CS 521: Systems Programming

System Calls and Processes

Lecture 14

Today's Schedule

- System Calls
- Processes

Today's Schedule

- **System Calls**
- Processes

Virtualizing the CPU

- Our operating system (*Linux*, *macOS*, *Windows*, etc.), or **OS**, virtualizes the CPU to allow multiple programs to run concurrently
  - …or at least with the illusion of concurrency
  - even if we only have one physical CPU / core
- It switches between processes quickly to give them all a chance to use the hardware resources
- If we are the OS, how can we run other programs… within our program?

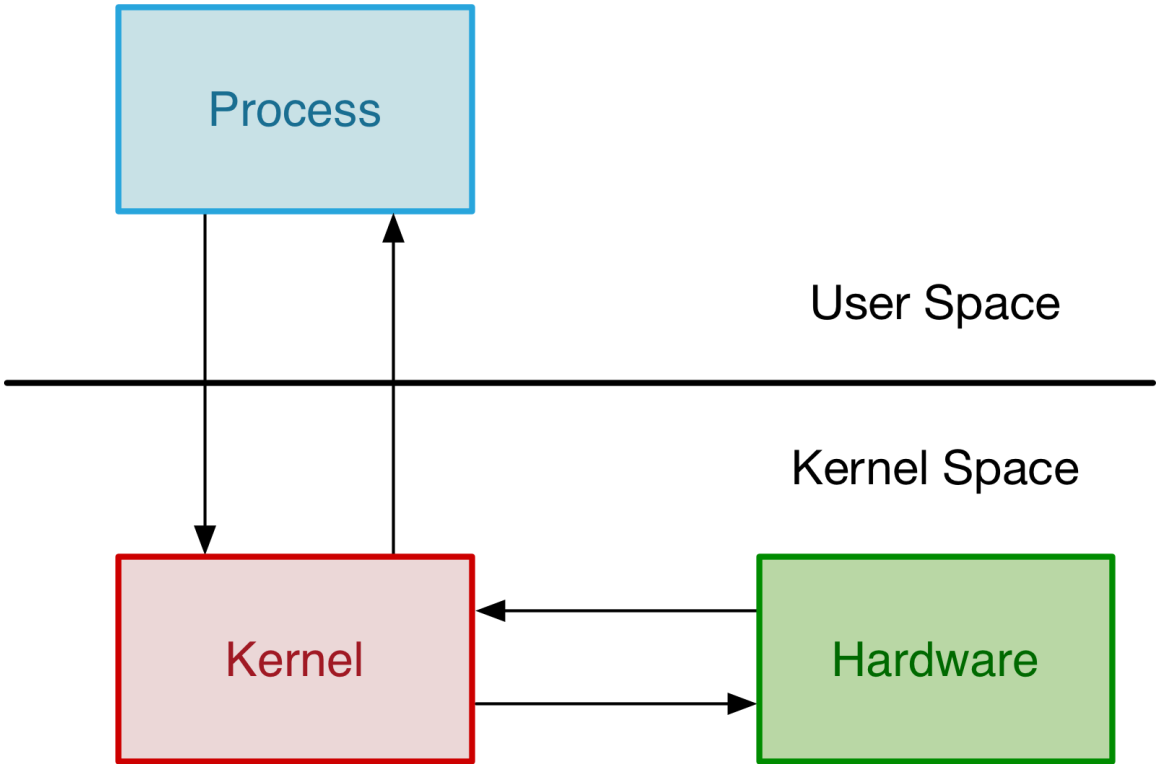
Execution Strategies

- To let a program run another program, we have a couple options:
  - Execute its instructions directly, giving it full control
  - Read the program's instructions, *interpret* them to make sure they're safe, then execute them
- OS designers came to a compromise between these two extremes
  - For certain (**safe**) operations, processes are given full access to the CPU/hardware!
  - Some **privileged** operations are not allowed

System Calls

- These privileged operations are **system calls**
- System calls include performing I/O, setting the current time, or launching other processes (fork!)
- Instructions (in your program binary) are flagged with a permission level
- This is where we derive the division between two halves of the OS:
  - User space
  - Kernel space (kernel = core of the OS)

System Calls



Overhead

- Using the kernel as an intermediary does have downsides
- Still slower than executing instructions directly
- 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

Portability

- When you're writing general-purpose C programs, it is recommended to avoid system calls (if possible)
  - (e.g., use `fread(3)` instead of `read(2)`)
- While many Unix-like OS implement a standard set of system calls (defined by POSIX) you can't assume they're available everywhere
  - Linux supports `clone`, `getdents`, and many others… but macOS does not.
  - Windows is not Unix-like, so it may not support any of the common system calls

Using System Calls

- You've already seen some system calls in Project 1!
  - `opendir`, `readdir`, `closedir`
  - The C standard does **NOT** assume that all systems will have the concept of a directory/folder hierarchy!
- Try compiling P1 on Windows, and you might be out of luck (unless you're using WSL, Cygwin, etc.)

