Lecture 05: `fork` and Understanding `execvp`

Principles of Computer Systems

Winter 2021

Stanford University

Computer Science Department

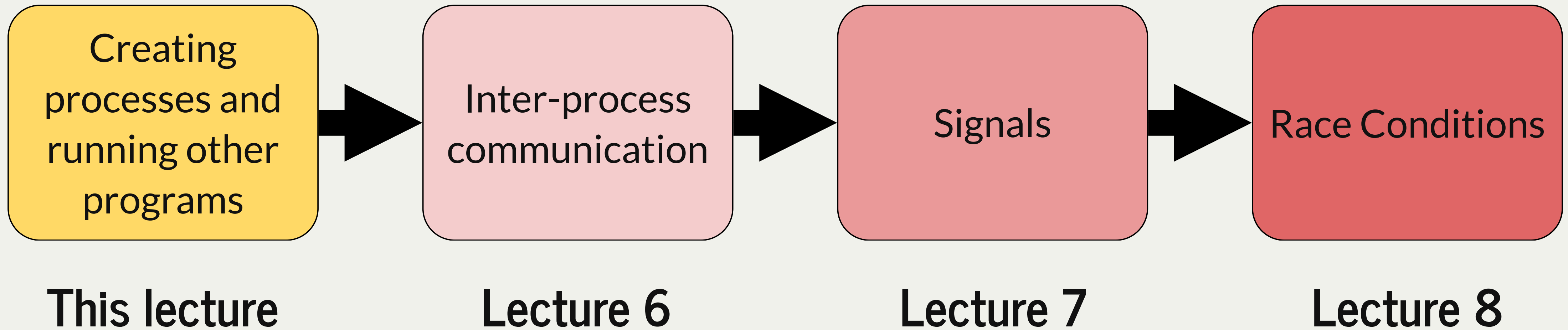
Instructors: Chris Gregg and
Nick Troccoli

Reading: Bryant & O'Hallaron, Chapters 10 and 8

PDF of this presentation

<u>CS110 Topic 2:</u> How can our programs create and interact with other programs?

Learning About Processes



This lecture

Lecture 6

Lecture 7

Lecture 8

Learning Goals

- Get more practice with using **fork()** to create new processes
- Understand how to use **waitpid()** to coordinate between processes
- Learn how **execvp()** lets us execute another program within a process
- **End Goal:** write our first implementation of a shell!

nicktroccoli — troccoli@myth52: ~/CS110-Winter-2020/lecture-examples/processes — ssh troccoli@myth.stanford.edu...

```
troccoli@myth52:~/CS110-Winter-2020/lecture-examples/processes$ ./first-shell
> ls fork*.c
fork-ints.c  forkprob0.c  forkprobloop.c  fork-puzzle.c
return code = 0
> make first-shell
make: 'first-shell' is up to date.
return code = 0
> echo "Hello, world!"
Hello, world!
return code = 0
>
troccoli@myth52:~/CS110-Winter-2020/lecture-examples/processes$
```

first-shell-soln.c

Lecture Plan

- Reintroducing **fork()**
- **Practice:** Seeing....Quadruple?
- **waitpid()** and waiting for child processes
- **Demo**: Waiting For Children
- **execvp()**
- **Putting it all together:** `first-shell`

Lecture Plan

- Reintroducing **fork()**
- **Practice:** Seeing....Quadruple?
- **waitpid()** and waiting for child processes
- **Demo**: Waiting For Children
- **execvp()**
- **Putting it all together:** `first-shell`

fork()

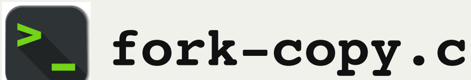
- A system call that creates a new *child process*
- The "parent" is the process that creates the other "child" process
- From then on, both processes are running the code after the fork
- The child process is *identical* to the parent, except:
  - it has a new Process ID (PID)
  - for the parent, fork()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

fork()

What happens to variables and addresses?

```
 1 int main(int argc, char *argv[]) {
 2     char str[128];
 3     strcpy(str, "Hello");
 4     printf("str's address is %p\n", str);
 5 
 6     pid_t pid = fork();
 7 
 8     if (pid == 0) {
 9         // The child should modify str
10         printf("I am the child. str's address is %p\n", str);
11         strcpy(str, "Howdy");
12         printf("I am the child and I changed str to %s. str's address is still %p\n", str, str);
13     } else {
14         // The parent should sleep and print out str
15         printf("I am the parent. str's address is %p\n", str);
16         printf("I am the parent, and I'm going to sleep for 2 seconds.\n");
17         sleep(2);
18         printf("I am the parent. I just woke up. str's address is %p, and its value is %s\n", str, str);
19     }
20 
21     return 0;
22 }
```

 **fork-copy.c**

fork()

```
1 $ ./fork-copy
2 str's address is 0x7ffc8cfa9990
3 I am the parent. str's address is 0x7ffc8cfa9990
4 I am the parent, and I'm going to sleep for 2 seconds.
5 I am the child. str's address is 0x7ffc8cfa9990
6 I am the child and I changed str to Howdy. str's address is still 0x7ffc8cfa9990
7 I am the parent. I just woke up. str's address is 0x7ffc8cfa9990, and its value is Hello
```

How can the parent and child use the same address to store different data?

