CG2271 Cheatsheet by Luke Aidan Tan

Lect 1: Introduction to OS

OS role: Manages hardware resources; provides abstraction + protection + sharing for user programs. Acts as intermediary between hardware and applications.

	Monolithic OS	Microkernel OS
Structure	One big special program; all services integral	Very small, clean core; everything else as services
Internal Services	Built-in; software engineering is maintained, modularized	Basic facilities (IPC, memory, scheduling) in kernel; rest as high-level services
Advantages	Well understood; good perf; folder comms; kernel crash stays alive	More robust; better isolation/ protection; kernel services stay alive
Disadvantages	Highly coupled components; very complicated internal structures	Lower performance due to overhead of IPC and service communication

OS layers (bottom → top): Hardware → Device Driver / OS (IPC, Process Mgmt, Memory Mgmt, File System, Interrupt Handler) → Library → User Programs.
Execution modes: User Mode — normal program execution. **Kernel Mode** — privileged OS execution. Switch via TRAP (system call), exception, or interrupt.

OS Services

Process Mgmt: create/terminate processes, scheduling, IPC, synchronization. **Memory Mgmt:** allocate/ deallocate, virtual memory. **File System:** files, directories, permissions. **I/O:** device drivers, buffering. **Protection & Security:** enforce access control.

Lect 2: Process Abstraction

A **process** (task/job) is a dynamic abstraction for an executing program. Described by three contexts:

Context	Contents
Memory	Text (instructions), Data (globals), Stack (function frames), Heap (dynamic alloc)
Hardware	GPRs, PC, SP, FP, PSW
OS	PID, Process State, resources used

Memory Layout

Text: machine instructions. **Data:** global/static variables. **Stack:** grows downward; holds stack frames for active function calls. **Heap:** grows upward; dynamically allocated memory (`malloc`/`new`); variable size & lifetime → needs separate region.

Stack Frame

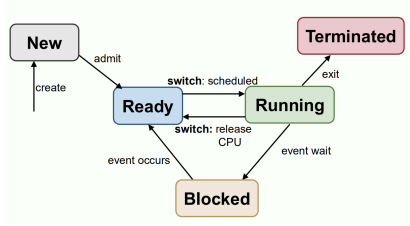
Each function invocation creates a **stack frame** containing: Return PC · Parameters · Local Variables · Saved FP · Saved registers. **SP** points to top (first unused location). **FP** points to fixed location in frame for stable displacement access.
Setup (caller → callee): caller pushes args + return PC → transfer control → callee saves old SP/FP → callee allocates locals → adjusts SP.
Teardown (callee → caller): callee places return value → restores saved FP/SP → transfers control via saved PC → caller continues.

Process Control Block (PCB) & Process Table

OS maintains one **PCB** per process (= Process Table Entry) storing full execution context.
Process Table = collection of all PCBs.
PCB fields: PID · Process State · PC, FP, SP · GPRs · Memory Region Info (pointers to Text/Data/Heap/ Stack).

Process States (5-State Model)

New: initializing, not yet ready.
Ready: waiting to be scheduled.
Running: executing on CPU (≤1 per CPU).
Blocked: waiting for event (I/O, syscall).
Terminated: finished, awaiting OS cleanup.
Queuing model: ready queue → (scheduler) → CPU → exit / blocked queue → (event) → ready queue.



System Calls

API from user → OS kernel. NOT a normal function call — requires mode switch (User → Kernel) via **TRAP** instruction.
Mechanism: user calls library wrapper → library places syscall number in register → executes TRAP → kernel dispatcher looks up handler → handler executes → returns to user mode → library returns to caller.
Unix (100 syscalls, POSIX);
Windows (1000 calls, Win32 API).

Exceptions & Interrupts

Exception: synchronous; caused by executing instruction (div-by-zero, illegal memory access, overflow).
Interrupt: asynchronous; caused by external hardware event (timer, keyboard, mouse).
Handler steps: Save CPU/register state → perform handler routine → restore state → return (resume program).

