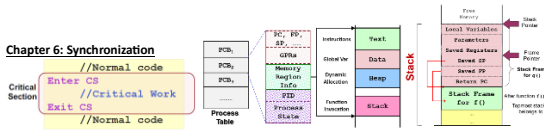
CS2106 AY23/24 Finals CheatSheet

Chapter 6: Synchronization



4 Properties of Correct Critical Section Implementation

Mutex	If process P_i is executing in critical section, all other processes are prevented from entering the critical section.
Prog-ress	If no process is in a critical section, one of the waiting processes should be granted access.
Bound-ed Wait	• After process P_i request to enter critical section, there exists an upper bound of number of times other processes can enter the critical section before P_i - No process hogs the CS (there is a time limit) - Decision of next process (i.e CS scheduling) should be fair. - Each process should have guaranteed access to the CS
Indpt	Process **not** executing in CS should never block other process.

Wrong implementations of Synchronization result in:

Deadlock	• All process blocked, no progress. (Circular wait) - Caused by accessing limited resources in an interleaving way.
Livelock	• Happens when deadlock avoidance mechanism malfunctions. • Processes are NOT Blocked. They just constantly change state to avoid deadlock. (Instead of running, jump on the spot)
Starvation	• Some Processes may be blocked forever with bad scheduling algo
Race Condition	Processes compete to get to access shared resources (concurrency), resulting in unpredictable/inconsistent outputs.

Assembly Level Implementation of Synchronization

- Implemented via Test and Set (Lock mechanism)
- Works, but uses busy waiting → Avoids simultaneous access to shared mem.
  - Wastes resources, as CPU time is used for waiting for the process's turn.
- Variants: Compare & Exchange, Store Conditional, Atomic Swap, Load Link.
- Peterson's Algorithm (Has both Turn and Want[] variables)

High Level Implementation of Synchronization

- Ensure that the 4 properties of synchronization are met.
- Why Disabling/Enabling **Interrupts** may not be a good idea for synchronization?
  - Buggy CS may stall whole system. Interrupts disables → Cannot Ctrl+C to end code, since keyboard interrupts are disabled.
  - Security issue. Interrupts → Kernel level method, need privileged access.
  - Busy waiting → waste clock cycles.
  - Interrupt is isolated within a single core. This synchronization method will not work for a multicore system. (Mutex not resolved)

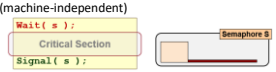
Turn based system is NOT good. (Turn = 0 or 1).

- Other programs will be starved if the program in CS does not finish executing, since the turn is only updated once the program finishes executing in CS.
  - If P_0 has yet to enter CS → turn not updated → P_1 gets starved.
- The process not executing in CS should not block other processes.
  - If P_0 is not in CS SHOULD NOT block P_1 (Need to update turn pre-empted)

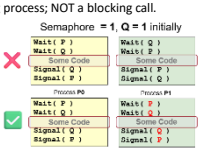
Global Check Array (Not good)

- This is similar to turn bases system, except that the turn/"want" parameter gets updated when another program is pre-empted.
- However, this can result in deadlocks (violates Progress)
  - If pre-emption happens right after P_0 sets want[0] = 1, and P_1 set want[1]=1
  - P_1 cannot escape the while (want[0]) loop.
  - When P_0 then gets pre-empted, it also gets stuck in while (want[1]) loop.

Semaphores 😊 (machine-independent)



- Uses Wait() and Signal() operations.
  - Wait() → If S ≥ 0, blocks; Decrements S
  - Signal(S) →Increments S; wakes up sleeping process; NOT a blocking call.
- Properties of Semaphores:
  - $S_{Initial} \geq 0$
  - $S_{current} = S_{Initial}$ + #Signal(S) - #Wait(S)
- Wait() & Signal() MUST BE USED CORRECTLY.
- Interleaving processes with wait(S) will cause deadlocks, since the signals() are unreachable.

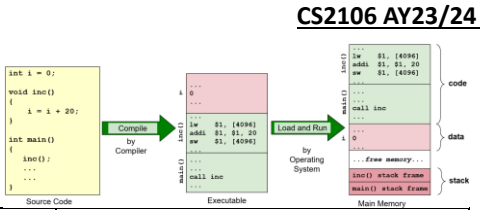


- When **user-level threads share a single kernel-level thread**, they can run **concurrently, but not in parallel**.
  - The instructions of user-level threads may be interleaved, giving the appearance that all threads are progressing together over time.
  - However, this does not mean they are running at the exact same moment.
- **True parallel execution**, where threads run simultaneously, **requires a computer with multiple processors or cores**.
  - Given that the **scheduler only see kernel-level threads**, it can allocate only one CPU core per kernel-level thread.
  - Consequently, user-level threads sharing a single kernel-level thread cannot run in parallel.