- Each program thinks it is given all memory addresses to use
- The operating system maps these *virtual* addresses to *physical* addresses
- When a process forks, its virtual address space stays the same
- The operating system will map the child's virtual addresses to different physical addresses than for the parent

fork()

```
1 $ ./fork-copy
2 str's address is 0x7ffc8cfa9990
3 I am the parent. str's address is 0x7ffc8cfa9990
4 I am the parent, and I'm going to sleep for 2 seconds.
5 I am the child. str's address is 0x7ffc8cfa9990
6 I am the child and I changed str to Howdy. str's address is still 0x7ffc8cfa9990
7 I am the parent. I just woke up. str's address is 0x7ffc8cfa9990, and its value is Hello
```

Isn't it expensive to make copies of all memory when forking?

- The operating system only *lazily* makes copies.
- It will have them share physical addresses until one of them changes its memory contents to be different than the other.
- This is called *copy on write* (only make copies when they are written to).

Example: Loaded Dice

```
1  int main(int argc, char *argv[]) {
2      // Initialize the random number with a "seed value"
3      // this seed state is used to generate future random numbers
4      srandom(time(NULL));
5
6      printf("This program will make you question what 'randomness' means...\n");
7      pid_t pidOrZero = fork();
8
9      // Parent goes first - both processes *always* get the same roll (why?)
10     if (pidOrZero != 0) {
11         int diceRoll = (random() % 6) + 1;
12         printf("I am the parent and I rolled a %d\n", diceRoll);
13         sleep(1);
14     } else {
15         sleep(1);
16         int diceRoll = (random() % 6) + 1;
17         printf("I am the child and I'm guessing the parent rolled a %d\n", diceRoll);
18     }
19
20     return 0;
21 }
```

not-so-random.c

Example: Loaded Dice

```
1  int main(int argc, char *argv[]) {
2      // Initialize the random number with a "seed value"
3      // this seed state is used to generate future random numbers
4      srandom(time(NULL));
5
6      printf("This program will make you question what 'randomness' means...\n");
7      pid_t pidOrZero = fork();
8
9      // Parent goes first - both processes *always* get the same roll (why?)
10     if (pidOrZero != 0) {
11         int diceRoll = (random() % 6) + 1;
12         printf("I am the parent and I rolled a %d\n", diceRoll);
13         sleep(1);
14     } else {
15         sleep(1);
16         int diceRoll = (random() % 6) + 1;
17         printf("I am the child and I'm guessing the parent rolled a %d\n", diceRoll);
18     }
19
20     return 0;
21 }
```

Key Idea: all state is copied from the parent to the child, even the random number generator seed! *Both the parent and child will get the same return value from random().*

Lecture Plan

- Reintroducing **fork()**
- **Practice:** Seeing....Quadruple?
- **waitpid()** and waiting for child processes
- **Demo**: Waiting For Children
- **execvp()**
- **Putting it all together:** `first-shell`

It would be nice if there was a function we could call that would ”stall” our program until the child is finished.

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);
```

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 (we'll see other options later)

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 (we'll see other options later)
- **status:** where to put info about the child's termination (or NULL)

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 (we'll see other options later)
- **status:** where to put info about the child's termination (or NULL)
- **options:** optional flags to customize behavior (always 0 for now)

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 (we'll see other options later)
- **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

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 (we'll see other options later)
- **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)

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 (we'll see other options later)
- **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

waitpid()

```
1  int main(int argc, char *argv[]) {
2      printf("Before.\n");
3      pid_t pidOrZero = fork();
4  
5      if (pidOrZero == 0) {
6          sleep(2);
7          printf("I (the child) slept and the parent still waited up for me.\n");
8      } else {
9          pid_t result = waitpid(pidOrZero, NULL, 0);
10         printf("I (the parent) finished waiting for the child.  This always prints last.\n");
11     }
12 
13     return 0;
14 }
```

```
$ ./waitpid
Before.
I (the child) slept and the parent still waited up for me.
I (the parent) finished waiting for the child.  This always prints last.
$
```

waitpid.c

waitpid()

```
1  int main(int argc, char *argv[]) {
2      printf("Before.\n");
3      pid_t pidOrZero = fork();
4  
5      if (pidOrZero == 0) {
6          sleep(2);
7          printf("I (the child) slept and the parent still waited up for me.\n");
8      } else {
9          pid_t result = waitpid(pidOrZero, NULL, 0);
10         printf("I (the parent) finished waiting for the child.  This always prints last.\n");
11     }
12 
13     return 0;
14 }
```

