Operating Systems
Tutorial 9: Memory Management

Yonghao Lee

June 2, 2025

1 Fundamental Concepts and Calculations

Understanding memory management requires familiarity with basic computational units and binary arithmetic. This section establishes the foundational knowledge necessary for subsequent topics.

1.1 Memory Units and Powers of Two

Memory sizes in computer systems are typically expressed as powers of two:

- KB = 2^{10} bytes = 1,024 bytes
- MB = 2^{20} bytes = 1,048,576 bytes
- GB = 2^{30} bytes = 1,073,741,824 bytes

1.2 Essential Calculations

1.2.1 Memory Division Example

To calculate how many 8KB blocks fit in 4GB:

$$\frac{4\text{GB}}{8\text{KB}} = \frac{4 \times 2^{30}}{8 \times 2^{10}} = \frac{2^2 \times 2^{30}}{2^3 \times 2^{10}} \tag{1}$$

$$= \frac{2^{32}}{2^{13}} = 2^{32-13} = 2^{19} = 524,288 \text{ blocks} \tag{2}$$

1.2.2 Binary Representation

Representing values with n bits: An n-bit number can represent 2^n different values, ranging from 0 to $2^n - 1$.

Example: An 8-bit number can represent $2^8 = 256$ values (0 through 255).

1.2.3 Number Base Conversions

Binary to Decimal: To convert binary 1101 to decimal:

$$1101_2 = 1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 \tag{3}$$

$$= 8 + 4 + 0 + 1 = 13_{10} \tag{4}$$

Decimal to Binary: To convert decimal 13 to binary using successive division:

$$13 \div 2 = 6 \text{ remainder } 1 \tag{5}$$

$$6 \div 2 = 3 \text{ remainder } 0 \tag{6}$$

$$3 \div 2 = 1 \text{ remainder } 1 \tag{7}$$

$$1 \div 2 = 0 \text{ remainder } 1 \tag{8}$$

Reading remainders from bottom to top: $13_{10} = 1101_2$

2 CPU Architecture and Instruction Processing

The Central Processing Unit (CPU) serves as the computational core of a computer system, executing instructions that manipulate data stored in the memory hierarchy. Understanding CPU operation is fundamental to comprehending memory management requirements.

2.1 CPU Instruction Categories

The CPU executes a well-defined set of instructions organized into several categories:

2.1.1 Data Movement Instructions

- **Load**: Transfer data from memory to CPU registers
- **Store**: Transfer data from CPU registers to memory
- **Move**: Copy data between registers or assign immediate values

2.1.2 Arithmetic and Logic Operations

- **Arithmetic**: Addition, subtraction, multiplication, division
- **Bitwise operations**: AND, OR, XOR, NOT, bit shifts and rotations
- **Comparison**: Compare values and set condition flags for decision making

2.1.3 Control Flow Instructions

- **Unconditional branch**: Jump to a specific memory address (GOTO-like behavior)
- **Conditional branch**: Jump based on condition flags or register comparisons (IF-like behavior)
- **Function calls**: Jump to subroutines with automatic return address management

2.2 Memory Hierarchy and Data Flow

The CPU operates within a multi-level memory hierarchy designed to balance speed, capacity, and cost:

Registers ↔ Cache ↔ RAM ↔ Storage

Data must flow through this hierarchy before the CPU can perform operations. Specifically, data must be brought from RAM into CPU registers (typically through cache) before arithmetic or logic operations can be performed. This architectural constraint creates the fundamental requirement that programs must be loaded from disk storage into memory before execution can begin.

2.3 Program Execution Model

The CPU follows the fetch-decode-execute cycle for each instruction:

1. **Fetch**: Retrieve the next instruction from memory using the program counter
2. **Decode**: Interpret the instruction opcode and identify required operands
3. **Execute**: Perform the operation, potentially involving data movement between memory hierarchy levels
4. **Update**: Modify program counter and condition flags as needed for the next cycle

This cycle repeats continuously during program execution, with the memory management unit facilitating address translation during memory accesses.

3 Address Spaces and Memory Organization

Modern computer systems distinguish between physical and virtual memory to provide process isolation, memory protection, and efficient resource utilization.

3.1 Physical Address Space

A **physical address** is the actual address of a specific storage cell in main memory (RAM). The physical address space is determined by the system's total memory size and addressing capabilities.

