

# CS110 Lecture 07: Signals

**Principles of Computer Systems**

Winter 2020

Stanford University

Computer Science Department

**Instructors:** Chris Gregg and  
Nick Troccoli

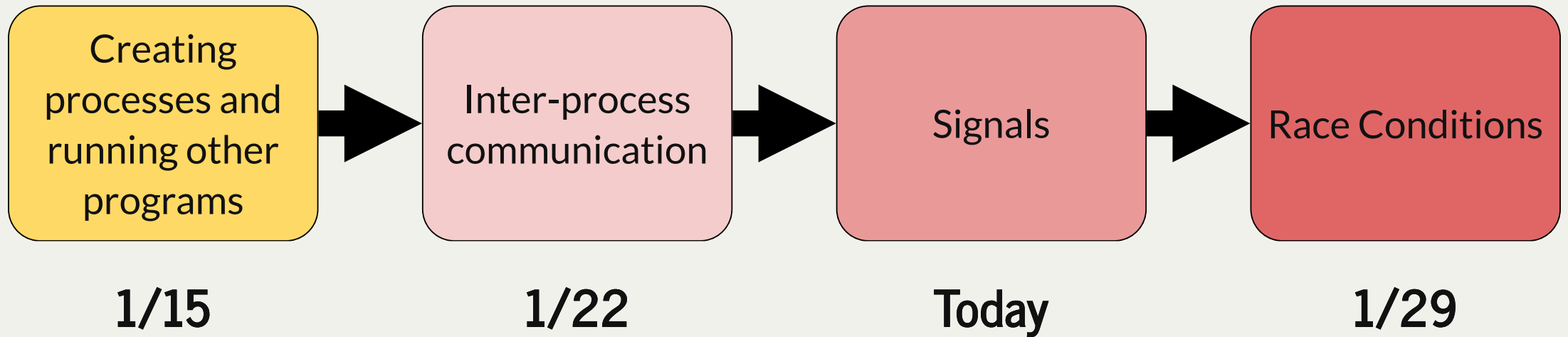


[PDF of this presentation](#)

# CS110 Topic 2: How can our programs create and interact with other programs?



# Learning About Processes



# Today's Learning Goals

- Introduce *signals* as another way for processes to communicate
- Learn how to execute code in our program when we receive a signal
- Learn how to block and resume signals



# Plan For Today

- Introducing Signals
- **Demo:** Disneyland
- Signals Aren't Queued
- **Demo:** Return Trip To Disneyland
- Concurrency
- Blocking Signals
- **Demo:** Revisiting Our Shell
- Waiting For Signals



# Plan For Today

- **Introducing Signals**
- **Demo:** Disneyland
- Signals Aren't Queued
- **Demo:** Return Trip To Disneyland
- Concurrency
- Blocking Signals
- **Demo:** Revisiting Our Shell
- Waiting For Signals



# Interprocess Communication

- It's useful for a parent process to be able to communicate with its child (and vice versa)
- There are two key ways we will learn to do this: **pipes** and **signals**
  - **Pipes** let two processes send and receive arbitrary data
  - **Signals** let two processes send and receive certain "signals" that indicate something special has happened.



# Signals

A **signal** is a way to notify a process that an event has occurred

- There is a list of defined signals that can be sent (or you can define your own): SIGINT, SIGSTOP, SIGKILL, SIGCONT, etc.
- A signal is really a number (e.g. SIGSEGV is 11)
- A program can have a function executed when a type of signal is received
- Signals are sent either by the operating system, or by another process
- You can send a signal to yourself or to another process you own





# Signals

Here are some examples of signals:

- **SIGINT** - when you type Ctl-c in the terminal, the kernel sends a SIGINT to the foreground process group. The default behavior is to terminate.
- **SIGTSTP** - when you type Ctl-z in the terminal, the kernel sends a SIGTSTP to the foreground process group. The default behavior is to halt it until it is told to continue.
- **SIGSEGV** - when your program attempts to access an invalid memory address, the kernel sends a SIGSEGV ("seg fault"). The default behavior is to terminate.