```
$ ./waitpid
Before.
I (the child) slept and the parent still waited up for me.
I (the parent) finished waiting for the child.  This always prints last.
$
```

waitpid.c

waitpid()

```
1  int main(int argc, char *argv[]) {
2      printf("Before.\n");
3      pid_t pidOrZero = fork();
4
5      if (pidOrZero == 0) {
6          sleep(2);
7          printf("I (the child) slept and the parent still waited up for me.\n");
8      } else {
9          pid_t result = waitpid(pidOrZero, NULL, 0);
10         printf("I (the parent) finished waiting for the child.  This always prints last.\n");
11     }
12
13     return 0;
14 }
```

```
$ ./waitpid
Before.
I (the child) slept and the parent still waited up for me.
I (the parent) finished waiting for the child.  This always prints last.
$
```

waitpid.c

waitpid()

```c
 1 int main(int argc, char *argv[]) {
 2     printf("Before.\n");
 3     pid_t pidOrZero = fork();
 4 
 5     if (pidOrZero == 0) {
 6         sleep(2);
 7         printf("I (the child) slept and the parent still waited up for me.\n");
 8     } else {
 9         pid_t result = waitpid(pidOrZero, NULL, 0);
10         printf("I (the parent) finished waiting for the child.  This always prints last.\n");
11     }
12 
13     return 0;
14 }
```

```
$ ./waitpid
Before.
I (the child) slept and the parent still waited up for me.
I (the parent) finished waiting for the child.  This always prints last.
$
```

waitpid()

Pass in the address of an integer as the second parameter to get the child's status.

```
1  int main(int argc, char *argv[]) {
2      pid_t pid = fork();
3      if (pid == 0) {
4          printf("I'm the child, and the parent will wait up for me.\n");
5          return 110; // contrived exit status (not a bad number, though)
6      } else {
7          int status;
8          int result = waitpid(pid, &status, 0);
9
10         if (WIFEXITED(status)) {
11             printf("Child exited with status %d.\n", WEXITSTATUS(status));
12         } else {
13             printf("Child terminated abnormally.\n");
14         }
15         return 0;
16     }
17 }
```

```
$ ./separate
I am the child, and the parent will wait up for me.
Child exited with status 110.
$
```

separate.c

waitpid()

Pass in the address of an integer as the second parameter to get the child's status.

```
1  int main(int argc, char *argv[]) {
2      pid_t pid = fork();
3      if (pid == 0) {
4          printf("I'm the child, and the parent will wait up for me.\n");
5          return 110; // contrived exit status (not a bad number, though)
6      } else {
7          int status;
8          int result = waitpid(pid, &status, 0);
9
10         if (WIFEXITED(status)) {
11             printf("Child exited with status %d.\n", WEXITSTATUS(status));
12         } else {
13             printf("Child terminated abnormally.\n");
14         }
15         return 0;
16     }
17 }
```

```
$ ./separate
I am the child, and the parent will wait up for me.
Child exited with status 110.
$
```

separate.c

- We can use **WIFEXITED** and **WEXITSTATUS** (among others) to extract info from the status. (full program, with error checking, is right here)

waitpid()

Pass in the address of an integer as the second parameter to get the child's status.

```
1  int main(int argc, char *argv[]) {
2      pid_t pid = fork();
3      if (pid == 0) {
4          printf("I'm the child, and the parent will wait up for me.\n");
5          return 110; // contrived exit status (not a bad number, though)
6      } else {
7          int status;
8          int result = waitpid(pid, &status, 0);
9
10         if (WIFEXITED(status)) {
11             printf("Child exited with status %d.\n", WEXITSTATUS(status));
12         } else {
13             printf("Child terminated abnormally.\n");
14         }
15         return 0;
16     }
17 }
```

```
$ ./separate
I am the child, and the parent will wait up for me.
Child exited with status 110.
$
```

separate.c

- We can use **WIFEXITED** and **WEXITSTATUS** (among others) to extract info from the status. (full program, with error checking, is right here)
- The output will be the same every time! The parent will always wait for the child to finish before continuing.

waitpid()

Pass in the address of an integer as the second parameter to get the child's status.

```
 1 int main(int argc, char *argv[]) {
 2     pid_t pid = fork();
 3     if (pid == 0) {
 4         printf("I'm the child, and the parent will wait up for me.\n");
 5         return 110; // contrived exit status (not a bad number, though)
 6     } else {
 7         int status;
 8         int result = waitpid(pid, &status, 0);
 9
10         if (WIFEXITED(status)) {
11             printf("Child exited with status %d.\n", WEXITSTATUS(status));
12         } else {
13             printf("Child terminated abnormally.\n");
14         }
15         return 0;
16     }
17 }
```

```
$ ./separate
I am the child, and the parent will wait up for me.
Child exited with status 110.
$
```

separate.c

- We can use **WIFEXITED** and **WEXITSTATUS** (among others) to extract info from the status. (full program, with error checking, is right here)
- The output will be the same every time! The parent will always wait for the child to finish before continuing.

waitpid()

Pass in the address of an integer as the second parameter to get the child's status.

