CS110 Lecture 06: Pipes, Signals, and Concurrency

Principles of Computer Systems

Winter 2020

Stanford University

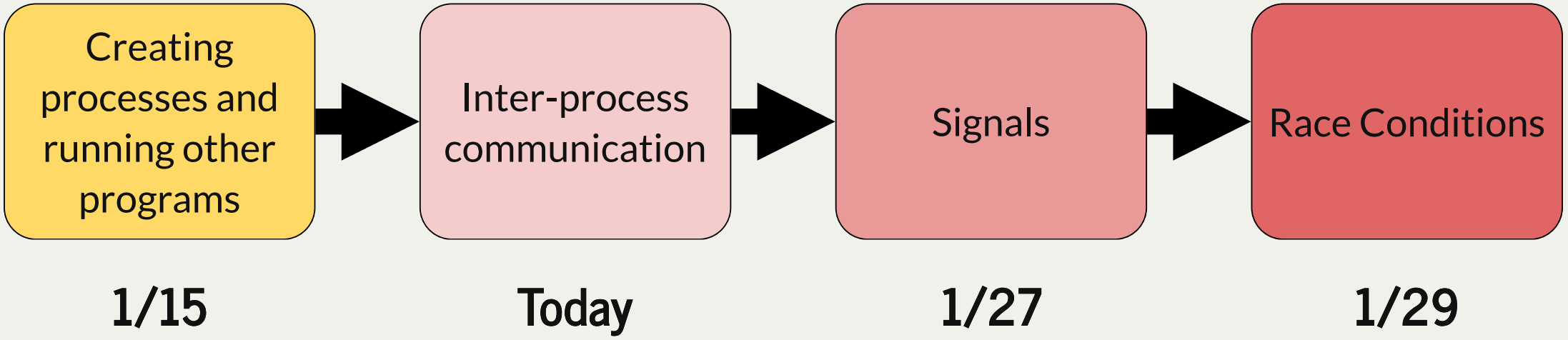
Computer Science Department

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

- Get more practice with using **fork()** and **execvp**
- Learn about **pipe** and **dup2** to create and manipulate file descriptors
- Introduce *signals* as another way for processes to communicate

Plan For Today

- Review: **fork()** and **execvp()**
- ***Practice:*** Revisiting **`first-shell`**
- Running in the background
- **Break:** Announcements
- Introducing Pipes
- ***Practice:*** Implementing **subprocess**
- Introducing Signals
- **Demo**: Disneyland

fork()

- A system call that creates a new *child process*
- The "parent" is the process that creates the other "child" process
- From then on, both processes are running the code after the fork
- The child process is *identical* to the parent, except:
  - it has a new Process ID (PID)
  - for the parent, fork() returns the PID of the child; for the child, fork() returns 0
  - fork() is **called once**, but **returns twice**

```
1 pid_t pidOrZero = fork();
2 // both parent and child run code here onwards
3 printf("This is printed by two processes.\n");
```

waitpid()

A function that a parent can call to wait for its child to exit:

```
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 termination (or NULL)
- **options:** optional flags to customize behavior (always 0 for now)

The function returns when the specified **child process** exits.

- the return value is the PID of the child that exited, or -1 on error (e.g. no child to wait on)
- If the child process has already exited, this returns immediately - otherwise, it blocks
- It's important to wait on all children to clean up system resources

execvp()

`execvp` is a function that lets us run *another program* in the current process.

```
int execvp(const char *path, char *argv[]);
```

It runs the specified program executable, *completely cannibalizing the current process.*

- **`path`** identifies the name of the executable to be invoked.
- **`argv`** is the argument vector that should be passed to the new executable's **`main`** function.
- For the purposes of CS110, **`path`** and **`argv[0]`** end up being the same exact string.
- If **`execvp`** fails to cannibalize the process and install a new executable image within it, it returns -1 to express failure.
- If **`execvp`** succeeds, it **never returns** in the calling process.
- **`execvp`** has many variants (**`execle`**, **`execlp`**, and so forth. Type **`man execvp`** to see all of them). We generally rely on **`execvp`** in this course.

execvp() Example

What does the following code output, assuming **execvp** executes successfully?

```
1 int main(int argc, char *argv[]) {
2     char *args[] = {"/bin/ls", "-l", "/usr/class/cs110", NULL};
3     execvp(args[0], args);
4     printf("Hello world!\n");
5     return 0;
6 }
```

execvp() Example

What does the following code output, assuming **execvp** executes successfully?

```
1 int main(int argc, char *argv[]) {
2     char *args[] = {"/bin/ls", "-l", "/usr/class/cs110", NULL};
3     execvp(args[0], args);
4     printf("Hello world!\n");
5     return 0;
6 }
```

- This process will be *completely consumed* by the new program being run (**ls**).

execvp() Example

What does the following code output, assuming **execvp** executes successfully?

