Advanced scheduling
Real-time OS: guarantee performance speed exp
Hard realtime: meet deadlines or reject if not feas
Soft realtime: meet deadlines with high prob
Periodic: recurrent over time - deadline < interval -> deadline = period, determine feas at compile
Aperiodic: unpredictable start times, no deadlines
Sporadic: Unpredictable start times, has deadlines, feas at run
EDF: priority with preemption, when feas - hard
- feas - n tasks with comp time (sum $c_i/d_i \leq 1$)
- good: simple, cpu always utilized
- bad: which job is gonna miss?
Rate Monotonic - priority with preemption fixed 1/D
- feas 69% Util = sum $c_i/d_i \leq n(2^{0.5} - 1)$
- - maybe at $n(2^{0.5} - 1) \leq U \leq 1$, no $U > 1$
- good: simple stable
Best CPU throughput - FIFO
Best avg turnaround - SRPT
Best avg response - RR
Favor important tasks - Priority
Fair CPU usage - Linux CFS
Deadlines - EDF, RMS

Device IO
USB - gen purpose, short range, power devices
PCIe - lots of data, fast, short range, no power
Ethernet - long range, lots of data, no power
Devices: in (r), out (w), store (rw)
Connecting: busses (wires) determine protocol, closer = faster, reliable (GPU), further = flexible (USB, even ethernet)
- half duplex (1 way) or full duplex (2 way)
Parallel port - write 8b data, set strobe bit low to mark busy low, when busy turns high mark strobe high and write again
Serial port - before USB, 1 wire to transmit, 1 to receive, serial
Serial Adv Tech Attachment - SSD/HDD, short length = high speed
Peripheral Component Interconnect Express - wifi/ssd, parallel
Talking: assembly to read/write device
Port-Mapped IO - x86 IN port reg, OUT reg port
Mem-Mapped IO - part of main mem given to ports
- manage in kernel memory, device not in memory, no cache
Device interactions: while busy wait, write data and command, while busy wait, read data
Sync events: poll for device to be ready
Async events: interrupts to block and free cpu, conc concerns
Programmed IO: programmer writes IO funs they take cpu time
Direct mem access: tell dev address and size

Device drivers
OS: dev discovery on boot, abstractions deal with varied devs
- broad classes: char (bytes) block (chunks) network (packets)
Application layer: use file syscall/wrappers for files in /dev
- or use posix aio lib funcs
Kernel IO subsys: does perms, routes call to driver, sched req, addr trans, journal
Dev driver + interrupt handler: translate OS/upper layer to dev specific code

Virtual memory
CPU reality: limited ram for many processes
OS goals: process as ~2^64B start at 0x0, process can't see other processes, ram size doesn't matter, code reuse
Mem manage unit: OS sets up PT for process, then MMU hardware does translation w/o context switch unless fault
Segmentation
- Translation: find segment (# seg bits (upper) = log_2(# seg), find PA (PA base + VA offset <= PA bound), check perms
- Context switch: switch to kernel + deactivate MMU, process seg table saved, new process seg table loaded, switch to user + jump to new process
- Good: can grow seg (stack/heap), change individual seg protections, dynamic relocation, simple hardware, PT fits in registers
- Bad: Memory fragmentation because entire irregularly sized seg must fist
Paging: solve fragmentation w/ fixed 4KB sized pages, # pages = 2^add_bits / page size
- Translation: lookup VP# (upper) in PT for PP# (upper), go to VP=PP offset (lower) in memory
- Good: no bounds, secure, share, load only relevant VP ram
- Bad: many pages so large PT must be in RAM
Improve translation speed: TLB - cache holds subset of PT so if hit skips 2 mem seeks, flush tlb on cont switch or track proc corresponding to entries
Improve table size: 1 PTE for every VP is a lot, eliminate invalid entires and make hierarchies (more mem seeks), VA = {PT1 IDX}{PT2 IDX}{PT3 ITX}{PT4 IDX}{Offset}
Thrashing - VM resources are overused, constant page faults

Swapping
On cont switch, change PA in PT for new VA %CR3
Lazy: on fault (PA not in PT) go get it
mmap(): space to put file in vas for speed
Disks: nonvolatile, boot, fs, swap space, holds pages for ram
Page faults: soft (PP in RAM, PTE not configured, update PTE), hard (PTE valid but not in RAM, lazy), invalid (PP not valid and not present, bad perms, terminate)
Page eviction (PT too full): optimal (page accessed furthest in future), FIFO (fair but some pages are always needed, stack), least recently used (closest to optimal, need to track access, bad for cyclical pages), clock (imperfect LRU, mark accessed bit 1 when used, go down PT to kick next page with accessed=0, point to next PTE for next kick - improvements: accessed counter w more bits, timestamp for old enough)

