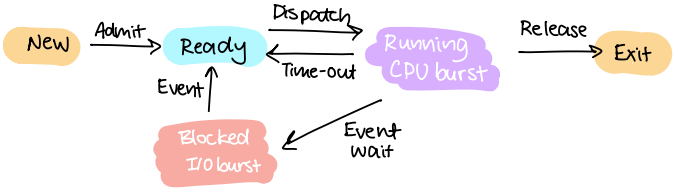
Virtualization illusion of unlimited resources, I own whole system
Concurrency coordinate multiple activities, ensure correctness
Persistence survive crashes & power failures
primary: convenience for user, secondary: efficient use of system
Limited Direct Execution user / system mode
Interrupts 1. OS fills Interrupt Table (boot time), sets IDTR
2. CPU fetch, decode, execute. 3. Interrupt
4. CPU disable interrupt 5. Save interrupted PC
6. PC ← interrupt handler = IDTR start + int. num.
7. Continue execution
Startup initialize data struct, create init process,
kernel → user mode, start first process, wait for interrupt
OS entirely driven by external events
Process a program in execution (running program)
Context switch save old state, load new saved state



System call kernel verify arguements, copy dat from user space into kernel buffers, save in a register

Thread a single control flow through a program
separate add. space from execution state (individual stacks)
A thread cannot anticipate ⏱ forced away from CPU
Synchronization enforce single use of shared resoure
prevent **race condition**
Critical Section mutual exclusion, fairness, performance
Lock data type abstraction lock() acquire, unlock() release
Spin lock busy waits, starvation, deadlock through prio^{-1}
test_and_set for atomic instructions
Sleep lock "blocked" thread goes into sleep queue, shared lock var
pthread_cond_wait, cond_signal, cond_broadcast
Lost wake up ⏱ signal happens before wait: add shr. var for signal sent
Bounded buffer one cond for not full, one for not empty

Semaphore has count, wait (P/decrement), signal (V/increment/post)
count > 0 # threads that can wait before blocks
< 0 # threads waiting; 0 no waiting, but no new can pass
Binary used like a lock, no lost wake-up
Bugs atomicity violation, order, deadlock
Deadlock conditions
1. Mutual exclusion use lock-free data struct, compare_&_swap
2. Hold & wait trylock() if can't get all resources try later
3. No pre-emption don't even try
4. Circular wait lock ordering, avoid, detect, Ostrich Algorithm
I/O opposite
Scheduling CPU bound long CPU bursts, infrequent I/O bursts
Service complete a run, **Wait** scheduled-arrival, **Turnaround** service + wait
First-come-first-served (FCFS) long wait, no starvation
Shortest Job First (SJF) minimize wait, non-preemptive, possible starvation
Round Robin
Priority Scheduling multiple ready queues MLFQ, each queue has diff. prio. level
choose highest prio., then RR

More Scheduling
Prio.$^{-1}$ ⏱ lower prio. prevents higher prio. from running
UNIX CPU Scheduling
favours interactive, small CPU slices initally given to new processes.
more CPU time ⟹ lower prio., I/O-bound gains higher prio.
Rescheduling every 0.1s, priority recomputed at every time slice.
MLFQ w/ RR within each prio queue
Prio. based on process type & execution history
Memory CPU generates virtual addresses, translations done by MMU
relocation, protection, sharing, logical/physical organization
Address translation bind variable names to physical locations.
1. Compile absolute code, no relocation
2. Load relocation table, can be relocated at start, not later
3. Execution executable contains logical addresses → physical
Fixed partitioning internal fragmentation **Dynamic** external
First, next, best, worst, quick-fit (keep mutiple free lists)
Paging
Logically partition physical mem into equal, fixed sized chunks (frames)
Divide VM into chunks of same size called **pages**
CPU uses **Page Table Base Register** to find start of page table
page# = vaddr / page_size, offset = vaddr % size
paddr = frame# * page_size + page_offset.
Translation Lookaside Buffer (TLB) implemented in hardware
Access Evicted Page OS ensures consistency
1. OS sets evicted page PTE as invalid, stores location of page w/ swap file in PTE
2. ⏱ process accesses, invalid PTE causes trap (page fault)
3. Trap runs OS page fault handler
4. Handler uses invalid PTE to locate page in swap file
5. Reads page into physical frame, update PTE, resume
All paging systems depend on locality
Principle of Locality processes only use some pages that need to be mapped.
OS evicts an old page for new ⏱ mem full,
has to be loaded from disk ⏱ referenced again → **miss**
Prefetch/Prepage try to predict future use of page
Belady's Algorithm optimal, replace page that will not be used for longest period of time.
Belady's Anomaly fault rate might increase ⏱ given more mem.
First-in-First-Out (FIFO) suffers from Belady's Anomaly
Least Recently Used (LRU) time stamp (large PLE), stack (costly)
vulnerable to scanning ref patterns.
Second Chance Clock frames arranged in circular list
clock hand selects good LRU candidate, sweep in circle
Ref off: page not used recently, turn on ⏱ used
low overhead ⏱ plenty memory, low accuracy
Simplified 2Q A_1 limited-size FIFO, A_m main LRU queue

On reference to page p
(equivalently, the frame allocated for page p):

```
if p is on the Am queue then
  move p to MRU position of Am
else if p is on the A1 queue then
  remove p from A1
  put p on Am queue in MRU position
else // first access we know about for p
  put p on A1 queue in youngest position
```

To allocate a frame for page p
(when all frames are in use):

```
if A1's size is above its threshold then
  evict oldest page from A1 (first-in)
  put p in the freed frame
else
  delete LRU page from Am
  put p in the freed frame
```

Page Buffering maintain pool of free pages

Kernel VM direct mapping, simple translation, simplifies hardware sup.
consume memory for lifetime of Virtual AS

Managing Swap Space
1. Raw disk partition : faster, requires disk reformat to size
2. Ordinary large file in FS : flexible, fragmented over time

Virtual Memory Area each process maintains an "address space"
VMA give way to track space process expects to use
Linux doesn't allocate mem on malloc(), mmap(), sbrk() until first use.
Seg fault page fault handler checks VMAs first to see valid address

Page replacement algo. should avoid **thrashing**
☹ more time is spent by OS paging
swap everything, kill some processes, buy more memory

Working set models dynamic locality of process' memory usage.