Tracing System Calls

- You can install `strace` on your VM to monitor system calls as processes run (see `dtrace` on a mac)
- `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)

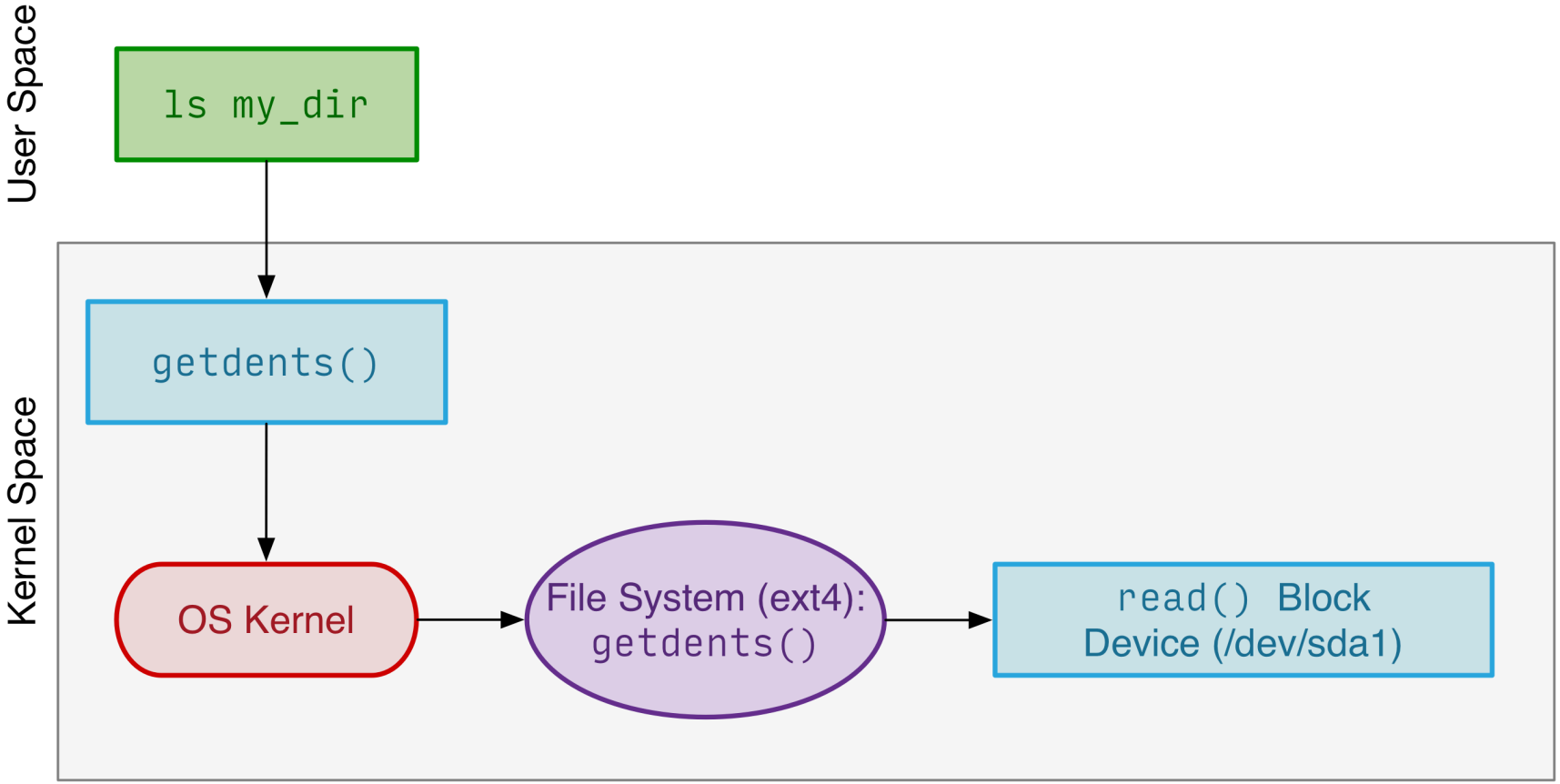
Identifying System Calls

- System calls will look exactly the same as regular C functions in your code
- So how do we know which is which?
- Usually the best way is the man pages!
  - Section 2 is system calls
  - Section 3 is the C library
  - `man 2 xyz` vs `man 3 xyz`