Lect 3: Process Scheduling

Scheduling problem: choose which ready process runs next. **Scheduler** = part of OS that decides. **Dispatcher** = mechanism that performs the context switch.
Context switch: save current process context (PCB) → load next process context. Overhead: pure overhead (no useful work during switch).

Scheduling Criteria

All environments: Fairness (fair CPU share, no starvation), Balance (utilize all system parts).
Batch: throughput, turnaround time, CPU utilization.
Interactive: response time, proportionality.
Real-time: meet deadlines, predictability.
Turnaround time = completion − arrival
Waiting time = turnaround − execution
Response time = first time task receives CPU time − arrival

Batch Scheduling Algorithms

FCFS (First-Come First-Served): non-preemptive; simple queue. Convoy effect: short jobs stuck behind long ones.
SJF (Shortest Job First): non-preemptive; minimum average waiting time (provably optimal for batch). Requires knowing burst time.
SRTF (Shortest Remaining Time First): preemptive SJF; preempts if new job shorter than remaining time.

Interactive Scheduling

Round Robin (RR): FIFO queue, each process gets one **time quantum** q then preempted. Large $q \to$ FCFS behavior; small $q \to$ high context-switch overhead. Typically $q = 5$–100 ms.
Priority Scheduling: each process has priority; highest priority runs. **Preemptive** version preempts lower-priority running process. Risk: starvation of low-priority → fix with **aging** (raise priority over time).
Multi-level Queue: separate queues per priority class; processes don't move between queues.
Multi-level Feedback Queue (MLFQ): processes can move between queues based on behavior (CPU-bound → lower priority; I/O-bound → higher priority).

Real-Time Scheduling

Tasks characterized by **Period** P and **Execution time** C. Schedulable iff $\sum \frac{C_i}{P_i} \le 1$.
Rate Monotonic Scheduling (RMS): fixed priority; shorter period → higher priority. Optimal among fixed-priority. Liu & Layland bound: $U \le n\left(2^{\frac{1}{n}} - 1\right)$ ($\to$ 69.3% as $n \to \infty$).

$B(1) = 1.0$	$B(4) = 0.756$	$B(7) = 0.728$
$B(2) = 0.828$	$B(5) = 0.743$	$B(8) = 0.724$
$B(3) = 0.779$	$B(6) = 0.734$	$U(\infty) = 0.693$

Earliest Deadline First (EDF): dynamic priority; closest deadline → highest priority. Optimal (can achieve 100% utilization).

Critical Instance Analysis (RMS)

Sort tasks by period. For each task i: compute $S_{i,k} = 1 + \sum_{j<i} \left\lceil \frac{S_{i,k-1}}{P_j} \right\rceil \cdot C_j$. Stop when $S_{i,k} = S_{i,k-1}$; schedulable if $S_i \le P_i$.

Lect 4: Threads

Thread = lightweight process; unit of execution within a process. Multiple threads share the same process (Text, Data, Heap, OS resources) but each has own **hardware context** (PC, SP, FP, registers) and **stack**.
Benefits: Economy (cheaper create/switch than process); Resource sharing (no IPC needed for shared data); Responsiveness (one thread blocks, others continue); Scalability (exploit multi-CPU).

	Process Context Switch	Thread Switch (same process)
Saves	OS + Hardware + Memory context	Hardware context only (registers, SP/FP)
Cost	High	Much lower

Thread Implementation Models

Model	Pros	Cons
User Thread	Flexible; portable; fast switch (no kernel), runs on any OS	One thread blocks → all block; can't use multiple CPUs
Kernel Thread	True parallelism; one blocks → others run, can schedule simul across multiple CPUs	Slower (syscall overhead); less flexible, expensive and overkill if implemented with many features
Hybrid	Both kernel & user threads; great flexibility	Complex; Solaris-style implementation