```
1 int main(int argc, char *argv[]) {
2     char *args[] = {"/bin/ls", "-l", "/usr/class/cs110", NULL};
3     execvp(args[0], args);
4     printf("Hello world!\n");
5     return 0;
6 }
```

- This process will be *completely consumed* by the new program being run (**ls**).
- Lines 4+ will *never execute* unless an error occurs in **execvp**.

Plan For Today

- Review: **fork()** and **execvp()**
- ***Practice:*** Revisiting `first-shell`
- Running in the background
- **Break:** Announcements
- Introducing Pipes
- ***Practice:*** Implementing **subprocess**
- Introducing Signals
- **Demo**: Disneyland

Revisiting *mysystem*

mysystem is our own version of the built-in function **system**.

- It takes in a terminal command (e.g. "ls -l /usr/class/cs110"), executes it in a separate process, and returns when that process is finished.
  - We can use **fork** to create the child process
  - We can use **execvp** in that child process to execute the terminal command
  - We can use **waitpid** in the parent process to wait for the child to terminate

Revisiting *mysystem*

```
 1 static int mysystem(char *command) {
 2     pid_t pidOrZero = fork();
 3     if (pidOrZero == 0) {
 4         // If we are the child, execute the shell command
 5         char *arguments[] = {"/bin/sh", "-c", command, NULL};
 6         execvp(arguments[0], arguments);
 7         // If the child gets here, there was an error
 8         exitIf(true, kExecFailed, stderr, "execvp failed to invoke this: %s.\n", command);
 9     }
10
11     // If we are the parent, wait for the child
12     int status;
13     waitpid(pidOrZero, &status, 0);
14     if (WIFEXITED(status)) {
15         return WEXITSTATUS(status);
16     } else {
17         return -WTERMSIG(status);
18     }
19 }
```

Revisiting *mysystem*

```
1  static int mysystem(char *command) {
2      pid_t pidOrZero = fork();
3      if (pidOrZero == 0) {
4          // If we are the child, execute the shell command
5          char *arguments[] = {"/bin/sh", "-c", command, NULL};
6          execvp(arguments[0], arguments);
7          // If the child gets here, there was an error
8          exitIf(true, kExecFailed, stderr, "execvp failed to invoke this: %s.\n", command);
9      }
10
11     // If we are the parent, wait for the child
12     int status;
13     waitpid(pidOrZero, &status, 0);
14     if (WIFEXITED(status)) {
15         return WEXITSTATUS(status);
16     } else {
17         return -WTERMSIG(status);
18     }
19 }
```

Line 2: First, fork off a child process.

Revisiting *mysystem*

```
1  static int mysystem(char *command) {
2      pid_t pidOrZero = fork();
3      if (pidOrZero == 0) {
4          // If we are the child, execute the shell command
5          char *arguments[] = {"/bin/sh", "-c", command, NULL};
6          execvp(arguments[0], arguments);
7          // If the child gets here, there was an error
8          exitIf(true, kExecFailed, stderr, "execvp failed to invoke this: %s.\n", command);
9      }
10
11     // If we are the parent, wait for the child
12     int status;
13     waitpid(pidOrZero, &status, 0);
14     if (WIFEXITED(status)) {
15         return WEXITSTATUS(status);
16     } else {
17         return -WTERMSIG(status);
18     }
19 }
```

Lines 4-6: In the child, execute the **/bin/sh** program, which can execute any shell command.

Revisiting *mysystem*

```
1  static int mysystem(char *command) {
2      pid_t pidOrZero = fork();
3      if (pidOrZero == 0) {
4          // If we are the child, execute the shell command
5          char *arguments[] = {"/bin/sh", "-c", command, NULL};
6          execvp(arguments[0], arguments);
7          // If the child gets here, there was an error
8          exitIf(true, kExecFailed, stderr, "execvp failed to invoke this: %s.\n", command);
9      }
10
11     // If we are the parent, wait for the child
12     int status;
13     waitpid(pidOrZero, &status, 0);
14     if (WIFEXITED(status)) {
15         return WEXITSTATUS(status);
16     } else {
17         return -WTERMSIG(status);
18     }
19 }
```

Line 8: The child will only get to this line if execvp fails.

Revisiting *mysystem*

```
 1 static int mysystem(char *command) {
 2     pid_t pidOrZero = fork();
 3     if (pidOrZero == 0) {
 4         // If we are the child, execute the shell command
 5         char *arguments[] = {"/bin/sh", "-c", command, NULL};
 6         execvp(arguments[0], arguments);
 7         // If the child gets here, there was an error
 8         exitIf(true, kExecFailed, stderr, "execvp failed to invoke this: %s.\n", command);
 9     }
10
11     // If we are the parent, wait for the child
12     int status;
13     waitpid(pidOrZero, &status, 0);
14     if (WIFEXITED(status)) {
15         return WEXITSTATUS(status);
16     } else {
17         return -WTERMSIG(status);
18     }
19 }
```

Lines 11-13: In the parent, wait for the child to terminate.

Revisiting *mysystem*