3.1.1 Address Space Calculation

The number of address bits required depends on the total addressable memory:
For a system with 8GB of RAM:

$$\text{Required address bits} = \log_2(8 \text{ GB}) = \log_2(8 \times 2^{30} \text{ bytes}) \tag{9}$$

$$= \log_2(2^3 \times 2^{30}) = \log_2(2^{33}) = 33 \text{ bits} \tag{10}$$

This allows addressing from 0 to $2^{33} - 1$ (0 to 8GB - 1 byte), where each address points to a single byte in memory.

3.1.2 Architecture-Based Addressing Limitations

Computer systems are designed around standard architectures that determine both register size and addressing capabilities:
32-bit Architecture:

- CPU registers are 32 bits (4 bytes) in width
- Address registers can hold 32-bit addresses
- Maximum addressable memory: 2^{32} addresses = 4GB

- Since each address points to 1 byte, total addressable space = 4GB of RAM

64-bit Architecture:

- CPU registers are 64 bits (8 bytes) in width
- Address registers can hold 64-bit addresses
- Theoretical maximum addressable memory: 2^{64} addresses = 16 exabytes
- Practical implementations often limit this to 48 bits (2^{48} = 256 TB)

3.2 Virtual Memory Abstraction

Processes are generally unaware of the memory usage of other processes or their own exact physical memory locations. To enable this abstraction, the operating system provides each process with a virtual address space where the process perceives it has vast memory resources available and operates as if it were the only process running on the system.

3.2.1 Virtual Address Space (VAS)

The **virtual address space** refers to the set of virtual address ranges that an operating system makes available to a process, also known as the logical address space.

- **32-bit OS**: VAS size = 2^{32} = 4 GB per process
- **64-bit OS**: Theoretical maximum = 2^{64} bytes per process
- **Practical 64-bit systems**: Often limited to 48 bits = 2^{48} = 256 TB per process

When a program uses a pointer to reference a function or variable, that pointer contains a virtual address from the process's VAS. The virtual addresses appear continuous to the process, while the corresponding physical memory locations may be fragmented or non-contiguous.

3.2.2 Virtual-to-Physical Mapping Challenge

The virtual address space of a single process is typically much larger than the actual physical memory available in the system. This raises a fundamental question: how is this made possible?

If a process were to utilize its entire address space—which rarely happens in practice—a direct one-to-one mapping between virtual and physical memory would be impossible. To address this limitation, the operating system utilizes **swap space**, a region on secondary storage that extends the effective memory capacity beyond physical RAM through demand paging techniques.

3.3 Memory Management Unit (MMU)

User programs operate exclusively with logical (virtual) addresses and never directly access real physical addresses. The **Memory Management Unit (MMU)** is the specialized hardware component responsible for translating virtual addresses to their corresponding physical addresses transparently and efficiently.

The MMU enables the operating system to implement sophisticated memory management policies while maintaining the illusion that each process has its own private, contiguous memory space.

4 Paging: Fundamental Mechanism

Paging is the primary technique used by modern operating systems to implement virtual memory. It allows a process's logical address space to be noncontiguous in physical memory while maintaining the illusion of a continuous address space for the process.

4.1 Core Paging Concepts

Under the paging scheme:

- **Physical memory** is partitioned into fixed-size blocks called **frames**
- **Logical memory** is partitioned into blocks of the same size called **pages**
- The operating system maintains a record of all available free frames
- A **page table** establishes the mapping between logical pages and physical frames

To execute a program requiring n pages, the system must locate n free frames in physical memory and load the program's pages into these frames. The page table then provides the translation mechanism between the process's virtual addresses and actual physical memory locations.

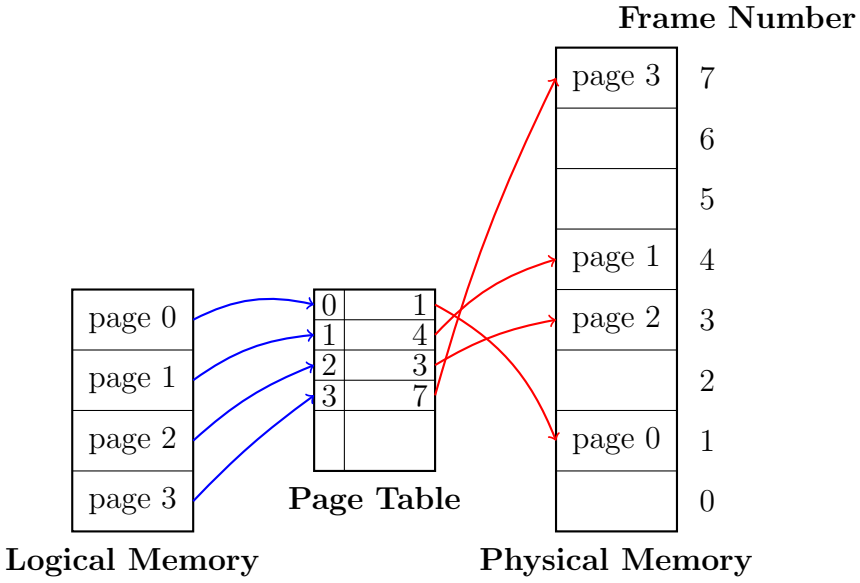
4.2 Benefits of Paging

Paging provides several key advantages:

- **Eliminates external fragmentation**: Pages fit exactly into frames of the same size
- **Enables non-contiguous allocation**: Logical pages can map to any available physical frames
- **Simplifies memory allocation**: No need to find large contiguous blocks
- **Bounded internal fragmentation**: Limited to at most one page size when the last page is partially filled

4.3 Paging Example

The following diagram illustrates how paging enables noncontiguous memory allocation. A process with four logical pages (0-3) is mapped to physical memory frames through a page table. Notice how the logical pages, which appear contiguous to the process, are actually scattered across different physical frames.



When the CPU generates a logical address, the MMU uses the page table to translate it to the corresponding physical address. For example:

- Logical page 0 is stored in physical frame 1
- Logical page 1 is stored in physical frame 4
- Logical page 2 is stored in physical frame 3
- Logical page 3 is stored in physical frame 7

5 Address Translation Mechanism

The fundamental operation of paging relies on systematically dividing virtual addresses into components that facilitate efficient translation to physical addresses.

5.1 Address Structure

An address generated by the CPU is divided into two distinct components:

- **Page number (p)**: Used as an index into the page table to locate the corresponding frame number
- **Page offset (d)**: Combined with the frame number to define the exact physical memory address

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

- **Page offset**: n bits (to address any word within a page)
- **Page number**: $(m - n)$ bits (to identify which page)

5.2 Bit Allocation Rationale

Page offset requires n bits:

- Each page contains 2^n words
- To address any word within a page, we need exactly n bits
- Example: 4KB page = 2^{12} bytes, so $n = 12$ bits for offset

Page number requires $(m - n)$ bits:

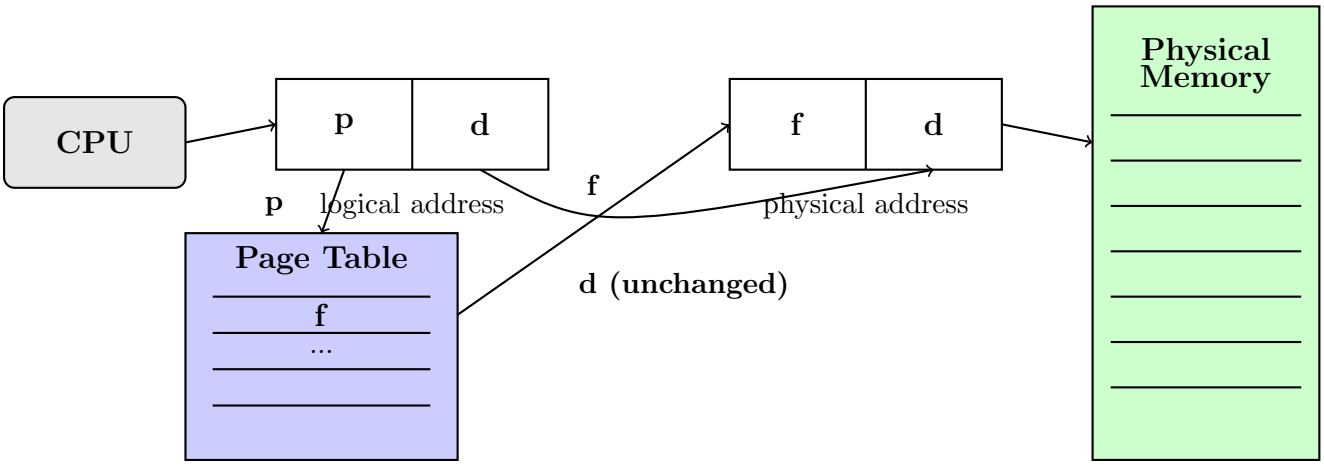
- Total address space = 2^m words
- Each page = 2^n words
- Number of pages = $\frac{2^m}{2^n} = 2^{m-n}$
- To identify any page, we need $(m - n)$ bits

5.2.1 Practical Example: 32-bit System with 4KB Pages

- $m = 32$ bits (4GB total address space)
- $n = 12$ bits (4KB = 2^{12} bytes per page)
- Page number: $32 - 12 = 20$ bits (can address $2^{20} = 1$M pages)
- Page offset: 12 bits (can address 4096 bytes within each page)

5.3 Hardware Address Translation

The MMU hardware automatically translates virtual addresses to physical addresses using the page table lookup mechanism.



