CS110: Principles of Computer Systems

Autumn 2021
Jerry Cain
PDF

Lecture 05: Understanding System Calls

- **System calls** (syscalls) are functions that a user program invoke to interact with the OS and request some core service be executed on its behalf.
    - Examples of system calls we've already seen this quarter: **`open`**, **`read`**, **`write`**, and **`close`**. We'll see many others in the coming weeks.
    - Functions like **`printf`**, **`malloc`**, **`fopen`**, and **`opendir`** are not syscalls. They're C library functions that themselves rely on syscalls 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 and necessarily 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 hidden from the user, and permission information should be respected as private.
- The information loaded and manipulated by **`open`** musn't be visible to the user functions that call **`open`**. Restated, privileged information shouldn't be discoverable.
- That means the OS needs a **different call and return model** for system calls than we have for traditional functions.

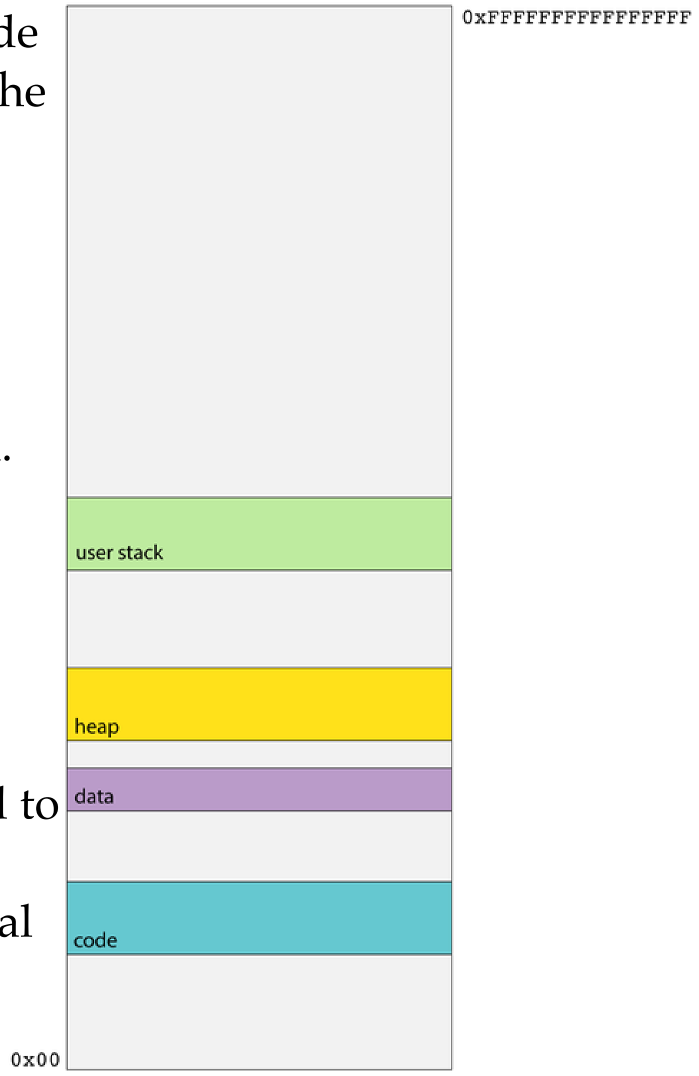
Lecture 05: Understanding System Calls

- 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, 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.
- That means the OS needs a **different call and return model** for system calls than we have for traditional functions.

