Curriculum for Operating Systems
CPS 310/CPS 510
Duke University

Instructors: Jeff Chase, Mike Hewner, Danyang Zhuo

This outline and schedule of topics is from Fall 2020. In that semester the introductory (CPS 310) and advanced (CPS 510) courses are synchronized and interleaved week-by-week. The courses are in a FLIP style with prerecorded lectures and synchronous meetings for discussion or structured activities. There are lab projects due each week. Labs labeled advanced are optional for CPS 310. CPS 510 students focus on an independent project with no labs beginning in Week 10.

Here is an outline of lecture topics and class activities for each week. The header for each week is a general theme of the week for both courses, though there is some topic drift to meet the demands of the lab schedule. We list CPS 310 topics in detail along with selected advanced (CPS 510) topics for each week.

Week 1: Introduction

- **Course intro**. Programs and platforms; platform abstractions. Operating systems as long-lived software platforms on changing hardware. Example: Unix and Linux. Foundational concepts and abstractions: process as executing program; virtual address space as sandbox and lockbox; thread as a stream of instructions. Code sharing in runtime libraries. The OS kernel as servant, enforcer, and guard. Software layering and nesting.
- **Welcome to the machine**. Hardware and software. Von Neumann architecture: CPU, registers, memory, I/O and DMA. Data in memory: basic data types, size, and alignment. Code as instructions in memory. Operand addressing and registers. The concept of context: saving and restoring registers. Context switch and interleaving. Multi-core and parallelism. CPU privilege and kernel mode. Protecting entry to the kernel: traps, faults, and interrupts. The memory hiearchy, devices, and DMA.
- **Programs and processes**. Programs as files: source files, object files, executable files. Birth of a program: compiler, assembler, linker. Symbols and linking. Program entry point: main(). How the linker resolves symbols: the symbol table and relocation records. Libraries. Executable/object/binary files: ELF (Linux) and Mach-O (Apple) file formats. Program sections: text, global data, constants. Launching a program as a process. Initialization of segments in the process virtual memory.
- **Advanced**. The Unix kernel interface and Linux. The container abstraction and its isolation mechanisms. Discussion of Docker and its security.
- **Lab/project work**. Activity: ELF file format for Linux. Lab: program execution and fault handling in the JOS operating system.

Week 2: Virtualization

- **View from a process**. Focus on the Unix programming environment. Timeslicing and the process context. Multi-programming and data residency in memory. The file tree, files, file system calls, and memory mapping. File descriptors and standard I/O: stdin, stdout, and stderr. Where does your data live? Stack, heap, globals. Data lifetime and allocation: implicit (stack), explicit (heap), static (globals). Command line arguments. Dynamic memory and the heap. Two-level memory allocation in the kernel (sbrk or mmap system calls) and heap library (malloc/free). Stack frames and the stack discipline: procedure call (push) and return (pop). Addressing data in the stack, heap, and global segment.

- **Heap management**. The heap manager contract and the parking lot analogy. Fixed vs. variable partitioning. Allocation policies: first fit, best fit, worst fit. Which is best? It depends on workload. Size and alignment of heap blocks. Pointer arithmetic. Wild pointers, dangling references, and memory leaks. The importance of initializing heap blocks. Data layout in heap blocks. Byte ordering: little-endian (Intel) and big-endian (Internet).
- **Virtual memory and paging**. Virtual pages and page frames. Virtual address translation. Translation tables. Page protection bits. Pros and cons of paging.
- **Advanced**. Hierarchical page table layout, memory mapping, IA-32 and x64 address translation, the translation lookaside buffer (TLB), internal kernel data structures for virtual memory, and the Mach VM system used in MacOS. Kernel space and the Meltdown attack.
- **Lab/project work**. Activity: steps to a heap allocator. Lab: heap manager library.

