CS 326: Operating Systems

System Calls

Lecture 7

Virtualizing the CPU

- The operating system virtualizes the CPU to allow multiple programs to run concurrently
  - …or at least with the illusion of concurrency
- How does this work at the OS level?
- Let's have a quick thought experiment: how would we implement this ourselves, right now?
  - We actually have a pretty good point of reference for this…

Full Virtualization

- Trap and translate all instructions to make the process believe it's running on some other platform
  - Now we can monitor **everything**
    - If I see `printf("Matthew = terrible")`, I just block it. Perfect!
  - Inefficient! Roughly doubles the amount of instructions needed
- This is basically building an emulator…
  - Performance is going to take a big hit
  - What are the benefits?

Direct Execution

- The opposite of full virtualization is direct execution
- This allows the process to run directly on the hardware
  - While doing so, it has complete control
- This is the old DOS/Win 3.1 approach
- Unfortunately, direct execution has some downsides…

Issues with Direct Execution

- If you give a process full control of the hardware, it can do whatever it wants
- Access other processes' memory, for instance
- Manipulate I/O devices directly
  - Who cares about disk permissions when you can just make the device read the files you want?
- So while the performance of direct execution is great, it poses security and reliability risks

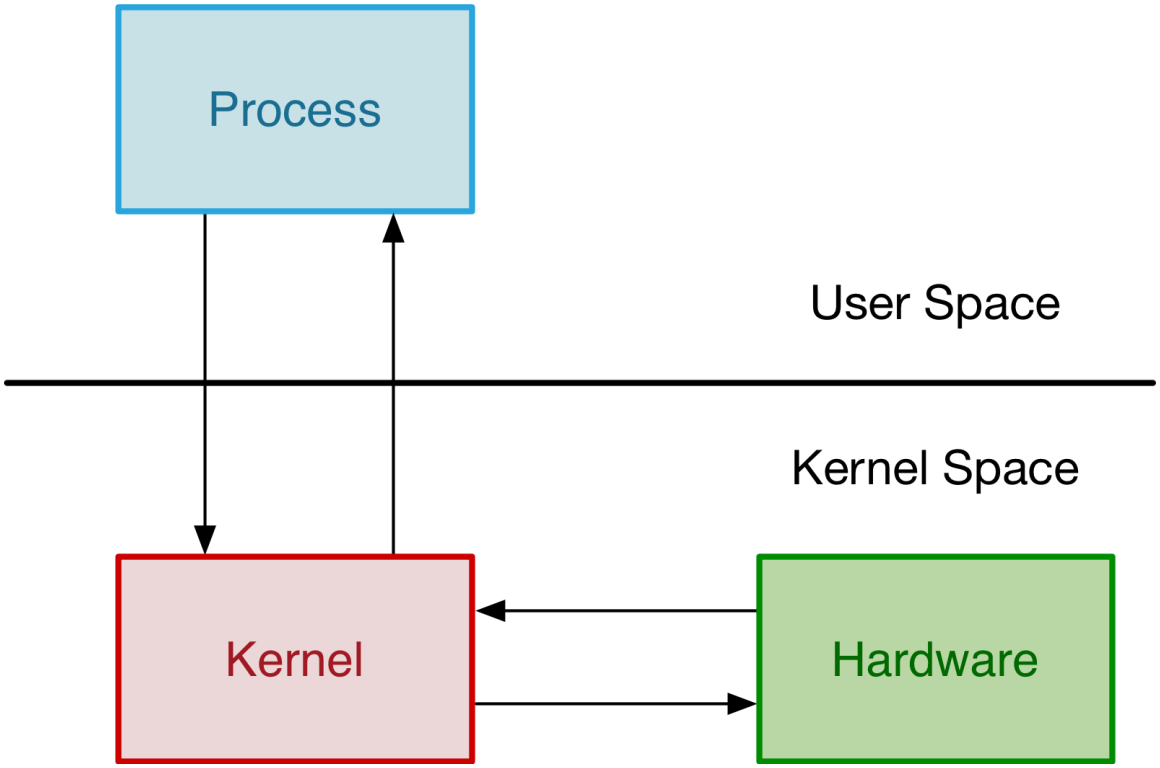
Limited Direct Execution

- OS designers came to a compromise between full virtualization and direct execution
  - **Limited** direct execution
- For certain (**safe**) operations, processes are given full access to the CPU/hardware!
- Some **privileged** operations are not allowed, however
  - In this case, the process must ask the kernel for help

