CS110 Lecture 10: Signals, Part 1

CS110: Principles of Computer Systems

Winter 2021-2022

Stanford University

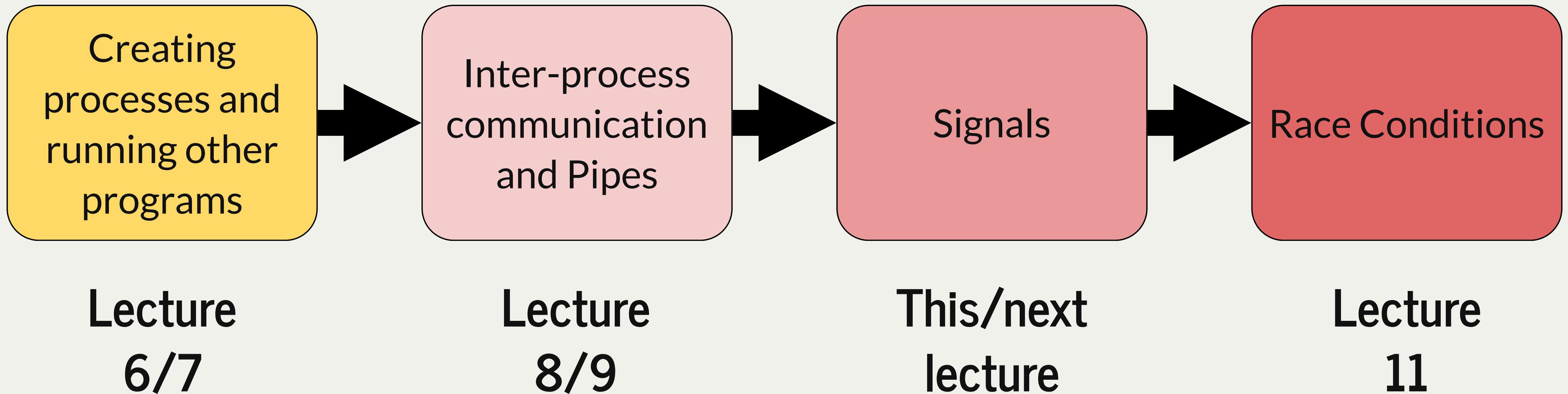
Instructors: Nick Troccoli and Jerry Cain

#	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 *signals* as another way for processes to communicate
- Gain practice with how to execute code in our program when we receive a signal

Plan For Today

- Revisiting I/O Redirection
- Introducing Signals
- **Demo**: Disneyland
- Signals Aren't Queued

Plan For Today

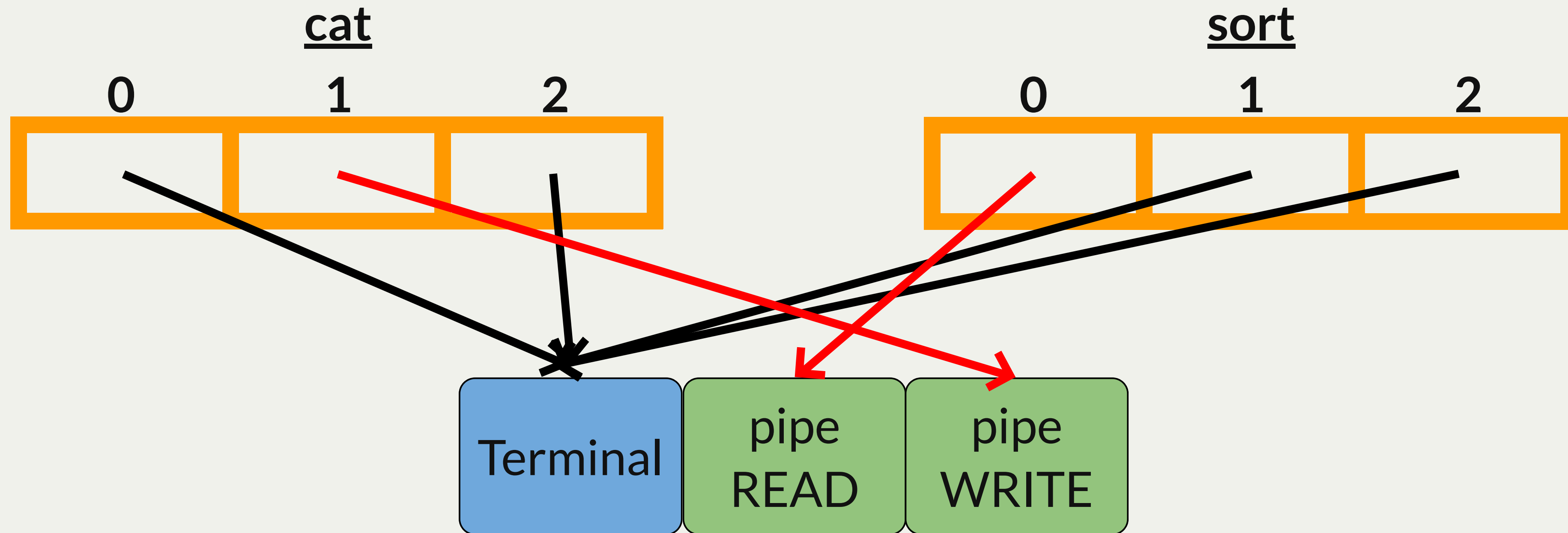
- **Revisiting I/O Redirection**
- Introducing Signals
- **Demo**: Disneyland
- Signals Aren't Queued