# Process Lifecycle

**Running** - a process is either executing or waiting to execute

**Stopped** - a process is suspended due to receiving a SIGSTOP or similar signal. A process will resume if it receives a SIGCONT signal.

**Terminated** - a process is permanently stopped, either due to finishing, or receiving a signal such as SIGSEGV or SIGKILL whose default behavior is to terminate the process.

When a child changes state, the kernel sends a SIGCHLD signal to its parent.



# Sending Signals

The operating system sends many signals, but we can also send signals manually.

```
int kill(pid_t pid, int signum);  
  
// same as kill(getpid(), signum)  
int raise(int signum);
```

- **kill** sends the specified signal to the specified process (poorly-named; previously, default was to just terminate target process)
- **pid** parameter can be  $> 0$  (specify single pid),  $< -1$  (specify process group  $\text{abs}(\text{pid})$ ), or  $0/-1$  (we ignore these).
- **raise** sends the specified signal to yourself



# waitpid()

Waitpid can be used to wait on children to terminate *or change state*:

```
pid_t waitpid(pid_t pid, int *status, int options);
```

- **pid**: the PID of the child to wait on, or -1 to wait on any of our children
- **status**: where to put info about the child's status (or NULL)
- the return value is the PID of the child that was waited on, -1 on error, or 0 if there are other children to wait for, but we are not blocking.

The default behavior is to wait for the specified child process to exit. **options** lets us customize this further (can combine these flags using | ):

- **WUNTRACED** - also wait on a child to be stopped
- **WCONTINUED** - also wait on a child to be continued
- **WNOHANG** - don't block



# Signal Handlers

We can have a function of our choice execute when a certain signal is received.

- We must register this "signal handler" with the operating system, and then it will be called for us.

```
typedef void (*sighandler_t)(int);  
...  
sighandler_t signal(int signum, sighandler_t handler);
```

- **signum** is the signal (e.g. SIGCHLD) we are interested in.
- **handler** is a function pointer for the function to call when this signal is received.
- (Note: no handlers allowed for SIGSTOP or SIGKILL)



# Plan For Today

- Introducing Signals
- **Demo: Disneyland**
- Signals Aren't Queued
- **Demo: Return Trip To Disneyland**
- Concurrency
- Blocking Signals
- **Demo: Revisiting Our Shell**
- Waiting For Signals



# Signal Handlers (five-children.c)

```
1 // five-children.c
2 static const size_t kNumChildren = 5;
3 static size_t numChildrenDonePlaying = 0;
4
5 static void reapChild(int sig) {
6     waitpid(-1, NULL, 0);
7     numChildrenDonePlaying++;
8 }
9
10 int main(int argc, char *argv[]) {
11     printf("Let my five children play while I take a nap.\n");
12     signal(SIGCHLD, reapChild);
13     for (size_t kid = 1; kid <= kNumChildren; kid++) {
14         if (fork() == 0) {
15             sleep(3 * kid); // sleep emulates "play" time
16             printf("Child #%zu tired... returns to parent.\n", kid);
17             return 0;
18         }
19     }
20
21     while (numChildrenDonePlaying < kNumChildren) {
22         printf("At least one child still playing, so parent nods off.\n");
23         snooze(5); // custom fn to sleep uninterrupted
24         printf("Parent wakes up! ");
25     }
26     printf("All children accounted for. Good job, parent!\n");
27     return 0;
28 }
```

- In this program, a parent takes their five children 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.
- Similar to many parallel data processing applications where parent does other work while children are busy
- (Full program is [right here.](#))



# Signal Handlers

A signal can be received at any time, and a signal handler can execute at any time.

