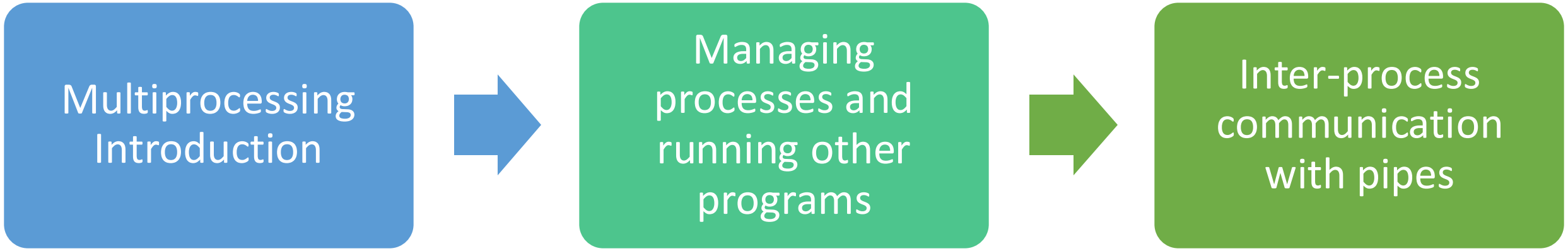
CS111, Lecture 9
Multiprocessing System Calls

CS111 Topic 2: Multiprocessing

Key Question: *How can our program create and interact with other programs? How does the operating system manage user programs?*



assign3: implement your own shell!

Learning Goals

- Learn how to use **waitpid()** to wait for a child process to finish.
- Understand how to use **execvp()** to run a new program within a process.
- See how a shell is implemented using **fork** + **execvp** + **waitpid**

Plan For Today

- **Recap**: **fork()**
- **waitpid()** and waiting for child processes
- **Demo:** waiting for children
- **execvp()**
- Building our first shell

```
cp -r /afs/ir/class/cs111/lecture-code/lect9 .
```

Plan For Today

- **Recap**: **fork()**
- **waitpid()** and waiting for child processes
- **Demo:** waiting for children
- **execvp()**
- Building our first shell

```
cp -r /afs/ir/class/cs111/lecture-code/lect9 .
```

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**

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

Child runs until the end of the program

```
void helperFn() {
    pid_t pidOrZero = fork();
    if (pidOrZero == 0) {
        printf("I am the child\n");
    } else {
        printf("I am the parent\n");
    }
}

int main(int argc, char *argv[]) {
    helperFn();
    printf("This is printed once\n");
}
```

Child runs until the end of the program

```
void helperFn() {
    pid_t pidOrZero = fork();
    if (pidOrZero == 0) {
        printf("I am the child\n");
        exit(0);
    } else {
        printf("I am the parent\n");
    }
}

int main(int argc, char *argv[]) {
    helperFn();
    printf("This is printed once\n");
}
```

Process Clones

```
$ ./fork-copy
str's address is 0x7ffc8cfa9990
I am the parent. str's address is 0x7ffc8cfa9990
I am the parent, and I'm going to sleep for 2sec.
I am the child. str's address is 0x7ffc8cfa9990
I am the child and I changed str to Howdy. str's address is still
0x7ffc8cfa9990
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
- **Copy on write:** only as needed, the operating system will map the child's virtual addresses to different physical addresses than for the parent

Our Goal: Shell

A *shell* is a program that prompts the user for a command to run, runs that command, waits for the command to finish, and then prompts the user again.

```
while (true) {
        char *user_command = … // user input
        pid_t pidOrZero = fork();
        if (pidOrZero == 0) {
                // run user's command in the child, then terminate
                        ???
        }

        // parent waits for child before continuing
                        ???
}
```

Our Goal: Shell

A *shell* is a program that prompts the user for a command to run, runs that command, waits for the command to finish, and then prompts the user again.

```
while (true) {
        char *user_command = … // user input
        pid_t pidOrZero = fork();
        if (pidOrZero == 0) {
                // run user's command in the child,
                        ???
        }

        // parent waits for child before continui
                        ???
}
```

Key idea: we can only run one program per process, so we need to run the user's command in another process – otherwise, the shell will go away!

Our Goal: Shell

A *shell* is a program that prompts the user for a command to run, runs that command, waits for the command to finish, and then prompts the user again.

