Lecture 04: Files, Memory, and Processes

Principles of Computer Systems

Winter 2021

Stanford University

Computer Science Department

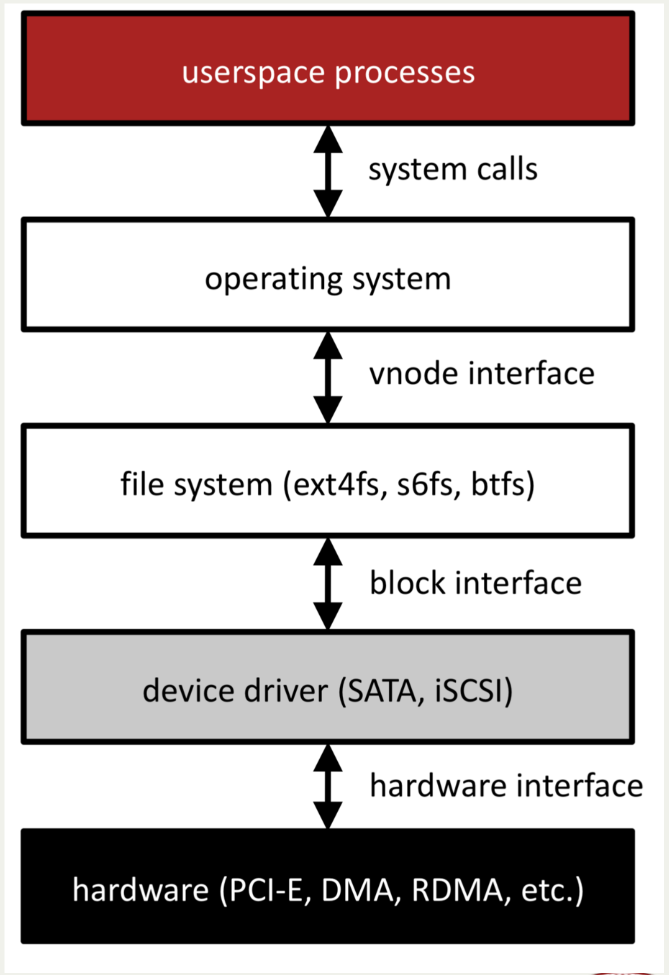
Lecturer: Chris Gregg and

Nick Troccoli

PDF of this presentation

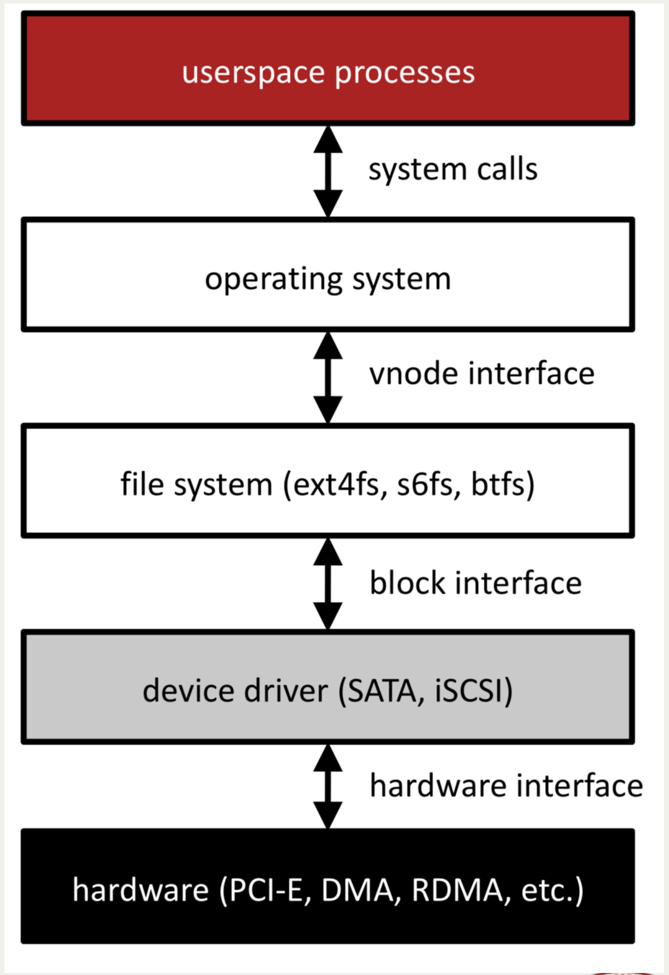
Recap of Lectures 1-3

- You've seen how a file system (e.g., the System 6 file system, s6fs) *layers* on top of a block device to present an *abstraction* of files and directories
- *Layering*: decomposing systems into components with well-defined responsibilities, specifying repcise APIs between them (above and below)
  - s6fs works on top of anything that provides a block interface: hard disk, solid state disk, RAM disk, loopback disk, etc.
  - Many different file systems can sit on top of a block device: s6fs, ext2fs, ext4fs, btfs, ntfs, etc., etc.
- *Abstraction*: defining an API of an underlying resource that is simultaneously simple to use, allows great flexibility in implementation, and can perform well
  - Userspace programs operate on files and directories
  - File system has great flexibility in how it represents files and directories on a disk
  - Not always perfect: block interface for flash and FTLs
- *Names and name resolution*: files are resources, directory entries (file names) are the way we name and refer to those resources

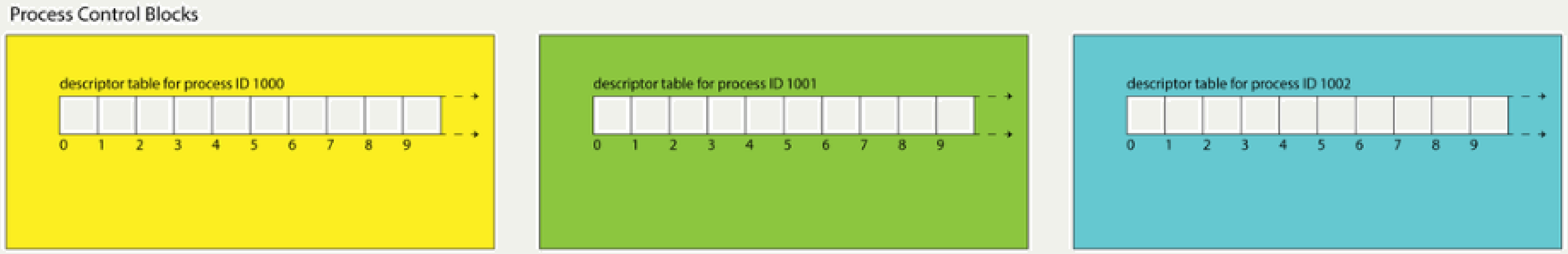


Today's Lecture

- How files on disk are presented to a program as file descriptors
- How programs open, control and manipulate files
  - How does the command below work?
  - `$cat people.txt | sort | uniq > list.txt`
- File descriptors vs. open files (many-to-one mapping)
- **`vnode`** abstraction of a file within the kernel
- Memory mapped files and the buffer cache
- The concept of a process and what it represents
  - Address space: virtualization of memory
  - Seamless thread(s) of execution: virtualization of CPU
- Creating and managing processes
- Concurrency: challenges when you have multiple processes running and how you manage them



File Descriptor Table and File Descriptors

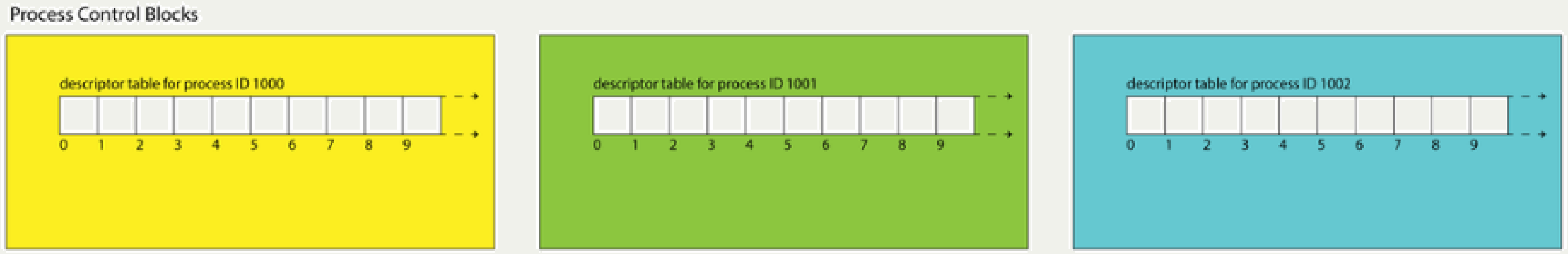

Process Control Blocks
descriptor table for process ID 1000
0 1 2 3 4 5 6 7 8 9
descriptor table for process ID 1001
0 1 2 3 4 5 6 7 8 9
descriptor table for process ID 1002
0 1 2 3 4 5 6 7 8 9

- Linux maintains a data structure for each active process. These data structures are called **`process control blocks`**, and they are stored in the **`process table`**
  - We'll explain exactly what a process is later in lecture

- Process control blocks store many things (the user who launched it, what time it was launched, CPU state, etc.). Among the many items it stores is the **`file descriptor table`**

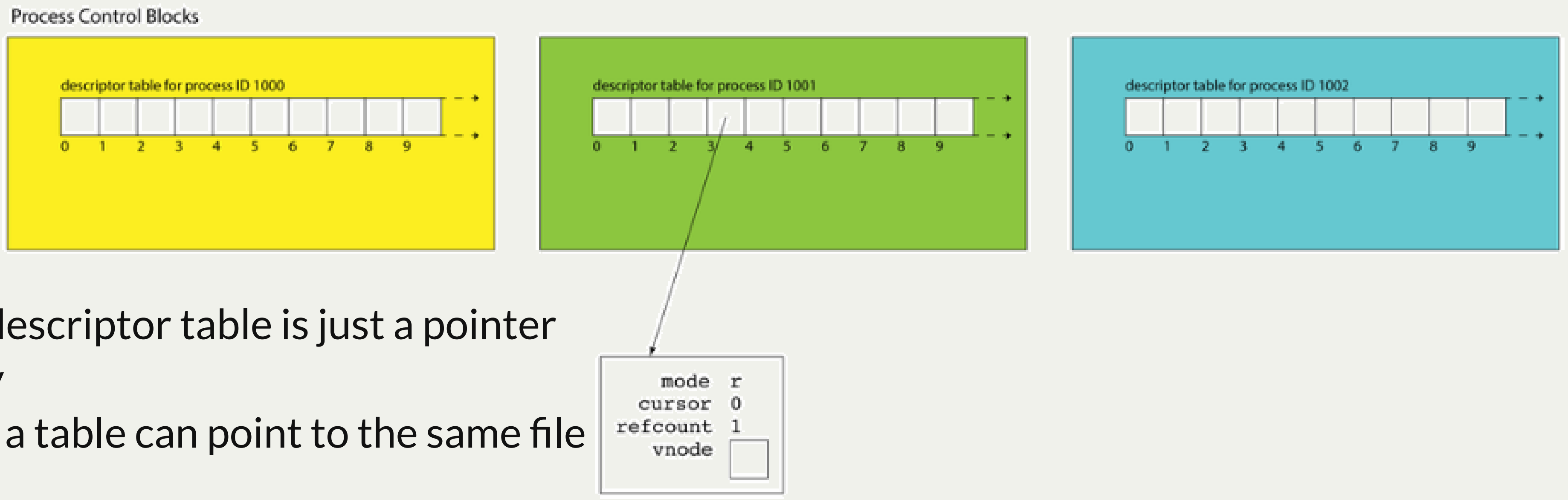
- A file descriptor (used by your program) is a small integer that's an index into this table
  - Descriptors 0, 1, and 2 are standard input, standard output, and standard error, but there are no predefined meanings for descriptors 3 and up. When you run a program from the terminal, descriptors 0, 1, and 2 are most often bound to the terminal

Creating and Using File Descriptors


Process Control Blocks
descriptor table for process ID 1000
0 1 2 3 4 5 6 7 8 9
descriptor table for process ID 1001
0 1 2 3 4 5 6 7 8 9
descriptor table for process ID 1002
0 1 2 3 4 5 6 7 8 9

- A file descriptor is the *identifier* needed to interact with a resource (most often a file) via system calls (e.g., **`read`**, **`write`**, and **`close`**)
- A *name* has semantic meaning, an *address* denotes a location; an *identifier* has no meaning
  - /etc/passwd vs.34.196.104.129 vs. file descriptor 5
- Many system calls allocate file descriptors
  - read: open a file
  - pipe: create two unidirectional byte streams (one read, one write) between processes
  - accept: accept a TCP connection request, returns descriptor to new socket
- When allocating a new file descriptor, kernel chooses the smallest available number
  - These semantics are important! If you close stdout (1) then open a file, it will be assigned to file descriptor 1 so act as stdout (this is how **`$ cat in.txt > out.txt`** works)

File Descriptor vs. File Table Entries


Process Control Blocks
descriptor table for process ID 1000
descriptor table for process ID 1001
descriptor table for process ID 1002
0 1 2 3 4 5 6 7 8 9
mode r
cursor 0
refcount 1
vnode

- A entry in the file descriptor table is just a pointer to a file table entry
- Multiple entries in a table can point to the same file table entry
- Entries in different file descriptor tables (different processes!) can point to the same file table entry

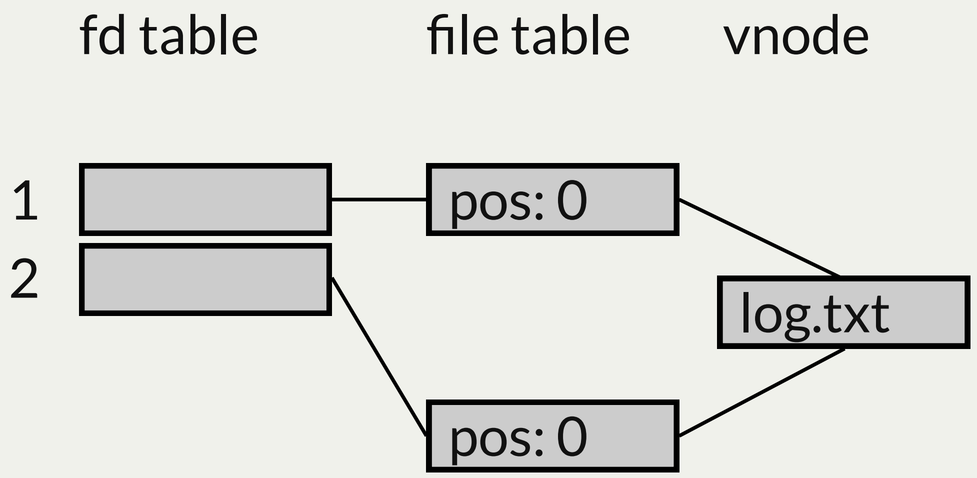
- E.g., a file table entry (for a regular file) keeps track of a current position in the file
  - If you read 1000 bytes, the next read will be from 1000 bytes after the preceding one
  - If you write 380 bytes, the next write will start 380 bytes after the preceding one
- If you want multiple processes to write to the same log file and have the results be intelligible, then you have all of them share a single file table entry: their calls to write will be serialized and occur in some linear order

