Lecture 04: Files, Memory, and Processes

Principles of Computer Systems

Autumn 2019

Stanford University

Computer Science Department

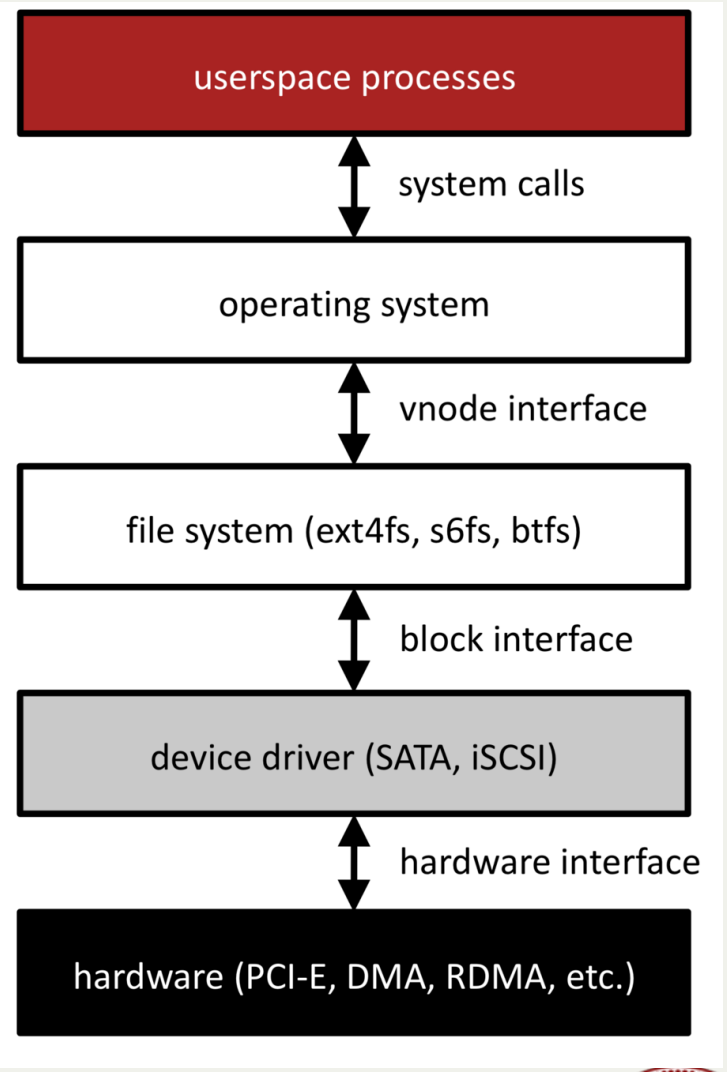
Lecturer: Chris Gregg and

Philip Levis

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

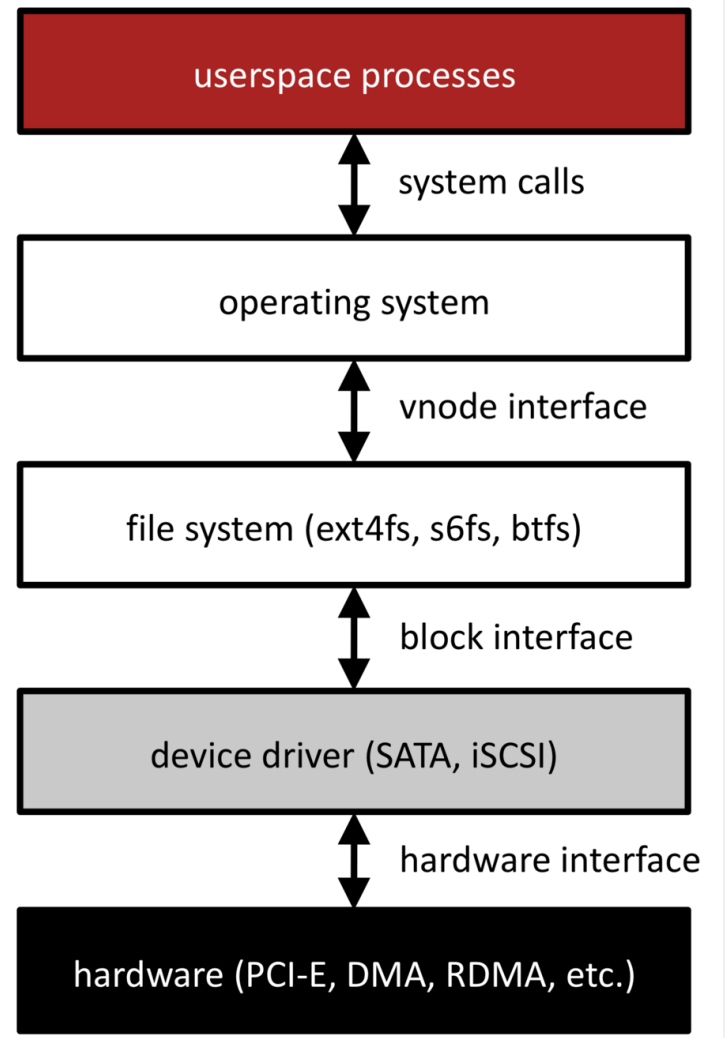


Today's Lecture in 3 Parts

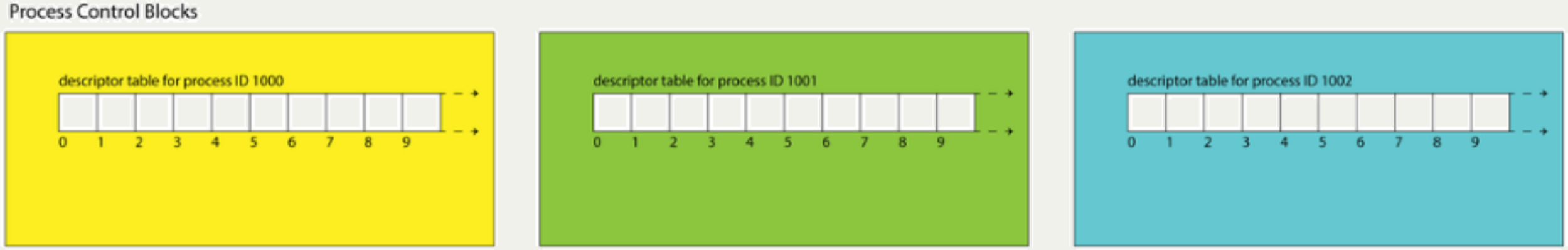
- How files on disk are presented to a program as file descriptors
- How programs open, control and manipulate files
  - How does the command below work?
  - `$cat people.txt | uniq | sort > list.txt`
- File descriptors vs. open files (many-to-one mapping)
- `vnode` abstraction of a file within the kernel

