OS = ① Referee ② Illusionist ③ Glue

Kernel vs User

Kernel MANAGES	User MANAGES
- Process	User-level apps
- File system	Standard user-level code within restricted virtual addresses
- Memory	most shell commands
- Device	
system calls	
security & protection	

Hardware support

① Privileged instructions
↳ unsafe instr ≠ user mode
② Memory isolation → no access outside of process addr space
③ Interrupts → kernel gain control from running process
④ safe transfers → transfer from user mode ↔ kernel mode

File Descriptors

- points to OPEN file description in global table
- File description
  ① file offset
  ② file access mode
  ③ file status flags
  ④ ref to physical loc
  ⑤ # of times opened

two open calls ✱ created two different entries in FD table and points to diff file description in global table

✱ dup to duplicate FD to point to same file
✱ dup2 = replace existing FD

Default FD Table

(0) STDIN_FILENO
(1) STDOUT_FILENO ← printf writes here
(2) STDERR_FILENO

✱ fork = duplicated descriptors point to SAME entry in global file table

Pipes = ONE-WAY communication between two processes

pipefd[0] = read-only
pipefd[1] = write-only

Empty: blocks read
FULL: blocks write

✱ effectively closed after last write descriptor → reads until EOF
✱ uses a kernel buffer in MEMORY

Sockets = 2 queues: 1 in each dir

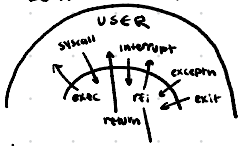
✱ bidirectional communication between processes

✱ can read/write to either end
✱ communicate between multiple processes on diff machines
✱ FD obtained w/ socket/bind/connect listen/accept

Kernel ↔ User

① Syscalls = user processes request certain kernel services
cannot trust
✱ SYNCHRONOUS event
✱ exec, read, write
✱ args pushed onto stack
✱ args validated in kernel memory

② Traps = software interrupts
exceptions → internal
✱ SYNCHRONOUS event
✱ exceptions



⑤ Hardware interrupts = external
✱ ASYNC event
✱ Timer, I/O

✱ OS transitions from kernel → user to enter signal handler

To handle interrupt:
① processor detects interrupt
② suspend user prog, switch to kernel
③ ID interrupt type, invoke handler
④ restore user program

HIGH / LOW API

LOW LEVEL ← (less overhead)

- direct use of syscall interface
- open(), read(), write(), close()
- open() → returns file descriptor
- FD meaningful to kernel
- read() or write() causes syscall
- call read in a loop
✱ write in while LOOP since not guaranteed to write all bytes out

FD API = immediate I/O

HIGH LEVEL

- buffered access: fopen() fread, fwrite, fclose BUFFERED!!
- opening returns FILE ▼ struct
- FILE has memory for buffer and FD for the file
- fread / fwrite filters through BUFFER
- loop to get bytes w/ fread

FILE ▼ = BUFFERED IO
↳ per-file user-level buffer
✱ operates on user space first

Threads = single execution sequence that can be separately schedulable

- inside same process: not isolated from each other
  - share address space
  - share I/O state
- individual execution
  - individual stack & register state

pthread_create = thread that runs start_routine
pthread_exit = terminates calling thread & returns value_ptr to any successful join call
pthread_join = suspends execution of calling thread until target thread terminates

Comparison category	Across Processes	Across Threads
creation	fork()	pthread_create()
page table	Distinct	same
registers/instruction ptr	Distinct	Distinct
stack	separate/inaccessible	separate BUT accessible
Heap/static vars	separate	shared
file descriptors	separate	shared
synchronization	wait() waitpid()	pthread_join() semaphores, locks
overhead	Higher	Lower
protection	Higher	Lower

Process = container where threads execute

Process = instance of running program

✱ process control block (PCB): stores
- process state
- PID
- open file list
- memory limits
- PTBR (ptr page table per process)

✱ executing program w/ restricted rights
✱ illusions of: private CPU, private memory (virtual address per process)
✱ resources allocated to each process
✱ isolate processes from all others

Process Management API

exit	terminate process
fork	copy current process as child ✱ allocate PCB w/ duplicated memory content ✱ new PID, same EIP child=0, parent>0 DUPLICATES: - address space - code/data - registers - stack - open file - file pointers ✱ same global FD as parent ✱ process ID and return value NOT copied
exec	change program currently running on process → REPLACES it (replaces code/data) ✱ EIP → start of new program, reinit SP, FP
wait	wait for process to finish
kill (& not just sigkill)	send signal (interrupt-like) to another process(es) ✱ can choose ANY SIGNAL to send
sigaction	set handlers for signals → can override default w/ signal handler execute signal handler when event has occurred

