CS110 Lecture 11: Signals, Part 2

CS110: Principles of Computer Systems

Winter 2021-2022

Stanford University

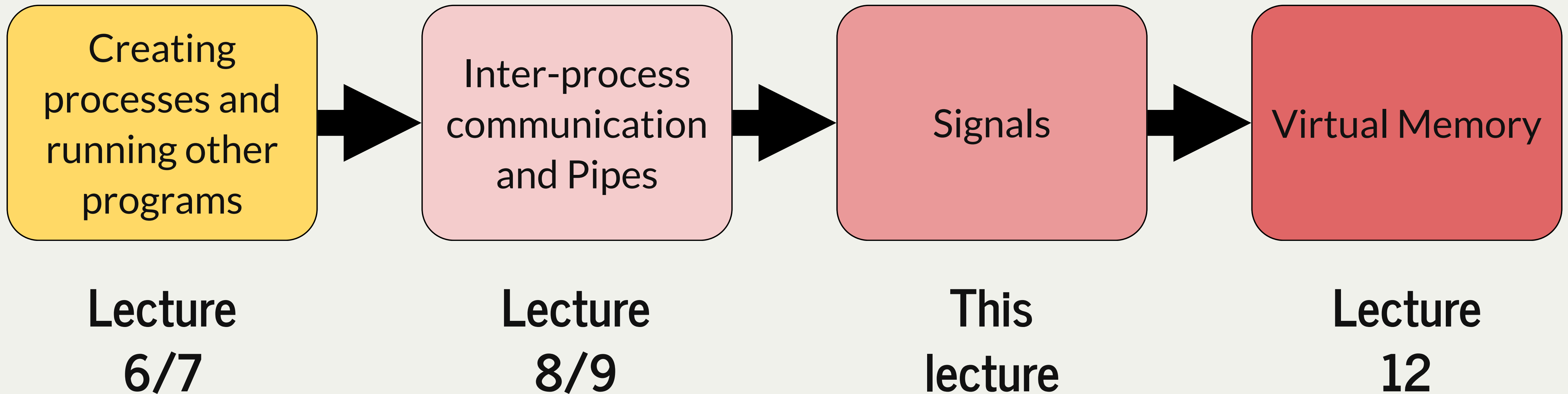
Instructors: Nick Troccoli and Jerry Cain

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

PDF of this presentation

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

Learning About Processes



assign3: implement multiprocessing programs like "trace" (to trace another program's behavior) and "farm" (parallelize tasks)

assign4: implement your own shell!

Learning Goals

- Learn about how to handle SIGCHLD signals to clean up child processes
- Get practice writing signal handlers
- See the challenges and pitfalls of signal handlers
- Learn how to temporarily ignore signals with **sigprocmask** and wait for signals with **sigwait**

Plan For Today

- Recap: Signals so far
- SIGCHLD Handlers
- **Demo:** Return Trip To Disneyland
- Concurrency Challenges
- Waiting for Signals with **sigwait** and **sigprocmask**

Plan For Today

- **Recap: Signals so far**
- SIGCHLD Handlers
- **Demo:** Return Trip To Disneyland
- Concurrency Challenges
- Waiting for Signals with **sigwait** and **sigprocmask**

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
  - e.g. SIGCHLD sent by OS to parent when child changes state
- You can send a signal to yourself or to another process you own

Signals

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

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

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 for most signals was to just terminate target process)
- **pid** parameter can be > 0 (specify single pid), < -1 (specify process group abs(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)

Signal Handlers

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

- Signals aren't always 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!