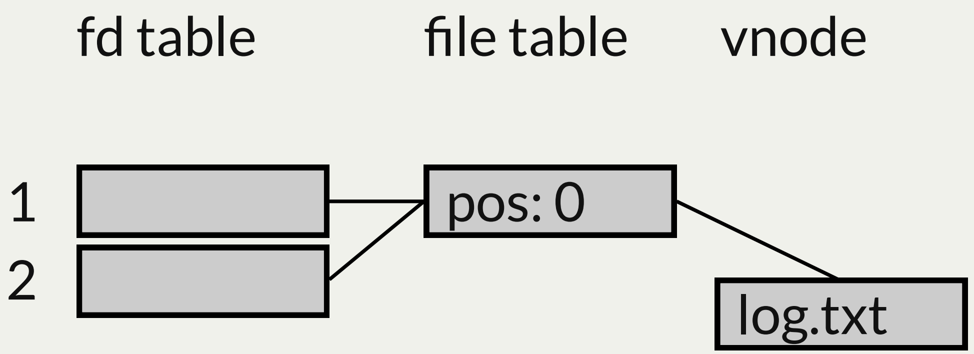
File Descriptors vs. File Table Entries Example

`$ ./main 1> log.txt 2> log.txt`	Opens log.txt twice (two file table entries)
`$ ./main 1> log.txt 2>&1`	Opens log.txt once, two descriptors for same file table entry

```c
// file: testfd.c
#include <stdio.h>
#include <unistd.h>
#include <string.h>

int main(int argc, char **argv)
{
    const char* error = "One plus one is\ntwo.\n";
    const char* msg   = "One plus two is\n";

    write(2, error, strlen(error));
    write(1, msg, strlen(msg));
    return 0;
}
```

File Descriptors vs. File Table Entries Example

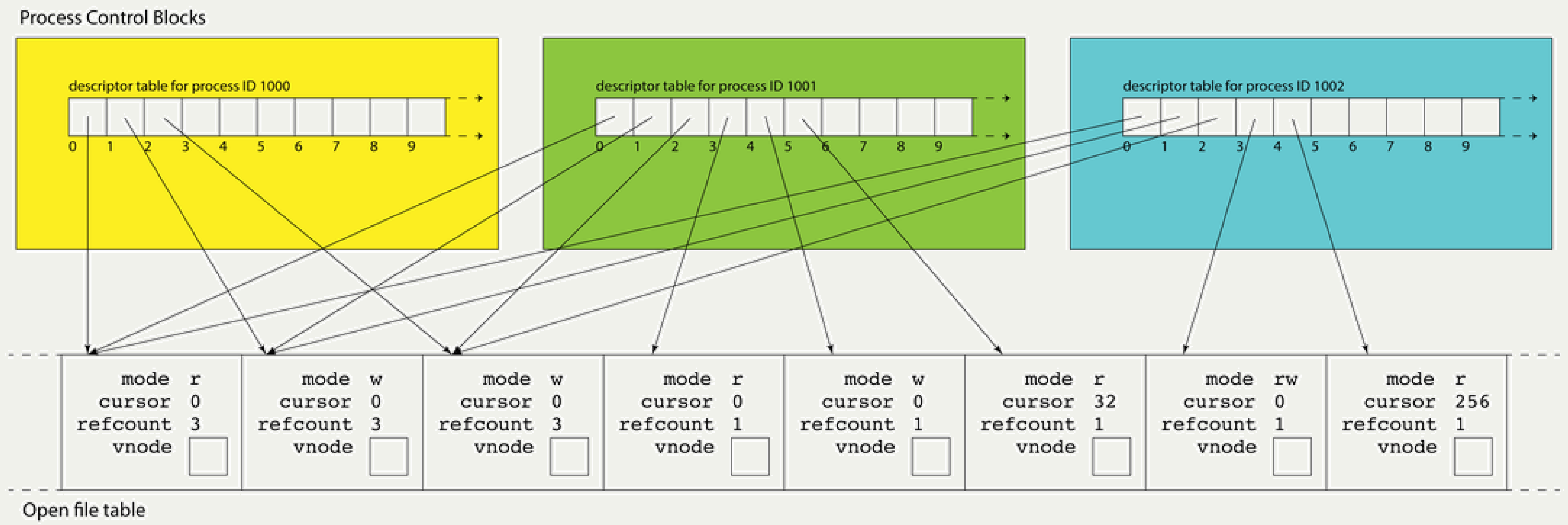




```
cgregg@myth60:$ ./testfd 1> log.txt 2> log.txt
cgregg@myth60:$ cat log.txt
One plus two is
two.
cgregg@myth60:$
```

```
cgregg@myth60:$ ./testfd 1> log.txt 2>&1
cgregg@myth60:$ cat log.txt
One plus one is
two.
One plus two is
cgregg@myth60:$
```

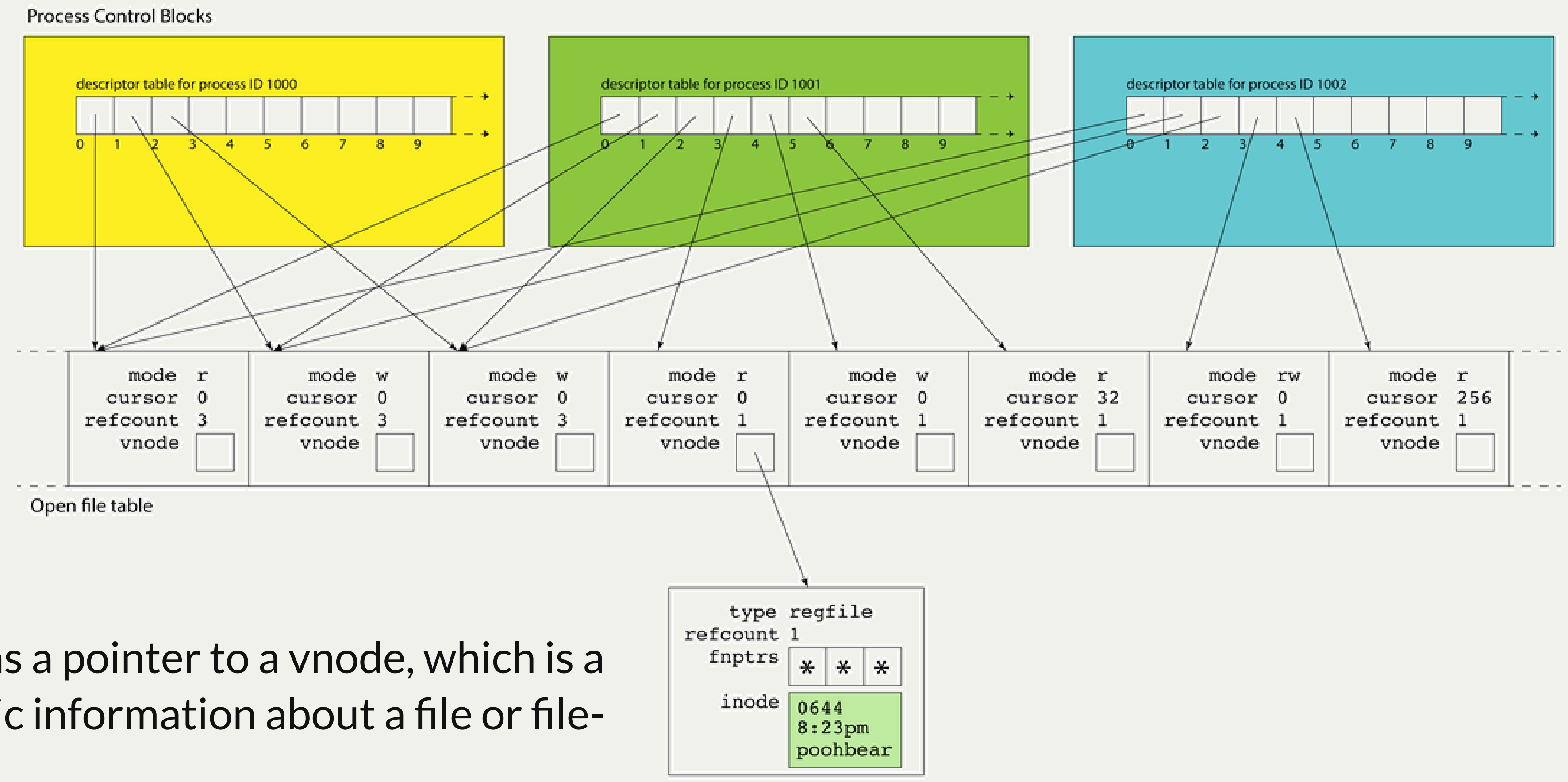
File Table Details



Process Control Blocks
descriptor table for process ID 1000
descriptor table for process ID 1001
descriptor table for process ID 1002
0 1 2 3 4 5 6 7 8 9
mode r cursor 0 refcount 3 vnode
mode w cursor 0 refcount 3 vnode
mode w cursor 0 refcount 3 vnode
mode r cursor 0 refcount 1 vnode
mode w cursor 0 refcount 1 vnode
mode r cursor 32 refcount 1 vnode
mode rw cursor 0 refcount 1 vnode
mode r cursor 256 refcount 1 vnode
Open file table

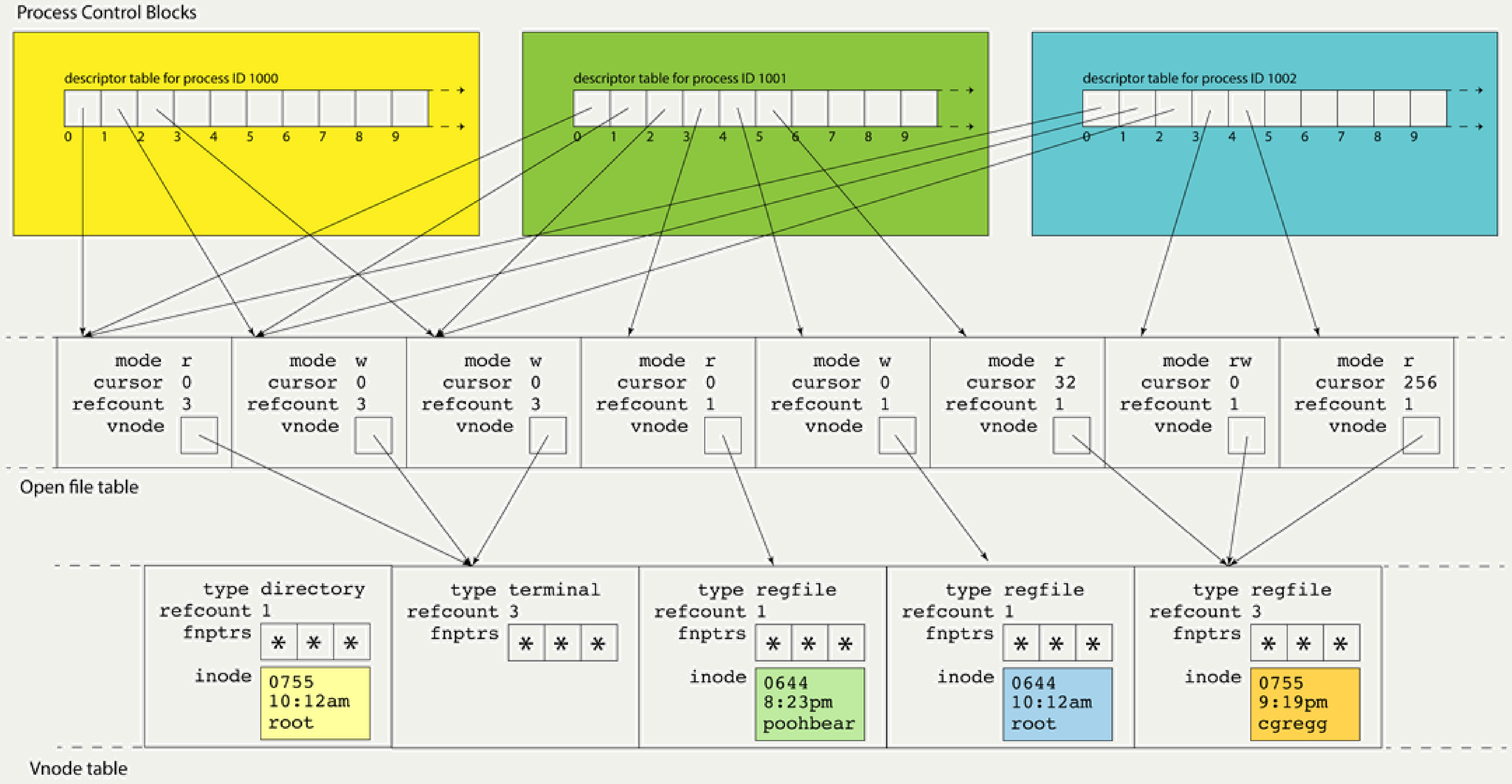
- Each process maintains its own descriptor table, but there is one, system-wide open file table. This allows for file resources to be shared between processes, as we've seen
- As drawn above, descriptors 0, 1, and 2 in each of the three PCBs alias the same three open files. That's why each of the referred table entries have refcounts of 3 instead of 1.
- This shouldn't surprise you. If your **`bash`** shell calls **`make`**, which itself calls **`g++`**, each of them inserts text into the same terminal window: those three files could be stdin, stdout, and stderr for a terminal

vnodes


Process Control Blocks
descriptor table for process ID 1000
descriptor table for process ID 1001
descriptor table for process ID 1002
0 1 2 3 4 5 6 7 8 9
mode r cursor 0 refcount 3 vnode
mode w cursor 0 refcount 3 vnode
mode w cursor 0 refcount 3 vnode
mode r cursor 0 refcount 1 vnode
mode w cursor 0 refcount 1 vnode
mode r cursor 32 refcount 1 vnode
mode rw cursor 0 refcount 1 vnode
mode r cursor 256 refcount 1 vnode
Open file table
type regfile
refcount 1
fnptrs * * *
inode 0644 8:23pm poohbear

- Each open file entry has a pointer to a vnode, which is a structure housing static information about a file or file-like resource.
- The vnode is the kernel's abstraction of an actual file: it includes information on what kind of file it is, how many file table entries reference it, and function pointers for performing operations.
- A vnode's interface is file-system independent, but its implementation is file-system specific; any file system (or file abstraction) can put state it needs to in the vnode (e.g., inode number)
- The term vnode comes from BSD UNIX; in Linux source it's called a *generic inode* (CONFUSING!)

