

UNIT 5

Memory Management

(7 Hrs) — Operating System (CACS251)

5.1 Basic Memory Management (3 Hrs)

5.1.1 Introduction

Main memory (RAM) is a large array of words or bytes, each with its own address. The CPU fetches instructions from memory according to the program counter. Since a program must be brought into memory to run, and many programs share the memory, the operating system must manage it. The part of the OS that does this is the **memory manager**. Its tasks are: keeping track of which parts of memory are in use and which are free; allocating and de-allocating memory to processes; swapping between main memory and disk when memory is too small; and providing protection.

5.1.2 Memory Hierarchy

Programmers want memory that is large, fast and non-volatile — and cheap. Since no such memory exists, computers use a **memory hierarchy**: a small amount of very fast, expensive memory (registers and cache), a moderate amount of medium-speed main memory, and a large amount of slow, cheap disk storage.

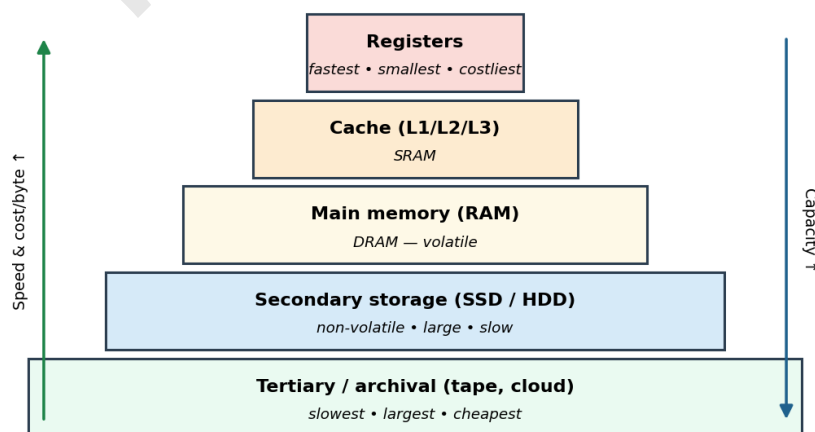


Figure 5.1 — Memory hierarchy

Level	Typical size	Access time	Volatile?
Registers	< 1 KB	≈ 1 ns	Yes

Level	Typical size	Access time	Volatile?
Cache (L1–L3)	KB – tens of MB	$\approx 2 - 20$ ns	Yes
Main memory (RAM)	GB	$\approx 10 - 100$ ns	Yes
Secondary storage (SSD/HDD)	100s of GB – TB	$\approx 0.1 - 10$ ms	No

5.1.3 Logical versus Physical Address Space

- **Logical (virtual) address** — address generated by the CPU; the set of all of them is the **logical address space**.
- **Physical address** — the address seen by the memory unit (loaded into the memory-address register); the set is the **physical address space**.
- The run-time mapping from logical to physical addresses is done by the hardware **Memory Management Unit (MMU)**. In the simplest scheme the MMU adds the value in a **relocation (base) register** to every address generated by the process.

Example — Relocation register

A process has a logical address 346 and the relocation register holds 14000.

Physical address = $14000 + 346 = \mathbf{14346}$.

5.1.4 Memory Management without Swapping or Paging (Monoprogramming)

The simplest scheme: only **one program** runs at a time and shares memory with the OS. The user program may occupy the space above the OS (in RAM), or the OS may be in ROM, or device drivers in ROM (as in MS-DOS with the BIOS). When the user types a command the OS loads the program from disk; when it finishes, it prompts for the next. No swapping, no relocation problem, but poor CPU utilisation and no multiprogramming. Protection of the OS from the user program uses a fence or a base register.

5.1.5 Memory Management with Swapping

On a time-sharing system there is not enough RAM for all processes. **Swapping** brings a process entirely into memory, runs it for a while, then puts it back on disk (**swap out**) to make room for another (**swap in**). Because processes may be loaded in different places each time, relocation is needed. Swapping creates multiple holes in memory; the operating system must keep track of them. Two common ways are bitmaps and linked lists.