POSIX Threads (pthreads)

```
// Create
pthread_create(&tid, NULL, startFn, arg);
// Wait for termination
pthread_join(tid, &status);
// Exit (from within thread)
pthread_exit(exitValue);
```

Memory Sharing Risk

Shared variables accessed concurrently are **not** atomic. E.g. `globalVar++` = read → modify → write-back. A thread switch between steps → **race condition** → inconsistent data. → Synchronization

Key Unix Process Syscalls

Call	Purpose	Returns
`fork()`	Duplicate process	0 to child; child PID to parent
`exec()`	Replace process image	Only on error
`exit()`	Terminate process	Notifies parent
`wait()`	Wait for any child	PID of terminated child
`waitpid()`	Wait for specific child	PID of child
`getpid()`	Get own PID	Current PID
`getppid()`	Get parent PID	Parent PID
`kill()`	Send signal	—

Lect 5: Inter-Process Communication (IPC)

Each process has independent memory; explicit IPC needed. Two main types: **Shared Memory** and **Message Passing**. Unix-specific: **Pipe** and **Signal**.

	Shared Mem	Msg Passing
OS overhead	Low (setup only)	High (every op)
Sync needed?	Yes (manual)	Built-in
Scope	Related procs	Any procs
Notes	Fastest after setup	Portable

Shared Memory

POSIX steps: `shmget()` (create) → `shmat()` (attach) → read/write via pointer → `shmdt()` (detach) → `shmctl(IPC_RMID)` (destroy). OS involved only at create/ attach; subsequent ops bypass kernel.
Pros: Efficient, ease of use
Cons: Harder sync, harder implementation

```
int shmid = shmget(key, size, IPC_CREAT | 0666);
// create segment
void *ptr = shmat(shmid, NULL, 0); // attach
// read / write via pointer (no kernel)
ptr[0] = 'A';
shmdt(ptr); // detach
shmctl(shmid, IPC_RMID, NULL); // destroy segment
```

Message Passing

Direct: `Send(P2,Msg)` / `Receive(P1,Msg)` — parties must name each other.
Indirect (mailbox): `Send(MB,Msg)` / `Receive(MB,Msg)` — shared among many.
Blocking = synchronous; **Non-blocking** = async (null if no message).
Pros: Portable,easier sync **Cons**: Inefficient,harder to use

Unix Pipes

Unidirectional FIFO byte buffer. `pipe(fd[])` → `fd[0]` read, `fd[1]` write. Writer **blocks** when full; reader **blocks** when empty. After `fork`: parent closes `fd[0]`, child closes `fd[1]`. `dup`/`dup2` redirect stdin/stdout. Named pipes (FIFOs) allow unrelated processes.

Unix Signals

Async notification. Register: `signal(SIGXXX, handler)`. Common: `SIGSEGV`, `SIGINT`, `SIGTERM`, `SIGKILL` (uncatchable), `SIGSTOP`, `SIGFPE`.

Lect 6: Synchronization

Race Conditions & Critical Section

Concurrent access to shared mutable resource → non-deterministic output. **Critical Section (CS):** `Enter CS` → work → `Exit CS`.

Property	Requirement
Mutual Exclusion	Only one in CS at a time
Progress	Waiting process eventually enters
Bounded Wait	Upper bound on others entering first
Independence	Non-CS processes don't block others

Deadlock: all blocked. **Livelock:** changing state, no progress. **Starvation:** some blocked forever.

Test-and-Set

Atomic: loads value, stores 1.
`while(TestAndSet(Lock)==1);` to enter; `*Lock=0` to exit. Satisfies mutual exclusion & progress. **Not** bounded wait (starvation possible). Busy-waits.

Semaphore

Integer S + sleeping-process list. `Wait(S)`: if $S \le 0$ block, decrement. `Signal(S)`: increment, wake one. Binary ($S \in \{0,1\}$) = mutex; counting = resource tracker. Incorrect use → deadlock.