Chapter 7: Memory Abstraction

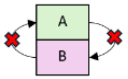
- A single **byte** (**byte addressable**) or ** char = 1 byte
- A single **word** (**word addressable**) **1 word = 4 bytes (Word is usually 2^n bytes)
  - Word alignment → the memory address is a multiple of the word size

- Buffer is flushed when 1. Buffer full, 2 fflush() called. 3. (\n) printed
- FD table is at the kernel side and is never copy-on-write.
- The threads in a single process will share the same fdtable, so no need to copy.



Transient Data	• Data that is valid only for a limited duration (e.g during fn calls) • These data have limited scope (can access it as long as the function is alive) E.g Parameters, local variables
Persistent Data	Data that is valid for the whole duration of the program, unless explicitly removed (if applicable). This data is valid till the program dies. E.g variables, dynamically allocated memory.

- The OS handles the following memory related tasks:
  - **Allocate** memory space to new processes.
  - **Manage** memory space for processes.
  - **Protect** memory space of processes from each other.
  - Provides memory related **system calls** to processes.
  - Manage memory space for **internal use.**



Given process A & B, the mem space of A&B should be independent. They cannot access each other's mem segment. They should be protected from rouge mem access

Aims of Memory Abstraction

1. Processes should NOT directly use physical address.
   - Directly using physical address → NO MEMORY ABSTRACTION
2. Memory locations should be **accessible** in a process.
3. Multiple processes can **share** the physical memory correctly.
   - Processes A, B and C share the same physical mem (ofc diff logical/virtual memory)
4. The address space of a **process** is protected.
   - I.e Processes A, B and C's mem space are isolated/independent of each other.

How to Abstract Memory? → USE LOGICAL ADDRESS (Generally, LA != PA)

- Logical address == how the process views its memory space
  - Instead of directly changing memory in HW, there is a mapping between LA & PA.
  - Each process has a self-contained, independent logical memory space.
- The mapping of logical address to physical address is given by the compiler.
- Moreover, programs are relocatable, since the mapping is set by the compiler.
  - This means that programs can be loaded into different memory locations without requiring changes to the code, improving flexibility and efficiency in memory usage.

Continuous Memory Management

Internal Frag	• Occurs when a process does not occupy the whole memory partition. (E.g worst fit algo). Occurs with paging and fixed partition. • Can be resolved/reduced using best-fit algo
External Frag	• Occurs when free memory are not continuous. I.e many holes in the memory. (Even though free space > space needed). • Occurs with segmentation and dynamic partitioning. • Can be resolved via compacting/defrag/ paging/merging.

Memory Partitioning Schemes:

Fixed-Size Partition

MIN=MAX overhead

Pros	Cons
• Easy management • Fast allocation	• Partition need to be big enuf • Internal Fragmentation

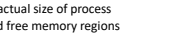
Variable Size Partition (Dynamic)

- Partition is created based on the actual size of process
- OS keep track of the occupied and free memory regions
  - Perform splitting and merging when necessary

Pros	Cons
• Flexible • Removes Internal Frag	• Req more info in OS • Takes more time (slow)

Allocation Algorithms *** LAB 4

If Process is size 6, Choose which hole?



First Fit	• Take the first hole that is large enough	Choose B
Next Fit Wraps around	• Searches from last allocated block, to find next hole that is large enough. • Uses mem more evenly + faster allocation (front indexes not saturated) compared to FF	Choose B
Best Fit	Find the smallest hole that is large enough	Choose C
Worst Fit	Find the LARGEST hole	Choose D

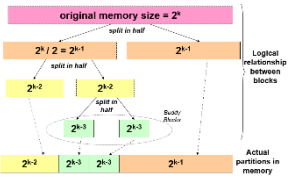
- With worst-fit, we won't end up with many small holes. Bigger holes → more efficient in assigning holes to new processes (in the future)

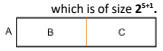
Linked List (Dynamic Partitioning Implementation)

- OS maintains a linked list for the partition information. O(n) time for access/ search.

