** signal instructs it to continue).

timing related

  - A user program can set a timer using a system call like **setitimer**. When the timer goes off, the kernel sends a **SIGALRM** to the process.

Lecture 10: Introduction to Signals

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

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

Lecture 10: Introduction to Signals

- **Signal handlers** are ordinary functions we ask the operating system to run whenever a signal comes in. The **signal** function installs a signal handler with the intent that the OS execute that signal handler to, well, handler, the signal.
  - Some signals are handled **synchronously**, which means any installed signal handler is executed **immediately**, the instant the signal is generated. These types of signals generally result from the process doing something bad to itself (e.g. dividing by zero, or dereferencing a bad address, or executing an illegal instruction). These type of signals are often called **traps**.
  - Other signals are handled **asynchronously**, which means they were generated outside of the process (e.g. you typed CTRL-C to generate a **SIGINT**, or a previously established timer expired to generate a **SIGALRM**). The generation of these signals often happens while the target process is off the CPU, so the signal handler is executed as soon as gets the CPU again. These types of asynchronously handled signals are often called **interrupts**.
  - Programs can very rarely recover from traps, though there certainly can recover from interrupts. That's why the default handler for most traps is to terminate the process and the default handler for most interrupts is something else.
- Signal handlers take a single argument, which is used to specify which signal it is handling.

Lecture 10: Introduction to Signals

- Here are two very small examples we'll tinker with during lecture.
  - This first one installs a signal handler to execute code that results from an intentional **NULL** pointer dereference. Note that our signal handler takes the opportunity to print a custom message ahead of termination.

```
 1 static void handleSIGSEGV(int sig) {
 2   assert(sig == SIGSEGV);
 3   cout << "There's no recovering from this." << endl;
 4   exit(139); // as per https://www.geeksforgeeks.org/exit-codes-in-c-c-with-examples
 5 } // SIGSEGV handlers should still end the program.
 6
 7 int main(int argc, char *argv[]) {
 8   signal(SIGSEGV, handleSIGSEGV);
 9   *(int *)NULL = 110;
10   return 0;
11 }
```

  - The second one installed a custom handler to be executed in response to CTRL-C.

```
 1 static void handleSIGINT(int sig) {
 2   assert(sig == SIGINT);
 3   cout << "I'm ignoring you." << endl;
 4 } // By the way, Doris is over Thor.  She likes Brutus now.
 5
 6 int main(int argc, char *argv[]) {
 7   signal(SIGINT, handleSIGINT);
 8   cout << "Just try to interrupt me, Thor!" << endl;
 9   for (size_t i = 0; i < 50; i++) { sleep(1); }
10   return 0;
11 }
```

Lecture 10: Introduction to Signals

- One asynchronous signal type important to **multiprocessing**? **SIGCHLD**
  - Whenever a 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 all **fork** examples until now and we've gotten away with it.
  - Doing so allows forked child processes to run in the background while allowing the parent to do its own meaningful work without blocking in some **waitpid** call.
  - The parent process, however, is still required to reap child process resources, so a parent will typically register a custom **SIGCHLD** handler—using that **signal** function we just learn about—to be asynchronously invoked whenever a child process changes state.
  - These custom **SIGCHLD** handlers almost always include calls to **waitpid** to surface the pids of child processes that've changed state. If the child process of interest actually terminated or crashed, the **waitpid** also reaps the now-zombie process's resources (e.g. its process control block).
  - One key takeaway from that: **waitpid** calls aren't really optional.

Lecture 10: Introduction to Signals

- Our first signal handler example: Disneyland, Take I
  - 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 others model his children.

```
static const size_t kNumChildren = 5;  // constant
static size_t numDone = 0;             // global variable!

int main(int argc, char *argv[]) {
  cout << "Let my five children play while I take a nap." << endl;
  signal(SIGCHLD, reapChildProcesses);
  for (size_t kid = 1; kid <= kNumChildren; kid++) {
    pid_t pid = fork();
    if (pid == 0) {
      sleep(3 * kid); // sleep emulates "play" time
      cout << "Child " << kid << " tires... returns to dad." << endl;
      return 0;
    }
  }

    while (numDone < kNumChildren) {
    cout << "At least one child still playing, so dad nods off." << endl;
    snooze(5); // signal-safe version of sleep
    cout << "Dad wakes up! ";
  }

  cout << "All children accounted for.  Good job, dad!" << endl;
  return 0;
}
```

Lecture 10: Introduction to Signals

- Our first signal handler example: Disneyland, Take I
  - Each child process exits at three-second intervals. **reapChildProcesses** 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. That means that the implementation of **reapChildProcesses**—that's our signal handler here—needs to coordinate with the main execution flow via the **numDone** global variable**.**