```
1  int main(int argc, char *argv[]) {
2      pid_t pid = fork();
3      if (pid == 0) {
4          printf("I'm the child, and the parent will wait up for me.\n");
5          return 110; // contrived exit status (not a bad number, though)
6      } else {
7          int status;
8          int result = waitpid(pid, &status, 0);
9
10         if (WIFEXITED(status)) {
11             printf("Child exited with status %d.\n", WEXITSTATUS(status));
12         } else {
13             printf("Child terminated abnormally.\n");
14         }
15         return 0;
16     }
17 }
```

```
$ ./separate
I am the child, and the parent will wait up for me.
Child exited with status 110.
$
```

separate.c

- We can use **WIFEXITED** and **WEXITSTATUS** (among others) to extract info from the status. (full program, with error checking, is right here)
- The output will be the same every time! The parent will always wait for the child to finish before continuing.

`waitpid()`

A parent process should always wait on its children processes.

`waitpid()`

A parent process should always wait on its children processes.

- A process that finished but was not waited on by its parent is called a *zombie* 🧟.

`waitpid()`

A parent process should always wait on its children processes.

- A process that finished but was not waited on by its parent is called a *zombie* 🧟.
- Zombies take up system resources (until they are ultimately cleaned up later by the OS)

waitpid()

A parent process should always wait on its children processes.

- A process that finished but was not waited on by its parent is called a *zombie* 🧟.
- Zombies take up system resources (until they are ultimately cleaned up later by the OS)
- Calling waitpid in the parent "reaps" the child process (cleans it up)
  - If a child is still running, waitpid in the parent will block until it finishes, and then clean it up
  - If a child process is a zombie, waitpid will return immediately and clean it up

`waitpid()`

A parent process should always wait on its children processes.

- A process that finished but was not waited on by its parent is called a *zombie* 🧟.
- Zombies take up system resources (until they are ultimately cleaned up later by the OS)
- Calling waitpid in the parent "reaps" the child process (cleans it up)
  - If a child is still running, waitpid in the parent will block until it finishes, and then clean it up
  - If a child process is a zombie, waitpid will return immediately and clean it up
- Orphaned child processes get "adopted" by the **init** process (PID 1)

Make sure to reap your zombie children.

(Wait, what?)

Lecture Plan

- Reintroducing **fork()**
- **Practice:** Seeing....Quadruple?
- **waitpid()** and waiting for child processes
- **Demo**: Waiting For Children
- **execvp()**
- **Putting it all together:** `first-shell`

Waiting On Multiple Children, No Order

A parent can call `fork` multiple times, but must reap all the child processes.

- A parent can use **waitpid** to wait on *any of its children* by passing in **-1** as the PID.
- **Key Idea:** The children may terminate in *any* order!
- If **waitpid** returns -1 and sets **errno** to **ECHILD**, this means there are no more children.

Demo: Let's see how we might use this (**reap-as-they-exit.c**)

`reap-as-they-exit.c`

Waiting On Multiple Children, In Order

What if we want to wait for children in the order in which they were created?

Check out the abbreviated program below (full program with error checking right here):

```
1  int main(int argc, char *argv[]) {
2      pid_t children[kNumChildren];
3
4      for (size_t i = 0; i < kNumChildren; i++) {
5          children[i] = fork();
6          if (children[i] == 0) exit(110 + i);
7      }
8
9      for (size_t i = 0; i < kNumChildren; i++) {
10         int status;
11         pid_t pid = waitpid(children[i], &status, 0);
12         assert(WIFEXITED(status));
13         printf("Child with pid %d accounted for (return status of %d).\n", children[i], WEXITSTATUS(status));
14     }
15
16     return 0;
17 }
```

reap-in-fork-order.c

Waiting On Multiple Children, In Order

What if we want to wait for children in the order in which they were created?

Check out the abbreviated program below (full program with error checking right here):

```
 1 int main(int argc, char *argv[]) {
 2     pid_t children[kNumChildren];
 3 
 4     for (size_t i = 0; i < kNumChildren; i++) {
 5         children[i] = fork();
 6         if (children[i] == 0) exit(110 + i);
 7     }
 8 
 9     for (size_t i = 0; i < kNumChildren; i++) {
10         int status;
11         pid_t pid = waitpid(children[i], &status, 0);
12         assert(WIFEXITED(status));
13         printf("Child with pid %d accounted for (return status of %d).\n", children[i], WEXITSTATUS(status));
14     }
15 
16     return 0;
17 }
```

reap-in-fork-order.c

Waiting On Multiple Children, In Order

What if we want to wait for children in the order in which they were created?

Check out the abbreviated program below (full program with error checking right here):