System Calls

- These privileged operations are **system calls**
- System calls include performing I/O, setting the current time, or launching other processes (fork!)
- This is where we derive the division between two halves of the OS:
  - User space
  - Kernel space

System Calls



Executing System Call

- When you call **fork**, you aren't directly calling into kernel space
- The C library is responsible for interfacing with the kernel
  - Sets up parameters and return address for the call
  - Jumps into a predefined memory address in kernel space
- In doing so, execution is transferred to the kernel

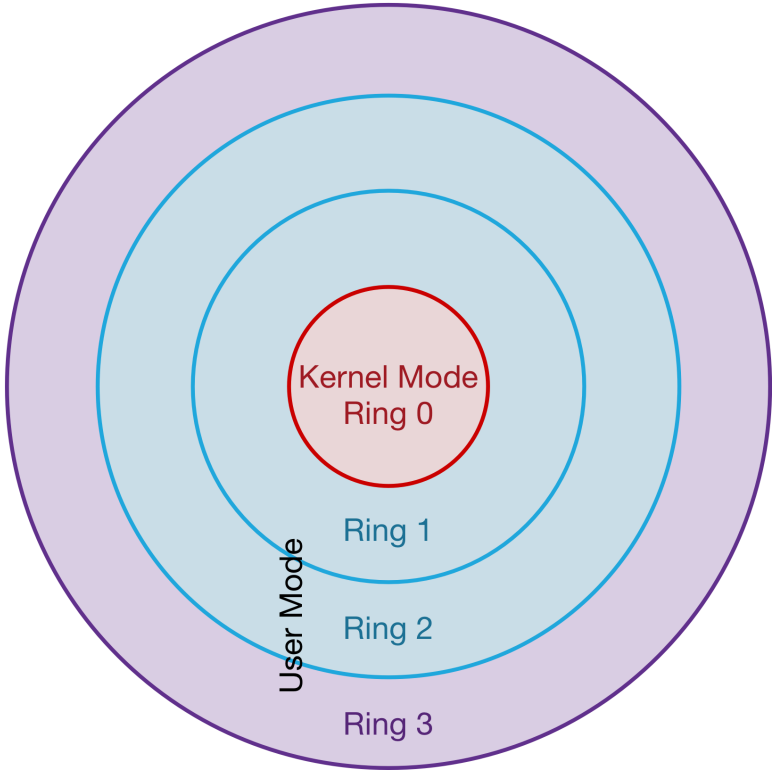
Protecting System Calls

- In the early days, operating systems allowed processes to call OS routines directly
  - This approach has security issues!
- **CPU Protection Rings** enable a separation between system-level functions and user space applications
  - Ring 0: highest level of privilege. OS functionality
  - Rings 1-2: drivers, VM Hypervisors, or simply unused
  - Ring 3: user applications

Privileged Instructions

- Instructions are flagged with a permission level
- Some of these instructions can only be run in ring 0
- Things you can't do in **user mode**, a.k.a. ring 3:
  - Change the protection ring (probably a good idea!!)
  - Modify the page table
    - Mapping between virtual and physical memory addresses (more on this later in the semester!)
  - Change interrupt handlers
  - Read/write model-specific registers

x86 Protection Rings



RISC-V Privilege Levels

- Machine
- Supervisor
- User
- Let’s take a look at `start.c` to see this transition…

Transferring Control

- When the jump occurs between user and kernel space, we must adjust the privilege level to transition successfully
- This is called a **trap**, which moves the execution pointer and changes the privilege level
- Once the the operation is complete, a **return-from-trap** is executed
  - Privilege level is dropped, and control is returned to process routines

Securing Kernel Space

- During boot, the OS has exclusive control of the hardware and sets up the **trap table**
- This defines protected regions that may be used for programs and hardware to interface with kernel space
- Examples:
  - Where to jump to execute a system call
  - How to handle hardware **interrupts**

Interrupts

- An **interrupt** is a signal to the processor that some type of event needs to be handled
- Software interrupt: **trap**
- Hardware interrupts inform the OS of a key being pressed, I/O completing, timer ticks, etc.
- Much better than polling: actively checking the state of all the devices continuously
- What happens when too many interrupts are produced?

