Lecture 04: Filesystem Data Structures and System Calls

Principles of Computer Systems

Spring 2019

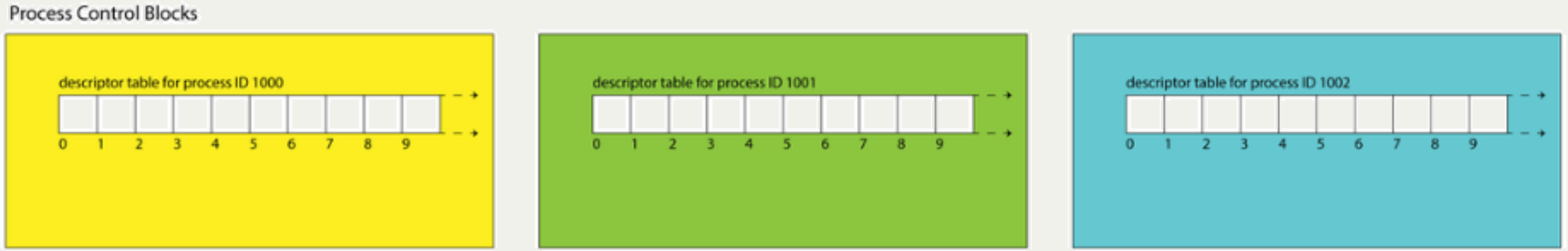
Stanford University

Computer Science Department

Lecturer: Chris Gregg

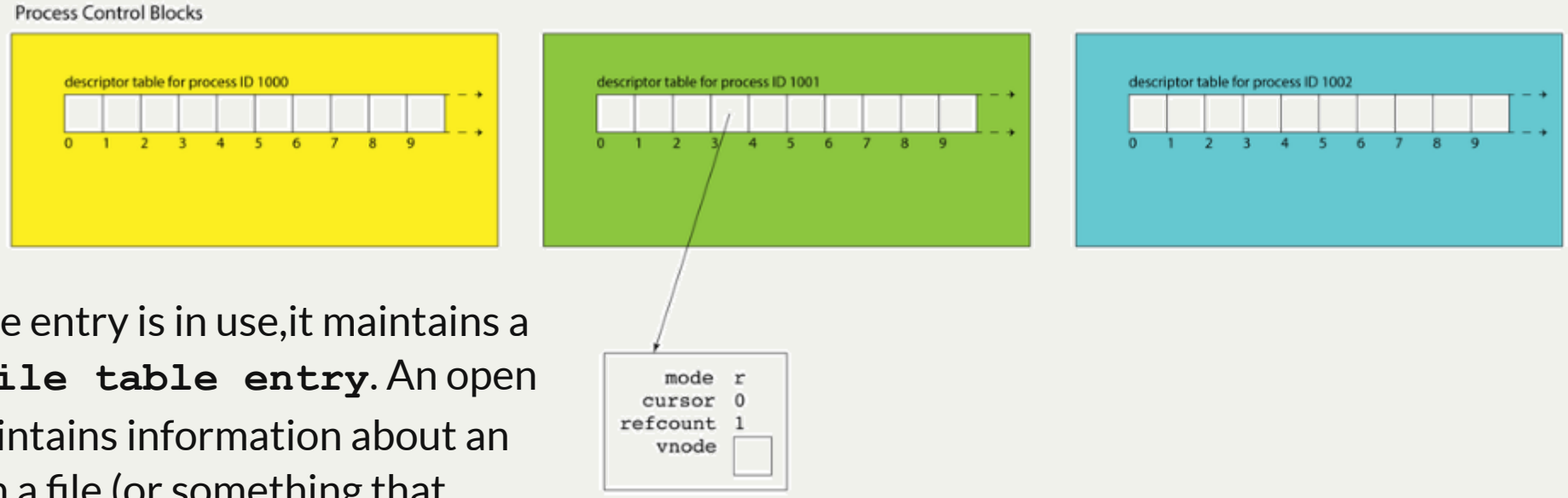
PDF of this presentation

Lecture 04: Filesystem Data Structures



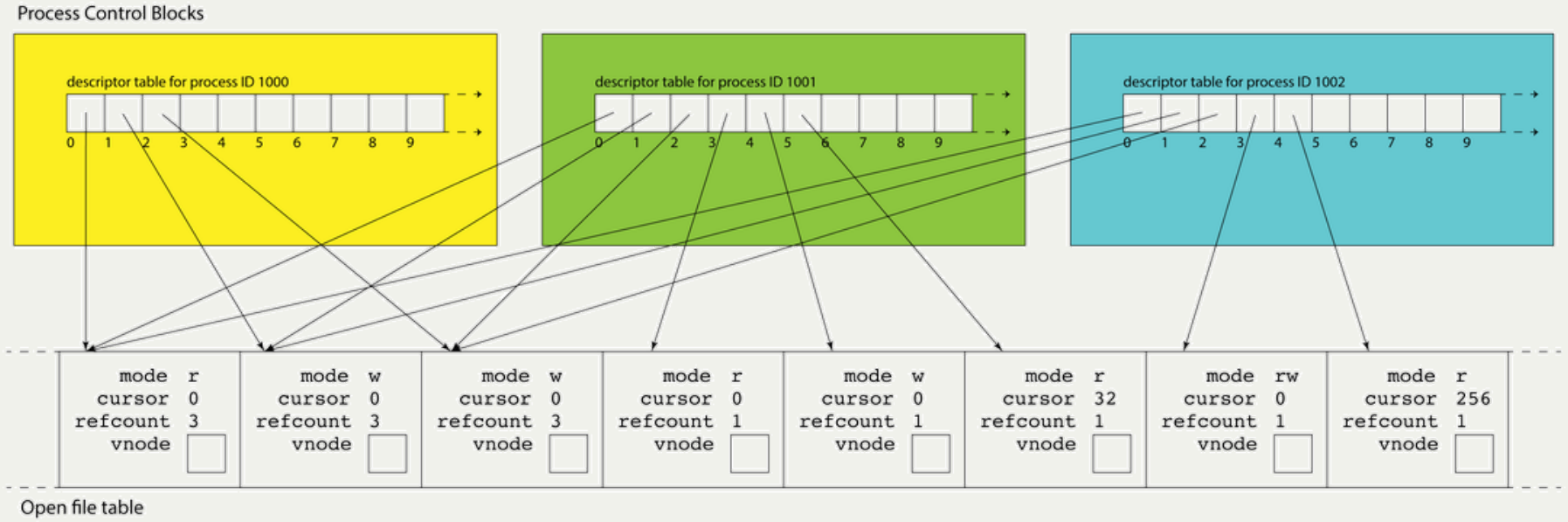
- Linux maintains a data structure for each active process. These data structures are called **`process control blocks`**, and they are stored in the **`process table`**.
- Process control blocks store many things (the user who launched it, what time it was launched, CPU state, etc.). Among the many items it stores is the **`descriptor table`**.
- Each process maintains its own set of descriptors. Descriptors 0, 1, and 2 are understood to be treated as standard input, standard output, and standard error, but there are no predefined meanings for descriptors 3 and up. Descriptors 0, 1, and 2 are most often bound to the terminal.
- The user program sees a descriptor as the identifier needed to interact with a resource (most often a file) via **`read`**, **`write`**, and **`close`** calls. Internally, that descriptor is an index into the descriptor table.
- The process control block tracks which descriptors are in use and which ones aren't. When allocating a new descriptor, the OS typically chooses the smallest available number.

