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

Instructors: Chris Gregg and Phil Levis

PDF of this presentation

Lecture 07: Signals - revisiting Disneyland

- Last time, Phil discussed the Disneyland example, and discussed one issue with signal handling. To recap:
  - Five kids run about Disneyland for the same amount of time. In our code, they all **`sleep(3)`** so all five children finish at a almost the same time (they are running in parallel, remember).
  - The code is here, except **`sleep(3 * kid)`** has become **`sleep(3)`**
  - 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

- As Phil discussed, if multiple children finish 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`** and **`WNOHANG`**, in a loop, 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++;
  }
}
```

- Now, all of the children that have stopped get cleaned up by the **`waitpid`**, and there is no blocking.
- You might rightly ask -- what happens if a signal comes in *while the signal handler is running?*
- This is an important question! Let's take a few minutes to think about the consequences of a signal coming in *during* the **`reapChild`** function.

Lecture 07: Signals

```
1 static void reapChild(int unused) {
2   while (true) {
3     pid_t pid = waitpid(-1, NULL, WNOHANG);
4     if (pid <= 0) break;
5     numDone++;
6   }
7 }
```

- Let's assume that three children finish at almost exactly the same time, and the **`reapChild`** function gets called.
- The while loop gets executed three times, to clean up the three children that triggered **`SIGCHLD`** handler.
- But, what if another child finishes somewhere in the loop? The designers of Linux could have chosen to assume that **`waitpid`** will clean up after that one, as well. If that was the case, let's first look at what happens if the fourth child finishes after line 5 above.
    - The while loop will continue, and then **`waitpid`** will properly clean up after the fourth child. Yay!

Lecture 07: Signals

```
1 static void reapChild(int unused) {
2   while (true) {
3     pid_t pid = waitpid(-1, NULL, WNOHANG);
4     if (pid <= 0) break;
5     numDone++;
6   }
7 }
```

- But, what happens if the fourth child ends right after line 3?
  - In this case, the return value for **`waitpid`** will be 0 to indicate that there are still remaining children to finish (correct at that point in time), and then proceed to line 4, which will break out of the loop, and leave the function. The fourth child would not get cleaned up.
- If Linux assumed that the signal handler cleaned up fourth child, this would be a bug!
- So, it turns out that Linux made the following decision:
  - "*When the handler for a particular signal is invoked, that signal is automatically **blocked** until the handler returns. That means that if two signals of the same kind arrive close together, the second one will be held until the first has been handled.*" (bold mine)
  - The "close together" part is a bit confusing -- if two children finish *really* close together, only one signal will be generated (and must be cleaned up as above). If a further child ends, it will *generate a new signal* that will call the signal handler again, after the currently-running function ends.

Lecture 07: Signals

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

- To reiterate:
  - If one or more children end almost simultaneously, only one signal will be generated, and the children must be cleaned up in a loop, as above.
  - If another child ends while the signal handler above is running, another signal will be generated, and it will fire after the above function ends.
    - The function will run again, and then can properly clean up the child that just ended.
- As we saw, if the Linux designers had not done it this way, there could be a *race condition*, where we have behavior that we cannot rely on. Race conditions crop up *everywhere* when writing concurrent code (i.e., processes that run concurrently), so not only do we as programmers need to be aware of it, but we also have to understand how the system works so we know what we really can expect.

Lecture 07: Signals

- All **`SIGCHLD`** handlers generally have this **`while`** loop structure.
  - Call **`waitpid(-1, &status, WNOHANG)`**
  - 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 ("stopped" in this case really means "paused").
  - **`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.

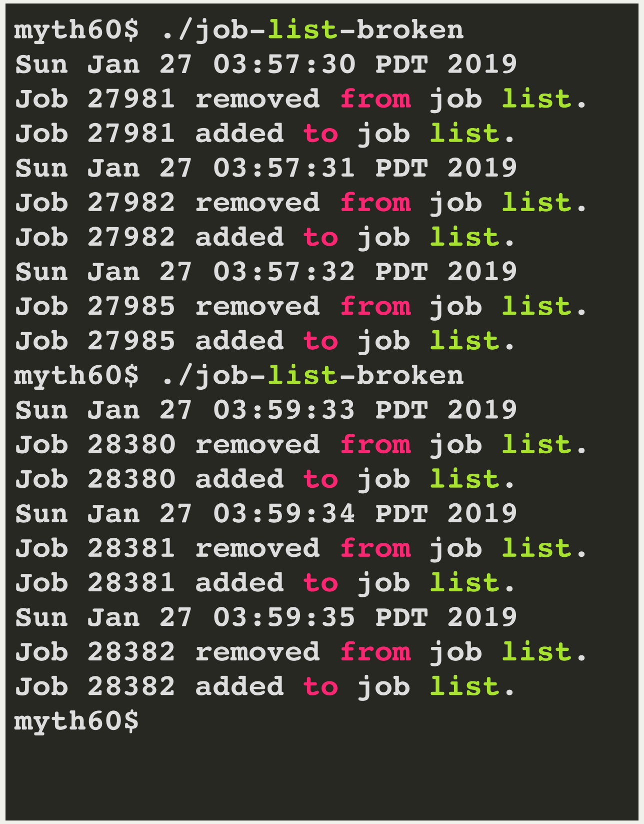
```c
// job-list-broken.c
static void reapProcesses(int sig) {
  while (true) {
    pid_t pid = waitpid(-1, NULL, WNOHANG);
    if (pid <= 0) break;
    printf("Job %d removed from job list.\n", pid);
  }
}