read() Clarifications

- **read()** waits if no bytes are available but they could be written later.
- **read()** *doesn't wait* if there are bytes available but it's less than we asked for.

Pipeline

I/O redirection and pipes allow us to handle piping in our shell: e.g. **cat file.txt | sort**



pipeline

Last time, we implemented a custom function called **pipeline**.

```
void pipeline(char *argv1[], char *argv2[], pid_t pids[]);
```

pipeline is similar to **subprocess**, except it also spawns a second child and directs its STDOUT to write to the pipe. Both children should run in parallel.

It doesn't return anything, but it writes the two children PIDs to the specified **pids** array

```c
void pipeline(char *argv1[], char *argv2[], pid_t pids[]) {
  int fds[2];
  pipe(fds);

  // Spawn the first child
  pids[0] = fork();
  if (pids[0] == 0) {
    // The first child's STDOUT should be the write end of the pipe
    close(fds[0]);
    dup2(fds[1], STDOUT_FILENO);
    close(fds[1]);
    execvp(argv1[0], argv1);
  }

  // We no longer need the write end of the pipe
  close(fds[1]);

  // Spawn the second child
  pids[1] = fork();
  if (pids[1] == 0) {
    // The second child's STDIN should be the read end of the pipe
    dup2(fds[0], STDIN_FILENO);
    close(fds[0]);
    execvp(argv2[0], argv2);
  }

  // We no longer need the read end of the pipe
  close(fds[0]);
}
```

pipeline-soln.c

pipe2

There were a lot of **close()** calls! Is there a way for any of them to be done automatically?

```
int pipe2(int fds[], int flags);
```

pipe2 is the same as **pipe** except it lets you customize the pipe with some optional flags.

- if flags is 0, it's the same as **pipe**
- if flags is **O_CLOEXEC**, the pipe FDs will *be automatically closed when the surrounding process calls **execvp**.*

pipeline

```
 1 void pipeline(char *argv1[], char *argv2[], pid_t pids[]) {
 2   int fds[2];
 3   pipe(fds);
 4 
 5   pids[0] = fork();
 6   if (pids[0] == 0) {
 7     close(fds[0]);
 8     dup2(fds[1], STDOUT_FILENO);
 9     close(fds[1]);
10     execvp(argv1[0], argv1);
11   }
12 
13   close(fds[1]);
14 
15   pids[1] = fork();
16   if (pids[1] == 0) {
17     dup2(fds[0], STDIN_FILENO);
18     close(fds[0]);
19     execvp(argv2[0], argv2);
20   }
21 
22   close(fds[0]);
23 }
```

The highlighted calls to **close()** would no longer be necessary if we use **pipe2** with **O_CLOEXEC** because the surrounding process for each calls **execvp**.

Note that the parent must still close them because it doesn't call execvp.

pipeline with pipe2

```
 1 void pipeline(char *argv1[], char *argv2[], pid_t pids[]) {
 2   int fds[2];
 3   pipe2(fds, O_CLOEXEC);
 4 
 5   pids[0] = fork();
 6   if (pids[0] == 0) {
 7     dup2(fds[1], STDOUT_FILENO);
 8     execvp(argv1[0], argv1);
 9   }
10 
11   close(fds[1]);
12 
13   pids[1] = fork();
14   if (pids[1] == 0) {
15     dup2(fds[0], STDIN_FILENO);
16     execvp(argv2[0], argv2);
17   }
18 
19   close(fds[0]);
20 }
```

This version of pipeline uses **pipe2** with **O_CLOEXEC.**

Pipes and I/O Redirection: Key Takeaways

- Pipes are sets of file descriptors that allow us to communicate across processes.
- Processes can share these file descriptors because they are copied on **fork()**
- File descriptors 0,1 and 2 are special and assumed to represent STDIN, STDOUT and STDERR
- If we change those file descriptors to point to other resources, we can redirect STDIN/STDOUT/STDERR to be something else without the program knowing!
- Pipes are how terminal support for piping and redirection (**command1 | command2** and **command1 > file.txt**) are implemented!

