Operating Systems Notes - Chapter 9

July 27, 2025

Contents

1 Memory Management

1.1 9.1 Background

Things to learn

- Explain difference between logical and physical address and role of MMU in translating addresses.
- Apply first-, best-, and worst-fit strategies for allocating memory contiguously.
- Explain distinction between internal and external fragmentation.
- Translate logical to physical addresses in paging system with TLB.
- Describe hierarchical paging, hashed paging, and inverted page tables.
- Describe address translation for IA-32, x86-64, and ARM v8 architectures.

Overview

- Main purpose of computer system: execute programs.
- Programs and data must be partially in main memory during execution.
- Modern systems maintain several processes in memory.
- Memory-management schemes vary; effectiveness depends on situation.
- Most algorithms require hardware support.

Introduction

- CPU shared by processes (CPU Scheduling).
- Improves CPU utilization and response speed.
- Requires keeping many processes in memory (sharing memory).
- Discuss various memory management ways (primitive bare-machine to paging).
- Each approach: advantages/disadvantages.
- Selection depends on hardware design.
- Many algorithms require hardware support; integrated hardware/OS memory management common.

Background

- Memory central to modern computer system operation.
- Memory: large array of bytes, each with own address.
- CPU fetches instructions from memory (program counter).
- Instructions may cause loading/storing to memory addresses.
- Instruction-execution cycle: fetch instruction, decode, fetch operands, execute, store results.
- Memory unit sees stream of addresses; doesn't know generation or purpose.
- Focus: sequence of memory addresses generated by running program.
- Issues pertinent to managing memory: basic hardware, binding symbolic/virtual addresses to physical, logical vs. physical addresses.
- Discussion concludes with dynamic linking and shared libraries.

Basic hardware

- Main memory and registers: only general-purpose storage CPU can access directly.
- Instructions/data must be in direct-access storage for CPU operation.
- Registers: accessible within one CPU clock cycle.
- Main memory: accessed via memory bus; may take many CPU cycles.
- Processor may stall waiting for data.
- Remedy: add fast memory (**cache**) between CPU and main memory (often on CPU chip).
- Cache management: hardware automatically speeds up memory access (no OS control).
- Multithreaded core can switch threads during memory stall.
- Concern: correct operation, protection of OS from user processes, user processes from each other.
- Protection by hardware (OS doesn't intervene for performance).
- Each process needs separate memory space for protection and concurrent execution.
- Protection: determine legal address range, ensure access only to legal addresses.
- Implementation: **base register** (smallest legal physical memory address) and **limit register** (size of range).
- CPU hardware compares every address generated in user mode with registers.
- Attempt to access OS/other user memory → trap to OS (fatal error).

- Prevents accidental/deliberate modification of OS/other user code/data.
- Base/limit registers loaded only by OS (privileged instruction, kernel mode).
- OS has unrestricted access to OS/user memory.
- Allows OS to load programs, dump on errors, access system call parameters, perform I/O, context switches.

Address binding

- Program on disk as binary executable.
- To run: brought into memory, placed in process context, eligible for execution.
- Process accesses instructions/data from memory.
- Terminates: memory reclaimed.
- Most systems: user process can reside anywhere in physical memory.
- User program steps before execution (optional):
  - Addresses symbolic (e.g., `count`) in source program.
  - Compiler **binds** symbolic to relocatable ("14 bytes from module start").
  - Linker/loader binds relocatable to absolute (e.g., 74014).
  - Each binding: mapping from one address space to another.
- Binding instructions/data to memory addresses:
  - **Compile time**: If process location known, **absolute code** generated.
  - **Load time**: If process location unknown at compile time, compiler generates **relocatable code**.
  - **Execution time**: If process can move during execution, binding delayed until run time.

Logical versus physical address space

- Address generated by CPU: **logical address**.
- Address seen by memory unit (loaded into memory-address register): **physical address**.
- Compile/load time binding: identical logical and physical addresses.
- Execution-time binding: differing logical and physical addresses.
  - Logical address also called **virtual address**.
  - **Logical address space**: set of all logical addresses generated by program.
  - **Physical address space**: set of all physical addresses corresponding to logical addresses.
- Run-time mapping (virtual to physical): done by **memory-management unit (MMU)**.
- Simple MMU scheme: generalization of base-register.
  - Base register called **relocation register**.
  - Value in relocation register added to every address generated by user process.
- User program never accesses real physical addresses.
- Program deals with logical addresses.
- Memory-mapping hardware converts logical to physical.
- Final location of referenced memory address determined at reference time.
- Two address types: logical (0 to max), physical (R+0 to R+max).
- Logical addresses mapped to physical before use.
- Concept of logical address space bound to separate physical address space: central to proper memory management.

Dynamic loading

- Traditionally: entire program and data in physical memory for execution.
- Process size limited by physical memory size.
- Better memory-space utilization: **dynamic loading**.
- Routine not loaded until called.
- All routines on disk in relocatable load format.
- Main program loaded, executed.
- When routine calls another: checks if loaded. If not, relocatable linking loader loads routine, updates address tables, passes control.
- Advantage: routine loaded only when needed.
- Useful for large code amounts handling infrequent cases (e.g., error routines).
- Total program size large, but portion used (loaded) much smaller.