```
1  static int mysystem(char *command) {
2      pid_t pidOrZero = fork();
3      if (pidOrZero == 0) {
4          // If we are the child, execute the shell command
5          char *arguments[] = {"/bin/sh", "-c", command, NULL};
6          execvp(arguments[0], arguments);
7          // If the child gets here, there was an error
8          exitIf(true, kExecFailed, stderr, "execvp failed to invoke this: %s.\n", command);
9      }
10
11     // If we are the parent, wait for the child
12     int status;
13     waitpid(pidOrZero, &status, 0);
14     if (WIFEXITED(status)) {
15         return WEXITSTATUS(status);
16     } else {
17         return -WTERMSIG(status);
18     }
19 }
```

Lines 14-18: In the parent, after the child terminates, return its status.

Revisiting *first-shell*

```
 1 int main(int argc, char *argv[]) {
 2     char command[kMaxLineLength];
 3     while (true) {
 4         printf("> ");
 5         fgets(command, sizeof(command), stdin);
 6
 7         // If the user entered Ctl-d, stop
 8         if (feof(stdin)) {
 9             break;
10         }
11
12         // Remove the \n that fgets puts at the end
13         command[strlen(command) - 1] = '\0';
14
15         int commandReturnCode = mysystem(command);
16         printf("return code = %d\n", commandReturnCode);
17     }
18
19     printf("\n");
20     return 0;
21 }
```

Our **first-shell** program is a loop in **main** that parses the user input and passes it to **mysystem**.

first-shell Takeaways

- A shell is a program that repeats: read command from the user, execute that command
- In order to execute a program and continue running the shell afterwards, we fork off another process and run the program in that process
- We rely on **`fork`**, **`execvp`**, and **`waitpid`** to do this!
- Real shells have more advanced functionality that we will add going forward.
- For your fourth assignment, you'll build on this with your own shell, **`stsh`** ("Stanford shell") with much of the functionality of real Unix shells.

More Shell Functionality

Shells have a variety of supported commands:

- **emacs &** - create an emacs process and run it in the background
- **cat file.txt | uniq | sort** - pipe the output of one command to the input of another
- **uniq < file.txt | sort > list.txt** - make file.txt the input of uniq and output sort to list.txt
- Let's see how we can implement these - but first, a demo.

Plan For Today

- Review: **fork()** and **execvp()**
- ***Practice:*** Revisiting **`first-shell`**
- Running in the background
- **Break:** Announcements
- Introducing Pipes
- ***Practice:*** Implementing **subprocess**
- Introducing Signals
- **Demo**: Disneyland

Supporting Background Execution

```
1  static void executeCommand(char *command, bool inBackground) {
2      pid_t pidOrZero = fork();
3      if (pidOrZero == 0) {
4          // If we are the child, execute the shell command
5          char *arguments[] = {"/bin/sh", "-c", command, NULL};
6          execvp(arguments[0], arguments);
7          // If the child gets here, there was an error
8          exitIf(true, kExecFailed, stderr, "execvp failed to invoke this: %s.\n", command);
9      }
10
11     // If we are the parent, either wait or return immediately
12     if (inBackground) {
13         printf("%d %s\n", pidOrZero, command);
14     } else {
15         waitpid(pidOrZero, NULL, 0);
16     }
17 }
```

Supporting Background Execution

```
1  static void executeCommand(char *command, bool inBackground) {
2      pid_t pidOrZero = fork();
3      if (pidOrZero == 0) {
4          // If we are the child, execute the shell command
5          char *arguments[] = {"/bin/sh", "-c", command, NULL};
6          execvp(arguments[0], arguments);
7          // If the child gets here, there was an error
8          exitIf(true, kExecFailed, stderr, "execvp failed to invoke this: %s.\n", command);
9      }
10
11     // If we are the parent, either wait or return immediately
12     if (inBackground) {
13         printf("%d %s\n", pidOrZero, command);
14     } else {
15         waitpid(pidOrZero, NULL, 0);
16     }
17 }
```

Line 1: Now, the caller can optionally run the command in the background.

Supporting Background Execution