- 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

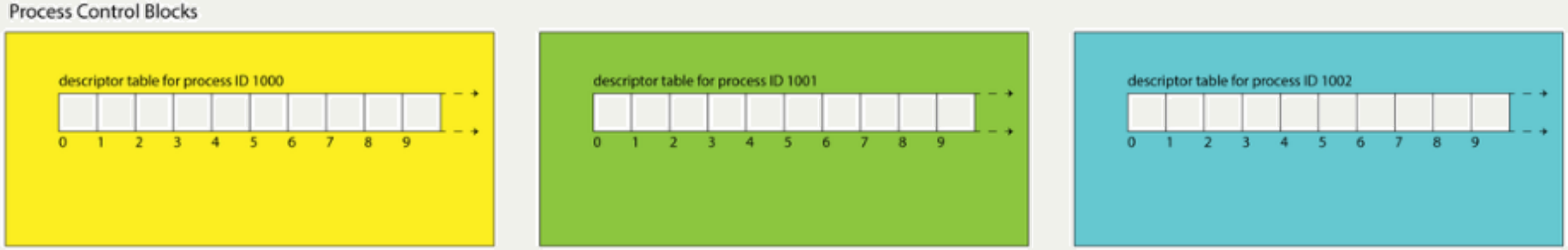


- Linux maintains a data structure for each active process. These data structures are called **`process control blocks`**, and they are stored in the **`process table`**
  - We'll explain exactly what a process is later in lecture

- 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`**

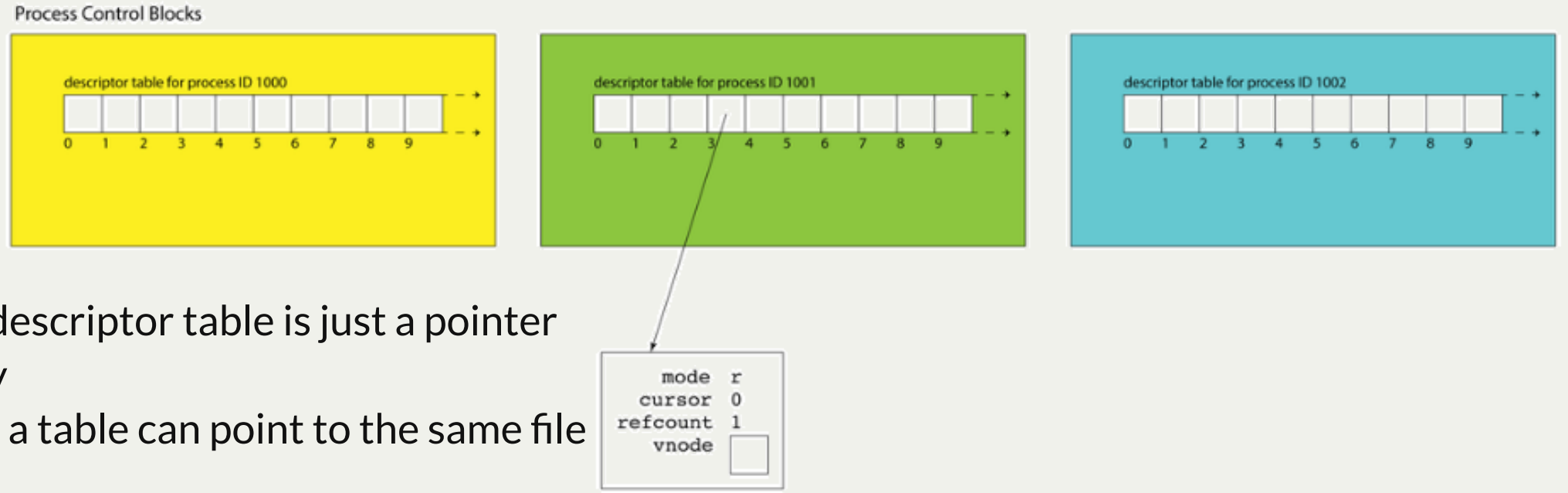
- A file descriptor (used by your program) is a small integer that's an index into this table
  - Descriptors 0, 1, and 2 are standard input, standard output, and standard error, but there are no predefined meanings for descriptors 3 and up. When you run a program from the terminal, descriptors 0, 1,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

Creating and Using File Descriptors



- 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



- 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 fle 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 tnry: 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
```

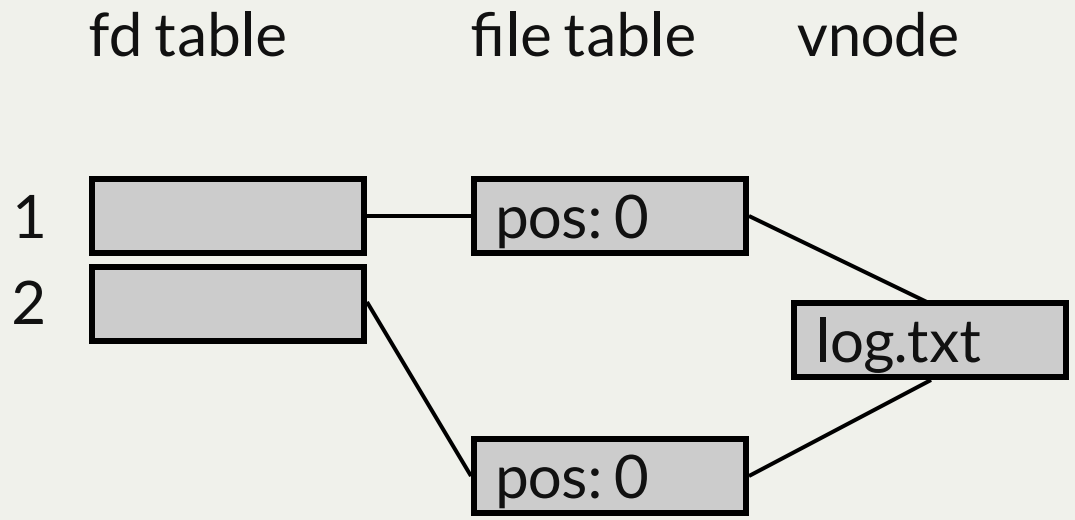
Opens log.txt twice (two file table entries)

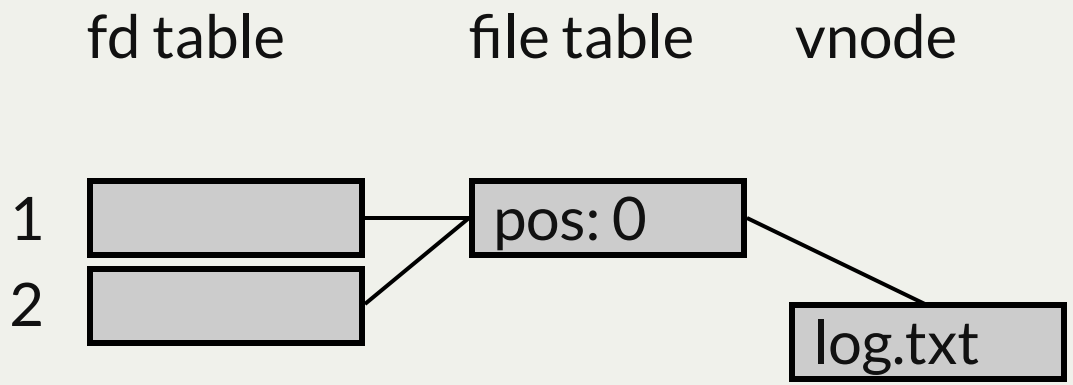
```
$ ./main 1> log.txt 2>&1
```

Opens log.txt once, two descriptors for same file table entry