- Signals aren't handled immediately (there can be delays)
- Signal handlers can execute at any point during the program execution (eg. pause main() execution, execute handler, resume main() execution)
  - **Goal:** keep signal handlers simple!
- Similar to hardware interrupts -- POSIX brings that model to software





# Signal Handlers (five-children.c)

```
1 // five-children.c
2 static const size_t kNumChildren = 5;
3 static size_t numChildrenDonePlaying = 0;
4
5 static void reapChild(int sig) {
6     waitpid(-1, NULL, 0);
7     numChildrenDonePlaying++;
8 }
9
10 int main(int argc, char *argv[]) {
11     printf("Let my five children play while I take a nap.\n");
12     signal(SIGCHLD, reapChild);
13     for (size_t kid = 1; kid <= kNumChildren; kid++) {
14         if (fork() == 0) {
15             sleep(3); // sleep emulates "play" time
16             printf("Child #%zu tired... returns to parent.\n", kid);
17             return 0;
18         }
19     }
20
21     while (numChildrenDonePlaying < kNumChildren) {
22         printf("At least one child still playing, so parent nods off.\n");
23         snooze(5); // custom fn to sleep uninterrupted
24         printf("Parent wakes up! ");
25     }
26     printf("All children accounted for. Good job, parent!\n");
27     return 0;
28 }
```

What happens if all children sleep for the same amount of time? (E.g. change line 15 from `sleep(3 * kid)` to `sleep(3)`).



# Plan For Today

- Introducing Signals
- **Demo:** Disneyland
- **Signals Aren't Queued**
- **Demo:** Return Trip To Disneyland
- Concurrency
- Blocking Signals
- **Demo:** Revisiting Our Shell
- Waiting For Signals



# Signal Handlers

**Problem:** a signal handler is called if *one or more* signals are sent.

- Like a notification that "one or more signals are waiting for you!"
- The kernel tracks only *what* signals should be sent to you, not *how many*
- When we are sleeping, multiple children could terminate, but result in 1 SIGCHLD!

**Solution:** signal handler should clean up as many children as possible.



# Signal Handlers

```
1 static void reapChild(int sig) {  
2     waitpid(-1, NULL, 0);  
3     numChildrenDonePlaying++;  
4 }
```

Let's add a loop to reap as many children as possible.



# Signal Handlers

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

Let's add a loop to reap as many children as possible.



# Signal Handlers

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

Let's add a loop to reap as many children as possible.

**Problem:** this may block if other children are taking longer! We only want to clean up children that are done *now*. Others will signal later.



# Signal Handlers

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

Let's add a loop to reap as many children as possible.

**Problem:** this may block if other children are taking longer! We only want to clean up children that are done *now*. Others will signal later.

**Solution:** use **WNOHANG**, which means don't block. If there are children we *would have* waited on but aren't, returns 0. -1 typically means no children left.



# Signal Handlers

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

Let's add a loop to reap as many children as possible.

**Solution:** use **WNOHANG**, which means don't block. If there are children we *would* have waited on but aren't, returns 0. -1 typically means no children left.

**Note:** the kernel blocks additional signals of that type while a signal handler is running (they are sent later).





# Plan For Today

- Introducing Signals
- **Demo:** Disneyland
- Signals Aren't Queued
- **Demo: Return Trip To Disneyland**
- Concurrency
- Blocking Signals
- **Demo:** Revisiting Our Shell
- Waiting For Signals



# Plan For Today

- Introducing Signals
- **Demo:** Disneyland
- Signals Aren't Queued
- **Demo:** Return Trip To Disneyland
- **Concurrency**
- Blocking Signals
- **Demo:** Revisiting Our Shell
- Waiting For Signals



# Concurrency

**Concurrency** means performing multiple actions at the same time.

- Concurrency is extremely powerful: it can make your systems faster, more responsive, and more efficient. It's fundamental to all modern software.
- When you introduce multiprocessing (e.g. **fork**) and asynchronous signal handling (e.g. **signal**), it's possible to have concurrency issues. These are tricky!
- Most challenges come with shared data - e.g. two routines using the same variable.
- Many large systems parallelize computations by trying to eliminate shared data - e.g. split the data into independent chunks and process in parallel.
- A **race condition** is an unpredictable ordering of events (due to e.g. OS scheduling) where some orderings may cause undesired behavior.



# Off To The Races

Consider the following program, which is similar to Assignment 4. (The full program, with error checking, is [right here](#)):

- The program spawns off three child processes at one-second intervals.
- 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.
- Let's look at **job-list-broken.c**.