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

Lecture 10: Introduction to Signals

- Our first signal handler example: Disneyland, Take I
  - Here's the output of the previous program.

```
poohbear@myth63:$ ./five-children
Let my five children play while I take a nap.
At least one child still playing, so dad nods off.
Child 1 tires... returns to dad.
Dad wakes up! At least one child still playing, so dad nods off.
Child 2 tires... returns to dad.
Child 3 tires... returns to dad.
Dad wakes up! At least one child still playing, so dad nods off.
Child 4 tires... returns to dad.
Child 5 tires... returns to dad.
Dad wakes up! At least one child still playing, so dad nods off.
Dad wakes up! All children accounted for.  Good job, dad!
poohbear@myth63:$
```

  - 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 above reflects that.
  - The **SIGCHLD** handler is invoked 5 times, each in response to some isolated child process finishing up.

Lecture 10: Introduction to Signals

- Let's advance our understanding of **SIGCHLD** 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**. So, if three **SIGCHLD** signals are delivered while dad is off the processor, the operating system only records the fact that one or more **SIGCHLD**s came in! Restated, the OS maintains a Boolean, not a count.
  - When the parent process executed its **SIGCHLD** handler, it must do so on behalf of the **one or more signals** that may have been delivered.
  - That means our **SIGCHLD** handler needs to call **waitpid** in a loop, as with:

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

Lecture 10: Introduction to Signals

- Let's advance our understanding of **SIGCHLD** delivery and handling even more!
  - The improved **reapChildProcesses** implementation seemingly fixes the **confused-pentuplets** program, but it changes the behavior of our original **five-children** program. Now why is that?
    - When the first child in **five-children** exits, the other children are still playing.
    - The **SIGCHLD** handler calls **waitpid**, and it returns 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 manage this by including **WNOHANG** in the third parameter passed to **waitpid**.

```
static void reapChildProcesses(int unused) {
  while (true) {
    pid_t pid = waitpid(-1, NULL, WNOHANG); // check out the WNOHANG!!!
    if (pid <= 0) {  // note the < is now a <=
      assert(pid == 0 || errno == ECHILD); // pid could be 0 now
      break;
    }
    numDone++;
  }
```

  - If the calling process has no more children, then **waitpid** returns -1 and sets **errno** to **ECHILD.**

Lecture 10: Introduction to 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 generally means that there are no child processes left.
  - A return value of 0—that's a new possible return value as of the last slide—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** in argument 3.
- Recall the third argument supplied to **waitpid** can include several flags bitwise-OR'ed together. That collection of flags now includes **WNOHANG**.
  - **WUNTRACED** informs **waitpid** to block until some child process has either ended or stopped.
  - **WCONTINUED** informs **waitpid** to block until some child process has either ended or been continued 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 10: Introduction to Signals

- Signal Aside: Software-supplied signals: **kill** and **raise**
  - Processes can message other processes using signals via the **kill** system call. Processes can even send themselves signals using **raise**.

```
int kill(pid_t pid, int sig);
```

```
int raise(int sig);
```

    - **raise(sig)** is programmatically identical to **kill(getpid(), sig)**
  - The **kill** system call is analogous to the **/bin/kill** shell command.
    - It is unfortunately named, since kill implies SIGKILL implies death, and that's not cute. It got its name because the default response to **most** signals in early UNIX implementations was to just terminate the target process.
    - We generally ignore the return values of **kill** and **raise**. Just 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 never rely on those two possibilities in CS110.

Lecture 10: Introduction to Signals

- Here's a playful program where parent and child play ping pong using **kill** and **raise**.

Lecture 10: Introduction to Signals

This slide was written by Ryan Eberhardt and edited by Jerry.

- Sadly, asynchronous signal handling has its drawback!
  - Signal handlers are difficult to use properly, and the consequences can be severe. Many regard signals to be one of the worst parts of Unix's design.
    - This installment of Ghosts of Unix Past explains why asynchronous signal handling can be such a headache.
    - The article's primary point: The trouble with signal handlers is that they can be invoked at a really, really bad time (e.g. while the main execution flow is in the middle of a **malloc** call, or accessing a complex data structure).
  - Here's a short program illustrating the dangers.

```
 1 static vector<string> strings;
 2
 3 void handleSIGINT(int sig) {
 4   stringsToProcess.clear();
 5 }
 6
 7 int main(int argc, char *argv[]) {
 8   buildStringVector(strings); // assume anything reasonable
 9   signal(SIGINT, handleSIGINT);
10   for (string &s: strings) processString(s);
11   return 0;
12 }
```

  - What if I type CTRL-C from the terminal while main execution is in the middle of that for loop? Whatever happens, it will bring sadness.