```
const char* error = "This is my error message.\n It's terrible.\n";
const char* msg   = "This is my coolest msg. You are\n";

int main() {
   write(2, error, strlen(error));
   write(1, msg, strlen(msg));
}
```

File Descriptors vs. File Table Entries Example

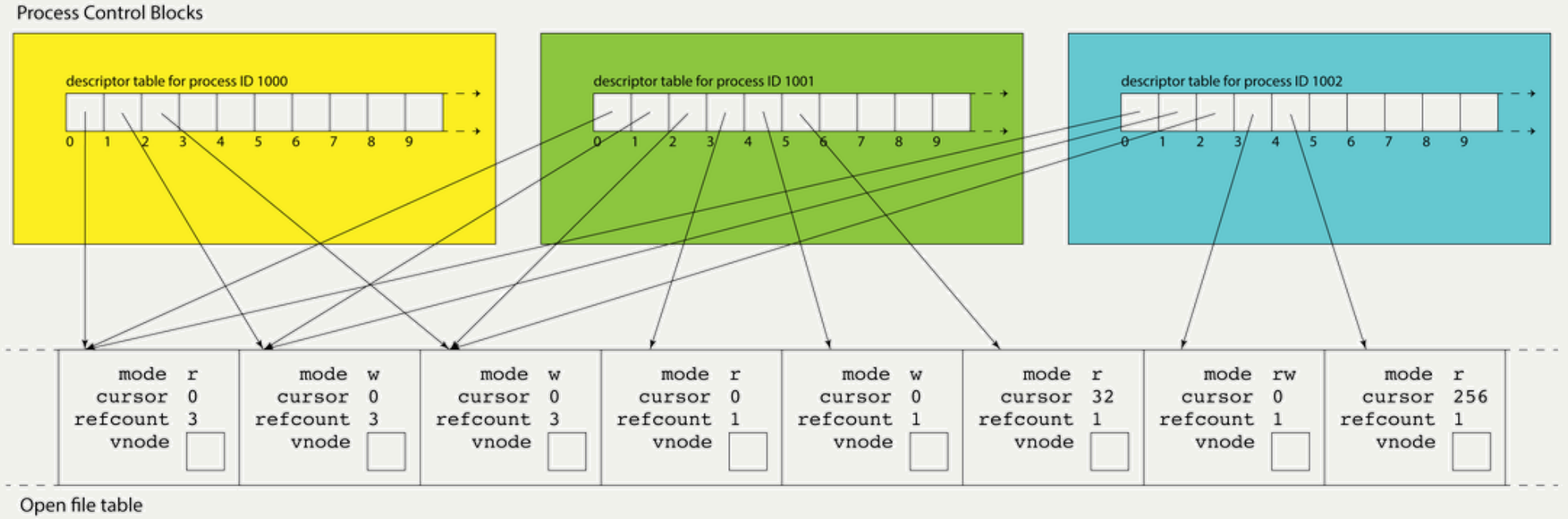





```
$ ./main 1> log.txt 2> log.txt

This is my coolest msg. You are
terrible.
```

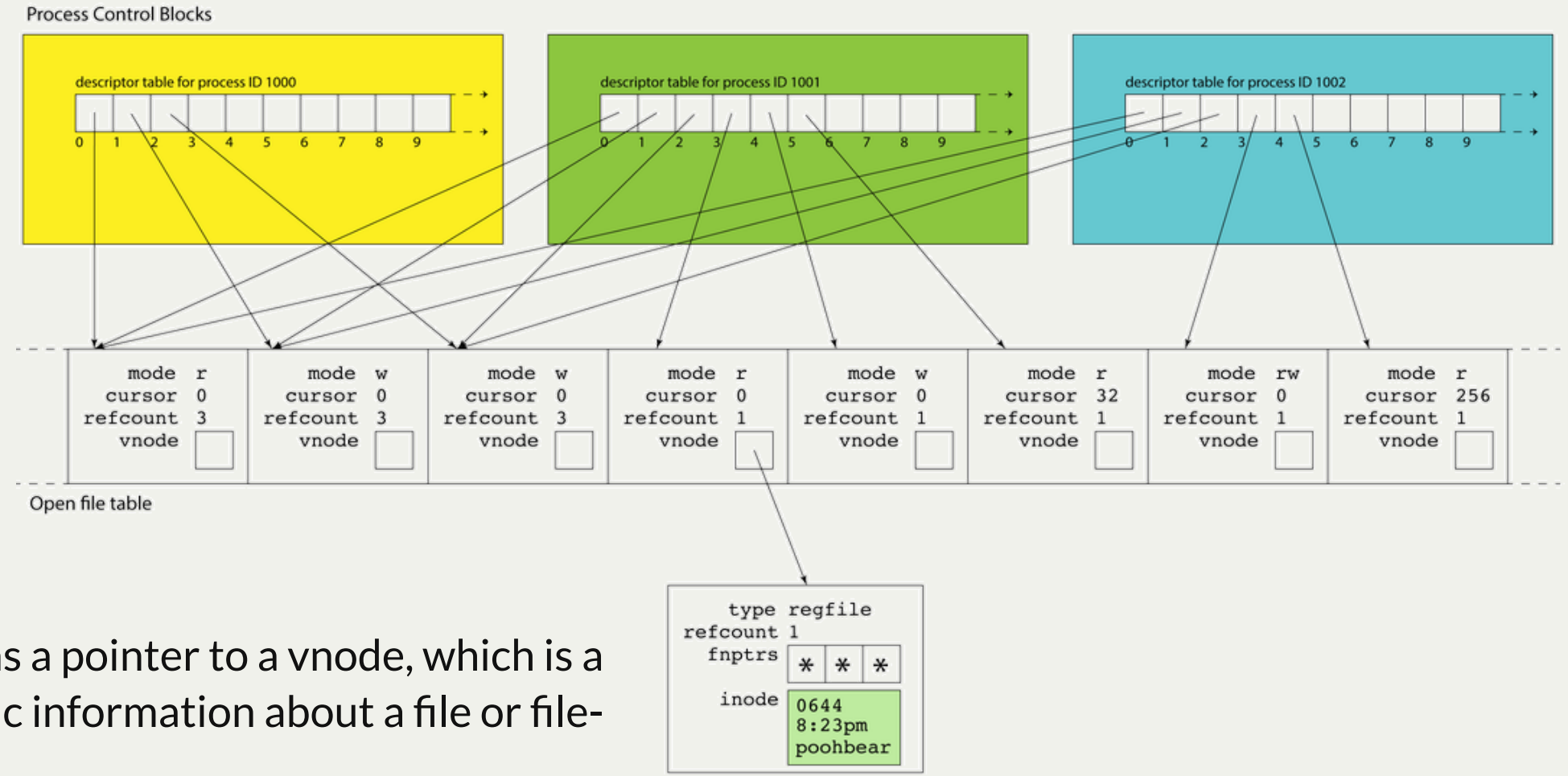
```
$ ./main 1> log.txt 2>&1