Lecture 04: Filesystem Data Structures



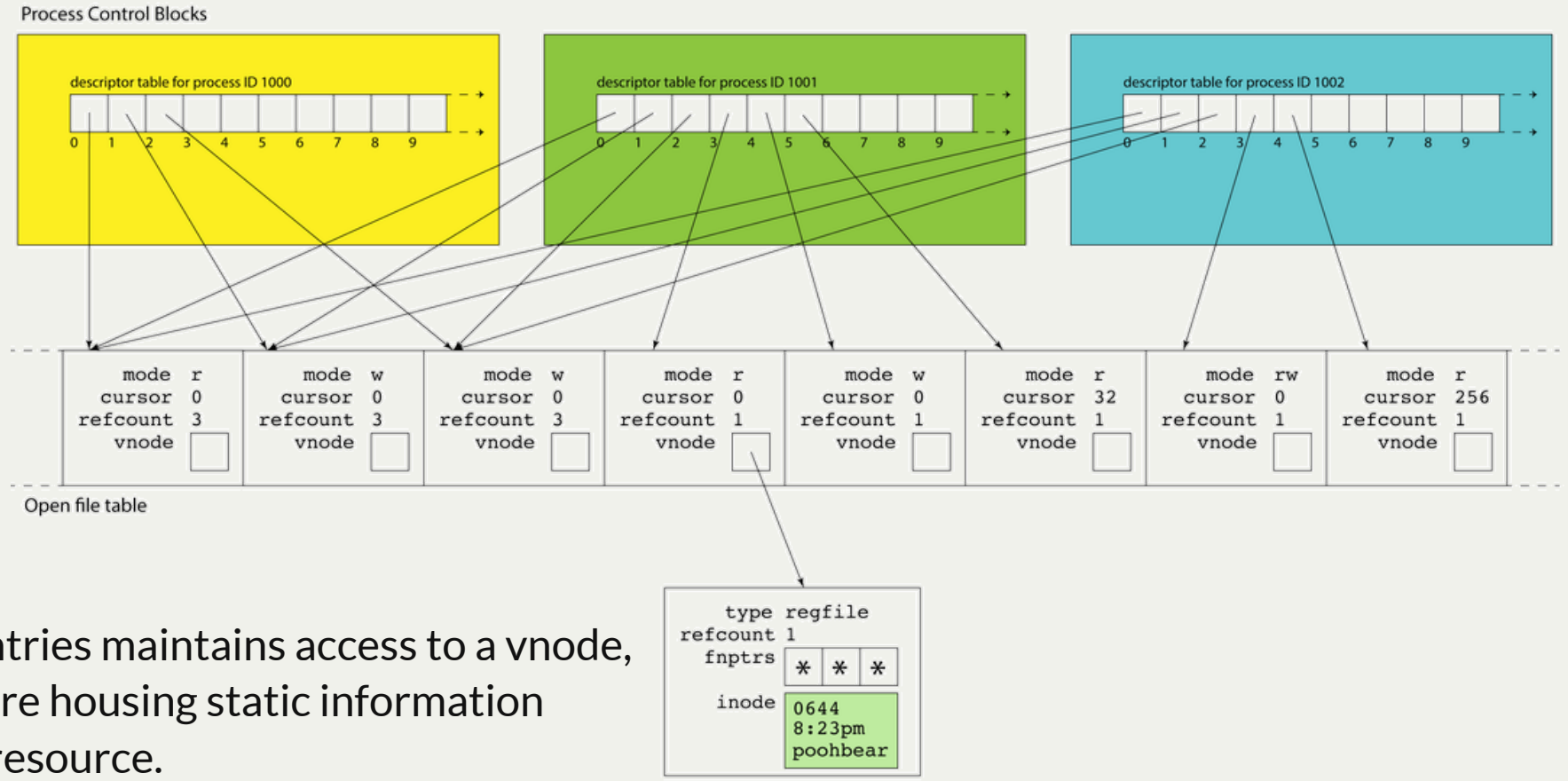
- If a descriptor table entry is in use,it maintains a link to an **`open file table entry`**. An open file table entry maintains information about an active session with a file (or something that behaves like a file, like terminal, or a network connection).
- Each table entry tracks information specific to the dynamics of that session. **`mode`** tracks whether we're reading, writing, or both. **`cursor`** tracks a position within the file payload. **`refcount`** tracks the number of descriptors across all processes that refer to that session. (We'll discuss the **`vnode`** field in a moment.)
- The illustration here calls out just one file table entry referenced by process 1001, descriptor 3. A call to **`open(filename, O_RDONLY)`** from within the second of three processes might result in the above.

Lecture 04: Filesystem Data Structures



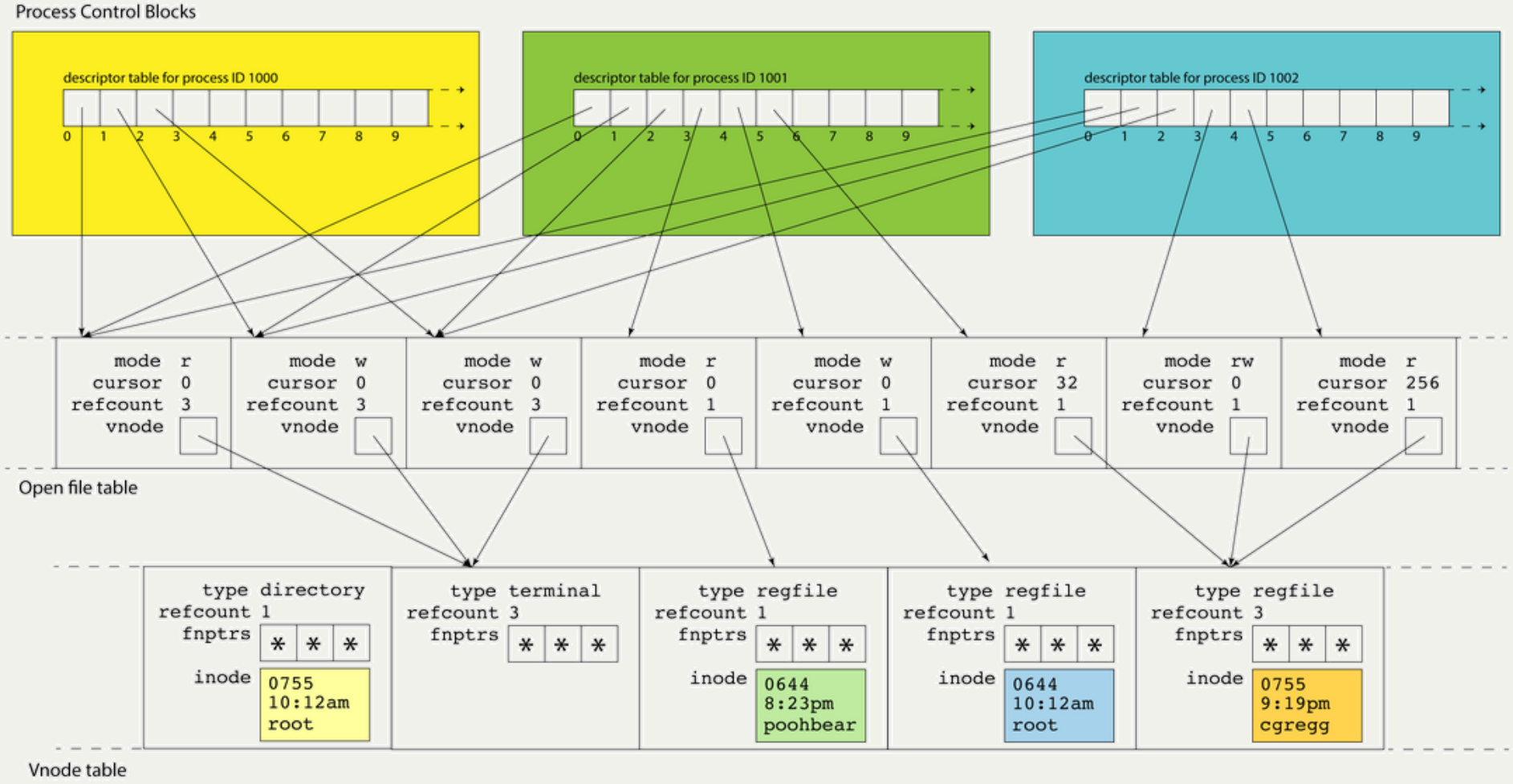
- At any one time, there are multiple active processes, each of which typically has many—well, at least three—open descriptors.
- Each process maintains its own descriptor table, but there is only one, system-wide open file table. This allows for file resources to be shared between processes, and we'll soon see just how common shared file resources really are.
- As drawn above, descriptors 0, 1, and 2 in each of the three PCBs alias the same three sessions. 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.