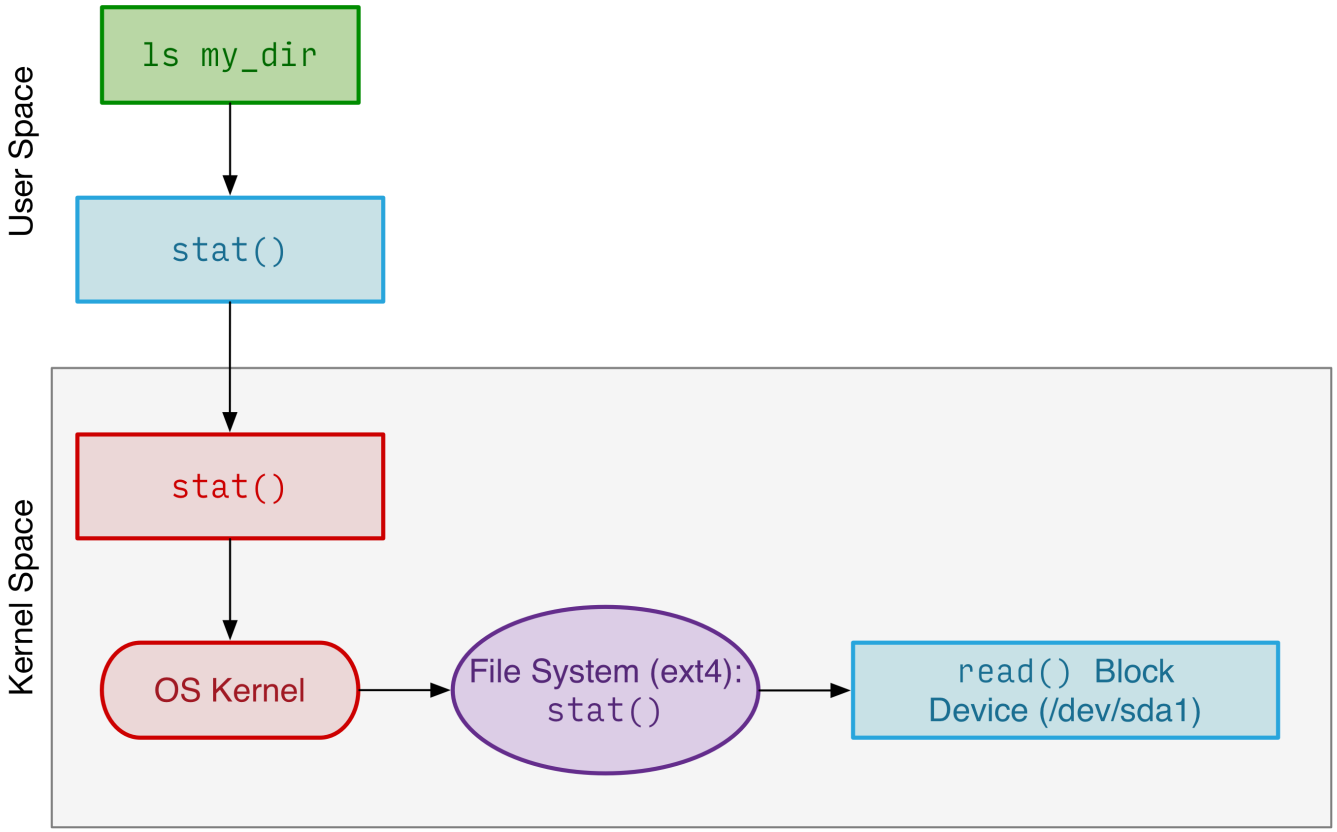
Is it actually a syscall?

- 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`
- On Linux, `readdir` is implemented via `getdents()`
  - One more layer of abstraction

