Memory Management

Advanced Operating Systems — Week 6

Hunter College, CUNY

Faye

czhang2@gradcenter.cuny.edu

March 2026

References:
Silberschatz, Galvin, Gagne — *Operating System Concepts*, 10th Ed., Chapters 9–10
https://os-book.com/OS10/slide-dir/index.html
https://www.cs.hunter.cuny.edu/~sweiss

Contents

Part I — Main Memory (Chapter 9)

Background

A program must be brought from disk into memory and placed within a process for it to run. Main memory and registers are the *only* storage the CPU can access directly. The memory unit sees a stream of

- addresses + read requests, or
- addresses + data + write requests.

Register access completes in one CPU clock (or less). Main memory access may take many cycles, causing a **stall**. A **cache** sits between main memory and the CPU registers to bridge this gap.

Memory Protection

Each process must access only addresses in its own address space. A pair of **base** and **limit** registers defines the logical address space:

- The CPU checks every user-mode memory access to ensure it lies between the base and the base + limit.
- Instructions that load the base and limit registers are *privileged*.

Address Binding

Addresses are represented differently at different stages of a program's life:

1. **Source code** — symbolic addresses.
2. **Compiled code** — relocatable addresses (e.g. "14 bytes from the beginning of this module").
3. **Linker / Loader** — binds relocatable addresses to absolute addresses.

Binding can happen at three stages:

Compile time: If the memory location is known *a priori*, absolute code is generated; must recompile if the starting location changes.

Load time: Relocatable code is generated when the memory location is not known at compile time.

Execution time: Binding is delayed until run time if the process can move during execution. Requires hardware support (e.g. base and limit registers).

Logical vs. Physical Address Space

Key Distinction

- **Logical (virtual) address** — generated by the CPU.
- **Physical address** — seen by the memory unit.

Logical and physical addresses are the same in compile-time and load-time binding; they differ in execution-time binding.

Memory-Management Unit (MMU)

The MMU is a hardware device that maps virtual addresses to physical addresses at run time. A simple scheme uses a **relocation register** whose value is added to every address

generated by a user process before it is sent to memory. The user program works entirely with logical addresses and never sees real physical addresses.

Dynamic Loading and Linking

Dynamic Loading

- A routine is not loaded until it is called.
- Unused routines are never loaded, improving memory utilisation.
- No special OS support required; implemented through program design.

Dynamic Linking

- Linking is postponed until execution time.
- A small **stub** locates the appropriate memory-resident library routine and replaces itself with the routine's address.
- Also known as **shared libraries**; useful for patching and versioning.

Contiguous Allocation

Main memory is divided into two partitions:

1. Resident OS (usually low memory, with the interrupt vector).
2. User processes (high memory).

Each process is contained in a single contiguous section.

Variable Partition

- Hole — a block of available memory; holes of various sizes are scattered throughout memory.
- When a process arrives, it is allocated memory from a hole large enough to accommodate it.
- Exiting processes free their partitions; adjacent free partitions are combined.

Dynamic Storage-Allocation Problem

How to satisfy a request of size n from a list of free holes?

First-fit: Allocate the *first* hole that is big enough.

Best-fit: Allocate the *smallest* hole that is big enough (produces the smallest leftover hole).

Worst-fit: Allocate the *largest* hole (produces the largest leftover hole).

First-fit and best-fit generally outperform worst-fit in speed and storage utilisation.

Fragmentation

- **External fragmentation** — total free memory is sufficient but not contiguous.
- **Internal fragmentation** — allocated memory slightly exceeds the request; the difference is wasted.
- **50-percent rule**: given N allocated blocks, roughly $0.5N$ blocks are lost to fragmentation.

- **Compaction** can reduce external fragmentation but requires dynamic relocation at execution time.

Paging

Paging allows the physical address space of a process to be *non-contiguous*.

- Divide physical memory into fixed-size blocks called **frames** (size is a power of 2, typically 2^{12} to 2^{24} bytes).
- Divide logical memory into blocks of the same size called **pages**.
- To run a program of N pages, find N free frames and load the program.
- A **page table** translates logical to physical addresses.