Producer–Consumer (buffer size K)

```
mutex=1, notFull=K, notEmpty=0
// Producer                  // Consumer
wait(notFull);               wait(notEmpty);
wait(mutex);                 wait(mutex);
  add item;                    take item;
signal(mutex);               signal(mutex);
signal(notEmpty);            signal(notFull);
```

Readers–Writers

```
roomEmpty=1, mutex=1, nReader=0
// Writer                    // Reader
wait(roomEmpty);             wait(mutex); nReader++;
// modify data               if(nReader==1)
signal(roomEmpty);             wait(roomEmpty);
                             signal(mutex);
                             // read data
                             wait(mutex); nReader--;
                             if(nReader==0)
                               signal(roomEmpty);
                             signal(mutex);
```

Issue: writers may starve.

POSIX APIs

Semaphore: `sem_init`, `sem_wait`(=Wait), `sem_post`(=Signal), `sem_destroy`.
Mutex: `pthread_mutex_lock/unlock`.
Cond var: `pthread_cond_wait/signal/broadcast`.

Lect 7–8: Memory Management

Memory Layout

Text (instructions) · **Data** (globals) · **Heap** (grows ↑) · **Stack** (grows ↓). **Base+Limit:** PA = Base + LA; check LA < Limit.

Contiguous Allocation

Fixed partitions: internal fragmentation, simple.
avg IF = 50% of partition
Dynamic partitions: external fragmentation, needs compaction. **Algorithms:** First-Fit (first hole ≥ N), Best-Fit (smallest hole ≥ N), Worst-Fit (largest hole).

Buddy System

Free lists $A[0..K]$; $A[J]$ = blocks of 2^J. Free lists $A[0..K]$; $A[J]$ = blocks of 2^J.
Alloc: find smallest $2^S \ge N$; split if needed.
Free: merge with buddy if free; $\text{buddy}(A,S) = A \oplus 2^S$.
$S = a \oplus b, \quad S = 2^k, \quad a \bmod S = 0, \quad b \bmod S = 0$
Avg IF = 25%

Paging

Fixed-size frames/pages. Logical address: p (page num) | d (offset, n bits). PA = PT$[p] \times 2^n + d$.
TLB: hardware cache; flush on context switch.

- h = TLB hit rate
- t_{TLB} = time to check TLB
- t_m = one main memory access

h = (TLB hits) / (total memory accesses)
$\text{EAT} = h(t_{\text{TLB}} + t_m) + (1-h)(t_{\text{TLB}} + (k+1)t_m)$

PTE bits: readable (r), writable (w), executable (x), valid (v).
COW: parent+child share pages after `fork`; copy on first write.
Frag: no external, internal ≤ $2^n - 1$ bytes (last page).
Avg IF = $\frac{1}{2}$ page

Segmentation

Logical: ⟨SegID, Offset⟩. PA = Base[SegID] + Offset; check Offset < Limit[SegID]. Variable-size segments per region. Segments disjointed in RAM.
Can grow/shrink and be protected/shared independently.
vs Paging: external frag, no internal frag, programmer-visible, flexible protection.

Segmentation + Paging

Logical: ⟨S, P, D⟩. Use S → segment's page table base; look up P → frame F; PA = F × pagesize + D. No external frag; logical structure preserved.

Scheme	Ext	Int	Notes
Fixed partition	Y	Y	Simple
Dynamic	Y	N	Compaction
Buddy	Some	Y	Fast coalesce
Paging	N	Y	TLB needed
Segmentation	Y	N	Variable
Seg+Paging	N	Y	Best of both

Lect 9: Virtual Memory Management

Virtual Memory: Motivation & Idea

Previous assumptions (process fits entirely in RAM, contiguous) are too restrictive. **Idea:** split logical address space into pages; keep some in RAM, rest on secondary storage. Secondary storage >> physical memory. Works because of **locality principles**:

- **Temporal:** recently used address likely reused soon.
- **Spatial:** nearby addresses likely used soon.