(a) Memory Management with Bitmaps

Memory is divided into **allocation units** (a few words to several KB). Each unit corresponds to one bit in a bitmap: **0 = free, 1 = occupied**. Simple, and the size of the bitmap is fixed by the size of memory and the unit size (a unit of 4 bytes needs 1 bit per 32 bits = 1/32 of memory).

Problem: to bring in a process needing k units the manager must search the map for a run of k consecutive 0 bits — a slow operation.

(b) Memory Management with Linked Lists

A linked list of **segments** is maintained; each entry specifies a **Process (P) or Hole (H)**, the starting address, the length and a pointer to the next item. The list sorted by address makes it easy to **coalesce** (merge) adjacent holes when a process terminates or is swapped out. A doubly linked list makes the merging faster.

Placement (allocation) algorithms

Algorithm	Rule	Comment
First Fit	Scan from the beginning; use the first hole that is big enough	Fast; leaves small holes at the front of memory
Next Fit	Like first fit but starts scanning where the last search ended	Slightly worse than first fit
Best Fit	Search the entire list; use the smallest hole that is big enough	Slower; creates many tiny useless holes
Worst Fit	Use the largest hole	Leaves large leftover holes, but performs poorly
Quick Fit	Separate lists for common sizes of holes	Fast to allocate; costly to merge holes

Example — Placement algorithms

Free partitions in memory order: 100 KB, 500 KB, 200 KB, 300 KB, 600 KB. Place processes of size 212 KB, 417 KB, 112 KB and 426 KB (in that order).

Solution

First fit: 212 → 500 (leaves 288); 417 → 600 (leaves 183); 112 → 288 (the leftover of the 500 partition); 426 → **must wait** (no hole large enough).

Best fit: 212 → 300; 417 → 500; 112 → 200; 426 → 600. **All four processes are placed.**

Worst fit: 212 → 600 (leaves 388); 417 → 500 (leaves 83); 112 → 388; 426 → **must wait**.

In this example best fit makes the most efficient use of memory.

5.1.6 Contiguous Memory Allocation

In **contiguous allocation** each process is contained in a single contiguous section of memory. Memory is usually divided into two partitions: one for the resident operating system (usually in low memory with the interrupt vector) and one for user processes.

(a) Memory Protection

The OS must be protected from user processes and user processes from each other. Two registers are used: the **relocation (base) register** holds the smallest physical address of the process; the **limit register** holds the range of logical addresses. The MMU checks every address: if *logical address* < *limit* then *physical* = *logical* + *base*; else a **trap (addressing error)** is raised to the OS.

(b) Memory Allocation — Fixed vs Variable Partitioning

Fixed (static) partitioning	Variable (dynamic) partitioning
Memory divided into fixed-size partitions at boot time	Partitions created dynamically to fit each process exactly
Degree of multiprogramming limited by number of partitions	Degree of multiprogramming varies
Internal fragmentation occurs	External fragmentation occurs
Simple, low overhead	More complex; needs hole management and compaction
Used in early IBM OS/360 (MFT)	IBM OS/360 (MVT), modern segmented systems

(c) Fragmentation

Internal fragmentation	External fragmentation
Memory inside an allocated partition that is unused because the process is smaller than the block	Total free memory is enough for a request but is not contiguous — it is scattered in small holes
Example: a 3 KB process placed in a 4 KB fixed partition wastes 1 KB	Example: holes of 10 KB, 20 KB and 15 KB cannot satisfy a 30 KB request although 45 KB is free
Occurs in fixed partitioning and paging (last page)	Occurs in variable partitioning and segmentation
Reduced by smaller / variable partition sizes	Reduced by compaction or paging

The **50-percent rule**: statistically, with first fit, for N allocated blocks about $0.5 N$ more blocks are lost to fragmentation, so one-third of memory may be unusable.

5.1.7 Coalescing and Compaction

- **Coalescing** — merging adjacent free holes into a single larger hole when a process terminates.
- **Compaction** — shuffling all occupied memory towards one end so that all free memory forms one large block. It is possible only if relocation is dynamic (done at execution time) and is expensive because much memory must be copied.