This is my error message.
 It's terrible.
This is my coolest msg. You are
```

File Table Details



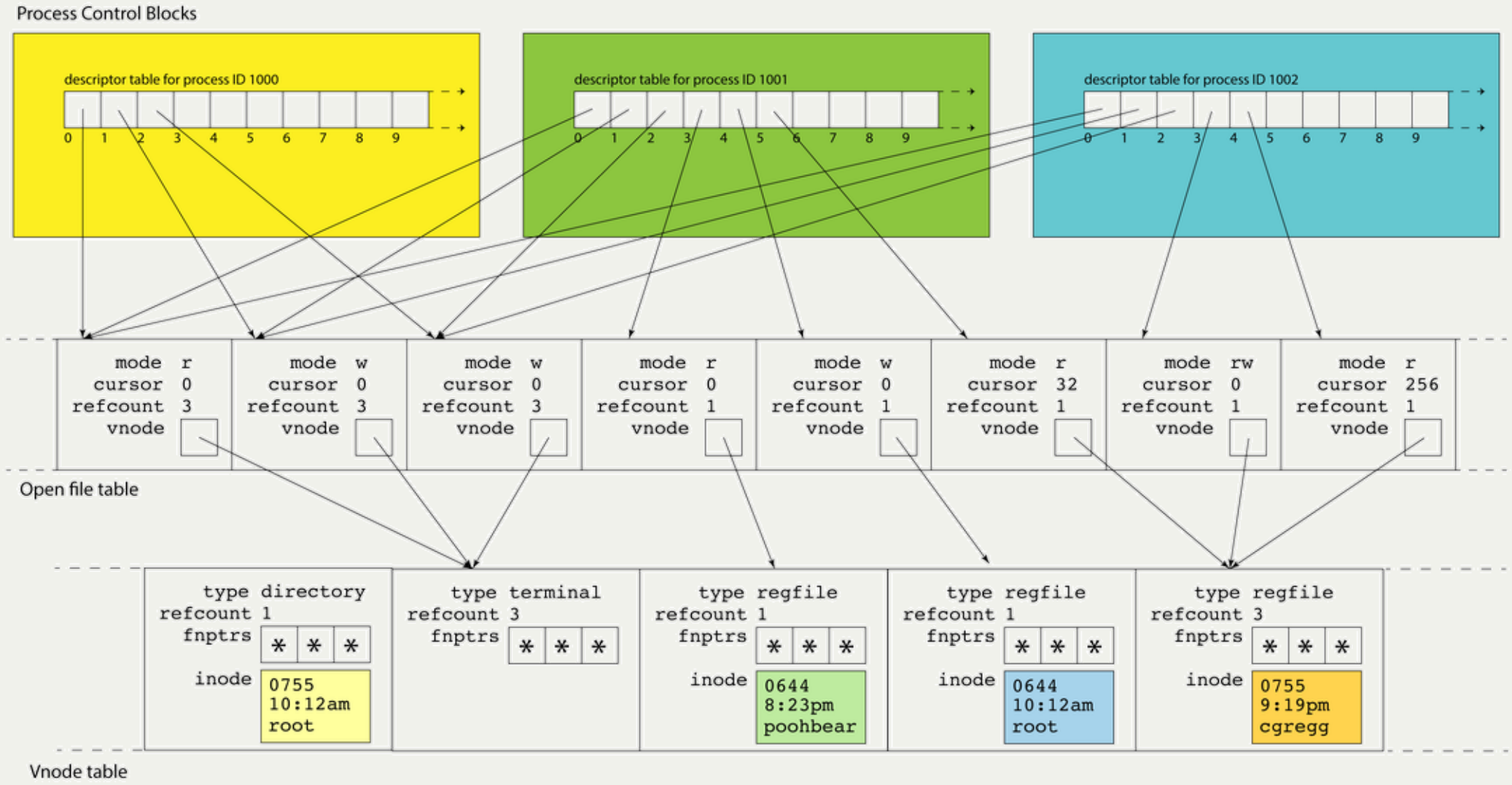
- Each process maintains its own descriptor table, but there is one, system-wide open file table. This allows for file resources to be shared between processes, as we've seen
- As drawn above, descriptors 0, 1, and 2 in each of the three PCBs alias the same three open files. That's why each of the referred 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



- 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!)

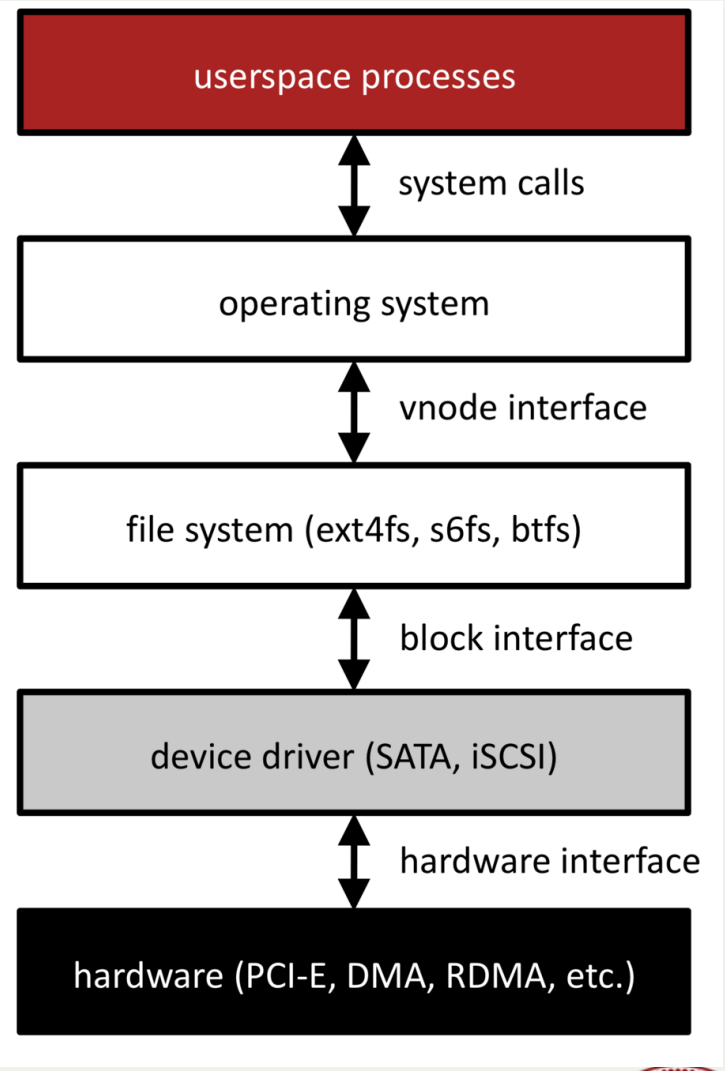
File Decriptors -> File Table -> vnode Table



- 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*



Questions about files in UNIX?

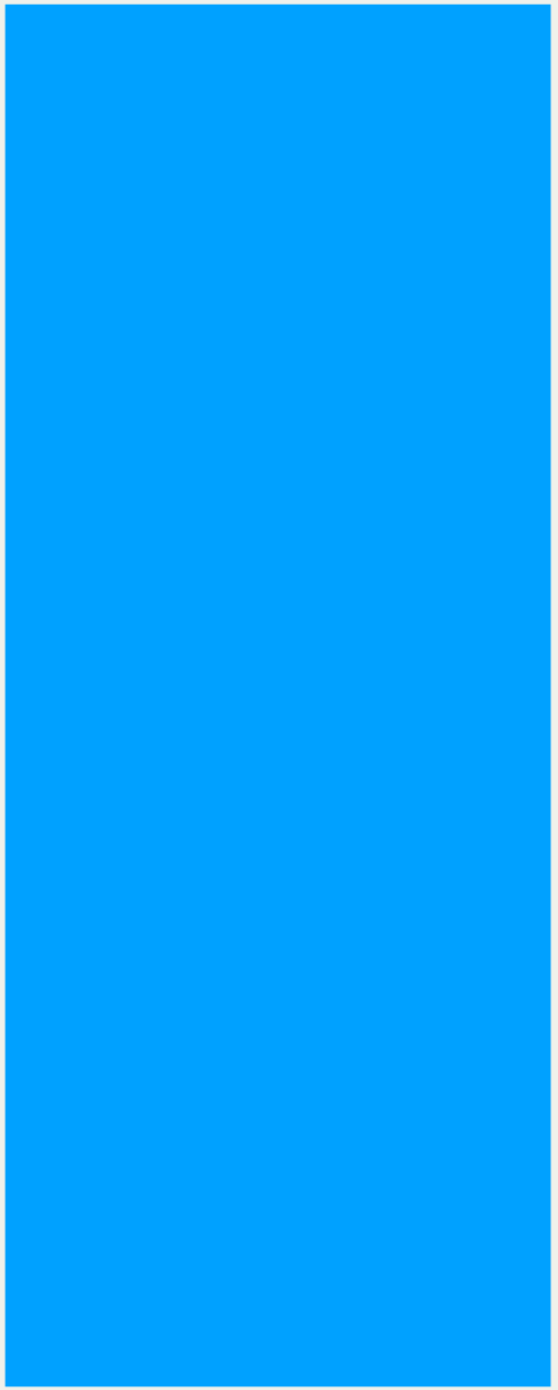
Memory mapped files

- **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"