If page faults occur most of the time → **Thrashing** (heavy I/O). Locality justifies why thrashing is uncommon in practice.

Page Fault Handling

Steps (hardware → OS):
1. Check page table: memory resident? → access & done.
2. **Page Fault:** trap to OS.
3. OS locates page in secondary storage.
4. OS loads page into a free frame.
5. OS updates page table (set resident bit = T).
6. Retry the instruction.

Demand Paging

Process starts with **no** memory-resident pages. Pages loaded only on page fault.
Pros: fast startup, small footprint.
Cons: sluggish initially; cascading faults possible (thrashing).

Page Table Structures

Direct Paging: single flat table; 2^P entries. E.g. 32-bit VA, 4KiB page, 2-byte PTE → 2 MiB table per process.
2-Level Paging: split into page directory + smaller page tables. Unneeded regions → no allocation. E.g. same settings, only 3 tables used: 1 KiB directory + 12 KiB tables = 13 KiB total (vs 2 MiB).
Address: `[dir-num | page-num | offset]` → directory lookup → page table lookup → frame.
Inverted Page Table: one global table indexed by **frame number**; entries = ⟨PID, page-num⟩.
Pros: huge space saving (1 table for all processes).
Cons: slow translation (must search by ⟨PID, page-num⟩); often paired with hash.

Page Replacement Algorithms

When no free frame on page fault, evict a page. **Clean page:** no write-back needed. **Dirty page:** must write back first.

$$T_{\text{access}} = (1-p)\cdot T_{\text{mem}} + p \cdot T_{\text{page\_fault}}$$

Since $T_{\text{page\_fault}} \gg T_{\text{mem}}$, minimise p (page fault rate).

OPT (Optimal)

Replace page not used for longest future time.
Minimum page faults guaranteed. **Not realizable** (requires future knowledge). Used as benchmark.

FIFO

Evict oldest-loaded page (queue). Simple, no hardware needed. **Belady's Anomaly:** more frames can → more faults! Reason: ignores temporal locality.

LRU (Least Recently Used)

Replace page unused for longest past time. Exploits temporal locality. No Belady's anomaly.
Hard to implement:
- **Counter approach:** timestamp each access; replace min timestamp. Cons: search all pages, counter overflow.
- **Stack approach:** move referenced page to top; replace bottom. Cons: arbitrary removal from stack, hard in hardware.

Second-Chance (CLOCK)

Modified FIFO + reference bit per PTE. On eviction: if ref-bit=1, clear bit & give second chance (treat as newly loaded); if ref-bit=0, evict. Degenerates to FIFO if all bits=1. Implemented as circular queue.

Algorithm	Faults	Anomaly	HW needed
OPT	Min	No	Future refs
FIFO	Most	Belady's	None
LRU	Good	No	Substantial
CLOCK	Good	No	Ref bit only

Frame Allocation

Equal: each process gets N/M frames.
Proportional: process p gets $\left(\frac{s_p}{s_{\text{total}}}\right) \times N$ frames.
Local replacement: victim chosen from faulting process's own frames → stable performance, but may starve the process.
Global replacement: victim chosen from any frame → self-adjusting, but badly-behaved process can hurt others.
Insufficient frames → Thrashing. Global: cascading thrashing. Local: thrashing contained but hogs I/O.

Working Set Model

Observation: process accesses a relatively stable **working set** of pages in any time window. $\mathbf{W(t, \Delta)}$ = distinct pages referenced in interval [t−Δ, t]. Allocate enough frames for W(t,Δ) to minimise faults.
Δ too small: misses pages in current locality → more faults. **Δ too large:** includes stale pages from old localities → waste.
Example: W(t1,5)={1,2,5,6,7} (5 frames), W(t2,5)={3,4} (2 frames).