5.1.8 Relocation and Protection

When a program is linked, the linker assumes it begins at address 0. If it is loaded at address 100K, every address must be relocated. **Static relocation** modifies the instructions while loading (slow, and it gives no protection). **Dynamic relocation** uses **base and limit registers**: the base is added to every address and the limit register checks that the address is within range — solving relocation and protection simultaneously.

5.1.9 Non-Contiguous Memory Allocation

A process's physical memory may be scattered over several non-adjacent areas. This eliminates external fragmentation and is implemented by **paging, segmentation** or a combination of both — discussed below in virtual memory.

5.2 Virtual Memory (4 Hrs)

5.2.1 Background

Programs are often larger than available memory. In the 1960s programmers used **overlays** manually. **Virtual memory** (Fotheringham, 1961) lets the OS keep only part of the process in main memory, the rest on disk. Each program has its own address space, broken into chunks called **pages**; only the needed pages are in physical memory. The user sees a huge logical memory; the physical memory holds only the working set.

Benefits: programs can be larger than physical memory; more programs can run at once (higher CPU utilisation); less I/O is needed to load or swap programs; simpler programming; sharing of pages between processes.

5.2.2 Paging

Paging divides the **logical address space** into fixed-size blocks called **pages** and **physical memory** into blocks of the same size called **frames**. A process of n pages needs n free frames, which need not be contiguous. A **page table** maps each page number to a frame number. There is **no external fragmentation**, but there can be **internal fragmentation** in the last page (on average half a page per process).

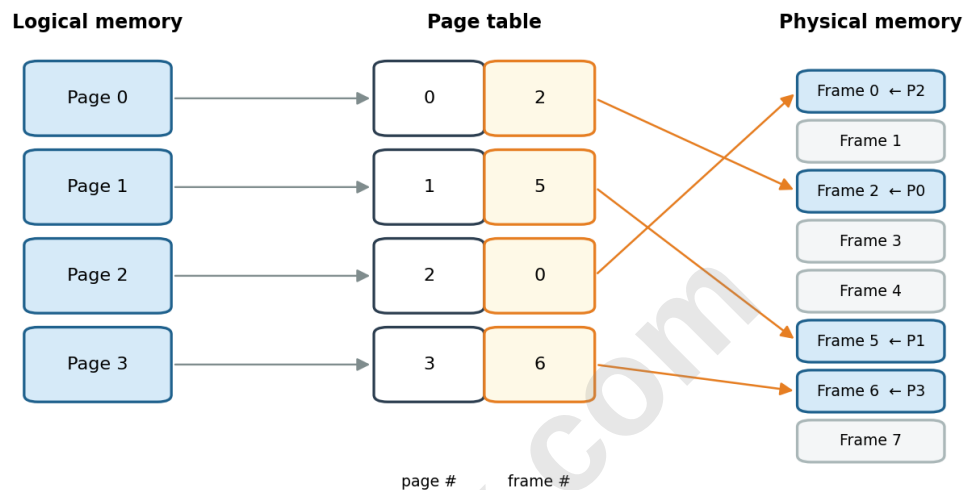


Figure 5.2 — Paging: logical pages mapped to physical frames through a page table

Address translation

A logical address is split into a **page number (p)** — an index into the page table — and a **page offset (d)**. With page size = 2^n and logical address space 2^m : the high-order $(m - n)$ bits are the page number and the low n bits are the offset. Physical address = (frame number $\times$ page size) + offset.

Example — Address translation

Page size = 1 KB (1024 bytes). The page table maps page 0 $\rightarrow$ frame 2, page 1 $\rightarrow$ frame 5, page 2 $\rightarrow$ frame 0, page 3 $\rightarrow$ frame 6. Find the physical address for logical address **2500**.

Solution

Page number = $2500 \div 1024 = 2$ (quotient); offset = $2500 \bmod 1024 = 2500 - 2048 = 452$.

Page 2 $\rightarrow$ frame 0. Physical address = $0 \times 1024 + 452 = 452$.