✱ SIGSEGV = seg fault
✱ SIGKILL can NOT be overridden

concurrency

- atomic operation = always runs to completion/not
  - ex: loads/stores of words
- synchronization = use atomic ops to ensure thread cooperation
- mutual exclusion = ensure only 1 thread does a thing @ a particular time
- critical section = piece of code that only ONE thread can execute at once
- race condition = when multiple threads or process access or modify shared resources at same time
  - lock() before entering critical section
  - unlock() when leaving
  - WAIT if locked → all sync involves waiting
  - ① Busy wait = rely on atomic loads/stores → DO NOT DO - consumes cycles
  - ② Enable/disable interrupts

test&set = simple lock that provides mutual exclusion w/o entry into the kernel

```
test&set(&address) {
    result = M[address];
    M[address] = 1;
    return result;
}
```

```
int mylock = 0
acquire(int *thelock) {
    while(test&set(thelock)); // busy wait
}
```

```
release(int *thelock) {
    *thelock = 0;
}
```

spinlock implementation = busy waiting so thread loops until lock is free

- atomic operations

Futex = similar to test&set in linux space

① futex_wait → check if condn still holds, sleep until we hear futex_wake (no more busy waiting)

② futex_wake → wake up at most val waiting threads

- user level to maintain queues for sleeping threads
- while loop to just consistently keep trying to regain lock & avoid race conditions
- for efficiency → want to minimize # of syscalls

```
int mylock = 0
acquire(int *thelock) {
    while(test&set(thelock)) {
        futex(thelock, FUTEX_WAIT, 1)
    }
}
```

```
release(int *thelock) {
    *thelock = 0;
    futex(&thelock, FUTEX_WAKE, 1)
}
```

- avoids syscalls when possible

Semaphores = lock w/ non-negative int value

Down() : wait for semaphore to be positive, ↳ decrement by 1 (like wait)

Up() : increment semaphore by 1 → wakes up waiting p

down on semphore of 0 = putting thread to SLEEP

Mutual Exclusion: value is 0 or 1 (like lock)

Scheduling : value can be >=0

```
typedef struct semaphore {
    unsigned value
    struct list waiters
} semaphore
```

```
sema_down(semaphore *sema)
    Disable interrupts
    While (semaphore value == 0)
        [sleep + schedule other thread/ up to will re-enable interrupts]
    Decrement semaphore value by 1
    Enable interrupts
```

```
sema_up(semaphore *sema)
    Disable interrupts
    [Wake up sleeping thread]
    Increment semaphore val by 1
    Enable interrupts
```

locks = prevent others from changing critical section

↳ can use internal semaphore, init to 1

```
lock_init(lock *lock)
    initialize holder of lock to NULL
    sema_init(&lock->sema, 1)
```

```
lock_acquire(lock *lock)
    sema_down(&lock->sema)
    Set holder of lock to caller
```

```
lock_release(lock *lock)
    reset lock holder back to NULL
    sema_up(&lock->sema)
```

acquire(&lock): wait till lock is free, then grab, run critical section

release(&lock): unlock, wake up anyone waiting

Implementation:

Acquire():
- disable interrupts
- if (val == BUSY)
  - put thread onto queue
  - go to sleep
- else: value = BUSY
- enable interrupts

Release()
- disable interrupts
- if anyone on wait queue
  - take thread off queue
  - Place on ready queue
- else: value = 0
- enable interrupts

Mutual Exclusion

① init sema = 1

② down sema in critical section

③ up sema when exit critical

SCHEDULING

① init sema to 0

② down sema in waiting thread

③ after work, up sema in working thread

conditional variables = queue of threads waiting for stn inside critical section

AVOID BUSY WAITING

- allows sleeping inside critical section by releasing lock at time we go to sleep
- purpose: threads can access/modify shared data w/ certain condition

```
typedef struct cond_var {
    struct list waiters
} cond_var
```

need lock before cond_wait, signal, broadcast

```
cond_wait(cond_var *cond, lock *lock)
    create semaphore
    sema_init(&waiter_sema, 0)
    push waiter's semaphore into waiter queue
    lock_release(lock)
    sema_down(&waiter_sema)
    lock_acquire(lock)
```

atomically release lock & suspend calling thread execution

```
cond_init(cond_var* cond)
    initialize cond's list of waiters
```

```
cond_signal(cond_var *cond)
    if waiters queue not empty
        pop waiter off
        execute sema_up
```

wake up next waiting thread

- only wakes up ONE waiting thread

```
cond_broadcast(cond_var cond)
    while waiters queue not empty
        pop waiter off
        execute sema_up
```

wake up ALL threads

Monitor = lock & zero+ cond variables for managing concurrent access to shared data

- uses infinite synchronized buffer
- must hold lock when doing cond var ops

HOARE

```
if (isEmpty(&queue))
    cond_wait(&buf_CV, &buf_lock)
item = dequeue(&queue)
```

IF STATEMENT

- check once if coke available

Hoare = when thread signals cond var, the waiting thread immediately takes control of lock & starts executing