Memory Hierarchy (Reference)

Level	Access Time	Size
Registers	0.25–0.5 ns	128 B
Cache	0.5–25 ns	6 MB
Main Memory	80–250 ns	16 GB
Disk	5,000,000 ns	500 GB

Address Translation Quick Reference

Base+Limit: PA = Base + LA; LA < Limit
Paging: $p = \text{LA} \gg n$; $d = \text{LA}\ (2^n - 1)$; PA = PT[p] × $2^n + d$
Segmentation: PA = ST[S].base + D; D < ST[S].limit
Seg+Paging: PA = $\text{PT}_S[P] \times 2^n$ + D
Memory access: $T_{\text{access}} = (1-p)\cdot T_{\text{mem}} + p \cdot T_{\text{pf}}$

Lect 10: File System — Abstractions

Physical memory is volatile → use external storage for persistent info. Direct hardware access is not portable → FS provides abstraction + protection + sharing.
Criteria:
(1) **Self-contained** — media carries enough info to describe its own organisation
(2) **Persistent** — survives OS/process lifetime.
(3) **Efficient** — good free/used space management, minimal bookkeeping overhead.

	Memory Mgmt	File System
Storage	RAM	Disk
Access Speed	Constant	Variable (disk I/O)
Unit	Phys. address	Disk sector
Usage	Process addr. space (implicit)	Non-volatile data (explicit)
Org.	Paging/ Segmentation	ext, FAT, HFS …

File: Basics

A **file** is a logical unit of information created by a process — an Abstract Data Type with data + metadata (file attributes).
Metadata fields: Name (human-readable), Identifier (unique internal ID), Type (executable/text/ directory/…), Size (bytes/words/blocks), Protection (access permissions), Time/date/owner info, Table of content (block locations).
File types: Regular (ASCII or binary), Directory (FS structure), Special (char/block device).
Distinguishing type: Windows → file extension (`.docx`). Unix → **magic number** embedded at start of file.

File Protection

Access types: Read, Write, Execute, Append, Delete, List. Most common approach: restrict by **user identity**.
Access Control List (ACL): list of (user, allowed access) pairs — flexible but large.
Unix condensed scheme: classify users as Owner / Group / Universe; define R/W/X bits for each class. Unix also supports extended ACL (named users/groups) via `getfacl`/`setfacl`.

File Data

Structures: Array of bytes (offset from start), Fixed-length records (jump to Nth record: offset = size×(N-1)), Variable-length records (flexible but harder to locate).
Access methods: Sequential (read in order, no skipping, can rewind), Random/Direct (`Read(offset)` or `Seek(offset)` + `Read`). Unix/Windows use `seek` approach.

File Operations & Open-File Tables

Generic ops: Create, Open, Read, Write, Repositioning (seek), Truncate.
OS provides these as **system calls** to protect, allow concurrent access, and maintain info. Per opened file the OS tracks: **File Pointer** (current position), **Disk Location**, **Open Count** (# processes with file open).
Two-level table approach:
- **System-wide open-file table:** one entry per unique open file.
- **Per-process file descriptor table:** one entry per file used, pointing into system-wide table.

Unix sharing cases (Process Sharing File):
(1) Two processes open independently → separate file descriptors, independent offsets.
(2) Parent/child share same fd (e.g. after `fork`) → **shared** offset; I/O in one affects the other.

Unix File System Calls

```
int fd = open("f.txt", O_RDONLY);          // open
int fd = open("f.txt", O_RDWR | O_CREAT); // create
// Default fds: STDIN=0, STDOUT=1, STDERR=2
int n  = read(fd, buf, nbytes);  // returns bytes read
int n  = write(fd, buf, nbytes); // returns bytes written
off_t p = lseek(fd, offset, SEEK_SET); // SEEK_SET/ CUR/END
int r  = close(fd);
```

Directory

Purpose: (1) Logical grouping of files (user view).
(2) Tracks files and maps name → file info (system view).
Structures:

Type	Description
Single-level	One flat dir; all files visible globally
Tree	Dirs recursively nested; abs. or rel. path
DAG	File shared in multiple dirs via links
General Graph	DAG + cycles; hard to traverse, avoid

Paths: Absolute = from root `/`. Relative = from current working directory (CWD).