Buddy System (Dynamic Partitioning Implementation)

- Takes O(log n) time
- Blocks are always in powers of 2.
- If two blocks **B** and **C** of size 2^S are buddies:
  - The lowest **S** bits (bits 0 .. S-1) of **B** and **C** are identical
  - Bit **S** of **B** and **C** is different.
  - E.g **B = 0**(000000_2), size = 16; **C = 16**(010000_2), size = 16
  - Combining **B & C**, we get **A** which is of size 2^{S+1}.





- Physical frames, page table, TLB are not stored in the memory.
- PCB & page table will be stored in RAM.
- The PCB stores info crucial for process management, in RAM.

2 n-bit address B[n-1:0], C[n-1:0] are starting address of two buddy blocks of size 2^S if:

- The least significant s bits are zero, i.e. B[s-1:0] = C[s-1:0] = 0, and
- The most significant n-s-1 bits are identical, i.e. B[n-1:s+1] = C[n-1:s+1], and
- The s-th bit are complement, i.e. B[s+1] != C[s+1]

Chapter 8: Disjoint Memory (Paging Scheme + Segmentation Scheme)

- Storing data in contiguous blocks is inefficient as it results in many holes.
- Disjoint chunks are more memory efficient. (OS splits the process into disjoint chunks, and git them into the chunks in memory)
  - Disjoint chunks can either be fixed or variable sizes.
- The frame size is decided by hardware. (Memory Management Unit, MMU in the CPU)
- Mem addresses are abstracted into logical mem, and this mapping is done by the MMU

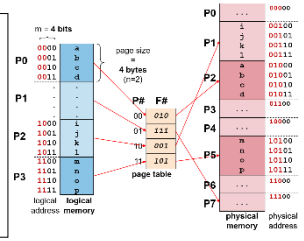
Paging Scheme

To support **multitasking**:

- Allow multiple processes in the physical mem simul. So that we can switch btwn processes.

Free up mem by (phy mem full):

- **Removing** terminated process (Reclaim used space for other processes)
- **Swapping** blocked process to secondary storage (Increases efficiency)

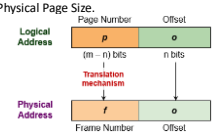


- Paging: split the logical address into fixed size pages
- A page table is used to translate between Logical Page and Physical Page.

$$Number\ of\ Frames = \frac{\#Bytes\ Needed}{PageSize}$$

$$Physical\ Address = FrameNumber \times sizeof(PhysicalFrame) + Offset$$
$$Logical\ Address = PageNumber \times sizeof(LogicalPage) + Offset$$

- Frame sizes are usually in the power of 2 → just need to bit-shift to multiply.
- Logical Frame SIZE = Physical Page Size.



- In Paging scheme, External Fragmentation is NOT POSSIBLE
- Internal Fragmentation is still possible if the process size != page size → Holes
  - There will only 1 page in a process that has holes. (Last page will have internal frag)
  - E.g 3kb needed, but page size = 2kb → split 2kb, 1kb → Hole = 1kb (Int Frag)
- Page table is PER PROCCESS, not global. It is kept tracked via the Process Control Block.
  - Keeps track of the pointer to base address of the page table.

Translation Look-Aside Buffer (TLB)

- "Cache" for Page Table entries.. **PTBR → Page Table Base Register
- TLB is a REGISTER → VERY FAST, but expensive, small storage. (TLB part of hardware cxt)
  - For every contact switch, update PTBR & update address in register.
- TLB maintains the page and frame number, and stores most used/recently accesses pages/frames in the TLB. → faster access time compared to 2 * mem reference.

With TLB	1 * Memory Access + 1 TLB look up: 1. Check if TLB has a hit (Fast, using registers!), if yes, return frame number + offset, if not, look up in the Page Table. 2. Access the actual memory item in Physical Memory. Average Mem Access Time: Latency: hit → 1MA, 1TLB; miss → 2MA, 1TLB TLB hit + TLB miss = P(hit) x latency(hit) + P(miss) x latency(miss)
No TLB	Requires 2 memory access for every memory reference: 1. Access the page table entry to get the frame number. 2. Access the actual memory item in Physical Memory.

TLB reduces the performance overhead of logical-to-physical address translation.
TLB reduces #page table access. TLB contains permission bits.