- Dynamic loading: no special OS support required (user responsibility).
- OS may help by providing library routines for dynamic loading.

Dynamic linking and shared libraries

- **Dynamically linked libraries (DLLs)**: system libraries linked to user programs at run time.
- **Static linking**: system libraries combined by loader into binary program image.
- Dynamic linking: linking postponed until execution time (similar to dynamic loading).
- Usually with system libraries (e.g., standard C library).
- Without DLLs: each program includes copy of language library in executable image.
  - Increases executable size, wastes main memory.
- Second advantage of DLLs: shared among multiple processes (one instance in memory).
  - Also known as **shared libraries**.
  - Extensively used in Windows and Linux.
- Program references dynamic library routine: loader locates DLL, loads if needed.
- Adjusts addresses referencing DLL functions to DLL's memory location.
- DLLs extended to library updates (bug fixes).
- Library replaced by new version: all referencing programs automatically use new version.
- Without dynamic linking: programs need relinking for new library.
- Version information in program/library to prevent incompatible versions.
- Multiple library versions loaded; program uses its version info.
- Minor changes: same version number. Major changes: increment number.
- Only programs compiled with new library version affected by incompatible changes.
- Programs linked before new library: continue using older.
- Dynamic linking/shared libraries: generally require OS help.
- If processes protected: OS checks if routine in another process's memory, allows multiple processes to access same addresses.

Section glossary

Term	Definition
stall	CPU state when CPU waiting for data from main memory, delays execution.
cache	Temporary copy of data in reserved memory area to improve performance.
base register	CPU register with starting address of an address space. Defines logical address space with limit register.
limit register	CPU register defining size of range. Defines logical address space with base register.
bind	Tie together. Compiler binds symbolic to relocatable address.
absolute code	Code with bindings to absolute memory addresses.
relocatable code	Code with bindings to memory addresses changed at loading time to reflect location in main memory.
logical address	Address generated by CPU; translated to physical address before use.
physical address	Actual location in physical memory of code or data.
virtual address	Address generated by CPU; translated to physical address before use.
logical address space	Set of all logical addresses generated by a program.
physical address space	Set of all physical addresses generated by a program.
memory-management unit (MMU)	Hardware component of CPU/motherboard allowing memory access.
MMU	Hardware component of CPU/motherboard allowing memory access.
relocation register	CPU register whose value added to every logical address to create physical address.
dynamic loading	Loading of process routine when called, not at process start.
dynamically linked libraries (DLLs)	System libraries linked to user programs at run time; linking postponed until execution time
static linking	Linking where system libraries treated like object modules, combined by loader into binary program image.
shared libraries	Libraries loaded once, used by many processes; used in systems supporting dynamic linking

1.2 9.2 Contiguous memory allocation

- Main memory accommodates OS and user processes.
- Need efficient memory allocation.
- Memory usually divided into two partitions: one for OS, one for user processes.
- OS can be in low or high memory (many OS, including Linux/Windows, use high memory).
- Several user processes reside in memory concurrently.
- **Contiguous memory allocation**: each process in single contiguous memory section.

Memory protection

- Prevent process from accessing unowned memory.
- Combine **relocation register** (smallest physical address) and **limit register** (range of logical addresses).
- Each logical address must fall within limit register range.
- MMU maps logical address dynamically by adding relocation register value.
- Mapped address sent to memory.
- CPU scheduler loads relocation and limit registers during context switch.
- Every CPU-generated address checked against these registers.
- Protects OS and other user programs/data from modification by running process.
- Relocation-register scheme allows dynamic OS size changes.
- Desirable for device drivers: load only when needed, remove when no longer needed.

Memory allocation

- Simplest method: assign processes to variably sized partitions.
- Each partition contains exactly one process (**variable-partition** scheme).
- OS keeps table of available/occupied memory parts.
- Initially: all memory available for user processes, one large block (**hole**).
- Eventually: memory contains set of holes of various sizes.
- When process arrives: OS considers memory requirements and available space.
- Allocated space: process loaded into memory, competes for CPU time.
- Process terminates: releases memory, OS provides to another process.
- Insufficient memory for arriving process: reject or place in wait queue.
- Memory blocks: set of holes scattered throughout memory.
- Process needs memory: system searches for large enough hole.
- If hole too large: split into two parts (one allocated, other returned to holes).
- Process terminates: releases block, placed back in holes.
- New hole adjacent to others: merged to form larger hole.
- This procedure: instance of **dynamic storage-allocation problem**.
- Common strategies to select free hole:
  - **First-fit**: Allocate first hole big enough. Search from beginning or last search end. Stop when large enough hole found.
  - **Best-fit**: Allocate smallest hole big enough. Must search entire list (unless ordered by size). Produces smallest leftover hole.
  - **Worst-fit**: Allocate largest hole. Must search entire list (unless sorted by size). Produces largest leftover hole (may be more useful).
- Simulations: first-fit and best-fit better than worst-fit (decreasing time, storage utilization).
- Neither first-fit nor best-fit clearly better for storage utilization, but first-fit generally faster.