Lecture 04: Filesystem Data Structures



- Each of the open file entries maintains access to a vnode, which itself is a structure housing static information about a file or file-like resource.
- The data structure stores file type (e.g. regular file, directory, symlink, terminal), a refcount, the collection of function pointers that should be used to read, write, and otherwise interact with the resource, and, if applicable, a copy of the inode that resides on the filesystem on behalf of that file. In this sense, the vnode is an inode cache that brings information about the file (e.g. file size, owner, permissions, etc.) so that it can be accessed more quickly.

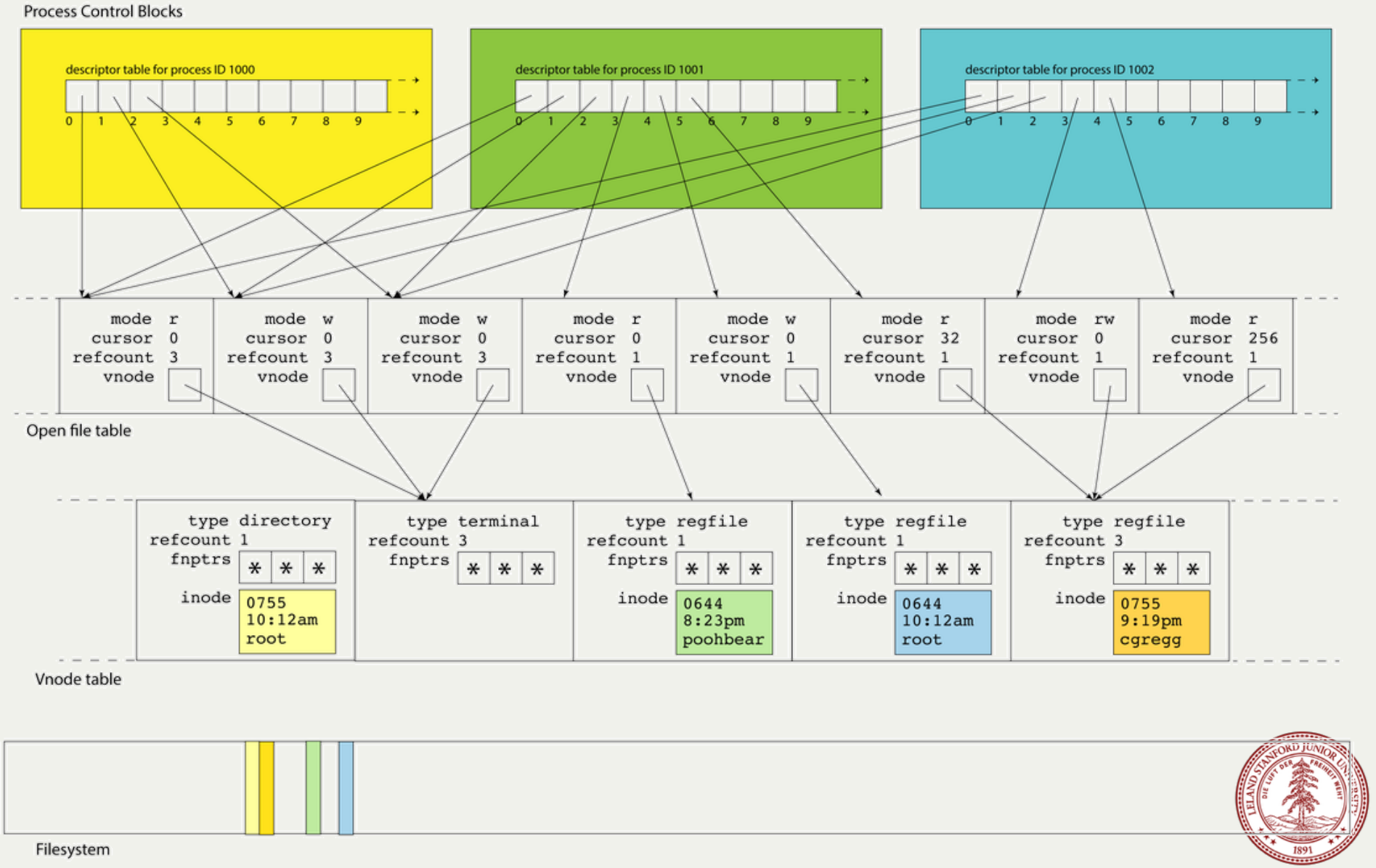
Lecture 04: Filesystem Data Structures



- 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.

Lecture 04: Filesystem Data Structures

- None of these kernel-resident data structures are visible to users. Note the filesystem itself is a completely different component, and that filesystem inodes of open files are loaded into vnode table entries. The yellow inode in the vnode is an in-memory replica of the yellow sliver of memory in the filesystem.

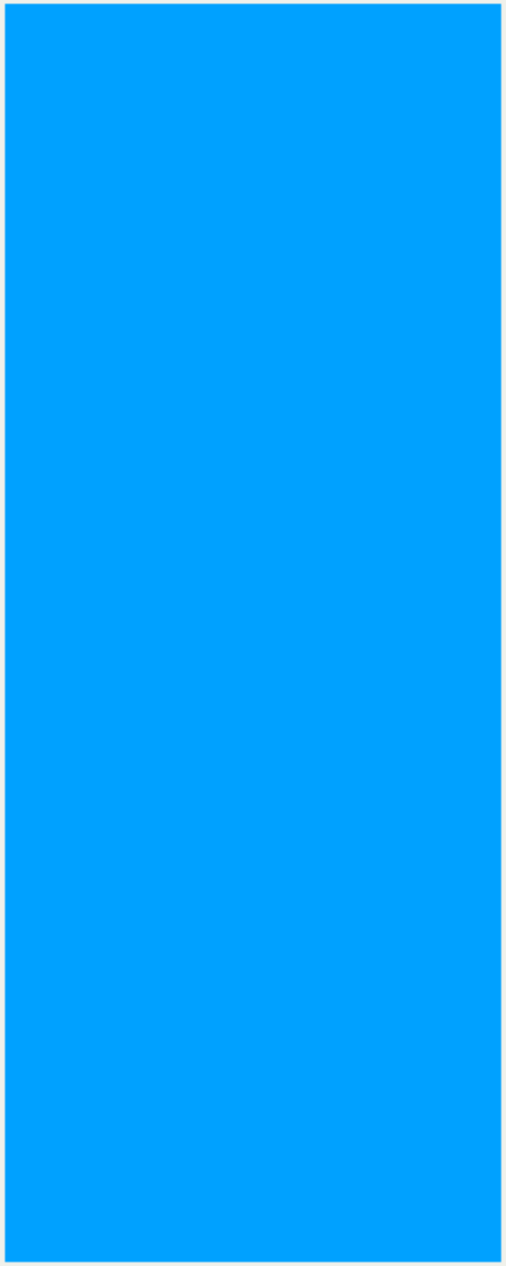


Lecture 04: System Calls

- **System calls** are functions that our programs use to interact with the OS and request some core service be executed on their behalf.
  - Examples of system calls we've seen this quarter: **`open`**, **`read`**, **`write`**, **`close`**, **`stat`**, and **`lstat`**. We'll see may others in the coming weeks.
  - Functions like **`printf`**, **`malloc`**, and **`opendir`** aren't themselves system calls. They're C library functions that themselves rely on system calls to get their jobs done.
- Unlike traditional user functions (the ones we write ourselves, **`libc`** and **`libstdc++`** library functions), system calls need to execute in some privileged mode so they can access data structures, system information, and other OS resources intentionally hidden from user code.
- The implementation of **`open`**, for instance, needs to access all of the filesystem data structures for existence and permissioning. Filesystem implementation details should be be hidden from the user, and permission information should be respected as private.
- The information loaded and manipulated by **`open`** should be inaccessible to the user functions that call **`open`**.
  - Restated, privileged information shouldn't be discoverable.
- That means we need a different call and return model for system calls than we have for traditional functions.

