CS110 Summer 2019
Lecture 8: Race Conditions, Deadlock, and Data Integrity

Principles of Computer Systems
Stanford University, Dept. Of Computer Science
Lecturer: Roslyn Michelle Cyrus
Content adapted from material by Jerry Cain.
Announcements

- Assignment 1: grade explanation on Piazza.
Lecture Overview

- We'll work through some improvements to a **simplesh** program.
- We will discuss some subtle yet interesting race conditions that can lead to a state called **deadlock**, where a process seemingly freezes.
- We will also talk about how to properly use signal handlers to clean up all children.
Accessing Code Examples

- Today's lecture examples reside within:
  /usr/class/cs110/lecture-examples/processes.
  - First `ssh` into a myth machine (ssh `yourusername@myth.stanford.edu`). When prompted for your password, it is normal for the text not to appear as you enter your password. Once logged onto a myth machine, `cd` into the above directory.
  - To get started, type:
    `git clone /usr/class/cs110/lecture-examples cs110-lecture-examples` at the command prompt to create a local copy of the master.
  - Each time I mention there are new examples (or whenever you think to), descend into your local copy and type `git pull`. Doing so will update your local copy to match whatever the master has become.
Signals and Handlers Recap

Here’s an improved version of the job synchronization example.

```c
// job-list-synchronization-improved.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$
```bash
./job-list-synchronization-improved
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$
```bash
./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$
Another shell: simplesh

- Let's work through the implementation of a more sophisticated shell: the simplesh.
  - This is the best example of fork, waitpid, and execvp I can think of: a miniature shell not unlike those you've been using since the day you first logged into a myth machine.
  - 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 in an earlier 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.
  - A truncated implementation of simplesh is presented on the next slide.
Another shell: simplesh (continued)

- Here's the core implementation of an imperfect version of simplesh (full implementation is right here):

```c
int main(int argc, char *argv[]) {
    while (true) {
        // code to initialize command, argv, and isbg omitted for brevity
        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;
}
```

- The problem to be addressed: background processes are left as zombies for the lifetime of the shell. We now know how to fix this with signals and signal handlers.
Another shell: simplesh (continued)

- 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
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;
}
```
Another shell: simplesh (continued)

The last version actually works, but it relies on a sketchy call to \texttt{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 \texttt{waitpid} call to block until that process has terminated.
- When the foreground process finishes, however, the \texttt{SIGCHLD} handler is invoked, and its \texttt{waitpid} call is the one that reaps the foreground process's resources.
- When the \texttt{SIGCHLD} handler exits, normal execution resumes, and the original call to \texttt{waitpid} returns -1 to state that there is no trace of a process with the supplied \texttt{pid}.
- The version on the last slide is effectively calling \texttt{waitpid} from \texttt{main} just to block until the foreground process vanishes.
- Even if you're content with this unorthodox use of \texttt{waitpid}—i.e. invoking a system call when you know it will fail—the \texttt{waitpid} call is redundant and replicates functionality better managed in the \texttt{SIGCHLD} handler.
  - We should only be calling \texttt{waitpid} in one place: the \texttt{SIGCHLD} handler.
  - This will be all the more apparent when we implement shells (like Assignment 3's \texttt{stsh}) where multiple processes are running in the foreground as part of a pipeline (e.g. \texttt{more words.txt | tee copy.txt | sort | uniq})
Another shell: simplesh (continued)

Here's a buggy updated version (simplesh-with-race-and-spin.c) that's careful to call `waitpid` from only one place. There's a race condition here; do you see where?

```c
static pid_t fgpid = 0; // global, initially 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
", pid, command);
        else waitForForegroundProcess(pid);
    }
    printf("\n");
    return 0;
}
```
Another shell: simplesh (continued)

- The version on the last page introduces a global variable called \texttt{fgpid} to hold the process ID of the foreground process. When there's no foreground process, \texttt{fgpid} is 0.
  - Because we don't control the signature of \texttt{reapProcesses}, we have to make \texttt{fgpid} a global.
  - Every time a new foreground process is created, \texttt{fgpid} is set to hold that process's pid. The shell then blocks by \textit{spinning} in place until \texttt{fgpid} is cleared by \texttt{reapProcesses}.
- This version consolidates the \texttt{waitpid} code to reside in the handler and nowhere else.
- \textbf{This version introduces two serious problems, so it's far from an A+ solution.}
  - It's possible the foreground process finishes and \texttt{reapProcesses} is invoked on its behalf \textbf{before} normal execution flow updates \texttt{fgpid}. If that happens, the shell will spin forever and never advance up to the shell prompt. This is a race condition that leads to \textit{deadlock} and race conditions are no-nos.
  - The \texttt{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.
  
