CS111, Lecture 8
Multiprocessing Introduction

Optional reading:

Operating Systems: Principles and Practice (2nd Edition): Chapter 4

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

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

Why is answering this question important?

- Helps us understand how programs are spawned and run (e.g. shells, web servers)
- Introduces us to the challenges of *concurrency* – managing concurrent events
- Allows us to understand how shells work and implement our own!

assign3: implement your own shell program!

CS111 Topic 2: Multiprocessing

Multiprocessing Introduction	→	Managing processes and running other programs	→	Inter-process communication with pipes
Today		**Lecture 9**		**Lecture 10 / 11**

assign3: implement your own shell!

Learning Goals

- Learn how to use the **fork()** function to create a new process
- Understand how a process is cloned and run by the OS
- Learn how to use **waitpid()** to wait for a child process to finish.

Plan For Today

- Multiprocessing overview
- Introducing **fork()**
- Cloning Processes

```
cp -r /afs/ir/class/cs111/lecture-code/lect8 .
```

Plan For Today

- **Multiprocessing overview**
- Introducing **fork()**
- Cloning Processes

```
cp -r /afs/ir/class/cs111/lecture-code/lect8 .
```

Multiprocessing Terminology

Program: code you write to execute tasks

Process: an instance of your program running; consists of program and execution state.

<u>Key idea:</u> multiple processes can run the same program

<u>Process 5621</u>

```
int main(int argc, char *argv[]) {
    printf("Hello, world!\n");
    printf("Goodbye!\n");
    return 0;
}
```

Multiprocessing

Your computer runs many processes simultaneously - even with just 1 processor core (how?)

- "simultaneously" = switch between them so fast humans don't notice
- Your program thinks it's the only thing running
- OS *schedules* tasks - who gets to run when
- Each gets a little time, then has to wait
- Many times, waiting is good! E.g. waiting for key press, waiting for disk
- *Caveat*: multicore computers can truly multitask

Playing with Processes

When you run a program from the terminal, it runs in a new process.

- The OS gives each process a unique "process ID" number (PID)
- PIDs are useful once we start managing multiple processes
- **getpid()** returns the PID of the current process (**pid_t** is a numeric type)

```c
// getpid.c
#include <stdio.h>
#include <unistd.h>
int main(int argc, char *argv[]) {
	pid_t myPid = getpid();
	printf("My process ID is %d\n", myPid);
	return 0;
}
```

```
$ ./getpid
My process ID is 18814

$ ./getpid
My process ID is 18831
```

Plan For Today

- Multiprocessing overview
- **Introducing fork()**
- Cloning Processes

```
cp -r /afs/ir/class/cs111/lecture-code/lect8 .
```

Fork is a system call that creates a second process which is a clone of the first.

fork()

fork() creates a second process that is a **clone** of the first:

```
pid_t fork();
```

Process A

```
int main(int argc, char *argv[]) {
    printf("Hello, world!\n");
    fork();
    printf("Goodbye!\n");
    return 0;
}
```

```
$ ./myprogram
```

fork()

fork() creates a second process that is a **clone** of the first:

```
pid_t fork();
```

Process A

```
int main(int argc, char *argv[]) {
    printf("Hello, world!\n");
    fork();
    printf("Goodbye!\n");
    return 0;
}
```

```
$ ./myprogram
Hello, world!
```

fork()

fork() creates a second process that is a **clone** of the first:

```
pid_t fork();
```

Process A

```
int main(int argc, char *argv[]) {
    printf("Hello, world!\n");
    fork();
    printf("Goodbye!\n");
    return 0;
}
```

```
$ ./myprogram
Hello, world!
```

fork()

fork() creates a second process that is a **clone** of the first: **`pid_t fork();`**

Process A

```c
int main(int argc, char *argv[]) {
    printf("Hello, world!\n");
    fork();
    printf("Goodbye!\n");
    return 0;
}
```

Process B

```c
int main(int argc, char *argv[]) {
    printf("Hello, world!\n");
    fork();
    printf("Goodbye!\n");
    return 0;
}
```

```
$ ./myprogram
Hello, world!
```

fork()

fork() creates a second process that is a **clone** of the first: **`pid_t fork();`**

Process A

```
int main(int argc, char *argv[]) {
    printf("Hello, world!\n");
    fork();
    printf("Goodbye!\n");
    return 0;
}
```