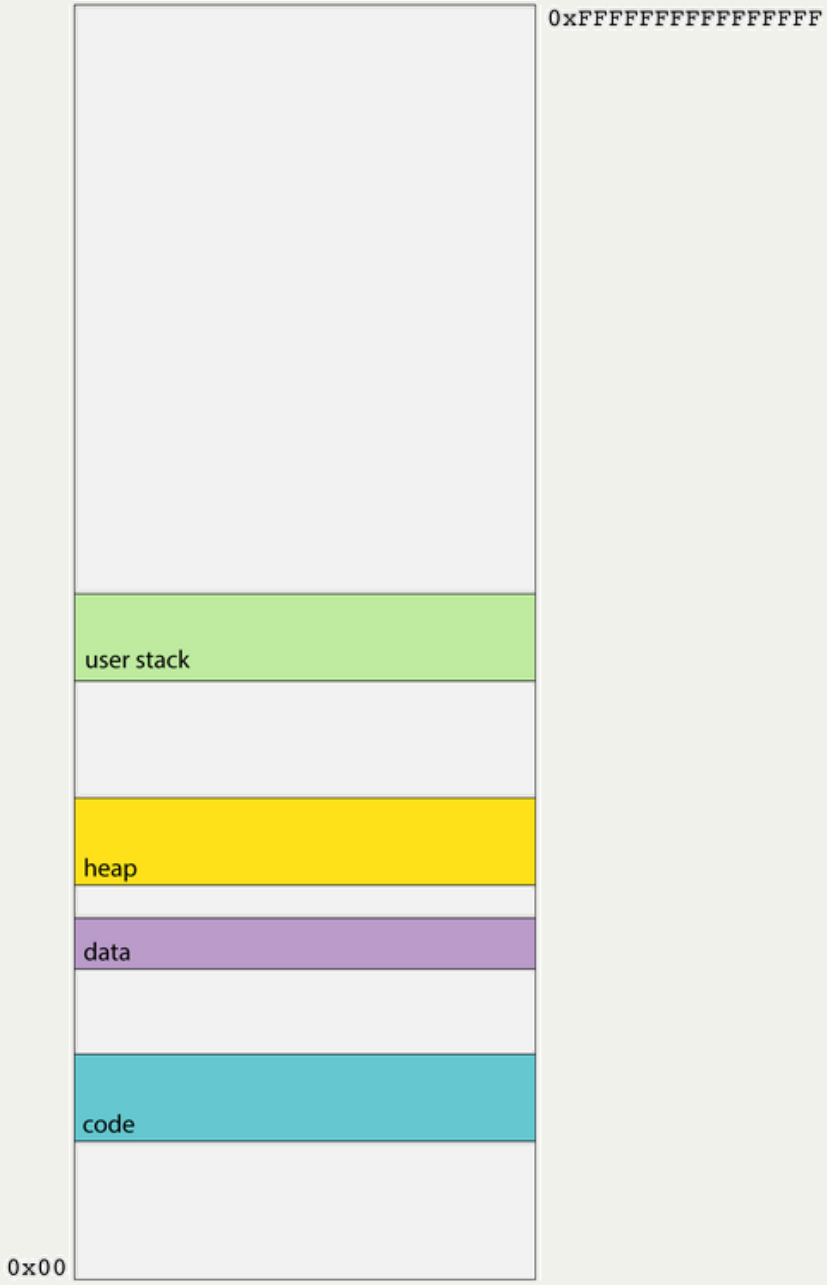
Lecture 04: System Calls

- 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.



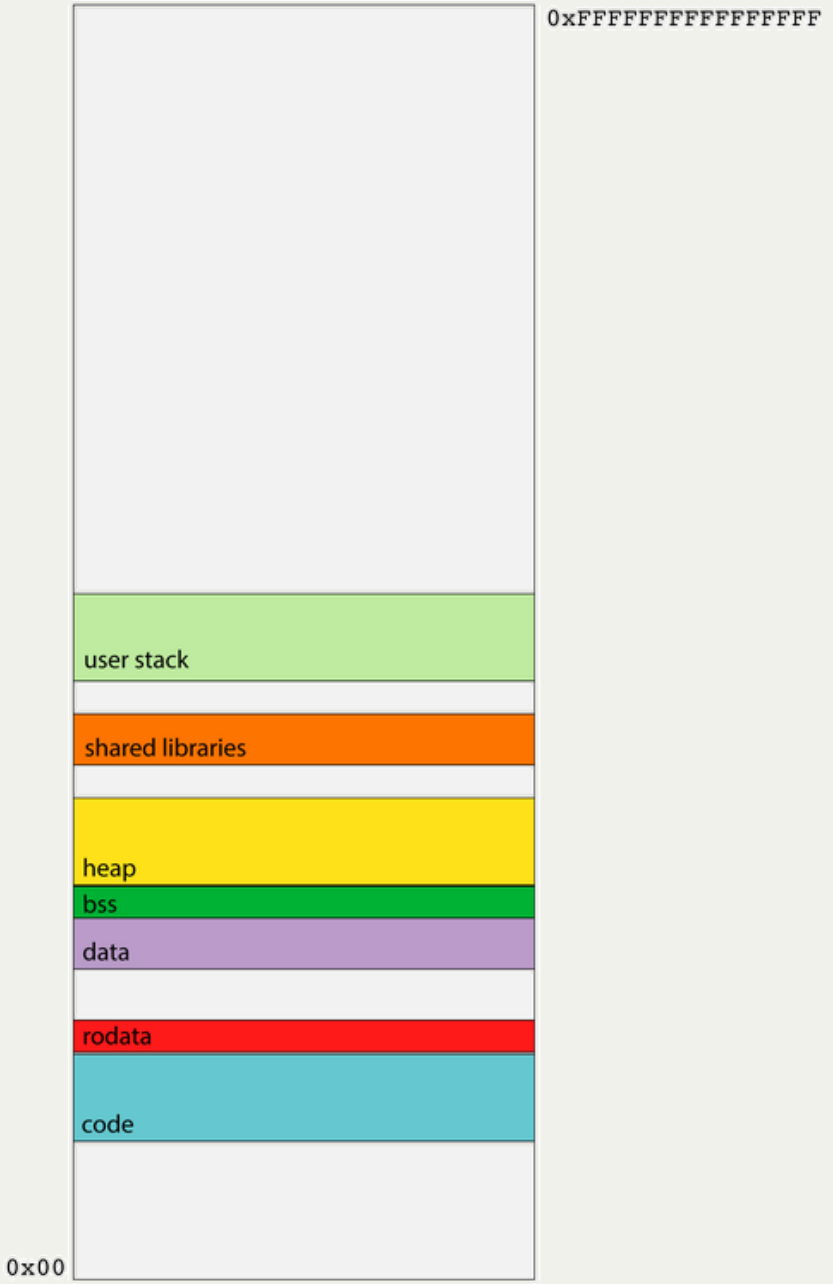
Lecture 04: System Calls

- The code segment stores all of the assembly code instructions specific to your process. The address of the currently executing instruction is stored in the **`%rip`** register, and that address is typically drawn from the range of addresses managed by the code segment.
- The data segment intentionally rests directly on top of the code segment, and it houses all of the explicitly initialized global variables that can be modified by the program.
- The heap is a software-managed segment used to support the implementation of **`malloc`**, **`realloc`**, **`free`**, and their C++ equivalents. It's initially very small, but grows as needed for processes requiring a good amount of dynamically allocated memory.
- The user stack segment provides the memory needed to manage user function call and return along with the scratch space needed by function parameters and local variables.