...2 6 1 5 7 7 7 7 5 1 6 2 3 4 1 2 3 4 4 4 3 4 3 4 4 4 1 3

7 only appears once

$WS(t_1,5) = \{1,5,6,7\}$ t_1-Δ t_1 $WS(t_2,5) = \{1,3,4\}$ t_2-Δ t_2

Shared mem allows processes to share data using direct mem references
PTE in both tables map to same physical, update both ☺ PTE invalid.
Diff: flexible, pointers invalid; Same: valid pointers, less flexible

Copy-on-Write defer large copies for as long as possible
Create shared mappings, shared pages are read-only
Writes protection fault, trap OS, copy page, change page mapping in PT, write permission, restarte write

Mapped Files mmap (addr, length, prot, flags, fd, offset)

File Systems long-term information storage
Store large amounts of info, info must survive termination of process using it, multiple processes must be able to access

User view convenient logical organization of info
OS view manage physical storage media, enforce access restrictions
Abstraction (files), permit sharing of data, organization (directory)
protect from unwanted access (security)

Creation find space in FS, add entry to directory, map file name to location & attributes
Writing overwrite/append, use "curr file position"
Reading dominant abstraction is "file as a stream"
Deleting Remove file, free storage space
Truncating may erase contents while keeping attributes

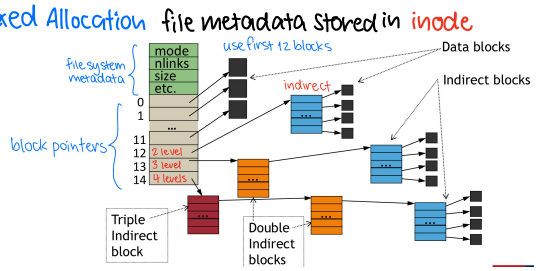
File Access Methods Sequential (read in order), direct (block/byte #)
Record (fixed/variable len), indexed

Directory list of entries; names & metadata
single-level, two-level, tree, acyclic-graph
1. List : slow 2. Hash Tables : space

Links pointer to another file / subdirectory
hard identical directory **symbolic** refers to file that holds true path to linked file

Allocation **Contiguous** continuous blocks, have start & end
Linked store start & end block, each block points to next file block

Indexed Allocation file metadata stored in **inode**



Disks block access devices. min disk block size is **sector size**
FS use **blocks** size must be multiple of sector size
Superblock holds metadata about overall file system
Free map tracks free blocks, bitmap
remaining are data blocks, metadata: inode, inode table, bmaps.

Extent-Based Allocation **extent** disk pointer + length (in # blocks)
(start, length) less metadata, more compact, less flexible

Link-Based Allocation inode just has pointer to first data
poor search through file

File systems need to be aware of disk characteristics
Request Scheduling to reduce seek time
Disks perform best if seeks reduced, large transfers used
1. Closeness put related things close together
2. Amortization grab lots of useful data at once

Problems 1. inodes at beginning of disk, far from data
2. fragmentation of aging FS causes more seeking

Berkely Fast File System (FFS)
cylinder group, disk must have free space scattered across cylinders

New Technonology File System tries to allocate files in runs of consecutive blocks, each volume has Master File Table (MFT)
each volume / partition is linear sequence of blocks
directory entry contains MFT index for file, length of file name, name

Virtual FS abstract file system interface
user level → kernal → VFS → actual FS

If crash between writes to disk, inconsistent data

fsck superblock, free blocks, inode state, inode links (count), duplicates, bad blocks, directory checks
only verifies consistency, too slow

Journaling write ahead what you are going to do first

TxBegin	I[v2]	B[v2]	Db	TxEnd
	write to inode	bitmap	data block	

split transaction logging into 2 steps using **barrier**
1. Write all blocks except Tx End to journal (write)
2. Write Tx End (commite) → safe final state — barrier
3. Write actual data & metadata (checkpoint) — barrier
4. Mark transaction as free in journal

If crashes, just redo checkpoint **redo logging**
Circular log, if cannot fit entire operation, split it
Transaction Batching batch multiple FS ops. into one transaction
commits on size limit, time limit, or explicit user rqst.
Faster recovery, slower normal op compared to fsck
May break sequential writing : journal writes & main region writes
Write data before writing metadata to journal

SSD use transistors, volatile on-device memory
after page is written, it has to be erased first before rewriting
limited endurance : need to distribute writes equally to all blocks

Flash Translation Layer (FTL) performs algos
Wear-leveling always write to new location
Garbage Collection reclaim stale page, create empty erase blocks
RAID 5 striping across I/O channels to multiple NAND chips

Authentication knows, has, is, does
Authentication / Access Control: restrict what (object) user can do, access, who, how (Subject action)
Lampson Access Matrix shows access control
AC List for each object, maintain list of subjects & permited actions (easier to manage)
Capabilities is for each subject