Week 3: Protection and authority

- **The kernel**. Processes and threads. User-mode vs. kernel-mode thread execution. Partitioning of the process virtual address space into user space and kernel space. Safe entry to kernel mode: traps, faults, and interrupts. How the kernel controls the virtual memory translations in effect: protected control registers, the page table as a kernel data structure, and IA-32 page table entries. Causes of faults. How the kernel uses tiimer interrupts to gain control of the CPU. System calls: stubs, and the Application Binary Interface (ABI). Bullet-proofing the syscall interface: arguments, copyin/copyout, system call dispatch table, kernel objects and their descriptors. The OSTEP concept of limited direct execution.
- **The process lifecycle**. Thread and process states: ready, running, blocked. Context switches: preemption, yielding, waiting/blocking. Per-thread stacks. Thread/process exit.
- **Threads in the kernel: blocking**. Threads and cores: the car-share analogy. Causes of context switches. Thread states and the role of the kernel in state transitions: sleep, wakeup, preempt, dispatch. The ready queue and sleep queues. Mode switches vs. context switches. Examples: page fault, I/O system call. The thread control block (TCB). Kernel stacks and trap/fault handling. The view from a core.
- **Advanced**. Virtual machines: host/guest mode and nested page tables. Alternative approaches to memory protection: language protection (e.g., Rust), memory protection unit (MPU) in ARM/Cortex embedded processors.
- **Lab/project work**. Activity: user-space threading and context switch with ucontext primitives: makecontext and swapcontext. Lab: user-level virtual memory (advanced).

Week 4: Concurrency

- **Concurrency and races**. Logical vs. physical concurrency. Interleaving and schedules; interleaving of load/store instructions. Safe and unsafe schedules. The Posix thread example from OSTEP. Modeling schedules with resource trajectory graphs. Races and critical sections. The need for mutual exclusion. Introduction to monitors: locking and mutexes; conditions and notifications; wait and signal. Synchronization in Java.
- **Monitors: digging deeper**. Mesa monitors and mesa semantics. Monitor APIs for the threads lab. Using monitors: invariants; shared vs. per-thread state, the missed wakeup problem and how monitors fix it; the monitor contract and "commandments"; spurious wakeups; the use of broadcast/notifyAll; thundering herds. Example: producer/consumer bounded buffer (soda machine).

- **Starvation and deadlock**. Building a reader/writer lock (SharedLock) using monitors: issues and design alternatives. The problem of starvation. Adding priority to control starvation. Dining Philosophers and the problem of deadlock. Formalizing deadlock: the wait-for graph model, effect of cycles. Mitigating deadlock.
- **Advanced**. Data race detection (Eraser). Memory consistency: the x64 memory model, sequential consistency, barriers, and total store order (TSO). Kernel concurrency: interrupts and nesting, disabling interrupts, synchronizing with ISRs, x64 spinlocks.
- **Lab/project work**. Activity: using pthreads. Lab: concurrent disk scheduler (deli).

Week 5: Synchronization

- **Implementing synchronization**. Overview of synchronization in the threads lab. Implementing locks: blocking, handoff locks, atomicity. Disabling interrupts to control logical concurrency with preemptive timeslicing.
- **Atomic instructions and spinlocks**. The need for atomic instructions on multicore/multiprocessor computers. Examples: test-and-set, compare-and-swap. Using atomic instructions to implement spinlocks. Spinning vs. blocking.
- **Semaphores**. Semaphores as an alernative to monitors. The semaphore abstraction. Using semaphores. Example: producer/consumer.
- **Advanced**. Fast locking: hybrid spinning/blocking locks, the Linux futex primitive, non-blocking synchronization, read-copy-update (RCU) in the Linux kernel. Discussion of kernel scalability.
- **Lab/project work**. Activity: thread/concurrency exercises. Lab: atomic instructions (advanced).