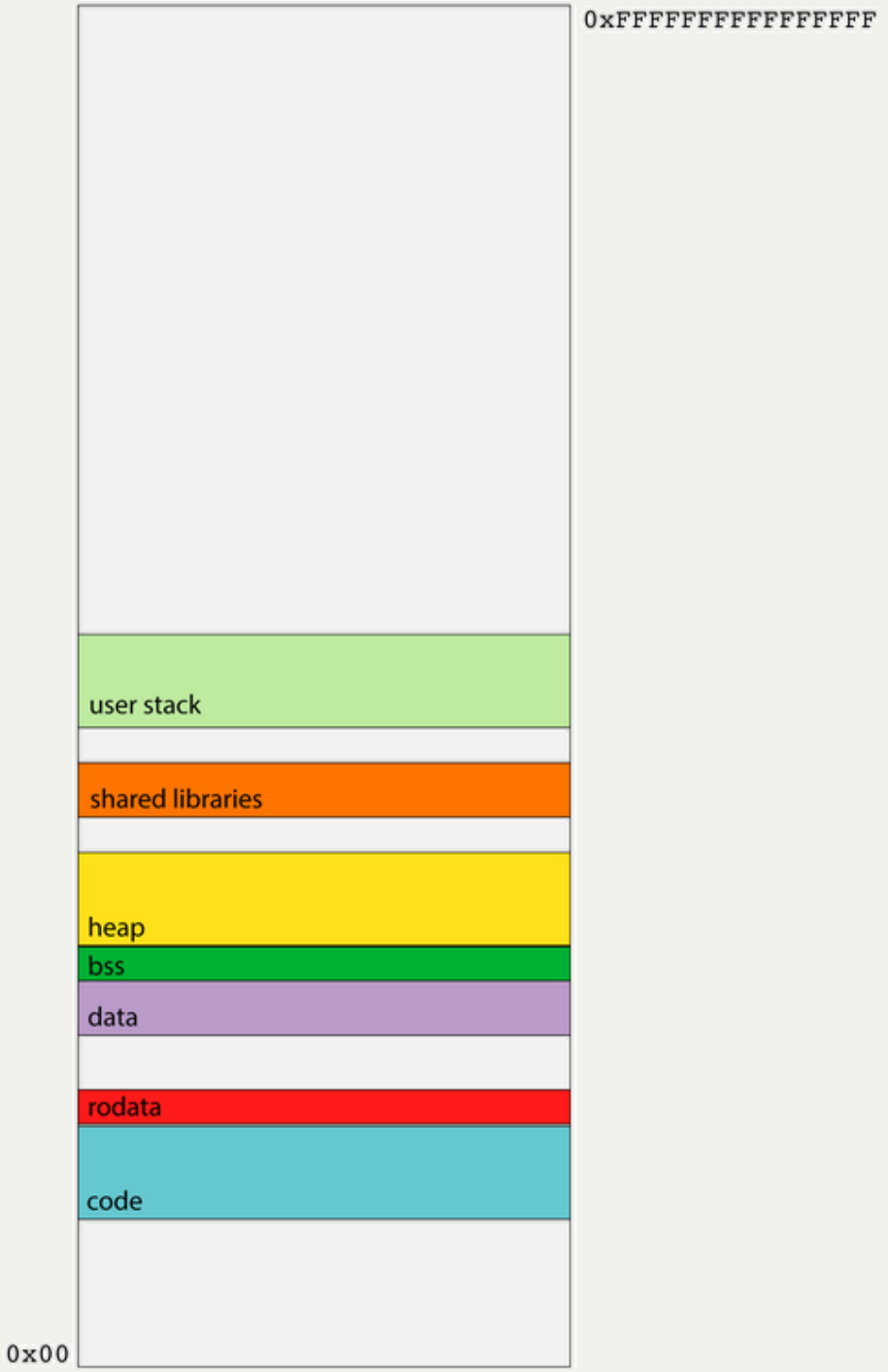
Process Address Spaces

- 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)



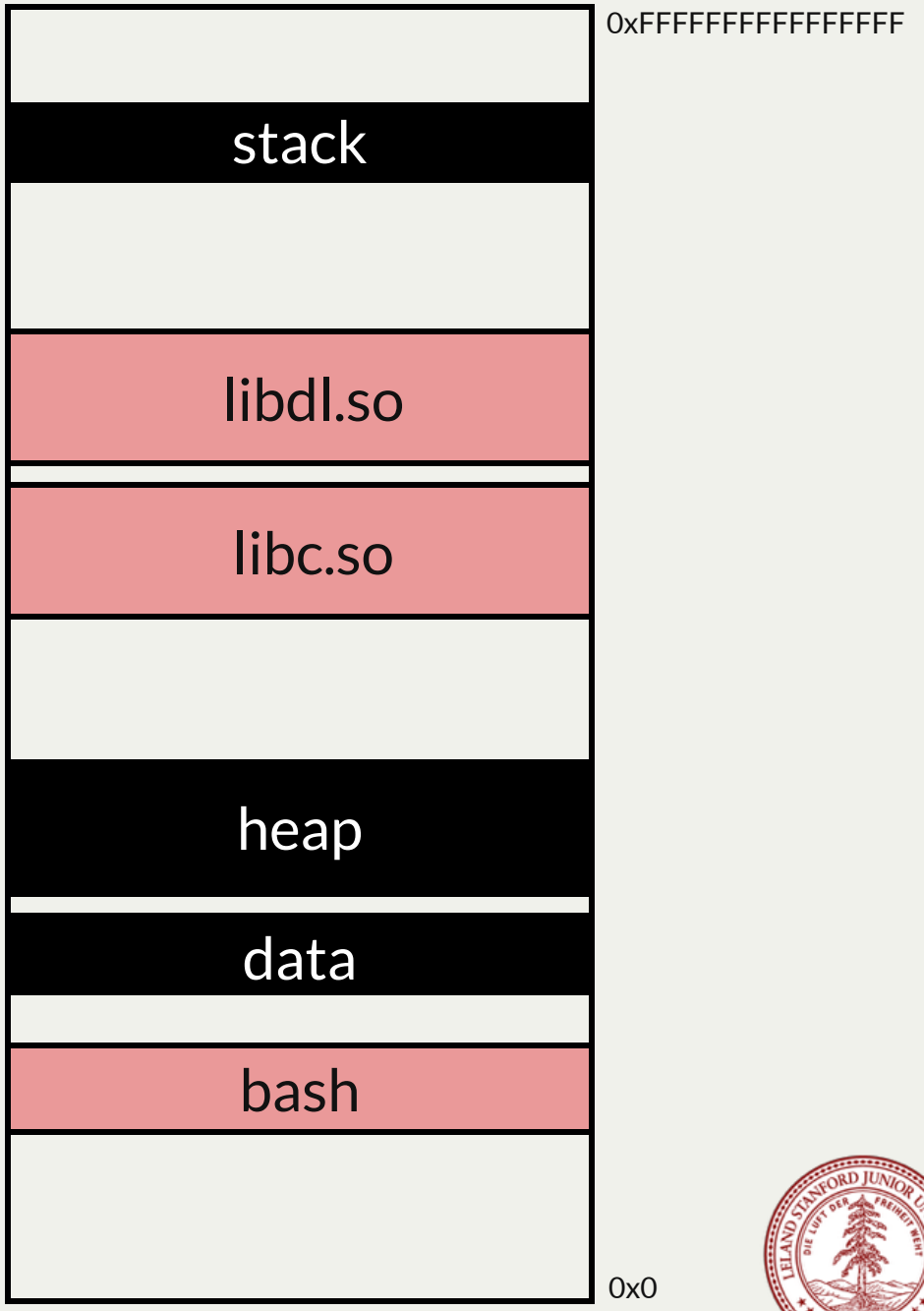
Memory Regions in a Process

- 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
  - Let's pmap a tcsh 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



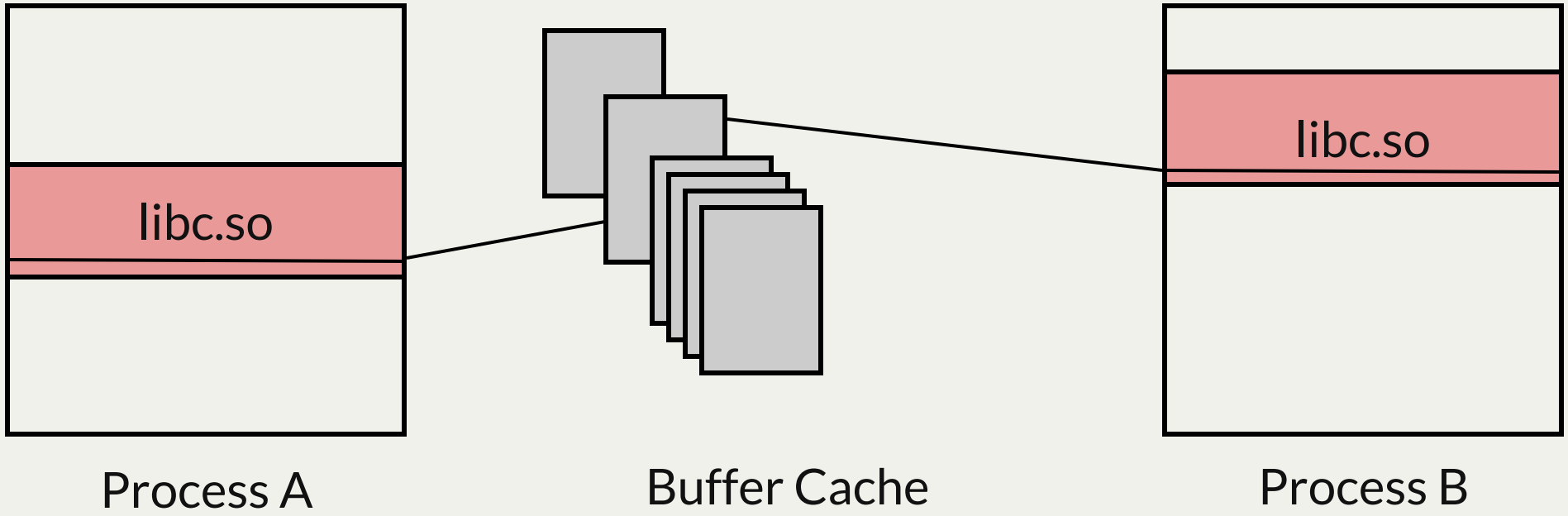
Memory Mapped Files

- 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

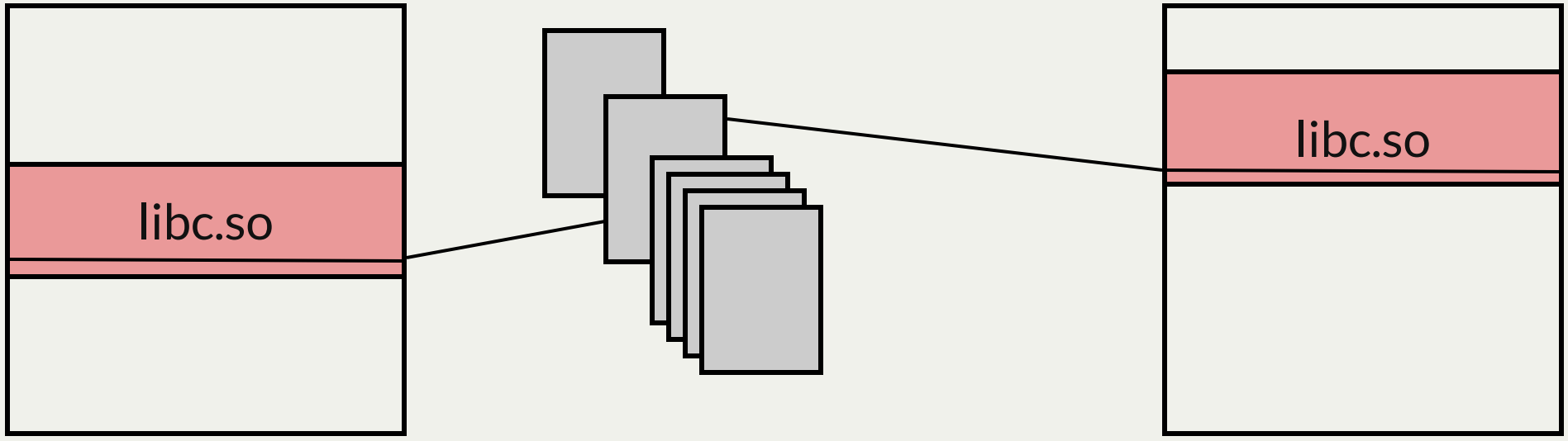
- The operating system maintains a *buffer cache*: a pool of (page-sized) pieces of files that are in memory
  - Caching: keeping pieces of used data in faster storage to improve performance
  - Buffer cache: keeping parts of files in use in memory so you don't have to hit disk
- Calls to **`read()`** 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

0xFFFFFFFFFFFFFFFF

- 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



0x0

Questions about mmap()?

Introduction to Multiprocessing

In the CS curriculum so far, your programs have operated in a *single process*, meaning, basically, that one program was running your code. The operating system gives your program the illusion that it was the only thing running, and that was that.

Now, we are going to move into the realm of multiprocessing, where you control more than one process at a time with your programs. You will ask the OS, "do these things *concurrently*", and it will.

Process

- **What is a process?**
  - When you start a program, it runs in a *single process*. It has a *process id* (an integer) that the OS assigns. A program can get its own process id with the **`getpid`** system call:

```c
// file: getpidEx.c
#include<stdio.h>
#include<stdlib.h>
#include <unistd.h> // getpid

int main(int argc, char **argv)
{
    pid_t pid = getpid();
    printf("My process id: %d\n",pid);
    return 0;
}
```

```
cgregg@myth57$ ./getpidEx
My process id: 7526
```

- The operating system *schedules* runnable processes to run on CPU cores.
  - Some processes are not runnable: waiting on **`sleep()`** , a **`read()`**, etc.
- There may be tens, hundreds, or thousands of processes "running" at once, but on a single-core system, only one can run at a time, and the OS decides which.
- On multicore machines (like most modern computers), multiple programs can literally run at the same time, one on each core.

fork()

- The **`fork()`** system call creates a new process
- If a program wants to launch a second process, it uses **`fork()`**
- **`fork()`** does exactly this:
  - It creates a new process that starts on the following instruction after the original, *parent*, process. The parent process *also* continues on the following instruction, as well.
  - The **`fork`** call returns a **`pid_t`** (an integer) to both processes. Neither is the actual **`pid`** of the process that receives it:
    - The parent process gets a return value that is the pid of the *child* process.
    - The child process gets a return value of 0, indicating that it is the child.
      - The child process does, indeed, have its own pid, but it would need to call **`getpid`** itself to retrieve it.
  - Everything is duplicated in the child process
    - File descriptors (increasing reference counts on file table entries)
    - Mapped memory regions (the address space)
      - But anonymous regions like stack, heap, etc. are copied (why?)

Fork Return Value

- The reason that the parent and its child get different return values from **`fork`** is twofold:
  - It is useful for the parent to know its child's pid. There is no other way for a parent to easily get its children's process ids (if a child wants to get its parent's pid, it can call **`getppid`**)
  - It differentiates them. It is almost a certainty that the parent and child will have different objectives after the **`fork`**, and it is useful for a process to know whether it is the parent or the child.
  - Here's a simple program that knows how to spawn new processes. It uses the system calls **`fork`**, **`getpid`**, and **`getppid`**. The full program can be viewed right here.

```
1 int main(int argc, char *argv[]) {
2   printf("Greetings from process %d! (parent %d)\n", getpid(), getppid());
3   pid_t pid = fork();
4   assert(pid >= 0);
5   printf("Bye-bye from process %d! (parent %d)\n", getpid(), getppid());
6   return 0;
7 }
```

Fork in Action!

```
1 int main(int argc, char *argv[]) {
2   printf("Greetings from process %d! (parent %d)\n", getpid(), getppid());
3   pid_t pid = fork();
4   assert(pid >= 0);
5   printf("Bye-bye from process %d! (parent %d)\n", getpid(), getppid());
6   return 0;
7 }
```

- Here is the output of two consecutive runs of the program:

```
1 myth60$ ./basic-fork
2 Greetings from process 29686! (parent 29351)
3 Bye-bye from process 29686! (parent 29351)
4 Bye-bye from process 29687! (parent 29686)
5
6 myth60$ ./basic-fork
7 Greetings from process 29688! (parent 29351)
8 Bye-bye from process 29688! (parent 29351)
9 Bye-bye from process 29689! (parent 29688
```

- There are a couple of things to note about this program:
  - The original process has a parent, which is the *shell* -- that is 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 - why?

Debugging Multiprocess Programs

- You might be asking yourself, *How do I debug two processes at once?* This is a very good question! **`gdb`** has built-in support for debugging multiple processes, as follows:
  - **`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.

A Tree of Forks

- **Second example: A tree of `fork` calls**
  - While you rarely have reason to use **`fork`** this way, it's instructive to trace through a short program where spawned processes themselves call **`fork`**. The full program can be viewed right here.

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

A Tree of Forks, Continued

- **Second example: A tree of `fork` calls**
  - Two samples runs on the right
  - Reasonably obvious: A single **`a`** is printed by the soon-to-be-great-grandparent process.
  - Less obvious: The first child and the parent each return from **`fork`** and continue running in mirror processes, each with their own copy of the global **`"abcd"`** string, and each advancing to the **`i++`** line within a loop that promotes a 0 to 1. It's hopefully clear now that two **`b`**'s will be printed.
  - Key questions to answer:
    - Why aren't the two **`b`**'s always consecutive?
    - How many **`c`**'s get printed?
    - How many **`d`**'s get printed?
    - Why is there a shell prompt in the middle of the output of the second run on the right?