```
 1 static void executeCommand(char *command, bool inBackground) {
 2     pid_t pidOrZero = fork();
 3     if (pidOrZero == 0) {
 4         // If we are the child, execute the shell command
 5         char *arguments[] = {"/bin/sh", "-c", command, NULL};
 6         execvp(arguments[0], arguments);
 7         // If the child gets here, there was an error
 8         exitIf(true, kExecFailed, stderr, "execvp failed to invoke this: %s.\n", command);
 9     }
10
11     // If we are the parent, either wait or return immediately
12     if (inBackground) {
13         printf("%d %s\n", pidOrZero, command);
14     } else {
15         waitpid(pidOrZero, NULL, 0);
16     }
17 }
```

Lines 11-16: The parent waits on a foreground child, but not a background child.

Supporting Background Execution

```
 1 int main(int argc, char *argv[]) {
 2     char command[kMaxLineLength];
 3     while (true) {
 4         printf("> ");
 5         fgets(command, sizeof(command), stdin);
 6
 7         // If the user entered Ctl-d, stop
 8         if (feof(stdin)) {
 9             break;
10         }
11
12         // Remove the \n that fgets puts at the end
13         command[strlen(command) - 1] = '\0';
14
15         if (strcmp(command, "quit") == 0) break;
16
17         bool isbg = command[strlen(command) - 1] == '&';
18         if (isbg) {
19             command[strlen(command) - 1] = '\0';
20         }
21
22         executeCommand(command, isbg);
23     }
24
25     printf("\n");
26     return 0;
27 }
```

In main, on lines 15-22, we check for the "quit" command, and also for whether to run the command in the background.

Plan For Today

- Review: **fork()** and **execvp()**
- ***Practice:*** Revisiting **`first-shell`**
- Running in the background
- **Break:** Announcements
- Introducing Pipes
- ***Practice:*** Implementing **subprocess**
- Introducing Signals
- **Demo**: Disneyland

Announcements

- Assign2 due tomorrow at 11:59PM PST
- Assign3 goes out tomorrow - all about multiprocessing
  - This Monday's lecture needed for the last part
- Section 2 starts tomorrow
  - previous week's section solutions released tomorrow

Mid-Lecture Checkin

Now we can answer the following questions:

- when writing a shell, why is it essential to call execvp in the child process?
- how can we update our shell to support background execution of commands?

Plan For Today

- Review: **fork()** and **execvp()**
- ***Practice:*** Revisiting **`first-shell`**
- Running in the background
- **Break:** Announcements
- Introducing Pipes
- ***Practice:*** Implementing **subprocess**
- Introducing Signals
- **Demo**: Disneyland

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.

Pipes

- How can we let two processes send arbitrary data back and forth?
- A core Unix principle is how many things can be modeled as *files*. Could we use a "file"?
- **Idea:** what if we used a file that one process could write to, and another process could read from?
- **Problem:** we don't want to clutter the filesystem with actual files every time two processes want to communicate.
- **Solution:** have the operating system set this up for us.
  - It will give us two new file descriptors - one for writing, another for reading.
  - If someone writes data to the write FD, it can be read from the read FD.
  - It's *not actually a physical file on disk* - we are just using files as an abstraction

pipe()

```
int pipe(int fds[]);
```

- The **`pipe`** system call takes an uninitialized array of two integers and populates it with two file descriptors such that everything *written* to **`fds[1]`** can be *read* from **`fds[0]`**.
- **`pipe`** can allow parent processes to communicate with spawned child processes.
  - Because they're file descriptors, there's no global name for the pipe (another process can't "connect" to the pipe)
  - The parent's table is replicated in the child, so the child automatically gets access to the same file descriptors

pipe()

Here's an example program showing how pipe works (which you can find here):

```
 1 int main(int argc, char *argv[]) {
 2   int fds[2];
 3   pipe(fds);
 4   pid_t pid = fork();
 5   if (pid == 0) {
 6     close(fds[1]);
 7     char buffer[6];
 8     read(fds[0], buffer, sizeof(buffer));
 9     printf("Read from pipe bridging processes: %s.\n", buffer)
10     close(fds[0]);
11     return 0;
12   }
13   close(fds[0]);
14   write(fds[1], "hello", 6);
15   waitpid(pid, NULL, 0);
16   close(fds[1]);
17   return 0;
18 }
```

pipe()

Here's an example program showing how pipe works (which you can find here):

```
1  int main(int argc, char *argv[]) {
2    int fds[2];
3    pipe(fds);
4    pid_t pid = fork();
5    if (pid == 0) {
6      close(fds[1]);
7      char buffer[6];
8      read(fds[0], buffer, sizeof(buffer));
9      printf("Read from pipe bridging processes: %s.\n", buffer)
10     close(fds[0]);
11     return 0;
12   }
13   close(fds[0]);
14   write(fds[1], "hello", 6);
15   waitpid(pid, NULL, 0);
16   close(fds[1]);
17   return 0;
18 }
```

Lines 2-3: We ask the operating system to create a pipe for us. This gives us two file descriptors, one for reading and one for writing.

Tip: you learn to read before you learn to write (read = fds[0], write = fds[1]).

pipe()

Here's an example program showing how pipe works (which you can find here):