# Off To The Races

```
1 // job-list-broken.c
2 static void reapProcesses(int sig) {
3     while (true) {
4         pid_t pid = waitpid(-1, NULL, WNOHANG);
5         if (pid <= 0) break;
6         printf("Job %d removed from job list.\n", pid);
7     }
8 }
9
10 char * const kArguments[] = {"date", NULL};
11 int main(int argc, char *argv[]) {
12     signal(SIGCHLD, reapProcesses);
13     for (size_t i = 0; i < 3; i++) {
14         pid_t pid = fork();
15         if (pid == 0) execvp(kArguments[0], kArguments);
16         sleep(1); // force parent off CPU
17         printf("Job %d added to job list.\n", pid);
18     }
19     return 0;
20 }
```

```
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$
```

**Symptom:** it looks like jobs are being removed from the list before being added! How is this possible?



# Off To The Races

```
1 // job-list-broken.c
2 static void reapProcesses(int sig) {
3     while (true) {
4         pid_t pid = waitpid(-1, NULL, WNOHANG);
5         if (pid <= 0) break;
6         printf("Job %d removed from job list.\n", pid);
7     }
8 }
9
10 char * const kArguments[] = {"date", NULL};
11 int main(int argc, char *argv[]) {
12     signal(SIGCHLD, reapProcesses);
13     for (size_t i = 0; i < 3; i++) {
14         pid_t pid = fork();
15         if (pid == 0) execvp(kArguments[0], kArguments);
16         sleep(1); // force parent off CPU
17         printf("Job %d added to job list.\n", pid);
18     }
19     return 0;
20 }
```

**Cause:** there is a *race condition* with the signal handler. It is possible for the child to execute and terminate before the parent adds the job to the job list.

Therefore, the signal handler will be called to remove the job before the parent adds the job!



# Off To The Races

```
1 // job-list-broken.c
2 static void reapProcesses(int sig) {
3     while (true) {
4         pid_t pid = waitpid(-1, NULL, WNOHANG);
5         if (pid <= 0) break;
6         printf("Job %d removed from job list.\n", pid);
7     }
8 }
9
10 char * const kArguments[] = {"date", NULL};
11 int main(int argc, char *argv[]) {
12     signal(SIGCHLD, reapProcesses);
13     for (size_t i = 0; i < 3; i++) {
14         pid_t pid = fork();
15         if (pid == 0) execvp(kArguments[0], kArguments);
16         sleep(1); // force parent off CPU
17         printf("Job %d added to job list.\n", pid);
18     }
19     return 0;
20 }
```

**Cause:** there is a *race condition* with the signal handler. It is possible for the child to execute and terminate before the parent adds the job to the job list.

It would be nice if there was a *do not disturb* for signals so that we could temporarily block SIGCHLD in the parent while adding a new job.



# Plan For Today

- Introducing Signals
- **Demo:** Disneyland
- Signals Aren't Queued
- **Demo:** Return Trip To Disneyland
- Concurrency
- **Blocking Signals**
- **Demo:** Revisiting Our Shell
- Waiting For Signals





# Do Not Disturb

The `sigprocmask` function lets us temporarily block signals of the specified types. Instead, they will be queued up and delivered when the block is removed.

```
int sigprocmask(int how, const sigset_t *set, sigset_t *oldset);
```

- **how** is **SIG\_BLOCK** (add this to the list of signals to block), **SIG\_UNBLOCK** (remove this from the list of signals to block) or **SIG\_SETMASK** (make this the list of signals to block)
- **set** is a special type that specifies the signals to add/remove/replace with
- **oldset** is the location of where to store the *previous* blocked set that we are overwriting.