	Hard Link (`ln`)	Symbolic Link (`ln -s`)
What	2nd dir pointer to same inode	Special file with path to target
Scope	Files only	Files or dirs
Delete B's link	A still has file	Target F unchanged
Delete A's file	B still has file (ref count)	B's link broken (dangling)
Overhead	Low (just a pointer)	Higher (extra file on disk)

Lect 11: File System — Implementation

Disk Organisation

Disk = 1-D array of **logical blocks** (smallest accessible unit, typically 512 B – 4 KB), mapped to hardware sectors. Layout:
`[MBR | Partition 1 | Partition 2 | … ]`
Each partition: `[Boot Block | Partition Details | Directory Structure | Files Info | File Data]`
MBR (sector 0) holds partition table; each partition can have an independent FS.

File Block Allocation

A file = collection of logical blocks. Last block may have **internal fragmentation**. Goal: track blocks, allow efficient access, use disk space well.

1. Contiguous Allocation

Consecutive blocks per file. Directory entry stores: (start, length).
Pros: Simple tracking; fast sequential and random access (seek once).
Cons: External fragmentation; file size must be declared in advance.

2. Linked List Allocation

Each disk block stores: (next block pointer, data).
Directory entry: (start, end).
Pros: No external fragmentation; file can grow freely.
Cons: Random access is slow (must traverse chain); pointer wastes space per block; reliability risk (broken pointer = lost chain).

2b. FAT (File Allocation Table) — Linked List V2

Move all pointers into an in-memory table indexed by block number.
FAT[block] = next block (-1 = EOF). Used by MS-DOS/ Windows FAT32.
Pros: Faster random access (traverse FAT in memory, not on disk).
Cons: FAT can be huge for large disks; consumes valuable RAM.

3. Indexed Allocation

Each file has an **index block** = array of block addresses. IndexBlock[N] = address of Nth data block. Directory entry: (index block location).
Pros: Low memory overhead (only open file's index block needed); fast direct access.
Cons: Max file size limited by index block size; index block overhead.
Variations for large files:
- **Linked scheme:** chain multiple index blocks.
- **Multi-level index:** index block points to more index blocks (generalises to any depth).
- **Combined scheme (Unix inode):** mix of direct pointers, single-indirect, double-indirect, triple-indirect blocks → supports small files efficiently while allowing very large ones.

Disk reads (Unix inode): 1 read per inode + 1 additional read per level of indirection (direct: 1, single: 2, double: 3, triple: 4 reads to reach data).

Scheme	Ext Frag	Random Access	Notes
Contiguous	Yes	Fast	Size fixed upfront
Linked List	No	Slow	Pointer overhead
FAT	No	Medium	Large table in RAM
Indexed	No	Fast	Index block overhead

Free Space Management

OS must track which blocks are free for allocation/ deallocation.

Bitmap

One bit per block (0 = occupied, 1 = free).
`0 1 0 1 1 1 0 0 1 0 1 1 ...`
Pros: Easy manipulation (bitwise ops to find free blocks efficiently).
Cons: Must be kept in memory for performance — large for big disks.

Linked List of Free Blocks

Each free-list block stores: a batch of free block numbers + pointer to next free-list block.
Pros: Only first pointer needed in memory (rest cached as needed).
Cons: High overhead — need disk I/O to traverse list.

Directory Implementation

Directory maps file name → file information. Must support: locate file (open), add/remove entries.
Path resolution: given `/dir2/dir3/data.txt`, recursively search each directory along path. Sub-directories are special file entries within a directory.