0xFFFFFFFFFFFFFFFF
0x00

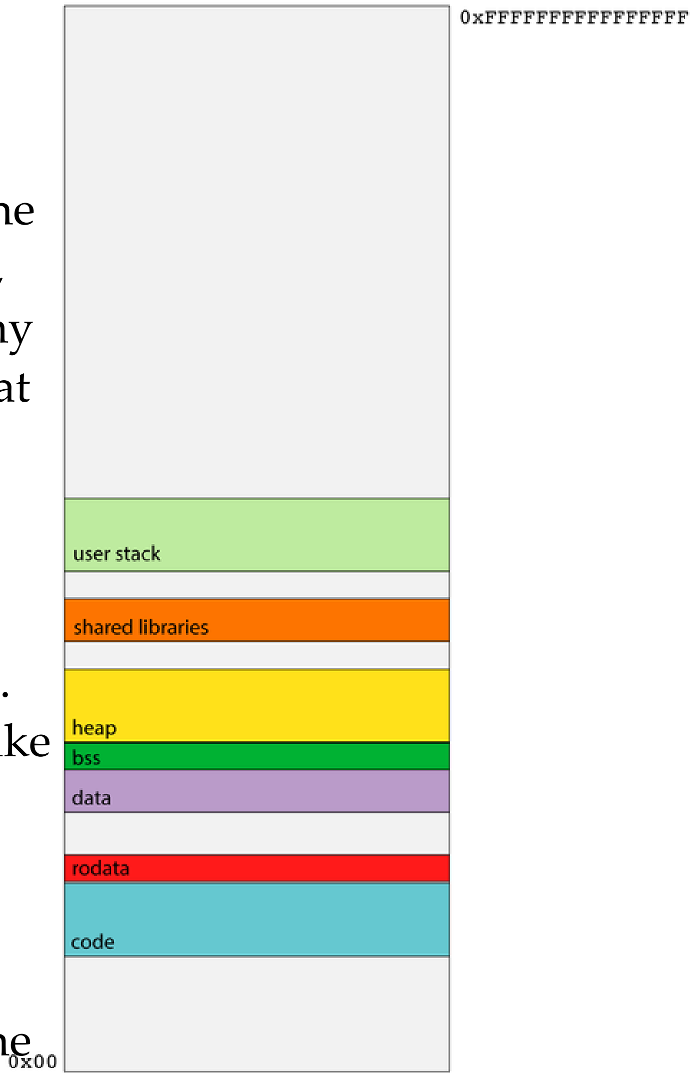
Lecture 05: Understanding System Calls

- Recall 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 typically sits above 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.



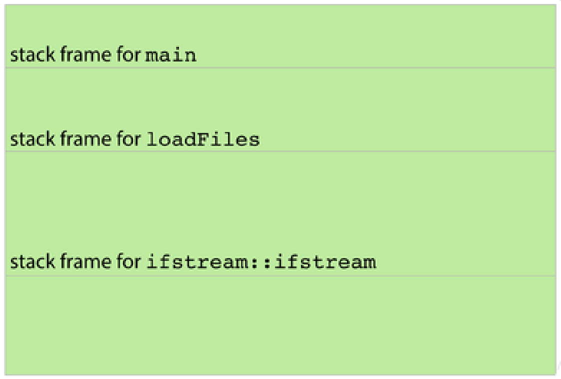
Lecture 05: Understanding System Calls

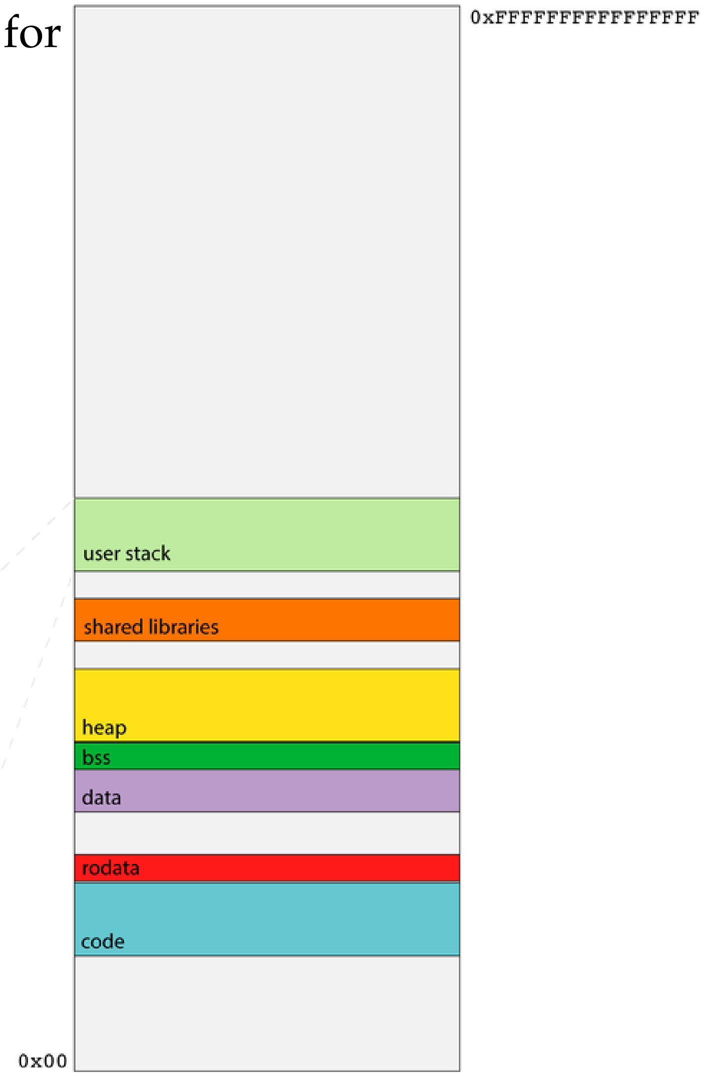
- There are other relevant segments that haven't been called out in prior classes, or at least not in CS107.
- The **rodata** segment also stores global variables, but only those which are immutable—i.e. constants. As the runtime isn't supposed to change anything read-only, the segment housing constants can be protected so any attempts to modify it are blocked by the OS. Note that the rodata segment sits directly on top of the code segment, which is also (not surprisingly) read-only.
- 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 on behalf of all processes.



