CS111, Lecture 9
Multiprocessing System Calls

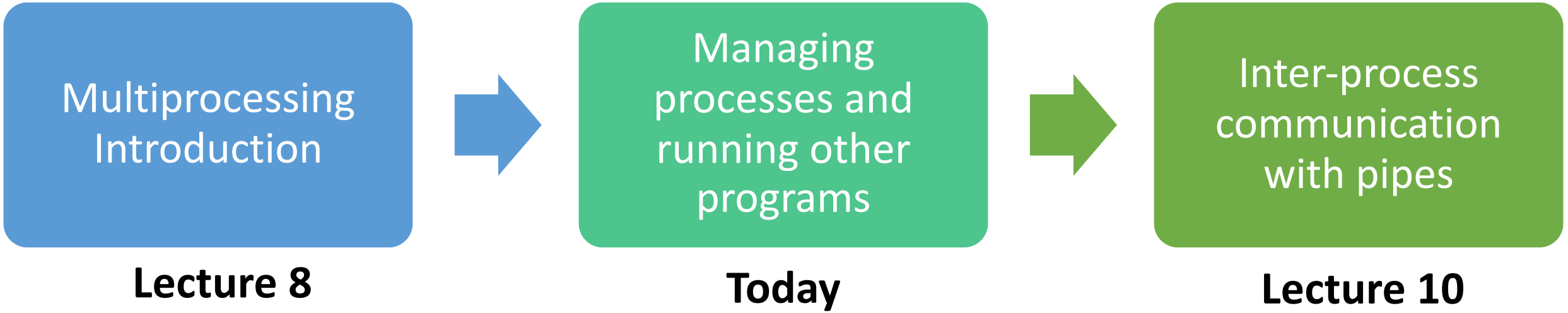
CS198 Section Leading!

cs198@cs.stanford.edu

cs198.stanford.edu – application due 10/20

Topic 2: Multiprocessing - How can our program create and interact with other programs? How does the operating system manage user programs?

CS111 Topic 2: Multiprocessing



assign3: implement your own shell!

Learning Goals

- Understand how a process is cloned and run by the OS
- 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.

Plan For Today

- **Recap**: **fork()**
- Cloning Processes
- **waitpid()** and waiting for child processes
- **Demo:** waiting for children
- **execvp()**

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

Plan For Today

- **Recap**: **fork()**
- Cloning Processes
- **waitpid()** and waiting for child processes
- **Demo:** waiting for children
- **execvp()**

```
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");
```

fork()

fork() is used pervasively in applications and systems. For example:

- A shell forks a new process to run an entered program command
- Most network servers run many copies of the server in different processes
- When your kernel boots, it starts the **system.d** program, which forks off all the services and systems for your computer

Processes are the first step in understanding *concurrency*, another key principle in computing systems.

fork()

```c
int main(int argc, char *argv[]) {
    printf("Hello from process %d! (parent %d)\n", getpid(), getppid());
    pid_t pidOrZero = fork();
    assert(pidOrZero >= 0);
    printf("Bye from process %d! (parent %d)\n", getpid(), getppid());
    return 0;
}
```

```
$ ./intro-fork
Hello from process 29686! (parent 29351)
Bye from process 29686! (parent 29351)
Bye from process 29687! (parent 29686)

$ ./intro-fork
Hello from process 29688! (parent 29351)
Bye from process 29689! (parent 29688)
Bye from process 29688! (parent 29351)
```

- The parent of the original process is the *shell* - the program that you run in the terminal.
- The ordering of the parent and child output is *up to the OS!*

Which of these outputs is not possible?

```
// Assume parent PID 111, child PID 112
pid_t pidOrZero = fork();
printf("hello, world!\n");
printf("goodbye! (fork returned %d)\n", pidOrZero);
```

A)

hello, world!
hello, world!
goodbye! (fork returned 0)
goodbye! (fork returned 112)

C)

hello, world!
goodbye! (fork returned 112)
hello, world!
goodbye! (fork returned 0)

B)