```
 1 int main(int argc, char *argv[]) {
 2   int fds[2];
 3   pipe(fds);
 4   pid_t pid = fork();
 5   if (pid == 0) {
 6     close(fds[1]);
 7     char buffer[6];
 8     read(fds[0], buffer, sizeof(buffer));
 9     printf("Read from pipe bridging processes: %s.\n", buffer)
10     close(fds[0]);
11     return 0;
12   }
13   close(fds[0]);
14   write(fds[1], "hello", 6);
15   waitpid(pid, NULL, 0);
16   close(fds[1]);
17   return 0;
18 }
```

Lines 13-14: after forking, in the parent we close the reader FD, and write to the writer FD to send a message to the child.

pipe()

Here's an example program showing how pipe works (which you can find here):

```
 1 int main(int argc, char *argv[]) {
 2   int fds[2];
 3   pipe(fds);
 4   pid_t pid = fork();
 5   if (pid == 0) {
 6     close(fds[1]);
 7     char buffer[6];
 8     read(fds[0], buffer, sizeof(buffer));
 9     printf("Read from pipe bridging processes: %s.\n", buffer)
10     close(fds[0]);
11     return 0;
12   }
13   close(fds[0]);
14   write(fds[1], "hello", 6);
15   waitpid(pid, NULL, 0);
16   close(fds[1]);
17   return 0;
18 }
```

Lines 15-16: after sending a message, we wait for the child to receive it and terminate. Then we clean it up and close the writer FD.

pipe()

Here's an example program showing how pipe works (which you can find here):

```
1  int main(int argc, char *argv[]) {
2    int fds[2];
3    pipe(fds);
4    pid_t pid = fork();
5    if (pid == 0) {
6      close(fds[1]);
7      char buffer[6];
8      read(fds[0], buffer, sizeof(buffer));
9      printf("Read from pipe bridging processes: %s.\n", buffer)
10     close(fds[0]);
11     return 0;
12   }
13   close(fds[0]);
14   write(fds[1], "hello", 6);
15   waitpid(pid, NULL, 0);
16   close(fds[1]);
17   return 0;
18 }
```

Line 4: when we fork, the child gets an *identical copy* of the parent's file descriptor table. This means its file descriptor table entries point to the same open file table entries.

This means the open file table entries for the two pipe FDs both have reference counts of 2.

pipe()

Here's an example program showing how pipe works (which you can find here):

```
1  int main(int argc, char *argv[]) {
2    int fds[2];
3    pipe(fds);
4    pid_t pid = fork();
5    if (pid == 0) {
6      close(fds[1]);
7      char buffer[6];
8      read(fds[0], buffer, sizeof(buffer));
9      printf("Read from pipe bridging processes: %s.\n", buffer)
10     close(fds[0]);
11     return 0;
12   }
13   close(fds[0]);
14   write(fds[1], "hello", 6);
15   waitpid(pid, NULL, 0);
16   close(fds[1]);
17   return 0;
18 }
```

Lines 6-8: after forking, the child closes the writer FD and reads 6 bytes from the reader FD.

pipe()

Here's an example program showing how pipe works (which you can find here):

```
1  int main(int argc, char *argv[]) {
2    int fds[2];
3    pipe(fds);
4    pid_t pid = fork();
5    if (pid == 0) {
6      close(fds[1]);
7      char buffer[6];
8      read(fds[0], buffer, sizeof(buffer));
9      printf("Read from pipe bridging processes: %s.\n", buffer);
10     close(fds[0]);
11     return 0;
12   }
13   close(fds[0]);
14   write(fds[1], "hello", 6);
15   waitpid(pid, NULL, 0);
16   close(fds[1]);
17   return 0;
18 }
```

Lines 9-11: after reading data, it prints it to the screen, closes the reader FD, and terminates.

pipe (man page section 2) code example

https://cplayground.com/?p=okapi-grasshopper-bear

pipe()

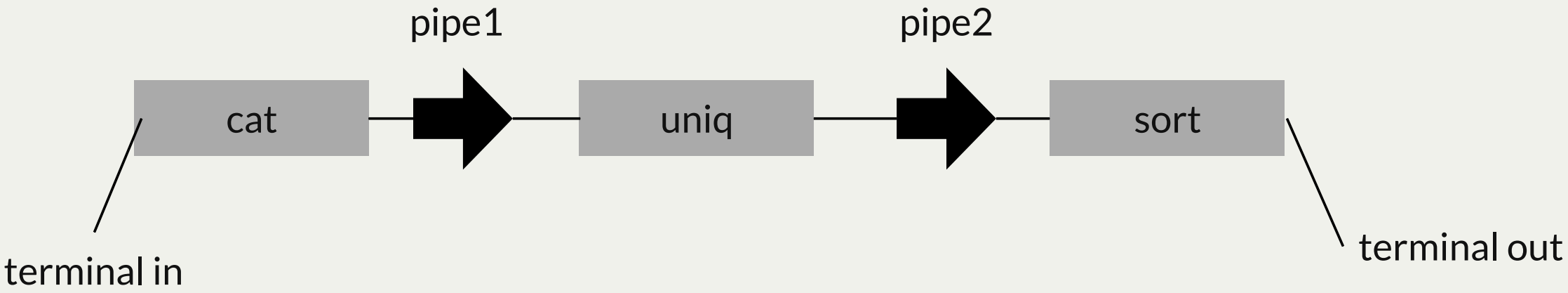
This method of communication between processes relies on the fact that file descriptors are duplicated when forking.

- each process has its own copy of both file descriptors for the pipe
- each process must therefore close both file descriptors for the pipe when finished
- both processes could read or write to the pipe if they wanted.

pipe()

This is how a shell can support piping between processes (e.g. **cat file.txt | uniq | sort**):

- Shell creates three child processes: cat, uniq and sort
- Shell creates two pipes: one between cat and uniq, one between uniq and sort