```
myth60$ ./fork-puzzle
a
b
c
b
d
c
d
c
c
d
d
d
d
d
myth60$
```

```
myth60$ ./fork-puzzle
a
b
b
c
d
c
d
c
d
d
c
d
myth60$ d
d
d
```

Where and When Do We 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

Managing Your Progeny

- You've spawned a child and let it loose in the world: now what?

Managing Your Progeny

- **`waitpid`** can be used to temporarily block a process until a child process exits.

```
pid_t waitpid(pid_t pid, int *status, int options);
```

  - The first argument specifies whom to wait on, which for the moment is just the id of a child process that needs to complete before **`waitpid`** can return.
  - The second argument supplies the address of an integer where termination information can be placed (or we can pass in **`NULL`** if we don't care for the information).
  - The third argument is a collection of bitwise-or'ed flags we'll study later. For the time being, we'll just go with 0 as the required parameter value, which means that **`waitpid`** should only return when a process in the supplied wait set exits.
  - The return value is the pid of the child that exited, or -1 if **`waitpid`** was called and there were no child processes in the supplied wait set.

Example of waitpid()

- **Third example: Synchronizing between parent and child using `waitpid`**
  - Consider the following program, which is more representative of how **`fork`** really gets used in practice (full program, with error checking, is right here):

```
int main(int argc, char *argv[]) {
  printf("Before.\n");
  pid_t pid = fork();
  printf("After.\n");
  if (pid == 0) {
    printf("I am the child, and the parent will wait up for me.\n");
    return 110; // contrived exit status
  } else {
    int status;
    waitpid(pid, &status, 0)
    if (WIFEXITED(status)) {
      printf("Child exited with status %d.\n", WEXITSTATUS(status));
    } else {
      printf("Child terminated abnormally.\n");
    }
    return 0;
  }
 }
```

- Work with neighbor: what do you expect the output to be?

Synchronizing Between Parent and Child Using waitpid()

- The output is the same every single time the above program executes.

```
myth60$ ./separate
Before.
After.
After.
I am the child, and the parent will wait up for me.
Child exited with status 110.
myth60$
```

- The implementation directs the child process one way, the parent another.
- The parent process correctly waits for the child to complete using **`waitpid`**.
- The parent lifts child exit information out of the **`waitpid`** call, and uses the **`WIFEXITED`** macro to examine some high-order bits of its argument to confirm the process exited normally, and it uses the **`WEXITSTATUS`** macro to extract the lower eight bits of its argument to produce the child return value (which we can see is, and should be, 110).
- The **`waitpid`** call also donates child process-oriented resources back to the system.

How Deep is the Copy?

- This next example is more of a brain teaser, but it illustrates just how deep a clone the process created by **`fork`** really is (full program, with more error checking, is right here).

```
int main(int argc, char *argv[]) {
  printf("I'm unique and just get printed once.\n");
  bool parent = fork() != 0;
  if ((random() % 2 == 0) == parent) sleep(1); // force exactly one of the two to sleep
  if (parent) waitpid(pid, NULL, 0); // parent shouldn't exit until child has finished
  printf("I get printed twice (this one is being printed from the %s).\n",
         parent  ? "parent" : "child");
  return 0;
}
```

- The code emulates a coin flip to seduce exactly one of the two processes to sleep for a second, which is more than enough time for the child process to finish.
- The parent waits for the child to exit before it allows itself to exit. It's akin to the parent not being able to fall asleep until he/she knows the child has, and it's emblematic of the types of synchronization directives we'll be seeing a lot of this quarter.
- The final **`printf`** gets executed twice. The child is always the first to execute it, because the parent is blocked in its **`waitpid`** call until the child executes **`everything`**.

Multiple Children

- **Spawning and synchronizing with multiple child processes**
  - A parent can call **`fork`** multiple times, provided it reaps the child processes (via **`waitpid`**) once they exit. If we want to reap processes as they exit without concern for the order they were spawned, then this does the trick (full program checking right here):

```
int main(int argc, char *argv[]) {
  for (size_t i = 0; i < 8; i++) {
    if (fork() == 0) exit(110 + i);
  }
  while (true) {
    int status;
    pid_t pid = waitpid(-1, &status, 0);
    if (pid == -1) { assert(errno == ECHILD); break; }
    if (WIFEXITED(status)) {
      printf("Child %d exited: status %d\n", pid, WEXITSTATUS(status));
    } else {
      printf("Child %d exited abnormally.\n", pid);
    }
  }
  return 0;
}
```

Calling waitpid() on Multiple Children

- Pass -1 as the first argument to **`waitpid`**. **`man waitpid`**:
  - "The value of pid can be:
    - < -1 meaning wait for any child process whose process group ID is equal to the absolute value of pid.
    - -1 meaning wait for any child process.
    - 0 meaning wait for any child process whose process group ID is equal to that of the calling process.
    - > 0 meaning wait for the child whose process ID is equal to the value of pid.
  - Eventually, all children exit and **`waitpid`** correctly returns -1 to signal there are no more processes under the parent's jurisdiction.
  - When **`waitpid`** returns -1, it sets a global variable called **`errno`** to the constant **`ECHILD`** to signal **`waitpid`** returned -1 because all child processes have terminated. That's the "error" we want.

What's the difference?

- What's different about these two different invocations at the shell?
  - `$ xemacs main.c`
  - `$ xemacs main.c &`
- What does & make the shell do differently?

Questions about fork()?

Lecture 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

- Processes 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 processes to share a single copy of data (e.g., library code)
  - Processes *virtualize* memory and the CPU

- Processes create new processes with fork(), the processes run *concurrently*
  - The parent and child are identical except for the return value
    - Share file descriptors, memory regions, etc. (some memory, like the heap, is copied)
  - Parent can wait for children to exit (and learn about their exit) with waitpid()
  - This *concurrency* lets your program use more than one core: much of the quarter will look at the complications