Address Translation Scheme

For a logical address space of 2^m and page size 2^n:

- **Page number** p ($m - n$ bits) — index into the page table.
- **Page offset** d (n bits) — combined with the frame base address to form the physical address.

Paging Example

$n = 2$, $m = 4$: page size = 4 bytes, physical memory = 32 bytes (8 frames).

Calculating Internal Fragmentation

$$\begin{aligned} \text{Page size} &= 2{,}048 \text{ bytes} \\ \text{Process size} &= 72{,}766 \text{ bytes} = 35 \text{ pages} + 1{,}086 \text{ bytes} \\ \text{Internal fragmentation} &= 2{,}048 - 1{,}086 = 962 \text{ bytes} \end{aligned}$$

Worst case: 1 frame − 1 byte. Average: $\frac{1}{2}$ frame size.

Implementation of the Page Table

- The page table is kept in main memory.
- **PTBR** (page-table base register) points to it; **PTLR** (page-table length register) indicates its size.
- Every data/instruction access requires *two* memory accesses (one for the page table, one for the data).

Translation Look-Aside Buffer (TLB)

A fast-lookup hardware cache that solves the two-memory-access problem.

- Typically 64 to 1,024 entries.
- Some TLBs store **ASIDs** (address-space identifiers) to distinguish processes; otherwise the TLB must be flushed on every context switch.
- Some entries can be **wired down** for permanent fast access.

Effective Access Time (EAT)

Let α be the TLB hit ratio and t_m the memory access time:

$$\text{EAT} = \alpha \cdot t_m + (1 - \alpha) \cdot 2t_m$$

Example

$t_m = 10$ ns, hit ratio = 80%: EAT $= 0.80 \times 10 + 0.20 \times 20 = 12$ ns (20% slowdown).
Hit ratio = 99%: EAT $= 0.99 \times 10 + 0.01 \times 20 = 10.1$ ns (1% slowdown).

Memory Protection in Paging

- A **protection bit** with each frame indicates read-only or read-write access.
- A **valid–invalid bit** in each page-table entry: `v` = page is in the process's logical address space; `i` = page is not.
- Violations trigger a trap to the kernel.

Shared Pages

- One copy of read-only (reentrant) code can be shared among processes.
- Private code and data are kept in separate copies for each process.
- Sharing read-write pages enables inter-process communication.

Structure of the Page Table

A 32-bit address space with 4 KB pages yields 2^{20} ($\approx$ 1 million) entries. Solutions to avoid allocating this contiguously:

Hierarchical Paging (Two-Level)

- Page the page table itself.
- A 32-bit logical address with 4 KB pages is split into:
  - p_1 (10 bits) — index into the outer page table.
  - p_2 (10 bits) — displacement within the inner page table.
  - d (12 bits) — page offset.
- Known as a **forward-mapped** page table.
- For 64-bit spaces, even three or four levels may be needed.

Hashed Page Tables

- Common for address spaces > 32 bits.
- The virtual page number is hashed; each bucket contains a chain of $(vpn, frame, next)$ elements.
- **Clustered** variant: each entry refers to several pages (e.g. 16), useful for sparse address spaces.

Inverted Page Tables

- One entry per *real* (physical) frame instead of per logical page.
- Decreases memory for page tables but increases search time.
- A hash table limits the search; TLB accelerates access.

Swapping

- A process can be **swapped out** to a backing store and later **swapped in** for continued execution.
- **Roll out / roll in** — variant for priority-based scheduling.
- Transfer time is proportional to the amount of memory swapped.
- Modern OSes use modified swapping: swap only when free memory is extremely low.
- **Mobile systems** typically do not support swapping (flash limitations). iOS asks apps to free memory voluntarily; Android terminates apps but saves state for fast restart.

Architecture Examples

Intel IA-32

- Supports segmentation and segmentation with paging.
- Up to 16 K segments per process (8 K private via LDT, 8 K shared via GDT); each segment up to 4 GB.
- Page sizes: 4 KB or 4 MB.
- **PAE** (Page Address Extension): 3-level paging, 64-bit entries, extends physical address space to 36 bits (64 GB).

Intel x86-64

- 48-bit virtual addresses (practically), four-level paging hierarchy.
- Page sizes: 4 KB, 2 MB, 1 GB.
- With PAE: 48-bit virtual, 52-bit physical.