Lecture 05: Understanding System Calls

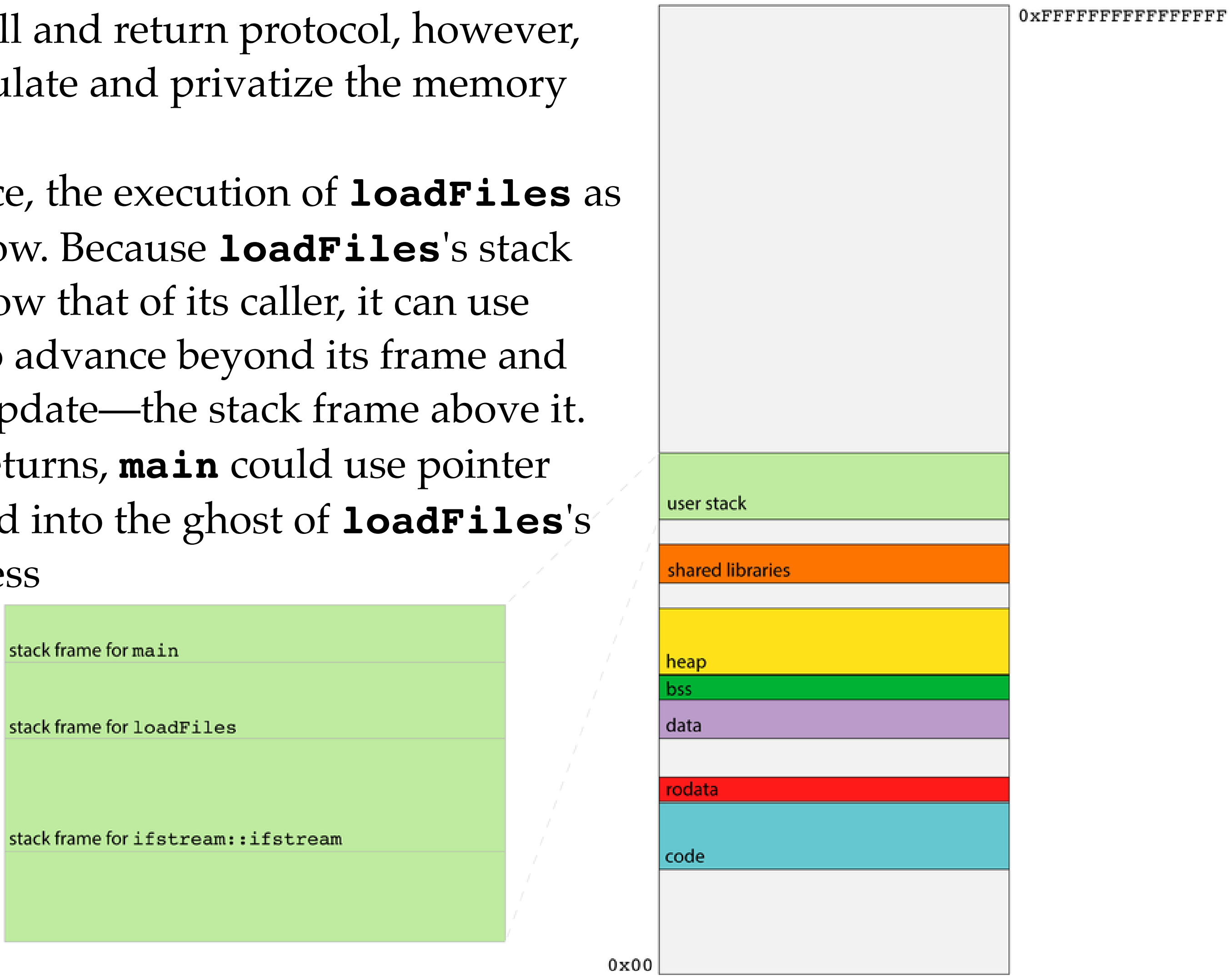
- The user stack maintains a collection of stack frames for the trail of currently executing user functions.
- 64-bit process runtimes rely on **`%rsp`** to track the address boundary between the in-use portion of the user stack and the portion that's on deck to be used should the currently executing 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)