Fragmentation

- First-fit and best-fit suffer from **external fragmentation**.
- Processes loaded/removed: free memory broken into small pieces.
- External fragmentation: enough total memory, but spaces not contiguous (storage fragmented).
- Problem can be severe: wasted memory between processes.
- If small pieces were one big block: could run more processes.
- Strategy (first-fit/best-fit) affects fragmentation amount.
- Which end of free block allocated also a factor.
- External fragmentation always a problem.

- Statistical analysis (first fit): 0.5 N blocks lost to fragmentation for N allocated blocks.
- **50-percent rule**: one-third of memory unusable.
- Memory fragmentation: internal and external.
- **Internal fragmentation**: unused memory internal to a partition.
- Occurs when allocated memory slightly larger than requested (e.g., fixed-sized blocks).
- Solution to external fragmentation: **compaction**.
- Goal: shuffle memory contents, place all free memory together in one large block.
- Compaction not always possible:
  - If relocation static (assembly/load time): cannot compact.
  - Possible only if relocation dynamic (execution time).
- If dynamic relocation: move program/data, change base register.
- Compaction can be expensive (e.g., move all processes to one end).
- Another solution to external fragmentation: permit noncontiguous logical address space.
- Allows process to be allocated physical memory wherever available.
- Strategy used in **paging** (most common memory-management technique).
- Fragmentation: general problem in computing (storage management chapters).

Section glossary

Term	Definition
contiguous memory allocation	Memory allocation method where each process is in a single contiguous memory section.
variable-partition	Memory-allocation scheme where each memory partition contains exactly one process.
hole	In variable partition memory allocation, a contiguous section of unused memory.
dynamic storage-allocation problem	Problem of satisfying a memory request of size n from a list of free holes.
first-fit	In memory allocation, selecting the first hole large enough for a request.
best-fit	In memory allocation, selecting the smallest hole large enough for a request.
worst-fit	In memory allocation, selecting the largest hole available.
external fragmentation	Fragmentation where available memory has holes that together are enough, but no single hole is large enough.
50-percent rule	Statistical finding that fragmentation may result in 50 percent space loss.
internal fragmentation	Unused memory internal to a partition.
compaction	Shuffling storage to consolidate used space and create one or more large holes.

1.3 9.3 Paging

- Memory management previously required contiguous physical address space.
- **Paging**: memory-management scheme allowing noncontiguous physical address space.
- Avoids external fragmentation and compaction issues.
- Used in most operating systems (servers to mobile devices) due to advantages.
- Implemented through OS and hardware cooperation.

Basic method

- Breaking physical memory into fixed-sized blocks: **frames**.
- Breaking logical memory into same-sized blocks: **pages**.
- Process execution: pages loaded into any available memory frames (from file system or backing store).
- Backing store divided into fixed-sized blocks (same size as frames or clusters).
- Logical address space totally separate from physical address space.
- CPU-generated address divided into two parts:
  - **page number** (**p**)
  - **page offset** (**d**)
- Page number: index into per-process **page table**.
- Page table: contains base address of each frame in physical memory.
- Offset: location in the referenced frame.
- Base address of frame + page offset = physical memory address.
- MMU steps to translate logical to physical address:
  1. Extract page number p, use as index into page table.
  2. Extract corresponding frame number f from page table.
  3. Replace page number p with frame number f.
- Offset d does not change; frame number and offset comprise physical address.
- Page size (like frame size) defined by hardware.
- Page size: power of 2 (typically 4 KB to 1 GB).
- Power of 2 page size: easy translation of logical address to page number/offset.
- Logical address space 2^m, page size 2^n bytes:
  - High-order $m - n$ bits: page number.
  - Low-order n bits: page offset.
- Paging: form of dynamic relocation.
- Every logical address bound by paging hardware to physical address.
- No external fragmentation with paging (any free frame allocated).
- May have internal fragmentation: last frame allocated may not be full.
- Average internal fragmentation: one-half page per process.
- Small page sizes desirable (less internal fragmentation).
- Overhead per page-table entry reduced with larger page sizes.
- Disk I/O more efficient with larger data transfers.
- Page sizes grown over time (processes, data sets, main memory larger).
- Typical page sizes: 4 KB or 8 KB. Some systems support multiple sizes (e.g., Windows 10: 4 KB, 2 MB; Linux: default 4 KB, **huge pages**).
- 32-bit CPU: page-table entry typically 4 bytes.
- 32-bit entry can point to 2^{32} physical page frames.
- Frame size 4 KB (2^{12}): system with 4-byte entries can address 2^{44} bytes (16 TB) physical memory.
- Physical memory size typically different from max logical size of process.
- Page-table entries contain other info, reducing bits for frame addresses.
- Process arrives: size in pages examined. Each page needs one frame.
- If n pages needed, n frames must be available.
- First page loaded into allocated frame, frame number in page table.
- Next page into another frame, its frame number in page table, etc.
- Clear separation: programmer's view of memory vs. actual physical memory.