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 how, const sigset_t *set, sigset_t *oldset);
```

  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 **`how`** is set to **`SIG_BLOCK`**) or removes (if **`how`** is set to **`SIG_UNBLOCK`**) the signals reachable from **`set`** to/from the set of signals being ignored at the moment. **`oldset`** is the location of a **`sigset_t`** that can be updated with the set of signals being blocked at the time of the call, so you can restore it later if you need to.

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.

Lecture 07: A more sophisticated shell (but not quite right yet...)

- Last week, we looked at **`mysystem`**, which was a very simple shell that allowed us to run commands:

```
$ ./simplesh
simplesh> ls -l five-children.c
-rw------- 1 cgregg operator 1670 Sep 23 09:29 five-children.c
simplesh> date
Wed Oct  9 09:48:14 PDT 2019
simplesh>
```

- Now, let's look at a more sophisticated program, **`simplesh`** , which will allow us to run programs in the background as well as in the foreground.
- **`simplesh`** operates as a read-eval-print loop—often called a *repl*—which itself responds to the many things we type in by forking off child processes.
    - Each child process is initially a deep clone of the **`simplesh`** process.
    - Each proceeds to replace its own process image with the new one we specify, e.g. **`ls`**, **`cp`**, our own CS110 **`search`** (which we wrote during our second lecture), or even **`emacs`**.
    - As with traditional shells, a trailing ampersand—e.g. as with **`emacs &`**—is an instruction to execute the new process in the background without forcing the shell to wait for it to finish.
- Implementation of **`simplesh`** is presented on the next slide. Where helper functions don't rely on CS110 concepts, I omit their implementations.

Lecture 07: A more sophisticated shell (but not quite right yet...)

- Here's the core implementation of **`simplesh`** (full implementation is right here, and you can run the code on the following slide):

```c
int main(int argc, char *argv[]) {
  while (true) {
    char command[kMaxCommandLength + 1];
    readCommand(command, kMaxCommandLength);
    char *arguments[kMaxArgumentCount + 1];
    int count = parseCommandLine(command, arguments, kMaxArgumentCount);
    if (count == 0) continue;
    if (strcmp(arguments[0], "quit") ==) break; // hardcoded builtin to exit shell
    bool isbg = strcmp(arguments[count - 1], "&") == 0;
    if (isbg) arguments[--count] = NULL; // overwrite "&"
    pid_t pid = fork();
    if (pid == 0) execvp(arguments[0], arguments);
    if (isbg) { // background process, don't wait for child to finish
      printf("%d %s\n", pid, command);
    } else {    // otherwise block until child process is complete
      waitpid(pid, NULL, 0);
    }
  }
  printf("\n");
  return 0;
}
```

Lecture 07: A more sophisticated shell (but not quite right yet...)

https://cplayground.com/embed?p=fox-chimpanzee-hawk

Lecture 07: A more sophisticated shell (but not quite right yet...)

```c
int main(int argc, char *argv[]) {
  while (true) {
    char command[kMaxCommandLength + 1];
    readCommand(command, kMaxCommandLength);
    char *arguments[kMaxArgumentCount + 1];
    int count = parseCommandLine(command, arguments, kMaxArgumentCount);
    if (count == 0) continue;
    if (strcmp(arguments[0], "quit") ==) break; // hardcoded builtin to exit shell
    bool isbg = strcmp(arguments[count - 1], "&") == 0;
    if (isbg) arguments[--count] = NULL; // overwrite "&"
    pid_t pid = fork();
    if (pid == 0) execvp(arguments[0], arguments);
    if (isbg) { // background process, don't wait for child to finish
      printf("%d %s\n", pid, command);
    } else {    // otherwise block until child process is complete
      waitpid(pid, NULL, 0);
    }
  }
  printf("\n");
  return 0;
}
```

- What is the main issue with our **`simplesh`**?

Lecture 07: A more sophisticated shell (but not quite right yet...)