✓ strong guarantee that cond is true when waiting thread wakes up

✗ dependent on OS scheduler

MESA

```
while (isEmpty(&queue))
    cond_wait(&buf_CV, &buf_lock)
item = dequeue(&queue)
```

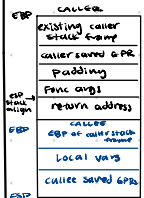
WHILE LOOP

- more practical
- repeatedly check for coke

MESA = when thread signals cond var, waiting thread moved to ready queue but NOT run immediately

- no guarantee abt execution order
- have to keep checking cond

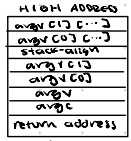
calling convention



CALLER Before calling function:
① Save caller-saved GPRs (EAX, ECX, EDX) onto stack if needed AFTER func call
② Push param onto stack in REVERSE order
Add padding BEFORE param to ensure 16 byte alignment
③ CALL FUNCTION (esp=16 byte aligned on) and push return address & jump to func

CALLEE
① Push EBP onto stack, set new EBP = ESP
② Allocate stack space for vars
③ Save callee saved GPRs (EBX, EDI, ESI) if used
④ Perform func logic
EPILOGUE (before returning)
⑤ Store return value in EAX
⑥ Restore callee saved GPRs (if any) from prologue
⑦ De allocate local vars
⑧ Restore caller's EBP from stack
⑨ Return from func call by popping caller return addr & jumping

CALLER Epilogue after func returns
① remove params from stack
② restore caller-saved GPRs (if any)



- stack grows DOWNWARD
- ESP = stack pointer

Scheduling = deciding which threads to give access to

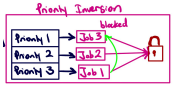
Starvation = thread fails to make progress indefinitely

↓ **completion time** = waiting time + run time of process
- ✱ time sensitive like I/O
- wait time = total time on queue

↑ **throughput** = rate at which tasks completed (tasks/sec)
- ↓ overhead (context switching)

↑ **fairness** = sharing resources in equitable manner

Choose	for
FCFS / FIFO	CPU Throughput
SRTF Approx.	Avg Response Time, I/O Throughput
Linux CFS	Fairness (CPU Time)
Round Robin	Fairness (CPU Not Time)
EDF	Meeting Deadlines
Priority	Favor important



Priority Inversion = high priority task BLOCKED waiting on low priority task → medium runs

For RR
1. At end of quantum X, add prev thread to ready queue
2. Add any new threads arriving at quantum X+1 to ready list

POLICIES

First Come, First Served (FCFS/FIFO)

✱ schedule in order of arrival → one program scheduled until done
✓ simple, good for throughput (↓ context switching)
✗ **CONVOY EFFECT** (short tasks stuck behind long)
variable avg completion time

Shortest Job First (SJF) NON PREEMPTIVE

✱ schedule shortest task first
✓ optimal avg completion time (simultaneous arrival)
✗ subject to starvation, convoy effect
have to know job duration

Shortest Remaining Time First (SRTF) ↑ PREEMPTIVE

✱ PREEMPT resource if task arrives & has SHORTER completion time
✓ always optimal avg completion time, NO convoy effect
✗ starvation, have to know job duration

Generalized Processor Sharing

- infinitely fine grained context switching where to share processor among multiple threads

ideal service time = time that task i should receive

$S_i(t_0,t_1) = f_i(t_0) * (t_1 - t_0)$

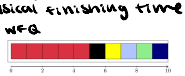
real service time = time that task i actually receives

lag = ideal − real service time

$f_i(t) = \frac{w_i}{\sum_{j \in A(t)} w_j}$ ← weight; set of active threads

proportional share of processor that thread i should receive at time t

weighted fair queueing = schedule threads in order of EARLIEST physical finishing time



EEVDF = earliest eligible virtual deadline

$\frac{dV}{dt} = \frac{1}{\sum w}$

$V_e = V_{d-1} + lag$

$V_d = V_e + \frac{r}{w}$

used for rescheduling

scheduling (eligible time ✓, virtual time ✓)
1. look @ threads w/ eligible requests v(e) ≤ v(t)
2. serve request w/ earliest virtual deadline v(d)
3. reschedule using the V_e, V_d equations if thread finished

$t=0$ to $t=1$: $\frac{dV}{dt} = \frac{1}{w_1}$

$t=1$ to $t=2$: $\frac{dV}{dt} = \frac{1}{w_1+w_2}$ ← use this to calculate virtual time V(t)

✱ requests made in real time not virtual time
✱ when NEW thread becomes active, set V_e = curr virtual time
✱ lag = ideal − real

POSITIVE LAG = EARLIER next eligible time | NEG LAG = LATER eligible time since it took so much CPU time

✱ If lag: $V_{e,k+1} = V_{ek} + \frac{U^k}{w_i}$ ← time thread took to complete k+1 request