```
while (true) {
        char *user_command = … // user input
        pid_t pidOrZero = fork();
        if (pidOrZero == 0) {
                // run user's command in the child, then terminate
                        execvp
        }

        // parent waits for child before continuing
                        waitpid
}
```

Plan For Today

- Recap: **fork()**
- **waitpid()** and waiting for child processes
- **Demo:** waiting for children
- **execvp()**
- Building our first shell

```
cp -r /afs/ir/class/cs111/lecture-code/lect9 .
```

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

waitpid()

A system call 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()

```c
// waitpid.c
int main(int argc, char *argv[]) {
    printf("Before.\n");
    pid_t pidOrZero = fork();
    if (pidOrZero == 0) {
        sleep(2);
        printf("I (the child) slept and the parent waited for me.\n");
    } else {
        pid_t result = waitpid(pidOrZero, NULL, 0);
        printf("I (the parent) finished waiting for the child. This
               always prints last.\n");
    }
    return 0;
}
```

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

waitpid()

```c
// waitpid-status.c
int main(int argc, char *argv[]) {
    pid_t pid = fork();
    if (pid == 0) {
        printf("I'm the child, and the parent will wait up for me.\n");
        return 111; // contrived exit status (not a bad number, though)
    } else {
        int status;
        int result = waitpid(pid, &status, 0);
        if (WIFEXITED(status)) {
            printf("Child exited with status %d.\n", WEXITSTATUS(status));
        } else {
            printf("Child terminated abnormally.\n");
        }
        return 0;
    }
}
```

```
I'm the child, and the parent will wait up for me.
Child exited with status 111.
```

waitpid()

```
...
int status;
int result = waitpid(pid, &status, 0);
if (WIFEXITED(status)) {
    printf("Child exited with status %d.\n", WEXITSTATUS(status));
} else {
    printf("Child terminated abnormally.\n");
}
...
```

Provided macros (see man page for full list) let us extract info from the status.

- **WIFEXITED** – check if child terminated normally
- **WEXITSTATUS** – get exit status of child

This output will be the same every time! The parent will always wait for the child to finish before continuing.

waitpid()

Another benefit of **waitpid**: it cleans up the state of the terminated child process

- A process that finished but hasn't yet been waited on by its parent is called a *zombie*  .
- Zombies take up system resources (until they are ultimately cleaned up later by the OS). Therefore, a parent process should always wait on its children processes.
- If a child is still running, waitpid in the parent will block until the child finishes, and then clean it up. If a child process is a zombie, waitpid will return immediately and clean it up.
- Child processes whose parent process terminates without waiting on them get the **init** process (PID 1) as their parent.

Make sure to clean up after your zombie children.

(wait, what?)

Which output is *not* possible?

```
int main() {
    pid_t pidOrZero1 = fork();
    if (pidOrZero1 == 0) {
        printf("Hello 1!\n");
        return 0;
    }

    pid_t pidOrZero2 = fork();
    if (pidOrZero2 == 0) {
        printf("Hi 2!\n");
        return 0;
    }

    waitpid(pidOrZero1, NULL, 0);
    printf("Goodbye 1\n");
    waitpid(pidOrZero2, NULL, 0);
    printf("Goodbye 2\n");
    return 0;
}
```

A) Hello 1! Hi 2! Goodbye 1 Goodbye 2	B) Hi 2! Hello 1! Goodbye 1 Goodbye 2
C) Hello 1! Goodbye 1 Hi 2! Goodbye 2	D) Hi 2! Goodbye 1 Hello 1! Goodbye 2

Respond on PollEv: pollev.com/cs111 or text CS111 to 22333 once to join.

Which of these outputs is *not* possible?

A

0%

B

0%

C

0%

D

0%

How do these differ?