# Do Not Disturb

`sigset_t` is a special type (usually a 32-bit int) used as a bit vector. It must be created and initialized using special functions (we generally ignore the return values).

```
// Initialize to the empty set of signals
int sigemptyset(sigset_t *set);

// Set to contain all signals
int sigfillset(sigset_t *set);

// Add the specified signal
int sigaddset(sigset_t *set, int signum);

// Remove the specified signal
int sigdelset(sigset_t *set, int signum);
```

```
static void imposeSIGCHLDBlock() {
    sigset_t set;
    sigemptyset(&set);
    sigaddset(&set, SIGCHLD);
    sigprocmask(SIG_BLOCK, &set, NULL);
}

static void unblockSignals(int *signals, int numSignals) {
    sigset_t set;
    sigemptyset(&set);
    for (int i = 0; i < numSignals; i++) {
        sigaddset(&set, signals[i]);
    }
    sigprocmask(SIG_UNBLOCK, &set, NULL);
}
```



# Off To The Races

```
1 // job-list-broken.c
2 static void reapProcesses(int sig) {
3     while (true) {
4         pid_t pid = waitpid(-1, NULL, WNOHANG);
5         if (pid <= 0) break;
6         printf("Job %d removed from job list.\n", pid);
7     }
8 }
9
10 char * const kArguments[] = {"date", NULL};
11 int main(int argc, char *argv[]) {
12     signal(SIGCHLD, reapProcesses);
13     for (size_t i = 0; i < 3; i++) {
14         pid_t pid = fork();
15         if (pid == 0) execvp(kArguments[0], kArguments);
16         sleep(1); // force parent off CPU
17         printf("Job %d added to job list.\n", pid);
18     }
19     return 0;
20 }
```

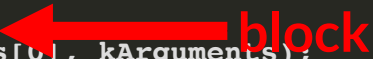
Where should we block and unblock SIGCHLD signals in the parent to fix the race condition?

**Goal:** we want to block signals for as little time as possible to maximize performance.



# Off To The Races

```
1 // job-list-broken.c
2 static void reapProcesses(int sig) {
3     while (true) {
4         pid_t pid = waitpid(-1, NULL, WNOHANG);
5         if (pid <= 0) break;
6         printf("Job %d removed from job list.\n", pid);
7     }
8 }
9
10 char * const kArguments[] = {"date", NULL};
11 int main(int argc, char *argv[]) {
12     signal(SIGCHLD, reapProcesses);
13     for (size_t i = 0; i < 3; i++) {
14         pid_t pid = fork();
15         if (pid == 0) execvp(kArguments[0], kArguments);
16         sleep(1); // force parent off CPU
17         printf("Job %d added to job list.\n", pid);
18     }
19     return 0;
20 }
```



Where should we block and unblock SIGCHLD signals in the parent to fix the race condition?

**Goal:** we want to block signals for as little time as possible to maximize performance.

**Block** just before line 14.



# Off To The Races

```
1 // job-list-broken.c
2 static void reapProcesses(int sig) {
3     while (true) {
4         pid_t pid = waitpid(-1, NULL, WNOHANG);
5         if (pid <= 0) break;
6         printf("Job %d removed from job list.\n", pid);
7     }
8 }
9
10 char * const kArguments[] = {"date", NULL};
11 int main(int argc, char *argv[]) {
12     signal(SIGCHLD, reapProcesses);
13     for (size_t i = 0; i < 3; i++) {
14         pid_t pid = fork();
15         if (pid == 0) execvp(kArguments[0], kArguments);
16         sleep(1); // force parent off CPU
17         printf("Job %d added to job list.\n", pid);
18     }
19     return 0;
20 }
```

← block

← unblock

Where should we block and unblock SIGCHLD signals in the parent to fix the race condition?

**Goal:** we want to block signals for as little time as possible to maximize performance.

Block just before line 14.

Unblock after line 17.



