310 MIDTERM STUDY SHEET | byte = 8 bits, KB = 2^10 bytes, MB = 2^20, GB=2^30, TB=2^40 | n-bit addr. space = 2^n addressable bytes

OPERATING SYSTEMS / KERNEL - 3 ROLES OF OS

1. Multiplexing of hardware (hw) resources

- CPU while finite, is able to handle many processes that want to run → OS does through **scheduling**. Many processes load & store to/from memory & disc, even though RAM & disc are finite. OS does this through **VM**, **VA** spaces, **Page table** management/**TLB** management.

2. Abstraction of hardware platform

- Written, compiled, & executing progs abstracted to processes. Syscalls provided to access hardware resources. **VM** abstracts physical memory, allows programs to see their own large address spaces. Use of **threads** & **locks** to abstract low level registers & instructions for a process. Turning data storage into the filesystem.

3. Protection of software principles from each other with use of hw

- OS guarantees isolation through **VM** & **VA** spaces. Per process **page table** structure. **Access permissions** in PTEs (reading, writing, user bit, etc). Privileged mode on processor turned on through **trap** process → 'protects' hardware resources. **File system permissions.**

WHAT PRIMITIVES DOES HARDWARE PROVIDE TO THE OS?

- Execution modes on a CPU. Interrupts by other devices. Exceptions by CPU for illegal behav. Timers & clock. TLB, MMU, hw that walks page tables for addr. transl. Access to RAM for multi-level PT. Bit manipulation for PTEs. Registers & instr. set.
- Ways to transition between execution modes:
  - **user→kernel** / **kernel→kernel(traps)**: syscalls, interrupts (async, external event to be handled, ctrlc timer etc), exceptions (generally sync, illegal instructions like divby0 or page/seg faults)
  - **kernel→user**: upcalls/signals (async notifs from kernel space to user space, don't alter execution). Call some kind of signal handler every time a # of timer interrupts occur (LAB **ALARM**).
    - USES - Networking: I/O interrupts from receiving a packet are passed through applications later, calling user level functionality. Signals: sigint, sigterm implemented thru upcalls. Interrupts/ exceptions that affect user prog could be handled thru upcalls.
  - User→Signal Handler should never happen!
- Implementation of transitions between exec modes: TRAPS
  - User→Kernel (syscalls): We have diff execution modes supervisor & user (set in mstatus register in RISCV that's changed when user thr. transitions to kernel thr.). Need mapping of virt to phys addr spaces - trampoline and trapframe saved at top of VA space, allowing kernel to execute w/out page table switch; SATP register contains address of root level page table, changes based on process
  - User mode passes args down to kernel level syscall - wrapper calls ecall: args placed in a0-a6, a7 contains syscall #; users PC register w instruction saved in sepc; caller's mode saved in sstatus; pc changed to trap handler stored in stvec. Traphandler called inside kernel, which determines cause of trap w usertrap(), syscall() in syscall.c determines funct to call, see syscall # from a7; usertrapret() called in trap.c after syscall executes→sets trapframe vals to future vals, sets privilege mode to user, resets pc, jumps back to trampoline.S
  - K→K (exceptions, interrupts): Instead of going to usertrap() in trap.c, it goes to kerneltrap() in same file. Before going to kerneltrap in trap.c, control flow goes to kernelvec, rather than uservec. Device interrupts, timer interrupts, & kernel exceptions are handled here.
  - Upcalls/signals in Alarm: We pass in a fnctn handler that's stored in proc structure. When timer interr occurs, kernel upcalls to user program by setting epc to addr. of the handler, calling user function.

PROCESSES & THREADS

Program: List of instructions, stores global vars, stored indisk

Process: Abstraction created by OS, instance of a program. Have ids, addr. spaces with stack heap code data, file desc table, & state.

- Running→ready(descheduling), running→blocked(proc waiting on I/O oper, parent proc wait()), blocked→ready(proc done waiting, child exited), ready→running(schedule proc)

Thread: abstraction that represents execution contexts within process. Execution sequence for task scheduled by OS. Borrow from process addr. space, have own stacks & register state & gen state.