Linear List

Each entry: file name + metadata (or pointer to metadata). **Pros:** Simple. **Cons:** Linear search O(n) — slow for large dirs; use cache to speed up recent lookups.

Hash Table (size N)

Hash file name → index K in [0, N-1]; inspect `HashTable[K]`; use chained collision resolution.
Pros: Fast O(1) lookup. **Cons:** Fixed table size; depends on good hash function.

File Information Storage (2 approaches)

(1) Store all metadata directly in directory entry (simple, one disk access).
(2) Store only file name + pointer to separate data structure (e.g. Unix inode) — more flexible, separates naming from metadata.

File System at Runtime

On open: OS searches system-wide table → if found, add per-process entry pointing to it; if not, load file info from disk into system-wide table, then add per-process entry. Return fd (index into per-process table).
In-memory structures: system-wide open-file table, per-process fd table, **disk block buffers** (cache blocks read from / written to disk).
File creation steps: traverse path to parent dir → check no duplicate name → allocate free block(s) → add entry in parent directory with file info.

Hex conversion table

Hex	Dec	Binary
0	0	0000
1	1	0001
2	2	0010
3	3	0011
4	4	0100
5	5	0101
6	6	0110
7	7	0111

Hex	Dec	Binary
8	8	1000
9	9	1001
A	10	1010
B	11	1011
C	12	1100
D	13	1101
E	14	1110
F	15	1111

Buddy Block Size Table

Block Size	Power	Hex Value
512 B	2^9	`0x0200`
1 KiB	2^{10}	`0x0400`
2 KiB	2^{11}	`0x0800`
4 KiB	2^{12}	`0x1000`
8 KiB	2^{13}	`0x2000`

Race Condition Formulas

$$\text{Total interleavings} = \frac{(n_1+n_2+n_3)!}{n_1!\cdot n_2!\cdot n_3!}$$

Memory System Formulas

Bits

$$\text{Offset bits} = \log_2(\text{page size})$$
$$\text{VPN bits} = \text{VA bits} - \text{offset bits}$$
$$\text{PFN bits} = \text{PA bits} - \text{offset bits}$$

Counts

$$\text{Logical pages} = 2^{\text{VPN bits}} = \frac{\text{VAS}}{\text{page size}}$$
$$\text{Physical frames} = 2^{\text{PFN bits}} = \frac{\text{PAS}}{\text{page size}}$$

PTE Size

$$\text{PTE bits} = \text{PFN bits} + \text{control bits}$$
$$\text{PTE size} = \lceil \text{PTE bits} / 8 \rceil$$

Page Table (Single-level)

$$\text{Total PTE count} = 2^{\text{VPN bits}}$$
$$\text{Total PT size} = 2^{\text{VPN bits}} \times \text{PTE size}$$

Page Table (Multi-level)

$$\text{Entries per PT} = \text{page size} / \text{PTE size}$$
$$\text{Size of one PT} = \text{entries per PT} \times \text{PTE size}$$

Multi-level

$$\text{Bits per level} = \log_2(\text{entries per PT})$$
$$\text{Levels needed} = \lceil \text{VPN bits} / \text{bits per level} \rceil$$
$$\text{Top level bits} = \text{VPN bits} - (\text{levels} - 1) \times \text{bits per level}$$

Address Translation

$$\text{VPN} = \left\lfloor \frac{\text{virtual address}}{\text{page size}} \right\rfloor$$
$$\text{Offset} = \text{virtual address} \bmod \text{page size}$$
$$\text{Physical address} = (\text{PFN} \times \text{page size}) + \text{offset}$$

Segmentation

$$\text{Base}_{\text{next}} = \text{Base}_{\text{current}} + \text{Limit}_{\text{current}}$$
$$\text{End of segment} = \text{Base} + \text{Limit} - 1$$
$$\text{Physical address} = \text{Base} + \text{offset}, \quad \text{valid if} \quad \text{offset} < \text{Limit}$$