Week 6: Scheduling and I/O

- **Thread design patterns**. Threads as asynchronous tasks: parallelism and the join primitive. Pipelines and producer/consumer bounded buffer: mailbox, message queue. Event-driven programming: the main thread or User Interface thread (e.g., in Windows or Android), AsyncTask. Upcalls. The actor model and reactive programming.
- **Threads and servers**. Messaging, request/response, the client/server model, Remote Procedure Call (RPC). Threading alternatives for servers: serial, event-driven, multiprogrammed (thread pool).
- **CPU Scheduling**. Scheduling as a resource allocation problem. Policy vs. mechanism. Performance metrics: response time, throughput, overhead. Scheduling for low average response time. The role of workload and the problem of fairness. Scheduling policies and their tradeoffs: shortest-first, FIFO, round robin. Importance of responsiveness for I/O utilization and throughput. Use of task priority for I/O-bound and CPU-bound tasks. Multi-level feedback queue.
- **Advanced**. CPU performance isolation: proportional-share allocation, Linux cgroups, lottery scheduling. Processor management for fast thread systems: the problems of processor sharing and I/O; scheduler activations.
- **Lab/project work**. Activity: semaphore problems. Lab: implementing threads (Part 1).

Week 7: Servers

- **Sockets and servers**. Network communication and networked servers. The socket abstraction. Example: stream (TCP) sockets on the Internet. IP addressing and ports. Network I/O: devices, packets, byte swapping, demultiplexing. The network protocol stack. The HTTP protocol and the Web.
- **Performance**. Server performance metrics and basic queuing models. Understanding utilization and throughput: capacity and bottlenecks. Understanding response time: Little's Law. Predicting average response time. The problem of tail latency. Quantiles.
- **Overload and scaling**. Scale-up vs. scale-out, limits to scaling: Amdahl's Law. Admission control and load conditioning. Elastic services, capacity scaling, and the cloud.
- **Advanced**. Fast networking and direct device access; discussion of Dune. Secure containers: Google's gVisor and Amazon's FireCracker.
- **Lab/project work**. Activity: multi-threaded Web server. Lab: implementing threads (Part 2).

Week 8: The Unix process model

- **Fork and exec**. Unix process management. Launching a process and executing a program. The fork abstraction. The wait() system call. The process tree: login and init. The Unix command shell.
- **Access control**. The need for access control: subjects, objects, and authorization. The OS kernel as a reference monitor. The Unix protection system. Process identity and inheritance. The superuser. Protecting changes to process identity: the setuid system call and the setuid bit. Trusted programs.
- **Pipes**. The Unix inter-process communication (IPC) model. Process inheritance of I/O descriptors. Standard I/O descriptors and the dup() system call. I/O reference counting. Design of the Unix command shell.
- **Advanced**. Discussion of the roots of Unix. Revisionist view of the fork abstraction and its problems.
- **Lab/project work**. Activity: pipes and filters. Lab: unit testing with fork/exec.

Week 9: Security

- **Login and passwords**. The problem of authentication. Passwords and their quality. Protecting passwords at rest: cryptographic hashing and dictionary attacks.
- **Crypto**. Principals in a networked system: Alice, Bob, and Mallory. Basic cryptosystems and keys: symmetric vs. asymmetric cryptography. Using asymmetric crypto: authentication, integrity, and confidentiality. Hybrid cryptosystems: secure sockets and transport-layer security (TLS). Choosing a key size.
- **Certificates and the secure Web.** Mallory-in-the-Middle attacks. Protecting passwords in transit: phishing, spoofing, and secure HTTP (HTTPS) using TLS. Authenticating a server: certificates. Digital signatures and their use in certificates. Certifying authorities. Browsers and the root of trust.
- **Advanced**. Sandboxing: isolation with NaCL. Browser security. Secure Linux kernel extensions with BPF.
- **Lab/project work**. Activity: lab discussion. Lab: shell (advanced).