- Running→ready(thread called yield() or descheduled), running→blocked(thread called join(), waiting on I/O oper), blocked→ready(I/O oper done, joined thread exited), ready→running(scheduler schedules thread)

1:1 model: 1 kernel thread / user thread. OS controls all threads w scheduler. Used in xv6 & linux. Syscalls/traps related to user thread are handled by corresp. kernel thread. Needs lot of state and needs OS kernel for context switch between user & user→kernel thread.

N:1 model: OS sees single thread. User level threads implemented through a library run on 1 kernel thread. Preemptive scheduling w timer interrupt by CPU, timer triggers kernel signal to do curr user thread. If syscall block, go in kernel, all threads blocked. Block=bad.

Implementation for processes & threads in xv6

- **Process control block** (struct proc) contains trapframe, process state, pid, page table, fd's etc. Processes have own page table & addr. space. PCB stores context for kernel thread.
- Each process has user thr(curr prog) & kernel thr(for traps). Uservec & userret handle transitions, save user context (registers in trapframe), restoring kernel stack pointer & page table, restoring user context in userret. Kernel level thr. exists for sched. & yield() switches to this; scheduler thread determines next user-level process to run. Switches happen with swth(). Kernel mode thr. share page table.

PROGRAMMING INTERFACE

file descriptors: "handles" to resources, syscalls take them as args. 0 points to stdin, 1 stdout, 2 stderr, etc

syscall: APIS for user to access privileged hw resources via OS kernel. High level: User level wrapper calls ecall, which saves status of user prog, current pc, loads in args, & jumps to trampoline + trap handler. Trap handler calls the syscall, loads the return val into a0. Trap returns, & tramp. code restores user state. Back in user space. (see ← for low level)

- **fork()**: Creates copy of parent proc & addr. space. Child has own pid, addr. space, & copy of fd table & executes. Returns child pid for parent, 0 for child, use this to differentiate.
- **wait()**: Waits for child proc to finish, block parent until. Using with fork makes output deterministic. Child is in zombie state.
- **exec()**: Changes the program being run by proc that called. Used with fork; fork creates child proc, exec replaces child.
- **pipe(fd[2])**: Each end = fd. fd[1] writes, fd[0] reads. Internal buffer holds data to be read. Info goes in 1 direction. Use for interprocess communication, use 2 for 2-way.
- **dup(fd)→fd** or dup2(fd,fd): Duplicates fd for same resource.
- **close()**: closes what a fd references. Opens up fd for new stuff.

synchronous = caused by execution of an instruction | trap = forced control transfer to handle event or action (syscalls, exceptions, interrupts)

(VIRTUAL) MEMORY

Mem. allocation mgmt: Initial allocation w syscall like mmap() or sbrk(). Allocation of VM is initial size of heap, free list of mem. Allocations & frees from mem take from free list & add back to it.

Mechanisms for mem allocations:

- **Splitting**: Each header in freelist is checked for a block size big enough for request. If big enough, could be split, leaving extra.
- **Normal**: Node is exactly size of allocation demands, remove from list & return to caller of malloc()
- **Coalescing**: Freed nodes can be combined w adjacent nodes when being inserted back into list if mem addr, aligns on a side

Allocation policies: First fit=choose node that fits allocation first in traversal. Best fit=choose smallest node that fits allocation. Worst fit=choose largest. Can cause external frag in freelist. None are best

Fragmentation: Waste in allocated mem. **External** is when mem to be allocated is in small chunks w spaces. **Internal** is when too much mem is given to the proc that's allocating, happens under worst fit.

Base+bounds: To get PA from VA, add VA+base. If VA≥bound, error. Pros=simple. Cons=internal frag, lots of space allocated for stack & heap growth; bad at multiplexing RAM, proc takes up addr. space

Segmentation: Each process maintains seg table, each entry is random sized segment w base & bounds. Segs can be anywhere, not fixed size, have access permission bits. Register points to seg table. 2 processes can share physical seg. VA is divided into **ID** and **offset**→ id indexes into seg table, seg table entry gets base and bounds, PA = seg base + VA offset. Exceptions: VA.id not found, VA.offset≥bound. Pros: less frag, shared code, access perms. Cons: mem of seg tables.

Paging: Each seg of mem is made of fixed size pages(4kb?).