Paging Scheme Protection

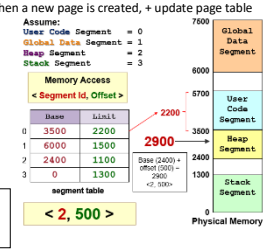
Access Right Bits	• Each Page Table Entry has { **writable, readable, executable** } bits • Every mem access is checked against the access right bits in hardware • Protect from unwanted behaviour via wrx.
Valid Bits	• Indicates whether the page is valid to access by the process • OS will set the valid bits as mem is being allocated to the process • Every memory access is checked against this bit in hardware: - Out-of-range access will be caught by OS in this manner

Paging Scheme Page Sharing

- With the use of Copy on Write(COW), we save memory and space.
- COW → Copy the page table instead of the frames → More efficient.
- COW → Only when a file is modified, then a new page is created, + update page table

Segmentation Scheme

While multiple processes reading the same file won't directly result in interleaved output due to file pointers and access control, there might be some internal interleaving within the operating system during concurrent reads.



When COW is applied to file systems, typically, only the metadata associated with the file (such as the inode in Unix-like file systems) is updated when a copy of the file is created.

- Segmentation: split the logical address into variable size seg according to their usage
- Each aspect of the memory (Code, Heap, Stack, Global Data) are abstracted.
- Each memory segment has a name (for reference) + limit (range/size of segment)
- Memory referenced by: **Segment Name + Offset**. <Seg ID, Offset> → e.g <2, 500>

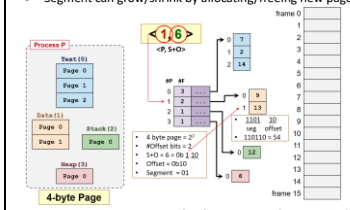
Segment Table

- Keeps track of Base Address + Limit Size. <Base, Limit>
- Access beyond limit → Segmentation fault. (Valid if Offset < Limit)
- Seg ID is used to lookup <Base, Limit> of the segment in the segment table
- Physical Address = Base + Offset.
- Size of segment table = Number of variables to track (e.g track code, global, heap, stack)

- Segments are disjoint pieces in memory, created to better fit a process. Segments are not further divided into subsegments.
- Segmentation does not suffer from external fragmentation
- A large mem space can be more efficiently represented by few segments, than by many pages.
  - The size of the segment table DOES not depend on the size of the corresponding segments, while the page table size increase with the #pages.

Segmentation with Paging

- Each segment is now composed of several pages instead of a contiguous- mem region
- Essentially, **each segment has a page table**.
- Segment can grow/shrink by allocating/freeing new page then add to its page table.



Operations incurring mem access:

- Read/Write/Fetch
- Load/Store
- PT lookups (in Virtual Mem)
- Cache Misses + TLB Miss
- Interrupt Handling, Page Fault

Generally: Accessing page table/accessing data/interrupts require memory access

- Individually keeps track of different aspects' page table (PT for Heap != Stack)

Chapter 9: Virtual Memory Management

- Store memory in storage devices (e.g HDD, SSD) instead of the RAM.
- Logical address space split into small chunks:
  - Some chunks reside in physical memory; Other are stored on secondary storage
- Gives us the illusion of "more memory"
- Logical Memory Space == Virtual Memory Space
- Use Page Table for Virtual (Logical) to Physical address translation

How to access Page X?

Hardware

1. Check **page table**:
   - Is page **X *memory resident***?
     - ➢ **Yes:** Access physical memory location. Done.
     - ➢ **No:** raise an exception!

How to access secondary storage?

OS

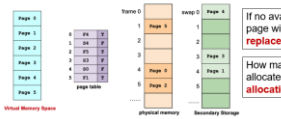
2. **Page Fault: OS takes control**
3. **Locate** page X in secondary storage
4. **Load** page X into a physical memory
5. **Update** page table
6. **G**o to step 1 to **re-execute the same instruction**
   - This time with the page in memory

Load into RAM from SSD

- Secondary Storage access time >>>> Physical mem access time (milli >>>> nano)
- If mem access results in page faults (i.e not mem resident) [**THRASHING**]
  - Slows down system, as we need to retrieve non-resident and paste it in memory.
- To prevent Thrashing, we can take advantage of the principles of locality.