```
int main() {
    pid_t pidOrZero1 = fork();
    if (pidOrZero1 == 0) {
        printf("Hello 1!\n");
        return 0;
    }

    pid_t pidOrZero2 = fork();
    if (pidOrZero2 == 0) {
        printf("Hi 2!\n");
        return 0;
    }

    waitpid(pidOrZero1, NULL, 0);
    printf("Goodbye 1\n");
    waitpid(pidOrZero2, NULL, 0);
    printf("Goodbye 2\n");
    return 0;
}
```

VS

```
int main() {
    pid_t pidOrZero1 = fork();
    if (pidOrZero1 == 0) {
        printf("Hello 1!\n");
        return 0;
    }

    waitpid(pidOrZero1, NULL, 0);
    printf("Goodbye 1\n");

    pid_t pidOrZero2 = fork();
    if (pidOrZero2 == 0) {
        printf("Hi 2!\n");
        return 0;
    }

    waitpid(pidOrZero2, NULL, 0);
    printf("Goodbye 2\n");
    return 0;
}
```

Plan For Today

- Recap: **fork()**
- **waitpid()** and waiting for child processes
- **Demo: waiting for children**
- **execvp()**
- Building our first shell

```
cp -r /afs/ir/class/cs111/lecture-code/lect9 .
```

Waiting for Children

Problem: if we have multiple children and want to wait on all of them, in what order do we wait on them to finish?

One idea: keep an array of child **pid**s, and call **waitpid** on each in a loop.

This works, though the child processes may not *finish* in this order.

`reap-in-fork-order.c and reap-as-they-exit.c`

Waiting for Children

Problem: if we have multiple children and want to wait on all of them, in what order do we wait on them to finish?

Ideally we could say "wait until one of my children finishes".

- A parent can pass **-1** as the PID to **waitpid** to wait on *any one* of its children.
- **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.
- We can do this in a loop to wait on all children in the order they exit

Let's see a demo!

`reap-in-fork-order.c and reap-as-they-exit.c`

Plan For Today

- Recap: **fork()**
- **waitpid()** and waiting for child processes
- **Demo:** waiting for children
- **execvp()**
- Building our first shell

```
cp -r /afs/ir/class/cs111/lecture-code/lect9 .
```

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 the behavior of a **shell**!

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 given 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 (see **man execvp**) but we'll just be using **execvp**.

execvp()

```c
// execvp-demo.c
int main(int argc, char *argv[]) {
    printf("Hello, world!\n");
    char *args[] = {"/bin/ls", "-l", "/usr/class/cs111/lecture-code",
                    NULL};
    execvp(args[0], args);
    printf("This only prints if an error occurred.\n");
    return 0;
}
```

```
$ ./execvp-demo
Hello, world!
total 4
drwx------ 2 troccoli operator 2048 Oct  9 16:21 lect5
drwx------ 2 troccoli operator 2048 Oct 13 22:19 lect9
```

How is execvp useful?

- This is the way that we can run other programs
- However, we often don't want to cannibalize the current process
- Instead: we will usually fork off a child process and call execvp there. The child process will be consumed, but that's ok
- Key idea: the process is still the child process, and **the parent can still wait on it**. It's just running another program.

Plan For Today

- Recap: **fork()**
- **waitpid()** and waiting for child processes
- **Demo:** waiting for children
- **execvp()**
- **Building our first shell**

```
cp -r /afs/ir/class/cs111/lecture-code/lect9 .
```

Implementing a Shell

A shell is essentially a program that repeats asking the user for a command and running that command

How do we run a command entered by the user?

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**

For assign3, you'll use this pattern to build your own shell, stsh ("Stanford shell") with various functionality of real Unix shells.

Implementing a Shell

```
while (true) {
      char *user_command = … // user input
      if (strcmp(user_command, "sort") == 0) {
            // code for sort here
      } else if (strcmp(user_command, "ls") == 0) {
            // code for ls functionality here
      }
      ...
}
```

Why do we need execvp? Why can't we implement a shell like this?

- Would need to implement every possible command's logic (yikes!)
- Couldn't run a command the shell doesn't know about (e.g. own programs)

Recap

- **Recap**: **fork()**
- **waitpid()** and waiting for child processes
- **Demo:** waiting for children
- **execvp**
- Our first shell

Next time: how processes can communicate with pipes

Lecture 9 takeaway: waitpid lets a parent process wait for a child process to finish. **execvp** takes over the calling process to run the specified program. Shells work by spawning child processes with **fork** that call **execvp,** and then waiting for them to finish with **waitpid**.