# Off To The Races

```
1 // job-list-fixed.c
2 char * const kArguments[] = {"date", NULL};
3 int main(int argc, char *argv[]) {
4     signal(SIGCHLD, reapProcesses);
5
6     // Create set with just SIGCHLD
7     sigset_t set;
8     sigemptyset(&set);
9     sigaddset(&set, SIGCHLD);
10
11     for (size_t i = 0; i < 3; i++) {
12         sigprocmask(SIG_BLOCK, &set, NULL);
13         pid_t pid = fork();
14         if (pid == 0) {
15             sigprocmask(SIG_UNBLOCK, &set, NULL);
16             execvp(kArguments[0], kArguments);
17         }
18         sleep(1); // force parent off CPU
19         printf("Job %d added to job list.\n", pid);
20         sigprocmask(SIG_UNBLOCK, &set, NULL);
21     }
22     return 0;
23 }
```

Where should we block and unblock SIGCHLD signals in the parent to fix the race condition?

Block SIGCHLD just before creating a child (line 12). Unblock SIGCHLD once we've added the job to the list (line 20).

(Full program [here](#))



# Off To The Races

```
1 // job-list-fixed.c
2 char * const kArguments[] = {"date", NULL};
3 int main(int argc, char *argv[]) {
4     signal(SIGCHLD, reapProcesses);
5
6     // Create set with just SIGCHLD
7     sigset_t set;
8     sigemptyset(&set);
9     sigaddset(&set, SIGCHLD);
10
11     for (size_t i = 0; i < 3; i++) {
12         sigprocmask(SIG_BLOCK, &set, NULL);
13         pid_t pid = fork();
14         if (pid == 0) {
15             sigprocmask(SIG_UNBLOCK, &set, NULL);
16             execvp(kArguments[0], kArguments);
17         }
18         sleep(1); // force parent off CPU
19         printf("Job %d added to job list.\n", pid);
20         sigprocmask(SIG_UNBLOCK, &set, NULL);
21     }
22     return 0;
23 }
```

Side note: forked children inherit blocked signals, so we must remove the block in the child (line 15).



# Plan For Today

- Introducing Signals
- **Demo:** Disneyland
- Signals Aren't Queued
- **Demo:** Return Trip To Disneyland
- Concurrency
- Blocking Signals
- **Demo: Revisiting Our Shell**
- Waiting For Signals





# Revisiting Our Shell

Last lecture we implemented a more advanced shell that could run commands in the foreground, or background (with &). Let's see a quick demo (second-shell-soln.c)



# Revisiting Our Shell

Last lecture we implemented a more advanced shell that could run commands in the foreground, or background (with &). Let's see a quick demo (second-shell-soln.c)

There's one core problem with this implementation, that we can now fix with our knowledge of signals and signal handlers. What is it?



# Revisiting Our Shell

Last lecture we implemented a more advanced shell that could run commands in the foreground, or background (with &). Let's see a quick demo (second-shell-soln.c)

There's one core problem with this implementation, that we can now fix with our knowledge of signals and signal handlers. What is it?

**If a process runs in the background, it's not cleaned up!**



# Revisiting Our Shell

Last lecture we implemented a more advanced shell that could run commands in the foreground, or background (with &). Let's see a quick demo (second-shell-soln.c)

There's one core problem with this implementation, that we can now fix with our knowledge of signals and signal handlers. What is it?

**If a process runs in the background, it's not cleaned up!**

**Fix:** let's add a SIGCHLD handler to clean them up.



# Revisiting Our Shell

```
1 static void executeCommand(char *command, bool inBackground) {
2     pid_t pidOrZero = fork();
3     if (pidOrZero == 0) {
4         char *arguments[] = {"/bin/sh", "-c", command, NULL};
5         execvp(arguments[0], arguments);
6         exit(1);
7     }
8
9     // If we are the parent, either wait or return immediately
10    if (inBackground) {
11        printf("%d %s\n", pidOrZero, command);
12    } else {
13        waitpid(pidOrZero, NULL, 0);
14    }
15 }
16
17 static void reapProcesses(int signum) {
18     while (true) {
19         int result = waitpid(-1, NULL, WNOHANG);
20         if (result <= 0) break;
21     }
22 }
23
24 int main(int argc, char *argv[]) {
25     signal(SIGCHLD, reapProcesses);
26     ...
27 }
```

Now we have a handler that cleans up terminated children.