Temporal	• Mem address used is likely to be used again (e.g loops) • After a page is loaded into phy mem, it may be needed again soon
Spatial	• Mem address close to that address is likely to be used soon (seq). • A page containing many conseq locations is likely to be accessed soon → Later access to nearby locations will not cause seg fault.
Demand Paging	• Processes start with NO MEMORY in resident page. • Only allocate a page when there is a page fault. • + Reduces startup cost, small memory footprint. • - Cascading effect → Many page faults back-to-back (initially)

When a new process is launched, should **ALL** pages be allocated in memory? – **Demand Paging**



If no available frames, which page will be replaced? – **Page replacement algorithm**

How many frames should be allocated to a process? – **Frame allocation**

What if page table is huge? – **Page table structure**

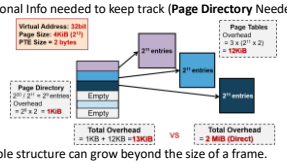
Page Table Structure **PT is PER PROCESS

Direct <P, PID> -> F	• size = # of frames (EXP) • Keep all entries in a single table. (2^P pages in logical mem space) • 2^P PTE with Physical Frame No + Addi Info bits (valid, rwx etc) • Not good → Constructing FULL page table waste space & mem.
2-Level /	• size = (# process) * (avg # pages / process) • Page the page table. (Split whole table into regions)

Data blocks must fill the direct blocks first, then indirect block, and after that doubly indirect blocks and so on.

Multi Level P->F

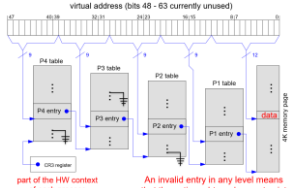
- Split 2^P table into smaller $2^{(P-M)}$ tables
- #Pages = VA space/ Page size | PT size = #Pages * Size PTE
- Additional Info needed to keep track (**Page Directory** Needed!)



\+ Page table structure can grow beyond the size of a frame.
\- Requires 2 * serialized memory access just to get frame number.
➢ Use TLB to remedy this issue, but will longer page-table walks.

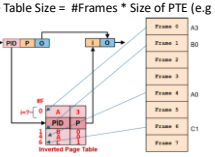
Hierar-chical

- Radix-tree structure. A table in each level in the hierarchy is git in a frame. All tables are setup by OS in SW, but traversed by HW



Invert-ed

- Size = Size of Physical Memory
- Keep a single mapping of physical frame to <pid, page#>
  - page# is not unique; pid + page# can uniquely identify a mem pg
  - #Frames = Size of Mem/Page Size
  - Inverted Page Table Size = #Frames * Size of PTE (e.g 2 bytes)



- In inverted page table, the entries are ordered by frame number
  - To lookup page X, need to search the whole table
  - Reduce memory needed, but longer lookup time; Easier to cache

Page Replacement Algorithms p → probability of a fault

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

Optimum (OPT)

- Theoretical Baseline to compare the other algorithms.
- Replace the page that **will not** be needed again for the **longest period of time** →Guarantees minimum number of page faults .
- However this is not feasible, as we cant predict the future.

FIFO

- Memory pages are evicted **based on their loading time**
- Evict the **oldest memory** page
- OS maintains a queue of resident page numbers, and removes the 1st page in the queue upon replacement request + Update queue.

Problems of FIFO

- FIFO does not exploit **temporal locality** (BAD PERFORMANCE)
- By right: If #phy frame ↑, #page faults should ↓.
- FIFO: ↑ frames →↑ page faults (Belady's Anomaly)
  - Any stack based algo will NOT suffer from Belady's Anomaly

LRU

- Make use of temporal locality
  - ***Recency:*** Replace page that hasn't been used in longest time.
  - ➢ *i.e* the **least recent page** one
  - Expect a page to be reused in a short time window
  - ➢ Page not used for some time → most likely will not be used again.
- Approximates OPT via predicting future by mirroring the past.
- Additional hardware is needed for Least Recently Used Algo.

1: Use Counter	2: Use "Stack"
• Tracks "Time of Usage" • Replace smallest TOU pg. • Overflow (32 bit int), while time is ↑ forever.	• Stack for page numbers. • Pop stack, use page then push to top of stack. • Bottom of stack replaced.

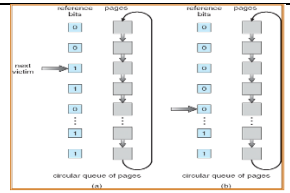
Clock 2nd chance

- BEST Algorithm (Closest to OPT)
- Maintains a Victim page, a pointer to victim page & reference ptr.
- A modified FIFO to give a second chance to pages that are accessed
- Each PTE now maintains a ***reference bit***:
  - 1 = Accessed since the last reset, 0 = Not accessed

Algorithm:

- The oldest FIFO page is selected (victim page)
- If reference bit == 0 → Page is replaced. **Done**.
- If reference bit == 1 → Page is skipped (given a 2nd chance)
  - Reference bit cleared to 0
  - Effectively resets the arrival time → page taken as newly loaded
  - Next FIFO page (victim) is selected, go to Step 2
- **Degenerate into FIFO** algorithm when **all pages has ref bit == 1**

When a page is evicted:
Clean page: not modified → no need to write back
Dirty page: modified → need to write back



- Maintain a **circular list** of all pages and a pointer pointing to the next potential **victim page**
- To find a victim:
  - Traverse till 1st page with '0' reference bit

Frame Allocation

- Enables us to know how many frames should be allocated to a process

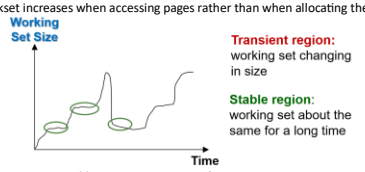
Equal Allocation	• Given N =#Phy Mem frames, M = #Process competing for frames • Each process gets same number of frames. (E.g N/M frames)
Proportional Allocation	• Process of different sizes gets equal proportion frame size. • E.g 10% of frames. (5 for N = 50, 10 for N = 100)

Frame Allocation & Page Replacement

	Local	Global
Def	Victim page selected among pages of the process	Victim page chosen among ALL physical frames
Pros	• #frames allocated to a process remain constant. • Performance is stable between multiple runs	Allow self-adjustment between processes. Process that needs more frame can get from others that need less
Cons	If the frames allocated are not enough, it may hinder the progress of a process	Badly behaved processes can affect others –steal frames from other processes. #frames allocated to a process can differ in subsequent runs
Thra-shing	Thrashing can be limited to one process, but that process may use up the I/O bandwidth & ↓ performance of other processes	A thrashing process steals pages from other processes → Can result in cascading thrashing. (Other processes also thrash 😩)
Thrashing: large memory consumption, low CPU utilization, slow progress. If Swap space not enough → memory error (malloc() failing)		

How to find the right number of frames to allocate?

- **Working Set Model**: monitors the memory usage patterns of processes and adjusts the frame allocations based on the working set of each process.
  - Defines Working Set Window Δ - An interval of time
  - **W(t,Δ)** = active pages in the interval at time **t**
  - Allocate enough frames for pages in **W(t, Δ)** to reduce possibility of page fault
  - Workset increases when accessing pages rather than when allocating them.



Chapter 10: File Systems ** Provides abstraction for Hard Disk

- A file is basically data + some operation. (It is an abstract data type)
- A file contains: **Data** (Info) + **Metadata** (Additional Info associated with the file)

Name	A human readable reference to the file
Identifier	A unique ID for the file used internally by File Systems
Type	Indicates different types of files E.g Executable, Text File, etc... • ASCII Files → Can be displayed/printed as it is (e.g txt file) • Binary Files → Needs special processing to read (e.g png file)
Size	Current size of file (in bytes, words or blocks)
Protection	Access permissions, can be classified as RWX rights. 000 to 777 XXX XXX XXX → OWNER GROUP OTHERS (7 = 111_2 RWX) Access: Read, Write, Execute, Append, Delete, List
Time,Data&OI	Creation, Last Modified Time, Owner ID etc... **OI** → Owner Info
TOC	Info for the File System to determine how to access the file.

- Operations on Metadata:
  - **Rename** → Change file's name
  - **Change Attributes** → File access permission, dates, ownership etc.
  - **Read Attributes** → Get file creation time

General Criteria for a good File System:

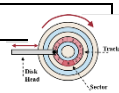
- **Self-Contained** (Plug&play, no extra configuration needed to read data e.g Thumbdrive)
- **Persistent** (Data should be independent of PC's state + Transferrable across PCs)
- **Efficient** (Provides good management of free/used space + Min overhead)

File Data Structure

- **Array of bytes**; each byte has a unique offset from the start of file
- **Fixed Length Records**; Offset of the Nth record = size of Record * (N – 1) **Int Frag
- **Variable Length Records**; flexible but harder to locate a record.