```
 1 int main(int argc, char *argv[]) {
 2     pid_t children[kNumChildren];
 3 
 4     for (size_t i = 0; i < kNumChildren; i++) {
 5         children[i] = fork();
 6         if (children[i] == 0) exit(110 + i);
 7     }
 8 
 9     for (size_t i = 0; i < kNumChildren; i++) {
10         int status;
11         pid_t pid = waitpid(children[i], &status, 0);
12         assert(WIFEXITED(status));
13         printf("Child with pid %d accounted for (return status of %d).\n", children[i], WEXITSTATUS(status));
14     }
15 
16     return 0;
17 }
```

reap-in-fork-order.c

Waiting On Multiple Children, In Order

What if we want to wait for children in the order in which they were created?

Check out the abbreviated program below (full program with error checking right here):

```
1  int main(int argc, char *argv[]) {
2      pid_t children[kNumChildren];
3
4      for (size_t i = 0; i < kNumChildren; i++) {
5          children[i] = fork();
6          if (children[i] == 0) exit(110 + i);
7      }
8
9      for (size_t i = 0; i < kNumChildren; i++) {
10         int status;
11         pid_t pid = waitpid(children[i], &status, 0);
12         assert(WIFEXITED(status));
13         printf("Child with pid %d accounted for (return status of %d).\n", children[i], WEXITSTATUS(status));
14     }
15
16     return 0;
17 }
```

reap-in-fork-order.c

Waiting On Multiple Children, In Order

What if we want to wait for children in the order in which they were created?

Check out the abbreviated program below (full program with error checking right here):

```
 1 int main(int argc, char *argv[]) {
 2     pid_t children[kNumChildren];
 3 
 4     for (size_t i = 0; i < kNumChildren; i++) {
 5         children[i] = fork();
 6         if (children[i] == 0) exit(110 + i);
 7     }
 8 
 9     for (size_t i = 0; i < kNumChildren; i++) {
10         int status;
11         pid_t pid = waitpid(children[i], &status, 0);
12         assert(WIFEXITED(status));
13         printf("Child with pid %d accounted for (return status of %d).\n", children[i], WEXITSTATUS(status));
14     }
15 
16     return 0;
17 }
```

reap-in-fork-order.c

Waiting On Multiple Children, In Order

What if we want to wait for children in the order in which they were created?

Check out the abbreviated program below (full program with error checking right here):

```
1  int main(int argc, char *argv[]) {
2      pid_t children[kNumChildren];
3  
4      for (size_t i = 0; i < kNumChildren; i++) {
5          children[i] = fork();
6          if (children[i] == 0) exit(110 + i);
7      }
8  
9      for (size_t i = 0; i < kNumChildren; i++) {
10         int status;
11         pid_t pid = waitpid(children[i], &status, 0);
12         assert(WIFEXITED(status));
13         printf("Child with pid %d accounted for (return status of %d).\n", children[i], WEXITSTATUS(status));
14     }
15 
16     return 0;
17 }
```

reap-in-fork-order.c

Waiting On Multiple Children, In Order

- This program reaps processes in the order they were spawned.
- Child processes may not *finish* in this order, but they are *reaped* in this order.
  - E.g. first child could finish last, holding up first loop iteration
- Sample run below - the pids change between runs, but even those are guaranteed to be published in increasing order.

```
$ ./reap-in-fork-order
Child with pid 12649 accounted for (return status of 110).
Child with pid 12650 accounted for (return status of 111).
Child with pid 12651 accounted for (return status of 112).
Child with pid 12652 accounted for (return status of 113).
Child with pid 12653 accounted for (return status of 114).
Child with pid 12654 accounted for (return status of 115).
Child with pid 12655 accounted for (return status of 116).
Child with pid 12656 accounted for (return status of 117).
$
```

Lecture Plan

- Reintroducing **fork()**
- **Practice:** Seeing....Quadruple?
- **waitpid()** and waiting for child processes
- **Demo**: Waiting For Children
- **execvp()**
- **Putting it all together:** `first-shell`

`execvp()`

The most common use for **fork** is not to spawn multiple processes to split up work, but instead to run a *completely separate program* under your control and communicate with it.

- This is what a **shell** is; it is a program that prompts you for commands, and it executes those commands in separate processes. Let's take a look.

execvp()

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

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

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 executable at the specified **path**, *completely cannibalizing the current process.*

- If successful, **execvp never returns** in the calling process
- If unsuccessful, **execvp** returns -1

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 executable at the specified **path**, *completely cannibalizing the current process.*

- If successful, **execvp never returns** in the calling process
- If unsuccessful, **execvp** returns -1

To run another executable, we must specify the (NULL-terminated) arguments to be passed into its **main** function, via the **argv** parameter.

- For our programs, **path** and **argv[0]** will be the same

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 executable at the specified **path**, *completely cannibalizing the current process.*

- If successful, **execvp never returns** in the calling process
- If unsuccessful, **execvp** returns -1

To run another executable, we must specify the (NULL-terminated) arguments to be passed into its **main** function, via the **argv** parameter.

- For our programs, **path** and **argv[0]** will be the same

execvp has many variants (**execle**, **execlp**, and so forth. Type **man execvp** for more). We rely on **execvp** in CS110.

execvp()

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

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