ARMv8

- Pages: 4 KB and 16 KB; sections: 1 MB and 16 MB.
- Two levels of TLBs: outer micro-TLBs (data + instruction), inner main TLB.

Part II — Virtual Memory (Chapter 10)

Background

Virtual Memory

Virtual memory is the separation of user logical memory from physical memory.

- Only part of the program needs to be in memory for execution.
- Logical address space can be much larger than physical address space.
- Allows address spaces to be shared by several processes.
- Enables more efficient process creation (e.g. `fork()`).

Implementation: **demand paging** or demand segmentation.

Virtual-Address Space

- Stack starts at the max logical address and grows *down*; heap grows *up*.
- Enables sparse address spaces with holes for growth and dynamically linked libraries.
- Pages can be shared during `fork()`, speeding process creation.

Demand Paging

A page is brought into memory *only when it is needed* (lazy swapper / pager).

- Less I/O, less memory, faster response, more users.
- Uses a **valid–invalid bit** per page-table entry: `v` = in memory, `i` = not in memory.
- Access to an `i`-marked page triggers a **page fault**.

Steps in Handling a Page Fault

1. Trap to the OS.
2. Check an internal table: invalid reference ⇒ abort; not-in-memory ⇒ continue.
3. Find a free frame.
4. Schedule a disk read to swap the page into the frame.
5. Update the page and frame tables (set valid bit to `v`).
6. Restart the instruction that caused the page fault.

Performance of Demand Paging

Let p be the page-fault rate ($0 \le p \le 1$):

$$\text{EAT} = (1 - p) \times t_{\text{mem}} + p \times t_{\text{page-fault}}$$

Example

$t_{\text{mem}} = 200$ ns, average page-fault service time = 8 ms:

$$\begin{aligned}\text{EAT} &= (1-p) \times 200 + p \times 8{,}000{,}000 \\ &= 200 + p \times 7{,}999{,}800\end{aligned}$$

If $p = 1/1000$: EAT = 8.2 μs (40× slowdown).
For $< 10\%$ degradation: $p < 0.0000025$ (< 1 fault per 400,000 accesses).

Copy-on-Write (COW)

- Parent and child initially share the same pages after `fork()`.
- A page is copied only when either process *modifies* it.
- Free pages are allocated from a pool of **zero-fill-on-demand** pages.
- `vfork()` suspends the parent and lets the child use the parent's address space directly (designed for immediate `exec()`).

Page Replacement

When no free frame is available:

1. Find the desired page on disk.
2. Select a **victim frame** using a page-replacement algorithm; write it to disk if dirty (**modify bit**).
3. Load the desired page into the freed frame; update tables.
4. Restart the instruction.

Reference string used in examples:

$$7,\ 0,\ 1,\ 2,\ 0,\ 3,\ 0,\ 4,\ 2,\ 3,\ 0,\ 3,\ 0,\ 3,\ 2,\ 1,\ 2,\ 0,\ 1,\ 7,\ 0,\ 1$$

FIFO Algorithm

- Replace the *oldest* page.
- With 3 frames: 15 page faults.
- **Belady's Anomaly**: adding more frames can *increase* page faults (e.g. string $1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5$).

Optimal (OPT) Algorithm

- Replace the page that will not be used for the *longest* period.
- 9 faults — theoretical lower bound.
- Not implementable in practice (requires future knowledge); used as a benchmark.

Least Recently Used (LRU) Algorithm

- Replace the page not used for the *most* amount of time.
- 12 faults — better than FIFO, worse than OPT.
- Implementations:
  Counter: each entry has a timestamp; search for the smallest.

Stack: doubly linked list of page numbers; referenced page moves to top (6 pointer changes, no search for replacement).

- LRU and OPT are **stack algorithms** — no Belady's Anomaly.

LRU Approximation Algorithms

- **Reference bit**: set to 1 on access; replace any page with bit = 0.
- **Second-chance (clock)**: FIFO with reference bit. If bit = 1, clear it and move on; if bit = 0, replace.
- **Enhanced second-chance**: use $(reference, modify)$ pairs to form four classes; replace from the lowest non-empty class.

