

CS110 Lecture 7: waitpid and execvp

CS110: Principles of Computer Systems

Winter 2021-2022

Stanford University

Instructors: Nick Troccoli and Jerry Cain

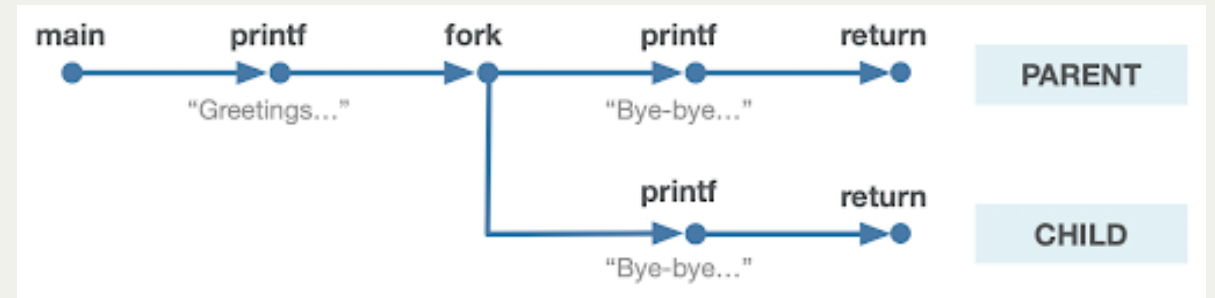


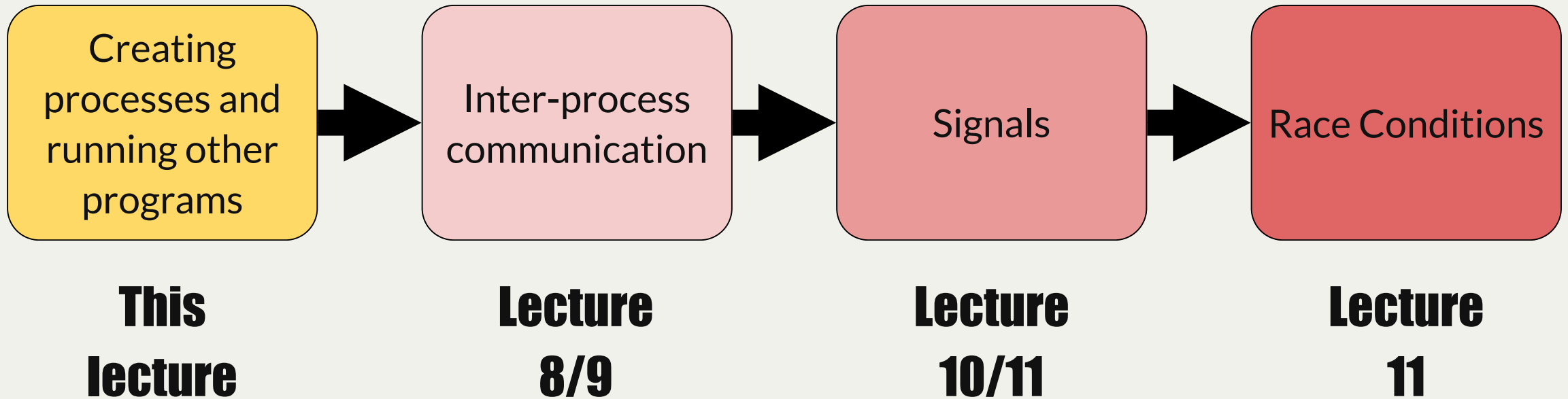
Illustration courtesy of Roz Cyrus.



[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

- 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
- **Goal:** write our first implementation of a shell!

```
nicktroccoli — troccoli@myth54: ~ — ssh troccoli@myth.stanford.edu — 88x15
troccoli@myth54:~$ ./first-shell-soln
> ls first-shell*
first-shell.c first-shell.c~ first-shell-soln first-shell-soln.c first-shell-soln.c~
return code = 0
> make first-shell
gcc -g -Wall -pedantic -O0 -std=gnu99 -I/afs/ir.stanford.edu/class/cs110/local/include
  first-shell.c  -o first-shell
return code = 0
> echo "Hello, world!"
Hello, world!
return code = 0
>
troccoli@myth54:~$
```



`first-shell-soln.c`

Lecture Plan

- Recap: `fork()`
- `waitpid()` and waiting for child processes
- Demo: Waiting For Children
- `execvp()`
- Putting it all together: `first-shell`
- Background execution

Lecture Plan

- Recap: fork()
- `waitpid()` and waiting for child processes
- **Demo: Waiting For Children**
- `execvp()`
- **Putting it all together:** `first-shell`
- Background execution

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");
```

Virtual Memory and Copy on Write

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 *lazily*
- 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).

Lecture Plan

- Recap: `fork()`
- `waitpid()` and waiting for child processes
- Demo: Waiting For Children
- `execvp()`
- Putting it all together: `first-shell`
- Background execution

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);
```

- **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()

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.
$
```

 `waitpid-status.c`

- We can use provided macros (see man page for full list) to extract info from the status. (full program, with error checking, linked below)
 - **WIFEXITED**: check if child terminated normally
 - **WEXITSTATUS**: get exit status of child
- 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.

- 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

- Recap: `fork()`
- `waitpid()` and waiting for child processes
- Demo: Waiting For Children
- `execvp()`
- Putting it all together: `first-shell`
- Background execution

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 (link to full program at the bottom):

```
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) return 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 }
```



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).
$
```

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.

Lecture Plan

- Recap: `fork()`
- `waitpid()` and waiting for child processes
- Demo: Waiting For Children
- `execvp()`
- Putting it all together: `first-shell`
- Background execution

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.

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 (**execl**, **exec1p**, 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`

Lecture Plan

- Recap: `fork()`
- `waitpid()` and waiting for child processes
- Demo: Waiting For Children
- `execvp()`
- Putting it all together: `first-shell`
- Background execution

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**



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

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 linked at the bottom):

```
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.
- Why must the child `exit` rather than `return`? Because that would cause the child to *also* execute code in `main`!



`first-shell-soln.c`

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.

Lecture Plan

- Recap: `fork()`
- `waitpid()` and waiting for child processes
- Demo: Waiting For Children
- `execvp()`
- Putting it all together: `first-shell`
- Background execution

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
- In lecture and assign4, we will see all these features!
- Today, we'll focus on background execution
 - only difference is specifying `&` with command
 - shell immediately re-prompts the user
 - process doesn't know "foreground" vs. "background"; this specifies whether or not shell waits



Supporting Background Execution

Let's make an updated version of `mysystem` called `executeCommand`.

- Takes an additional parameter `bool inBackground`
 - If `false`, same behavior as `mysystem` (spawn child, `execvp`, wait for child)
 - If `true`, spawn child, `execvp`, but *don't wait for child*



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 }
```



first-shell-soln-bg.c

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.



first-shell-soln-bg.c

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.



first-shell-soln-bg.c

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, we add two additional things:

- Check for the "quit" command to exit
- Allow the user to add "&" at the end of a command to run that command in the background

Note that a background child isn't reaped!
This is a problem - one we'll learn how to fix soon.



Lecture Recap

- Recap: `fork()`
- `waitpid()` and waiting for child processes
- Demo: Waiting For Children
- `execvp()`
- **Putting it all together:** `first-shell`
- Background execution

Next time: interprocess communication

Practice Problems

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 (link to full program at bottom):

```
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 }
```