**Issue:** this handler will be called to clean up *all* children, even foreground commands.



# Revisiting Our Shell

```
1 static void executeCommand(char *command, bool inBackground) {
2     pid_t pidOrZero = fork();
3     if (pidOrZero == 0) {
4         char *arguments[] = {"/bin/sh", "-c", command, NULL};
5         execvp(arguments[0], arguments);
6         exit(1);
7     }
8
9     // If we are the parent, either wait or return immediately
10    if (inBackground) {
11        printf("%d %s\n", pidOrZero, command);
12    } else {
13        waitpid(pidOrZero, NULL, 0);
14    }
15 }
16
17 static void reapProcesses(int signum) {
18     while (true) {
19         pid_t result = waitpid(-1, NULL, WNOHANG);
20         if (result <= 0) break;
21     }
22 }
23
24 int main(int argc, char *argv[]) {
25     signal(SIGCHLD, reapProcesses);
26     ...
27 }
```

Now we have a handler that cleans up terminated children.

**Issue:** this handler will be called to clean up *all* children, even foreground commands.

Therefore, the waitpid on line 13 will *always* return -1. Can we get rid of it?



# Revisiting Our Shell

```
1 // The currently-running foreground command PID
2 static pid_t foregroundPID = 0;
3
4 static void waitForForegroundCommand(pid_t pid) {
5     foregroundPID = pid;
6     while (foregroundPID == pid) {}
7 }
8
9 static void executeCommand(char *command, bool inBackground) {
10     // ...(omitted for brevity)...
11     if (inBackground) {
12         printf("%d %s\n", pidOrZero, command);
13     } else {
14         waitForForegroundCommand(pidOrZero);
15     }
16 }
17
18 static void reapProcesses(int signum) {
19     while (true) {
20         pid_t result = waitpid(-1, NULL, WNOHANG);
21         if (result <= 0) break;
22         if (result == foregroundPID) foregroundPID = 0;
23     }
24 }
```

- The second `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 critical 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`)
- Here's an [updated version](#) that's careful to call `waitpid` from only one place.



# Revisiting Our Shell

- Because we don't control the signature of `reapProcesses`, we must 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.
  - There is a hidden race condition... (addressed next lecture)
  - The `while (fgpid == pid) { ; }` is also not good. This allows the shell to spin on the CPU even when it can't do any meaningful work.
  - **Goal:** we want to *yield the CPU until we receive a SIGCHLD signal*.





# Plan For Today

- Introducing Signals
- **Demo:** Disneyland
- Signals Aren't Queued
- **Demo:** Return Trip To Disneyland
- Concurrency
- Blocking Signals
- **Demo:** Revisiting Our Shell
- **Waiting For Signals**



# Waiting For Signals

If our program can do no meaningful work until we receive a signal, we should tell the operating system so that it can take us off the CPU for now.

```
1 int sigsuspend(const sigset_t *mask);
```

- This function takes the process off the CPU until a signal is sent that is NOT in the specified mask.
- This is the model solution to our problem, and one you should emulate in your Assignment 3 **farm** and your Assignment 4 **stsh**.

```
1 // simplesh-all-better.c
2 static void waitForegroundProcess(pid_t pid) {
3     fgpid = pid;
4     sigset_t empty;
5     sigemptyset(&empty);
6     while (fgpid == pid) {
7         sigsuspend(&empty);
8     }
9 }
```

- (Note: there is one more race that we will fix next lecture)



# Overview: Signals and Concurrency

- Concurrency is powerful: it lets our code do many things at the same time
  - It can run faster (more cores!)
  - It can do more (run many programs in background)
  - It can respond faster (don't have to wait for current action to complete)
- Signals are a way for concurrent processes to interact
  - Send signals with kill and raise
  - Handle signals with signal
  - Control signal delivery with sigprocmask, sigsuspend
  - Preempt running code
  - Making sure code running in a signal handler works correctly is difficult
  - *Race conditions* occur when code can see data in an intermediate and invalid state (often KABOOM)
- Assignments 3 and 4 use signals, as a way to start easing into concurrency before we tackle multithreading
- Take CS149 if you want to learn how to write high concurrency code that runs 100x faster