Process B

```
int main(int argc, char *argv[]) {
    printf("Hello, world!\n");
    fork();
    printf("Goodbye!\n");
    return 0;
}
```

```
$ ./myprogram
Hello, world!
Goodbye!
Goodbye!
```

fork()

fork() creates a second process that is a **clone** of the first:

```
pid_t fork();
```

Process A

```
int main(int argc, char *argv[]) {
    int x = 2;
    printf("Hello, world!\n");
    fork();
    printf("Goodbye, %d!\n", x);
    return 0;
}
```

```
$ ./myprogram
```

fork()

fork() creates a second process that is a **clone** of the first: **pid_t fork();**

Process A

```
int main(int argc, char *argv[]) {
    int x = 2;
    printf("Hello, world!\n");
    fork();
    printf("Goodbye, %d!\n", x);
    return 0;
}
```

```
$ ./myprogram
Hello, world!
```

fork()

fork() creates a second process that is a **clone** of the first: **`pid_t fork();`**

Process A

```
int main(int argc, char *argv[]) {
    int x = 2;
    printf("Hello, world!\n");
    fork();
    printf("Goodbye, %d!\n", x);
    return 0;
}
```

Process B

```
int main(int argc, char *argv[]) {
    int x = 2;
    printf("Hello, world!\n");
    fork();
    printf("Goodbye, %d!\n", x);
    return 0;
}
```

```
$ ./myprogram
Hello, world!
```

fork()

fork() creates a second process that is a **clone** of the first: **pid_t fork();**

Process A

```
int main(int argc, char *argv[]) {
    int x = 2;
    printf("Hello, world!\n");
    fork();
    printf("Goodbye, %d!\n", x);
    return 0;
}
```

Process B

```
int main(int argc, char *argv[]) {
    int x = 2;
    printf("Hello, world!\n");
    fork();
    printf("Goodbye, %d!\n", x);
    return 0;
}
```

```
$ ./myprogram
Hello, world!
Goodbye, 2!
Goodbye, 2!
```

fork()

fork() creates a second process that is a **clone** of the first: `pid_t fork();`

- **parent** (original) process forks off a **child** (new) process
- The child **starts** execution on the next program instruction. The parent **continues** execution with the next program instruction. *The order from now on is up to the OS!*
- **fork()** is called once, but returns twice (why?)

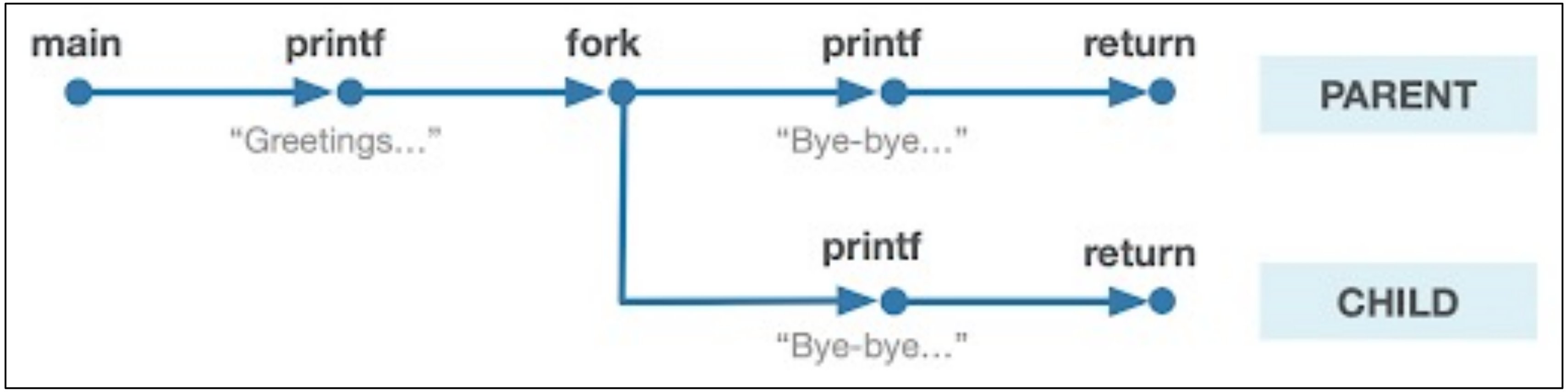


Illustration courtesy of Roz Cyrus.

fork()

fork() creates a second process that is a **clone** of the first: `pid_t fork();`

- **parent** (original) process forks off a **child** (new) process
- A child process could also then later call **fork**, thus being a parent itself
- Everything is duplicated in the child process (except PIDs are different)
  - File descriptor table - this explains how the child can still output to the same terminal!
  - Mapped memory regions (the address space) - regions like stack, heap, etc. are copied

fork()

(Am I the parent
or the child?)

