Memory - Ch 9 - 10

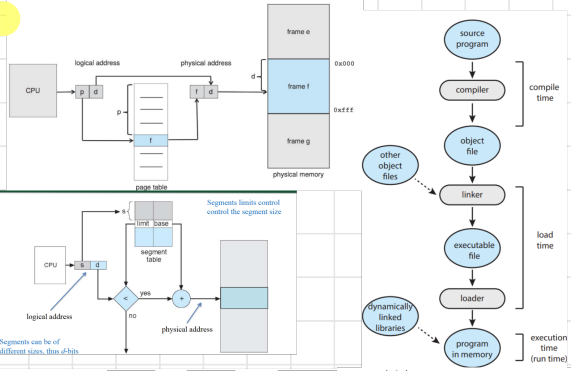
Chapter 9: Main Memory

Address Spaces

Logical Address	Used by CPU (Virtual)
Physical Address	Address in RAM
MMU	Hardware to map logical -> physical

Binding Times

- Compile time: absolute address embedded
- Load time: relocatable -> absolute
- Execution time: Uses MMU to bind dynamically



Contiguous Memory Allocation

Base & limit Registers

Base: Start of allocated memory
Limit: Bounds check -> if violate -> trap.

Allocation strategies:
- First Fit: First hole large enough
- Best Fit: smallest hole big enough
- Worst Fit: largest hole available

Fragmentation:
- External: Scattered free memory
- Internal: Unused memory within allocated block.

Compaction: Merge free holes -> one large hole
Only possible with dynamic Relocation.

Segmentation

Program = logical segments (code, data, stack)
Logical Address = (segment #, offset)
Segment table = base + limit per segment

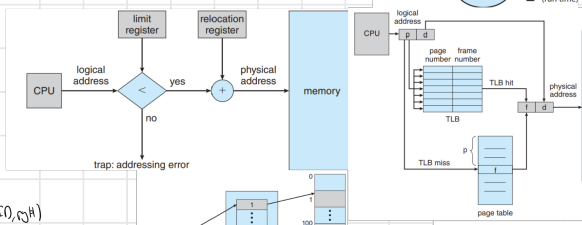
Advantages: Logical program structure preserved
Supports sharing, protection

Disadvantages: External Fragmentation.

Effective Access Time (EAT)

EAT = α(TLB+M) + (1-α)(TLB+2M)

- α = TLB hit ratio
- M = memory access time
- TLB = TLB lookup time



Paging

Frame: Fixed-size block in physical mem.
Page: Fixed-size block in logical memory
Page table: Maps pages to frames

TLB: (cache that stores (Page # -> Frame #)) Mapping
TLB hit: Faster access, (no mem ref.) TLB + mem
TLB miss: Must access Page in memory TLB + mem(page) + mem(desired)
(Translation lookaside buffer)

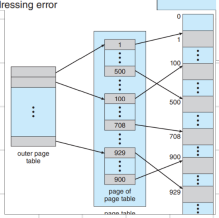
Logical Address (Virtual Address)
Issued by CPU; split into: Page #(p) | Offset(d)
Physical Address: Points to actual mem location
Produced by MMU: Frame #(f) | Offset(d)

Fragmentation: avoids external,
internal: waste in last page (avg ½ pg per proc)
Share pages: Re-entrant (non-modifying)
code can be shared (e.g. library, etc.)

Page table Structures

Hierarchical: Break large page table into multiple levels
Hashed: Suitable for large address spaces.
Inverted: 1 entry per frame, uses hashing of (PID, pg#)

Page table Registers: PTBR: points to base start
PTLR: pg tbl len register (address validity)

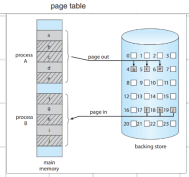


Swapping

Swap process out to disk, back in when needed.
- Requires dynamic relocation.
- Not used in Mobile OS.

Ex. Architectures
IA-32 (80)
- 2-level paging 10 bits for page dir +10 for pg table +12 offset.

