CS110: Principles of Computer Systems

Autumn 2021
Jerry Cain
PDF

Lecture 11: Introduction to Signals

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

- The previous lecture formally introduced the signal and the signal handler.
  - **Signals** are broad-brushstroke messages sent to a process to announce that something happened that the process should hear about.
  - Signals are most often sent by the kernel, though other processes can forward signals to other processes as long as they have permission to do so. Processes can even send themselves signals, as with the **raise(SIGSTOP)** call in **assign3**'s **trace.cc**.
  - **Signals handlers** are functions designed to handle the arrival of a signal. The signal handler often probes the surrounding process or the OS to gather information about what actually happened.
    - Signals like **SIGSEGV** and **SIGFPE** are considered to be *synchronous* and generally sent because the recipient of the signal committed some code crime. Synchronous signal handlers are typically invoked immediately, within the same time slice, after the offending instruction is executed.
    - Signals like **SIGCHLD** and **SIGINT** are considered to be *asynchronous* and are typically sent because something external to the process occurred. Asynchronous signals handlers are generally invoked at the beginning of the recipient's next time slice—that is, the next time it's given processor time.
      - If the recipient is incidentally on the CPU when the signal arrives, it can be executed immediately, but it's typically deferred until its next time slice begins.

Lecture 11: Signals and Signal Handling, Take II

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

- Asynchronous signal handling has its shortcomings!
  - Signal handlers are difficult to use properly, and the consequences can be severe. Some 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.

Lecture 11: Signals and Signal Handling, Take II

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

- Signal handlers are particularly dangerous, because you can rarely predict when a signal will arrive and whether the handler will execute during a window when the surrounding memory space is in an inconsistent or incompatible state.
- The code below is farcically unsafe and could theoretically cause any one of several problems—segmentation fault and deadlock are easily explained, but in principle the code could erase your entire hard drive.

```
 1 void handleSIGINT(int sig) {
 2     printf("Got SIGINT!\n");
 3 }
 4
 5 int main(int argc, char **argv[]) {
 6     signal(SIGINT, handleSIGINT);
 7     while (true) {
 8         printf("Sleeping...\n");
 9         sleep(1);
10     }
11 }
```

```
// vsprintf is almost 1800 lines long,
// and line 1311 makes is async-unsafe
1309   /* Lock stream.  */
1310   _IO_cleanup_region_start ((void_star_func) &_IO_funlockfile, s);
1311   _IO_flockfile (s);
```

- The code **looks** harmless enough, but the deadlock and segfault scenarios are more immediately apparent if you understand that **printf** calls **vsprintf**, and **vsprintf** is not *reentrant*.

Lecture 11: Signals and Signal Handling, Take II

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

- Previous iterations of CS110 have emphasized asynchronous signal handling and the various techniques that can be used to make it safe (or rather, less unsafe).
- If we're truly model signal handler citizens, our signal handler implementations should limit themselves to only rely on *signal-handler-safe* functions, a list of which is presented here.
  - That list is actually fairly long, but most of the functions listed are system calls. There are very few **libc** (and no **stdlibc++**) functions in that list.

- This quarter, we're going to circumvent the unpredictability and the cases that come with signal handlers and take a new approach.
- We'll instead handle all asynchronous signals of interest within the main flow of execution, without using the **signal** function.
- In general, we will not monitor synchronous signals like **SIGSEGV**, **SIGFPE**, **SIGBUS**, and **SIGILL**, primarily because
  - there's little one can do in response to these particular signals, because the program is likely ending no matter what we do, and
  - the default signal handlers fundamentally do the correct thing anyway, and they do so using only signal-handler-safe functions

Lecture 11: Signals and Signal Handling, Take II

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

- We'll follow these three steps when dealing with signals:
  - First, build a set of signals to include precisely those you're interested in monitoring, as with:

```
sigset_t monitoredSignals;
sigemptyset(&monitoredSignals);
sigaddset(&monitoredSignals, SIGINT);
sigaddset(&monitoredSignals, SIGTSTP);
```

