CS110 Lecture 6: Multiprocessing

CS110: Principles of Computer Systems

Winter 2021-2022

Stanford University

Instructors: Nick Troccoli and Jerry Cain

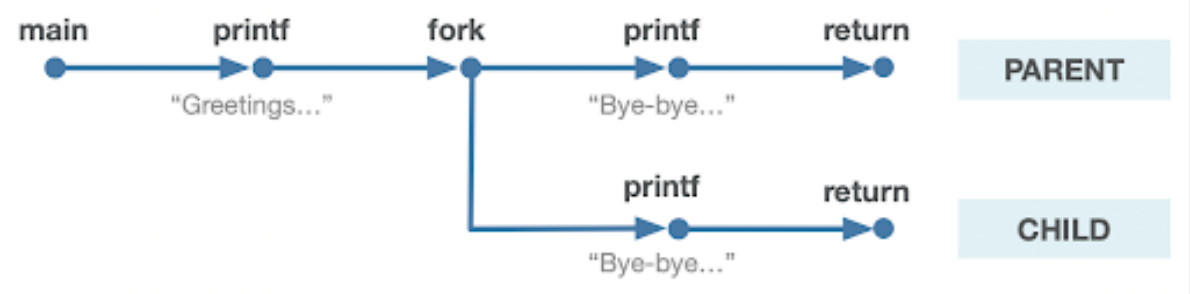
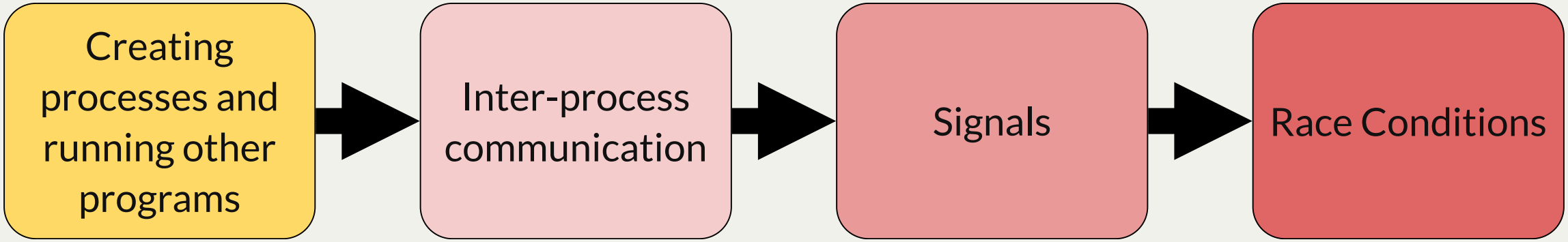


Illustration courtesy of Roz Cyrus.

PDF of this presentation

CS110 Topic 2: How can our program create and interact with other programs?

Learning About Processes



This/next lecture

Lecture 8/9

Lecture 10/11

Lecture 11

assign3: implement multiprocessing programs like "trace" (to trace another program's behavior) and "farm" (parallelize tasks)

assign4: 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
- Understand how to use **waitpid()** to coordinate between processes

Lecture Plan

- Multiprocessing overview
- Introducing **fork()**
- **Practice:** Fork Tree
- **waitpid()** and waiting for child processes

Today's Ed Thread: https://edstem.org/us/courses/16701/discussion/1002824

Lecture Plan

- **Multiprocessing overview**
- Introducing **fork()**
- **Practice:** Fork Tree
- **waitpid()** and waiting for child processes

Today's Ed Thread: https://edstem.org/us/courses/16701/discussion/1002824

Multiprocessing Terminology

Program: code you write to execute tasks

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

Key idea: multiple processes can run the same program

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* processes - who gets to run when
- Each process 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

```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
```

Lecture Plan

- Multiprocessing overview
- **Introducing fork()**
- **Practice**: Fork Tree
- **waitpid()** and waiting for child processes

Today's Ed Thread: https://edstem.org/us/courses/16701/discussion/1002824

fork()

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

Process A

```
1 int main(int argc, char *argv[]) {
2     printf("Hello, world!\n");
3     fork();
4     printf("Goodbye!\n");
5     return 0;
6 }
```

```
$ ./myprogram
```

fork()

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

Process A

```
1 int main(int argc, char *argv[]) {
2     printf("Hello, world!\n");
3     fork();
4     printf("Goodbye!\n");
5     return 0;
6 }
```

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

fork()

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

```
pid_t fork();
```

Process A

```
1 int main(int argc, char *argv[]) {
2     printf("Hello, world!\n");
3     fork();
4     printf("Goodbye!\n");
5     return 0;
6 }
```

Process B

```
1 int main(int argc, char *argv[]) {
2     printf("Hello, world!\n");
3     fork();
4     printf("Goodbye!\n");
5     return 0;
6 }
```

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

fork()

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

```
pid_t fork();
```

Process A

```
1 int main(int argc, char *argv[]) {
2     printf("Hello, world!\n");
3     fork();
4     printf("Goodbye!\n");
5     return 0;
6 }
```