Round Robin (RR) = run processes in looping order for fixed quanta

slice of CPU time

✱ after q expires: task preempted & added to end of ready queue
- more context switching = more cache misses

LARGE q = ~FCFS vs SMALL q → lots of interleavings

✓ no starvation, w/ n process, q time quanta, $wait_{max} = (n-1)q$
✗ lots of context switching, HIGH completion time

Multi-Level Feedback Queue (MLFQ): multiple queues w/ diff priorities

✱ Each queue has own scheduling policy
✱ ensure CPU doesn't hog all
IO priority > CPU priority

enters at highest priority
use up resource = move DOWN
does not use all = move UP

High → RR
low → FCFS

✓ Approximates SRTF
✗ Gaming strategy: request I/O to stay on higher priority

Lottery Scheduling: give each task some # of lottery tix

✱ draw random ticket at each time slice → task holding ticket is granted resource
✓ No starvation, no priority inversion, approximate SRTF w/ more tickets to shorter jobs, fewer tickets to longer jobs
✗ unfair over short run

Stride Scheduling = deterministic lottery scheduler

✱ Each task given some # of tickets n_i
✱ Stride = $\frac{w}{n_i}$ ← large # / # tix — larger share of tix = smaller stride

On every time slice: choose task with lowest PASS
init pass = min(existing tasks' pass vals) at start of task
↳ when task chosen: pass += stride

✓ No starvation
↳ smaller stride → task runs more often

Strict Priority = schedules tasks w/ highest priority

✓ important runs first
✗ starvation, priority inversion → priority donation where lower task given same EFFECTIVE priority as higher one

Deadlock = cyclic waiting for resources

deadlock → starvation, starvation ≠ deadlock

Safe space: can prevent by delaying acquisition
Unsafe space: can unavoidably lead to deadlock, with certain acquisition
Deadlocked state: exists a deadlock

Requirements for Deadlock

1) Mutual Exclusion and bounded resources
   - one thread at a time use resources
2) Hold and wait
   - thread holding resource waits to acquire more
3) No preemption
   - resource released voluntarily
4) Circular Wait
   - set of waiting threads waiting on each other

Deadlock Prevention

1) Provide sufficient resources, VM unlimited
2) Abort requests, acquire atomically
3) Fail if waiting too long, force give up
4) Order resources usage in same order

Necessary but NOT sufficient

Handle deadlock

1. Denial (ostrich-pretend it doesn't exist)
2. Prevention - write code that doesn't deadlock
3. Recovery - let deadlock happen & recover afterwards
4. Avoidance - dynamically delay resource reqs so no deadlock
   * cannot check for deadlock → check for unsafe state

* Draw out current, available, needed tables

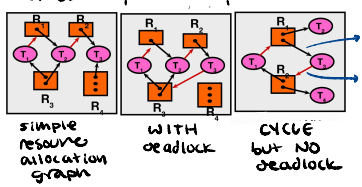
Banker's Algorithm

1. check if request → unsafe state
2. state max resource needs in advance
3. allow thread to continue if available resources − # requested ≥ max
4. release resources
   available += allocated [thread]

add to unfinished
for each thread unfinished
if (Request [Max − Alloc] ≤ Avail)
remove from unfinished
Avail = Avail + Alloc

regular algo checks if request[thread] ≤ available

* pretend request is granted → can we enter deadlock?
  ↳ if yes: deny/hang
* if unfinished NOT empty → deadlock is POSSIBLE



Demand Paging = keep only active pgs in memory, place others on disk, PTE = invalid

Working set = subset of addy spaced used during execution

Resident set = subset of addy space held in memory

THRASHING = memory too small for working set → LOTS of replacements & context switches
n · RSS > PHYS_MEM

Memory + Addressing

Numbers

1 B = 8 bits
1 KB = 2^{10} B
1 MB = 2^{20} B
1 GB = 2^{30} B
1 TB = 2^{40} B

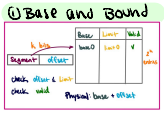
Time

1 ms = 10^{-3} s
1 µs = 10^{-6} s
1 ns = 10^{-9} s
1 ps = 10^{-12} s

Virtual memory:
- protection
- translation
- efficiency

PAGE FAULT = attempt to access virtual NOT in physical
invalid PTE
1. MMU traps to OS w/ page fault
2. Find & replace pg w/ pg from disk w/ policy
3. reset pg table, restart instruction

① Base and Bound



base register = starting address
bound register = boundary of memory segment

✓ simple, can easily relocate
✗ internal AND external fragmentation
Difficult interprocess sharing
Unused space in memory
Unused space in segment

* w/ hardware relocation:

physical address = virtual address + base
use MMU = hardware as memory management unit