Process A

```c
int main(int argc, char *argv[]) {
    int x = 2;
    printf("Hello, world!\n");
    fork();
    printf("Goodbye, %d!\n", x);
    return 0;
}
```

Process B

```c
int main(int argc, char *argv[]) {
    int x = 2;
    printf("Hello, world!\n");
    fork();
    printf("Goodbye, %d!\n", x);
    return 0;
}
```

Is there a way for the processes to tell which is the parent and which is the child?

fork()

<u>Key Idea:</u> **the return value of fork() is different in the parent and the child.**

fork() creates a second process that is a **<u>clone</u>** of the first: `pid_t fork();`

- **parent** (original) process forks off a **child** (new) process
- In the **parent**, **fork()** will return the PID of the child (only way for parent to get child's PID)
- In the **child, fork()** will return 0 (this is not the child's PID, it's just 0)
- This allows us to assign different tasks to the parent and child

fork()

In the **parent**, **fork()** will return the PID of the child. In the **child, fork()** will return 0 (this is not the child's PID, it's just 0).

<u>Process 111</u>

```
int main(int argc, char *argv[]) {
    printf("Hello, world!\n");
    pid_t pidOrZero = fork();
    printf("fork returned %d\n",
            pidOrZero);
    return 0;
}
```

```
$ ./myprogram
```

fork()

In the **parent**, **fork()** will return the PID of the child. In the **child, fork()** will return 0 (this is not the child's PID, it's just 0).

Process 111

```
int main(int argc, char *argv[]) {
    printf("Hello, world!\n");
    pid_t pidOrZero = fork();
    printf("fork returned %d\n",
            pidOrZero);
    return 0;
}
```

```
$ ./myprogram
Hello, world!
```

fork()

In the **parent**, **fork()** will return the PID of the child. In the **child, fork()** will return 0 (this is not the child's PID, it's just 0).

Process 111

```
int main(int argc, char *argv[]) {
    printf("Hello, world!\n");
    pid_t pidOrZero = fork();
    printf("fork returned %d\n",
           pidOrZero);
    return 0;
}
```

Process 112

```
int main(int argc, char *argv[]) {
    printf("Hello, world!\n");
    pid_t pidOrZero = fork();
    printf("fork returned %d\n",
           pidOrZero);
    return 0;
}
```

```
$ ./myprogram
Hello, world!
```

fork()

In the **parent**, **fork()** will return the PID of the child. In the **child, fork()** will return 0 (this is not the child's PID, it's just 0).

Process 111

```
int main(int argc, char *argv[]) {
    printf("Hello, world!\n");
    pid_t pidOrZero = fork();
    printf("fork returned %d\n",
            pidOrZero);
    return 0;
}
```

Process 112

```
int main(int argc, char *argv[]) {
    printf("Hello, world!\n");
    pid_t pidOrZero = fork();
    printf("fork returned %d\n",
            pidOrZero);
    return 0;
}
```

```
$ ./myprogram
Hello, world!
fork returned 112
fork returned 0
```

fork()

In the **parent**, **fork()** will return the PID of the child. In the **child, fork()** will return 0 (this is not the child's PID, it's just 0).