File Data Access Methods

Sequential	• Data read in order, starting from beginning • Cannot skip, but can rewind (e.g Cassette Tape)

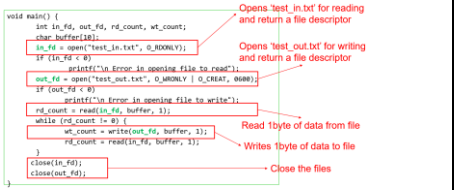


Disk Access Time = Seek Time + Rotational Latency + Transfer Time
Total Seek Time = Total head Movement * Seek Time

Random Special case of Direct	• Data can be read in ANY order. 1. Read(Offset) → Every read op explicitly state pos to access 2. Seek(Offset) → Special op to point at a specific address.
Direct	• Similar to random. Used for files containing fixed-length records. • Allow random access to ANY records directly, useful where there is a large amount of records. (Each record = 1 byte)

Operations in a File Data: (Usually a System call, e.g open(), read(), write(), lseek(), close()

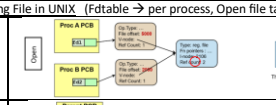
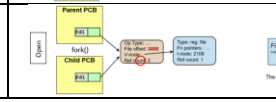
- **Create** → Make new file with empty data.
- **Open** → Performed before any other operations on file. Prep necessary info for file.
- **Read** → Read data from file, starting from the current position
- **Write** → Write data from file, usually from the current position
- **Reposition** → i.e seek(), Move the current position to a new position. Data not modified
- **Truncate** → Removes data between specified positions to the end of file.



Organizing File Information in OS (Open File Table Entry, OFTE methods)

Per Process	• Per process → Stored in PCB. • Keeps track of the open files for A PROCESS. • Each entry points to the System Wide Open File Table Entries
System Wide	• Global BIG table → Tracks ALL opened files in the system. • Keeps track of ALL the open files in the system. • Each entry points to a V-node entry
System Wide V-node Table	• Links the file to the physical device. • Contains info about the physical location of the file. • Maps the virtual addr of the file to the storage (for accessing)

Process Sharing File in UNIX (Fdtable → per process, Open file table → System wide)

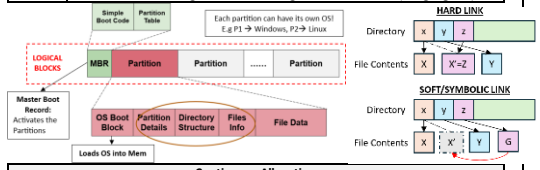
File opened in Multiple processes	
Multiple processes using the same OFTE	

Diff process opening same file twice → 2 entries in fd table → different offsets
Fork() → same offset

Directory

- **Single Level** → Bad, Files cannot have same name, not efficient. E.g CDs
- **Tree-Structure** →Can refer to file via absolute/relative path names.
- **DAG** → Shared file will use **Hard** Link, the pointers point to the SAME file.
- **General Graph** → Share file will use **Soft** Link. ** NOT DESIRABLE
  - Hard to traverse & to determine when to remove file/dir. (infinite loop)

Hard +1 inode	• Given Dir x, y, z, and file X, Y, Z. x share X with z. • x & z will have separate pointers that point to the actual file X in disk - Deletion of file x doesn't delete the file if z has hard-link to X
Soft +2 inode	• Z will create a special link file, G which contains the path of file x. • Z can access X via G, who has address of x and redirects to access x. - After deleting X', soft link through Z is not available (dangling link)



Contiguous Allocation

- Each file occupies a contiguous set of blocks on the disk.
- Need to keep track of starting + length of file to determine blocks occupied by file
  \+ Simple to track + Fast Access
  \- External Fragmentation, Increasing size of file is difficult. (Size is allocated in adv)

Linked List Allocation

- Each file is a linked list of disk blocks which need not be contiguous
- Each disk block stores: Pointer to nxt block + actual file data.
- File Just needs to store 1st and Last disk block number.
  \+ Resolves fragmentation, file size can be flexible
  \- Accessing a file is slow; Pointer = Overhead; If 1 link die, file corrupted

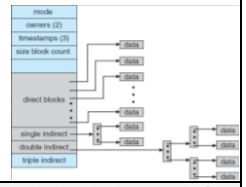
File Allocation Table (FAT) Allocation FAT16 → 2^{16} db →2^{16} entries (similar for FAT32)

- Similar to Linked List, but all block pointers are stored in a single table (FAT).
- FAT is in memory at all times and contains the values indicating nxt blocks address.
  \+ Faster Random access, since linked list traversal is in memory (not hard disk)
  \- Consumes valuable memory space → opportunity costs.
- FREE → unused; Num → nxt block; EOF → NULL ptr; BAD → unusable, error

- Minimize book keeping, Use good data structure, Minimize Fragmentations
- SOFT: Changing the name of linked file Q → cause soft link to be dangling