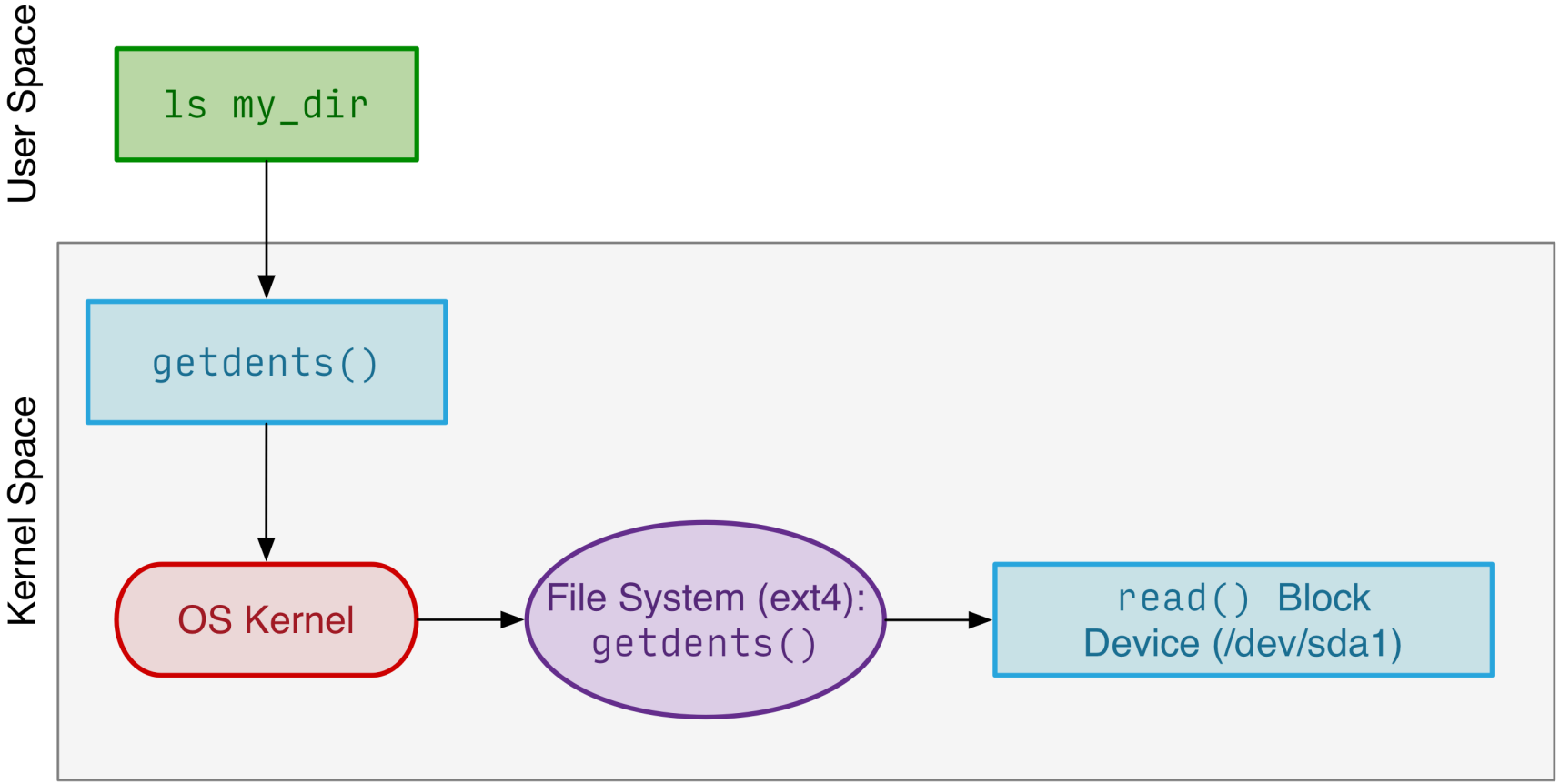
Security

- Protection rings are one line of defense between malicious programs and the kernel
- However, we still do not use **full virtualization**
  - We don’t inspect every operation before it runs
  - This would cause a noticeable decrease in performance
- This does open up the OS to vulnerabilities in both the software and hardware

Overhead

- Using the kernel as an intermediary does have downsides
- Still slower than direct execution
- This cost is called **overhead**, the amount of extra time spent in kernel space
  - Many privileged operations will be executed twice, once in each context

System Call Workflow: Linux ls



Tracing System Calls

- On Linux, you can use `strace` to monitor system calls as processes run
- `strace ls`
  - Prints each system call in the order they are executed
  - Memory allocation, opening files, etc
- Helpful: filtering
  - `strace -e trace=file ls`
  - (only prints system calls that deal with files)

Is it actually a syscall?