Process B

```
1 int main(int argc, char *argv[]) {
2     printf("Hello, world!\n");
3     fork();
4     printf("Goodbye!\n");
5     return 0;
6 }
```

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

fork()

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

Process A

```
1 int main(int argc, char *argv[]) {
2     int x = 2;
3     printf("Hello, world!\n");
4     fork();
5     printf("Goodbye, %d!\n", x);
6     return 0;
7 }
```

```
$ ./myprogram2
```

fork()

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

Process A

```
1 int main(int argc, char *argv[]) {
2     int x = 2;
3     printf("Hello, world!\n");
4     fork();
5     printf("Goodbye, %d!\n", x);
6     return 0;
7 }
```

```
$ ./myprogram2
Hello, world!
```

fork()

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

Process A

```
1 int main(int argc, char *argv[]) {
2     int x = 2;
3     printf("Hello, world!\n");
4     fork();
5     printf("Goodbye, %d!\n", x);
6     return 0;
7 }
```

Process B

```
1 int main(int argc, char *argv[]) {
2     int x = 2;
3     printf("Hello, world!\n");
4     fork();
5     printf("Goodbye, %d!\n", x);
6     return 0;
7 }
```

```
$ ./myprogram2
Hello, world!
```

fork()

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

Process A

```
1 int main(int argc, char *argv[]) {
2     int x = 2;
3     printf("Hello, world!\n");
4     fork();
5     printf("Goodbye, %d!\n", x);
6     return 0;
7 }
```

Process B

```
1 int main(int argc, char *argv[]) {
2     int x = 2;
3     printf("Hello, world!\n");
4     fork();
5     printf("Goodbye, %d!\n", x);
6     return 0;
7 }
```

```
$ ./myprogram2
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?)
- Everything is duplicated in the child process (except PIDs are different)
  - File descriptor table (increasing reference counts on open file table entries)
  - Mapped memory regions (the address space)
  - Regions like stack, heap, etc. are copied

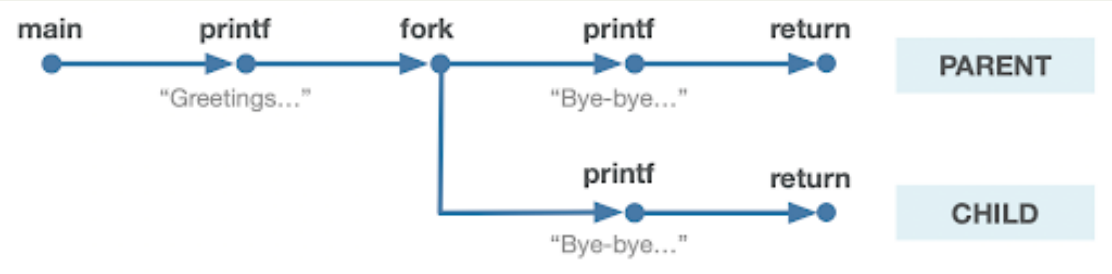


Illustration courtesy of Roz Cyrus.

Process Clones

The parent process' file descriptor table is **cloned** on **fork** and the reference counts within the relevant open file table entries are incremented. This explains how the child can still output to the same terminal!

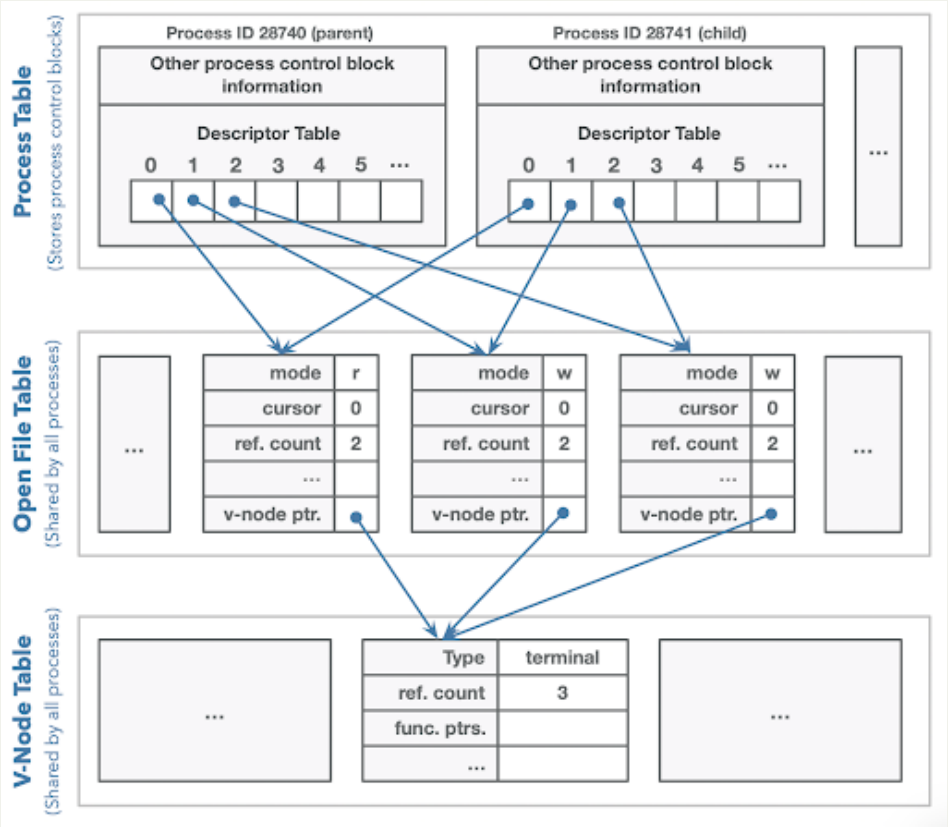
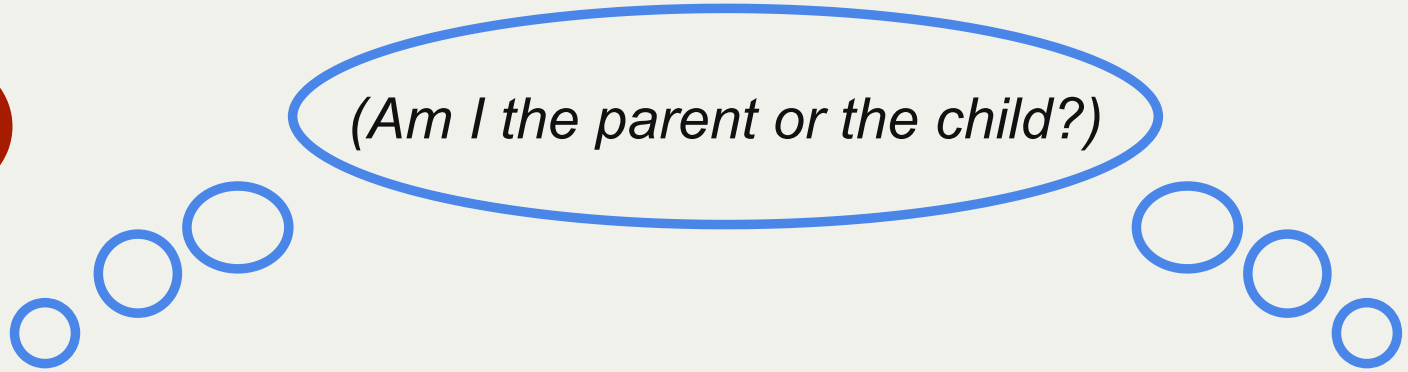


Illustration courtesy of Roz Cyrus.

fork()



Process A