TLB: cache in MMU that speeds up mem translations. Maintains cache of VPN⟶FPN mappings for valid pages. Can flush TLB on context switch(diff CPU instr)=ovrhd, or use tagged TLB w ASID field.

Multilevel page tables: VPNs split into indices for each level, and PFNs in higher level tables point to the addresses of lower level tables ⟶ solves problem of paging taking space/overhead

- L0 lowest table points to main mem; VA split into indexes for each level + offset like before
- Assume 3-level PT: L2 index indexes into the L2 table (the address of the L2 table is stored in the satp register in RISCV. The entry in L2 points to the address of L1. L1 index indexes into L1, getting the address of L0. L0 index indexes into L0, getting the PFN. The offset then indexes into the physical page to get the exact memory location.

HOW ACCESS TO VIRTUAL ADDRESS WORKS:

Find the VPN within the the virtual address. Lookup the VPN in the TLB:

- If it is a hit: Check protection bits: If they are properly set{Find the offset, Get the physical address using the PFN and offset, Access the memory} If they aren't{Raise an exception (protection fault)}
- If it is a miss: Get the address of the root level page table via the satp register, Traverse the multilevel page table using the VPN as index bits into each table, If at any point there is a page table entry with invalid protection bits{Raise an exception (protection fault)} Else if at any point the page table entry's present bit is true{Insert the vpn and pte into the TLB, Retry the instruction} Else if the present bits are false aka not in main memory{Raise an exception (page fault)}

Page faults are handled like this: Find a free physical page in memory. This requires a policy if there is no free page and you need to evict a page, such as LRU, FIFO, etc. Could potentially check bits such as accessed (for LRU), or dirty to see if you need to write the evicted page back into the disc swap space. Perform an IO operation to move memory from disc into RAM. To get the memory, use the disk address found in the page table entry corresponding to the original virtual address. Since it is not present in physical memory, the PFN will have the disc address rather than RAM address. Additionally, pass in the PFN of the newly found free page. Set the present bit to true in the page table entry. Set the PFN to the new PFN. Retry the instruction. If the page is NOT found in disc, a segfault will be thrown.

- After getting the physical address, the actual caches are checked for memory: L1, then L2, then L3, then DRAM

Applications of VM indirection:

- **Null pointer debug**: make user VAs start 1 pg above 0, "guard"
- **Lazy allocation**: When a proc calls malloc in xv6, sbrk() syscall increases "size" of heap w/out allocating mem. Each access to mem causes page fault, page fault handler checks if accesses are valid & then allocate on demand. Better for space & perf.
- **Demand paging**: VA space often > phys mem. Disk has space but is slower. Use disk as swap space for excess. PTE stores a disk address & valid bit = 0. Eviction policies: min, lru, etc.
- **Copy-on-write**: Reduce waste of copies; modify all p & child PT entries to read & point to same mem; if write, page fault handler copies the memory & protection bits are allocated

CONCURRENCY/SYNCHRONIZATION

Parallel computation needs more access to compute resources, make use of all CPU cores. I/O handling requires blocking operations, may want to compute alongside I/O operations.

Threads share address space (except individual thread stacks)

- **Data race**: 2+ threads access shared mem, at least 1 is write
- **Critical section:** code subject to ^; should be mutually exclusive, can't be accessed at same time by multiple threads
- Interleavings ⟶ thread scheduler is **non deterministic** (? out)
- Hw offers **barriers** for compiler & CPU, sync_synchronize()

Lock: abstraction around atomic instr. & barriers for mut. exclusion

- **Acquire**: attempts to set lock var to 1. If already at 1, calls barrier() to prevent moving a crit section ahead of lock access
- **Release**: set lock variable to 0 in **atomic instruction**
- Safety: bad never happens, liveness: good will happen
- Lock data, NOT code. Acquire before a critical section, wait if lock is held currently. Thread releases lock after finishing.

Spinlock: Acquire: disable interrupts, while loop with at.instr. to check if lock acquired. If no, create barrier, safe. If yes, do nothing/spin & iterate again. Enable interrupts⟵prevents stuck.

- Pros: simple, good for short crit.sects. so don't waste CPU cycles. Cons: **starvation** (thread may not be scheduled), waste

Yield lock: Instead of spinning, yield processor to something else

- Pros: "goodput" aka do smtg. Cons: starvation, time of restart