```
1 int main(int argc, char *argv[]) {
2     char *args[] = {"/bin/ls", "-l", "/usr/class/cs110/lecture-examples", NULL};
3     execvp(args[0], args);
4     printf("This only prints if an error occurred.\n");
5     return 0;
6 }
```

```
 1 $ ./execvp-demo
 2 total 26
 3 drwx------ 2 troccoli operator 2048 Jan 11 21:03 cpp-primer
 4 drwx------ 3 troccoli operator 2048 Jan 15 12:43 cs107review
 5 drwx------ 2 troccoli operator 2048 Jan 13 14:15 filesystems
 6 drwx------ 2 troccoli operator 2048 Jan 13 14:14 lambda
 7 drwxr-xr-x 3 poohbear root     2048 Nov 19 13:24 map-reduce
 8 drwx------ 2 poohbear root     4096 Nov 19 13:25 networking
 9 drwxr-xr-x 2 poohbear root     6144 Jan 22 08:58 processes
10 drwxr-xr-x 2 poohbear root     2048 Oct 29 06:57 threads-c
11 drwxr-xr-x 2 poohbear root     4096 Oct 29 06:57 threads-cpp
12 $
```

execvp-demo.c

execvp()

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

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

```
1 int main(int argc, char *argv[]) {
2     char *args[] = {"/bin/ls", "-l", "/usr/class/cs110/lecture-examples", NULL};
3     execvp(args[0], args);
4     printf("This only prints if an error occurred.\n");
5     return 0;
6 }
```

```
 1 $ ./execvp-demo
 2 total 26
 3 drwx------ 2 troccoli operator 2048 Jan 11 21:03 cpp-primer
 4 drwx------ 3 troccoli operator 2048 Jan 15 12:43 cs107review
 5 drwx------ 2 troccoli operator 2048 Jan 13 14:15 filesystems
 6 drwx------ 2 troccoli operator 2048 Jan 13 14:14 lambda
 7 drwxr-xr-x 3 poohbear root     2048 Nov 19 13:24 map-reduce
 8 drwx------ 2 poohbear root     4096 Nov 19 13:25 networking
 9 drwxr-xr-x 2 poohbear root     6144 Jan 22 08:58 processes
10 drwxr-xr-x 2 poohbear root     2048 Oct 29 06:57 threads-c
11 drwxr-xr-x 2 poohbear root     4096 Oct 29 06:57 threads-cpp
12 $
```

execvp-demo.c

Lecture Plan

- Reintroducing **fork()**
- **Practice:** Seeing....Quadruple?
- **waitpid()** and waiting for child processes
- **Demo**: Waiting For Children
- **execvp()**
- **Putting it all together:** `first-shell`

What Is A Shell?

A shell is essentially a program that repeats asking the user for a command and running that command (Demo: **first-shell-soln.c**)

first-shell-soln.c

What Is A Shell?

A shell is essentially a program that repeats asking the user for a command and running that command (Demo: **first-shell-soln.c**)

- Component 1: loop for asking for user input

`first-shell-soln.c`

What Is A Shell?

A shell is essentially a program that repeats asking the user for a command and running that command (Demo: **first-shell-soln.c**)

- Component 1: loop for asking for user input
- **Component 2: way to run an arbitrary command**

first-shell-soln.c

system()

The built-in **system** function can execute a given shell command.

```
int system(const char *command);
```

- **command** is a shell command (like you would type in the terminal); e.g. **"ls"** or **"./myProgram"**
- **system** forks off a child process that executes the given shell command, and waits for it
- on success, **system** returns the termination status of the child

system-demo.c

system()

The built-in **system** function can execute a given shell command.

```
int system(const char *command);
```

- **command** is a shell command (like you would type in the terminal); e.g. **"ls"** or **"./myProgram"**
- **system** forks off a child process that executes the given shell command, and waits for it
- on success, **system** returns the termination status of the child

```
1 int main(int argc, char *argv[]) {
2     int status = system(argv[1]);
3     printf("system returned %d\n", status);
4     return 0;
5 }
```

system-demo.c

system()

The built-in **system** function can execute a given shell command.

```
int system(const char *command);
```

- **command** is a shell command (like you would type in the terminal); e.g. **"ls"** or **"./myProgram"**
- **system** forks off a child process that executes the given shell command, and waits for it
- on success, **system** returns the termination status of the child

```
1 int main(int argc, char *argv[]) {
2     int status = system(argv[1]);
3     printf("system returned %d\n", status);
4     return 0;
5 }
```

```
 1 $ ./system-demo "ls -l"
 2 total 26
 3 drwx------ 2 troccoli operator 2048 Jan 11 21:03 cpp-primer
 4 drwx------ 3 troccoli operator 2048 Jan 15 12:43 cs107review
 5 drwx------ 2 troccoli operator 2048 Jan 13 14:15 filesystems
 6 drwx------ 2 troccoli operator 2048 Jan 13 14:14 lambda
 7 drwxr-xr-x 3 poohbear root     2048 Nov 19 13:24 map-reduce
 8 drwx------ 2 poohbear root     4096 Nov 19 13:25 networking
 9 drwxr-xr-x 2 poohbear root     6144 Jan 21 19:38 processes
10 drwxr-xr-x 2 poohbear root     2048 Oct 29 06:57 threads-c
11 drwxr-xr-x 2 poohbear root     4096 Oct 29 06:57 threads-cpp
12 system returned 0
13 $
```

system-demo.c

mysystem()

We can implement our own version of **system** with **fork(), waitpid() and execvp()**!