- Programmer views memory as single space for one program.
- User program scattered throughout physical memory (holds other programs).
- Address-translation hardware reconciles views.
- Logical addresses translated to physical addresses (hidden from programmer, controlled by OS).
- User process cannot access unowned memory (no way to address outside its page table).
- OS manages physical memory: aware of allocation details (allocated/available frames, total frames).
- Information kept in system-wide data structure: **frame table**.
- Frame table: one entry per physical page frame (free/allocated, to which process/page).
- OS aware user processes operate in user space; logical addresses mapped to physical.
- System calls with address parameters: address mapped to correct physical address.
- OS maintains copy of page table for each process (like instruction counter, registers).
- Used for manual logical-to-physical translation by OS.
- Used by CPU dispatcher to define hardware page table when process allocated CPU.
- Paging increases context-switch time.

Hardware support

- Page tables are per-process data structures.
- Pointer to page table stored in process control block (with other registers).
- CPU scheduler selects process: reloads user registers and hardware page-table values from stored user page table.
- Hardware implementation of page table:
  - Simplest: dedicated high-speed hardware registers (efficient translation).
  - Increases context-switch time (registers exchanged).
  - Feasible for small page tables (e.g., 256 entries).
- Contemporary CPUs: much larger page tables (e.g., 2^{20} entries).
- Registers not feasible for large page tables.
- Page table kept in main memory.
- **Page-table base register** (**PTBR**) points to page table.
- Changing page tables: only change PTBR (reduces context-switch time).

Translation look-aside buffer

- Storing page table in main memory: slower memory access times.
- Accessing data: two memory accesses needed (one for page-table entry, one for actual data).
- Memory access slowed by factor of 2 (intolerable).
- Standard solution: special, small, fast-lookup hardware cache: **translation look-aside buffer** (**TLB**).
- TLB: associative, high-speed memory.
- Each TLB entry: key (tag) and value.
- Item presented to associative memory: compared with all keys simultaneously.
- Item found: corresponding value returned (fast search).
- TLB lookup: part of instruction pipeline (no performance penalty).
- TLB must be small (32 to 1,024 entries).
- Some CPUs: separate instruction and data address TLBs (doubles entries).
- TLB used with page tables:
  - CPU generates logical address: MMU checks if page number in TLB.
  - Page number found (**TLB hit**): frame number immediately available, used to access memory.
  - Page number not in TLB (**TLB miss**): memory reference to page table made.
  - Frame number obtained: used to access memory.
  - Page number and frame number added to TLB for future quick reference.
- TLB full: existing entry replaced (LRU, round-robin, random policies).
- Some CPUs allow OS participation in LRU replacement.
- Some TLBs allow entries to be **wired down** (cannot be removed, e.g., kernel code).
- Some TLBs store **address-space identifiers** (**ASIDs**) in each entry.
- ASID: uniquely identifies process, provides address-space protection.

- TLB resolves virtual page numbers: ensures current process ASID matches virtual page ASID.
- ASIDs don't match: treated as TLB miss.
- ASID allows TLB to contain entries for multiple processes simultaneously.
- TLB without ASIDs: must be **flushed** (erased) on each context switch.
- Prevents next process from using wrong translation info (old entries with invalid physical addresses).
- **Hit ratio**: percentage of times page number found in TLB.
- Example: 80% hit ratio, 10 ns memory access.
  - Effective memory-access time = (0.80 * 10 ns) + (0.20 * 20 ns) = 8 ns + 4 ns = 12 ns.
- 99% hit ratio: (0.99 * 10 ns) + (0.01 * 20 ns) = 9.9 ns + 0.2 ns = 10.1 ns.
- Modern CPUs: multiple TLB levels (e.g., Intel Core i7: L1 instruction TLB, L1 data TLB, L2 TLB).
- Miss at L1: check L2. Miss at L2: walk page-table entries in memory (hundreds of cycles) or interrupt OS.
- Hardware features significantly affect memory performance.
- OS improvements (paging) affect and are affected by hardware changes (TLBs).
- TLBs are hardware feature, but OS designers must understand their function/features.
- Optimal OS design for platform: implement paging according to TLB design.
- TLB design changes may necessitate OS paging implementation changes.

Protection

- Memory protection in paged environment: protection bits with each frame (in page table).
- One bit: read-write or read-only page.
- Every memory reference through page table: computes physical address, checks protection bits.
- Attempt to write to read-only page: hardware trap to OS (memory-protection violation).
- Finer protection: read-only, read-write, execute-only (separate bits for each access type).
- Illegal attempts trapped to OS.
- Additional bit in page table entry: **valid-invalid** bit.
- Valid: page in process's logical address space (legal page).
- Invalid: page not in process's logical address space.
- Illegal addresses trapped by valid-invalid bit.
- OS sets this bit for each page to allow/disallow access.
- Example: 14-bit address space (0-16383), program uses 0-10468, page size 2 KB.
- Pages 0-5 mapped normally. Pages 6-7: valid-invalid bit set to invalid → trap to OS.
- Problem: program extends to 10468, but references to page 5 (up to 12287) are valid.
- This reflects internal fragmentation of paging.
- Rarely does process use all address range (many use small fraction).
- Wasteful to create page table for every page in address range if unused.
- Some systems: **page-table length register** (**PTLR**) indicates page table size.
- PTLR value checked against every logical address (verify in valid range).
- Test failure: error trap to OS.