```
int pipe1[2];
int pipe2[2];
pipe(pipe1);
pipe(pipe2);
```

Process	stdin	stdout
cat	terminal	pipe1[1]
uniq	pipe1[0]	pipe2[1]
sort	pipe2[0]	terminal

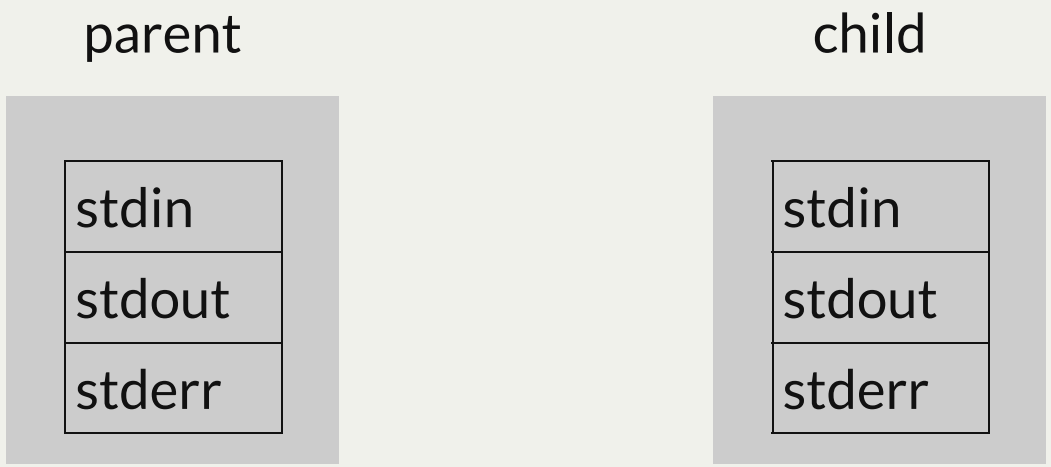
combining pipe() and dup2()

- Using **`pipe`**, **`fork`**, **`dup2`**, **`execvp`**, **`close`**, and **`waitpid`**, we can implement the **`subprocess`** function, which spawns a child process that we can communicate with (full implementation of everything is right here).
- Same as **mysystem**, but now we can send a message to the child that it can read *via STDIN*
- Let's see how we can do this.

subprocess File Descriptor Diagram

All processes are configured with FDs 0-2 for STDIN, STDOUT and STDERR, respectively.

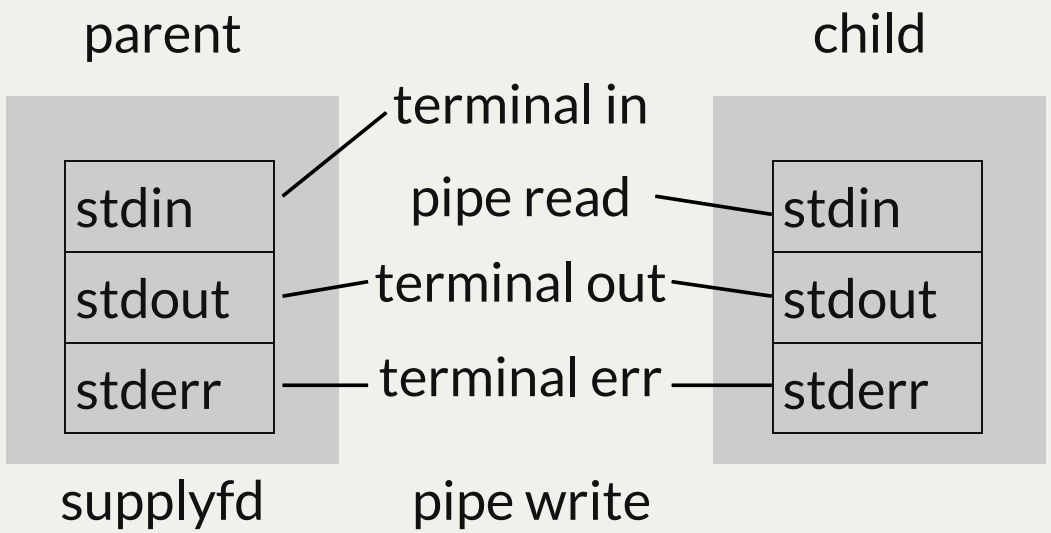
- Processes assume these indexes are for these methods of communication
- We can *change* one of these file descriptor table entries to change where its STDIN, STDOUT or STERR read from/write to!



subprocess File Descriptor Diagram

dup2 lets us make a copy of a file descriptor entry and put it in another file descriptor index. If the second parameter is an already-open file descriptor, it is closed before being used. We can use **`dup2`** to copy the pipe read file descriptor into child's standard input!