File Decriptors -> File Table -> vnode Table

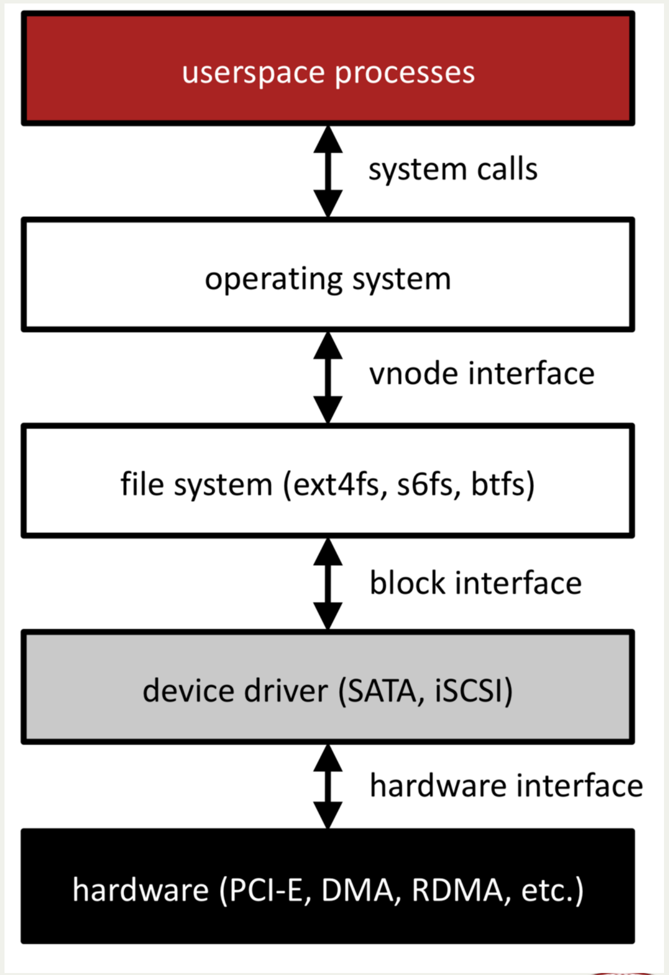


Process Control Blocks
descriptor table for process ID 1000
0 1 2 3 4 5 6 7 8 9
descriptor table for process ID 1001
0 1 2 3 4 5 6 7 8 9
descriptor table for process ID 1002
0 1 2 3 4 5 6 7 8 9
mode r cursor 0 refcount 3 vnode
mode w cursor 0 refcount 3 vnode
mode w cursor 0 refcount 3 vnode
mode r cursor 0 refcount 1 vnode
mode w cursor 0 refcount 1 vnode
mode r cursor 32 refcount 1 vnode
mode rw cursor 0 refcount 1 vnode
mode r cursor 256 refcount 1 vnode
Open file table
type directory refcount 1 fnptrs * * * inode 0755 10:12am root
type terminal refcount 3 fnptrs * * *
type regfile refcount 1 fnptrs * * * inode 0644 8:23pm poohbear
type regfile refcount 1 fnptrs * * * inode 0644 10:12am root
type regfile refcount 3 fnptrs * * * inode 0755 9:19pm cgregg
Vnode table

- There is one system-wide vnode table for the same reason there is one system-wide open file table. Independent file sessions reading from the same file don't need independent copies of the vnode. They can all alias the same one.

UNIX File Abstractions Summary

- Userspace programs interact through files through *file descriptors*, small integers which are indexes into a per-process *file descriptor table*
- Many file descriptors can point to the same *file table entry*
  - They share seek pointers (writes and reads are serialized)
  - Multiple programs share stdout/stderr in a terminal
- Many file table entries can point to the same file (*vnode*)
  - They concurrently access the file with different seek pointers
  - You run two instances of a python script in parallel: each invocation of Python opens the file separately, with a different file table entry
- Exactly how vnodes are implemented is filesystem/resource dependent
  - A terminal (tty) vnode is different than an ext4fs one
- Reference counting throughout
  - Free a file table entry when the last file descriptor closes it
  - Free a vnode when the last file table entry is freed
  - Free a file when its reference count is 0 and there is no vnode
- Key principles: *abstraction*, *layers*, *naming*



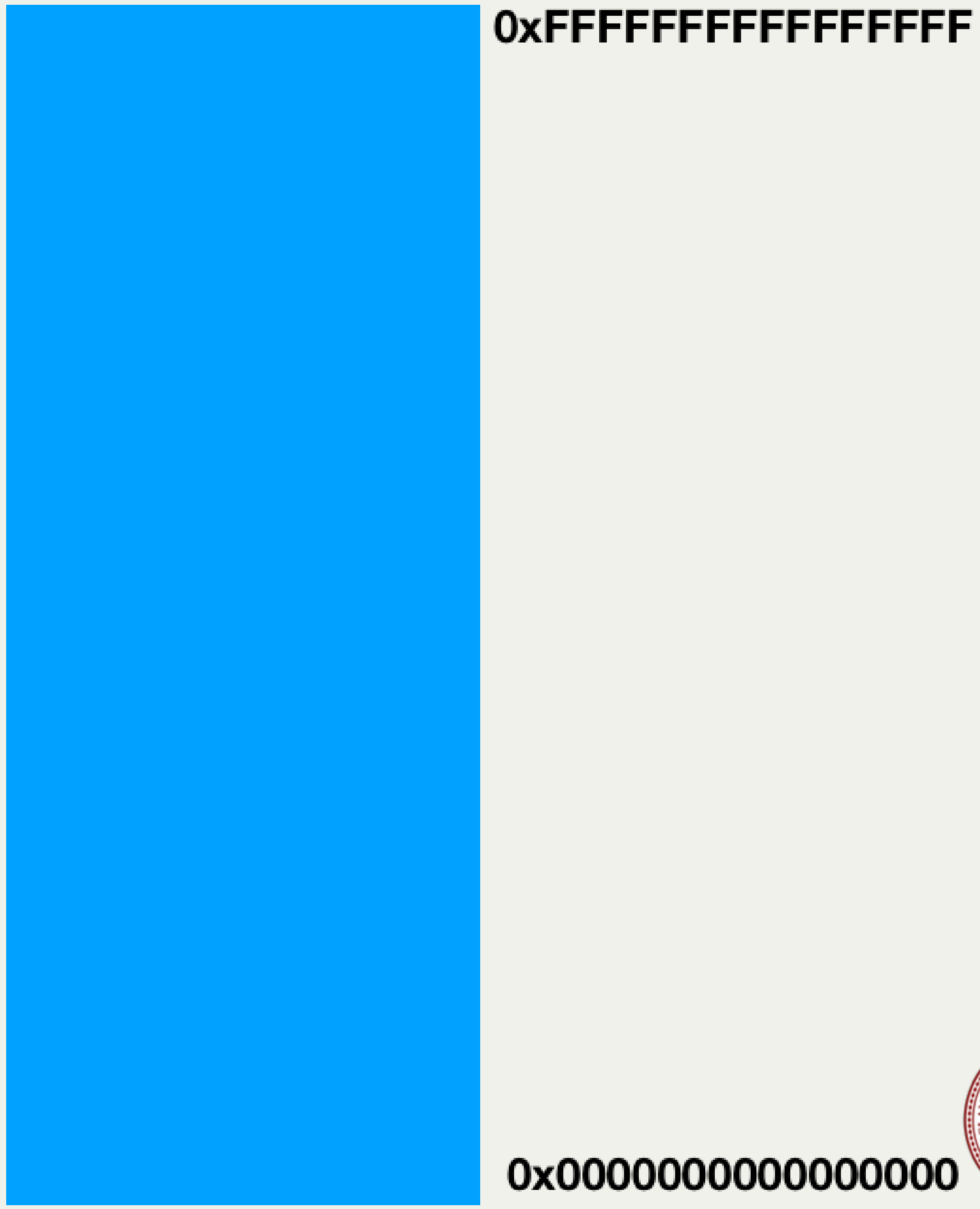
Memory mapped files

- **read(2)** is not the only way to for a program to access a file
- Read requires making a copy: program provides a buffer to read into
  - What if many programs want read-only access to the file at the same time?
- Example: libc.so
  - Almost program wants to read libc.so for some of the functions it provides
  - Imagine if every program had to read() all of its libraries into local memory
  - Let's use pmap to look at how much memory libraries take up

- Solution: memory mapped files
  - Ask the operating system: "please map this file into memory for me"

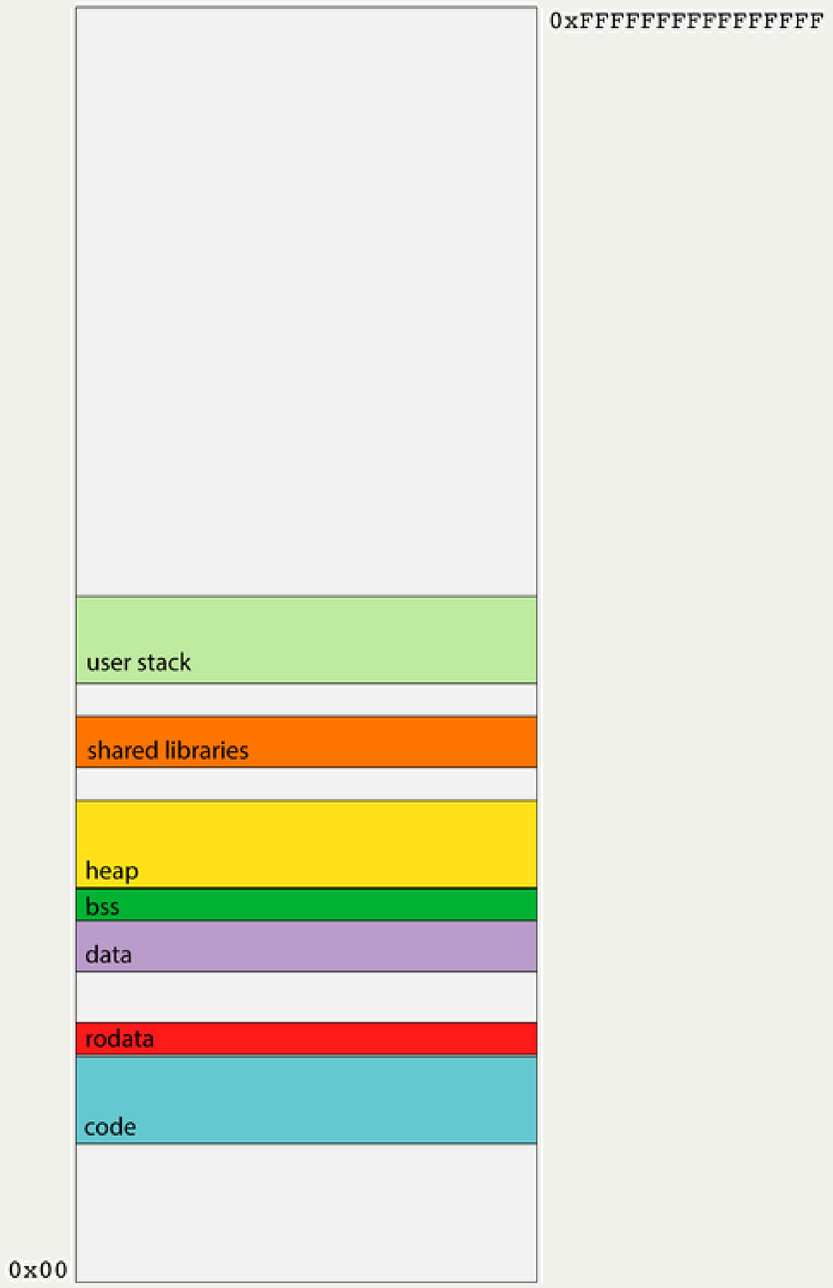
Process Address Spaces

- Recall that each process operates as if it owns all of main memory.
- The diagram on the right presents a 64-bit process's general memory playground that stretches from address 0 up through and including $2^{64} - 1$.
- CS107 and CS107-like intro-to-architecture courses present the diagram on the right, and discuss how various portions of the address space are cordoned off to manage traditional function call and return, dynamically allocated memory, access global data, and machine code storage and execution.
- No process actually uses all 2^{64} bytes of its address space. In fact, the vast majority of processes use a miniscule fraction of what they otherwise think they own.
- The OS *virtualizes* memory: each process thinks it as the complete system memory (but obviously it doesn't)



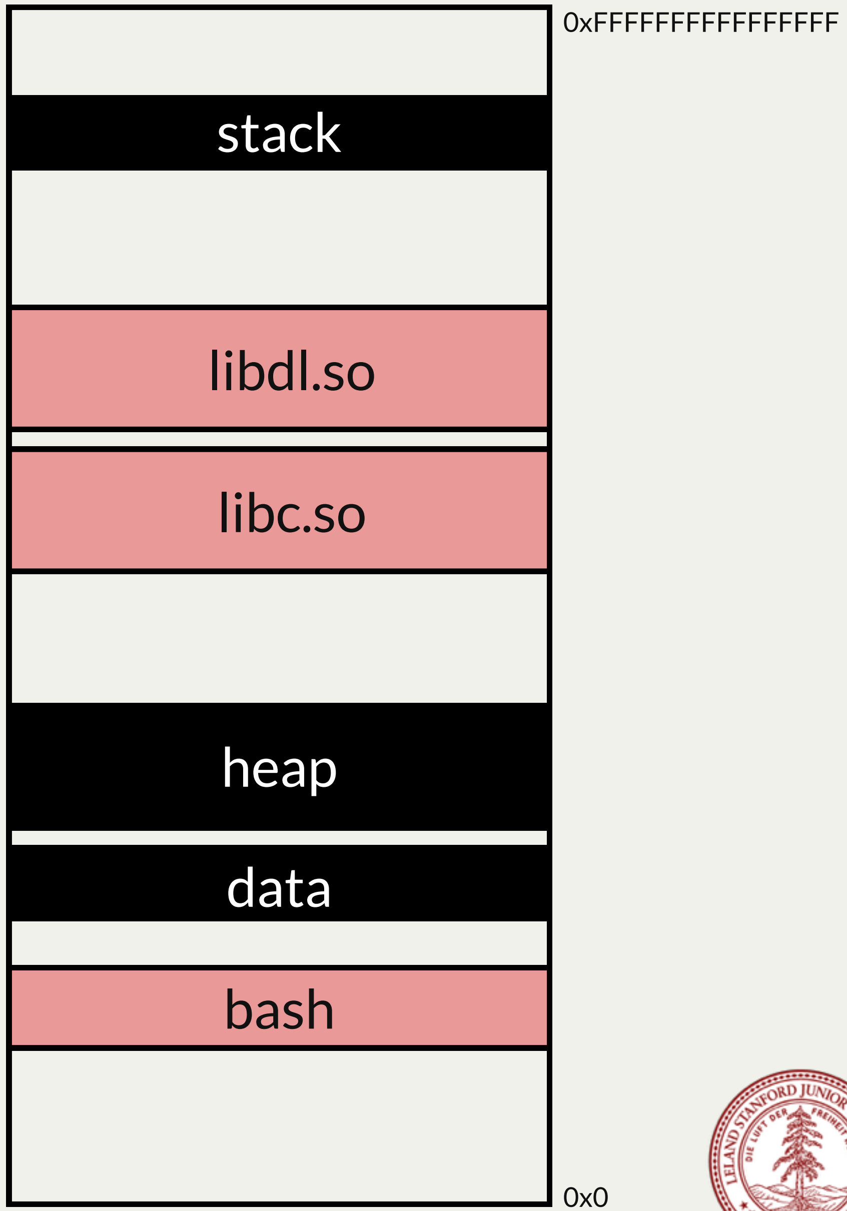
Memory Regions in a Process

- Most of a process's memory isn't used: valid regions are defined by *segments*, blocks of memory for a particular use
  - Quick quiz: what's a SEGV (segmentation violation)?
- Some segments you know quite well are the stack, heap, BSS, data, rodata, and code (where your executable is)
  - Quick quiz: differences between bss, data, and rodata?
- There are also segments for shared libraries
  - We just pmapped a process on myth
- Code is usually *not* read in through read: instead, it's memory mapped
- A memory mapped file acts like the whole file is read into a segment of memory, but it a single copy can be shared across many processes