Translation Process:

1. CPU generates logical address with page number **p** and offset **d**
2. Page number **p** serves as an index into the page table
3. Page table entry at index **p** contains frame number **f**
4. Physical address = frame number **f** concatenated with offset **d**
5. Memory access occurs at the computed physical address

6 Page Table Entries and Control Information

Page Table Entries (PTEs) store more than just frame numbers. They include control bits that manage memory protection, virtual memory operations, and performance optimization.

6.1 Essential Control Bits

6.1.1 Valid/Present Bit

- **Purpose**: Indicates whether the page is currently assigned to a physical frame
- **If 0**: Page is stored on disk (swap space), triggers page fault exception
- **If 1**: Page is resident in RAM, address translation can proceed normally

6.1.2 Modified/Dirty Bit

- **Purpose**: Tracks whether the page has been written to since being loaded
- **Usage**: Optimization for page replacement—clean pages don't need to be written back to disk
- **Impact**: Dirty pages must be saved to storage before being replaced, adding I/O overhead

6.1.3 Used/Referenced/Accessed Bit

- **Purpose**: Records recent page access for replacement algorithms
- **Application**: Used by algorithms like the clock algorithm for page replacement
- **Function**: Helps identify which pages are actively used versus unused

6.1.4 Access Permissions

- **Read-only**: Suitable for data pages and executable code pages
- **Read-write**: Required for stack, heap, and modifiable data structures
- **Execute**: Enables code pages to be run by the processor
- **Combinations**: Read-execute for program code, read-write for data sections

These control bits enable the operating system to implement sophisticated memory management policies including demand paging, copy-on-write, memory protection, and efficient page replacement strategies.

7 Page Table Implementation Challenges

As address spaces grow larger, the space requirements for page tables become a significant concern. This section examines the challenges and solutions for implementing page tables efficiently.

7.1 Space Requirements Analysis

Consider a modern 64-bit system with the following parameters:

- **Virtual address space**: 2^{48} bytes = 256 TB (typical for x86-64)
- **Page size**: 4 KB = 2^{12} bytes = 4,096 bytes

7.1.1 Calculating Page Table Size

$$\text{Number of pages} = \frac{\text{Total virtual space}}{\text{Page size}} = \frac{2^{48}}{2^{12}} = 2^{36} \text{ pages} \tag{11}$$

$$\text{Required PTEs} = 2^{36} \text{ entries (one per page)} \tag{12}$$

7.1.2 PTE Size Requirements

Critical Analysis: Why 1 Byte per PTE is Impossible

A PTE must store sufficient information to identify any physical frame in the system:
Frame number storage requirements:

- Number of possible frames = $\frac{\text{Physical memory size}}{\text{Frame size}}$
- For a 48-bit physical address space: $\frac{2^{48}}{2^{12}} = 2^{36}$ possible frames
- To represent 2^{36} different frame numbers requires **at least 36 bits**
- 1 byte = 8 bits can represent only $2^8 = 256$ values
- This is insufficient by a factor of 2^{28} (over 268 million times too small)

Additional space requirements: PTEs also need bits for valid/present, dirty, referenced, and permission flags.
Realistic PTE size: 8 bytes (64 bits) in modern 64-bit systems

7.1.3 Corrected Page Table Size Calculation

$$\text{Total page table size} = 2^{36} \times 8 \text{ bytes} = 2^{39} \text{ bytes} = \mathbf{512\ GB} \tag{13}$$

This demonstrates why **flat page tables are impractical** for large address spaces, necessitating more sophisticated approaches.

7.2 Solutions: Hierarchical Page Tables

To address the space efficiency problem, modern systems employ hierarchical (multi-level) page tables that break up the logical address space mapping into multiple, smaller page tables.

7.2.1 Advantages of Hierarchical Paging

- **Sparse allocation**: Only page table sections currently in use need to be allocated in memory
- **Reduced memory footprint**: Eliminates the need to store entire page tables for unused address ranges
- **Scalability**: Supports very large address spaces efficiently
- **Locality exploitation**: Programs often use their address space sparsely, making this approach highly effective

7.3 Two-Level Page Table Example

This example demonstrates a two-level paging system on a 32-bit machine with 1KB pages.