Shared pages

- Advantage of paging: possibility of **sharing** common code.
- Important in multi-process environment (e.g., standard C library `libc`).
- Option: each process loads own copy of `libc` (e.g., 40 processes, 2 MB `libc` → 80 MB memory).
- If code is **reentrant code**: can be shared.
- Reentrant code: non-self-modifying code (never changes during execution).
- Two or more processes can execute same code simultaneously.
- Each process: own copy of registers and data storage.
- Only one copy of `libc` in physical memory.
- Page table for each user process maps to same physical copy of `libc`.
- Significant memory saving (e.g., 2 MB instead of 80 MB for 40 processes).
- Other shared programs: compilers, window systems, database systems.
- Shared libraries (from Dynamic linking section) typically implemented with shared pages.

- Code must be reentrant to be sharable.
- OS should enforce read-only nature of shared code.
- Sharing memory among processes similar to sharing address space by threads.
- Shared memory (interprocess communication) implemented using shared pages.
- Paging provides numerous benefits beyond sharing (covered in Virtual Memory chapter).

Section glossary

Term	Definition
paging	Memory management scheme avoiding external fragmentation by splitting physical memory into fixed-sized frames and logical memory into pages.
frames	Fixed-sized blocks of physical memory.
page	Fixed-sized block of logical memory.
page number (p)	Part of CPU-generated memory address in paged system; index into page table.
page offset (d)	Part of CPU-generated memory address in paged system; offset of location within page.
page table	Table in paged memory containing base address of each physical memory frame, indexed by logical page number.
huge pages	Feature designating region of physical memory for especially large pages.
frame table	Table in paged memory containing frame details (allocated/free, total frames).
page-table base register (PTBR)	CPU register pointing to the in-memory page table.
translation look-aside buffer (TLB)	Small, fast-lookup hardware cache in paged memory address translation for fast access to subset of addresses.
TLB miss	TLB lookup failing to provide address translation because it's not in TLB.
wired down	TLB entry locked into TLB, not replaceable by usual algorithm.
address-space identifier (ASIDs)	Part of TLB entry identifying associated process; causes TLB miss if requesting process ID doesn't match.
flush	Erasure of entries in TLB or other cache to remove invalid data.
hit ratio	Percentage of times a cache provides a valid lookup (e.g., TLB effectiveness measure).
effective memory-access time	Statistical or real measure of CPU time to read/write to memory.
valid-invalid	Page-table bit indicating if entry points to page within process's logical address space.
page-table length register (PTLR)	CPU register indicating size of the page table.
reentrant code	Code supporting multiple concurrent threads (can be shared).

2 Structure of the page table

2.1 Hierarchical paging

- Explores common techniques for structuring the page table: hierarchical paging, hashed page tables, and inverted page tables.
- Modern computer systems support large logical address spaces (2^{32} to 2^{64}).
- Page table itself becomes excessively large.
- **Example**: 32-bit logical address space, 4 KB page size (2^{12}).
  - Page table: over 1 million entries (2^{20}).
  - Each entry: 4 bytes.
  - Each process: up to 4 MB physical address space for page table alone.
  - Solution: divide page table into smaller pieces.
- **Two-level paging algorithm**: page table itself is paged.
  - 32-bit logical address space, 4 KB page size.
  - Logical address divided into:
    - Page number: 20 bits.
    - Page offset: 12 bits.
  - Page number further divided:
    - p_1: 10-bit outer page number (index into outer page table).
    - p_2: 10-bit inner page offset (displacement within inner page table).
  - Address translation: from outer page table inward.
  - Also known as a **forward-mapped** page table.
- **64-bit logical address space**: two-level paging scheme inappropriate.
  - **Example**: 4 KB page size (2^{12}).
  - Page table: up to 2^{52} entries.
  - Inner page tables: one page long, 2^{10} 4-byte entries.
  - Outer page table: still 2^{42} entries, or 2^{44} bytes (16 GB).
- **Three-level paging scheme**: paging the outer page table.
  - Outer page table made up of standard-size pages (2^{10} entries, or 2^{12} bytes).
  - 64-bit address space still daunting.
  - 64-bit UltraSPARC: would require seven levels of paging (prohibitive memory accesses).
  - Hierarchical page tables generally inappropriate for 64-bit architectures.

2.2 Hashed page tables

- Approach for handling address spaces larger than 32 bits.
- Hash value: virtual page number.
- Each entry in hash table: linked list of elements (to handle collisions).
- Each element consists of three fields:
  1. Virtual page number.
  2. Value of mapped page frame.
  3. Pointer to next element in linked list.
- **Algorithm**:
  - Virtual page number in virtual address hashed into hash table.
  - Virtual page number compared with field 1 in first element of linked list.
  - Match: corresponding page frame (field 2) used to form physical address.
  - No match: subsequent entries in linked list searched.
- **Clustered page tables**: variation for 64-bit address spaces.
  - Similar to hashed page tables.
  - Each entry refers to several pages (e.g., 16) instead of a single page.
  - Single page-table entry stores mappings for multiple physical-page frames.
  - Useful for **sparse** address spaces (memory references noncontiguous, scattered).