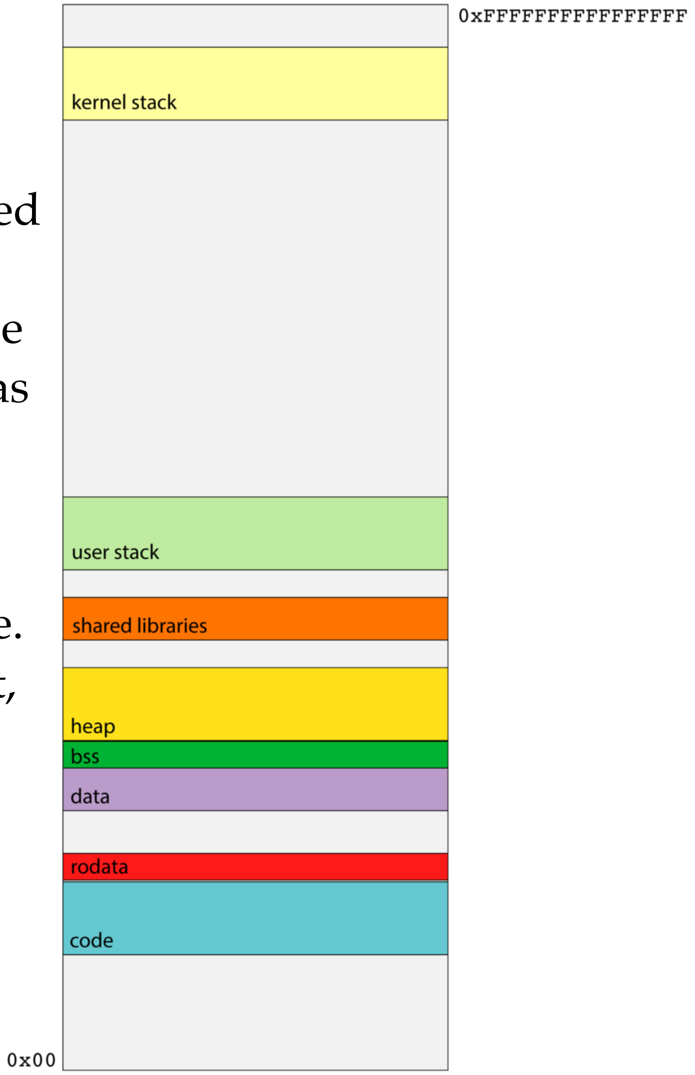
Lecture 05: Understanding System Calls

- 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.