7.3.1 System Parameters

- **Machine Architecture**: 32-bit
- **Page Size**: 1KB = 2^{10} bytes

7.3.2 Address Structure

A 32-bit logical address is divided as follows:

- **Page offset (d)**: 10 bits (addresses 1024 bytes within a page)
- **Page number**: 22 bits (total page identification)

Since the page table itself is paged, the 22-bit page number is further subdivided:

- p_1 **(Outer page table index)**: 12 bits (index into outer page table)
- p_2 **(Inner page table offset)**: 10 bits (offset within inner page table page)

7.3.3 Complete Address Structure

The 32-bit logical address structure:

p_1 (12 bits)	p_2 (10 bits)	d (10 bits)

Total: $12 + 10 + 10 = 32$ bits

7.3.4 Translation Process

1. **First level**: p_1 indexes into the outer page table to find a page containing part of the inner page table
2. **Second level**: p_2 serves as an offset within that inner page table page to locate the specific PTE
3. **Frame access**: The PTE contains the physical frame number
4. **Final address**: Frame number + offset d yields the physical address

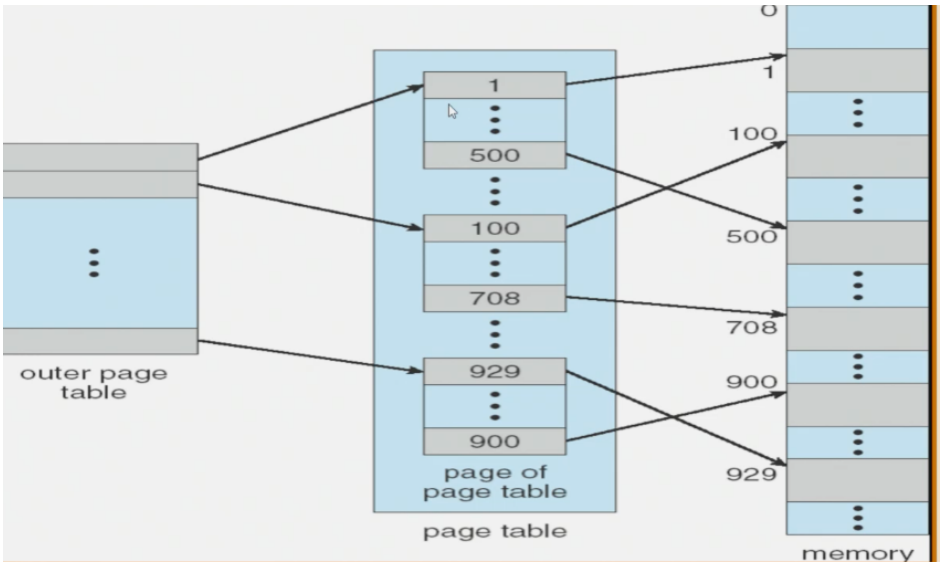


Figure 1: Two-Level Page Table Structure

This hierarchical approach significantly reduces memory requirements by allocating only the portions of the page table that correspond to actively used regions of the virtual address space.

8 Page Replacement and Virtual Memory

When physical memory becomes full and new pages must be loaded, the operating system must decide which existing pages to evict. This decision process, known as page replacement, is critical to virtual memory performance.

8.1 Page Fault Handling

When the valid bit in a page table entry is clear (0), indicating the page is not currently mapped to any frame, a memory access to that page triggers a hardware exception called a **page fault**.

8.1.1 Page Fault Response Sequence

1. **Exception generation**: MMU detects invalid page and transfers control to the OS
2. **Page location**: OS locates the required page in secondary storage (swap space)
3. **Frame allocation**: OS finds an available physical frame (may require eviction)
4. **I/O operation**: Page content is loaded from storage into the allocated frame
5. **Page table update**: PTE is updated with new frame number and valid bit set
6. **Instruction restart**: The faulting instruction is re-executed successfully

8.2 Page Replacement Algorithms

To choose the appropriate victim for eviction when no free frames are available, the operating system employs page replacement algorithms. The goal is to minimize future page faults by retaining pages that are likely to be accessed soon.

8.2.1 Common Replacement Strategies

- **Clock Algorithm**: Uses the referenced bit to give pages a "second chance" before eviction
- **Least Recently Used (LRU)**: Evicts the page that was accessed furthest in the past
- **Not Recently Used (NRU)**: Groups pages by reference and dirty bits, preferring clean, unreferenced pages
- **Working Set**: Attempts to keep pages that are part of the process's current working set in memory

These were discussed in lecture no. 9.