Process 111

```c
int main(int argc, char *argv[]) {
    printf("Hello, world!\n");
    pid_t pidOrZero = fork();
    printf("fork returned %d\n",
            pidOrZero);
    return 0;
}
```

Process 112

```c
int main(int argc, char *argv[]) {
    printf("Hello, world!\n");
    pid_t pidOrZero = fork();
    printf("fork returned %d\n",
            pidOrZero);
    return 0;
}
```

```
$ ./myprogram
Hello, world!
fork returned 112
fork returned 0
```

OR

```
$ ./myprogram
Hello, world!
fork returned 0
fork returned 112
```

We can no longer assume the order in which our program will execute! The OS decides the order.

fork()

- In the **parent**, **fork()** will return the PID of the child
- In the **child, fork()** will return 0 (this is not the child's PID, it's just 0)
- if **fork()** returns < 0, that means an error occurred
- **getppid()** gets the PID of your parent and **getpid()** gets your own PID
- **fork** allows us to implement a *shell* - 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.
  - shell (parent) forks off child process to run a command you enter. When you run a command, its parent is the shell.
  - **Key Idea**: we can only run one program per process, so to keep the shell running we need to run the user's command in another process.

fork()

```c
pid_t pidOrZero = fork();
if (pidOrZero == 0) {
    // Only executed by the child
} else {
    // Only executed by the parent
}

// Executed by both parent and child (if they get here)
```

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
            ???
}
```

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 <u>not</u> 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?

A

0%

B

0%

C

0%

D

0%

Processes all the way down

Even a child process can call **fork** to spawn its own child process!

```
int main(int argc, char *argv[]) {
    printf("Hello!\n");
    fork();
    printf("Howdy!\n");
    fork();
    printf("Hey there!\n");
    return 0;
}
```

- How many total processes are there (including the parent) in this program? 🤯
- How many times is each printf statement printed?

Processes all the way down

Even a child process can call **fork** to spawn its own child process!

```
int main(int argc, char *argv[]) {
    printf("Hello!\n");
    fork();
    printf("Howdy!\n");
    fork();
    printf("Hey there!\n");
    return 0;
}
```

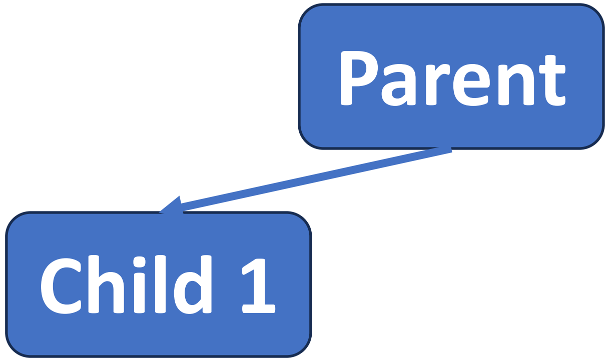
Parent

- How many total processes are there (including the parent) in this program? 🤯
- How many times is each printf statement printed?

Processes all the way down

Even a child process can call **fork** to spawn its own child process!

```c
int main(int argc, char *argv[]) {
    printf("Hello!\n");
    fork();
    printf("Howdy!\n");
    fork();
    printf("Hey there!\n");
    return 0;
}
```

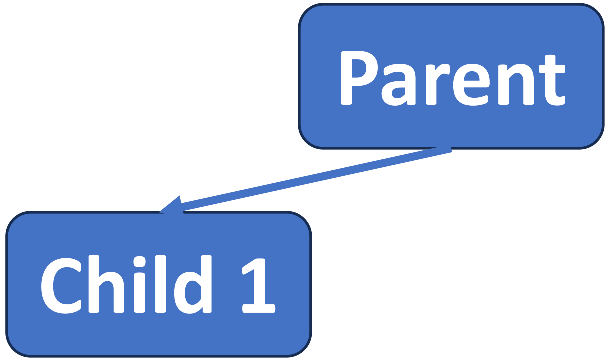


- How many total processes are there (including the parent) in this program?
- How many times is each printf statement printed?

Processes all the way down

Even a child process can call **fork** to spawn its own child process!