- Sometimes the POSIX API maps directly to underlying system calls
  - So you'll call a C library function named X, which then makes a system call X
- A good example: `stat()`
  - Gets information about files
  - See: `man 2 stat` vs `man 3 stat`

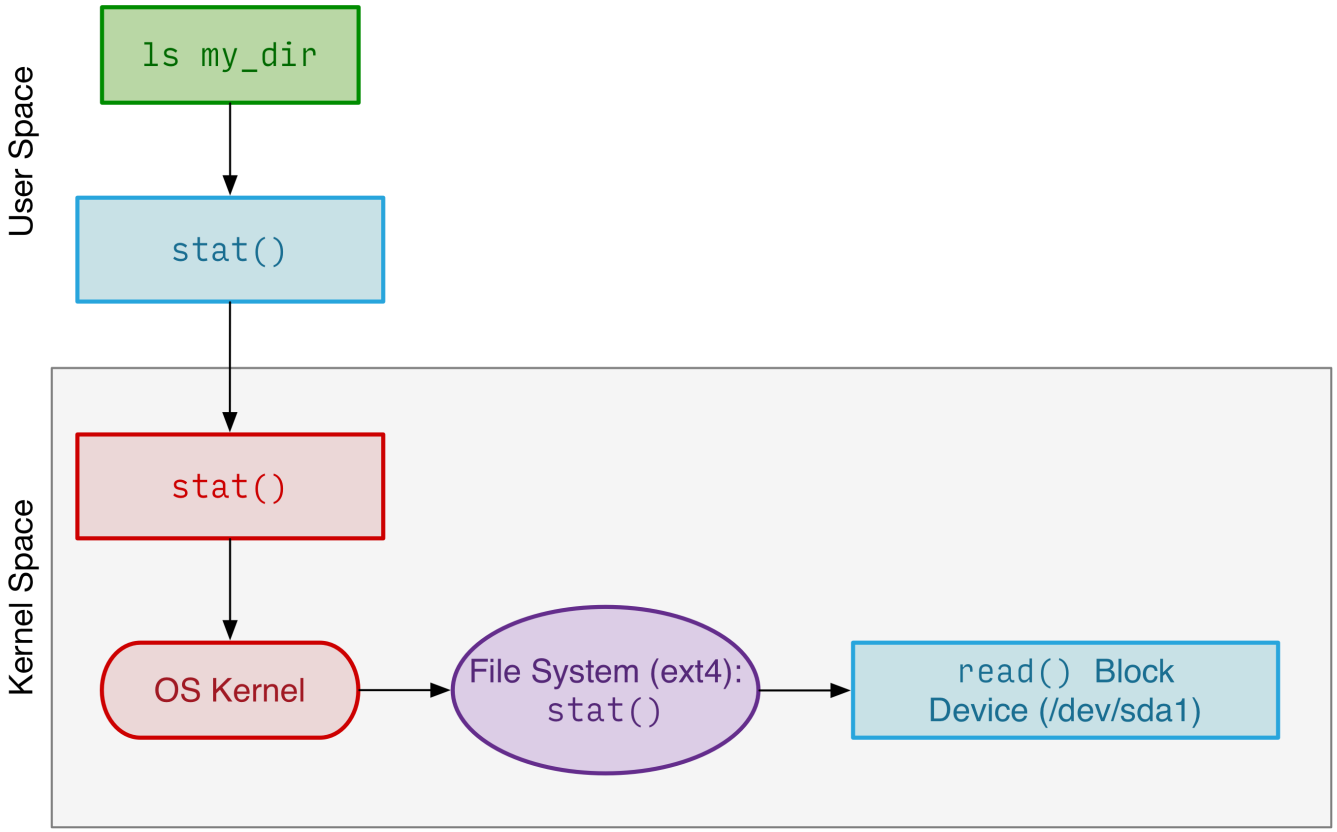
Layers of Abstraction

- On Linux, `readdir` is implemented via `getdents()`
- OS can conform to POSIX spec with different underlying syscall implementations
- `fork()` and `exec()` aren't the end of the Linux process saga… we can go deeper: `clone()`
  - In fact, `fork()` is implemented with the `clone()` system call!

clone

- clone is more flexible than fork: it lets us decide what the child process shares with the parent
- So we can decide whether they share the same heap, open files, etc.
- Here's a cool one: **CLONE_NEWPID**
  - This creates a new process **namespace** with the child process being assigned PID 1
    - And what is PID 1 again, class? Hmm?
    - It's almost like this new process is **contained** in this namespace…

Tracing stat on Linux



Overhead

- All these function calls will definitely add overhead
- However, this overhead is seen as a worthy trade-off: without it we'd have:
  - Processes running amok
    (crashing our system, probably)
  - Security issues
  - A much more brittle API for creating our programs

Demo: Tracing readdir