    - The **sigset_t** is a data type designed to model a set of signals. It's really just a 32-bit **int** that's not so much a number as it is an array of 32 Booleans.
      - Our flavor of Linux supports less than 32 signals, and each of them is backed by some number between 0 and 31, inclusive.
      - **sigset_t** membership of a particular signal amounts to whether a dedicated bit is a 0 or a 1.
      - The **sigemptyset** function is vital here, because it zeroes out the entire set to leave you with the empty signal set. Forget this, and you'll inherit a random set of signals and be sad.
      - The **sigaddset** function works as you might expect: It updates the supplied **sigset_t** to include the specific signal if it wasn't already present.
    - **monitoredSignals** can travel through the **main** execution flow and always represent the set of signals we're paying attention to.

Lecture 11: Signals and Signal Handling, Take II

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

- We'll follow these three steps when dealing with signals:
  - First, build a set of signals to include precisely those you're interested in monitoring, as with:

```
sigset_t monitoredSignals;
sigemptyset(&monitoredSignals);
sigaddset(&monitoredSignals, SIGINT);
sigaddset(&monitoredSignals, SIGTSTP);
```

  - Second, inform the OS to suspend the delivery of these signals until further notice.

```
sigprocmask(SIG_BLOCK, &monitoredSignals, NULL);
```

    - We don't want any of the signals we're monitoring to invoke any built-in behavior (e.g. stopping the program when CTRL-Z is pressed). The OS still compiles a list of any signals that have arrived, but it won't act on them. Such signals are called **deferred** or **pending**.
    - **sigprocmask** is short for **sig**nal **proc**ess **mask**, and it's used here to suppress the delivery of (i.e. "block") any signals in the monitored set until further notice.
      - **SIG_BLOCK** formally states we'd like to add the monitored signal set to any others already being suppressed. **SIG_UNBLOCK** subtracts the monitored signal set from the list being suppressed.
      - The third argument can be used to collect the set of signals being blocked prior to the **sigprocmask** call. Here, we pass in **NULL** as a statement that we don't need that information.

Lecture 11: Signals and Signal Handling, Take II

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

- We'll follow these three steps when dealing with signals:
  - First, build a set of signals to include precisely those you're interested in monitoring, as with:

```
sigset_t monitoredSignals;
sigemptyset(&monitoredSignals);
sigaddset(&monitoredSignals, SIGINT);
sigaddset(&monitoredSignals, SIGTSTP);
```

  - Second, inform the OS to suspend the delivery of these signals until further notice.

```
sigprocmask(SIG_BLOCK, &monitoredSignals, NULL);
```

  - Finally, call **sigwait**, which is prepared to halt program execution until one of the signals being monitored arrives (although it returns immediately if one is pending)

```
int delivered;
sigwait(&monitoredSet, &delivered);
cout << "Received signal: " << delivered << endl;
```

    - Whether a monitored signal arrives immediately or eventually, the signal is placed in the space whose location is shared via **sigwait**'s second argument.
    - Once **sigwait** returns and advertises what signal surfaced, you can process that signal *inline*—that is, synchronously—and avoid the ill-defined consequences of asynchronous signal handlers.

Lecture 11: Signals and Signal Handling, Take II

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

- We'll follow these three steps when dealing with signals:
  - First, build a set of signals to include precisely those you're interested in monitoring, as with:

```
sigset_t monitoredSignals;
sigemptyset(&monitoredSignals);
sigaddset(&monitoredSignals, SIGINT);
sigaddset(&monitoredSignals, SIGTSTP);
```

  - Second, inform the OS to suspend the delivery of these signals until further notice.

```
sigprocmask(SIG_BLOCK, &monitoredSignals, NULL);
```

  - Finally, call **sigwait**, which is prepared to halt program execution until one of the signals being monitored arrives (although it returns immediately if one is pending)

```
int delivered;
sigwait(&monitoredSet, &delivered);
cout << "Received signal: " << delivered << endl;
```

  - Want to see this work? Check out this cplayground.
    - The default behavior of **CTRL-C** and **CTRL-Z** are suppressed and effectively overridden, because they're monitored by the program and they're never permitted to activate the default **SIGINT** and **SIGTSTP** handlers.
    - Still want to end the program? Type **CTRL-**. **SIGQUIT** isn't being monitored or suppressed, so its default handler executes to terminate the program.