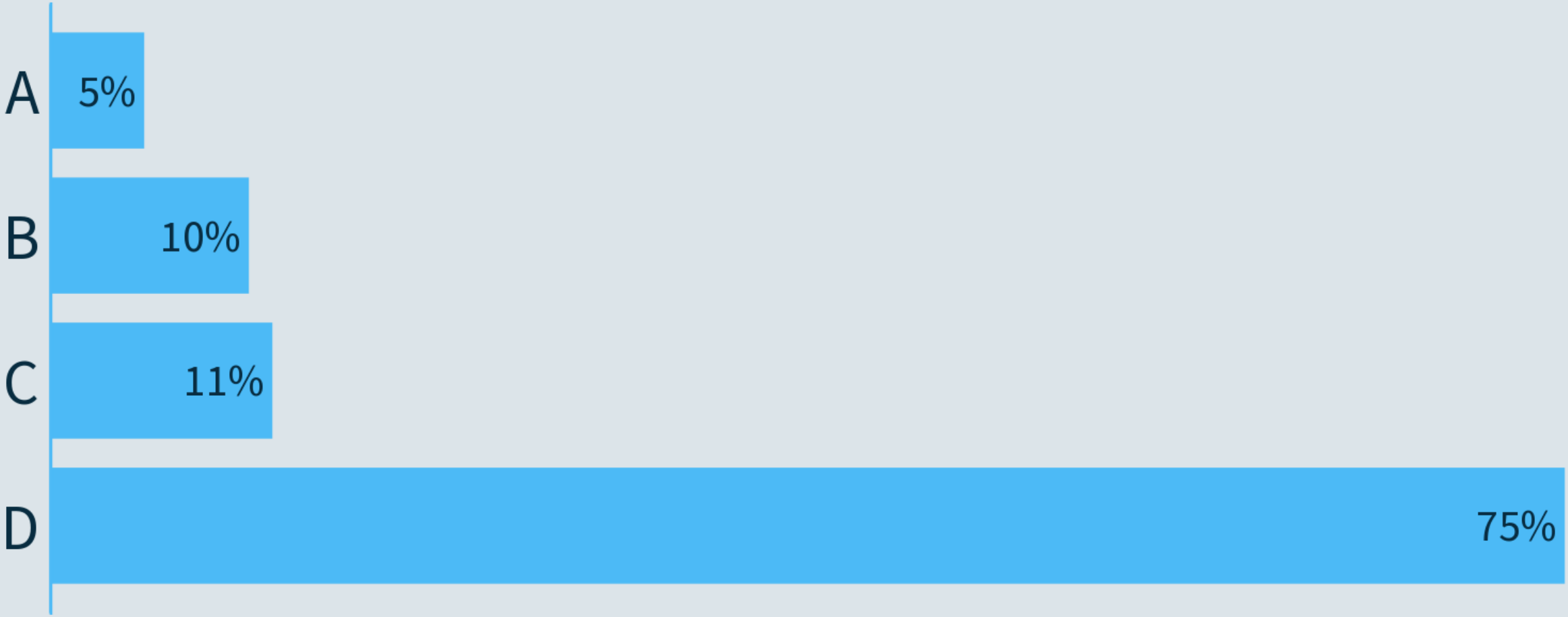
hello, world!
hello, world!
goodbye! (fork returned 112)
goodbye! (fork returned 0)

D)

hello, world!
goodbye! (fork returned 112)
goodbye! (fork returned 0)
hello, world!

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

Which of these outputs is *not* possible?



Plan For Today

- Recap: fork()
- **Cloning Processes**
- **waitpid()** and waiting for child processes
- **Demo:** waiting for children
- **execvp()**

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

What happens to variables/addresses?

```
int main(int argc, char *argv[]) {
    char str[128];
    strcpy(str, "Hello");
    printf("str's address is %p\n", str);
    pid_t pidOrZero = fork();
    if (pidOrZero == 0) { // The child should modify str
        printf("I am the child. str's address is %p\n", str);
        strcpy(str, "Howdy");
        printf("I am the child and I changed str to %s. str's address is
                still %p\n", str, str);
    } else { // The parent should sleep and print out str
        printf("I am the parent. str's address is %p\n", str);
        printf("I am the parent, and I'm going to sleep for 2sec.\n");
        sleep(2);
        printf("I am the parent. I just woke up. str's address is %p,
                and its value is %s\n", str, str);
    }
    return 0;
}
```

fork-copy.c

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
- The operating system will map the child's virtual addresses to different physical addresses than for the parent

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
```

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).

Plan For Today

- Recap: **fork()**
- Cloning Processes
- **waitpid()** and waiting for child processes
- **Demo:** waiting for children
- **execvp()**

```
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()

A parent process should always wait on its children processes.

- A process that finished but 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
- Child processes whose parent process terminates without waiting on them get the **init** process (PID 1) as their parent.

Make sure to reap your zombie children.

(wait, what?)

Plan For Today

- Recap: **fork()**
- Cloning Processes
- **waitpid()** and waiting for child processes
- **Demo: waiting for children**
- **execvp()**

```
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?

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* 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.

Let's see a demo!

reap-as-they-exit.c

Plan For Today

- Recap: **fork()**
- Cloning Processes
- **waitpid()** and waiting for child processes
- **Demo:** waiting for children
- **execvp()**

```
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 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 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
```

Implementing 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**)

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**

Recap

- **Recap**: **fork()**
- Cloning Processes
- **waitpid()** and waiting for child processes
- **Demo:** waiting for children
- **execvp()**

Next time: making our own shell, and how to have multiple processes communicate with pipes.

Lecture 9 takeaway:
processes can be run by the OS in any order. **waitpid** lets a parent process wait for a child process to finish.
execvp runs another program in the current process, completely cannibalizing the current process.