Segmentation = place segments independently

* READ-ONLY can be shared, use segment map
✓ minimizes internal fragmentation
✗ external fragmentation still problem

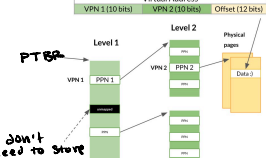
Pg replacement policies

1) FIFO: evict oldest page — NOT STACK
   Con: evicts heavily used pages
2) RANDOM: choose random page for replacement
   Con: unpredictable
3) MIN: replaces page not used for longest time, optimal
   Con: don't know future — STACK
4) LRU: replace page not used for longest time
   Con: too much overhead — STACK

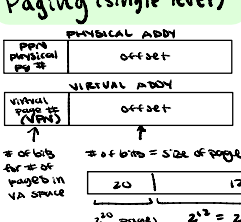
Belady anomaly: increasing cache can INCREASE miss rate → applies to NON STACK

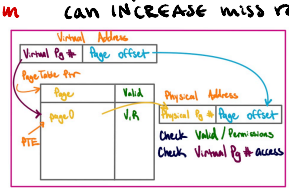
Multi level page tables

- sparse addy space = large pg table w/ lots of unused entries
- TREE of pg tables where each level maps to other pg table
- BOTTOM page table = translation to physical address



Paging (single level)





LARGER pages = fewer metadata
SMALLER pages = minimize internal fragmentation

Copy on write

1. Mark all pgs read only
2. copy pg table to new proc's)
3. once either attempt write
   ↳ PAGE FAULT
4. page fault handler → make physical copy → install new pg in pg table of triggered process

Translation Lookaside Buffer = cache for virtual-physical memory limitations

TAG	DATA	
VPN	PPN	metadata

PTBR = page table base register = starting addy of pg table
↳ PHYSICAL address

AMAT = (Hit rate × hit time) + (miss rate × miss time)

Improve context switching
1. Flush TLB
2. include PID in TLB

Page walk

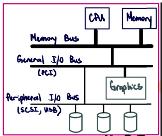
PTBR
1st PT: Addr: Base + $\text{Index}_{L_1} \times$ Entry Size
2nd PT: Addr: PPN_{L_1} (look in phys mem) || $\text{Index}_{L_2} \times$ Entry Size (concatenate)

* check metadata: convert to binary and check each bit
Phys addr: PPN_{L_2} (look in phys mem) || offset (concatenate)

# of virtual address bits = (# VPN bits) + (# offset bits)
physical address bits = (# PPN bits) + (# offset bits)
virtual pages = $2^{\wedge}$(# VPN bits)
physical pages = $2^{\wedge}$(# PPN bits)
Pg Size (bytes) = $2^{\wedge}$ (# offset bits)
virtual mem size (bytes) = (# virtual pgs) × (pg size B)
physical mem size (B) = (# physical pgs) × (pg size B)
pg table entries = # virtual pgs
PTE bits = (# PPN bits) + (# metadata bits)
PTE size (B) = # (PTE bits) / 8
Page table size (B) = # PTE × PTE size (B)

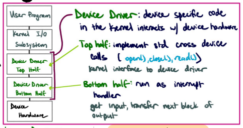
I/O

Access patterns:
1. **character** dev = character stream
   * NOT addressable
2. **Block** dev = fixed size blocks
   * YES addressable
3. **Network** dev = separate interface for networking



3 I/O registers
1. status
2. command
3. data

Access Timing
1. **synchronous/blocking** = WAIT until I/O request is fulfilled
2. **Non-blocking** = NO wait, quickly return
3. **Asynchronous** = allow other processing w/o waiting
   ↳ ≠ non-blocking



Block devices	Character Devices	Network Devices
- Access blocks of data, fs open(), read(), write(), seek() disk drives, DVD-ROM, raw I/O	- single chars at a time get(), put() keyboard, mice, USB	- diff from others, pipes, stream sockets, select() ethernet, wireless, bluetooth

response time = latency = time to complete

throughput = bandwidth = rate of tasks performed

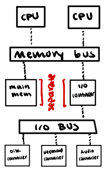
Device Drivers = connect I/O hardware
↓
* standard interface → OS high-level abstractions

$1 KiB = 2^{10}$
$1 MiB = 2^{20}$
$1 GiB = 2^{30}$
$1 TiB = 2^{40}$

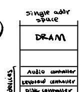
TOP half = start I/O operations
BOTTOM half = service interrupts
(flipped in Linux)

BUS = wires for comm/connecting n devices + protocols for data transfer
✓ cheap + easy to connect ✗ one at a time → arbitration logic for who gets control

PCIe I/O bus interface for high-speed, not parallel, fast serial channels
✓ Network cards, SSDs, graphics cards ✗ mouses, keyboards, USB