Lecture 11: Signals and Signal Handling, Take II

- Presented over the next several slides is a substantial reorganization of the **five-children** example we covered last time.
  - Remember that the parent process models a dad who takes his kids to Disneyland so he can take naps, which presumably he can't do at home.
  - Dad's five children run unsupervised throughout Disneyland until they tire out, at which point they return to papa, who's sleeping on a bench by the entrance.
  - Dad wakes up every so often—or rather, every five seconds—to poll for children. When he counts all five, the whole family goes home.

- Our new approach to monitoring and synchronously handling signals requires the parent process pay attention to two signal types:
  1. **SIGCHLD**: because the process modeling dad only learns one or more child processes has finished because the OS delivers a **SIGCHLD**.
  2. **SIGALRM**: because dad sleeps in five seconds intervals until five kids show up. Inlining **snooze(5)** calls into a **while** loop around a **sigwait** call would interfere with the timely bookkeeping we want if we're to track the number of child processes that've finished, so we instead rely on timers to fire **SIGALRM**s at five-second intervals.

Lecture 11: Signals and Signal Handling, Take II

- Here's the same Disneyland example using this new programming model:

```
 1 static const size_t kNumChildren = 5;
 2 static void constructMonitoredSet(sigset_t& monitored, const vector<int>& signals) {
 3   sigemptyset(&monitored);
 4   for (int signal: signals) sigaddset(&monitored, signal);
 5 }
 6
 7 static void blockMonitoredSet(const sigset_t& monitored) {
 8   sigprocmask(SIG_BLOCK, &monitored, NULL);
 9 }
10
11 static void unblockMonitoredSet(const sigset_t& monitored) {
12   sigprocmask(SIG_UNBLOCK, &monitored, NULL);
13 }
14
15 int main(int argc, char *argv[]) {
16   cout << "Let my five children play while I take a nap." << endl;
17   sigset_t monitored;
18   constructMonitoredSet(monitored, {SIGCHLD, SIGALRM});
19   blockMonitoredSet(monitored);
20   for (size_t kid = 1; kid <= kNumChildren; kid++) {
21     pid_t pid = fork();
22     if (pid == 0) {
23       unblockMonitoredSet(monitored); // lift block on signals, child may rely on them
24       sleep(3 * kid); // sleep emulates "play" time
25       cout << "Child " << kid << " tires... returns to dad." << endl;
26       return 0;
27     }
28   }
29
```

Lecture 11: Signals and Signal Handling, Take II

- And here's the rest of **main**:

```
 1   size_t numDone = 0;
 2   bool dadSeesEveryone = false;
 3   letDadSleep();
 4   while (!dadSeesEveryone) {
 5     int delivered;
 6     sigwait(&monitored, &delivered);
 7     switch (delivered) {
 8     case SIGCHLD:
 9       numDone = reapChildProcesses(numDone);
10       break;
11     case SIGALRM:
12       wakeUpDad(numDone);
13       dadSeesEveryone = numDone == kNumChildren;
14       break;
15     }
16   }
17
18   cout << "All children accounted for.  Good job, dad!" << endl;
19   return 0;
20 }
```

```
1 static size_t reapChildProcesses(size_t numDone)
2   while (true) {
3     pid_t pid = waitpid(-1, NULL, WNOHANG);
4     if (pid <= 0) break;
5     numDone++;
6   }
7   return numDone;
8 }
```

- Note we keep looping and **sigwait**ing until we're confident dad has counted to five.
- With each iteration, we decide why **sigwait** returned and then dispatch accordingly.
- **reapChildProcesses** operates much like it did last lecture, but here it's executed synchronously. All code is executed asynchronously, so we don't need any globals.
- We've yet to implement **letDadSleep** and **wakeUpDad**, but it's reasonable to expect that the first schedules a **SIGALRM** and the second responds to one.