ARMv8: similar hierarchical paging (4-DIR support.
- Supports both segmentation & paging



Chapter 10 (Virtual Memory)

Demand Paging

- Only load pages when referenced
- Page fault: access to non-resident page
  -> trap to OS -> bring from disk
- Benefit: Reduce mem usage, ↑ # of processes
- Lazy loading: pg not load til 1st access (faster program start)
- Valid/Invalid bit in page table entry.
  - V: page in mem
  - I: not in mem, page fault.

Page fault steps (Worst-Case)

1) Traps to OS
2) Checks if legit
3) Find location on disk
4) Get free frame (or use page replacement)
5) Read page in
6) Update page table
7) Restart instruction (restart)

EAT (Effective Access Time)

EAT = (1-p) · mem access + p (page fault time)

Example: mem access = 200 ns, pg fault time = 8 ms, p = 0.001
EAT = 200 + 0.001 · 8,000,000 = 8,200 ns = 40x slowdown

Copy-on-Write (COW)

Used by fork()
- Child & parent share pages til' a write -> duplicate only write
- Reduces mem. usage for dup. processes.

Page Replacement

Needed when no free frames

Alloc & Replacement Scope

Type	Description
Local Replacement	Victim frame selected from faulting process
Global Replacement	Victim can come from any process
Priority Allocation	Higher-priority processes get more frames

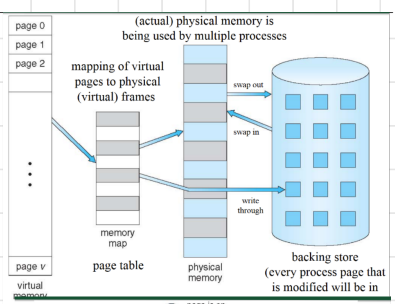
Algorithm	Description	Notes
FIFO	Evict oldest page	May suffer from Belady's Anomaly
Optimal	Replace page used farthest in future	Theoretical best
LRU	Replace least recently used page	Good performance, needs timestamp or stack
Clock / Second Chance	FIFO with use bit	If `use bit = 1`, clear and skip
Enhanced Second Chance	Uses `(ref, mod)`	Better approximation

Modify / Dirty bit -> only write dirty pages to disk

Thrashing (Too much pg. swapping)

High Page fault Rate -> low CPU use
- System adds more processes, workers starve.

Solution: WS model:
monitor & suspend processes if $D > m$
Page fault frequency (PFF) control:
adjust frames based on fault rate.
↓ decrease degree of multiprogramming



Memory Mapped Files: mmap()

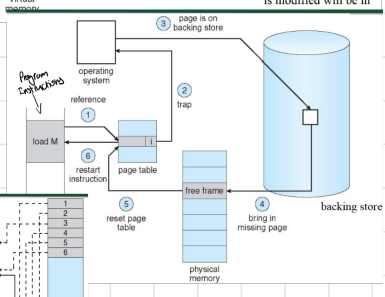
- Use mmap() to file virtual memory
- File access -> memory reads & writes
- Benefits: simpler I/O, interprocess communication, on demand loading
- munmap(): unmaps file

Kernel Mem allocation

- Different from user process paging
- Must allocate quickly & predictably.

Buddy System

- Free mem split into power of 2 sized blocks
- Combines Buddies when free to avoid fragmentation



Locality & Working Set

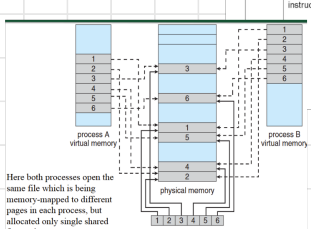
Locality of reference
- temporal locality: re-use of recently executed pages.
- Spatial locality: memory pages likely accessed soon.

Working set model
- $WS(t, \Delta)$ = pages accessed in last Δ time units
- WSS_i = working set size of process i
- $D = \sum WSS_i$ = total demand
- If $D > m$ (available frames) -> THRASHING

Belady's Anomaly: Occurs in FIFO Replacement

Definition: Belady's Anomaly is the counterintuitive situation where increasing the number of page frames results in more page faults in certain page replacement algorithms.

- FIFO only considers arrival order, not usage patterns.
- Replacing oldest pg may evict one that will be soon used again.
- Avoid this by using stack-based algs (LRU, OPT).