Week 10: More security

- **Smashing the stack**. Buffer overflows and remote code execution (RCE) vulnerabilities. Common vulnerabilities and exposures (CVEs) Stack smashing attacks: IA32 stack layout, smashing the return address, injecting shellcode. OS defenses for stack smashing: no-execute protection (NX), address space layout randomization (ASLR), stack canaries.
- **Theats: vulnerabilities and exploits**. A sampling of instructive CVEs and their exploits. Memory disclosure: Heartbleed. Side channels: Meltdown. Trojan horses, malware payloads, ransomware and spyware, rootkits. Drive-by installs: Pegasus and Sandworm.
- **Cyberwar**. Threat groups and advanced persistent threats (APT). Stages of an threat campaign. Spear phishing, privilege escalation, pass-the-hash. Software supply chain attacks. Illustrated with examples: Stuxnet, Sandworm. Zero-day exploits: the story of EternalBlue and WannaCry. Cyberwar and the future.
- **Advanced**. Trust in code and trusted computing. Thompson's Reflections on Trusting Trust. The secure enclave abstraction: Intel's Software Guard Extensions (SGX) and Microsoft's Haven for secure computing on untrusted clouds.
- **Lab/project work**. Activity: lab discussion. Lab: stack smash (advanced).

Week 11: Failure and recovery

- **Fault models and replication**. Datacenter services: replicating services for availability. Fail-stop faults and fail-over. The replicated state machine model (RSM/SMR) abstraction. Service-independent RSM protocols: consensus and convergence.
- **The problem of network partitions**. The consensus abstraction. Correctness properties: safety and liveness. The asynchronous communication model: network partitions and split-brain syndrome. Impossibility of consensus in asynchronous networks: the Fischer-Lynch-Patterson result. The consistency/availability dilemma and the CAP Theorem.
- **Raft consensus**. RSM consensus under partitions: voting and the quorum rule. Views, terms, and leader election. The Raft protocol and its handling of key failure scenarios. Proving safety properties. Raft vs. Viewstamped Replication and Multi-Paxos.
- **Advanced**. CPS 510 uses week 11 for further security topics: mandatory access control (MAC), decentralized information flow control (IFC and DIFC), information leaks and taint tracking. Discussion of Flume and TaintDroid.
- **Lab/project work**. Activity: lab discussion. Lab: Raft consensus (CPS 310 only).

Week 12: File systems

- **Files**. Naming in the file system: directories, links, symbolic links, system calls. Indexing file and directory contents: inodes, direct and indirect blocks, skewed tree block maps. Exemplary file system layout on disk: data vs. metadata.
- **Storage stack**. The block storage abstraction. Modular structure of kernel I/O software: device drivers, logical volumes, virtual file system (VFS), filesystem modules, removable file systems, and building the file tree with the mount system call. Role of VFS in supporting the network file system (NFS). Network storage models: files (network-attached storage or NAS) vs. blocks (storage area network or SAN).

- **I/O caching**. Review of block I/O via DMA and interrupts. Block caching and the buffer cache. Sequential access and heuristics for prefetching: read-ahead. Handling writes: delayed writes and asynchronous writes. Safety of metadata: the recovery problem and the need for atomic updates (crash consistency).
- **Advanced**. Crash consistency semantics and their impact on applications.
- **Lab/project work**. Activity: working with ext2. Lab: ext2 (advanced, CPS 310 only).

Week 13: Storage management

- **Files on the disk**. Block placement on disk. Performance impacts of access locality. Example: the Linux Ext2 filesystem. Block groups, superblocks, allocation bitmaps, inode table.
- **Updates and recovery**. The file system as a recoverable state machine: the meaning of crash consistency. Atomicity properties of disks. Logging and atomic snapshots. Failure-atomic transactions with write-ahead logging and checkpoints. Shadowing for atomic checkpoints and snapshots. Example: NetApp's Write Anywhere File Layout (WAFL).
- **RAID**. More about disks: access time, seek time, rotational delay. Solid-state storage devices (SSDs) and their properties. Combining disks in logical volumes: RAID, load spreading, failures and the need for redundancy. Striping, mirroring, and parity (RAID-5) and their tradeoffs of cost, access time, throughput, and availability. Striping in supercomputer file systems (Lustre).
- **Advanced**. NetApp's WAFL in depth, and its roots: write throughput, the log-structured filesystem (LFS) idea, RAID-4 and the parity bottleneck, shadowing and consistency points, write cost vs. read locality.
- **Lab/project work**. Enjoy the end of the semester!