Wrap Up

W4118 Operating Systems I

columbia-os.github.io

Credits to Jae and Hans

Advanced UNIX Programming

Processes, threads, concurrency, signals, non-blocking & async I/O

`hw2-shell:`

implement a simple shell

Crossing to the Kernel: System Calls

Sometimes a process needs to perform privileged operations, e.g.:

- File I/O: open(), read(), write(), close(), etc.
- Memory management: Allocate/free memory, protection
- Process management: fork(), exec(), etc.

Can't trust (nor expect) userspace processes to do bookkeeping & access control.

OS needs to provide a well-defined interface to the kernel!

`hw3-tabletop:`

add a new system call to Linux and install custom kernel to test it

inspect a running process's file descriptor table

Synchronization

Many threads of execution can concurrently access shared memory.

Race conditions can lead to data corruption and unpredictable behavior.

Need OS support to provide mutual exclusion and synchronization!

`hw4-fridge:`

implement an in-kernel hashtable accessible via system calls

use synchronization primitives to ensure safe data structure access

Scheduling

System may have many processes/threads to execute, but fixed # of CPUs…

OS needs to virtualize the CPU! (i.e. provide illusion of infinite CPUs):

- multiplex process execution across multiple CPUs
- permit higher priority processes to run sooner/for longer

`hw5-freezer:`

add a new scheduling policy to the Linux scheduler

replace the default Linux scheduling policy

Memory Management

Processes execute within a **byte-addressable linear virtual address space**

Perks: pointers, arrays, stack grows “downwards”, heap grows “upwards”

How is this possible given fixed RAM size and variable # of running processes?

OS needs to virtualize physical memory (i.e. provide illusion of linear vaddr spaces)

- map virtual addresses to physical addresses on-the-fly
- protect virtual memory mappings from other processes and illegal access

`hw6-farfetchd:`

“hack” a process’s address space by writing directly to its physical memory

File Systems

File access is made straightforward by the file API (syscalls), but there are many implementation details hidden behind the kernel:

- read/write/execute permission enforcement, user access validation
- resolving path names and fetching corresponding data at offset from disk
- persisting metadata and data on disk, keeping metadata synchronized

The OS needs to implement the file API and ensure data persistence

`hw7-ezfs:`

implement a simple file system and hook it into Linux VFS

Stuff we skimmed/missed

Deadlock theory

I/O systems

Network file system (NFS)

Interrupt handlers and bottom half

Virtualization

Final

2.5-hour long

No cheat sheet.
You are allowed 2 blank pages, please turn them in with your name on it.

Will focus on the second part of the course:

- Scheduler
- Locking (like writing and using locks)
- Memory Management (Page faults, COW, virtual to physical translation etc)
- Filesystems (EZFS, Block Devices, Disk IO route)
- Networking Basics

However, you will need concepts from throughout the course, e.g., locks

Final Reminders

Fill out Courseworks evaluation (!!!)

Remember your pledge

- Don't share class materials with friends
- Don't post any class-related code to GitHub
- Don't post any class materials to Chegg, CourseHero, etc

If you enjoyed OS, Columbia is doing some cool research on this stuff.
Reach out → CU Profs are Jason Nieh, Asaf Cidon, Kostis Kaffee, ..