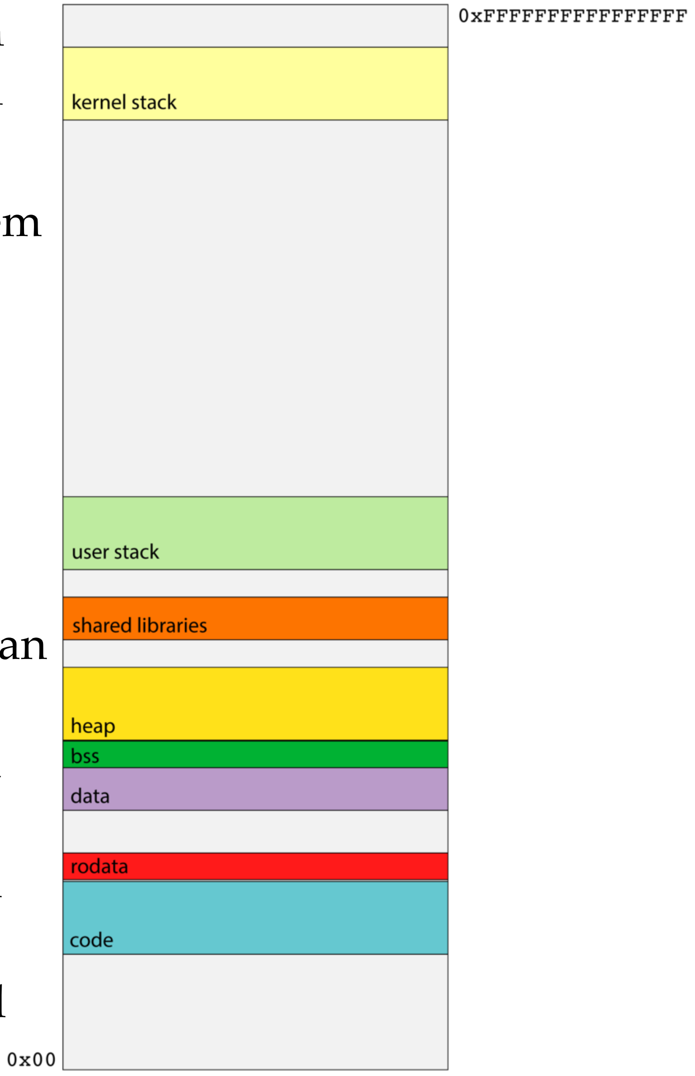
Lecture 05: Understanding System Calls

- System calls like **`open`** and **`close`** 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 to invoke a user function, but **`callq`** isn't the correct instruction for system calls.
- We need a different call and return model for system calls—one that doesn't rely on **`callq`**.



Lecture 05: Understanding System Calls

- Here's how function call and return works for system calls. All of this is done on your behalf whenever you call open, write, read, etc.
  - 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`**, and so forth).
  - The system call arguments—there are at most 6—are evaluated and placed in **`%rdi`**, **`%rsi`**, **`%rdx`**, **`%r10`** (not **`%rcx`**), **`%r8`**, and **`%r9`**.
  - The system issues a *software interrupt* (also 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 system call, places any return value in **`%rax`**, and then executes **`iretq`** to return from the handler, revert from superuser mode.
  - 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 **`%rax`** is then extracted by the caller as any return value would be.



Lecture 05: Creating and Coordinating Processes

- 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 as 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 05: Creating and Coordinating Processes

- **New system call: `fork`**
  - Here's a simple program that knows how to spawn new processes. It uses system calls named **`fork`**, **`getpid`**, and **`getppid`**. The full program can be viewed right here.

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

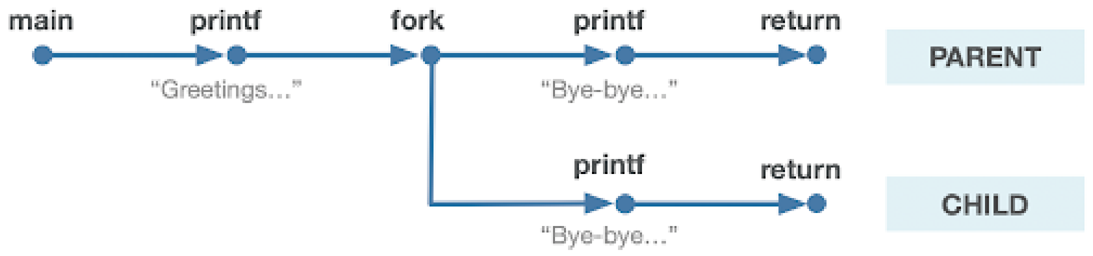
  - Here's the output of two consecutive runs of the above program.

```
myth60$ ./basic-fork
Greetings from process 29686! (parent 29351)
Bye-bye from process 29686! (parent 29351)
Bye-bye from process 29687! (parent 29686)
myth60$ ./basic-fork
Greetings from process 29688! (parent 29351)
Bye-bye from process 29688! (parent 29351)
Bye-bye from process 29689! (parent 29688)
```

Lecture 05: Creating and Coordinating Processes

- **`fork`** is called once, but it returns twice.
  - **`getpid`** and **`getppid`** return the process id of the caller and the process id of the caller's parent, respectively.
  - 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.

Illustration courtesy of Roz Cyrus.



  - **`fork`** knows how to clone the calling process, synthesize a nearly identical copy of it, and schedule the copy to run as if it's been running all along.
    - Think of it as a form of process mitosis, where one process becomes twins.
    - All segments (data, bss, init, stack, heap, text) are faithfully replicated to form an independent, protected virtual address space.
    - All open descriptors are replicated, and these copies are donated to the clone.

Lecture 05: Creating and Coordinating Processes

- Here's why the program output makes sense:
  - Process ids are generally assigned consecutively. That's why 29686 and 29687 are relevant to the first run, and why 29688 and 29689 are relevant to the second.
  - The 29351 is the pid of the terminal itself, and you can see that the initial **`basic-fork`** processes—with pids of 29686 and 29688—are direct child processes of the terminal. The output tells us so.
  - The clones of the originals are assigned pids of 29687 and 29689, and the output is clear about the parent-child relationship between 29686 and 29687, and then again between 29688 and 29689.
- Differences between parent calling **`fork`** and child generated by it:
  - The most obvious difference is that each gets its own 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. 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 to easily self-identify as the child.
    - 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 05: Creating and Coordinating Processes

- Reasonable question: Why does the child process insert its text into the same terminal that the parent does? And how?
  - Answer: The parent process' file descriptor table is **cloned** on **fork** and the reference counts within the relevant open file table entries are promoted.