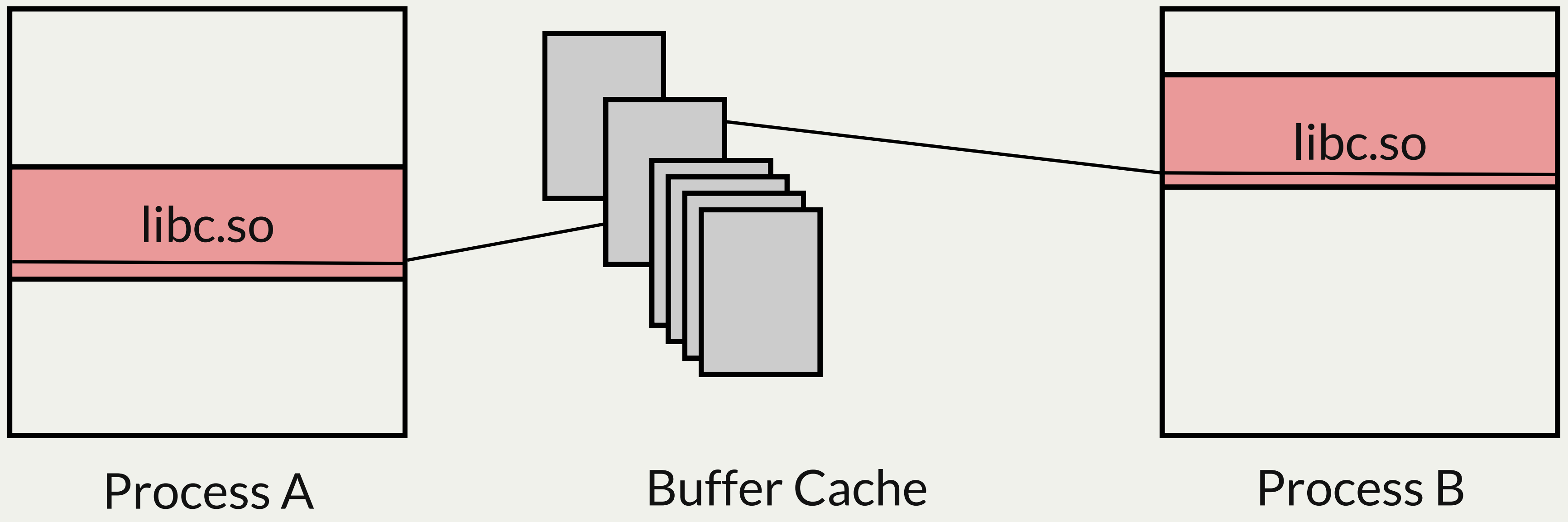
Memory Mapped Files

- void *mmap(void *addr,
  size_t len,
  int prot,
  int flags,
  int fd,
  off_t offset);
- "The mmap() system call causes the pages starting at addr and continuing for at most len bytes to be mapped from the object described by fd, starting at byte offset offset. If offset or len is not a multiple of the pagesize, the mapped region may extend past the specified range. Any extension beyond the end of the mapped object will be zero-filled."
  - A page (typically 4kB) is an operating system's unit of memory management, defined by hardware
- You can also mmap() anonymous memory, memory that has no backing file: pages in an anonymous region are zero (until written)
  - This is how the heap, stack, data, and bss are set up



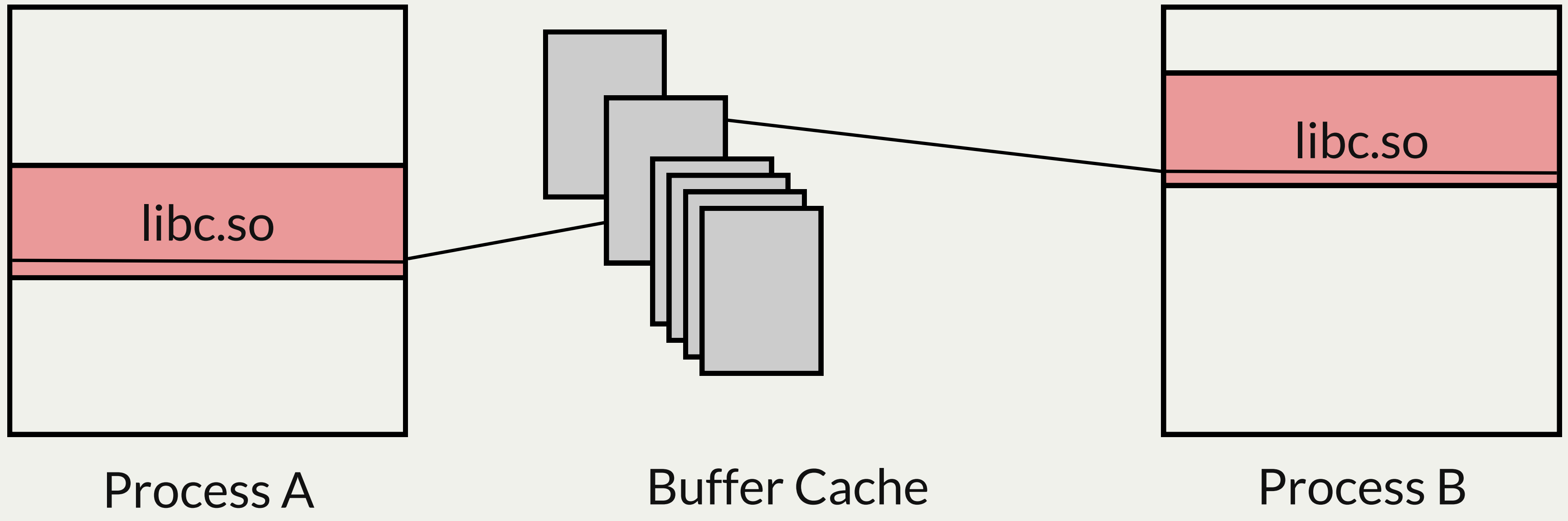
Memory Mapped Files and the Buffer Cache

- The operating system maintains a *buffer cache*: a pool of (page-sized) pieces of files that are in memory
  - Caching: keeping pieces of used data in faster storage to improve performance
  - Buffer cache: keeping parts of files in use in memory so you don't have to hit disk
- Calls to **`read()`** and **`write()`** operate on the buffer cache
- Blocks are read from disk and put into the buffer cache as needed
- Dirty pages in the buffer cache are written to disk when needed
  - **`sync()`** system call flushes buffers associated with file
- Two memory maps of the same file can point to the same buffer cache entry
- There's a bit more to this, but you'll have to wait for CS140 (we could spend 4 lectures on just the basics)



Memory Mapped Files Summary

- A program can map a file into its memory with **`mmap()`**
  - *Virtualization*: every process thinks it has its own copy, but in reality there's a single one in memory (exception: MAP_PRIVATE)
- Memory mapped files unify the idea of the process address space with its file descriptors
- Used parts of files are kept in memory in the buffer cache
  - *Caching*: don't force every process to read every file in entirety when it loads



Review

- The UNIX file system provides a *naming and name resolution* system for user data, through files and directories
- The UNIX file system is designed to use *abstraction* and *layering* so that it's easy to use new file systems and use existing file systems on new devices
    - Processes use the abstraction of a file descriptor, which refers to an open file in the file entry table
    - A file entry refers to a vnode, which describes the actual file itself
    - There can be many file descriptors for the same single file table entry, and many file table entries for the same vnode
    - This layering has important semantics which give you a lot of power in how you manipulate files

- Programs can directly map files into their memory with mmap(), which allows the OS to use *caching*
    - In-use parts of files are kept in memory by a system called the buffer cache
    - The buffer cache allows many programs to share a single copy of data (e.g., library code)

Topic #2: Multiprocessing

Key Question: How can my program create and interact with other programs?

Multiprocessing Terminology

Program: code you write to execute tasks

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

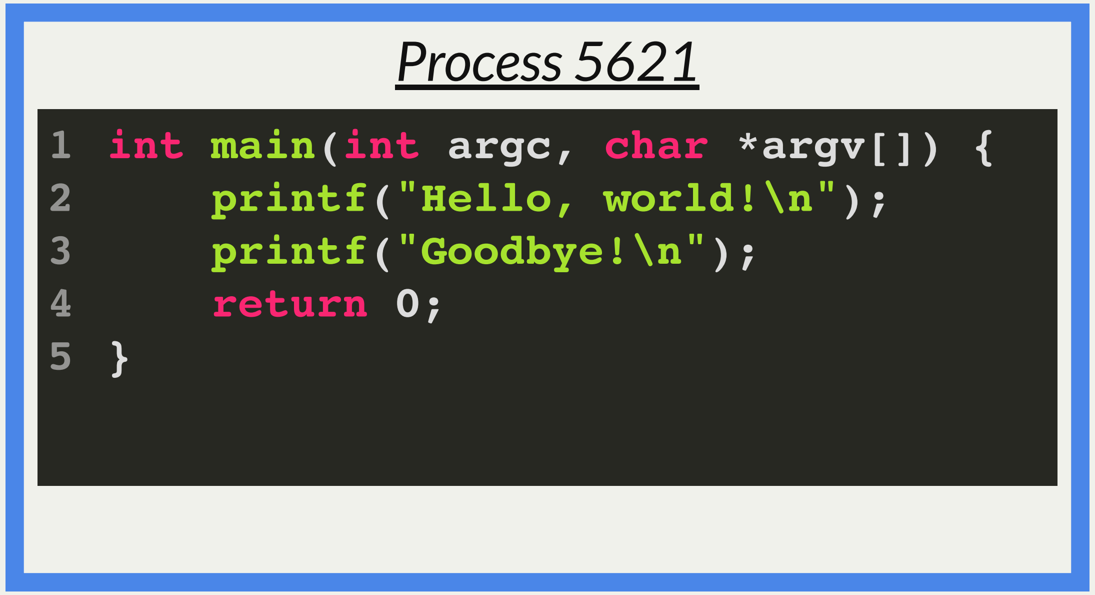
Key idea: multiple processes can run the same program

Multiprocessing 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



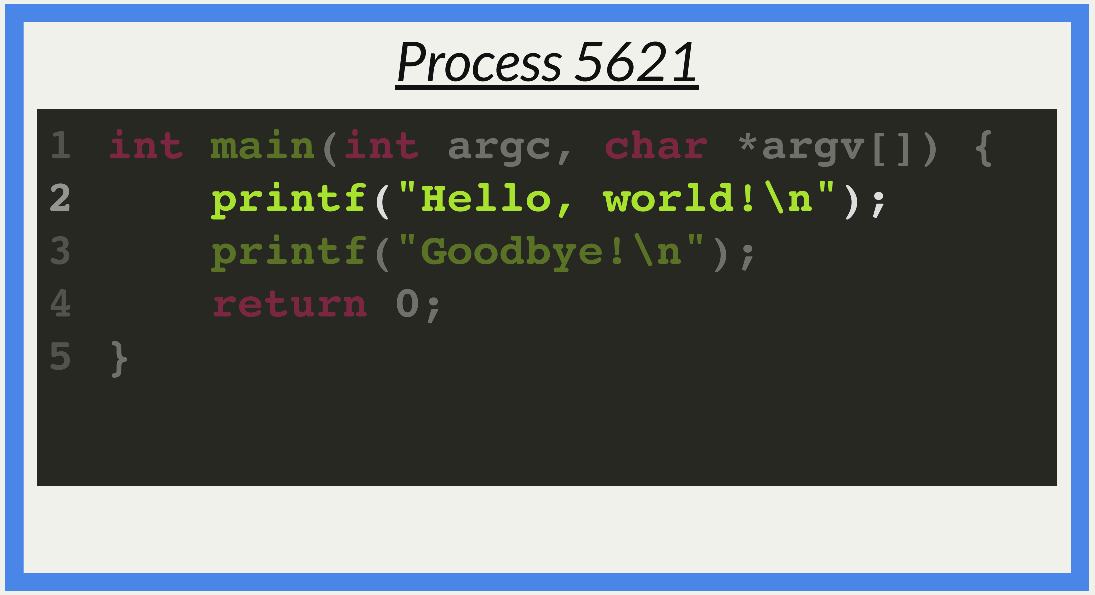
```c
int main(int argc, char *argv[]) {
    printf("Hello, world!\n");
    printf("Goodbye!\n");
    return 0;
}
```

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



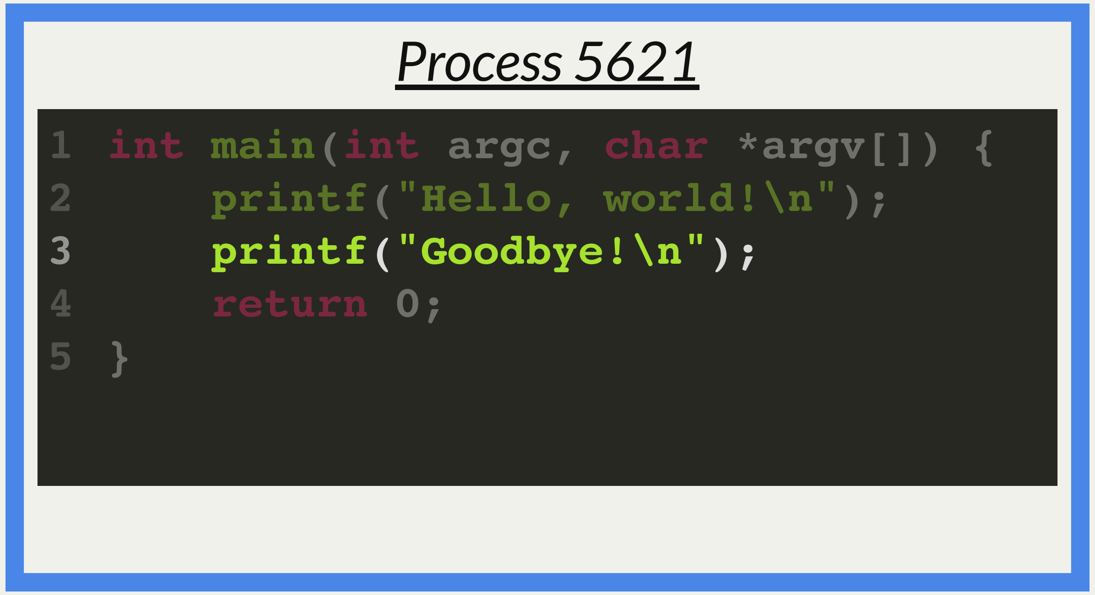
```c
int main(int argc, char *argv[]) {
    printf("Hello, world!\n");
    printf("Goodbye!\n");
    return 0;
}
```

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



```
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?)

Multiprocessing

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

- "simultaneously" = switch between them so fast humans don't notice

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

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

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

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

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
```