RAID
Redunant array of ind disks: several lower quality disks to make up big one
RAID 0 - striping
- divide logical disk into chunks
- distribute the chunks regularly over two phys disks
- + throughput for both random Traid0 = N*Tdisk
- + capacity scales by N
- + cost per byte is identical
- - mean time to failure is worse bc fail of single disk is bad
RAID 1 - mirroring
- duplicate each chunk on each of N physical disks
- + impossible to lose data unless both disks fail
- - write throughput is not improved
- - capacity is the same as a single disk

- - cost per byte is greater $raid1 = N * $disk
RAID 4 - parity
- distribute the chunks across the first N-1 disks
- on Nth disk, store a parity chunk (parity block is redundant data about a set of chunk, a stripe)
- can tolerate loss of any one disk
- - limit throughput bottleneck for writes
Parity
Even/Odd - add a 0 or 1 such that the total 1s is even/odd
Even ex: 0000_0000 par 0, 1111_1111 par 0, 0110_1101 par 1
Infer

RAID 5 - distributed parity
- distribute parity chunks across disks to avoid bottleneck
- + failure of one disk is okay
- + throughput is good Traid5 = (N-1) * Tdisk
- + cost per byte is good $raid5 = N/(N-1) * $disk
- - high overhead for small N
- - failure risk is high for large N
- N is typically 3 to 8

RAID 6 - double parity
- Add another disk and keep two parity chunks per stripe
- + failure of TWO disks is okay
- ~ throughput is less Traid6 = (N-2) * Tdisk
- ~ cost per byte is higher $traid6 = N/(N-2) * $disk
- N is typically > 8

Security
Trusted computing base
- everything the OS relies on to enforce security
- TCB includes scheduler, mem management, parts of fs, parts of dev drivers
- anything else must be assumed malicious
Properties: confidentiality, integrity, availability
Security concerns
- Processor access (integrity: user vs kernel mode; availability: timeslicing)
- Memory access (confidentiality and integrity: vmem and perms; availability: swapping)
- File access (confidentiality: user and group perms; integrity: only accessible through system calls)
- Device access (confidentiality: user perms sort of)
Memory attacks and defenses
- Buffer overflow
- Heartbleed attack

File system
Limited hardware interface to make convenient naming, organization, translation, protection, reliability
Challenges: disk performance, persistence of data, free space management
Links: hard (another entry in a directory with same disk address), sym/soft (contents is just string path of another file)
Track: free disk blocks, blocks containing data for files, files in a directory
Can't do sequential blocks
Allocation table - file tracking is an array of block pointers
Index node (inode)
- file tracking as an array of inodes (each one corresponds to a file)
- inode contents: file attributes, ordered list of pointers blocks
- ≤ one block in size

File system implementations
Cache popular blocks so the disk can be accessed less frequently, must be careful to prevent two threads from accessing different unsynchronized copies
Prefetch
- Load memory before needed
- Read multiple blocks from disk sequentially or load specific files based on usage patterns
- Need to balance prefetching requests with other disk access
FAT - File Allocation Table
- allocation table for tracking data blocks
- still in use for embedded systems
FFS - Fast File System
Unix - inode-based design
BSD - first disk aware file system
FFS groups - divide disks into cylinder groups and each has super block bitmaps inodes datablocks (inodes to data is slow)
- put directory data near directory inodes
- put file inodes near directory data
- put data blocks near file inodes
Crash tolerance
- filesystems are persistent and store important data, cannot rely on a graceful shutdown (power outage, kernel panic, usb unplug), structure updates are critical sections, all read and writes aren't necessarily guaranteed but the system needs to stay consistent
- crash before write to file's inode could leak a data block
File system checker (FSCK)
- after crash, scan disk for contradictions and fix (check data bitmap consistency)
- makes disks consistent, not correct
- very slow
Enforce atomic transactions
Journaling filesystems
- write all transactions to journal instead of actual locations
- write blocks to the log, then write a commit message to the log, then start writing all of the logged writes where they belong
- example
- - write transaction start to journal (transaction begin)
- - then actions for that transaction along with the data (write block 6, data: y; write block 7, data: z)
- - commit by writing transaction end (transaction end)
Resolving crashes with journaling
- no transactions happening when crash occurred: journal is empty, do nothing because there were no outstanding transactions
- crash occurred before commit: clear the log to roll back the transaction
- crash occurred after commit while writing data: replay the transaction from the beginning, then clear the journal
Journaling filesystems
- extended filesystem (default for Linux)
- NT file system (modern Windows filesystem)
Copy-on-write

- ZFS - concept of pools of storage (one filesystem manages multiple disks, replaces RAID by allowing filesystem to make choices)
- Log-structured file systems - can go further along copy-on-write path, no longer doing small writes all over disk