Signal Handlers

- Signals like SIGSEGV and SIGFPE are called "traps" and are typically sent if there was a problem with the program. Their signal handlers are called immediately, within the same time slice, after the offending instruction is executed.
- Signals like SIGCHLD and SIGINT are called "interrupts" and are typically sent because something external to the process occurred. Their signal handlers may not be called immediately:
  - Generally invoked at the beginning of the recipient's next time slice.
  - If the recipient is on the CPU when the signal arrives, it can be executed immediately, but it's typically deferred until its next time slice begins.
- Programs can rarely recover from traps, though may be able to recover from interrupts. That's why the default handler for most traps is to terminate the process and the default handler for most interrupts is something else.

Plan For Today

- Recap: Signals so far
- **SIGCHLD Handlers**
- **Demo:** Return Trip To Disneyland
- Concurrency Challenges
- Waiting for Signals with **sigwait** and **sigprocmask**

SIGCHLD

Key insight: when a child changes state, the kernel sends a SIGCHLD signal to its parent.

- This allows the parent to be notified its child has e.g. terminated while doing other work
- we can add a SIGCHLD handler to clean up children without waiting on them in the parent!

SIGCHLD Example: Disneyland

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

SIGCHLD Example: Disneyland

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

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

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 handler call!

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

SIGCHLD 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.

SIGCHLD 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     numChildrenDonePlaying++;
6   }
7 }
```

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

Question: what does the **waitpid** loop do if one child terminates but other children are still running?

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. (DEMO)

SIGCHLD 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     numChildrenDonePlaying++;
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

- Recap: Signals so far
- SIGCHLD Handlers
- **Demo: Return Trip To Disneyland**
- Concurrency Challenges
- Waiting for Signals with **sigwait** and **sigprocmask**

Demo: five-children.c

five-children-soln.c

Plan For Today

- Recap: Signals so far
- SIGCHLD Handlers
- **Demo:** Return Trip To Disneyland
- **Concurrency Challenges**
- Waiting for Signals with **sigwait** and **sigprocmask**

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, **job-list-broken.c**:

- The program spawns off three child processes at one-second intervals.
- Each child process prints the date and time it was spawned.
- The parent maintains a pretend job list (doesn't actually maintain a data structure, just prints where operations would have been performed).

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!

Signal Handler Challenges

- Signal handlers can interrupt execution at unpredictable times
- There are ways to guard against this, but it adds a lot of complexity.
- Also, ideally we rely only on signal-handler-safe functions in signal handlers, but most of them are system calls. E.g. no **printf**!
- Signal handlers are difficult to use properly, and the consequences can be severe. Many regard signals to be one of the worst parts of Unix's design.
    - This installment of Ghosts of Unix Past explains why asynchronous signal handling can be such a headache. Main point: can be executed at bad times (while the main execution flow is in the middle of a **malloc** call, or accessing a complex data structure).

Let's learn about another way to handle signals by *waiting* for them in our program instead of using signal handlers.

Plan For Today

- Recap: Signals so far
- SIGCHLD Handlers
- **Demo:** Return Trip To Disneyland
- Concurrency Challenges
- **Waiting for Signals with sigwait and sigprocmask**

Waiting For Signals

- Signal handlers allow us to do other work and be notified when signals arrive. But this means the notification is unpredictable.
- A more predictable approach would be to **designate times in our program where we stop doing other work and handle any pending signals.**
  - benefits: this allows us to control when signals are handled, avoiding concurrency issues
  - drawbacks: signals may not be handled as promptly, and our process blocks while waiting
- We will not have signal handlers; instead we will have code in our main execution that handles pending signals.

Waiting For Signals

We will designate times in our program where we stop doing other work and handle any pending signals.

1. we need a way to handle pending signals
2. we need a way to turn on "do not disturb" for signals when we do not wish to handle them

Waiting For Signals

We will designate times in our program where we stop doing other work and handle any pending signals.

1. **we need a way to handle pending signals**
2. we need a way to turn on "do not disturb" for signals when we do not wish to handle them

sigwait()

sigwait() can be used to wait (block) on a signal to come in:

```
int sigwait(const sigset_t *set, int *sig);
```

- **set:** the location of the set of signals to wait on
- **sig:** the location where it should store the number of the signal received
- the return value is 0 on success, or > 0 on error.

Cannot wait on SIGKILL or SIGSTOP, nor synchronous signals like SIGSEGV or SIGFPE.

Signal Sets

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);
```

```
// Create a set of SIGINT and SIGTSTP
sigset_t monitoredSignals;
sigemptyset(&monitoredSignals);
sigaddset(&monitoredSignals, SIGINT);
sigaddset(&monitoredSignals, SIGTSTP);
```

sigwait()

Here's a program that overrides the behavior for Ctl-z to print a message instead:

```
1  int main(int argc, char *argv[]) {
2    sigset_t monitoredSignals;
3    sigemptyset(&monitoredSignals);
4    sigaddset(&monitoredSignals, SIGTSTP);
5
6    printf("Just try to Ctl-z me!\n");
7    while (true) {
8      int delivered;
9      sigwait(&monitoredSignals, &delivered);
10     printf("\nReceived signal %d: %s\n", delivered, strsignal(delivered));
11   }
12
13   return 0;
14 }
```

sigwait()

Problem: what if the user hits Ctl-z *before we reach line 9, or between **sigwait** calls? It won't be handled by our code!*

```
1  int main(int argc, char *argv[]) {
2    sigset_t monitoredSignals;
3    sigemptyset(&monitoredSignals);
4    sigaddset(&monitoredSignals, SIGTSTP);
5
6    printf("Just try to Ctl-z me!\n");
7    while (true) {
8      int delivered;
9      sigwait(&monitoredSignals, &delivered);
10     printf("\nReceived signal %d: %s\n", delivered, strsignal(delivered));
11   }
12
13   return 0;
14 }
```

sigwait.c

sigwait()

Problem: what if the user hits Ctl-z *before we reach line 9*, or between **sigwait** calls? *It won't be handled by our code!*

```
1  int main(int argc, char *argv[]) {
2    sigset_t monitoredSignals;
3    sigemptyset(&monitoredSignals);
4    sigaddset(&monitoredSignals, SIGTSTP);
5    sleep(2);
6
7    printf("Just try to Ctl-z me!\n");
8    while (true) {
9      int delivered;
10     sigwait(&monitoredSignals, &delivered);
11     printf("\nReceived signal %d: %s\n", delivered, strsignal(delivered));
12     sleep(2);
13   }
14
15   return 0;
16 }
```

This is a race condition: an unpredictable ordering of events where some orderings may cause undesired behavior.

Waiting For Signals

We will designate times in our program where we stop doing other work and handle any pending signals.

1. we need a way to handle pending signals
2. **we need a way to turn on "do not disturb" for signals when we do not wish to handle them**

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.

Side note: forked children inherit blocked signals! We may wish to remove a block in the child.

Do Not Disturb

Here's the same program from before, but blocking SIGTSTP as soon as possible:

```
 1 int main(int argc, char *argv[]) {
 2   sigset_t monitoredSignals;
 3   sigemptyset(&monitoredSignals);
 4   sigaddset(&monitoredSignals, SIGTSTP);
 5   sigprocmask(SIG_BLOCK, &monitoredSignals, NULL);
 6 
 7   printf("Just try to Ctl-z me!\n");
 8   while (true) {
 9     int delivered;
10     sigwait(&monitoredSignals, &delivered);
11     printf("\nReceived signal %d: %s\n", delivered, strsignal(delivered));
12   }
13 
14   return 0;
15 }
```

Wait - if we call **sigwait** while signals are blocked, what happens?

Key insight: sigwait() doesn't care about blocked signals when it is called.

Recap

- Recap: Signals so far
- SIGCHLD Handlers
- **Demo:** Return Trip To Disneyland
- Concurrency Challenges
- Waiting for Signals with **sigwait** and **sigprocmask**

Next time: multiprocessing wrap-up and virtual memory