Week 2 - Friday

COMP 3400

Last time

- What did we talk about last time?
- Process memory
- Multiprogramming

Questions?

Assignment 1

Assignment 2

Kernel

Kernel

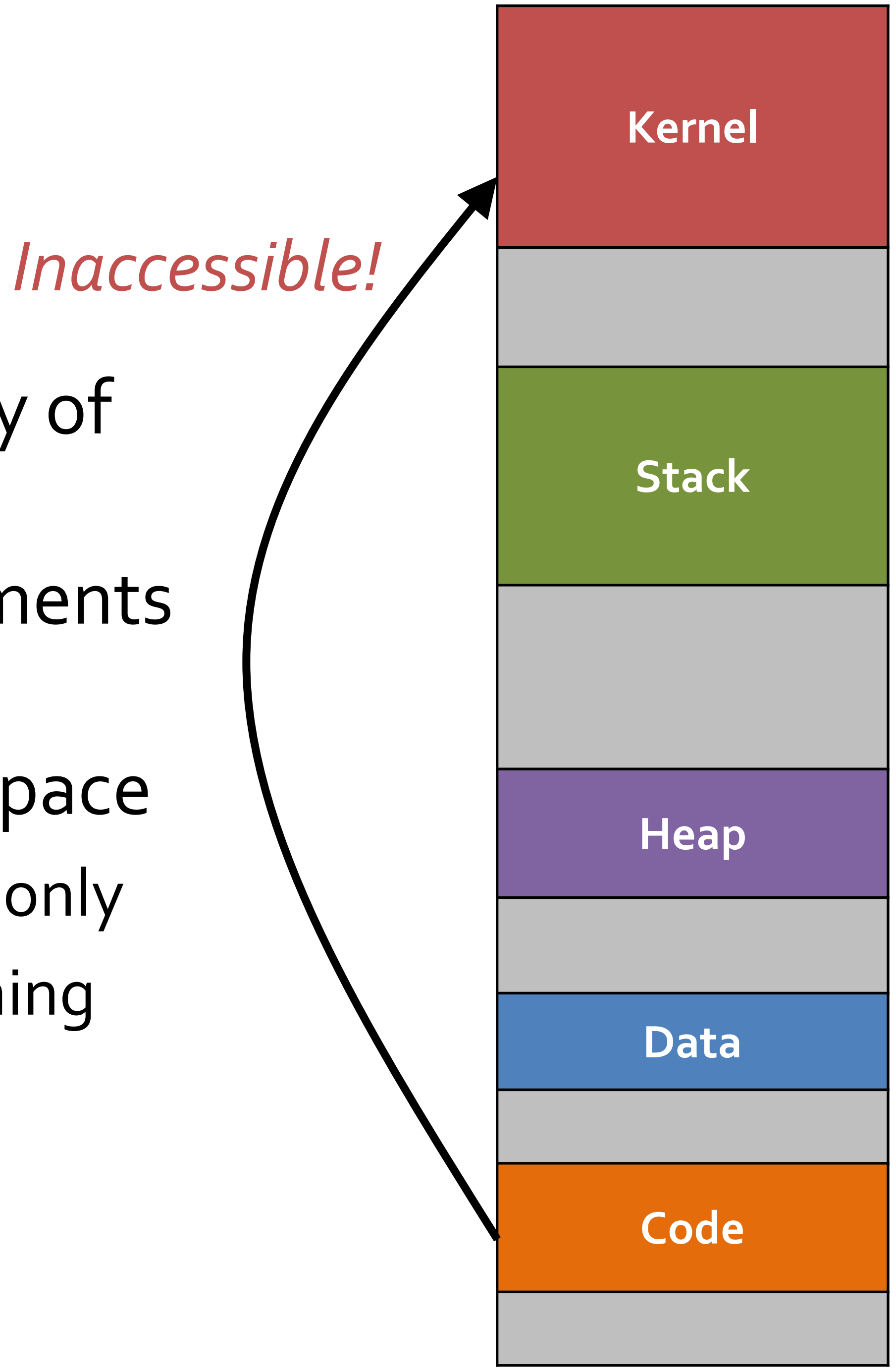
- The kernel runs with full access privileges to everything
- The kernel controls:
  - Physical memory
  - File system
  - I/O devices
- It handles power disruption and people attaching USB devices
- Jobs of the kernel
  - Resource manager: Giving access to hardware when needed
  - Control program: Handling errors and access violations
- Because it has to work consistently, the kernel doesn't change much over the years

x86 operating mode

- The **current privilege level (CPL)** is a 2-bit value set in x86 CPUs
  - Also called a **ring**
  - Ring 3 is user mode
  - Ring 0 is kernel mode
  - The other two rings aren't used
- When in kernel mode:
  - All memory addresses can be accessed
  - Some special CPU instructions like halting the CPU or invalidating the cache can be executed
  - Some normal CPU instructions work differently

Kernel memory structure

- Kernel memory exists in the virtual memory of every process
- The kernel has all the normal memory segments but also a stack for every process
- User mode code cannot access the kernel space
  - Bits are set in the CPU marking space as kernel-only
  - Otherwise, malicious code could access everything
  - And badly written code could do crazy stuff