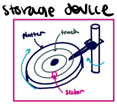
Device controllers = how processors talk to I/O devices
ways to interact w/ controller
1. **Programmed I/O (PIO)** = processor involved in every byte transfer
   - (A) **Port-mapped I/O (PMIO)** * separate bus
     * special privileged CPU instructions (in, OUT)
     * distinct memory address space from phys mem
       ↳ good for limited physical space since its part of physical address space
   - (B) **Memory-mapped I/O (MMIO)** (load, store)
     * standard load/store → responsibility on controller
     * NO physical address space → SHARED w/ physical memory



2. **Direct mapped access (DMA)** device can write DIRECTLY to main memory w/o CPU intervention
   1. CPU sets up DMA request
   2. Give DMA controller access
   3. DMA interrupts when done → interrupts OR poll

✓ Good for LARGE amts of data

Storage Devices
1. **Hard Disk Drive (HDD)**: magnetic disk storage device
   ✓ persistent memory → stays even computer is off
   * data stored in fixed size blocks = sectors
   ↑ sequential access ↓ random access



Request time: queuing + controller + seek + rotational + transfer
- queuing: software queue in device driver
- controller: hardware controller
- seek: position head/arm over track
- rotational: sector rotating
- transfer: transferring block of bits

* avg rotational latency = $\frac{1}{2}$ × Time for full rotation
* transfer time = $\frac{\text{Data size}}{\text{Transfer rate}}$ * throughput = $\frac{\text{sector size}}{\text{request time}}$
* Disk intelligence: track skewing (stagger sectors by track time) sector sparing (remap bad ones to spare), use buffer memory to store sectors read by disk head

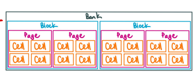
EXAMPLE
avg seek = 8 ms
rot = 7200 rpm
data transfer = 50 MiB/s
sectors = 4 KiB

Expected throughput
1. seek = 8 ms
2. rot time = $\frac{1}{2} \times \frac{1}{7200}$
3. transfer time = $\frac{4KiB}{50 MiB}$
4. Throughput = $\frac{4KiB}{8 + rot + transfer}$

* when head already in right spot can ignore seek time + rot time
* already on right track ignore seek time

data read w/ head, platter = 2 sides
✓ cheaper than flash ✗ slower than flash

2. **Flash memory** = electrical circuits (NOR, NAND) for persistent storage
   * can erase fixed # of times
   * request time = queuing + controller + transfer
   * writing to cell requires ERASING it
   ✓ faster read/write w/ pages



3. **Flash translation layer** map logical flash pgs → phys pages
   * indirection & copy on write to avoid wear out
   * wear levelling to spread out writes

write to new location → update mapping in FTL → erase old pg in background

File Systems = transforms block interface of disks into files/dir

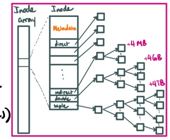
* abstraction layer maintained by OS, file has → metadata (size, owner, access control) → data (actual content)

Directory = list of mappings from human-readable file names to specific underlying files <name: file number> * majority small files

Hard links = map DIFF names to SAME file #
↳ ln [target] [dest] <name: file #>
✓ 1 file = many names
✗ cannot span across file systems
* cannot call read on directory

Soft links = directory entries that map name to another name <name: path>
↳ ln -s [target] [dest]
✓ move across file systems

File System Designs
1. **Fast File System** = multi level tree structure
   ✓ Block group (directory + files in common group)
   ✓ Bitmap allocation for free
   ✓ reserves 10% for efficiency
   ✓ Skip sectors to interleave other data in between sequential accesses
   * file # index → set of inode array
   * inode = file w/ metadata, ptrs to data
     ↳ NO file name/data in inode
   * FD entry → OPEN file descriptor (✓ duplicates)
   * ptrs = BLOCK numbers
     - Direct ptr = point directly to block w/ file data
     - Indirect ptr = point to array of direct ptrs
     - Double indirect ptr = point to array of indirect ptrs
   * i number = INDEX of inode in inode array (used as file #)



Pro: efficient for small/large files, locality, sequential, random access, no external frag
Con: inefficient for tiny, contiguous, reserve 10% space

Example
2 KiB blocks, 32 bit disk, file block ptrs: 12 direct ptrs, 1 indirect ptr, 1 double indirect ptr, 1 triple indirect ptr

#of ptrs = $\frac{\text{Block size (bytes)}}{4 \text{ (bytes)}}$

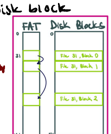
max disk size = $2^{\text{disk size}}$ blocks × block size ($2 KiB = 2^{11}$)
$= 2^{32} \times 2^{11} = 8 TiB$

max file size = block size × # of (all) direct ptrs (including from indirect)
$= 2^{11} \times (12 + 512 + 512^2 + 512^3)$ (dir, indir, double indir, triple indir)

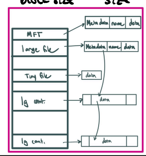
ptrs/block = $\frac{\text{Block size}}{\text{Ptr size}} = \frac{2048 B}{4 B} = 512$