System Call Workflow: ls



Tracing stat



Demo: Tracing readdir

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

Today's Schedule

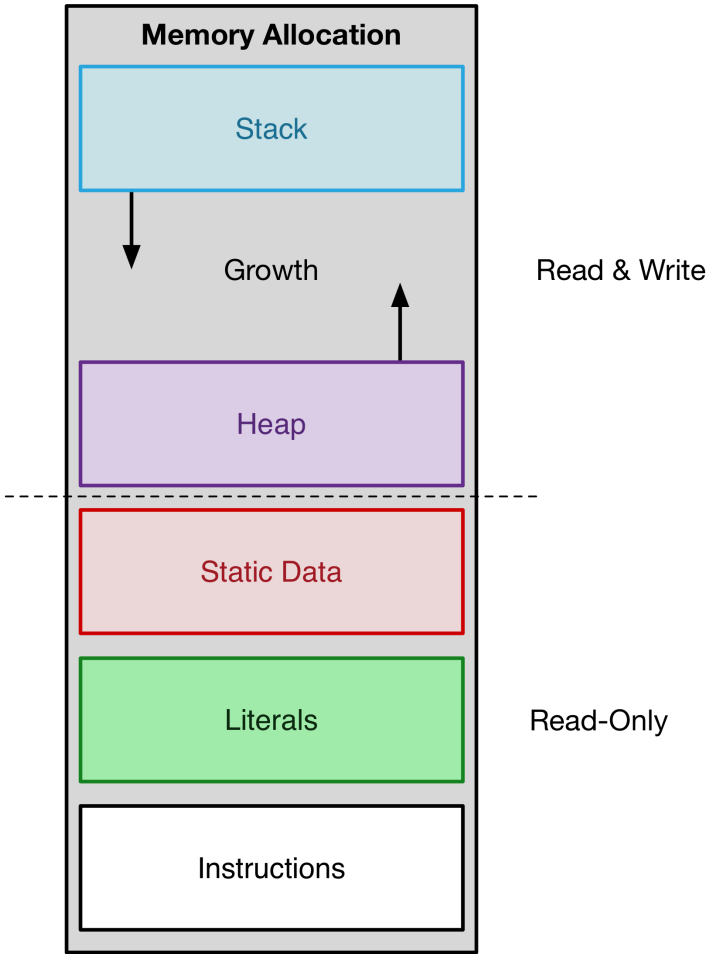
- System Calls
- **Processes**

From Program to Process

- When a program is executed, the operating system reads its static data from the disk and copies it into main memory
  - Program instructions, string literals, binary data
- A process ID (PID) is assigned
- Space is allocated for the **stack** and **heap**
- Streams are initialized
  - stdout, stderr, stdin
- Run-time permissions are applied

Process Memory Layout

- Processes are given a **zeroed out** virtual address space rather than accessing main memory directly
  - Prevents viewing/changing other process data
  - Makes memory allocation and management simpler
- Processes are still allowed to communicate, however, via inter-process communication (IPC) mechanisms



Inspecting the System

- Processes are limited to virtualized views of the hardware, but they are still able to inspect it
- Memory, CPU, disk availability and usage
- Other process names and command lines
- Logged in users
- Hardware specs, serial numbers, etc.
- This is usually **good**, especially in shared environments!

Demo: systemctl, ps, ...

Inspecting the System

```
[malensek@ruby:~]$ w
 23:11:06 up 1 day,  7:57, 11 users,  load average: 16.07, 15.17, 11.34
USER     TTY        LOGIN@   IDLE   JCPU   PCPU WHAT
mal      pts/0     07:16   14:37m  0.07s  0.07s -bash
zoe      pts/1     20:04    3:06m  0.43s  0.38s vim output/file-0
wash     pts/2     23:00    4:57   0.09s  0.05s vim Makefile
inara    pts/3     21:52    1:08m  0.82s  0.79s /usr/bin/python2
jayne    pts/4     23:10   12.00s  0.03s  0.03s -bash
malensek pts/5     23:11    0.00s  0.10s  0.04s w
```

Processes

- We also touched on **processes** before
- Processes are created with the **fork** function
- This creates a clone of an existing process
- After creating the clone, we know two things:
  - Which process is the parent
  - Which process is the child
- Logic branches from here, allowing the two processes to do different work

Dealing with Clones

- The cloning approach is particularly nice if you want to make your application work on multiple CPUs
- It doesn't quite help us if we want to launch a completely different process, though
- For instance, our program wants to start the `top` command
  - There is another function to accomplish this: `exec`

exec

- The `exec` family of functions allows us to launch other applications
- `exec` replaces the memory space of a clone with a new program and begins its execution
- After `fork()`: copy of my_program
- After `exec()`: separate process running `top`... or whatever you wanted to run!

Demo: fork + exec

Why Split fork + exec?

- Why not just have a nice C function called `launch_program` (or something like that) instead?
  - Or in other words: why does this need to be broken into two steps?
- Advantages of operating this way:
  - While the new process is still a clone, it can set up the target ***environment*** for the new application
  - No restriction on which process will be replaced (could be the parent or child… usually child)

Setting up the Environment

- The new process can inherit several aspects of its predecessor
  - try doing a `chdir` before executing the child
- Environment variables: the system path, current working directory, global program options
- Redirection: the new process may be set up to receive input on its stdin stream from the parent process
  - Pipes in the shell

Demo: env

Taking a Step Back

- Okay, so we've talked about system calls. But why should we care?
  - These details are not abstracted away from us like they are in Java, Python, etc.
  - System calls mean more overhead in our programs – if you can do something in user space, you'll get better performance
- And what about processes?
  - What we've covered today already gives us the basic building blocks for parallel programming