Virtualization

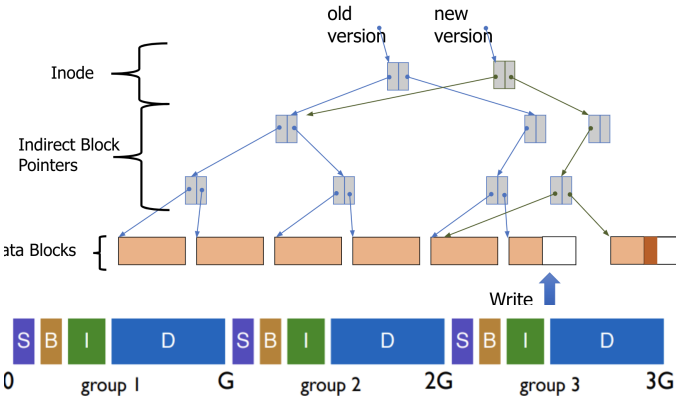
Emulation
- user software emulates the behavior of every single instruction
- ex. Gameboy emulator, QEMU
- any hardware you want entirely in userspace but slower than real hardware by definition

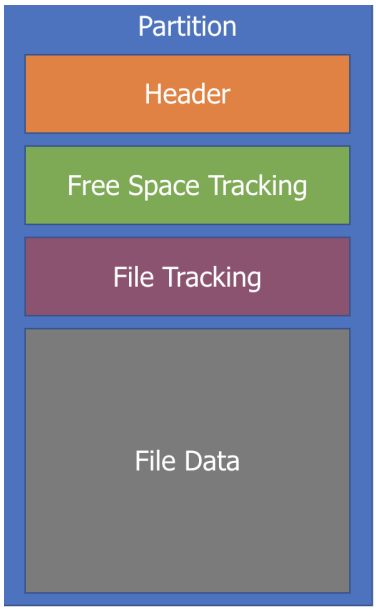
Hypervisors (Virtual Machine Monitor)
- OS kernel is the system “supervisor” and manages the computer
- creates the illusion that the OS has full control over the hardware
- ex. VirtualBox, VMWare, Parallels
- challenges include memory virtualization and I/O devices
- Bare Metal - entire hardware is hypervisor
- Hosted - hardware has OS which has a hypervisor and apps

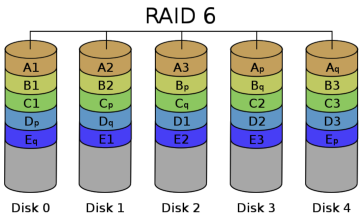
Containers
- provide each application with illusion of its own dedicated OS
- Linux cgroups - collection of processes treated as a group for resource allocation
- Docker
- - container packaging, distribution, and execution
- - images (snapshot of the system)
- - union file systems (file system as a sequence of layers)