```
dup2(fds[0], STDIN_FILENO);
```



Demo: subprocess

Questions about pipes?

Plan For Today

- Review: **fork()** and **execvp()**
- ***Practice:*** Revisiting **`first-shell`**
- Running in the background
- **Break:** Announcements
- Introducing Pipes
- ***Practice:*** Implementing **subprocess**
- Introducing Signals
- **Demo**: Disneyland

UNIX Signals

- A **signal** is a way to notify a process that an event occurred.
  - The kernel sends many signals (SIGSEGV, SIGBUS, SIGINT, ...)
    - Everyone who's programmed in C has unintentionally dereferenced a **`NULL`** pointer.
    - The kernel delivers a **`SIGSEGV`**, informally known as a segmentation fault (or a **SEG**mentation **V**iolation, or **`SIGSEGV`**, for short).
    - Unless you install a custom signal handler to manage the signal differently, a **`SIGSEGV`** terminates the program and generates a core dump.
  - Processes can send each other signals as well (SIGSTOP, SIGKILL)
- A **signal handler** is a function that executes when the signal arrives
  - Some signals have default handler(e.g., SIGSEGV terminates process and dumps core)
  - You can install custom handlers for most signals
- Each signal is represented internally by some number (e.g. **`SIGSEGV`** is 11).

Some Signals

- **`SIGFPE`**: whenever a process commits an integer-divide-by-zero (and, in some cases, a floating-point divide by zero on older architectures), the kernel hollers and issues a **`SIGFPE`** signal to the offending process. By default, the program handles the **`SIGFPE`** by printing an error message announcing the zero denominator and generating a core dump.
- **`SIGINT`**: when you type ctrl-c, the kernel sends a **`SIGINT`** to the foreground process group. The default handler terminates the process group.
- **`SIGTSTP`**: when you type ctrl-z, the kernel issues a **`SIGTSTP`** to the foreground process group. The foreground process group is halted until a **`SIGCONT`** signal.
- **`SIGPIPE`**: when a process attempts to write data to a pipe after the read end has closed, the kernel delivers a **`SIGPIPE`**. The default **`SIGPIPE`** handler prints a message identifying the pipe error and terminates the program.

A Systems Mystery

```
$ grep error file.txt > errors.txt &
[1] 4287
$
[1]+ Done                grep error file.txt > errors.txt
```

- How does this work?
  - The shell returns control to the user after forking the child (not calling waitpid on the child)
  - But the shell still knows when the child completes
- There must be a way for the shell to learn about when things have happened to its children

SIGCHLD

- Whenever a child process **changes state**—that is, it exits, crashes, stops, or resumes from a stopped state, the kernel sends a **`SIGCHLD`** signal to the process's parent.
  - By default, the signal is ignored. We've ignored it until right now and gotten away with it.
- This particular signal type is instrumental to allowing forked child processes to run in the background while keeping the parent immediately aware of when something happens.
- Custom **`SIGCHLD`** handlers can call **`waitpid`**, which tells them the pids of child processes that gave changed state. If the child process terminated, either normally or abnormally, the **`waitpid`** also cleans up/frees the child.

Signals at Disneyland

- Here's an example of when you might want to use a **`SIGCHLD`** handler.
- The premise? Dad takes his five kids 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.
  - If Dad has stuff to do (rather than nap), this is a very simple analogy to many parallel data processing applications (if Dad only naps just call **`wait`**)
- The parent is dad, and subprocesses are children. (Full program is right here.)