Counting Algorithms

- **LFU** (Least Frequently Used): replace the page with the smallest reference count.
- **MFU** (Most Frequently Used): the page with the smallest count was likely just brought in.

Page-Buffering Algorithms

- Maintain a pool of free frames; read the desired page into a free frame immediately and evict the victim later.
- Write modified pages during idle time.
- Keep free-frame contents intact; if re-referenced, no reload needed.

Allocation of Frames

Equal allocation: n frames / k processes each get n/k frames.
Proportional allocation: allocate according to process size.
Priority allocation: use priorities rather than sizes.
Global replacement: a process may take a frame from *any* other process; higher throughput but variable execution time.
Local replacement: a process selects from its own set only; more consistent but may underutilise memory.

Thrashing

Thrashing

A process is busy swapping pages in and out, resulting in very low CPU utilisation. The OS may mistakenly increase the degree of multiprogramming, worsening the situation.

Cause: the total size of localities exceeds available memory.
Mitigation: local or priority page replacement.

Working-Set Model

- Δ = working-set window (fixed number of page references).
- WSS_i = number of pages referenced in the most recent Δ references.

- $D = \sum WSS_i$ = total demand. If $D > m$ (total frames), thrashing occurs; suspend or swap out a process.
- Approximation: interval timer + reference bits.

Page-Fault Frequency (PFF)

- Set an acceptable PFF rate; use local replacement.
- If actual rate is too high, the process gains frames.
- If actual rate is too low, the process loses frames.

Allocating Kernel Memory

Buddy System

- Allocates from physically contiguous pages using a power-of-2 allocator.
- Chunks are recursively split into "buddies" until an appropriate size is available.
- Advantage: fast coalescing. Disadvantage: internal fragmentation.

Slab Allocator

- A **slab** is one or more contiguous pages; a **cache** consists of one or more slabs.
- One cache per unique kernel data structure (e.g. `struct task_struct`).
- Benefits: no fragmentation, fast allocation.
- Linux variants: **SLAB**, **SLOB** (limited memory), **SLUB** (performance-optimised).

Memory Compression

Instead of paging out modified frames to swap space, compress several frames into a single frame. This reduces memory usage without disk I/O overhead.

Other Considerations

Prepaging: load pages before they are referenced to reduce startup faults.
Page size: trade-offs between fragmentation, table size, I/O overhead, and TLB effectiveness.
TLB Reach: = TLB size × page size. Ideally covers the working set.
Program structure: row-major vs. column-major traversal can dramatically affect page-fault rates.
I/O interlock: pages used for device I/O must be **pinned** (locked) in memory.

OS Examples

Windows

- Demand paging with **clustering** (brings in surrounding pages).
- Each process has a working-set minimum (guaranteed) and maximum.
- **Working-set trimming** activates when free memory falls below a threshold.

Solaris

- Thresholds: `lotsfree` (begin paging), `desfree` (increase paging), `minfree` (begin swapping).
- `pageout` process uses a modified clock algorithm.
- Scan rate ranges from `slowscan` to `fastscan` depending on free memory.

Part III — Modern OS Memory Management

The Original Memory Crisis

For decades, OS theory relied on a static hierarchy (Cache → DRAM → Disk). With the explosion of AI workloads, CPU computing power vastly outpaced data movement capabilities—the **"Memory Wall"**. Traditional 4 KB page granularity is too coarse to catch internal Use-After-Free (UAF) and buffer overflow vulnerabilities.

I. Breaking the Memory Wall: HBM & Compute-in-Memory

High Bandwidth Memory (HBM)

- Uses **3D stacking** and silicon interposers to package memory chips directly next to compute units.
- Provides up to **819 GB/s** bandwidth, far exceeding traditional DDR.

New OS Challenges:

Capacity limits: 3D stacking restricts single-stack capacity to 16–32 GB. The OS must manage asymmetric memory topologies.

Thermal walls: Proximity to the CPU makes HBM heat-sensitive. Kernels must implement thermal-aware page scheduling.

Silent corruption: Scaling increases error rates. OS-level SEC-DED fault tolerance is required.

Compute-in-Memory (CIM)

- In traditional architectures, a large proportion of energy is wasted moving data across buses.
- CIM executes **Multiply-Accumulate (MAC)** operations directly inside the memory array.
- The OS treats memory not as passive storage but maps computational instructions directly to it.