✓ sequential, random, no external frag, big files, locality
✗ internal frag

2. **File Allocation Table** = array where each element corresponds to disk block (NOT INODE BASED)
   * File → collection of disk blocks
   * FAT → linked list one on one w/ blocks
   * index of first entry of file → set to file #
   NO HARD LINKS; metadata stored separately
   ✗ File size encoded in 32 bits → size limited to 4 GiB
   ✓ sequential, no external frag, big files
   ✗ random, internal frag, locality
   Entry = 0 = FREE
   ✓ in lots of portable & embedded devices, compatible
   ✗ allocating multiple sectors takes long time
   ✗ might not be able to exploit spatial locality
   ↳ quick format = mark all blocks as free
   full format = quick format + zero out data blocks
   zero out # FAT entries = $\frac{\text{FAT volume size}}{\text{block size}} \times \frac{\text{entry}}{\text{size}}$
   ENTRY → 1) index of next cluster → 2) special (end of file, free cluster, bad cluster)



3. **NTFS** = master file table, entries = file metadata + data
   - variable size extents w/ variable length (NOT fixed size blocks like FFS)
   - Big → use extents super big: use ptrs to lists of extents
   ✓ supports journaling



Buffer Cache = memory to cache disk blocks/name translations
implemented in OS software w/ LRU replacement policy
- Read ahead prefetching = fetch sequential blocks early
- Delayed writes = flush to disk → either @ failure & periodically

NOT write-through which immediately puts on disk

Reliability
(durability ⇏ availability) (imply)

availability = probability that system can accept & process requests
"nines" of probability: 99.9% = 3 nines

durability = ability of system to recover data despite faults / fault tolerance

small defects = Reed-Solomon error correcting codes (ex single disk failure)

Short term data preservation → battery backed RAM (NVRAM) (ex for buffer cache)
Long term data preservation → redundant array of inexpensive disks (RAID)

Redundant Arrays of Inexpensive Disks (RAID): Reliable Disk Storage
RAID 1: Disk Mirroring/Shadowing: disk fully duplicated
RAID 5+: High I/O Rate Parity: Data stripped across multiple disks
RAID 6: allow 2 disks in replication stripe to fail

reliability = ability of system to perform its required functions under conditions for specific period of time (FAT & FFS) + fsck

careful ordering & recovery: break into small chunks, write in specific order
recover scans looking for incomplete actions ✗ NOT foolproof, mostly works

versioning & copy on write (ZFS): write new versions of data into NEW place on disk NOT updating in place → update mappings/reference
✓ easy recovery bc old data still intact
✓ data only being added

CREATING A FILE

careful ordering (FFS)	Recovery (fsck)
1. Allocate data block 2. write data block 3. Allocate inode 4. write inode block 5. update free map 6. update directory entry 7. update modify time for dir entry	1. Scan inode table 2. If any unlinked (not in dir) delete & put in lost & found 3. compare free block bitmap against inode trees 4. Scan directories for missing update/access times

Transactions = atomic sequence of operations (→ so we never in-between)

consistent state → consistent state (updated disk w/ transactions)

* if any fail → roll back
  ↳ otherwise, commit!

NOT indiv read/write

Atomicity
Consistency (1 valid state to another)
Isolation (concurrent is ok)
Durability (once committed won't be erased)

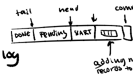
each entry = IDEMPOTENT
↓ same effect when executed once/many times $f(f(x)) = f(x)$

Methods: (BOTH USE LOG = CIRCULAR BUFFER ON DISK)

① **Journaling Filesystem** — * maintain in JOURNAL before committing

① Preparation phase = append all updates to log
② commit phase = log commit record
③ write back = async apply committed update
④ garbage collect any completed transaction in background

commit entry ↓ makes it atomic



* write to log instead of disk directly
* once WHOLE transaction written to log
  - disk applies changes
  ↓
  remove transaction from log

✗ not exact snapshot
✓ can roll back

* if crash when writing → incomplete, NOT applied
* if crash when applying → re-apply transaction after recovery

② **Log Structured File System (LSFS)**: log IS storage
write everything sequentially — * ALL DATA IN LOG FORM *
✓ writing random sectors - better than normal

Distributed Decision Making

General's Paradox = impossible to achieve simultaneous acknowledgement over UNRELIABLE network

Two Phase commit = decide if all processes commit/abort transaction eventually → 1 coordinator, rest participants

① coordinator asks all processes to vote: VOTE-REQ
② Participants vote VOTE-COMMIT / VOTE-ABORT, log
③ If ALL vote-commit → GLOBAL-COMMIT
   otherwise → GLOBAL-ABORT, log
④ participant commit or abort on receive, log
   ↓
   respond w/ ACK