```
1 int main(int argc, char *argv[]) {
2     int x = 2;
3     printf("Hello, world!\n");
4     fork();
5     printf("Goodbye, %d!\n", x);
6     return 0;
7 }
```

Process B

```
1 int main(int argc, char *argv[]) {
2     int x = 2;
3     printf("Hello, world!\n");
4     fork();
5     printf("Goodbye, %d!\n", x);
6     return 0;
7 }
```

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

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

`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
- 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)

fork()

- 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)

Process 110

```
1 int main(int argc, char *argv[]) {
2     printf("Hello, world!\n");
3     pid_t pidOrZero = fork();
4     printf("fork returned %d\n", pidOrZero);
5     return 0;
6 }
```

```
$ ./myprogram
```

fork()

- 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)

Process 110

```
1 int main(int argc, char *argv[]) {
2     printf("Hello, world!\n");
3     pid_t pidOrZero = fork();
4     printf("fork returned %d\n", pidOrZero);
5     return 0;
6 }
```

```
$ ./myprogram2
Hello, world!
```

fork()

- 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)

Process 110

```
1 int main(int argc, char *argv[]) {
2     printf("Hello, world!\n");
3     pid_t pidOrZero = fork(); // 111
4     printf("fork returned %d\n", pidOrZero);
5     return 0;
6 }
```

Process 111

```
1 int main(int argc, char *argv[]) {
2     printf("Hello, world!\n");
3     pid_t pidOrZero = fork(); // 0
4     printf("fork returned %d\n", pidOrZero);
5     return 0;
6 }
```

```
$ ./myprogram2
Hello, world!
```

fork()

- 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)

Process 110

```
1 int main(int argc, char *argv[]) {
2     printf("Hello, world!\n");
3     pid_t pidOrZero = fork(); // 111
4     printf("fork returned %d\n", pidOrZero);
5     return 0;
6 }
```

Process 111

```
1 int main(int argc, char *argv[]) {
2     printf("Hello, world!\n");
3     pid_t pidOrZero = fork(); // 0
4     printf("fork returned %d\n", pidOrZero);
5     return 0;
6 }
```

```
$ ./myprogram
Hello, world!
fork returned 111
fork returned 0
```

fork()

- 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)

Process 110