Booting

- The kernel is loaded during the **boot sequence**
- CPU executes firmware stored in non-volatile storage
  - Older BIOS system
  - Or newer UEFI system
- Firmware finds a boot loader, linked to by a special part of a hard drive or SSD or similar
  - GRUB is a common Linux bootloader
  - BOOTMGR is for Windows
  - BootX is macOS
  - Some boot loaders allow **dual-booting**, the ability to choose which OS to start
- The boot loader finds the file with the kernel in it and calls its **`main()`** function
- The kernel takes over and does everything else

Kernel invocation

- The kernel can be invoked in two different ways
- System call:
  - A user mode program wants to do something (like open a file) that requires OS involvement
  - Somewhere in the library, a special trap instruction will ask the kernel to do something
- Interrupt or exception:
  - Interrupts are hardware events that cause the kernel to react, like clicking a mouse
  - Exceptions are software events that notify the kernel of a problem, like a segmentation fault
  - This kind of exception isn't the same as an exception in Java, although the Java exception can be triggered by an OS exception

Mode switches

- A **mode switch** is when the ring changes from user mode to kernel mode
- The user-mode process has no idea this is happening
- After each instruction executes, there's a chance that a mode switch happened, causing the kernel to handle an interrupt
- One of the challenges of writing OS code is that parts of it have to be written in a way that doesn't cause exceptions

System Calls

System calls

- User-mode processes can do normal CPU operations
  - Add, subtract, multiply, divide
  - Test for equality
- They can't do anything outside the CPU on their own
  - Read or write hard drive data
  - Send messages over the network
- To do these things, processes make **system calls**, asking the kernel to do the operation

How system calls work

- In assembly, a special trap instruction triggers a mode switch so that the kernel will start doing stuff
  - The x86 trap instruction is **`syscall`**
- The kernel checks to make sure that the process has all the necessary privileges to do the operation first
- After the system call, the kernel runs the **`sysret`** instruction, returning to user mode
- Many system calls are referred to by the C functions that are called to run them, even though those functions just do set up before running the real system call
  - For example: **`write()`**

Organization of system calls

- A given OS has a fixed number of system calls
- You can't just add or remove them willy-nilly
- In Linux, each one has a number as well as a name
  - The number is what matters, but the name makes it easier to talk about
- C functions that wrap system calls are the same as the system calls without **`sys_`** in front
  - C function **`write()`** wraps the **`sys_write()`** system call
  - Because C is a low-level, systems language, a lot of standard library functions directly wrap systems calls
- A lot of other functions provide more features but eventually end up calling system calls
  - **`printf()`** has all kinds of formatting options, but it ultimately calls **`write()`**

Common system calls

- The 64-bit Linux kernel has more than 300 system calls
- These are just a few common ones:

System Call	Number	Purpose
`read`	0	Read from a file descriptor
`write`	1	Write to a file descriptor
`nanosleep`	35	High-resolution sleep (units in seconds and nanoseconds)
`exit`	60	Terminate the current process
`kill`	62	Send a signal to a process
`uname`	63	Get information about the current kernel
`gettimeofday`	96	Get the system time in seconds since midnight, January 1, 1970
`sysinfo`	99	Get information about memory usage and CPU load average
`ptrace`	101	Trace another process's execution

Using `syscall()`

- You can call a specific system call using the **`syscall()`** function
- Its first parameter is the system call number, and the others depend on the system call
- For example, a basic Hello, World program can call **`syscall()`** with arguments:
  - **`1`** System call number for **`write()`**
  - **`1`** File descriptor for **`stdout`**
  - **`message`** Pointer to **`"Hello, world\n"`**
  - **`13`** Number of bytes to write

Hello, world with system calls in C

```
#include <unistd.h>

char *message = "Hello, world\n";

int
main (void)
{
   syscall (1, 1, message, 13);  // Write message
   syscall (60, 0);              // Exit process

  return 0; // Unreachable
}
```

Process Life Cycle

Creating processes

- The lives of processes can be modeled with a state diagram, as in Assignment 2
  - A process goes into different states depending on events
- Rough outline:
  - When a process is created, there's a new virtual memory instance
  - Process code is executed until the **`halt`** instruction is reached
  - Process is destroyed and resources it was using are released by the kernel
- All processes have a parent process (except for the **`init`** process)

Creating processes in code

- Processes are, of course, created when you run a program from the command line
- However, you can also create processes from within a program, using calls to special functions
- The **`fork()`** function creates a new process that's exactly the same as the current process
- The **`exec()`** function allows you to replace the current process with another program
- Each process has a unique ID, its process ID or PID
  - **`getpid()`** returns the PID of the current process
  - **`getppid()`** returns the PID of the current process's parent process

Using fork()

- The **fork()** function is pretty crazy!
  - When you call it, the process you're inside of keeps running
  - And another process spawns at exactly the same point in code
  - Both processes have *exactly* the same memory layout
  - The only difference is that **fork()** returns the child PID for the original process and 0 if you're the process that just got forked

```
pid_t child_pid = fork ();

if (child_pid < 0)
  printf ("ERROR: No child process created\n");
else if (child_pid == 0)
  printf ("Hi, I'm the child!\n");
else
  printf ("Parent just gave birth to child %d\n", child_pid);
```

Upcoming

Next time…

- Finish process lifecycle
- Files

Reminders

- **Finish Assignment 1**
  - Due tonight by midnight!
- Start on Assignment 2
- Look over Project 1
- Read section 2.6