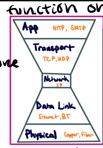
* If coord CRASH before (logging global commit)
↓
Lose All votes
↓
abort

Failure:
→ any participant error → coord votes ABORT
→ if all voted commit → wait on coord to recover

when coordinator logs → becomes source of truth
persistent decision

* does NOT solve General Paradox → still uncertainty

Internet = allow apps to function on all networks



Packet = basic unit of communication
↳ chunk of data, some metadata, source destination IP

Transport layer on top of packets

UDP = best effort delivery

TCP = reliable in-order, but MORE overhead

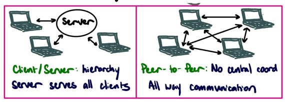
end to end: functions should be implemented at endpoints of communication system - NOT in middle
* implement at lower layer only as enhancement

Distributed Systems = coordinating multiple computers

Scalability: add resources to system to support more work

Transparency: mask complexity behind simple interface

Protocols: agreement on how to communicate (syntax/semantics)



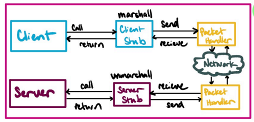
Remote Procedure Call = execute code on remote machine in simple way

Serialization: express object in byte sequence

Big/little Endian: first bit in addy = most / least sig bit

Marshalling: converting values to canonical form (serializing)

Binding: converting user-visible name to network endpoint
↳ dynamic binding allows server flexibility



STUB = glue on client/server

① client stub = marshalling/serializing args
   unmarshalling/deserializing return values
② server stub = unmarshalling/deserializing args
   marshalling/serializing return values

Distributed File Systems

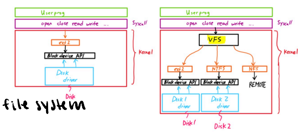
① **Virtual Filesystem Switch (VFS)** = interface between kernel's view of files & underlying file system (-AT, FFS, NFS ...)

* SAME system call interface API for diff types of file system

* VFS like translator for file systems *

4 object types
① superblock object = specific mounted filesystem
② inode object = specific file
③ dentry object = directory entry
④ File object = open file associated w/ process

* Distinguish between remote & local files
OLD / NEW



② **Network File System (NFS)**

Translates read/write VFS calls → RPCs for remote file system

* STATELESS *: RPC has all info to ID a file
readAt(iNumber, position) vs Read(openfile)
✓ scalability

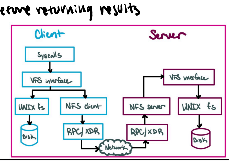
* IDEMPOTENT * re-read / re-write blocks w/o side effects

* weak consistency * client polls server periodically → client may have stale or partial updates

* write through caching * modified data committed to server's disk before returning results

want sequential ordering

✓ simple, portable, efficient
✗ sometimes inconsistent / doesn't scale



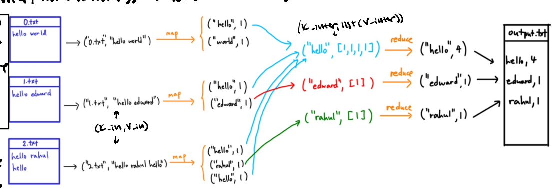
MapReduce & Spark

MR = model that distributes tasks over large clusters

map: (k_in, v_in) → list(k_inter, v_inter)
reduce: (k_inter, list(v_inter)) → list(k_out, v_out)

1 coordinator, many workers

MAP REDUCE STEPS
① convert files into pairs of (k,v)
② Define map function → apply to all files
③ Shuffle! All elements w/ same key go to same reduce
④ Define reduce function → apply to result of map



Reading /a/b/c.txt w/ FFS file system
find file # of /
① read /'s inode
② read /'s data page
③ read a's inode
④ read a's data page
⑤ read b's inode
⑥ read b's data page
↓
dir entry has inode for c.txt

6 accesses

To read 165 KiB give c.txt file#

$\frac{165\ KiB}{Block\ size} = \frac{165\ KiB}{4\ KiB} = round(41.25) = 42$

31 + 11 + 1 + 1 = 44
① read c's inode (1)
② read 11 direct data blocks (11)
③ read indirect block (1)
④ 31 remaining data blocks (42 - 11 = 31)

Fault Tolerance = large cluster w a lot of failures
① coordinator fails → ABORT
② Reassign in progress or completed map tasks if mapper fails
③ Reassign in progress reduce tasks if reducer fails
↑ check for completed map

Apache Spark = how to reuse intermediate results
→ cache instead of recomputing

** Resilient Distributed Dataset (RDDs)**: immutable collections of objects in memory
- Transformations: apply func on RDD → make another RDD
  ↳ lazily evaluated w/ iterator → replay transformation to regenerate
- Actions → return value to program / write data to disk

** Lineage: logged sequence of transformations to create RDD

** Failures: Node fails → lost RDD partition → recompute w/ lineage