  \underline{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.}
Another shell: simplesh (continued)

The race condition can be cured by blocking `SIGCHLD` before forking, and only lifting that block after the global `fgpid` has been set:

```
// simplesh-with-spin.c: reapProcesses is the same as before

static void waitForForegroundProcess(pid_t pid) {
    fgpid = pid;
    unblockSIGCHLD(); // lift only after fgpid has been set
    while (fgpid == pid) {};
}

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

// simplesh-utils.c: a collection of helper functions

static void toggleSIGCHLDBlock(int how) {
    sigset_t mask;
    sigemptyset(&mask);
    sigaddset(&mask, SIGCHLD);
    sigprocmask(how, &mask, NULL);
}

void blockSIGCHLD() {
    toggleSIGCHLDBlock(SIG_BLOCK);
}

void unblockSIGCHLD() {
    toggleSIGCHLDBlock(SIG_UNBLOCK);
}

Note that we call `unblockSIGCHLD` in the child, before the `execvp` call. We do so because the child will otherwise inherit the signal block.
Race condition is now gone!

Note that we call `blockSIGCHLD` before `fork`, and we don't lift the block until `fgpid` has been set to the `pid` of the new foreground process.

We also call `unblockSIGCHLD` in the child right before the `execvp` call.

- The child executable could very well depend on multiprocessing. If so, it would certainly call `fork` and rely on `SIGCHLD` signals and signal handling.
- If we forget to call `unblockSIGCHLD`, the child process inherits the `SIGCHLD` block across the `execvp` boundary. That would compromise the child's ability to work properly.

We also need to call `unblockSIGCHLD` for background processes. We do so after bookkeeping information is `printf`-ed to the screen, as we did for `job-list-fixed`.

We have not addressed the CPU spin issue, and we really need to.

- We could change the while loop from `while (fgpid == pid) {;}` to `while (fgpid == pid) {usleep(100000);}`, as we have in this version.
- `usleep` call will push the shell off the CPU every time it realizes it shouldn't have gotten it in the first place. But we'd really prefer to keep the shell off the CPU until the OS has some information suggesting the foreground process is done.
Another shell: simplesh (continued)

- The C libraries provide a `pause` function, which forces the process to sleep until some unblocked signal arrives. This sounds promising, because we know `fgpid` can only be changed because a `SIGCHLD` signal comes in and `reapProcesses` is executed.
- A version of `simplesh` whose `waitForForegroundProcess` implementation relies on `pause` is presented below on the left.
- **The problem here?** `SIGCHLD` may arrive after `fgpid == pid` evaluates to `true` but before the call to `pause` it's committed to. That would be unfortunate, because it's possible `simplesh` isn't managing any other processes, which means that no other signals, much less `SIGCHLD` signals, will arrive to lift `simplesh` out of its `pause` call. That would leave `simplesh` in a **deadlock** state (a state in which no progress can be made).
- You might think the second (lower right) version might help, but it has the same problem!

```c
// simplesh-with-pause-1.c
static void waitForForegroundProcess(pid_t pid) {
    fgpid = pid;
    unblockSIGCHLD();
    while (fgpid == pid) {
        pause();
    }
}
```

```c
// simplesh-with-pause-2.c
static void waitForForegroundProcess(pid_t pid) {
    fgpid = pid;
    while (fgpid == pid) {
        unblockSIGCHLD();
        pause();
        blockSIGCHLD();
    }
    unblockSIGCHLD();
}
```
Another shell: simplesh (continued)

- The problem with both versions of `waitForForegroundProcess` on the prior slide is that each lifts the block on `SIGCHLD` before going to sleep via `pause`.
- The one `SIGCHLD` you're relying on to notify the parent that the child has finished could very well arrive in the narrow space between lift and sleep. That would inspire deadlock.
- The solution is to rely on a more specialized, atomic version of `pause` called `sigsuspend`, which asks that the OS change the blocked set to the one provided, but only after the caller has been forced off the CPU. When some unblocked signal arrives, the process gets the CPU, the signal is handled, the original blocked set is restored, and `sigsuspend` returns.

```c
// simplesh-all-better.c
static void waitForForegroundProcess(pid_t pid) {
    fgpid = pid;
    sigset_t empty;
    sigemptyset(&empty);
    while (fgpid == pid) {
        sigsuspend(&empty);
    }
    unblockSIGCHLD();
}
```

This is the model solution to our problem, and one you should emulate in your Assignment 2’s `farm` and your Assignment 3 `stsh`. 
The **sigsuspend** function temporarily replaces the current blocked set with the mask passed to it and then suspends the process until the receipt of a signal whose action is either to run a handler or to terminate the process.

- If the action is to terminate, then the process terminates without returning from **sigsuspend**.
- If the action is to run a handler, then **sigsuspend** returns after the handler returns, restoring the blocked set to its state when **sigsuspend** was called.

**sigsuspend** is equivalent to an **atomic** (uninterruptible) version of the code on the right:

```c
// simplesh-all-better.c
static void waitForForegroundProcess(pid_t pid) {
    fgpid = pid;
    sigset_t empty;
    sigemptyset(&empty);
    while (fgpid == pid) {
        sigsuspend(&empty);
    }
    unblockSIGCHLD();
}
```

```c
  sigprocmask(SIG_SETMASK, &mask, &prev);
pause(); // sleep until signal comes in
  sigprocmask(SIG_SETMASK, &prev, NULL);
```
End of Lecture 8