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;
}
```

```
$ ./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

```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!
```

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[]) {
    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;
}
```

```
$ ./myprogram2
```

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

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

fork()

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

```c
pid_t fork();
```

Process A

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

```
$ ./myprogram2
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;
}
```

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

fork()

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

```c
pid_t fork();
```

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;
}
```

```
$ ./myprogram2
Hello, world!
Goodbye, 2!
Goodbye, 2!
```

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;
}
```

```
$ ./myprogram2
Hello, world!
Goodbye, 2!
Goodbye, 2!
```

fork()

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

```
pid_t fork();
```

`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

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.

`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.
- **fork()** is called once, but returns twice (why?)

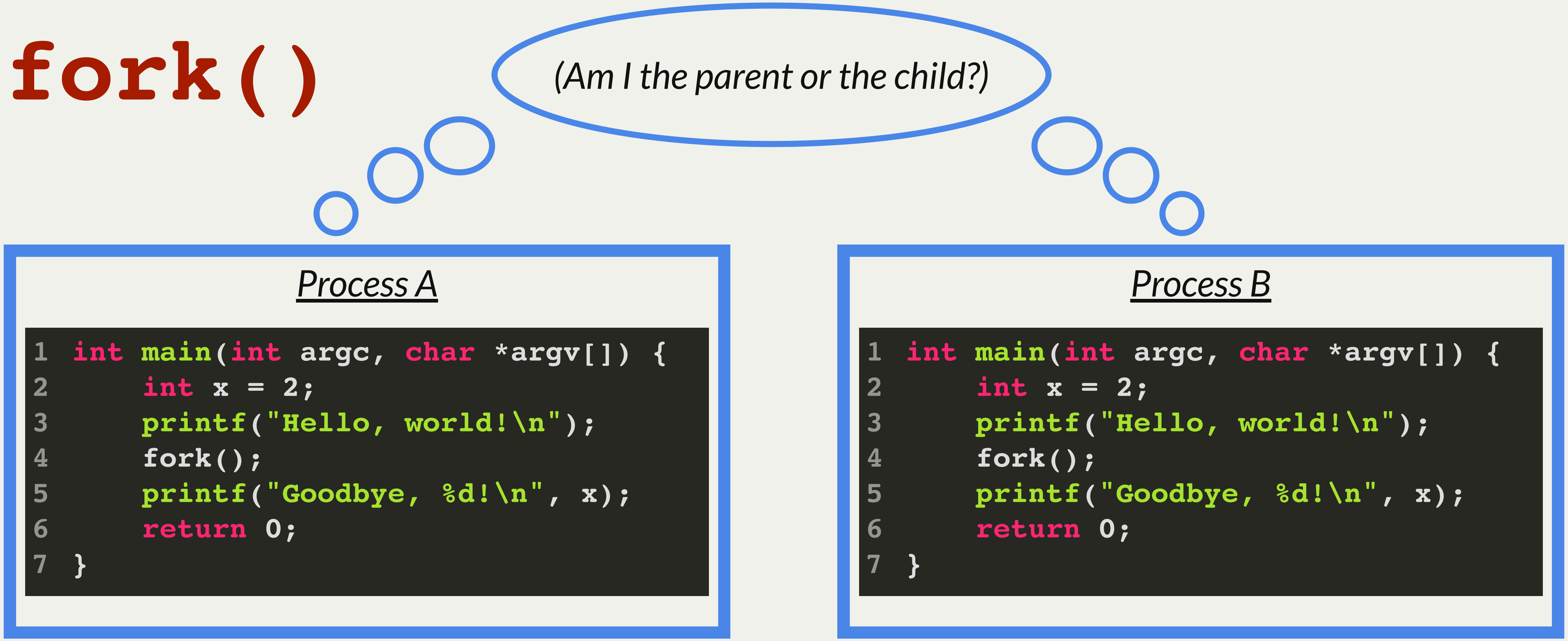
fork()

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.
- **fork()** is called once, but returns twice (why?)
- Everything is duplicated in the child process
  - File descriptors (increasing reference counts on file table entries)
  - Mapped memory regions (the address space)
  - Regions like stack, heap, etc. are copied

fork()



Process A

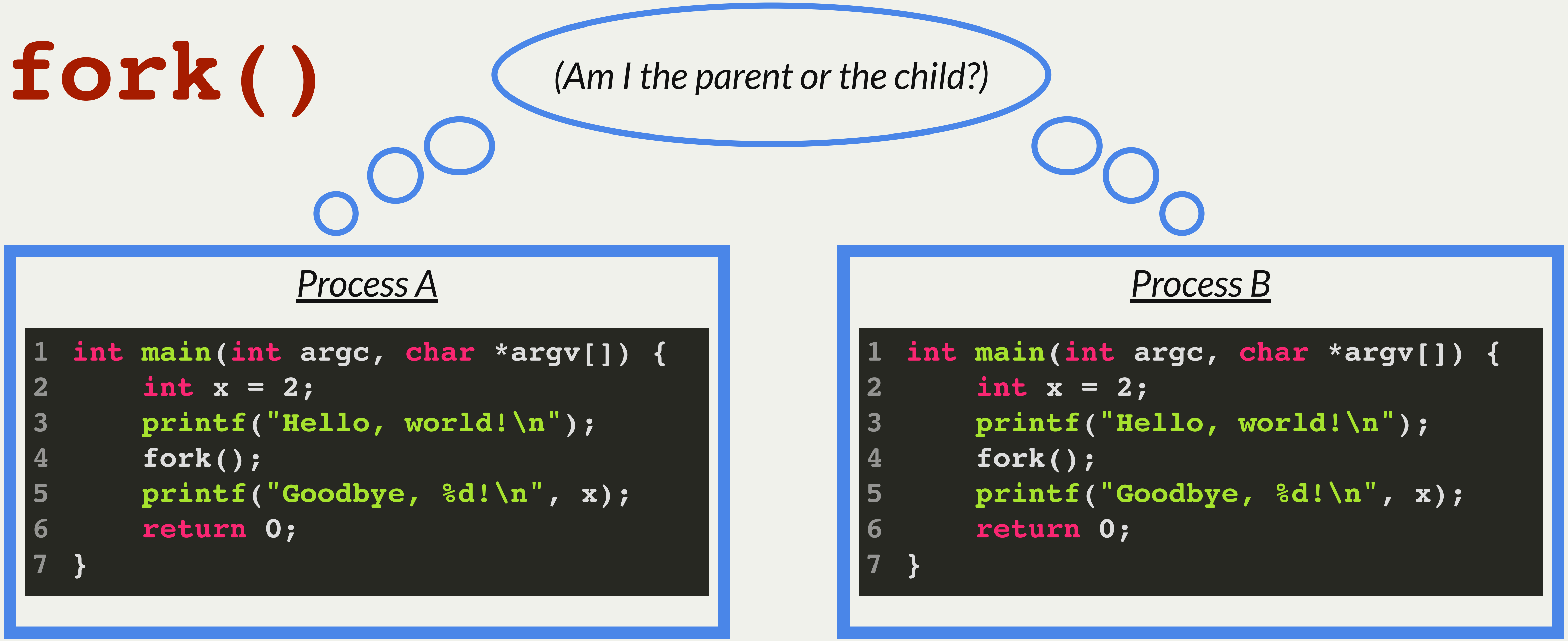
```c
int main(int argc, char *argv[]) {
    int x = 2;
    printf("Hello, world!\n");
    fork();
    printf("Goodbye, %d!\n", x);
    return 0;
}
```

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()



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

```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?

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

```c
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 (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

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

```
$ ./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

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

```
$ ./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

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

Process 111

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

```
$ ./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

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

Process 111

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

```
$ ./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

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

Process 111

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

```
$ ./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

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

Process 111

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

```
$ ./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

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

Process 111

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

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

```
$ ./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

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

Process 111

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

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

```
$ ./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)
- 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
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)
```

`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
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.

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
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!

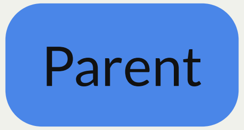
Debugging Multiprocess Programs

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.

Fork Tree

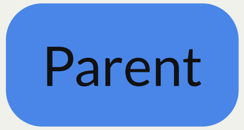
Here's a useful (but mind-melting) example of a program where child processes themselves call **fork()**:



```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

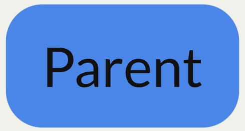
Here's a useful (but mind-melting) example of a program where child processes themselves call **fork()**:



```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);
    }
    return 0;
}
```

Fork Tree

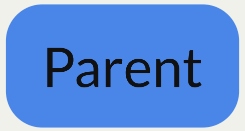
Here's a useful (but mind-melting) example of a program where child processes themselves call **fork()**:



```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()**:

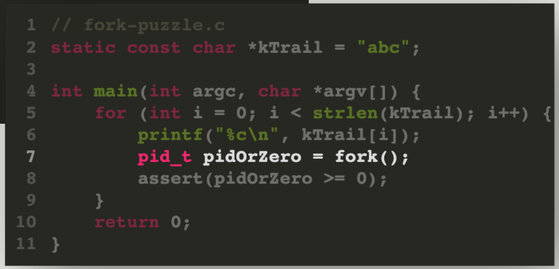


```c
// fork-puzzle.c
static const char *kTrail = "abc";

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);
    }
    return 0;
}
```

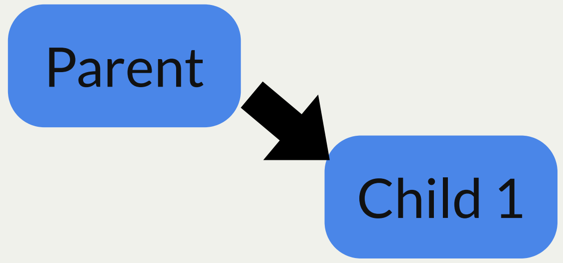
Fork Tree

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;
}
```



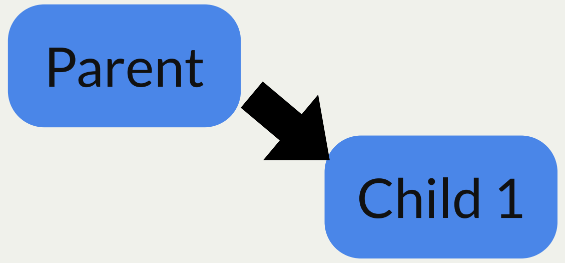
```c
// fork-puzzle.c
static const char *kTrail = "abc";

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;
}
```

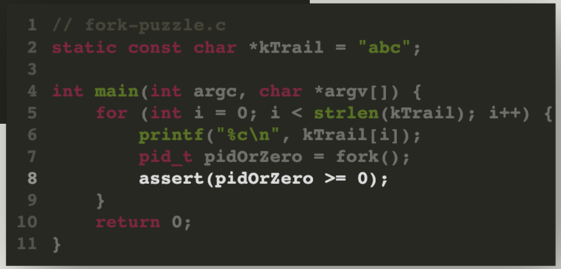


Fork Tree

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;
}
```



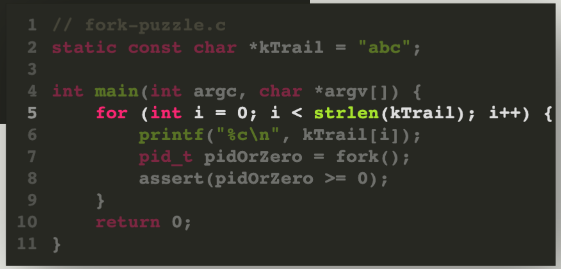


```c
// fork-puzzle.c
static const char *kTrail = "abc";

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;
}
```

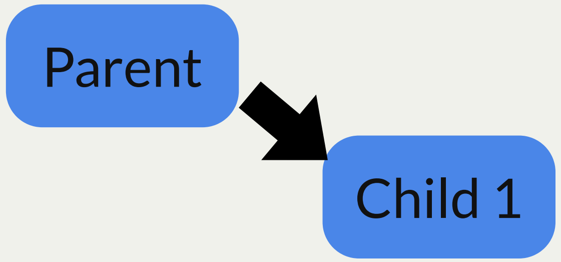
Fork Tree

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;
}
```



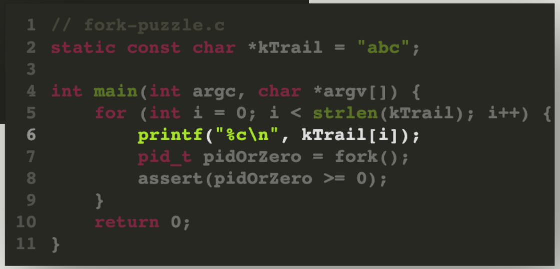
```c
// fork-puzzle.c
static const char *kTrail = "abc";

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;
}
```



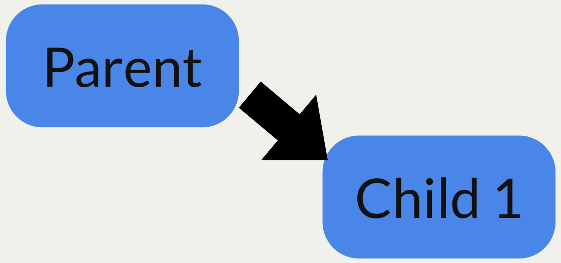
Fork Tree

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;
}
```



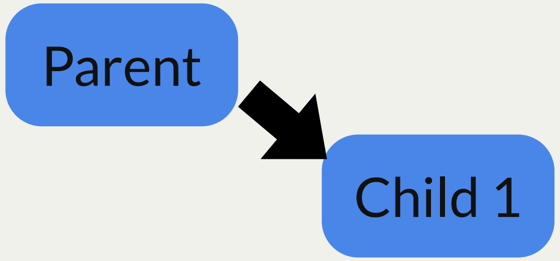
```c
// fork-puzzle.c
static const char *kTrail = "abc";

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()**:

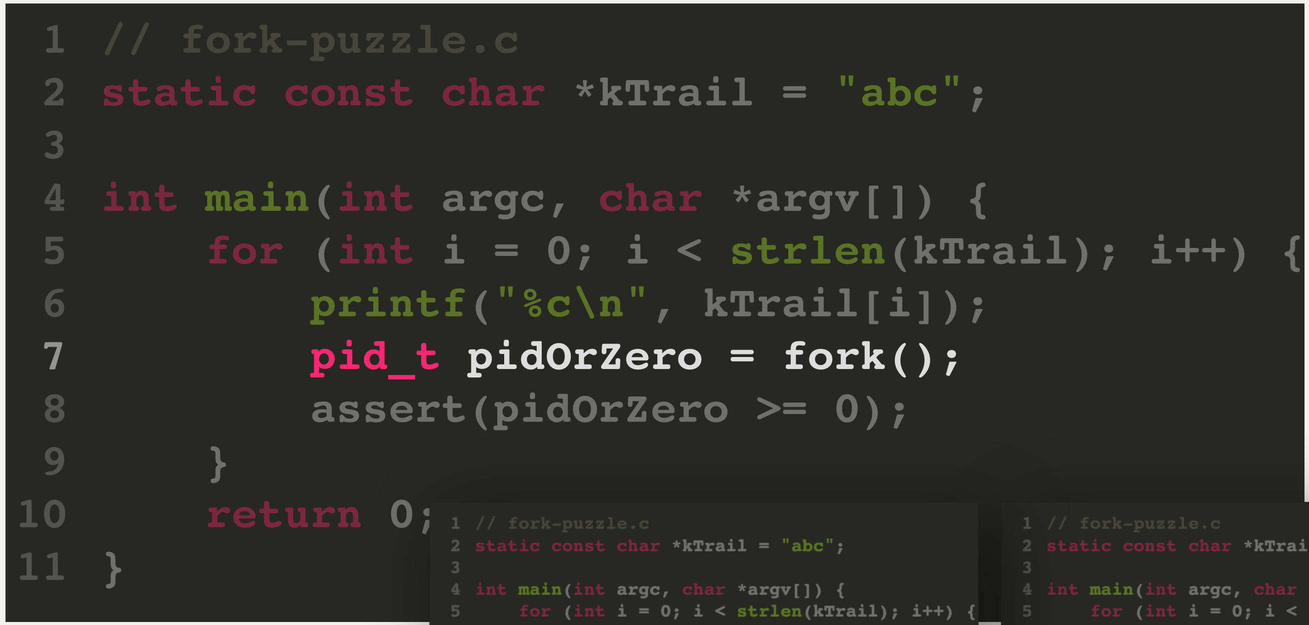
```c
// fork-puzzle.c
static const char *kTrail = "abc";