Here both processes open the same file which is being memory-mapped to different pages in each process, but allocated only single shared frames in memory.

Inverted Page Table

- 1 Entry per frame, not per process
- Entry = (PID, page#) -> frame #.
- Saves mem, but slower (needs hashing)

OPT (Page-Fault Frequency Control)

- Adjust # of frames needed on fault rate
- ↑ increase frames if faults too frequent;
  ↓ decrease if fewer misses

File System: Ch13-15

Chapter 13 - File System Interface

File Concept: File is a named collection of info stored on disk
- OS abstracts physical storage onto logical units (files)
- Files persist across program executions

File attributes:
- Name - human readable identifier
- Type - by extension or metadata
- location - disk address
- Size - current file size
- Protection - read/write/execute/etc.
- Timestamps - created, modified, accessed
- Owner/Group - for access control

File Operations:

```
create(), open(), read(), write(), seek(), close(), delete(), truncate()
```

- Open File table: holds open files + info (pointer, rights, loc)
- Seek repositions file pointer. Open/close: Sync between memory & disk.

File Access Methods

Method	Description	Use Case
Sequential	Read/Write in order	Editor, Logs
Direct	Access by block #	Database
Indexed	Uses index for quick lookup	OBJ, catalogs

File Structure
- No structure: Byte Stream
- Record Based: Fixed/variable size
- Complex: Formatted files, executables. File parser must understand format.

File Links:
- Hard link: multiple names → same inode
- Soft link (symbolic): points to another path
- Deletion: File removed when all hard links gone

Directory Structure

Type	Description
Single-Level	One flat directory
Two-Level	Separate directories per user
Tree	Hierarchical (common)
Acyclic Graph	Allows shared subdirs
General Graph	Supports cycles (w/ issues)

Directory Operations:
mkdir(), rmdir(), link(), unlink(), rename(), opendir(), closedir()

File Protection
- Types of Access: read, write, execute, append, delete, list, change_attr.

Protection Models

Model	Description
UNIX Permissions	Owner/Group/Others + rwx bits
ACLs	Fine-Grained list of who can do what
Access Matrix	List of subjects × objects = rights

Mounting & Portions
- Mount: Attach a FS to mount-point
- Unmounted FS = inaccessible
- Every volume has its own directory tree, mounted into the global FS

File Sharing
- File sharing across users or systems (NFS)
- Requires access control & Protection
- Shared directories & files need permissions & possibly remote access support

File System Structure
- File System resides on Secondary Storage (HDD, SSDs)
- Operations are layered:
- Application programs
- Logical FS → directories, metadata
- File organization module → block mapping
- Basic FS → issues block-level commands
- I/O control → device commands
- Drivers → interact w hardware (devices)

Chapter 14 - File System Implementation

On-disk & in-memory structures

Structure	Purpose
Boot control block	Boot OS from disk
Volume control block	Disk layout info: block count, Free blocks, FCB pointers
Directory structure	File names → inode or metadata
File control block	Permissions, size, timestamps, pointers to data blocks
Open File tables	Info on opened files (system-wide & per process)
Buffers/Caches	Store data in transit between memory & disk.

Directory Implementation
- Linear list: Simple, but w/ slow lookup
- Hash table: Fast access, but rehashing & collision management needed

Blocks & Tables
- Boot control block: Contains sys info needed to boot OS from that volume.
- File control block (FCB): a large structure w/ file info
- Volume control block: contains all block info (total #, # of free, block size, free block ptrs)

Mount table: Stores mounts, mtpts, fs types
System-wide open file table: contains FCB copy of file
Per-process open file table: has pointers to relevant entries of sys-wide open file tbl

Allocation Method - extent - contiguous block of disks

Type	Details	Best For
Contiguous	Start + length	Fast access, suffers from fragmentation
Linked	Pointers between blocks	Good for sequential access not random
FAT	Table of pointers	Better random access than linked
Indexed	Index block w/ block pointers	Efficient, Random access, small file overhead
UNIX inode	12 direct - single, double, triple indirect	Supports huge files (4+ GB)

