Operating System / Kernel [L2, L3]

What are the three roles of the OS?

How do these apply to various hardware resources (e.g., CPU, Memory)?

What primitives does hardware provide to the OS?

Isolation via execution modes and virtual memory

Ways to transition between execution modes

User/Kernel -> Kernel: Traps

- What are the types of traps (i.e., syscalls, interrupts, exceptions/faults)?
- How are the various types of traps handled?
  - High-level, but also for xv6 on RISC-V

Kernel -> User: Upcalls

- Why are these useful?
- How is this supported by the OS and the process?

Processes and Threads [L2, L3, L8]

What is a program? A process? A thread?

- How do processes and threads differ (e.g., purpose, state)?

What is the state machine for processes/threads exposed to the scheduler?

- What are the states (e.g., RUNNING, BLOCKED)?
- How do transitions occur between states?

How do processes and threads support concurrency?

- What are the implications of 1:1 and N:1 threading models?

What is the xv6 implementation for processes and threads?

Can Ignore:

- N:M threading model (aka scheduler activations)

Programming Interface [L2, L3]

Why are syscalls needed?

How are syscalls implemented?

- High level, but also xv6/risc-v syscall handling

What are file descriptors?

What do common syscalls do? fork, pipe, exec, dup, wait, close

- Don't need to know how to implement them, but just how they are used by programs. You should be able to look at an example program and determine its behavior

(Virtual) Memory [L4, L5, L6, L7]

How are memory allocations managed? Mechanisms? Policies?

- What is fragmentation? How can it occur? What are the flavors (external/internal)?

How does Base and Bounds work? Segmentation? Paging?

- What are the pros/cons for each?

How does the TLB work? What problem does it solve?

How do multi-level page tables work? What problem does it solve?

How is an access to a virtual address handled end-to-end?

- Cache, TLB, Page Table, etc

How can we apply virtual memory indirection in useful ways?

- Key Enablers: 1) Memory mappings, 2) Page fault handlers
- NULL pointer debugging, lazy allocation, demand paging, copy-on-write

Can Ignore:

- Heartbleed
- Specific details for the sv39 format (e.g., bit layout for PTEs), but know what fields there should be (e.g., PFN, RWX, V)

Concurrency/Synchronization [L8, L9, L10]

Why do we need synchronization for concurrent programs (e.g., multiple threads)?

- Data races, mutual exclusion

What primitives do hardware and compilers provide to build on top of?

- Atomic instructions (e.g., test and set)
- Barriers/Fences

What is a lock?

- What is the API (acquire / release)?
- What are the properties of a lock (e.g., safety)?

How do we build a simple lock (spinlock) leveraging the above primitives? What are the pros/cons?

- What is the issue with using spinlocks in both threads and interrupt handlers?
- Know yield locks generically

Can Ignore:

- Event-driven I/O (but understand pros/cons of thread based approach)
- Specific pthread_* API details (e.g., attributes)
- Anything onwards of sleeplocks in lecture notes