```
1 int main(int argc, char *argv[]) {
2     printf("Hello, world!\n");
3     pid_t pidOrZero = fork(); // 111
4     printf("fork returned %d\n", pidOrZero);
5     return 0;
6 }
```

Process 111

```
1 int main(int argc, char *argv[]) {
2     printf("Hello, world!\n");
3     pid_t pidOrZero = fork(); // 0
4     printf("fork returned %d\n", pidOrZero);
5     return 0;
6 }
```

```
$ ./myprogram
Hello, world!
fork returned 111
fork returned 0
```

OR

```
$ ./myprogram
Hello, world!
fork returned 0
fork returned 111
```

fork()

- 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)

Process 110

```
1 int main(int argc, char *argv[]) {
2     printf("Hello, world!\n");
3     pid_t pidOrZero = fork(); // 111
4     printf("fork returned %d\n", pidOrZero);
5     return 0;
6 }
```

Process 111

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

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

```
$ ./myprogram
Hello, world!
fork returned 111
fork returned 0
```

OR

```
Hello, world!
fork returned 0
fork returned 111
```

fork()

- 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)
- A process can use **getppid()** to get the PID of its parent
- if **fork()** returns < 0, that means an error occurred

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

basic-fork.c

```
$ ./basic-fork
Greetings from process 29686! (parent 29351)
Bye-bye from process 29686! (parent 29351)
Bye-bye from process 29687! (parent 29686)

$ ./basic-fork
Greetings from process 29688! (parent 29351)
Bye-bye from process 29689! (parent 29688
Bye-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 *nondeterministic*. Sometimes the parent prints first, and sometimes the child prints first!

Process Clones

What happens to variables and addresses?

```c
int main(int argc, char *argv[]) {
    char str[128];
    strcpy(str, "Hello");
    printf("str's address is %p\n", str);

    pid_t pid = fork();

    if (pid == 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 2 seconds.\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 2 seconds.
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 2 seconds.
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).

Example: Loaded Dice

```c
int main(int argc, char *argv[]) {
    // Initialize the random number with a "seed value"
    // this seed state is used to generate future random numbers
    srandom(time(NULL));

    printf("This program will make you question what 'randomness' means...\n");
    pid_t pidOrZero = fork();

    // Parent goes first - both processes *always* get the same roll (why?)
    if (pidOrZero != 0) {
        int diceRoll = (random() % 6) + 1;
        printf("I am the parent and I rolled a %d\n", diceRoll);
        sleep(1);
    } else {
        sleep(1);
        int diceRoll = (random() % 6) + 1;
        printf("I am the child and I'm guessing the parent rolled a %d\n", diceRoll);
    }

    return 0;
}
```

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().*

not-so-random.c

Debugging Multiprocess Programs

How do I debug two processes at once? **`gdb`** has built-in support for debugging multiple processes

- **`set detach-on-fork off`**
  - This tells **`gdb`** to capture any **`fork`**'d processes, though it pauses them upon the **`fork`**.
- **`info inferiors`**
  - This lists the processes that **`gdb`** has captured.
- **`inferior X`**
  - Switch to a different process to debug it.
- **`detach inferior X`**
  - Tell **`gdb`** to stop watching the process, and continue it
- You can see an entire debugging session on the **`basic-fork`** program right here.

Lecture Plan

- Multiprocessing overview
- Introducing **fork()**
- **Practice: Fork Tree**
- **waitpid()** and waiting for child processes

Today's Ed Thread: https://edstem.org/us/courses/16701/discussion/1002824

Fork Tree

Here's a useful (but mind-melting) example of a program where child processes themselves call **fork()**:

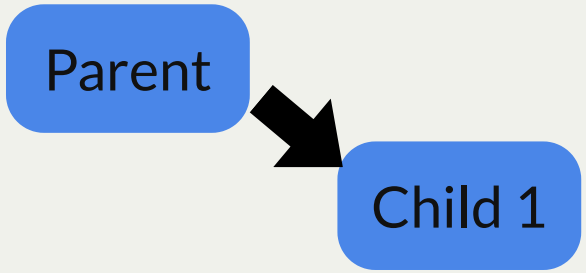
Parent

```
1  static const char *kTrail = "abc";
2
3  int main(int argc, char *argv[]) {
4      for (int i = 0; i < strlen(kTrail); i++) {
5          printf("%c\n", kTrail[i]);
6          pid_t pidOrZero = fork();
7          assert(pidOrZero >= 0);
8      }
9      return 0;
10 }
```

fork-puzzle.c

Fork Tree

Here's a useful (but mind-melting) example of a program where child processes themselves call **fork()**:



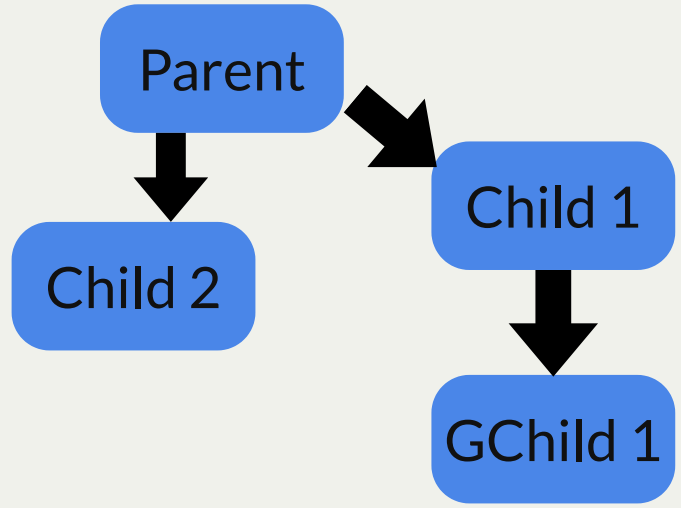