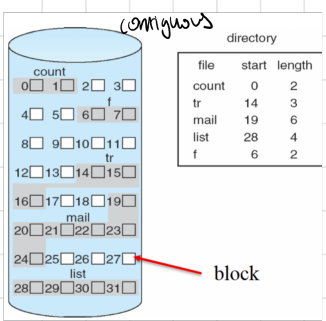
Performance Considerations
- Sequential vs random impacts allocation strategy
- Use clustering to reduce disk seeks
- Read-ahead, write-behind, & caching improve throughput
- Optimize CPU vs. disk trade-offs
- Id unnecessary overhead movement

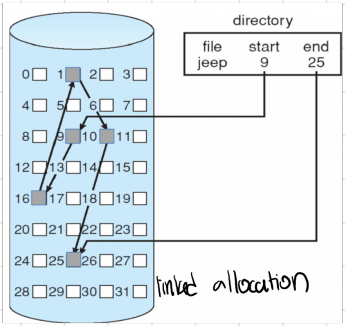
Free Space Management

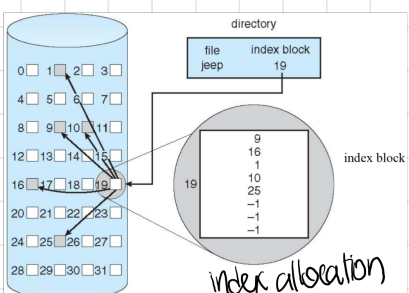
Method	Description
Bit Map	1 bit per block (0=used, 1=free); Fast search, uses memory
Linked List	Blocks point to next free one, slow traversal
Grouping	First block stores N free addresses; chain to next group
Counting	Track 1st block + count of free sequential blocks
Space Maps (ZFS)	Logs block activity per metaslab; compact & scalable

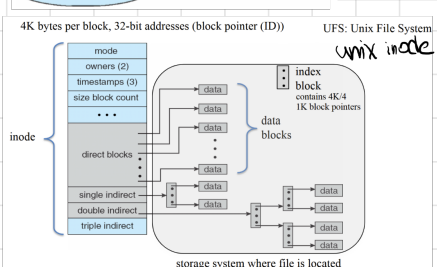
Recovery & Consistency
- Consistency Checker: scans FS
- Back-ups: allow recovery from corruption or loss
- Journaling/log-structured FS: track updates b4 implementation for errors

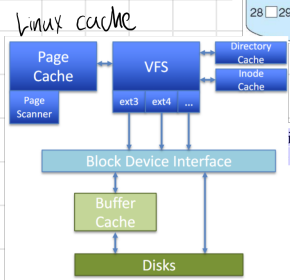
TRIM = "File system tells SSD: you can erase these now."
It bridges the gap between **FS free space** and **SSD block reuse** to optimize speed and durability.

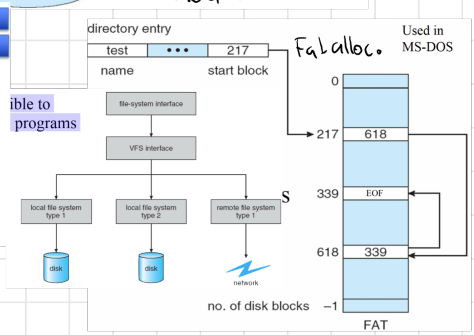


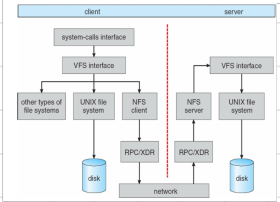








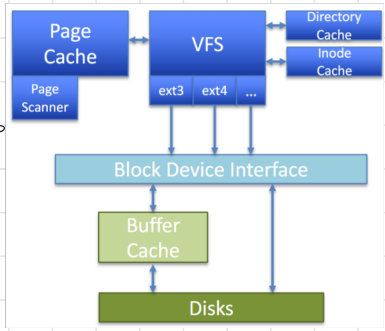




Chapter 15 - File System Internals

File Systems Basics

- A file system allows data to be stored, and retrieved
- A computer may support multiple FS types (ext4, tmpfs, ZFS, etc.)
- Storage hierarchy: physical devices → partitions → volumes → FS



file type	usual extension	function
executable	exe, com, bin or none	ready-to-run machine-language program
object	obj, o	compiled, machine language, not linked
source code	c, cc, java, pas, asm, a	source code in various languages
batch	bat, sh	commands to the command interpreter
text	txt, doc	textual data, documents
word processor	wp, tex, rtf, doc	various word-processor formats
library	lib, a, so, dll	libraries of routines for programmers
print or view	ps, pdf, jpg	ASCII or binary file in a format for printing or viewing
archive	arc, zip, tar	related files grouped into one file, sometimes compressed, for archiving or storage