Translation Look-aside Buffer (TLB)

The page table itself is in memory, so every access takes two memory accesses (one for the page table, one for the data). To speed this up the MMU has a small, fast associative cache called the **TLB** which holds recently used page-table entries. If the page number is found (**TLB hit**) the frame number is available at once; otherwise (**TLB miss**) the page table is consulted and the TLB is updated.

Example — Effective access time (EAT)

TLB lookup = 20 ns, memory access = 100 ns, hit ratio = 80%. Find the EAT.

Solution

$EAT = 0.80 \times (20 + 100) + 0.20 \times (20 + 100 + 100) = 96 + 44 = \mathbf{140 \text{ ns}}$.

A 98% hit ratio would give $0.98 \times 120 + 0.02 \times 220 = \mathbf{122 \text{ ns}}$.

5.2.3 Structure of the Page Table

With a 32-bit address and 4 KB pages there are $2^{20} \approx 1$ million entries per process — too large to keep contiguous in memory. Several schemes reduce this cost:

Scheme	How it works	Used when / features
Hierarchical (multilevel) page table	The page table is itself paged. A two-level scheme splits the page number into p_1 (index into the outer table) and p_2 (index into the inner page table).	Large address spaces; only the needed inner tables are kept in memory (e.g., 32-bit x86: 10 + 10 + 12 bits)
Hashed page table	The page number is hashed to a bucket that holds a chain of entries (page, frame, next). The chain is searched for the matching page number.	Address spaces larger than 32 bits; clustered variants store multiple pages per entry
Inverted page table	One entry <i>per physical frame</i> (not per page) containing (process-id, page number). The table is searched for the pair to find the frame index.	Saves memory (one table for the whole system); search is slow, so hashing is used; sharing pages is difficult (PowerPC, UltraSPARC)
Shared page table / shared pages	Read-only re-entrant code (editors, compilers) is kept in one physical copy; page tables of several processes point to the same frames.	Saves memory; used for shared libraries and copy-on-write after <code>fork()</code>

5.2.4 Block Mapping versus Direct Mapping

Direct mapping	Block mapping
The virtual address is translated to a physical address via a table entry <i>for every single page/word</i>	Addresses are grouped into blocks ; the table maps blocks (a whole page or segment) rather than individual addresses
Very large mapping table — one entry per address unit	Small table — one entry per block; offset within a block is unchanged
Impractical for big address spaces	Practical; used by paging and segmentation

5.2.5 Demand Paging

Pages are loaded into memory **only when needed** (on demand) — a **lazy swapper** (pager) never brings in a page unless it will be needed. Each page-table entry has a **valid–invalid** (present) bit: valid means the page is legal and in memory; invalid means the page is either illegal or currently on disk.

Page fault handling

- **1.** The process references a page whose valid bit is 0 → **trap** to the OS (a **page fault**).
- **2.** The OS checks an internal table: if the reference was illegal → terminate the process; if legal but not in memory, continue.
- **3.** Find a **free frame** (if none, run a page-replacement algorithm).
- **4.** Schedule a disk operation to read the desired page into the frame.
- **5.** When the read finishes, update the page table (frame number, valid bit = 1).
- **6.** Restart the instruction that caused the fault.

Example — Effective access time with page faults

Memory access time = 200 ns; average page-fault service time = 8 ms; page-fault rate $p = 1/1000$. Find EAT.

Solution

$$\text{EAT} = (1 - p) \times 200 + p \times 8,000,000 \text{ ns} = 0.999 \times 200 + 0.001 \times 8,000,000 = 199.8 + 8000 \approx 8200 \text{ ns (8.2 } \mu\text{s)}.$$

The slowdown is a factor of 40. To keep degradation below 10% ($\text{EAT} < 220 \text{ ns}$) the fault rate must be below one fault in about 400,000 accesses.