```c
// fork-puzzle.c
static const char *kTrail = "abc";

int main(int argc, char *argv[]) {
    for (int i = 0; i < strlen(kTrail); i++) {
        printf("%c\n", kTrail[i]);
        pid_t pidOrZero = fork();
        assert(pidOrZero >= 0);
    }
    return 0;
}
```


```c
// fork-puzzle.c
static const char *kTrail = "abc";

int main(int argc, char *argv[]) {
    for (int i = 0; i < strlen(kTrail); i++) {
        printf("%c\n", kTrail[i]);
        pid_t pidOrZero = fork();
        assert(pidOrZero >= 0);
    }
    return 0;
}
```

Fork Tree

Here's a useful (but mind-melting) example of a program where child processes themselves call **fork()**:

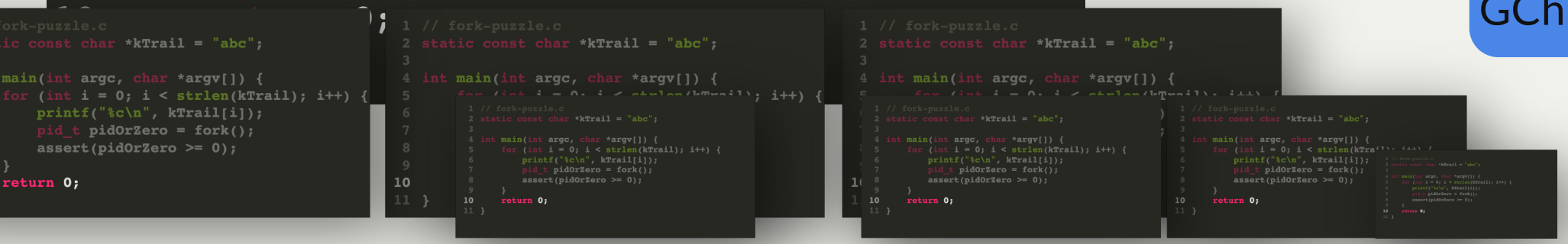
```c
// fork-puzzle.c
static const char *kTrail = "abc";

int main(int argc, char *argv[]) {
    for (int i = 0; i < strlen(kTrail); i++) {
        printf("%c\n", kTrail[i]);
        pid_t pidOrZero = fork();
        assert(pidOrZero >= 0);
    }
    return 0;
}
```

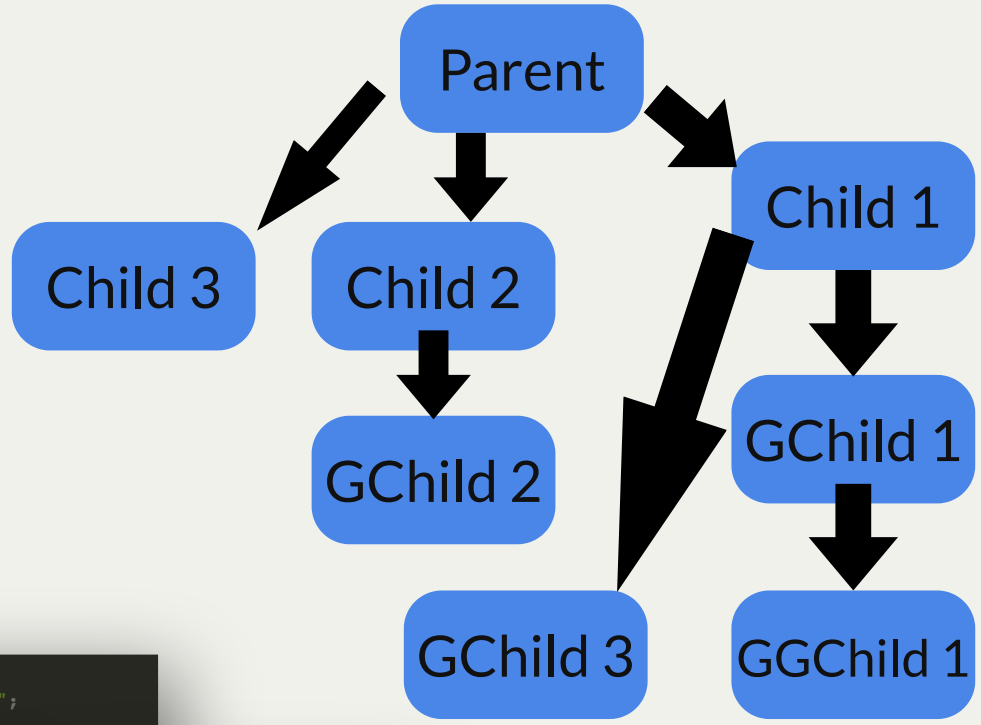


Fork Tree

Here's a useful (but mind-melting) example of a program where child processes themselves call **fork()**:

```c
// fork-puzzle.c
static const char *kTrail = "abc";

int main(int argc, char *argv[]) {
    for (int i = 0; i < strlen(kTrail); i++) {
        printf("%c\n", kTrail[i]);
        pid_t pidOrZero = fork();
        assert(pidOrZero >= 0);
    }
```





Fork Tree

```
 1 // fork-puzzle.c
 2 static const char *kTrail = "abc";
 3
 4 int main(int argc, char *argv[]) {
 5     for (int i = 0; i < strlen(kTrail); i++) {
 6         printf("%c\n", kTrail[i]);
 7         pid_t pidOrZero = fork();
 8         assert(pidOrZero >= 0);
 9     }
10     return 0;
11 }
```

```
$ ./fork-puzzle
a
b
c
b
c
c
c
```

```
$ ./fork-puzzle
a
b
b
c
c
c
c
```

```
$ ./fork-puzzle
a
b
c
b
c
c
$ c
```

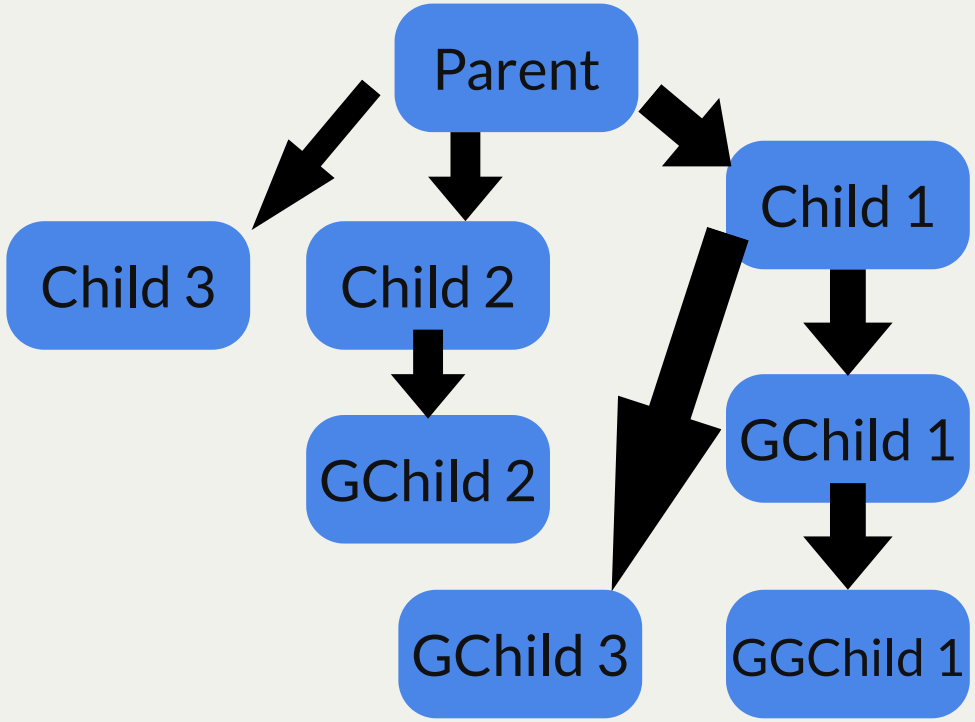
What happened here?

Observations:

- One **a** is printed by the original process.
- The original process and its child each print **b**.
- The two **b**s may not be consecutive - why?

Fork Tree