Plan For Today

- Revisiting I/O Redirection
- **Introducing Signals**
- **Demo**: Disneyland
- Signals Aren't Queued

Interprocess Communication

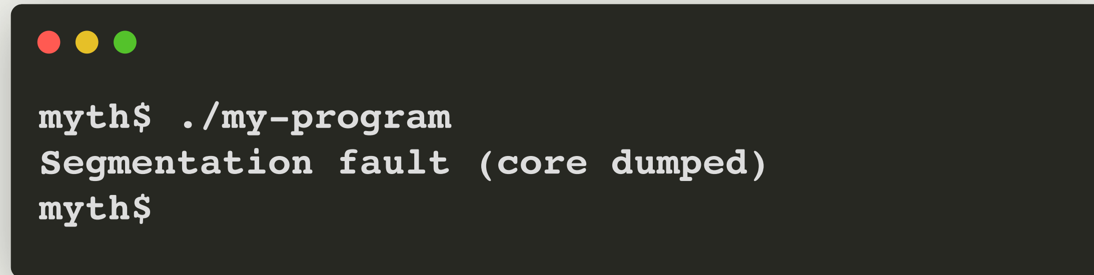
- It's useful for a parent process to be able to communicate with its child (and vice versa)
- There are two key ways we will learn to do this: **pipes** and **signals**
  - **Pipes** let two processes send and receive arbitrary data
  - **Signals** let two processes send and receive certain "signals"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

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. SIGINT is 2)
- A program can do something in response to a type of signal being 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



A **segmentation fault** is actually a signal (SIGSEGV) sent from the OS to your program.

- triggered when you try to access a memory address not in a valid program *segment*
- default behavior is to terminate the program

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.

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

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 abs(pid)), or 0/-1 (we ignore these).
- **raise** sends the specified signal to yourself

Parent/Child Ping Pong Example

https://cplayground.com/embed?p=hornet-impala-fly

Receiving Signals

There are two main ways we can respond to signals we have received:

- add *signal handlers* to our program: functions that run when a certain signal is received
- we can *block* in our program until a signal is received

Signal handlers are versatile but fraught with potential issues. We will learn about them to motivate the second approach (blocking until signal is received).

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

```c
static void handleSIGINT(int sig) {
  printf("Sigint received!\n");
}

int main(int argc, char *argv[]) {
  signal(SIGINT, handleSIGINT);
  printf("Just try to interrupt me!\n");
  while (true) {
    sleep(1);
  }
  return 0;
}
```

- The handler must be a function that returns nothing and takes in an int (the signal).
- A signal handler overrides the default behavior for that signal (if any).

sigint.c

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!

Plan For Today

- Revisiting I/O Redirection
- Introducing Signals
- **Demo: Disneyland**
- Signals Aren't Queued

SIGCHLD Example: Disneyland

Let's write a program where a parent spawns off five children to go play, and does something else (sleeps 😴) until all the children are done.

```
1  static const size_t kNumChildren = 5;
2
3  int main(int argc, char *argv[]) {
4    printf("Let my five children play while I take a nap.\n");
5
6    for (size_t kid = 1; kid <= kNumChildren; kid++) {
7      if (fork() == 0) {
8        sleep(3 * kid); // sleep emulates "play" time
9        printf("Child #%zu tired... returns to parent.\n", kid);
10       return 0;
11     }
12   }
13
14   // parent goes and does other work
15   snooze(5); // custom fn to sleep uninterrupted
16
17   return 0;
18 }
```

- Similar to many parallel data processing applications where parent does other work while children are busy
- **Problem: how do we clean up the child processes?**

five-children-soln.c

We can add a SIGCHLD handler in the parent that cleans up the child that terminated!

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 }
```

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!

SIGCHLD Example: Disneyland

```
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

- Revisiting I/O Redirection
- Introducing Signals
- **Demo**: Disneyland
- **Signals Aren't Queued**

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.

Recap

- Revisiting I/O Redirection
- Introducing Signals
- **Demo**: Disneyland
- Signals Aren't Queued

Next time: more signal handlers and another approach to signals