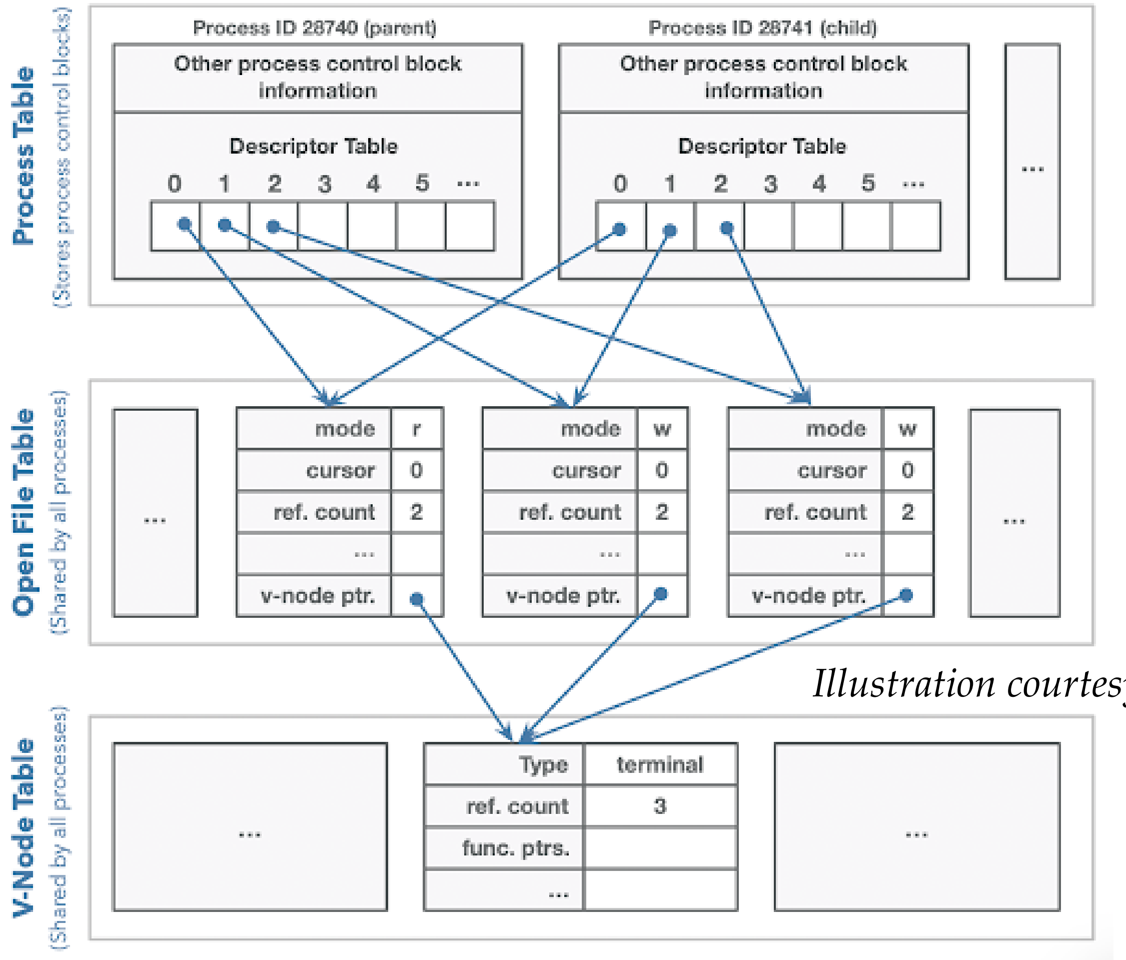


Illustration courtesy of Roz Cyrus.

Lecture 05: Creating and Coordinating Processes

- Aside: 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 all **`fork`**'d processes, though it pauses each at **`fork`**.
    - **`info inferiors`**
        - This lists the all of the processes that **`gdb`** has captured.
    - **`inferior X`**
        - Switch to a different process.
    - **`detach inferior X`**
        - Tell **`gdb`** to stop watching the process before continuing it
    - You can see an entire debugging session on the **`basic-fork`** program right here.