```
1  // fork-puzzle.c
2  static const char *kTrail = "abc";
3
4  int main(int argc, char *argv[]) {
5      for (int i = 0; i < strlen(kTrail); i++) {
6          printf("%c\n", kTrail[i]);
7          pid_t pidOrZero = fork();
8          assert(pidOrZero >= 0);
9      }
10     return 0;
11 }
```



Questions:

- **1 a** is printed.
- **2 b**s are printed.
- How many **c**s get printed? -> 4: parent, child 1, child 2, Gchild 1
- Who prints nothing? -> Child 3, GGchild 1, Gchild 3, Gchild 2

Why Fork?

- Fork is used pervasively in applications. A few examples:
  - Running a program in a shell: the shell forks a new process to run the program
  - Servers: most network servers run many copies of the server in different processes (why?)
- Fork is used pervasively in systems. A few examples:
  - When your kernel boots, it starts the system.d program, which forks off all of the services and systems for your computer
    - Let's take a look with pstree
  - Your window manager spawns processes when you start programs
  - Network servers spawn processes when they receive connections
    - E.g., when you ssh into myth, sshd spawns a process to run your shell in (after setting up file descriptors for your terminals over ssh)
- Processes are the first step in understanding *concurrency*, another key principle in computer systems; we'll look at other forms of concurrency later in the quarter

Review So Far

- A process is an instance of a program running
- Each process has a unique PID
- **fork**() creates a clone of the current process, and they run *concurrently*
- The parent and child are identical except for **fork**'s return value (child PID for parent, 0 for child)
- This *concurrency* lets your program multitask: much of the quarter will look at the complications
  - Nondeterministic ordering of execution across processes
  - A parent can wait for its children to terminate

Lecture Plan

- Multiprocessing overview
- Introducing **fork()**
- **Practice:** Fork Tree
- **waitpid() and waiting for child processes**

Today's Ed Thread: https://edstem.org/us/courses/16701/discussion/1002824

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()

```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 still waited up 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;
}
```

```
$ ./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()

Pass in the address of an integer as the second parameter to get the child's 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 110; // 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;
    }
}
```

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

waitpid-status.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.

Lecture Recap

- Multiprocessing overview
- Introducing **fork()**
- **Practice:** Fork Tree
- **waitpid()** and waiting for child processes

Next time: more **waitpid()**, **execvp()** and writing our first shell program

Extra Practice Problems

Practice: fork()

```c
int main(int argc, char *argv[]) {
    printf("Starting the program\n");
    pid_t pidOrZero1 = fork();
    pid_t pidOrZero2 = fork();

    if (pidOrZero1 != 0 && pidOrZero2 != 0) {
        printf("Hello\n");
    }

    if (pidOrZero2 != 0) {
        printf("Hi there\n");
    }

    return 0;
}
```

How many processes run in total?

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

How many times is "Hello" printed?

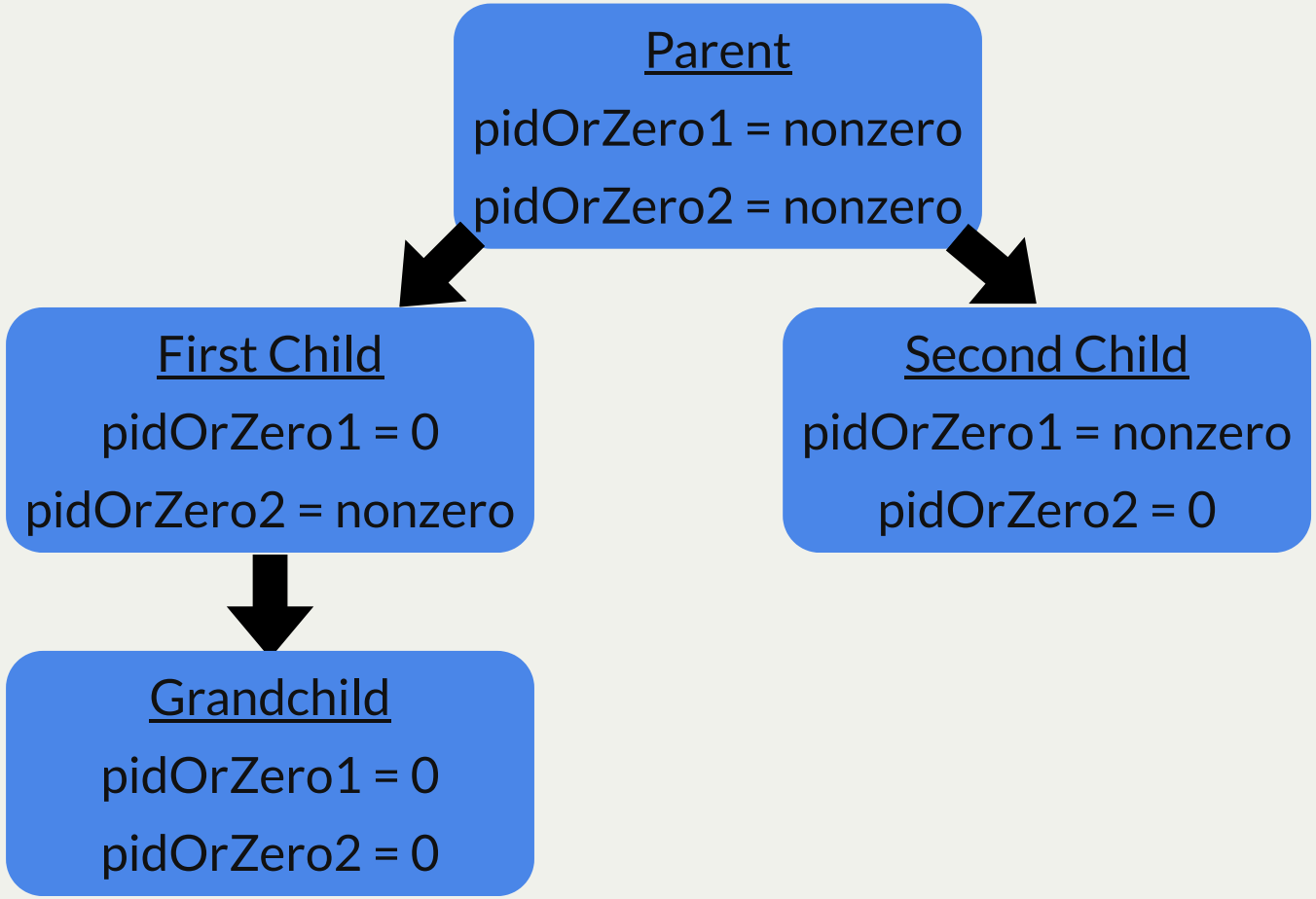
a) 1 b) 2 c) 3 d) 4

How many times is "Hi there" printed?

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

Practice: fork()