File-Systems Mounts

- File systems must be mounted before use.
- Mount point: a directory where a FS is attached (e.g. /home)
- Must contain valid FS structure; often validated at boot
- Some OSes (e.g. macOS) auto-mount detected volumes
- Mount Examples: `mount /dev/sda1 /mnt/home`
- Mounting hides existing contents of the directory

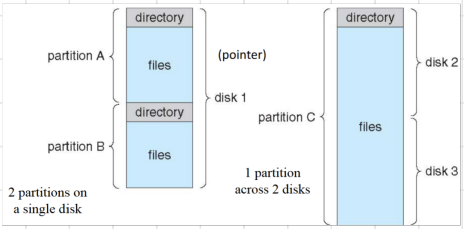
Partitions

A partition is a logical division of a disk

Partitions can be:

- RAW: no FS (e.g. swap) Boot partition: contains OS
- Cooked: contains FS

- Volume mangers can span FS across multiple partions/disks.



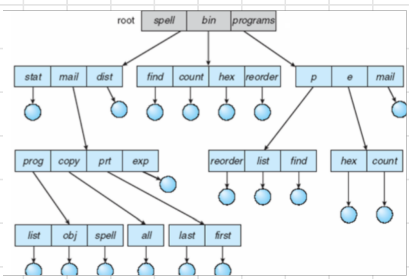
Virtual Files Systems (VFS)

Uses vnodes: holds index or network file details

- Provides abstraction layer for all FS types.
- Defines a common API for accessing files
- Every FS provides its own VFS operations (e.g. lookup, read)
- Used by UNIX, LINUX, Windows (Win32 API under VFS model)

Remote Files Systems

System	Use
NFS (Network File System)	UNIX-based, RPC-driven
CIFS/SMB	Microsoft; authentication via Active directory
FTP/Web	Simple transfers; less integration



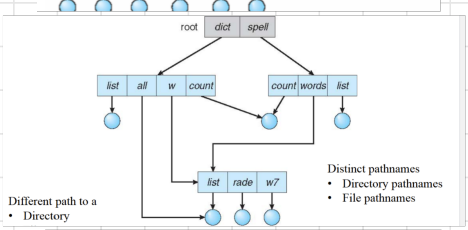
NFS Protocol

- Stateless: each request contains all needed info
- Uses Mount Protocol & Path-name Translation

Consistency Semantics

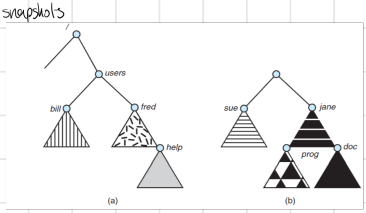
Model	Description
UNIX	Changes visible immediately
Session Semantics	Updates only visible after file closed
Immutable-Shared	No changes once file is opened

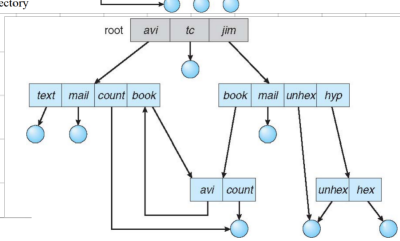
OS must coordinate how file changes are seen by different process



Files Semantics in Practice

- Systems like /proc use virtual FS that compute file contents dynamically
- Immutable FS useful for concurrency (e.g. config snapshots)