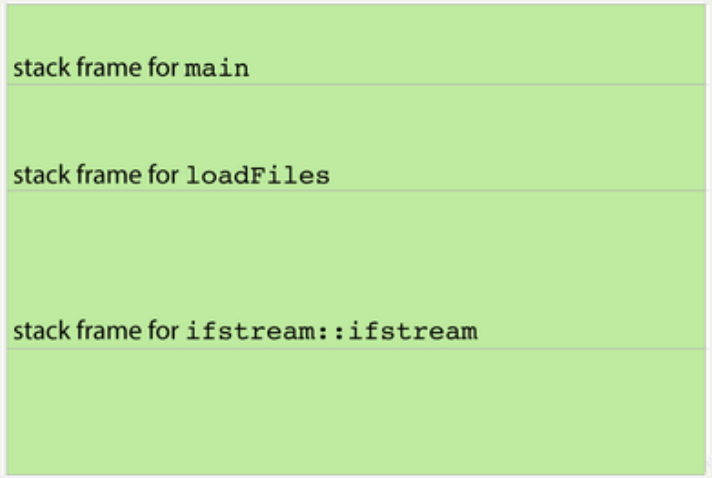
Lecture 04: System Calls

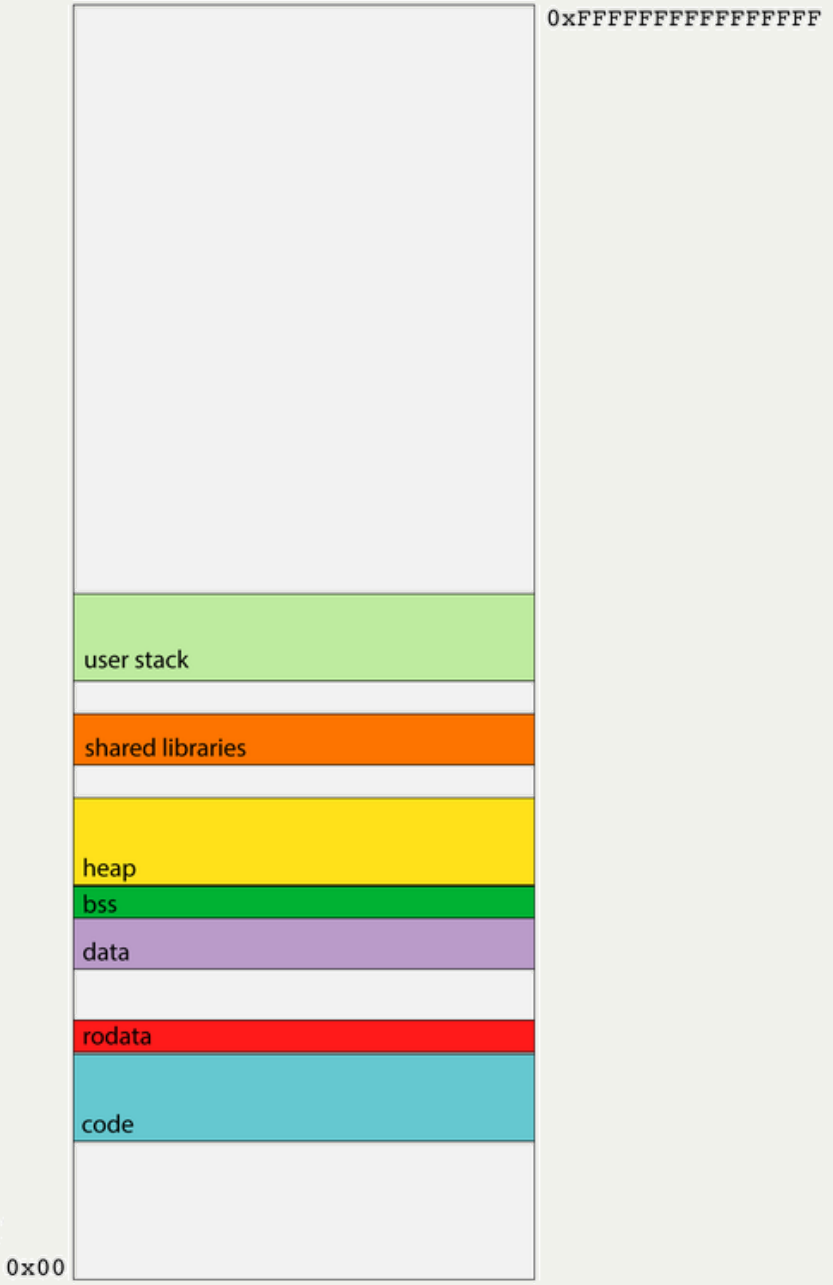
- There are other relevant segments that haven't been called out in prior classes—at least not in CS107 here (although some are covered in CS107E).
- The **`rodata`** segment also stores global variables, but only those which are immutable—i.e. constants. As the runtime isn't supposed to change anything read-only, the segment housing constants can be protected so any attempts to modify it are blocked by the OS.
- The **`bss`** segment houses the uninitialized global variables, which are defaulted to be zero (one of the few situations where pure C provides default values).
- The **`shared library`** segment links to shared libraries like **`libc`** and **`libstdc++`** with code for routines like C's **`printf`**, C's **`malloc`**, or C++'s **`getline`**. Shared libraries get their own segment so all processes can trampoline through some glue code—that is, the minimum amount of code necessary—to jump into the one copy of the library code that exists for all processes.



Lecture 04: System Calls

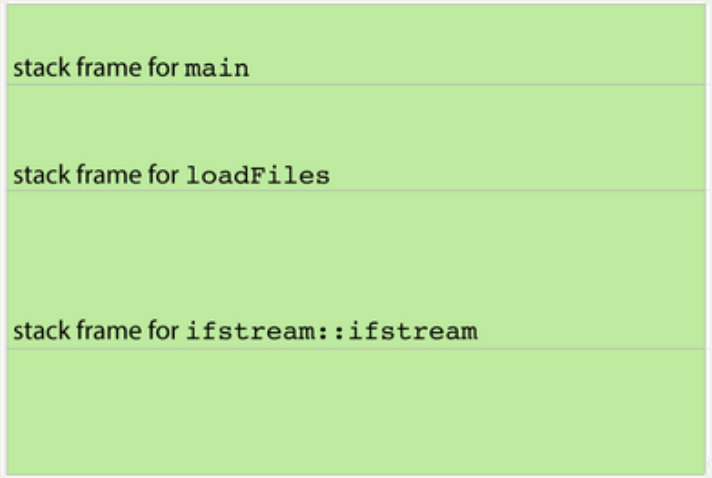
- The user stack maintains a collection of stack frames for the trail of currently executing user functions.
- 64-bit process runtimes rely on **`%rsp`** to track the address boundary between the in-use portion of the user stack and the portion that's on deck to be used should the currently executing function invoke a subroutine.
- The x86 64 runtime relies on **`callq`** and **`retq`** instructions for user function call and return.
- The first six parameters are passed through **`%rdi`**, **`%rsi`**, **`%rdx`**, **`%rcx`**, **`%r8`**, and **`%r9`**. The stack frame is used as general storage for partial results that need to be stored somewhere other than a register (e.g. a seventh incoming parameter)