```c
int main(int argc, char *argv[]) {
    printf("Starting the program\n");
    pid_t pidOrZero1 = fork();
    pid_t pidOrZero2 = fork();

    if (pidOrZero1 != 0 && pidOrZero2 != 0) {
        printf("Hello\n");
    }

    if (pidOrZero2 != 0) {
        printf("Hi there\n");
    }

    return 0;
}
```



Fork Tree Round 2

```
 1 // fork-puzzle-full.c
 2 static const char *kTrail = "abcd";
 3
 4 int main(int argc, char *argv[]) {
 5     for (int i = 0; i < strlen(kTrail); i++) {
 6         printf("%c\n", kTrail[i]);
 7         pid_t pidOrZero = fork();
 8         assert(pidOrZero >= 0);
 9     }
10     return 0;
11 }
```

fork-puzzle-full.c

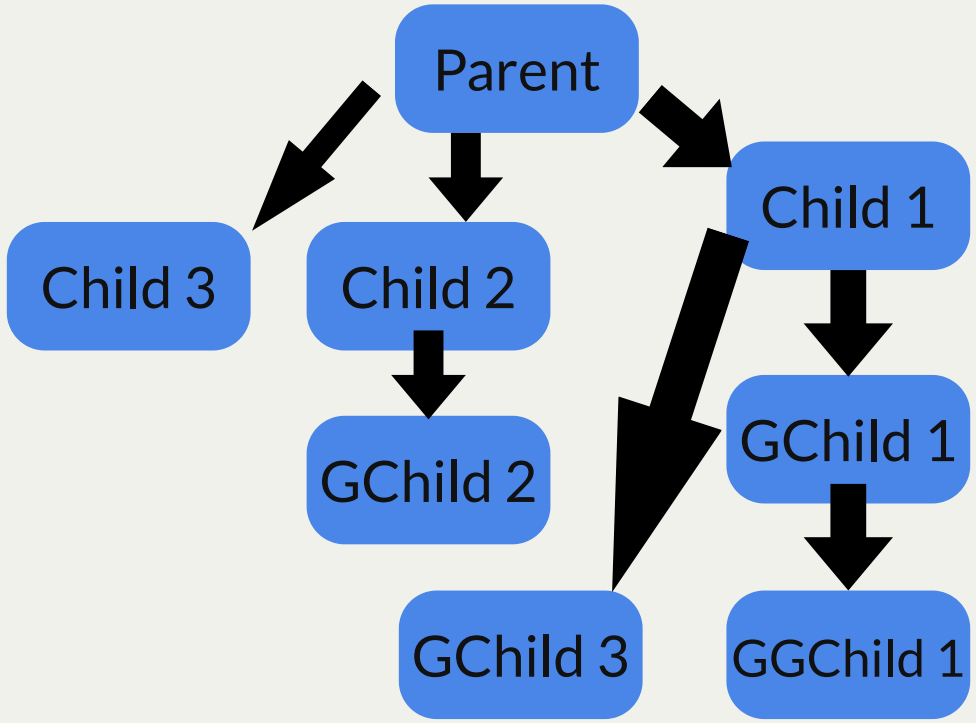
Questions:

- How many total processes are there when running this program?
- How many times is **d** printed?
- Could a **d** be printed before an: "a"? "b"? "c"?
- How many processes don't print anything?

Fork Tree Round 2

```
1  // fork-puzzle-full.c
2  static const char *kTrail = "abcd";
3
4  int main(int argc, char *argv[]) {
5      for (int i = 0; i < strlen(kTrail); i++) {
6          printf("%c\n", kTrail[i]);
7          pid_t pidOrZero = fork();
8          assert(pidOrZero >= 0);
9      }
10     return 0;
11 }
```

From earlier fork tree:



Questions:

- How many total processes are there when running this program?
- How many times is **d** printed?
- Could a **d** be printed before an: "a"? "b"? "c"?
- How many processes don't print anything?

- 16 total processes
- **d** is printed 8 times
- before a "b" or "c"
- 8 processes print nothing