```
1  static const size_t kNumChildren = 5;
2  static size_t numDone = 0;
3
4  int main(int argc, char *argv[]) {
5    printf("Let my five children play while I take a nap.\n");
6    signal(SIGCHLD, reapChild);
7    for (size_t kid = 1; kid <= 5; kid++) {
8      if (fork() == 0) {
9        sleep(3 * kid); // sleep emulates "play" time
10       printf("Child #%zu tired... returns to dad.\n", kid);
11       return 0;
12     }
13   }
```

Signals at Disneyland

- Our first signal handler example: Disneyland
  - The program is crafted so each child process exits at three-second intervals. **`reapChild`**, handles each of the **`SIGCHLD`** signals delivered as each child process exits.
  - The **`signal`** prototype doesn't allow for state to be shared via parameters, so we have no choice but to use global variables.

```
1   // code below is a continuation of that presented on the previous slide
2   while (numDone < kNumChildren) {
3     printf("At least one child still playing, so dad nods off.\n");
4     sleep(5);
5     printf("Dad wakes up! ");
6   }
7   printf("All children accounted for.  Good job, dad!\n");
8   return 0;
9 }
10
11 static void reapChild(int unused) {
12   waitpid(-1, NULL, 0);
13   numDone++;
14 }
```

Signals at Disneyland

- Here's the output of the above program.
  - Dad's wakeup times (at t = 5 sec, t = 10 sec, etc.) interleave the various finish times (3 sec, 6 sec, etc.) of the children, and the output published below reflects that.
  - Understand that the **`SIGCHLD`** handler is invoked 5 times, each in response to some child process finishing up.

```
cgregg@myth60$ ./five-children
Let my five children play while I take a nap.
At least one child still playing, so dad nods off.
Child #1 tired... returns to dad.
Dad wakes up! At least one child still playing, so dad nods off.
Child #2 tired... returns to dad.
Child #3 tired... returns to dad.
Dad wakes up! At least one child still playing, so dad nods off.
Child #4 tired... returns to dad.
Child #5 tired... returns to dad.
Dad wakes up! All children accounted for.  Good job, dad!
cgregg@myth60$
```

Signal Handling Semantics

- A signal is **not** like a function call
  - Signals aren't handled immediately (there can be delays)
  - If a signal is delivered multiple times, the handler is only called once
  - There's a bitmask in the kernel
    - Delivering a signal sets the bit
    - Handling the signal clears the bit
    - If multiple instances of the signal are delivered before handling, handler executes **once**
- Signals execute asynchronously: the kernel can push a stack frame onto the process stack that causes it to execute a handler, then return back to what it was doing
  - This makes signals sort-of-concurrent (technically, preemptive)
  - Keep your signal handlers simple or you will regret it
- This is much like how hardware behaves with interrupts -- POSIX brings that model to software

Example of Tricky Signal Semantics

- Consider the scenario where the five kids run about Disneyland for the same amount of time. Restated, **`sleep(3 * kid)`** is now **`sleep(3)`** so all five children flashmob dad when they're all done.
  - Dad never detects all five kids are present and accounted for, and the program runs forever because dad keeps going back to sleep. Why?

```
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$
```

Waiting without blocking

- Calling **`waitpid`** repeatedly fixes the problem, but it changes the behavior of the program.
  - Calls to **`waitpid`** can prevent dad from returning to his nap. For real programs, this means they can't continue to do work (e.g., respond to shell commands.
- We need to instruct **`waitpid`** to only reap children that have exited but to return without blocking, even if there are more children still running. We use **`WNOHANG`** for this, 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++;
  }
}
```

- Why not just call **`waitpid`** with **`WNOHANG`** in the main loop?
  - Mostly a style question: keeps main loop logic simpler.
  - Also means **`waitpid`** is called promptly, not determined by main loop.
  - Also, it means we learn about more than just termination (stopped, resumed, etc.).

Concurrency

- One of the seven key systems principles we'll be covering this quarter
- Concurrency: 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.
- But it's also very tricky to program -- we will spend a good deal of the quarter showing you all of the challenges and the mechanisms we use to tackle them (starting next lecture)
  - It boils down to shared data, and making sure code always sees that data in a consistent state, e.g., doesn't see counter_1 and counter_2 be different
  - Data analytics frameworks make it possible to massively parallelize computations by defining a data model where there is almost no shared data: the data is split into many independent chunks that are processed in parallel

Lecture Recap

- Review: **fork()** and **execvp()**
- ***Practice:*** Revisiting **`first-shell`**
- Running in the background
- **Break:** Announcements
- Introducing Pipes
- ***Practice:*** Implementing **subprocess**
- Introducing Signals
- **Demo**: Disneyland

Next time: more signals