5.2.6 Page Replacement and Page Faults

When a page fault occurs and **no free frame** exists, the OS must choose a page in memory to remove (the **victim**) and bring in the required page. If the victim was modified, it must first be written back to disk — the **modify (dirty) bit** avoids writing unmodified pages, halving the I/O cost. The goal is the **lowest page-fault rate**. Algorithms are evaluated on a **reference string**.

5.2.7 Page Replacement Algorithms

(1) Optimal Page Replacement (OPR / OPT)

Replace the page that will **not be used for the longest period of time**. It gives the lowest possible fault rate and never suffers Belady's anomaly, but it is **impossible to implement** because the future is unknown; it is used as a yardstick to compare other algorithms.

(2) First-In First-Out (FIFO)

The page that has been in memory the **longest** is replaced. Implemented with a FIFO queue. Easy but may remove heavily used pages; it suffers from **Belady's anomaly** — increasing the number of frames can *increase* the number of page faults.

(3) Least Recently Used (LRU)

Replace the page that has **not been used for the longest time** (looks backward in time — an approximation of OPT). Good performance and no Belady's anomaly (a stack algorithm). Full implementation needs hardware support: a **counter** (time-of-use field) or a **stack** of page numbers (a referenced page is moved to the top). Because this is expensive, approximations such as aging and the second-chance algorithm are used.

(4) Second-Chance (SCP) / Clock Algorithm

A modification of FIFO that inspects the **reference bit R** of the oldest page. If $R = 0$ the page is replaced at once; if $R = 1$ the bit is cleared, the page is moved to the tail of the queue as if it had just arrived (given a "second chance"), and the search continues. If all pages have $R = 1$ the algorithm degenerates into pure FIFO. The **clock** version keeps the frames in a circular list with a hand pointing to the oldest page — avoiding the cost of moving pages around the list.

Other algorithms

- **Not Recently Used (NRU)** — classifies pages by (R, M) bits into 4 classes; removes a random page from the lowest non-empty class.

- **Least Frequently Used (LFU) / Aging** — a software counter records how often (or how recently) the page is used.
- **Working-set / WSClock** — keeps the set of pages used in the last τ references.

5.2.8 Numerical Examples on Page Replacement

Example — Reference string with 3 frames

Reference string: 7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1 (20 references), 3 frames.
Compute the number of page faults for FIFO, LRU and Optimal.

FIFO — page faults = 15

Ref	7	0	1	2	0	3	0	4	2	3
F1	7	7	7	2	2	2	2	4	4	4
F2	–	0	0	0	0	3	3	3	2	2
F3	–	–	1	1	1	1	0	0	0	3
Fault	●	●	●	●		●	●	●	●	●

Ref	0	3	2	1	2	0	1	7	0	1
F1	0	0	0	0	0	0	0	7	7	7
F2	2	2	2	1	1	1	1	1	0	0
F3	3	3	3	3	2	2	2	2	2	1
Fault	●			●	●			●	●	●

LRU — page faults = 12

Ref	7	0	1	2	0	3	0	4	2	3
F1	7	7	7	2	2	2	2	4	4	4
F2	–	0	0	0	0	0	0	0	0	3
F3	–	–	1	1	1	3	3	3	2	2
Fault	●	●	●	●		●		●	●	●

Ref	0	3	2	1	2	0	1	7	0	1
F1	0	0	0	1	1	1	1	1	1	1

Ref	0	3	2	1	2	0	1	7	0	1
F2	3	3	3	3	3	0	0	0	0	0
F3	2	2	2	2	2	2	2	7	7	7
Fault	●			●		●		●		

Optimal — page faults = **9**

Ref	7	0	1	2	0	3	0	4	2	3
F1	7	7	7	2	2	2	2	2	2	2
F2	–	0	0	0	0	0	0	4	4	4
F3	–	–	1	1	1	3	3	3	3	3
Fault	●	●	●	●		●		●		

Ref	0	3	2	1	2	0	1	7	0	1
F1	2	2	2	2	2	2	2	7	7	7
F2	0	0	0	0	0	0	0	0	0	0
F3	3	3	3	1	1	1	1	1	1	1
Fault	●			●				●		

Result

FIFO = **15** faults, LRU = **12** faults, Optimal = **9** faults.