```
int mysystem(const char *command);
```

1. call **fork** to create a child process
2. In the child, call **execvp** with the command to execute
3. In the parent, wait for the child with **waitpid** and then return exit status info

mysystem()

We can implement our own version of **system** with **fork(), waitpid() and execvp()**!

```
int mysystem(const char *command);
```

1. call **fork** to create a child process
2. In the child, call **execvp** with the command to execute
3. In the parent, wait for the child with **waitpid** and then return exit status info

One twist; not all shell commands are executable programs, and some need parsing.

- We can't just pass the command to **execvp**
- **Solution:** there is a *program* called **sh** that runs any shell command
  - e.g. **/bin/sh -c "ls -a"** runs the command "ls -a"
  - We can call **execvp** to run **/bin/sh** with **-c** and the **command** as arguments

mysystem()

Here's the implementation, with minimal error checking (the full version is right here):

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

first-shell-soln.c

mysystem()

Here's the implementation, with minimal error checking (the full version is right here):

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

- If **execvp** returns at all, an error occurred

first-shell-soln.c

mysystem()

Here's the implementation, with minimal error checking (the full version is right here):

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

- If **execvp** returns at all, an error occurred
- Why not call **execvp** inside parent and forgo the child process altogether? Because **execvp** would consume the calling process, and that's not what we want.

first-shell-soln.c

mysystem()

Here's the implementation, with minimal error checking (the full version is right here):

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

- If **execvp** returns at all, an error occurred
- Why not call **execvp** inside parent and forgo the child process altogether? Because **execvp** would consume the calling process, and that's not what we want.

first-shell-soln.c

mysystem()

Here's the implementation, with minimal error checking (the full version is right here):

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

- If **execvp** returns at all, an error occurred
- Why not call **`execvp`** inside parent and forgo the child process altogether? Because **`execvp`** would consume the calling process, and that's not what we want.

`first-shell-soln.c`

`mysystem()`

Here's the implementation, with minimal error checking (the full version is right here):

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

- If **execvp** returns at all, an error occurred
- Why not call **`execvp`** inside parent and forgo the child process altogether? Because **`execvp`** would consume the calling process, and that's not what we want.

`first-shell-soln.c`

mysystem()

Here's the implementation, with minimal error checking (the full version is right here):

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

- If **execvp** returns at all, an error occurred
- Why not call **execvp** inside parent and forgo the child process altogether? Because **execvp** would consume the calling process, and that's not what we want.

first-shell-soln.c

mysystem()

Here's the implementation, with minimal error checking (the full version is right here):

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

- If **execvp** returns at all, an error occurred
- Why not call **execvp** inside parent and forgo the child process altogether? Because **execvp** would consume the calling process, and that's not what we want.

first-shell-soln.c

mysystem()

Here's the implementation, with minimal error checking (the full version is right here):

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

- If **execvp** returns at all, an error occurred
- Why not call **execvp** inside parent and forgo the child process altogether? Because **execvp** would consume the calling process, and that's not what we want.

first-shell-soln.c

Lecture Recap

- Reintroducing **fork()**
- **Practice:** Seeing....Quadruple?
- **waitpid()** and waiting for child processes
- **Demo**: Waiting For Children
- **execvp()**
- **Putting it all together:** `first-shell`

Next time: Interprocess communication

Practice Problems

Practice: fork()