Lecture 11: Signals and Signal Handling, Take II

- Here are the three outstanding functions of content that need to be discussed:

```
 1 static void setAlarm(double duration) { // fire SIGALRM 'duration' seconds from now
 2   int seconds = int(duration);
 3   int microseconds = 1000000 * (duration - seconds);
 4   struct itimerval next = {{0, 0}, {seconds, microseconds}};
 5   setitimer(ITIMER_REAL, &next, NULL);
 6 }
 7
 8 static const double kSleepTime = 5.0;
 9 static void letDadSleep() {
10   cout << "At least one child still playing, so dad nods off." << endl;
11   setAlarm(kSleepTime);
12 }
13
14 static void wakeUpDad(size_t numDone) {
15   cout << "Dad wakes up and sees " << numDone
16        << " " << (numDone == 1 ? "child" : "children") << "." << endl;
17   if (numDone < kNumChildren) letDadSleep();
18 }
```

literal snooze button

  - **letDadSleep** is a wrapper around **setAlarm**, which itself relies on **setitimer** (short for **set i**nterval **t**imer) to ask that a **SIGALRM** be sent after **duration** seconds.
    - That **{0, 0}** is a sentinel telling **setitimer** to schedule a one-shot alarm instead of one repeatedly fired at regular intervals.
    - The **ITIMER_REAL** constant tells the OS to monitor the amount of wall clock time that's passing (as opposed to, say, user and/or system time).
  - Notice how dad lets himself sleep more if he wakes up to find less than 5 kids.

Lecture 11 Coda: Playing Classical Guitar

- Most Linux distributions include a command line utility called **play** that can be used to sound the pluck of a guitar string for a specified length of time.
  - The following line, for example, plays a middle C that lasts for 1.5 seconds:

```
myth61:$ play -qn synth 1.5 pluck C4
```

  - To play the same pitch one octave higher for 0.75 seconds, you'd invoke:

```
myth61:$ play -qn synth 0.75 pluck C5
```

  - Feeling jazzy? Here's the Bb7(#11) chord big bands blast at the end of many standards.

```
myth61:$ play -qn synth 3.00 pluck Bb2 & \
>        play -qn synth 3.00 pluck Ab3 & \
>        play -qn synth 3.00 pluck D4 & \
>        play -qn synth 3.10 pluck E4 & \
>        play -qn synth 3.10 pluck G4 & \
>        play -qn synth 3.10 pluck C5 &
```

  - The last token of **"play -qn synth 1.5 pluck C4"** specifies the pitch and octave and will always be some note drawn from the traditional Western music scale—e.g. C2, D2, E2, F2, G2, A2, B2, C3, although # and b can be appended to alter the pitch half a tone, as with C# or Bb.
  - Some notes last longer than others. The exact duration is dictated by the number that sits in between **"synth"** and **"pluck"**.

Lecture 11 Coda: Playing Classical Guitar

- For this example, we're going to create a CLI classical guitarist!
  - We'll model a classical guitar song as a **vector** of **note**s, sorted by **start** time.
  - By scheduling **fork** and **execvp** to invoke **play** for each note at the appropriate times, our program will be able to play an entire piece from start to finish.

```
struct note {
  string pitch;     // "A4", "Bb2" or some other pitch
  double start;     // the time from launch when note should play
  double duration;  // the time the note should last once played
}; // all times are in seconds
```

  - We'll rely on **SIGCHLD** and synchronously manage calls to **waitpid** to reap system resources as our play processes exit.
  - We'll also tap the same **SIGALRM** signal we used to wake Disneyland dad up.
    - The primary difference? Our timers will schedule **SIGALRM**'s to fire at varying times that, in general, won't be evenly spaced.
    - Each timer—we'll rely on the same exact **setAlarm** function we wrote for dad—will prompt one or more notes to be played for their due durations.
    - It's uncommon for a guitarist to play only one note at a time. More often, they pluck several strings to play multiple, musically compatible notes simultaneously.