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;
}
```



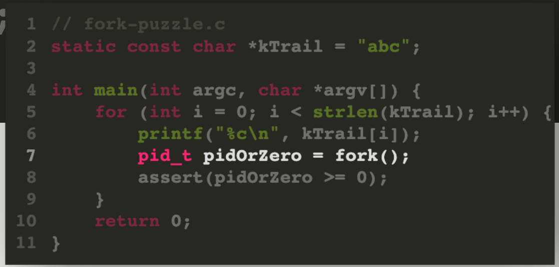
```c
// fork-puzzle.c
static const char *kTrail = "abc";

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

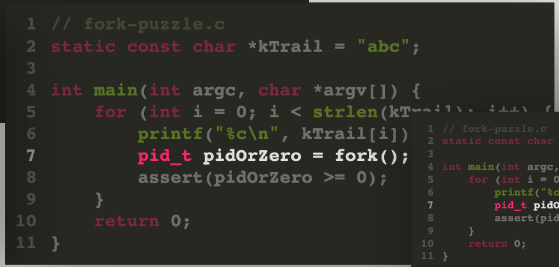
Here's a useful (but mind-melting) example of a program where child processes themselves call **fork()**:



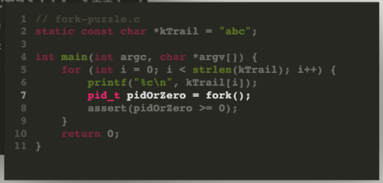
```c
// fork-puzzle.c
static const char *kTrail = "abc";

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;
}
```



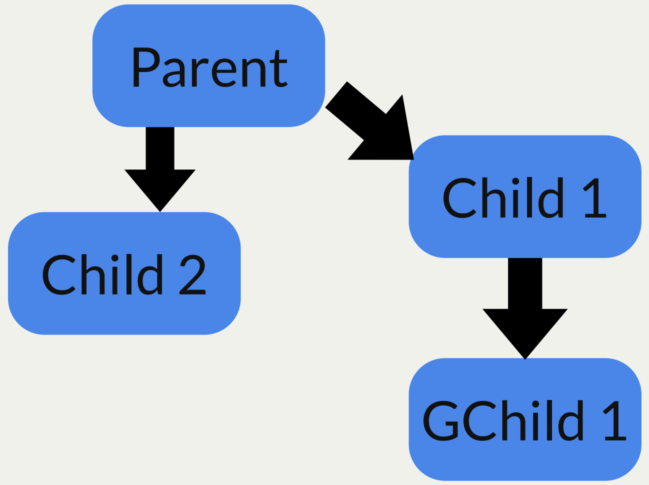
```c
// fork-puzzle.c
static const char *kTrail = "abc";

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(kT
        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;
}
```

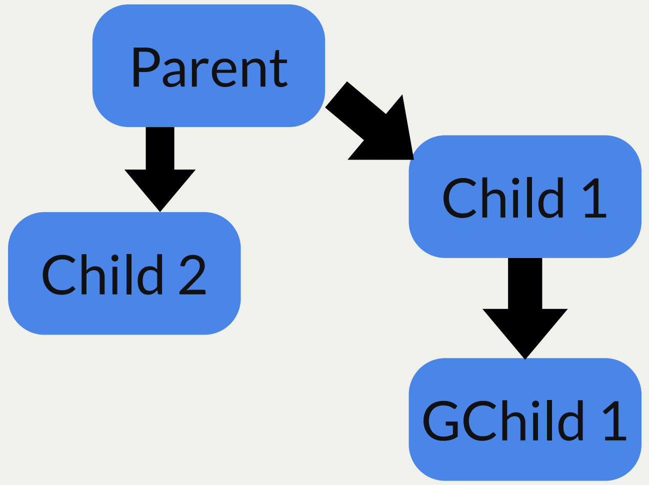


Fork Tree

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

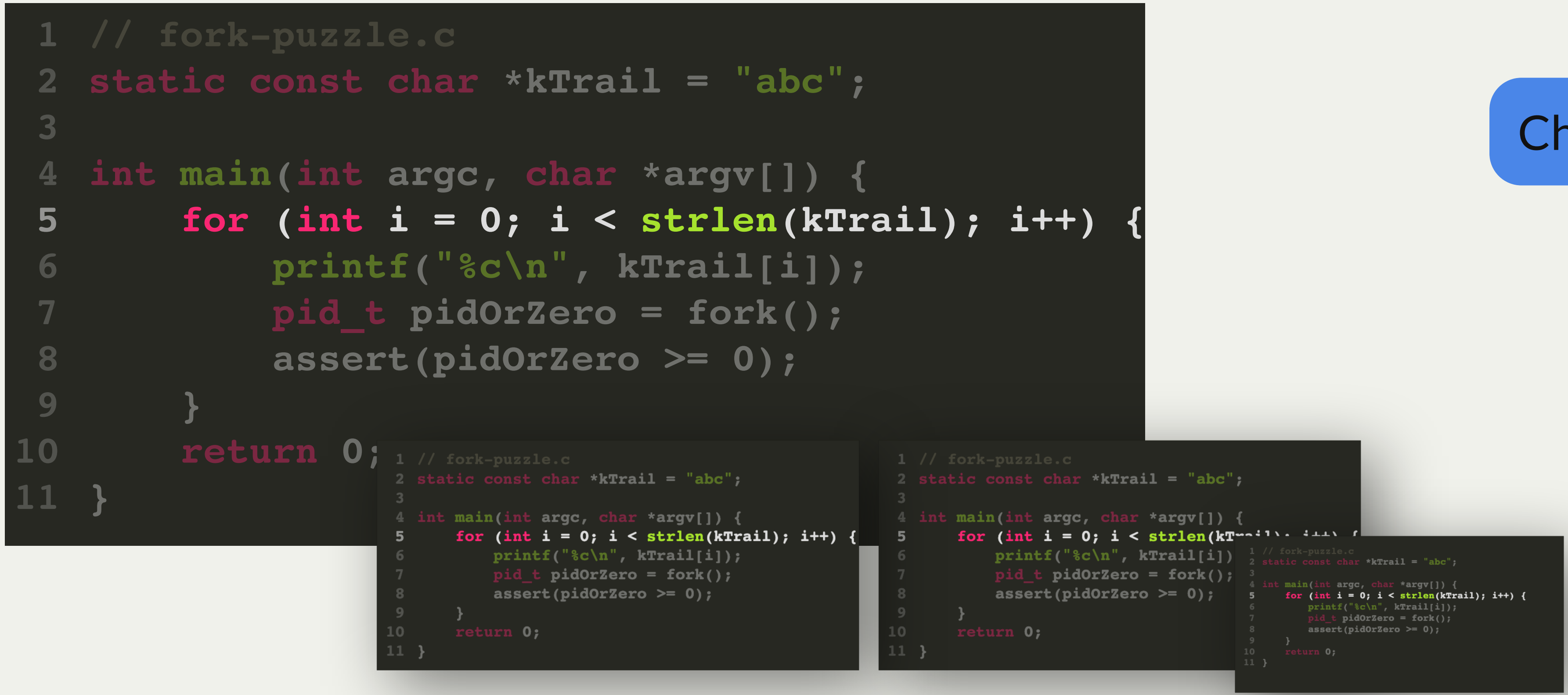
```c
// fork-puzzle.c
static const char *kTrail = "abc";

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;
}
```

```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;
}
```

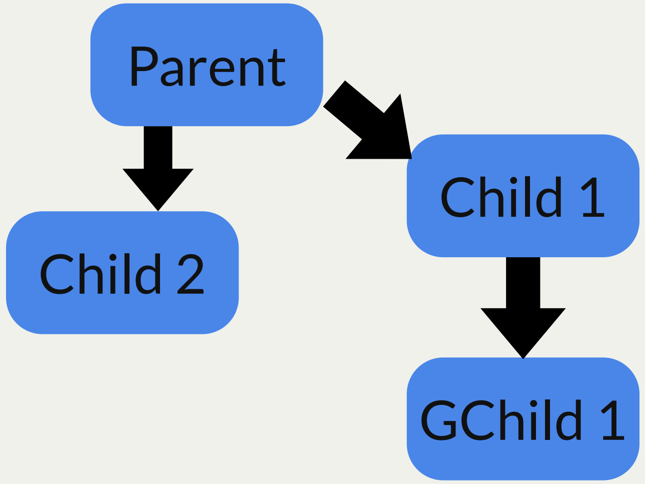


Fork Tree

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

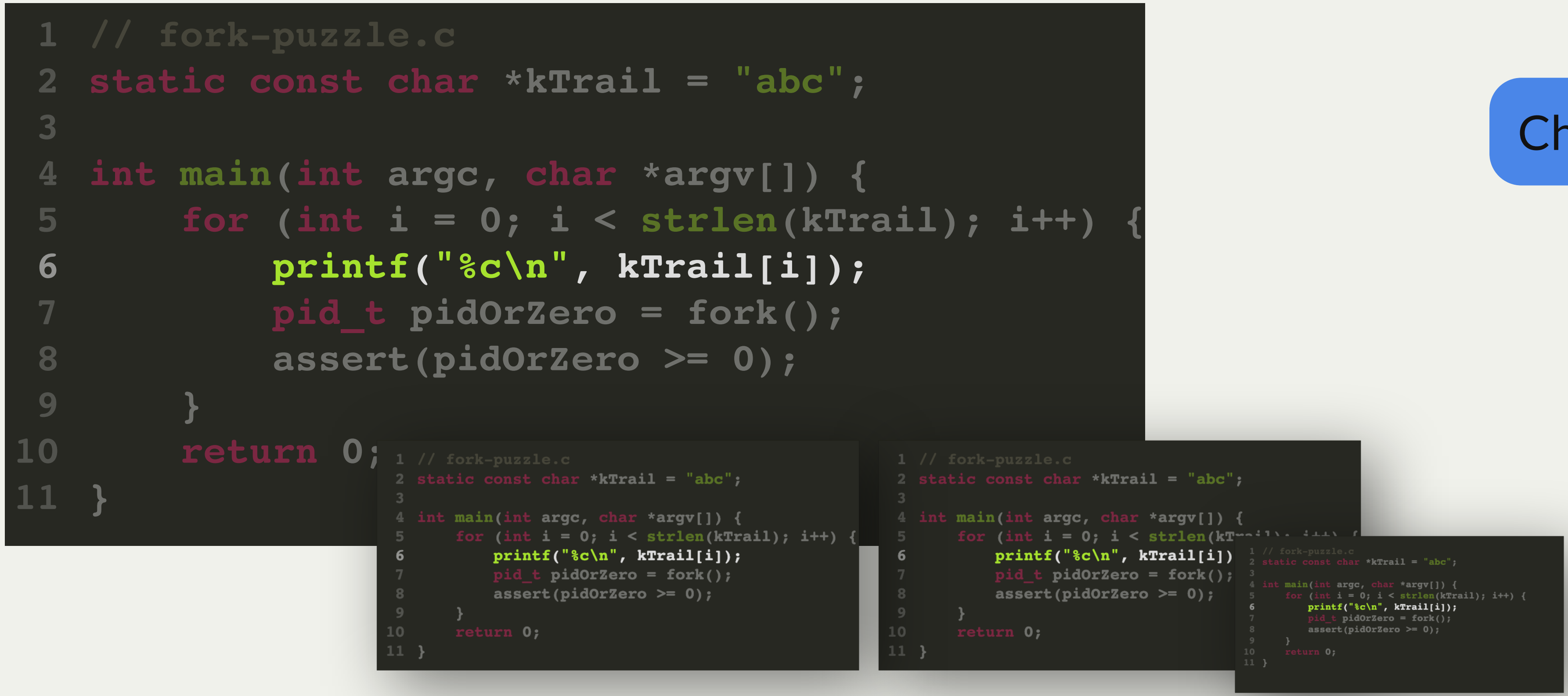


```c
// fork-puzzle.c
static const char *kTrail = "abc";

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;
}
```