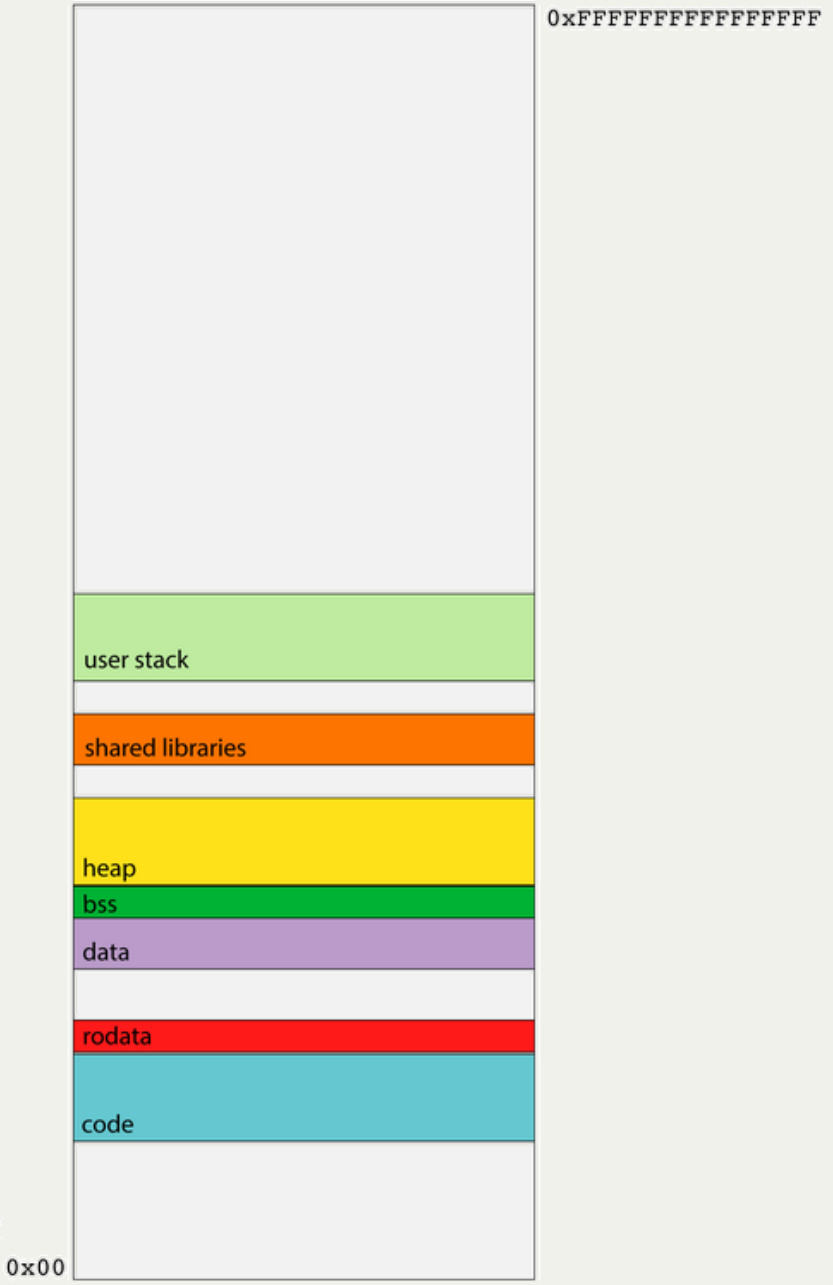




Lecture 04: System Calls

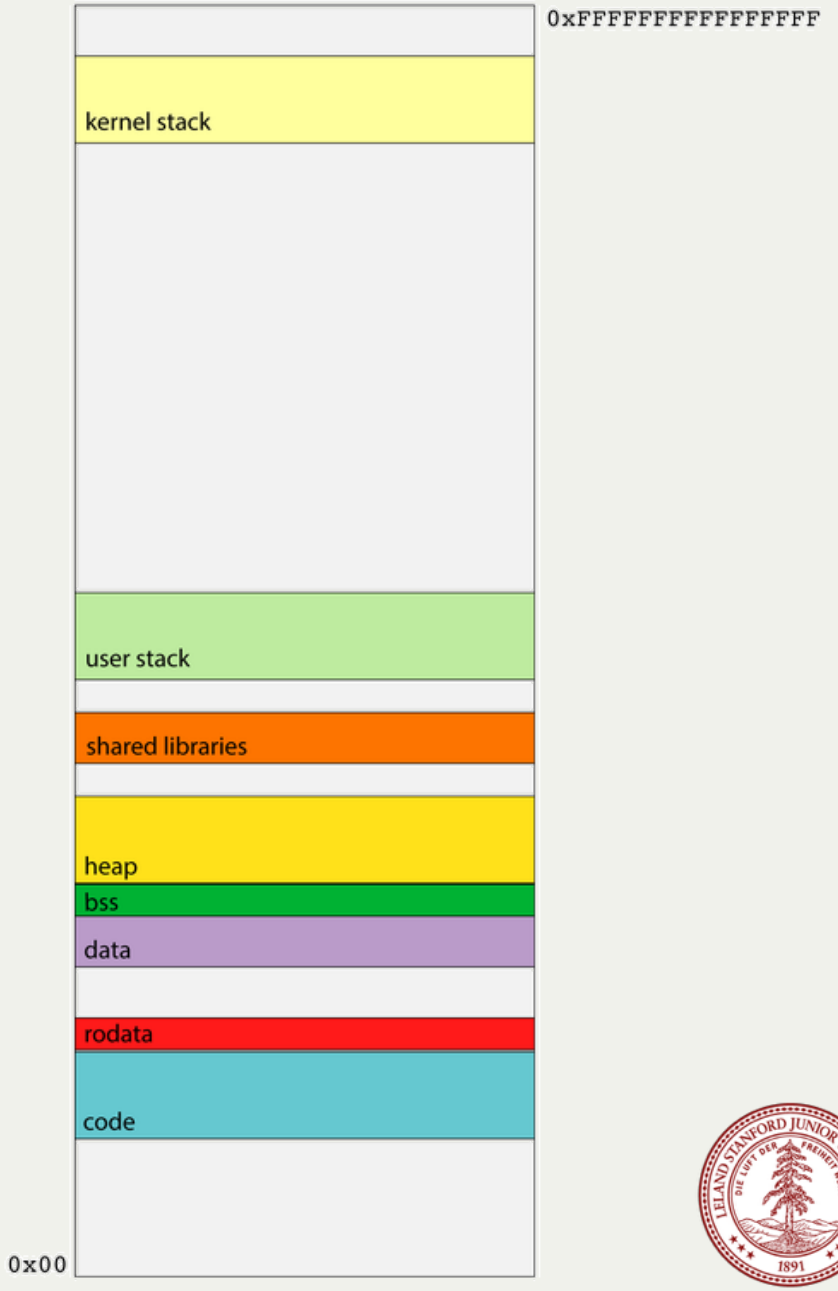
- The user function call and return protocol, however, does little to encapsulate and privatize the memory used by a function.
- Consider, for instance, the execution of **`loadFiles`** as per the diagram below. Because **`loadFiles`**'s stack frame is directly below that of its caller, it can use pointer arithmetic to advance beyond its frame and examine—or even update—the stack frame above it.
- After **`loadFiles`** returns, **`main`** could use pointer arithmetic to descend into the ghost of **`loadFiles`**'s stack frame and access data **`loadFiles`** never intended to expose.
- Functions are supposed to be modular, but the function call and return protocol's support for modularity and privacy is pretty soft.