```
int main(int argc, char *argv[]) {
    printf("Hello!\n");
    fork();
    printf("Howdy!\n");
    fork();
    printf("Hey there!\n");
    return 0;
}
```

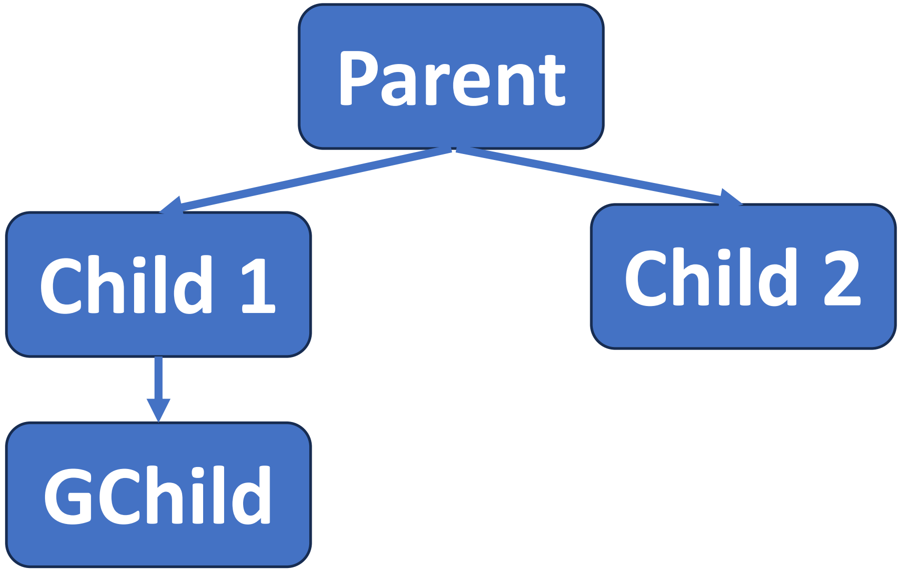


- How many total processes are there (including the parent) in this program?
- How many times is each printf statement printed?

Processes all the way down

Even a child process can call **fork** to spawn its own child process!

```
int main(int argc, char *argv[]) {
    printf("Hello!\n");
    fork();
    printf("Howdy!\n");
    fork();
    printf("Hey there!\n");
    return 0;
}
```

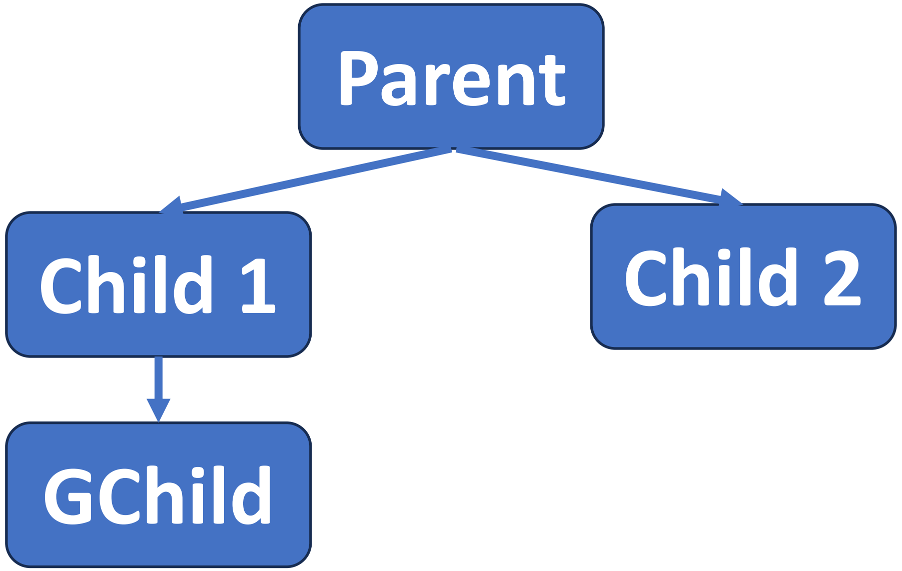


- How many total processes are there (including the parent) in this program? 🤯
- How many times is each printf statement printed?

Processes all the way down

Even a child process can call **fork** to spawn its own child process!