```
int main(int argc, char *argv[]) {
  while (true) {
    char command[kMaxCommandLength + 1];
    readCommand(command, kMaxCommandLength);
    char *arguments[kMaxArgumentCount + 1];
    int count = parseCommandLine(command, arguments, kMaxArgumentCount);
    if (count == 0) continue;
    if (strcmp(arguments[0], "quit") ==) break; // hardcoded builtin to exit shell
    bool isbg = strcmp(arguments[count - 1], "&") == 0;
    if (isbg) arguments[--count] = NULL; // overwrite "&"
    pid_t pid = fork();
    if (pid == 0) execvp(arguments[0], arguments);
    if (isbg) { // background process, don't wait for child to finish
      printf("%d %s\n", pid, command);
    } else {    // otherwise block until child process is complete
      waitpid(pid, NULL, 0);
    }
  }
  printf("\n");
  return 0;
}
```

- What is the main issue with our **`simplesh`**?
    - Background processes are left as zombies for the lifetime of the shell!
- We can handle this with signal handling.

Lecture 07: A more sophisticated shell (but not quite right yet...)

- Now we know about **`SIGCHLD`** signals and how to install **`SIGCHLD`** handlers to reap zombie processes. Let's upgrade our **`simplesh`** implementation to reap *all* process resources.

```c
// simplesh-with-redundancy.c
static void reapProcesses(int sig) {
  while (waitpid(-1, NULL, WNOHANG) > 0) {;} // nonblocking, iterate until retval is -1 or 0
}

int main(int argc, char *argv[]) {
  signal(SIGCHLD, reapProcesses);
  while (true) {
    // code to initialize command, argv, and isbg omitted for brevity
    pid_t pid = fork();
    if (pid == 0) {
      execvp(argv[0], argv);
      printf("%s: Command not found\n", argv[0]);
      exit(0);
    }
    if (isbg) {
      printf("%d %s\n", pid, command);
    } else {
      waitpid(pid, NULL, 0);
    }
  }
  printf("\n");
  return 0;
}
```

Lecture 07: A more sophisticated shell (but not quite right yet...)

- The last version actually works, but it relies on a sketchy call to **`waitpid`** to halt the shell until its foreground process has exited.
  - When the user creates a foreground process, normal execution flow advances to an isolated **`waitpid`** call to block until that process has terminated.
  - When the foreground process finishes, however, the **`SIGCHLD`** handler is invoked, and its **`waitpid`** call is the one that culls the foreground process's resources.
  - When the **`SIGCHLD`** handler exits, normal execution resumes, and the original call to **`waitpid`** returns -1 to state that there is no trace of a process with the supplied **`pid`**.
  - The version on the last slide deck is effectively calling **`waitpid`** from **`main`** just to block until the foreground process vanishes.
  - Even if you're content with this unorthodox use of **`waitpid`**—i.e. invoking a system call when you know it will fail—the **`waitpid`** call is redundant and replicates functionality better managed in the **`SIGCHLD`** handler.
    - We should only be calling **`waitpid`** in one place: the **`SIGCHLD`** handler.
    - This will be all the more apparent when we implement shells (e.g. Assignment 4's **`stsh`**) where multiple processes are running in the foreground as part of a pipeline (e.g. **`more words.txt | tee copy.txt | sort | uniq`**)

Lecture 07: A more sophisticated shell (but not quite right yet...)

- Here's an updated version that's careful to call **`waitpid`** from only one place.

```c
// simplesh-with-race-and-spin.c
static pid_t fgpid = 0; // global, intially 0, and 0 means no foreground process
static void reapProcesses(int sig) {
  while (true) {
    pid_t pid = waitpid(-1, NULL, WNOHANG);
    if (pid <= 0) break;
    if (pid == fgpid) fgpid = 0; // clear foreground process
  }
}

static void waitForForegroundProcess(pid_t pid) {
  fgpid = pid;
  while (fgpid == pid) {;}
}

int main(int argc, char *argv[]) {
  signal(SIGCHLD, reapProcesses);
  while (true) {
    // code to initialize command, argv, and isbg omitted for brevity
    pid_t pid = fork();
    if (pid == 0) execvp(argv[0], argv);
    if (isbg) {
      printf("%d %s\n", pid, command);
    } else {
      waitForForegroundProcess(pid);
    }
  }
  printf("\n");
  return 0;
}
```

Lecture 07: A more sophisticated shell (but not quite right yet...)

- The version on the last page introduces a global variable called `fgpid` to hold the process is of the foreground process. When there's no foreground process, `fgpid` is 0.
  - Because we don't control the signature of `reapProcesses`, we have to choice but to make `fgpid` a global.
  - Every time a new foreground process is created, `fgpid` is set to hold that process's pid. The shell then blocks by *spinning* in place until `fgpid` is cleared by `reapProcesses`.
- This version consolidates the `waitpid` code to reside in the handler and nowhere else.
- This version introduces two serious problems, so it's far from an A+ solution.
  - It's possible the foreground process finishes and `reapProcesses` is invoked on its behalf `before` normal execution flow updates `fgpid`. If that happens, the shell will spin forever and never advance up to the shell prompt. This is a race condition, and race conditions are no-nos.
  - The `while (fgpid == pid) {;}` is also a no-no. This allows the shell to spin on the CPU even when it can't do any meaningful work.
    - It would be substantially better for `simplesh` to yield the CPU and to only be considered for CPU time when there's a chance the foreground process has exited.