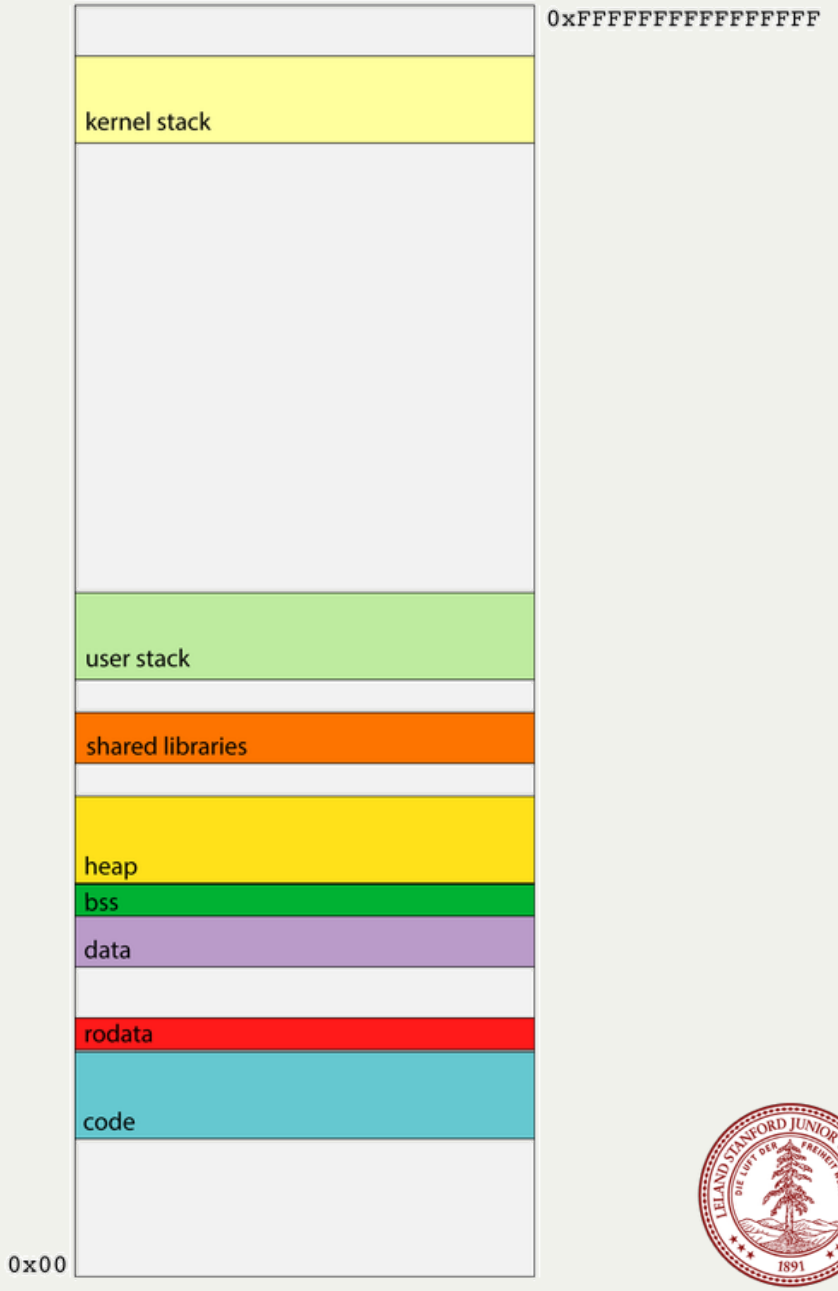
Lecture 04: System Calls

- System calls like **`open`** and **`stat`** need access to OS implementation detail that should not be exposed or otherwise accessible to the user program.
- That means the activation records for system calls need to be stored in a region of memory that users can't touch, and the system call implementations need to be executed in a privileged, superuser mode so that it has access to information and resources that traditional functions shouldn't have.
- The upper half of a process's address space is **`kernel space`**, and none of it is visible to traditional user code.
- Housed within kernel space is a kernel stack segment, itself used to organize stack frames for system calls.
- **`callq`** is used for user function call, but **`callq`** would dereference a function pointer we're **not permitted** to dereference, since it resides in kernel space.
- We need a different call and return model for system calls—one that doesn't rely on **`callq`**.



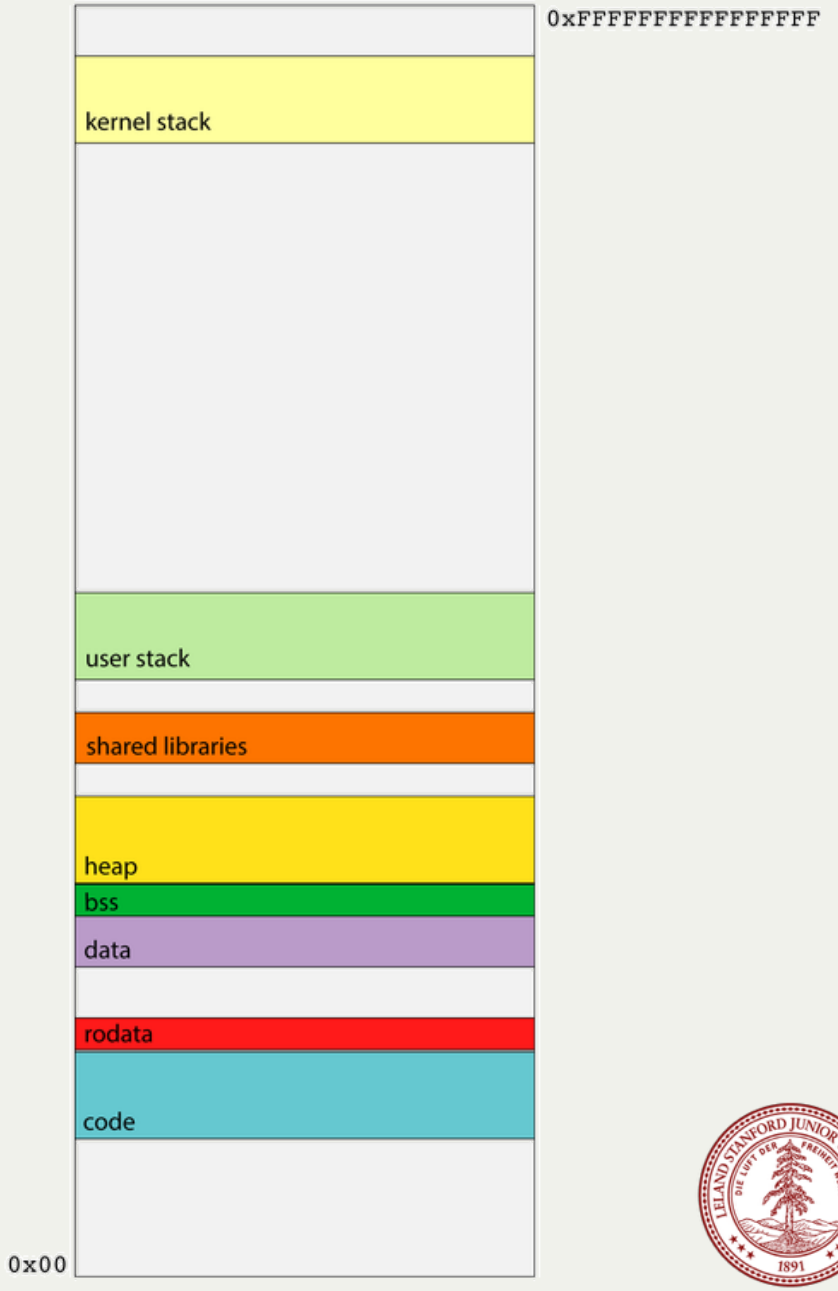
Lecture 04: System Calls

- The relevant opcode is placed in **`%rax`**. Each system call has its own opcode (e.g. 0 for **`read`**, 1 for **`write`**, 2 for **`open`**, 3 for **`close`**, 4 for **`stat`**, and so forth).
- The system call arguments—there are at most 6—are evaluated and placed in **`%rdi`**, **`%rsi`**, **`%rdx`**, **`%r10`**, **`%r8`**, and **`%r9`**. Note the fourth parameter is **`%r10`**, not **`%rcx`**.
- The system issues a software interrupt (otherwise known as a **trap**) by executing **`syscall`**, which prompts an interrupt **handler** to execute in superuser mode.
- The interrupt handler builds a frame in the kernel stack, executes the relevant code, places any return value in **`%rax`**, and then executes **`iretq`** to return from the interrupt handler, revert from superuser mode, and execute the instruction following the **`syscall`**.
- If **`%rax`** is negative, **`errno`** is set to abs(**`%rax`**) and **`%rax`** is updated to contain a **`-1`**. If **`%rax`** is nonnegative, it's left as is. The value in is **`%rax`** then extracted by the caller as any return value would be.