```
1  int main(int argc, char *argv[]) {
2      printf("Starting the program\n");
3      pid_t pidOrZero1 = fork();
4      pid_t pidOrZero2 = fork();
5
6      if (pidOrZero1 != 0 && pidOrZero2 != 0) {
7          printf("Hello\n");
8      }
9
10     if (pidOrZero2 != 0) {
11         printf("Hi there\n");
12     }
13
14     return 0;
15 }
```

How many processes run in total?

a) 1 b) 2 c) 3 d) 4

How many times is "Hello" printed?

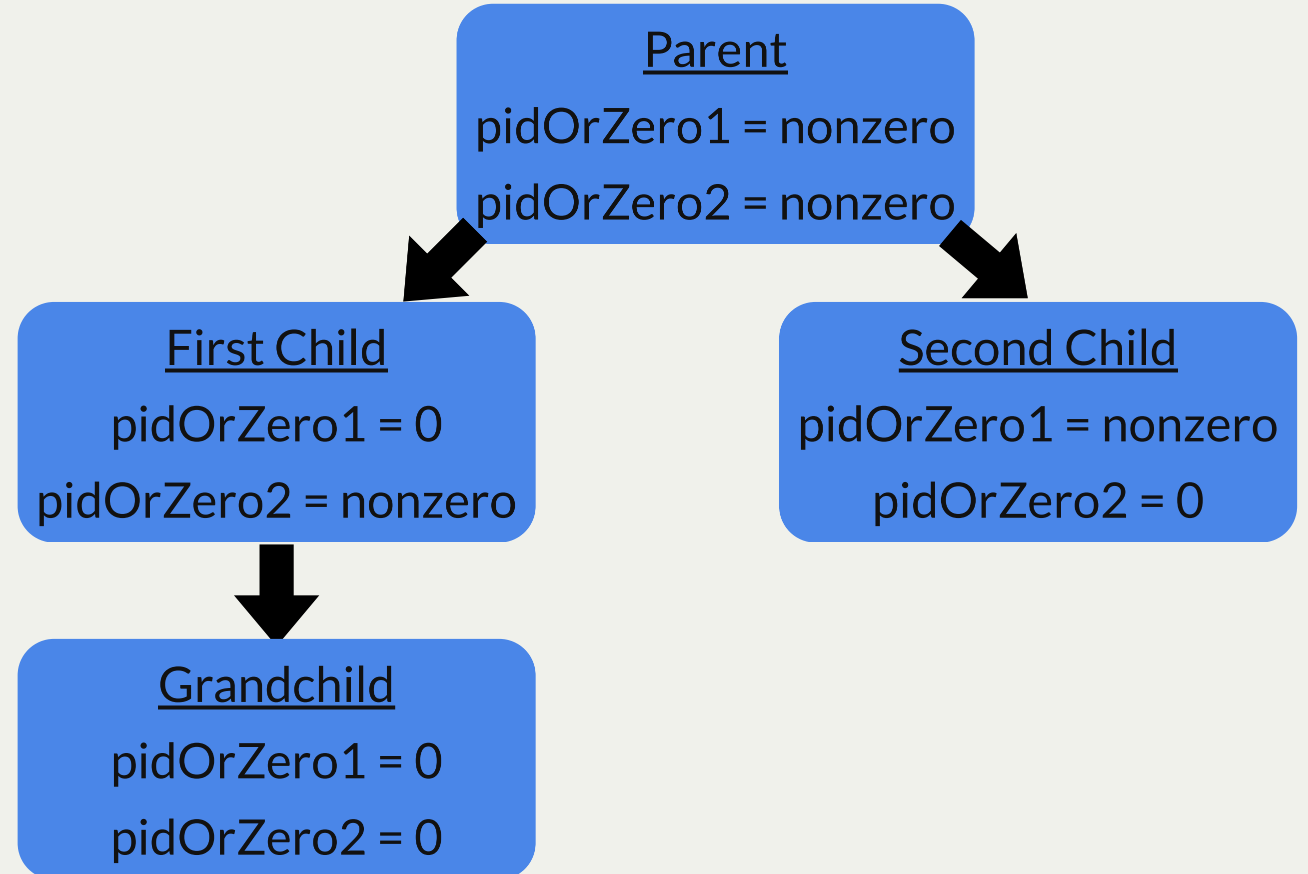
a) 1 b) 2 c) 3 d) 4

How many times is "Hi there" printed?

a) 1 b) 2 c) 3 d) 4

Practice: fork()

```
1  int main(int argc, char *argv[]) {
2      printf("Starting the program\n");
3      pid_t pidOrZero1 = fork();
4      pid_t pidOrZero2 = fork();
5
6      if (pidOrZero1 != 0 && pidOrZero2 != 0) {
7          printf("Hello\n");
8      }
9
10     if (pidOrZero2 != 0) {
11         printf("Hi there\n");
12     }
13
14     return 0;
15 }
```



Waiting On Children

What if we want to spawn a single child and wait for that child before spawning another child?

Waiting On Children

What if we want to spawn a single child and wait for that child before spawning another child? Check out the abbreviated program below (full program with error checking right here):

```
1  static const int kNumChildren = 8;\
2  int main(int argc, char *argv[]) {
3      for (size_t i = 0; i < kNumChildren; i++) {
4          pid_t pidOrZero = fork();
5          if (pidOrZero == 0) {
6              printf("Hello from child %d!\n", getpid());
7              return 110 + i;
8          }
9  
10         int status;
11         pid_t pid = waitpid(pidOrZero, &status, 0);
12         if (WIFEXITED(status)) {
13             printf("Child with pid %d exited normally with status %d\n", pid, WEXITSTATUS(status));
14         } else {
15             printf("Child with pid %d exited abnormally\n", pid);
16         }
17     }
18 
19     return 0;
20 }
```

spawn-and-reap.c