Lecture 05: Creating and Coordinating Processes

- **`fork` so far:**
  - **`fork`** is a system call that creates a near duplicate of the current process.
  - In the parent process, the return value of **`fork`** is the child's **`pid`**, and in the child, the return value is 0. This enables both the parent and the child to self-identify which of the two processes they are.
  - **All** 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 next time) 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. More on that in discussion section.

Lecture 05: Creating and Coordinating Processes

- 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.

```c
static const char const *kTrail = "abcd";
int main(int argc, char *argv[]) {
  size_t trailLength = strlen(kTrail);
  for (size_t i = 0; i < trailLength; i++) {
    printf("%c\n", kTrail[i]);
    pid_t pid = fork();
  }
  return 0;
}
```

Lecture 05: Creating and Coordinating Processes

- Second example: A tree of **fork** calls
  - Three samples runs are shown on the right.
  - Reasonably obvious: A single **a** is printed by the soon-to-be-great-great-granddaddy 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.
  - 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 are there prompts in the middle of the output of the second and third runs?

```
myth59$ ./fork-puzzle
a
b
b
c
c
c
d
d
c
d
d
d
d
d
d
myth59$
```

```
myth59$ ./fork-puzzle
a
b
c
b
d
c
c
d
c
d
d
myth59$ d
d
d
d
```

```
myth59$ ./fork-puzzle
a
b
c
b
d
c
c
d
d
c
d
d
d
myth59$ d
d
```

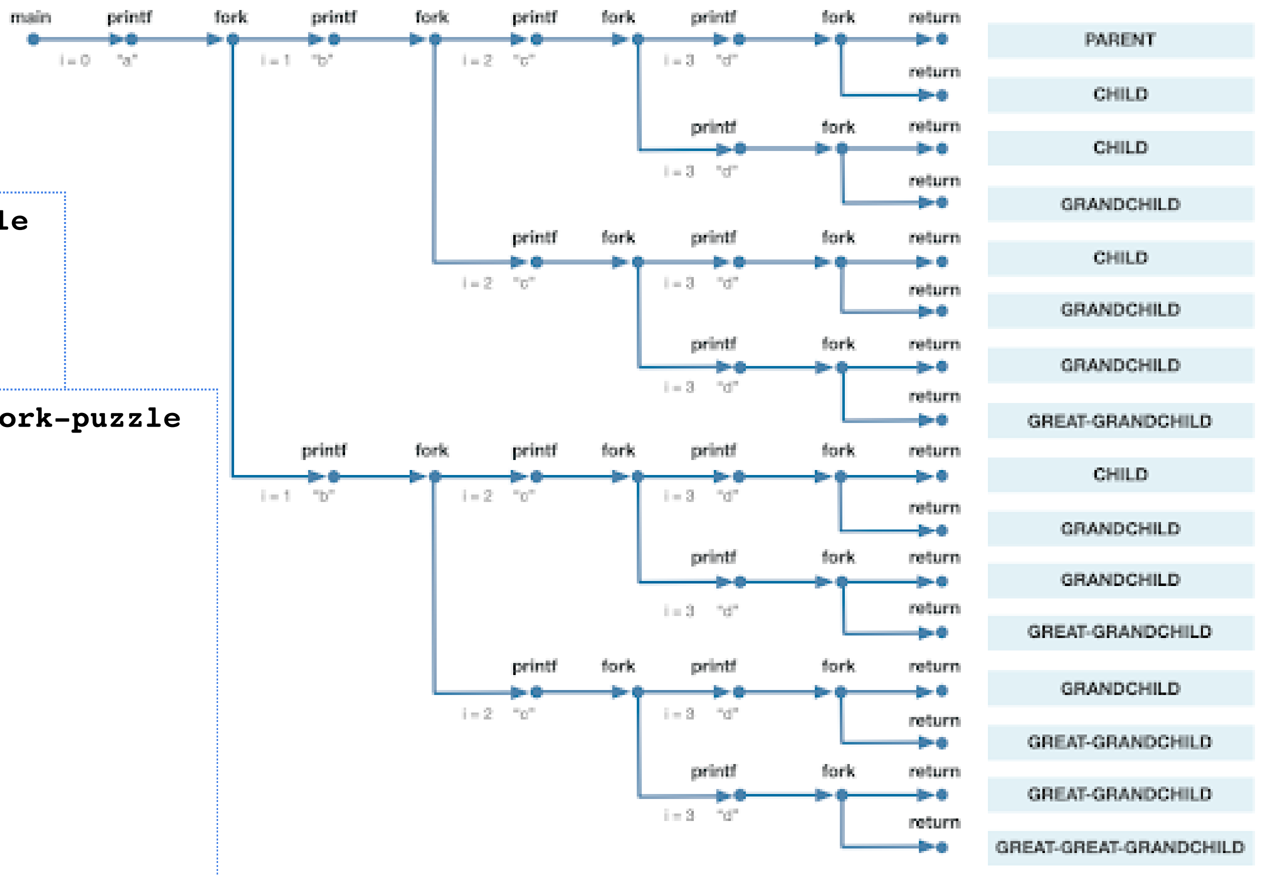
Lecture 05: Creating and Coordinating Processes