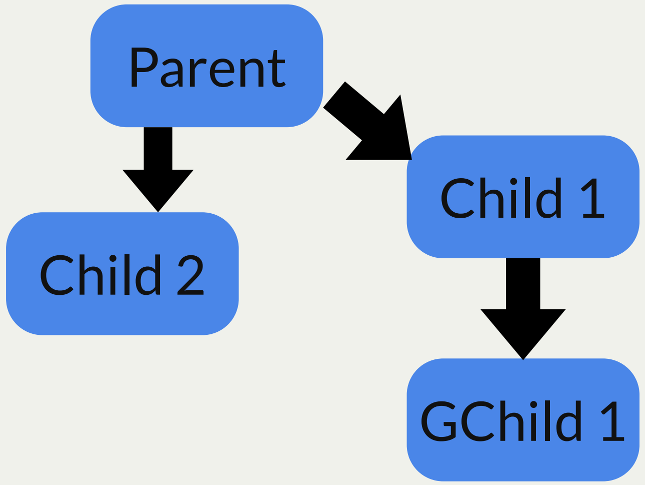


Fork Tree

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

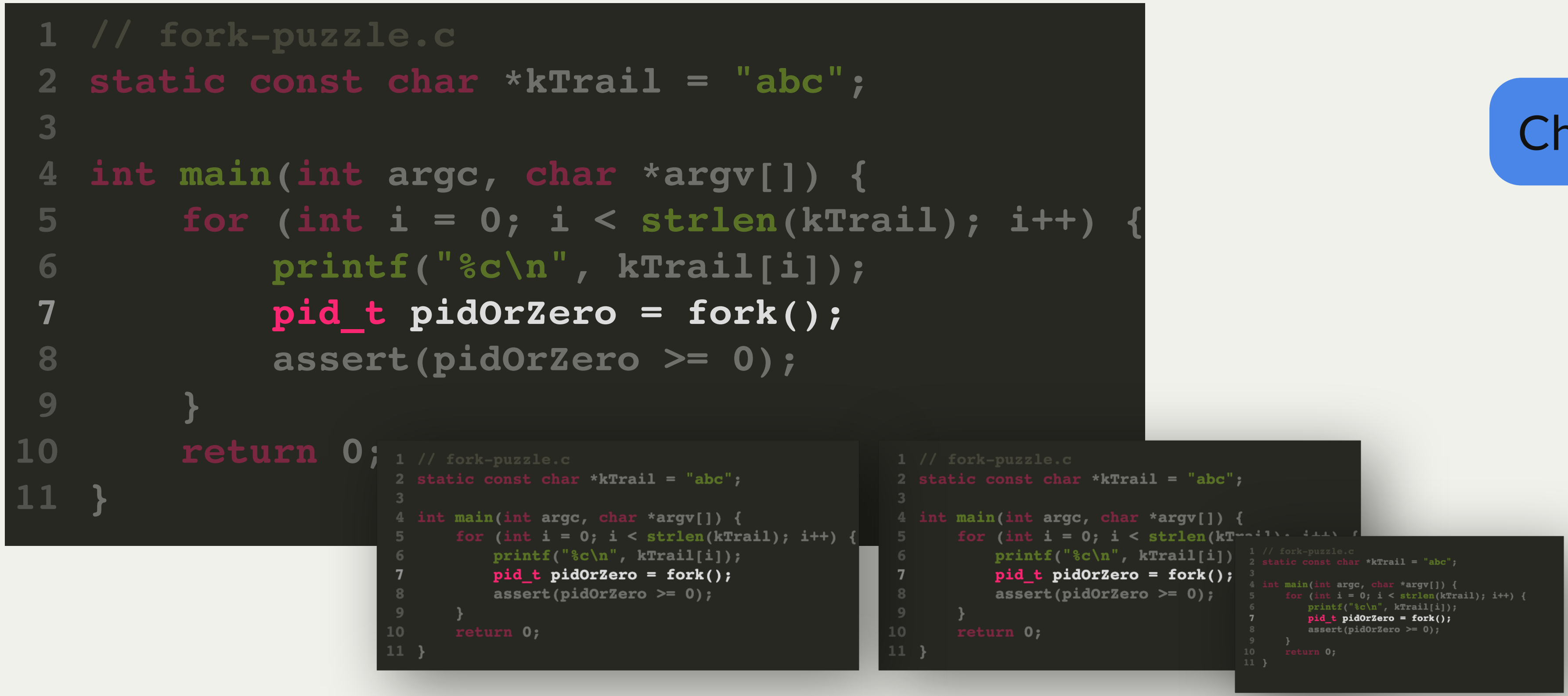


```c
// fork-puzzle.c
static const char *kTrail = "abc";

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);
    }
    return 0;
}
```

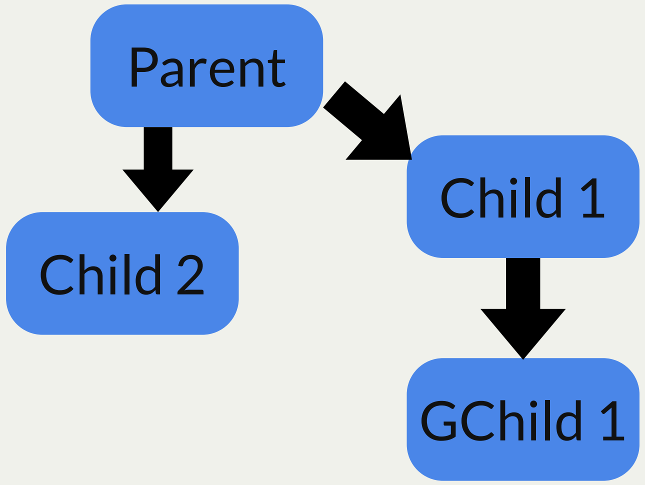


Fork Tree

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

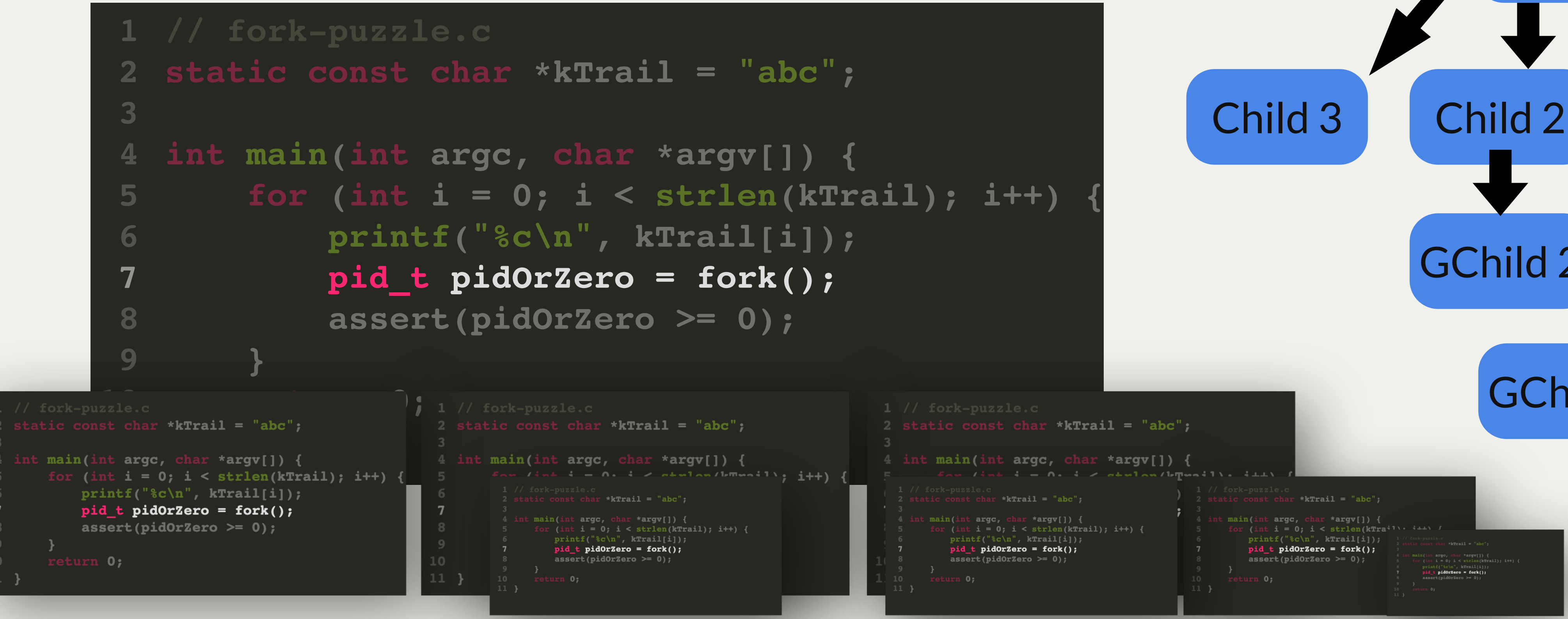


```c
// fork-puzzle.c
static const char *kTrail = "abc";

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;
}
```

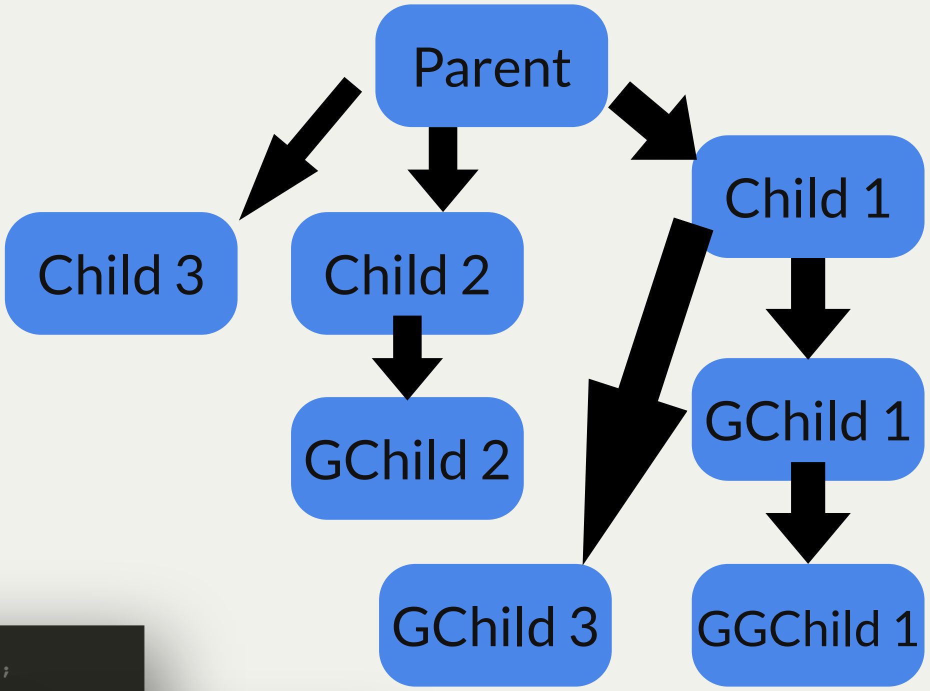


Fork Tree

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

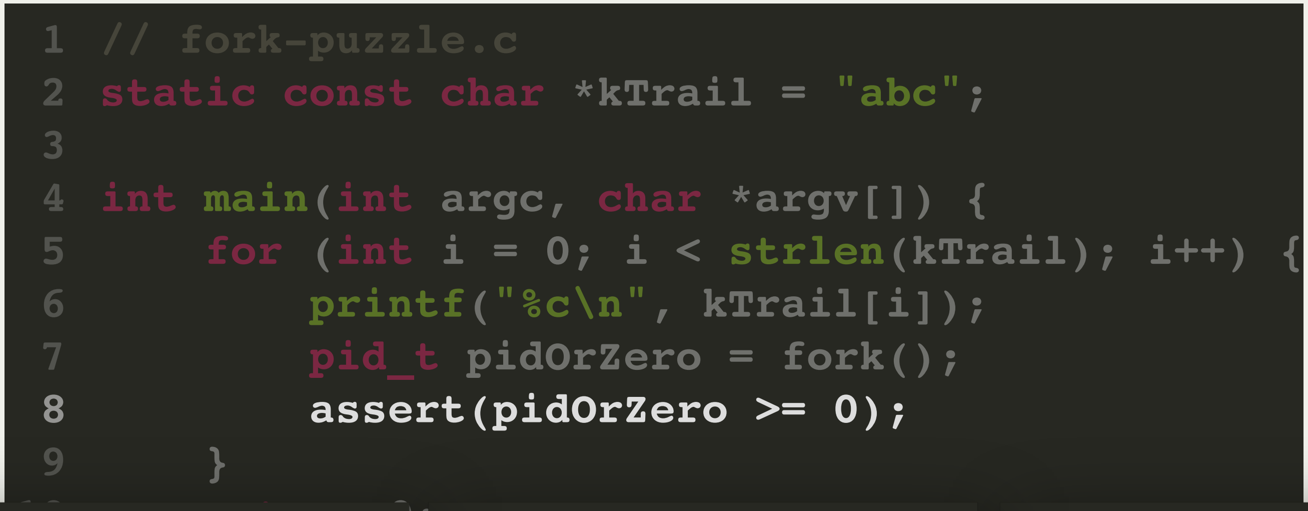


```c
// fork-puzzle.c
static const char *kTrail = "abc";

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);
    }
    return 0;
}
```

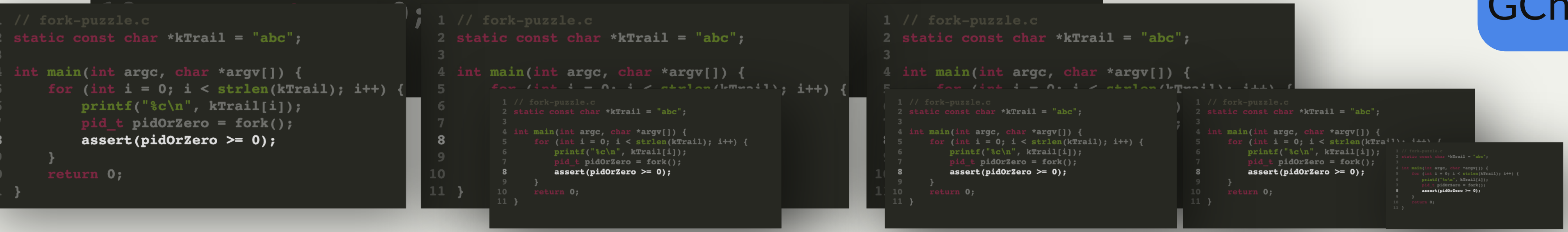


Fork Tree

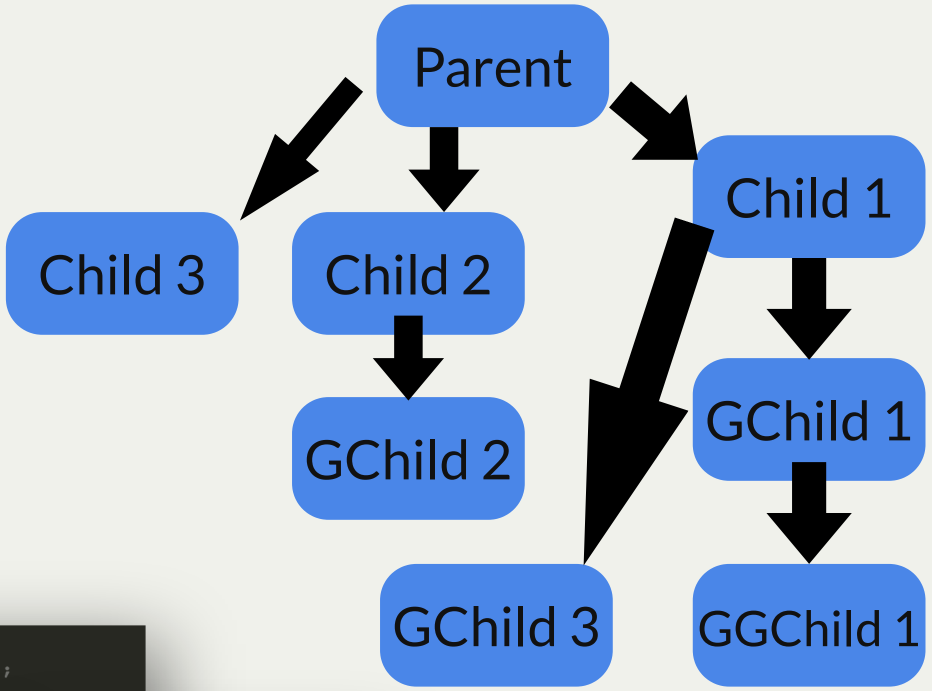
Here's a useful (but mind-melting) example of a program where child processes themselves call **fork()**:



```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);
    }
```