```
int main(int argc, char *argv[]) {
    printf("Hello!\n");
    fork();
    printf("Howdy!\n");
    fork();
    printf("Hey there!\n");
    return 0;
}
```

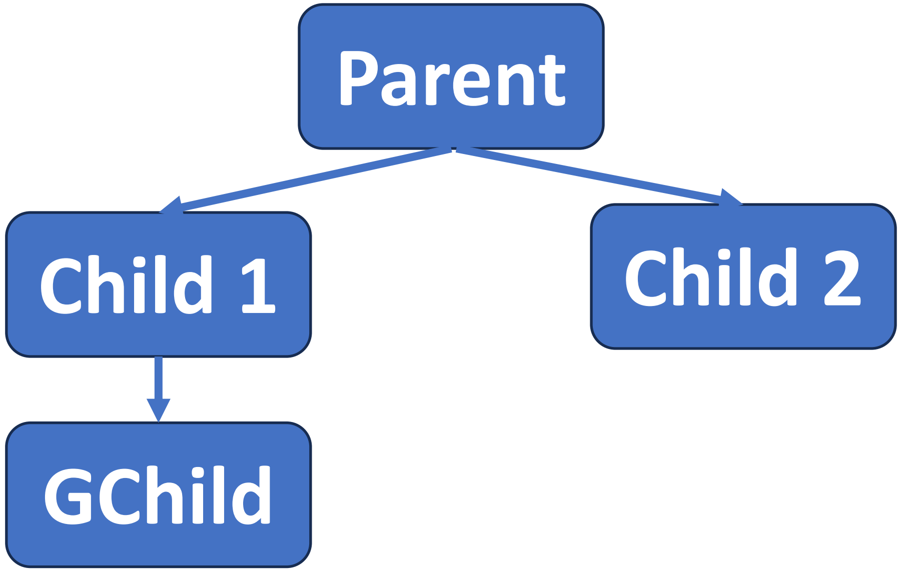


- How many total processes are there (including the parent) in this program? 🤯
- How many times is each printf statement printed?

Processes all the way down

Even a child process can call **fork** to spawn its own child process!

```
int main(int argc, char *argv[]) {
    printf("Hello!\n");
    fork();
    printf("Howdy!\n");
    fork();
    printf("Hey there!\n");
    return 0;
}
```

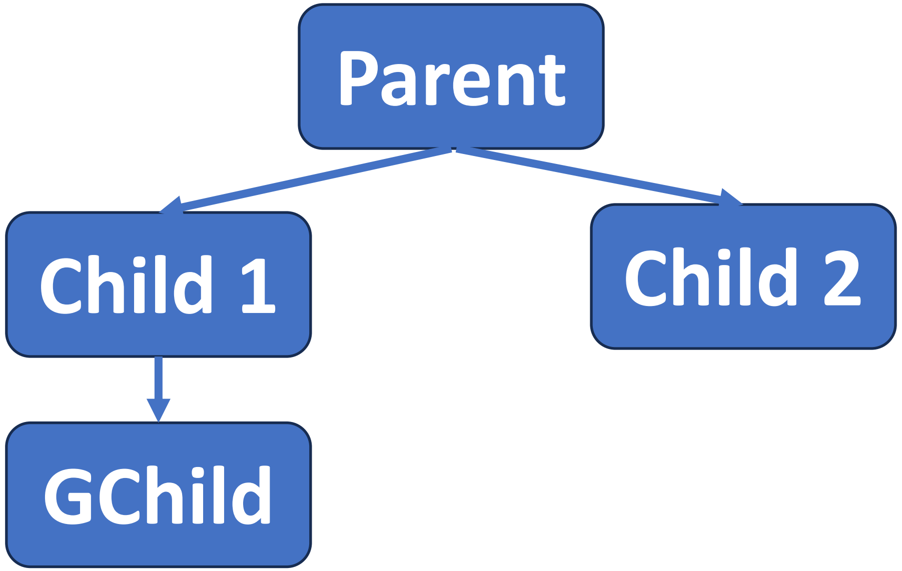


- How many total processes are there (including the parent) in this program? 🤯
- How many times is each printf statement printed?

Processes all the way down

Even a child process can call **fork** to spawn its own child process!

```c
int main(int argc, char *argv[]) {
    printf("Hello!\n");
    fork();
    printf("Howdy!\n");
    fork();
    printf("Hey there!\n");
    return 0;
}
```



- How many total processes are there (including the parent) in this program? 🤯
  - 4
- How many times is each printf statement printed?
  - Hello x 1, Howdy x 2, Hey there x 4, could be intermingled

Plan For Today

- Multiprocessing overview
- Introducing **fork()**
- **Cloning Processes**

```
cp -r /afs/ir/class/cs111/lecture-code/lect8 .
```

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).

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.

How can we stall our program until the child is finished? (next time!)

Recap

- Multiprocessing overview
- Introducing **fork()**
- Cloning Processes

Lecture 8 takeaway: fork() allows a process to fork off a cloned child process. The order of execution between parent and child is up to the OS! We can distinguish between parent and child using fork's return value (child PID in parent, 0 in child).

Next time: waiting on a child process, plus how to run other programs