- Second example: A tree of **fork** calls
  - Kudos to Roz Cyrus for constructing this elaborate **fork** graph!

```
myth59$ ./fork-puzzle
a
b
b
c
c
c
d
d
c
d
d
d
d
d
d
myth59$
```

```
myth59$ ./fork-puzzle
a
b
c
b
d
c
c
d
c
d
d
myth59$
d
d
d
```

```
myth59$ ./fork-puzzle
a
b
c
b
d
c
c
d
d
c
d
d
d
myth59$ d
d
```



Lecture 05: Creating and Coordinating Processes

- Third example: Synchronizing between parent and child using **waitpid**
  - Notice in the fork puzzle example that it was possible for the d's to be printed after the prompt. This is because the parent finishes before the child, but the child still has access to stdout and continues printing its data.
  - Synchronization between parent and child can be managed using yet another system call called **waitpid**. It can be used to temporarily block a process until that process terminates or stops.

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

  - The first argument specifies the *wait set*, which for now is just the pid of the child process that needs to complete before **waitpid** can return.
  - The second argument supplies the address of an integer where process termination information can be placed (or we can pass **NULL** if we don't need the information).
  - The third argument is a collection of bitwise-or'ed flags we'll study later. For the moment, we'll just go with 0 as the required parameter value, which means that **waitpid** should only return when the process with the given pid exits.
  - The return value is the pid of the process that successfully exited, or -1 if **waitpid** fails (perhaps because the pid is invalid, or you passed in other bogus arguments).

Lecture 05: Creating and Coordinating Processes

- 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).
  - The parent process correctly waits for the child to complete using **waitpid**.

```
int main(int argc, char *argv[]) {
  printf("Before.\n");
  pid_t pid = fork();
  printf("After.\n");
  if (pid == 0) {
    printf("I am the child, parent will wait for me.\n");
    return 110;
  } 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;
  }
}
```

- 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 (110 as expected).
- The **waitpid** call also donates child process-oriented resources back to the system.

Lecture 05: Creating and Coordinating Processes

- The output on the left below is most likely every single time the program is executed, since the parent generally continues running without halting when it calls **fork** (since all fork does is set up new data structures for a new process, returns, and then carries on).
- However, it is theoretically possible to get the output on the right if the child runs as soon as it comes into existence.

```
myth59$ ./separate
Before.
After.
After.
I am the child, parent will wait for me.
Child exited with status 110.
myth59$
```

```
myth59$ ./separate
Before.
After.
I am the child, parent will wait for me.
After.
Child exited with status 110.
myth59$
```

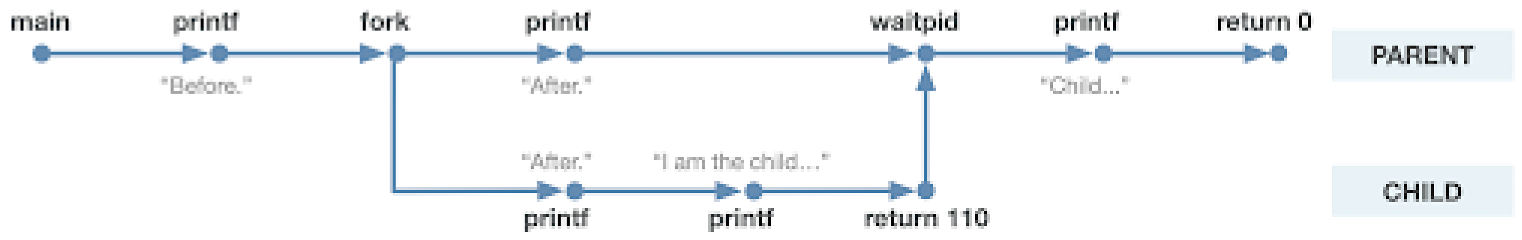


Illustration courtesy of Roz Cyrus.