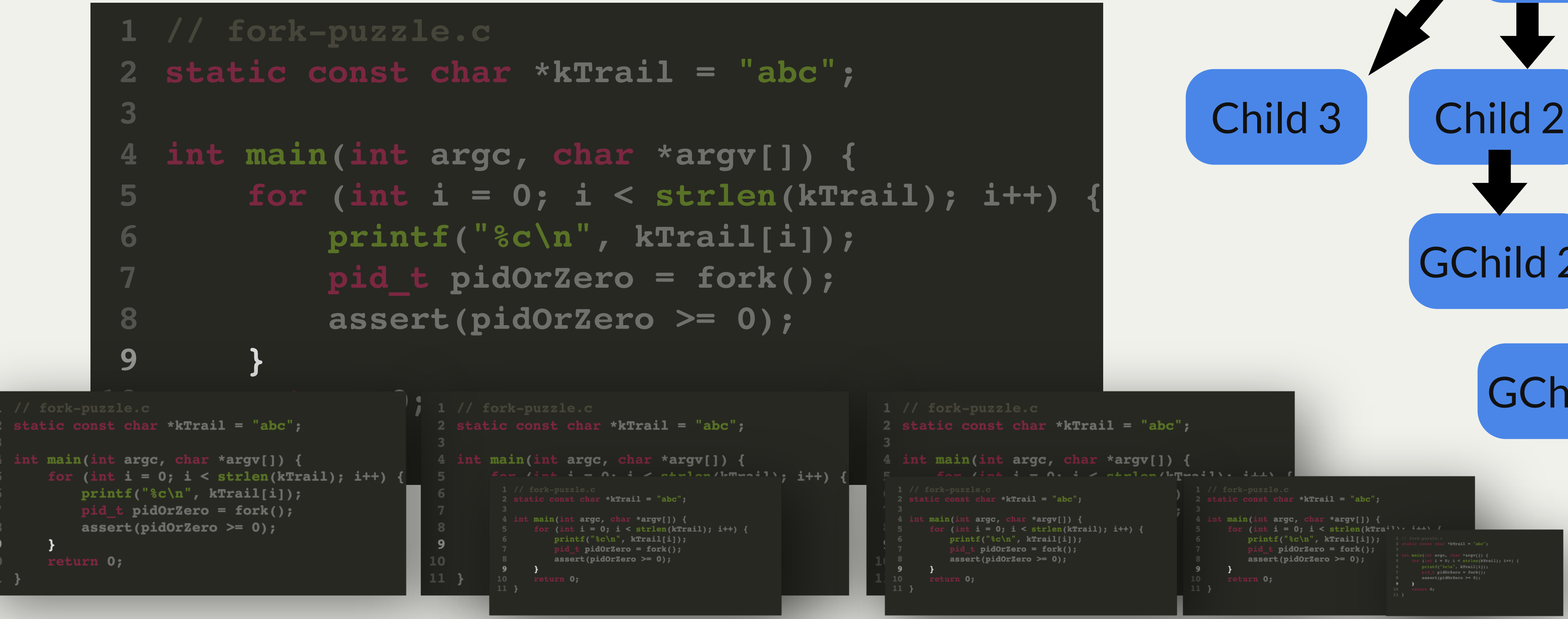


```c
// fork-puzzle.c
static const char *kTrail = "abc";

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;
}
```

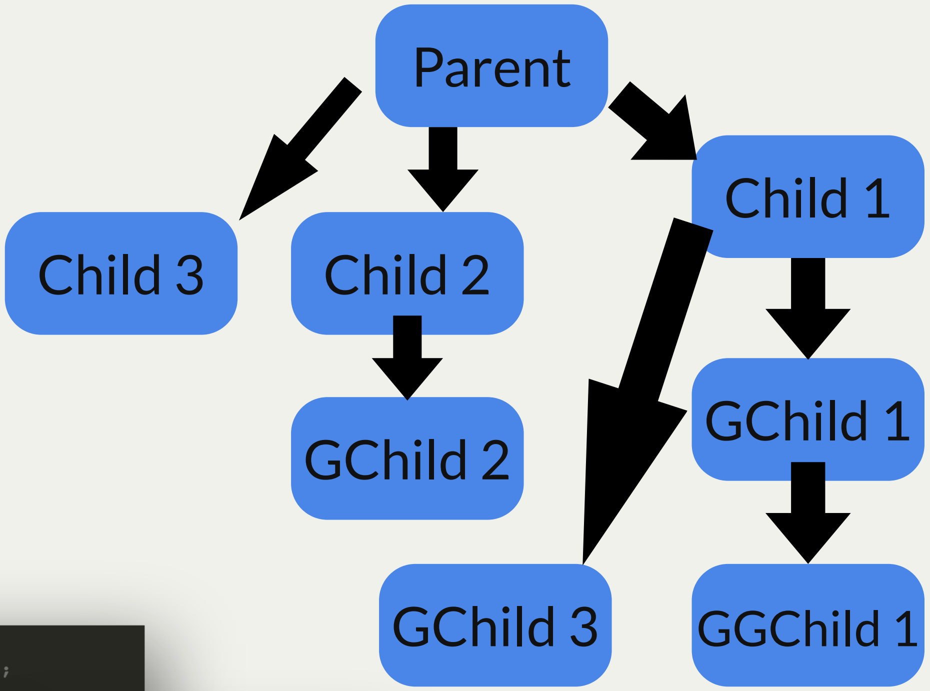


Fork Tree

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

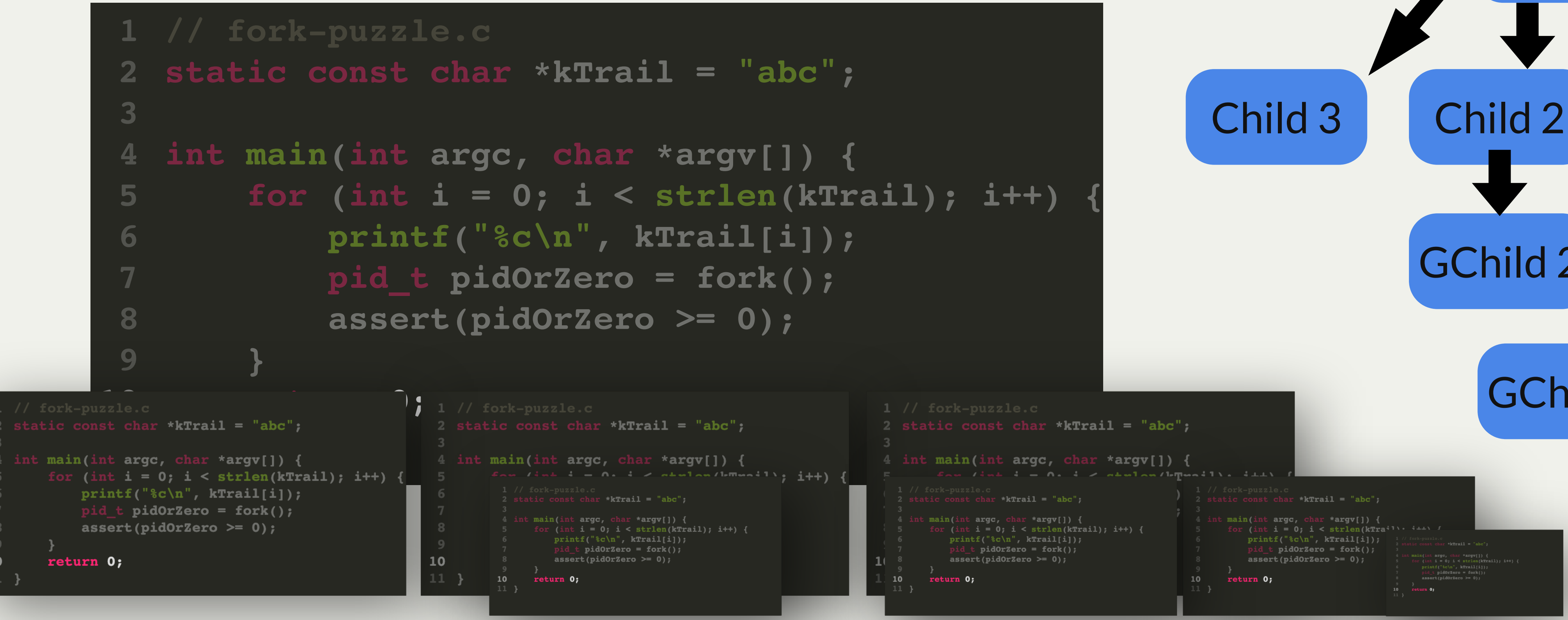


```c
// fork-puzzle.c
static const char *kTrail = "abc";

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;
}
```

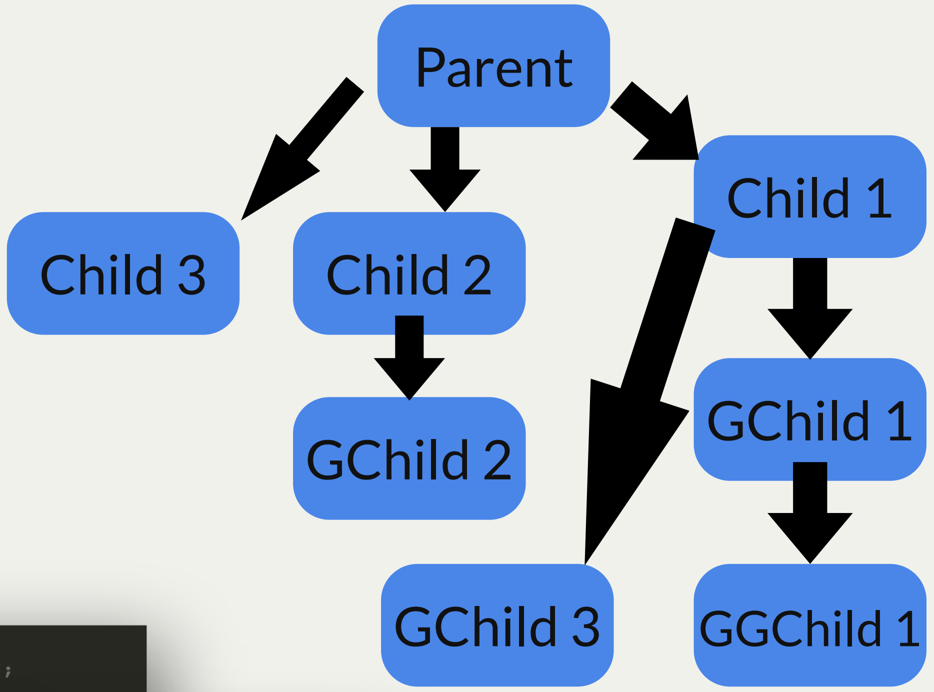


Fork Tree

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



```
// 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

```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;
}
```

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

```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-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
```

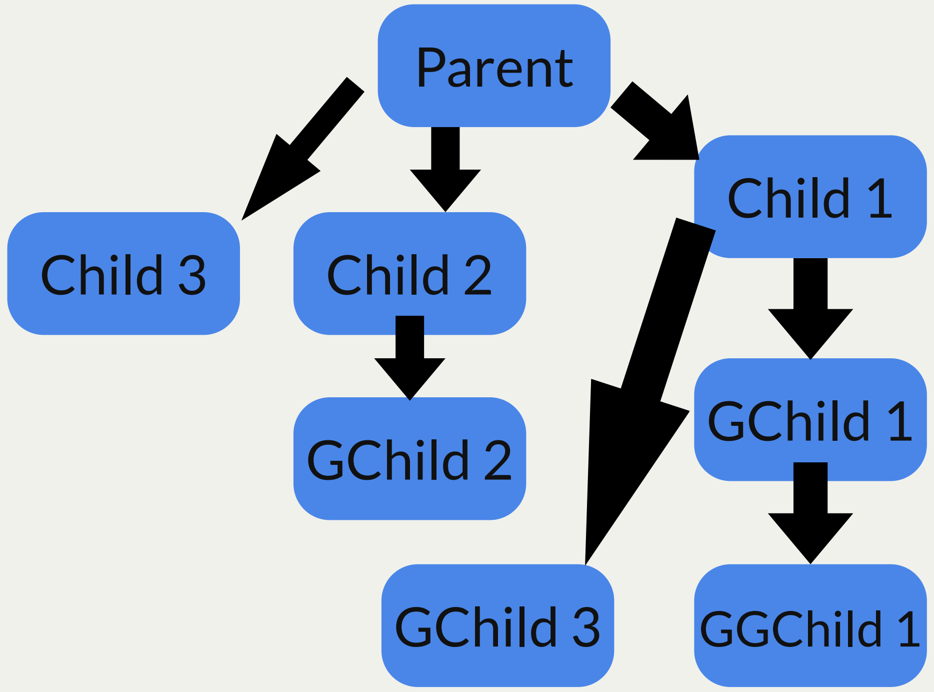
What happened here?

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

```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;
}
```



Questions:

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

Why Fork?

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?)

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:

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

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

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)

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

- A process is an instance of a program
- 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

Review

- A process is an instance of a program
- 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

Next time: the power of the **fork()**

Extra Problems

Fork Tree Round 2

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

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;
}
```

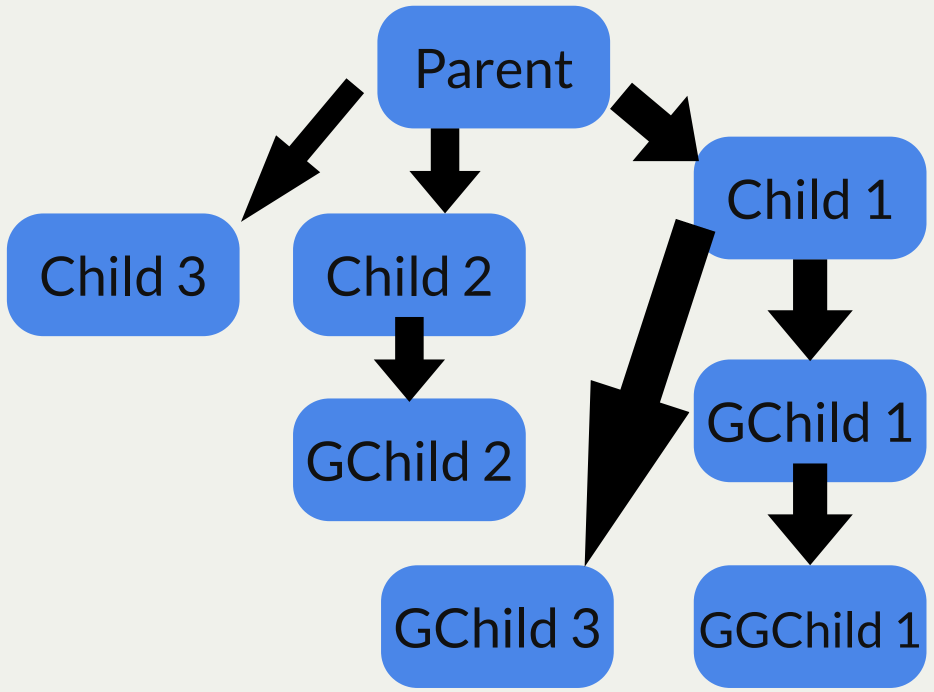
Questions:

- 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

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

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;
}
```

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