Indexed Allocation "Table of Contents", similar to hashmap

- Each file has an index block. Indexblock[N] == N^{th} Block Address (Its an array)
- The index block stores everything. It is also the ONLY overhead/info to access files.
- This is a per file basis. Each file will have their own index block & TOC.
  \+ Lesser overhead + Fast direct access + No external frag
  \- Limited max file size (Max#blocks = #index block entries)
- **Linked scheme:** Keep a linked list of index nodes, but expensive traversal cost
- **Multilevel index:** Similar to multi-level paging. Can generalize to any #levels
- **Combined scheme:**
  - Combination of direct indexing and multi-level index scheme
  - Fast access for small files(use direct index) + Handle large files (use multi-level)



UNIX Indexed Node (I-Node) [Ext2] (12 datablocks, 4bytes block, 1KiB diskblock)

Every file/dir will have a I-node associated with it (unique number).

- **Direct Blocks**: 12datablock * 1KiB = 12 KiB
- **Single Indirect Block**: #Entries = 1KiB / 4B = $2^{10}/4 = 256$ →256 * 1KiB = 256 KiB
- **Double Indirect Block**: 256^2 * 1KiB = 64MiB
- **Triple Indirect Block**: 256^3 * 1KiB = 16GiB

Theoretical ext2 limits under Linux[27]

Block size:	1 KiB	2 KiB	4 KiB	8 KiB
max. file size:	16 GiB	256 GiB	2 TiB	2 TiB
max. filesystem size:	4 TiB	8 TiB	16 TiB	32 TiB

Free Space Management (LAB4) Allocate → use the free space; Free → Delete/Trunc file

Bit Map (MIN=MAX overhead)

- Each disk block represented by 1bit. 0 → Allocated, 1 → Free
- Keeps track of ALL block info → wasteful; Size bitmap not affected by # partitions
- If 2^{64} blocks → bitmap size = 2^{64} (for which part of it is used)
- Bitmap cannot be manipulate like an array, we need to use bitmask!

Linked List

- Each disk block contains the #free disk blocks, or a pointer to next free space block
- Stored in the memory (same as bitmap, but uses less space)

Directory Implementation

- Need to recursively search the directories along the path to arrive at the file info.
- Sub-directory is usually stored as file entry with special type in a directory.

Linked List

- Stores file name (minimally) + other meta data. Store file info or pointer to file info
- Requires linear search, O(n), inefficient. Use cashing for better results

Hash Table

- Each directory contains a Hash Table of size N. ** HT[K] = file
- Locate by file name. File name is hased to index K from 0 to N -1.

File Information

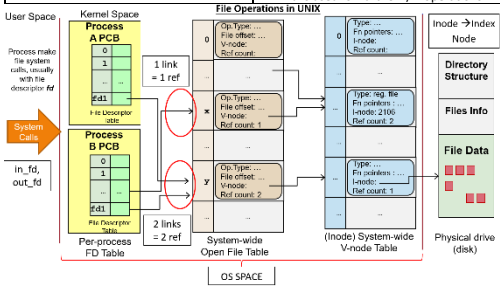
- File info consists of: File name + metadata, and Disk Block information.

1. Store everything in directory entry (Store info in the entry)	2. Store ONLY file name & point to some data structure for other info
	

- Every individual dir will have one of this block (containing dir entry)
- If there are sub-directories → Inception of blocks.

File System in Action

To create a file /../../parent/F:	To open file /../../../F:
• Use full pathname to find parent dir - Search filename F to avoid dupe ➢ If found, file creation terminates with error - Search could be on the cached directory structure. • Use free space list to find free blocks - Depends on allocation scheme • Add an entry to parent directory - With relevant file information - File name, disk block information etc	• Use full pathname to locate file F - If not found, open operation terminates with error • When F is located, its file info is loaded into a new entry E in system-wide table and V in I-node table if not already present. • Creates an entry in P's table to point to E (file descriptor) and pointer from E's entry to V • Return the fd of this. The returned fd is used for further r/w operations.



When the same file is opened again, all 3 tables will update.

- fd table is duplicated after fork(). (P & C will have same offset);(P=5fd → fork() →C=5fd)
- open(), creation of file → increases fd | n opens(), n new files = increase n fd.
- Given n processes → at least 3n file descriptors
- mmap() →maps a file on disk to a region of the process(es) VA/memory
- mmap() →Can be used to share files between processes wo syscalls
- Each process has 3 standard POSIX fd: 0stdin, 1stdout, 2stderr.