2.3 Inverted page tables

- Usually, each process has an associated page table.
- Standard page table: one entry for each page process is using (or each virtual address).
- Drawback: each page table may consist of millions of entries, consuming large amounts of physical memory.
- **Inverted page table**: one entry for each real page (frame) of memory.
- Each entry: virtual address of page stored in that real memory location, plus process information.
- Only one page table in system, one entry per physical memory page.
- Often requires an address-space identifier (process-id) in each entry.
- Ensures logical page for particular process maps to corresponding physical page frame.
- **Examples**: 64-bit UltraSPARC and PowerPC.
- **IBM RT simplified version**:
  - Virtual address: `<process-id, page-number, offset>`.
  - Inverted page-table entry: `<process-id, page-number>`.
  - Memory reference: `<process-id, page-number>` presented to memory subsystem.
  - Inverted page table searched for match.
  - Match at entry i: physical address `<i, offset>` generated.
  - No match: illegal address access.
- **Drawback**: increases time to search table (sorted by physical address, lookups by virtual address).
- **Alleviation**: use a hash table to limit search.
- Each access to hash table adds memory reference: one virtual memory reference requires at least two real memory reads (hash-table entry + page table).
- TLB searched first for performance improvement.
- **Shared memory issue**:
  - Standard paging: multiple virtual addresses map to same physical address.
  - Inverted page tables: only one virtual page entry for every physical page.
  - One physical page cannot have two (or more) shared virtual addresses.
  - Reference by another process sharing memory: page fault, replaces mapping.

2.4 Oracle SPARC Solaris

- Modern 64-bit CPU and OS tightly integrated for low-overhead virtual memory.
- **Solaris** running on **SPARC** CPU: fully 64-bit OS.
- Solves virtual memory problem efficiently using hashed page tables.
- Two hash tables: one for kernel, one for all user processes.
- Each maps virtual to physical memory.
- Each hash-table entry: contiguous area of mapped virtual memory (more efficient than per-page entry).
- Entry has base address and span (number of pages represented).
- **TLB (Translation Lookaside Buffer)**: holds translation table entries (TTEs) for fast hardware lookups.
- Cache of TTEs: **translation storage buffer (TSB)**.
- TSB includes entry per recently accessed page.
- **Virtual address reference process**:
  1. Hardware searches TLB for translation.
  2. None found: hardware walks through in-memory TSB for TTE. (**TLB walk**)
  3. Match in TSB: CPU copies TSB entry into TLB, memory translation completes.
  4. No match in TSB: kernel interrupted to search hash table.
  5. Kernel creates TTE from hash table, stores in TSB for automatic loading into TLB by MMU.
  6. Interrupt handler returns control to MMU, completes address translation, retrieves data.

Section glossary

Term	Definition
forward-mapped	Scheme for hierarchical page tables where address translation starts at the outer page table and moves inward.
hashed page table	A page table that is hashed for faster access; the hash value is the virtual page number.
clustered page table	Similar to a hashed page table, but an entry refers to a cluster of several pages.
sparse	In memory management, describes a page table with noncontiguous, scattered entries; an address space with many holes.
inverted page table	A page-table scheme with one entry for each real physical page frame in memory, mapping to a logical page (virtual address) value.
Solaris	A UNIX derivative, main operating system of Sun Microsystems (now Oracle); active open source version called Illumos.
SPARC	A proprietary RISC CPU created by Sun Microsystems (now Oracle); active open source version called OpenSPARC.
TLB walk	Steps involved in traversing page-table structures to locate a needed translation and copying the result into the TLB.

3 Swapping

- Process instructions and data must be in memory for execution.
- Process or portion can be **swapped** temporarily out of memory to a **backing store**.
- Then brought back into memory for continued execution (Figure 9.5.1).
- Swapping allows total physical address space of all processes to exceed real physical memory.
- Increases degree of multiprogramming.

3.1 Standard swapping

- Involves moving entire processes between main memory and backing store.
- Backing store: fast secondary storage, large enough for process parts, direct access to memory images.
- When process/part swapped to backing store, associated data structures (including per-thread data for multithreaded processes) must be written.
- OS maintains metadata for swapped-out processes for restoration.
- Advantage: allows physical memory to be oversubscribed, accommodates more processes than physical memory.
- Idle/mostly idle processes good candidates for swapping.
- Memory allocated to inactive processes can be dedicated to active processes.
- If inactive swapped-out process becomes active, must be swapped back in (Figure 9.5.1).

3.2 Swapping with paging

- Standard swapping generally no longer used in contemporary OS (exception: Solaris under dire circumstances).
- Reason: time to move entire processes prohibitive.
- Most systems (Linux, Windows) use variation: pages of a process swapped, not entire process.
- Still allows physical memory oversubscription.
- Does not incur cost of swapping entire processes (only small number of pages involved).
- Term **swapping** now generally refers to standard swapping.
- **Paging** refers to swapping with paging.
- **Page out**: moves page from memory to backing store.
- **Page in**: reverse process.
- Illustrated in Figure 9.5.2 (subset of pages for processes A and B paged-out/paged-in).
- Works well with virtual memory (Chapter Virtual Memory).