Lecture 11 Coda: Playing Classical Guitar

- Fundamentally, our program needs to load a song into memory, maintain a cursor to separate what's been played from what hasn't, and then gracefully terminate when the performance has ended.
  - The song is initialized from a file formatted like the one you see on the right.
  - The first note always has an effective start time of 0.0.
  - Neighboring notes may have the same start time if they're all intended to be played together (even if their durations vary).

```
guitar.txt
C4  0.0 0.5
D4  0.5 0.5
B3  0.5 0.5
E4  1.0 0.5
C4  1.0 0.5
Bb3 1.0 0.5
G4  3.0 2.5
// many more notes
```

  - The fact that the second set of notes starts at t = 0.5 seconds means we'd call **setAlarm(0.5)** after spawning off a single child process for that first **C4**.
  - When the **SIGALRM** signal is fired, we know the time has come to spawn off two more child processes—one to play a D4, and a second to play a B3—before calling **setAlarm(0.5)** again to schedule some E4/C4/Bb3 chord.
    - In programming terms, the function we implement to synchronously handle any **SIGALRM**s will **fork** off new **play** processes and set additional timers.
  - Presented across the next several slides is the full program that plays an entire piece on classical guitar.

Lecture 11 Coda: Playing Classical Guitar

- Here's the **main** function and the core of the **playSong** function that decomposes it:

```
1  static void playSong(const vector<note>& song) {
2    size_t pos = 0;
3    set<pid_t> processes;
4    sigset_t monitored;
5    constructMonitoredSet(monitored, {SIGINT, SIGALRM, SIGCHLD}); // same as for Disneylan
6    blockMonitoredSet(monitored); // same as for Disneyland
7    raise(SIGALRM); // start the metronome at t = 0.0
8    while (pos < song.size() || !processes.empty()) {
9      int delivered;
10     sigwait(&monitored, &delivered);
11     switch (delivered) {
12     case SIGINT:
13       stopPlaying(processes);
14       pos = song.size(); // sentinel value stating that no more notes should be played
15       break;
16     case SIGALRM:
17       pos = playNextNotes(song, pos, monitored, processes);
18       break;
19     case SIGCHLD:
20       reapChildProcesses(processes);
21       break;
22     }
23   }
24 }
25
26 int main(int argc, char *argv[]) {
27   if (argc > 2) usage();
28   vector<note> song;
29   initializeSong(song, argv[1]); // we omit the implementation of this
30   playSong(song);
31   return 0;
32 }
```

Lecture 11 Coda: Playing Classical Guitar

- Here's the **main** function and the core of the **playSong** function that decomposes it:

```
1  static size_t playNextNotes(const vector<note>& song, size_t pos,
2                              const sigset_t& monitored, set<pid_t>& processes) {
3    double current = song[pos].start;
4    while (pos < song.size() && song[pos].start == current) {
5      pid_t pid = fork();
6      if (pid == 0) {
7        unblockMonitoredSet(monitored); // same as for Disneyland
8        string duration = to_string(song[pos].duration);
9        const char *argv[] = {
10         "play", "-qn", "synth", duration.c_str(), "pl", song[pos].pitch.c_str(), NULL
11       };
12       execvp(argv[0], (char **) argv); // assume succeeds
13     }
14     pos++;
15     processes.insert(pid);
16   }
17   if (pos < song.size()) setAlarm(song[pos].start - current); // same as for Disneyland
18   return pos;
19 }
20
21 static void stopPlaying(const set<pid_t>& processes) {
22   for (pid_t pid: processes) kill(pid, SIGKILL); // kill child processes
23   setAlarm(0); // setting at alarm for 0 seconds into the future really disables the ala
24 }
25
26 static void reapChildProcesses(set<pid_t>& processes) {
27   while (true) {
28     pid_t pid = waitpid(-1, NULL, WNOHANG);
29     if (pid <= 0) break;
30     processes.erase(pid);
31   }
32 }
```