OPT is the minimum possible; LRU is close to OPT; FIFO is the worst of the three here.

Note: ● marks a page fault. Each frame column keeps its slot; a replaced page is overwritten in place.

Example — Belady's anomaly (FIFO)

Reference string: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5. Compare FIFO with 3 frames and 4 frames.

Solution

3 frames → **9** page faults; 4 frames → **10** page faults.

More frames produced **more** faults — this is Belady's anomaly, which never happens with LRU or OPT.

5.2.9 Thrashing

If a process does not have enough frames for its **active working pages** it faults constantly — replacing pages that will be needed again immediately. The process spends more time **paging than executing**; this high paging activity is called **thrashing**. Cause: as CPU utilisation falls, the OS may *increase* the degree of multiprogramming, bringing in more processes, which take frames from others and cause even more faults — utilisation collapses.

- **Working-set model** (Denning): the working set $WS(t, \Delta)$ is the set of pages referenced in the most recent Δ references. If the sum of working-set sizes exceeds the total frames → suspend a process.
- **Page-fault frequency (PFF)**: if the fault rate of a process is above an upper bound, give it more frames; if below a lower bound, take frames away.
- Use **local replacement** and reduce the degree of multiprogramming.

5.2.10 Segmentation

Segmentation supports the user's view of memory: a program is a collection of **segments** — logical units such as main program, procedures, functions, stack, symbol table, arrays. Segments have **variable lengths**. A logical address is a pair **<segment number s, offset d>**. A **segment table** holds, for each segment, a **base** (starting physical address) and a **limit** (length). If $d < \text{limit}$ then physical address = base + d; otherwise a trap is generated.

Example — Segment table

Segment 0: base 1400, limit 1000; Segment 1: base 6300, limit 400; Segment 2: base 4300, limit 400. Translate (2, 53), (1, 500) and (0, 999).

Solution

(2, 53): $53 < 400 \rightarrow \text{physical} = 4300 + 53 = \mathbf{4353}$.

(1, 500): $500 > 400 \rightarrow \mathbf{\text{trap — addressing error}}$.

(0, 999): $999 < 1000 \rightarrow \text{physical} = 1400 + 999 = \mathbf{2399}$.

Paging	Segmentation
Fixed-size pages; invisible to programmer	Variable-size segments; visible to programmer
Address = (page, offset)	Address = (segment, offset)
Internal fragmentation	External fragmentation

Paging	Segmentation
One linear address space	Many separate address spaces; easy sharing and protection
Aim: large linear address space without buying more memory	Aim: logical division for programs and data, sharing, protection

5.2.11 Segmentation with Paging

Combines the advantages of both. The logical address space is divided into segments; **each segment is then paged**. The logical address (s, d) → the segment table gives the base of the *page table* for the segment; d is split into page number and offset; the page table gives the frame. This removes external fragmentation and permits segment-level sharing and protection. Used in MULTICS and Intel x86 (386). The drawback is a more complicated translation with extra memory references, mitigated by the TLB.

Key Takeaways

- ✓ Memory hierarchy trades speed against cost/size; the MMU maps logical to physical addresses (base + limit registers).
- ✓ Swapping uses bitmaps or linked lists; placement algorithms: **first fit, best fit, worst fit, next fit**.
- ✓ Fixed partitioning → **internal** fragmentation; variable partitioning → **external** fragmentation; cure by **compaction**/coalescing or paging.
- ✓ **Paging**: fixed-size pages/frames, page table, TLB; EAT = $\text{hit} \times (t + m) + \text{miss} \times (t + 2m)$.
- ✓ Page-table structures: hierarchical, hashed, inverted, shared.
- ✓ **Demand paging** loads pages lazily; page replacement: **OPT (best), LRU, FIFO (Belady's anomaly), Second Chance**.
- ✓ **Thrashing**: excessive paging; prevent with working-set or PFF control.
- ✓ **Segmentation** gives a logical, variable-size view; **segmentation with paging** combines both.

— End of Unit 5 —