Lecture 04: System Calls

- System Calls Summary
  - We use system calls because we don't want user code to have access to sensitive parts of the system, such as raw hardware or networking.
  - It isn't possible to restrict access through the normal call/return stack-framed-based function calling procedure.
    - A program can modify any data it has access to, and a program's entire stack is completely accessible to the program.
    - If the OS let a use program muck around with hardware, shared data structures (e.g., the open file table), or other sensitive information, this would be a security nightmare.
  - A system call uses an *interrupt* to transfer control to the OS kernel. The user can only call a set of well-defined system calls, and there is little room for a security breach.
  - Once the kernel is running a system call, it is in complete control of the system, and can access the necessary resources to fulfill the system call's needs.
  - After a system call, the kernel returns control to the user program.



Lecture 04: Introduction to Multiprocessing

Until now, we have been studying how programs interact with hardware, and now we are going to start investigating how programs interact with the *operating system*.

In the CS curriculum so far, your programs have operated in a *single process*, meaning, basically, that one program was running your code, line-for-line. The operating system made it look like your program 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 tell the OS, "do these things *concurrently*", and it will.

Lecture 04: Introduction to Multiprocessing

- **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
```

- All programs are associated with one or more process, and a process is *scheduled* to run its code by the OS. The OS has to schedule all currently-running processes (i.e., processes that are waiting to run their next line of code), and it does so on a time-shared basis.
- In other words: there may be tens, hundreds, or thousands of processes "running" at once, but on a single-processor system, only one can run at a time, and this is coordinated by the OS.
- On multi-processor machines (like most modern computers), multiple programs can literally run at the same time, one-per processor.

Lecture 04: Introduction to Multiprocessing

- **New system call: `fork`**
  - A program may decide that it wants to run multiple processes itself. We will see many examples of why a program may want to do this as the course progresses.
  - If a program wants to launch a second process, it uses the **`fork`** system call.
  - **`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.
    - **All** memory is identical between the parent and child, though it is **not** shared (it is copied).

Lecture 04: Introduction to Multiprocessing

- **New system call: `fork`**
  - The reason that the parent and its child get different return values from **`fork`** is twofold:
    - 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.
    - 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`**)
  - 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 }
```

Lecture 04: Introduction to Multiprocessing

```
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.

Lecture 04: Introduction to Multiprocessing

- **`fork`** is called once, but it returns twice.
  - **`fork`** knows how to clone the calling process, synthesize a virtually identical copy of it, and schedule the copy as if it were running all along.
    - All segments (data, bss, init, stack, heap, text) are faithfully replicated.
    - All open file descriptors are replicated, and these copies are donated to the clone.
  - As a result, the output of our program is the output of two processes.
    - We should expect to see a single greeting but two separate bye-byes.
    - Each bye-bye is inserted into the console by two different processes. The OS's process scheduler dictates whether the child or the parent gets to print its bye-bye first.
  - **`getpid`** and **`getppid`** return the process id of the caller and the process id of the caller's parent, respectively.

Lecture 04: Introduction to Multiprocessing

- 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.

Lecture 04: Introduction to Multiprocessing

- Differences between parent calling **`fork`** and child generated by it:
  - The most obvious difference is that each gets a unique process id. That's important. Otherwise, the OS can't tell them apart.
  - Another key difference: **`fork`**'s return value in the two processes
    - When **`fork`** returns in the parent process, it returns the pid of the new child
    - When **`fork`** returns in the child process, it returns 0. Again, that isn't to say the child's pid is 0, but rather that **`fork`** elects to return a 0 as a way of allowing the child process to easily self-identify as the child process.
    - The return value can be used to dispatch each of the two processes in a different direction (although in this introductory example, we don't do that).

Lecture 04: Introduction to Multiprocessing

- Differences between parent calling **`fork`** and child generated by it:
  - The most obvious difference is that each gets a unique process id. That's important. Otherwise, the OS can't tell them apart.
  - Another key difference: **`fork`**'s return value in the two processes
    - When **`fork`** returns in the parent process, it returns the pid of the new child
    - When **`fork`** returns in the child process, it returns 0. Again, that isn't to say the child's pid is 0, but rather that **`fork`** elects to return a 0 as a way of allowing the child process to easily self-identify as the child process.
    - The return value can be used to dispatch each of the two processes in a different direction (although in this introductory example, we don't do that).
  - **All** data segments are replicated. Aside from checking the return value of **`fork`**, there is virtually no difference in the two processes, and they both continue after **`fork`** as if they were the original process.
  - There is **no** default sharing of data between the two processes, though the parent process can **`wait`** (more below) for child processes to complete.
  - You can use *shared memory* to communicate between processes, but this must be explicitly set up before making **`fork`** calls (you will not be responsible for shared memory in this course)

Lecture 04: Introduction to Multiprocessing

- **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 }
```

Lecture 04: Introduction to Multiprocessing

- **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-grandaddy 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
```

Lecture 04: Introduction to Multiprocessing

- **Third example: Synchronizing between parent and child using `waitpid`**
  - **`waitpid`** can be used to temporarily block one process until a child process exits.

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

    - The first argument specifies the wait set, which for the moment is just the id of the 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.

Lecture 04: Introduction to Multiprocessing

- **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;
  }
 }
```

Lecture 04: Introduction to Multiprocessing

- **Third example: Synchronizing between parent and child using `waitpid`**
  - The output is the same every single time the above program is executed.

```
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.

Lecture 04: Introduction to Multiprocessing

- **Synchronizing between parent and child using `waitpid`**
  - 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`**.

Lecture 04: Introduction to Multiprocessing

- **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;
}
```

Lecture 04: Introduction to Multiprocessing

- **Spawning and synchronizing with multiple child processes**
  - Note we feed a -1 as the first argument to **`waitpid`**. That -1 states we want to hear about **any** child as it exits, and pids are returned in the order their processes finish.
  - 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.

```
myth60$ ./reap-as-they-exit
Child 1209 exited: status 110
Child 1210 exited: status 111
Child 1211 exited: status 112
Child 1216 exited: status 117
Child 1212 exited: status 113
Child 1213 exited: status 114
Child 1214 exited: status 115
Child 1215 exited: status 116
myth60$
```

```
myth60$ ./reap-as-they-exit
Child 1453 exited: status 115
Child 1449 exited: status 111
Child 1448 exited: status 110
Child 1450 exited: status 112
Child 1451 exited: status 113
Child 1452 exited: status 114
Child 1455 exited: status 117
Child 1454 exited: status 116
myth60$
```

Lecture 04: Introduction to Multiprocessing

- **Spawning and synchronizing with multiple child processes**
  - We can do the same thing we did in the first program, but monitor and reap the child processes in the order they are forked.
  - Check out the abbreviated program below (full program with error checking right here):

```
int main(int argc, char *argv[]) {
  pid_t children[8];
  for (size_t i = 0; i < 8; i++) {
    if ((children[i] = fork()) == 0) exit(110 + i);
  }
  for (size_t i = 0; i < 8; i++) {
    int status;
    pid_t pid = waitpid(children[i], &status, 0);
    assert(pid == children[i]);
    assert(WIFEXITED(status) && (WEXITSTATUS(status) == (110 + i)));
    printf("Child with pid %d accounted for (return status of %d).\n",
           children[i], WEXITSTATUS(status));
  }
  return 0;
}
```

Lecture 04: Introduction to Multiprocessing

- **Spawning and synchronizing with multiple child processes**
  - This version spawns and reaps processes in some first-spawned-first-reaped manner.
  - The child processes aren't required to exit in FSFR order.
  - In theory, the first child thread could finish last, and the reap loop could be held up on its very first iteration until the first child really is done. But the process zombies—yes, that's what they're called—are reaped in the order they were forked.
  - Below is a sample run of the **`reap-in-fork-order`** executable. The pids change between runs, but even those are guaranteed to be published in increasing order.

```
myth60$ ./reap-as-they-exit
Child with pid 4689 accounted for (return status of 110).
Child with pid 4690 accounted for (return status of 111).
Child with pid 4691 accounted for (return status of 112).
Child with pid 4692 accounted for (return status of 113).
Child with pid 4693 accounted for (return status of 114).
Child with pid 4694 accounted for (return status of 115).
Child with pid 4695 accounted for (return status of 116).
Child with pid 4696 accounted for (return status of 117).
myth60$
```