3.3 Swapping on mobile systems

- Mobile systems typically do not support swapping.
- Reasons:
  - Flash memory (not hard disks) for nonvolatile storage → space constraint.
  - Limited number of writes flash memory tolerates before unreliability.
  - Poor throughput between main memory and flash memory.
- Instead of swapping:
  - Apple's iOS: asks applications to voluntarily relinquish allocated memory when free memory low.
    - Read-only data (code) removed from main memory, reloaded from flash if needed.
    - Modified data (stack) never removed.
    - Applications failing to free memory may be terminated by OS.
  - Android: similar strategy, may terminate process if insufficient free memory.
    - Before termination, writes **application state** to flash memory for quick restart.
- Developers for mobile systems must carefully allocate/release memory to avoid excessive use or leaks.

System performance under swapping

- Swapping (any form) often sign of more active processes than available physical memory.
- Two approaches:
  - Terminate some processes.
  - Get more physical memory.

Section glossary

Term	Definition
swapped	Moved between main memory and a backing store. Process swapped out to free mai memory, then swapped back in to continue execution.
backing store	Secondary storage area used for process swapping.
application state	Software construct for data storage.

9.6 Example: Intel 32- and 64-bit Architectures

IA-32 Architecture

- Intel chips: dominated PC landscape for decades.
- 16-bit Intel 8086 (late 1970s), then 8088 (original IBM PC).
- 32-bit chips: **IA-32**, included Pentium processors.
- 64-bit chips: based on **x86-64** architecture.
- Current PC OS (Windows, Mac, Linux) run on Intel chips.
- Intel dominance not in mobile systems; **ARM** architecture successful.
- Focus: major memory-management concepts of Intel CPUs.
- Memory management in IA-32: **segmentation** and **paging**.
- CPU generates **logical addresses** → segmentation unit.
- Segmentation unit produces **linear address** → paging unit.
- Paging unit generates **physical address** in main memory.
- Segmentation and paging units form **memory-management unit (MMU)**.
- See Figure 9.6.1: Logical to physical address translation in IA-32.

IA-32 Segmentation

- IA-32 segment size: up to 4 GB.
- Max segments per process: 16 K.
- Logical address space divided into two partitions:
  - First partition: up to 8 K segments, private to process.
  - Second partition: up to 8 K segments, shared among all processes.
- Information for first partition: **local descriptor table (LDT)**.
- Information for second partition: **global descriptor table (GDT)**.
- Each LDT/GDT entry: 8-byte **segment descriptor**.
- Segment descriptor: detailed info (base location, limit).
- Logical address: (selector, offset).
- Selector: 16-bit number.
  - s: segment number.
  - g: GDT or LDT.
  - p: protection.
- Offset: 32-bit number, byte location within segment.
- Machine has six segment registers: allows six segments addressed at once.
- Six 8-byte microprogram registers: hold descriptors (LDT/GDT cache).
- Cache avoids reading descriptor from memory for every reference.
- Linear address (IA-32): 32 bits long.
- Segment register points to LDT/GDT entry.
- Base and limit from segment descriptor generate linear address.
- Limit checks address validity; invalid → memory fault (trap to OS).
- Valid: offset added to base → 32-bit linear address.
- See Figure 9.6.2: IA-32 segmentation.

IA-32 Paging

- IA-32 page size: 4 KB or 4 MB.
- For 4-KB pages: two-level paging scheme.
- 32-bit linear address division:
  - Page number $p1$: 10 bits (high-order).
  - Page number $p2$: 10 bits (inner).
  - Page offset d: 12 bits (low-order).
- See Figure 9.6.3: Paging in the IA-32 architecture.
- 10 high-order bits reference entry in outermost page table: **page directory**.
- `CR3` register points to page directory for current process.

- Page directory entry points to inner page table (indexed by inner 10 bits).
- Low-order bits 0-11: offset in 4-KB page.
- Page directory entry: `Page_Size` flag.
- If `Page_Size` set: page frame is 4 MB (bypasses inner page table).
- 22 low-order bits in linear address: offset in 4-MB page frame.
- IA-32 page tables can be swapped to disk for efficiency.
- Invalid bit in page directory entry: indicates table in memory or on disk.
- If on disk: OS uses other 31 bits for disk location; table brought into memory on demand.
- 4-GB memory limitations of 32-bit architectures led to **page address extension (PAE)**.
- PAE: allows 32-bit processors to access physical address space > 4 GB.
- PAE changes paging from two-level to three-level scheme.
- Top two bits refer to **page directory pointer table**.
- See Figure 9.6.4: Page address extensions (PAE also supports 2-MB pages).
- PAE increased page-directory and page-table entries from 32 to 64 bits.
- Allowed base address of page tables/frames to extend from 20 to 24 bits.
- Combined with 12-bit offset: PAE increased address space to 36 bits.
- Supports up to 64 GB physical memory.
- OS support required for PAE (Linux, Mac support; 32-bit Windows desktop → 4 GB limit).

X86-64

- Intel's initial 64-bit architecture: **IA-64** (later **Itanium**), not widely adopted.
- AMD developed **x86-64**: extended existing IA-32 instruction set.
- x86-64: supported larger logical/physical address spaces, architectural advances.
- Intel adopted AMD's x86-64 architecture.
- Use general term **x86-64** (instead of AMD 64, Intel 64).
- 64-bit address space: potentially 2^{64} bytes (16 quintillion / 16 exabytes).
- In practice: fewer than 64 bits used for address representation.
- x86-64 architecture: 48-bit virtual address.
- Supports page sizes: 4 KB, 2 MB, or 1 GB.
- Uses four levels of paging hierarchy.
- See Figure 9.6.5: x86-64 linear address.
- Addressing scheme uses PAE.
- Virtual addresses 48 bits, support 52-bit physical addresses (4,096 terabytes).