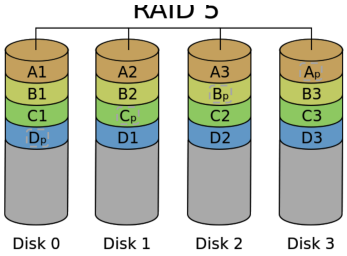
Embedded OS

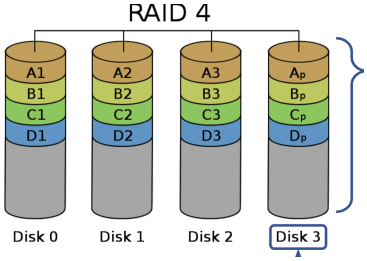
Tock
- kernel written in Rust

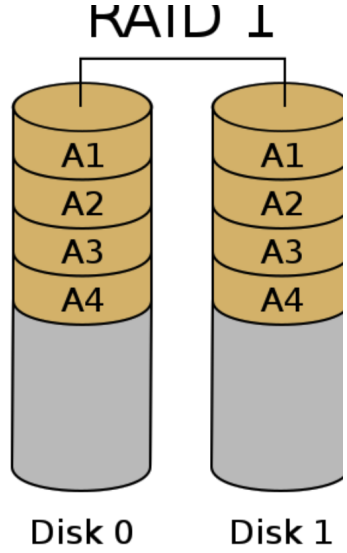


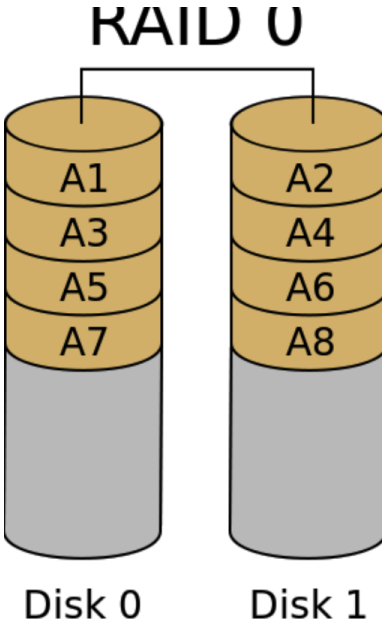


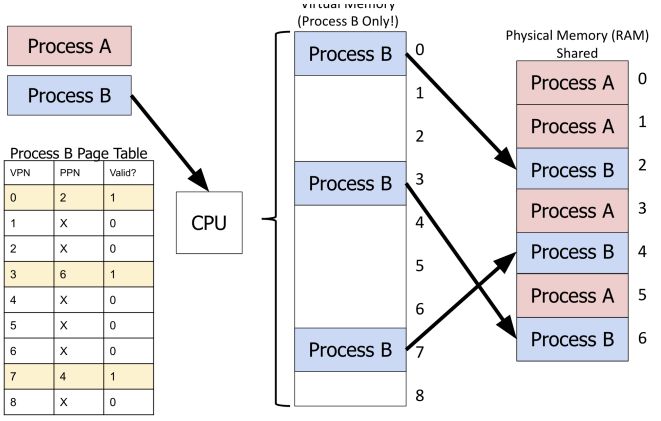


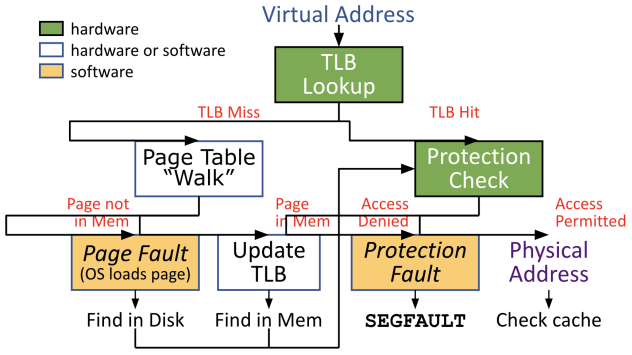


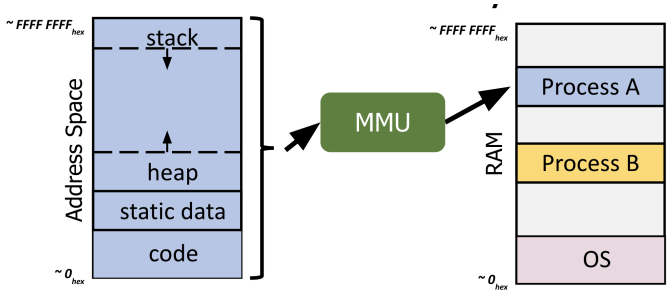




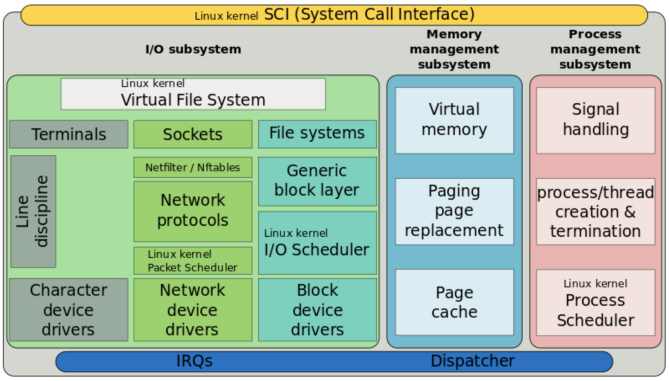








16



Page Requested

time

Optimal

D	D			Miss
D	D			Hit
B	D	B		Miss
B	D	B		Hit
A	D	B	A	Miss
C	D	B	C	Miss
B	D	B	C	Hit
D	D	B	C	Hit
B	D	B	C	Hit
D	D	B	C	Hit

Miss rate = 40%

FIFO

D	D			Miss
D	D			Hit
B	D	B		Miss
B	D	B		Hit
A	D	B	A	Miss
C	C	B	A	Miss
B	C	B	A	Hit
D	C	D	A	Miss
B	C	D	B	Miss
D	C	D	B	Hit

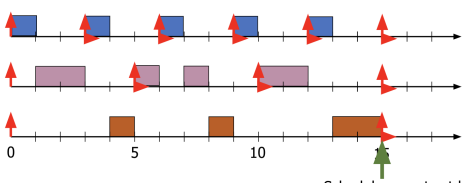
Miss rate = 60%

LRU

D	D			Miss
D	D			Hit
B	D	B		Miss
B	D	B		Hit
A	D	B	A	Miss
C	C	B	A	Miss
B	C	B	A	Hit
D	C	B	D	Miss
B	C	B	D	Hit
D	C	B	D	Hit

Miss rate = 50%

1/3 + 2/5 + 4/15 = 1



Schedule repeats at least common multiple