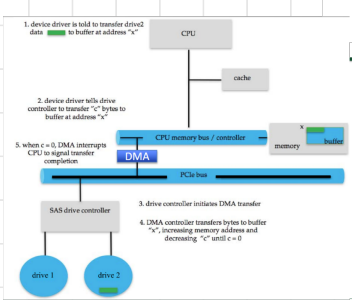


File Sharing & Protection

- Multi-user system → need controlled sharing
- Metadata includes
  - Owner
  - Permissions
- File sharing handled via:
  - local mount
  - Remote client (e.g. NFS, CIFS/SMB)

Case Study: NFS (Network File System)

- Mount protocol authenticates & mounts remote FS
- NFS Protocol
  - Stateless communication
  - Handles operations like read, write, lookup, getattr
- May suffer statelessness issues unless careful caching is used



- An access matrix is one way to represent policy.
  - Frequently used mechanism for describing policy
  - Columns are objects
  - Subjects are rows
  - Cells are rights
- To determine if S_i has right to access object O_j, find the appropriate entry.
- Succinct descriptor for $O(|S|*|O|)$ entries
- There is a matrix for each right

	O_1	O_2	O_3
S_1	Y	Y	N
S_2	N	Y	N
S_3	N	Y	Y

Chapter 12: I/O Systems

I/O System Overview

- I/O is slow relative to CPU/memory → needs OS optimization
- OS abstracts wide variety of devices: Device drivers: encapsulate
  - Character/block, sequential/random access, device details.
  - Shared/dedicated, read-only/read-write.

I/O Hardware components

Component	Role
Device Controller	Manages a device (registers + buffer)
Bus	Transfers data/commands between CPU, mem, I/O
Port	Interface to communicate w/ controller.
Fiber Channel (FC)	(complex controller/cable/switch circuit board)

- Memory-mapped I/O: device registers mapped into sys. mem space
- Bus: daisy chain or shared direct access:
  - PCI, PCIe • expansion bus • serial attached SCSI (SAS) common disk interface

I/O communication methods

Method	Description	Pros	Cons
Polling	CPU checks status repeatedly	Simple	CPU waste
Interrupts	Device signals CPU on readiness	Efficient	Overhead, needs handler
DMA	Device writes data directly to mem.	CPU offload	Set up complexity

DMA: Direct memory access → best for bulk transfers (disk/NIC)

DMA steps: CPU sets up → DMA controller handles blocks → interrupt on complete.

Application I/O interface

- Provided via system calls (read(), write())
  - hides device specific details behind standard API

I/O Modes

Mode	Behavior
Blocking	Waits until I/O completes
Non-Blocking	Returns immediately (poll status later)
Asynchronous	Completion notified via signal/call back
Vectored I/O	Multiple buffers in one I/O operation

I/O devices vary in many dimensions:
- Character stream or block
- Sequential or random access
- Synchronous or asynchronous (both?)
- Sharable or dedicated
- Speed of operation
- Read-write, read only, write only.

Kernel I/O subsystem

Responsibilities:
- I/O scheduling: optimize request order
- Buffering: temp storage to smooth to mismatch
- Caching: store frequently accessed blocks
- Spooling: queue jobs for non-shareable devices (printers)
- Error handling, protection, device reservation.

Buffering decouples producer/consumer, enables prefetching & delay tolerance.

I/O Request life cycle:

1) App issues sys call (read())
2) OS checks cache → if miss, schedule device I/O
3) DMA handles transfer (if applicable)
4) Device triggers interrupt on completion
5) OS returns app w/ data/result.

I/O scheduling

- Similar to CPU, but for I/O queuing (esp. disks)
- Prioritize time-sensitive ops (e.g. virtual mem pg in)
- Batch read/writes to reduce overhead.

Device Drivers

- OS component knows how to interact w/ a specific device
- Each driver implements a standard interface
- Layered Model: top layer: device-independent code
  bottom layer: hardware-specific logic

Streams (Advanced UNIX I/O)

- Message-passing architecture for I/O pipelines (System V)
- Enables modular drivers. Not common today, important historically

Performance Optimization

Technique	Purpose
Buffering	Smooth I/O variation
Read-Ahead	Pre-fetch likely needed data
Write-Behind	Delay writes for batching
Double Buffering	Avoid stalls by waiting
Interrupt Coalescing	Reduce interrupt frequency