Section Glossary

Term	Definition
page directory	In Intel IA-32 CPU architecture, the outermost page table.
page address extension (PAE)	Intel IA-32 CPU hardware allowing 32-bit processors to access physical address space large than 4GB.
PAE	Intel IA-32 CPU hardware allowing 32-bit processors to access physical address space large than 4GB.
page directory pointer table	PAE pointer to page tables.
Itanium	Intel IA-64 CPU.
AMD 64	A 64-bit CPU designed by Advanced Micro Devices; part of x86-64 class.
Intel 64	Intel 64-bit CPUs, part of x86-64 class.
x86-64	Class of 64-bit CPUs running identical instruction set; common in desktop/server systems

4 ARMv8 architecture

Overview

- ARM processors: common for mobile devices (smartphones, tablets)
- Intel: designs and manufactures chips
- ARM: only designs, licenses architectural designs to manufacturers
- Examples: Apple (iPhone, iPad), most Android devices use ARM
- Also for real-time embedded systems
- Over 100 billion ARM processors produced; most widely used architecture by quantity
- Focus: 64-bit ARM v8 architecture

Translation Granules, Pages, and Regions

- ARM v8 has three **translation granules**: 4 KB, 16 KB, and 64 KB
- Each granule provides different page sizes and larger contiguous memory sections called **regions**

Translation Granule Size	Page Size	Region Size
4 KB	4 KB	2 MB, 1 GB
16 KB	16 KB	32 MB
64 KB	64 KB	512 MB

Paging Structure

- 4-KB and 16-KB granules: up to four levels of paging
- 64-KB granules: up to three levels of paging
- ARM v8 is 64-bit architecture, but only 48 bits currently used
- **TTBR register**: **translation table base register**, points to level 0 table for current thread
- If all four levels used (4-KB granule): offset (bits 0-11) refers to offset within 4-KB page
- Table entries for level 1 and level 2 may refer to another table or a region:
  - Level-1 table refers to 1-GB region: low-order 30 bits (0-29) used as offset
  - Level-2 table refers to 2-MB region: low-order 21 bits (0-20) used as offset

TLBs (Translation Lookaside Buffers)

- ARM architecture supports two levels of TLBs:
  - Inner level: two **micro TLBs** (one for data, one for instructions); support ASIDs
  - Outer level: single **main TLB**
- Address translation process:
  - Begins at micro-TLB level
  - Micro-TLB miss: main TLB checked
  - Both TLBs miss: page table walk performed in hardware

Section glossary

Term	Definition
translation granules	Features of ARM v8 CPUs defining page sizes and regions.
regions	In ARM v8 CPUs, contiguous memory areas with separate privilege and access rules.
translation table base register	ARM v8 CPU register pointing to the level 0 (outer) page table for the current thread.
micro TLB	ARM CPU inner-level TLBs, one for instructions and one for data.
main TLB	ARM CPU outer-level TLB; checked after micro TLB lookup and before page table walk.

5 Summary

- Memory: central to modern computer systems; large array of bytes, each with own address.
- Address space allocation: using **base and limit registers**.
- **Base register**: smallest legal physical memory address.
- **Limit**: specifies size of address range.
- Binding symbolic address references to physical addresses:
  - Compile time
  - Load time
  - Execution time
- **Logical address**: generated by CPU.
- **Memory Management Unit (MMU)**: translates logical address to **physical address**.
- Memory allocation approach: contiguous memory partitions of varying sizes.
- Partition allocation strategies:
  - **First fit**
  - **Best fit**
  - **Worst fit**
- Modern OS: use **paging** to manage memory.
- **Physical memory**: divided into fixed-sized blocks called **frames**.
- **Logical memory**: divided into blocks of same size called **pages**.
- Paging: logical address divided into **page number** and **page offset**.
- **Page number**: index into per-process **page table**.
- **Page table**: contains frame in physical memory holding the page.
- **Offset**: specific location in the frame.
- **Translation Look-aside Buffer (TLB)**: hardware cache of page table.
- Each TLB entry: page number and corresponding frame.
- TLB in address translation:
  - Get page number from logical address.
  - Check if frame for page is in TLB.
  - If in TLB: frame obtained from TLB.
  - If not in TLB: retrieve from page table.
- **Hierarchical paging**: logical address divided into multiple parts for different page table levels.
- Problem with expanding addresses (beyond 32 bits): large number of hierarchical levels.
- Strategies to address this: **hashed page tables** and **inverted page tables**.
- **Swapping**: moves pages to disk to increase degree of multiprogramming.
- Intel 32-bit architecture: two levels of page tables; supports 4-KB or 4-MB page sizes.
- **Page-address extension**: allows 32-bit processors to access physical address space ¿ 4 GB.
- x86-64 and ARM v8 architectures: 64-bit architectures using hierarchical paging.