II. Tiered Memory via the CXL Protocol

Tier	Latency	Advantage	OS Strategy
Local DRAM	∼70–100 ns	Balanced latency & BW	Default hot-data residency
CXL Expansion	∼150–200 ns	Breaks slot limits	Cold-data demotion target
Stacked HBM	Extremely low	Extreme BW, low energy	Bound to explicit drivers

Hardware 2LM vs. OS-Level zNUMA

Hardware 2LM (inclusive): Local DRAM acts as a hardware cache for CXL memory. Total capacity = CXL capacity only (wastes local DRAM).

OS zNUMA (software): The OS treats CXL memory as an independent "CPU-less" NUMA node. Capacities are additively combined; the OS manages complex page migrations.

Meta's Transparent Page Placement (TPP)

- Proactively demotes cold pages to CXL in the background.
- A 64-bit bitmap system rapidly promotes hot pages back.
- Achieves a **17% throughput boost** over traditional AutoNUMA.

III. Memory Disaggregation

Memory Pooling (Azure Pond)

- About 50% of VMs use less than half their allocated memory ("stranded memory").
- CXL switches build high-speed shared memory pools across server racks.
- ML models predict and dynamically allocate local DRAM vs. pooled CXL memory per VM.

Virtualization Offloading (AWS Nitro)

- Traditional hypervisors consume host memory and CPU caches ("hypervisor tax").
- Nitro offloads hypervisor and SDN functions to custom ASIC cards, returning near 100% of host resources.
- Security bonus: leverages OS hot-unplug to partition physical memory, neutralizing side-channel attacks.

IV. Non-Volatile Memory (NVM)

- NVM is both **byte-addressable** and **persistent**.
- In **App Direct Mode**, the OS bypasses the page cache entirely; standard C/C++ pointers can directly persist data structures.
- Eliminates the traditional storage chain: Disk → DRAM → Modify → Serialize → Kernel → Disk.

V. Rebuilding Kernel Security

The Rust Revolution

- Up to **70%** of modern OS CVEs originate from memory bugs (UAF, dangling pointers) inherent in C/C++.
- **Ownership rules**: every value has a single owner; the compiler automatically inserts release code when the owner goes out of scope.
- **Borrow checker**: "shared immutable, mutable unshared"—static analysis prevents data races at compile time with zero runtime overhead.

VI. ARM Memory Tagging Extension (MTE)

- OS page protections (4 KB granularity) are too coarse to detect intra-page out-of-bounds access.

- MTE partitions physical memory into **16-byte granules**, each assigned a random 4-bit hardware tag.
- On every access, the CPU checks whether the pointer's tag matches the granule's tag, intercepting unauthorized access instantly.

VII. Virtual Threads & Concurrent GC

- **Problem**: 1:1 OS thread mapping traps execution during blocking I/O; old GCs enforce "Stop-The-World" pauses.
- **Virtual threads**: M:N multiplexing decouples from OS kernel limits, supporting millions of concurrent tasks.
- **Concurrent GC**: executes alongside application threads, capping pause times at sub-1 ms, drastically improving tail latencies.

VIII. Formal Verification of Memory Systems

The Ultimate Standard

"Memory management security no longer relies on billions of fuzzing heuristics, but is instead guaranteed by impregnable mathematical theorems."

Examples: the seL4 microkernel (formally verified) and StarMalloc (mathematically proven memory allocator).

Important Dates

1. **Project Idea Submission** — Due: March 9, 2026
   Via email to czhang2@gradcenter.cuny.edu (cc others). Subject: `AOS-idea-your name`.
2. **Proposal Submission** — Due: March 18, 2026
   Structured proposal format, submitted as PDF.
3. **Final Exam Request** (if not doing a project) — Due: March 9, 2026
   Email with subject: `AOS-Final Exam Request`.

About the Proposal

The proposal should:

- Clearly describe the project idea and explain the problem.
- Present the main goal and planned approach.
- Include related work and background literature.
- List relevant engineering resources, implementations, or GitHub repositories.
- Briefly describe expected outcomes and project scope.
- Be realistic and feasible within the course timeline.
- Include a reference list.