Operating Systems
Tutorial 10: Memory Management Continued

Yonghao Lee

June 2, 2025

1 Review: Foundation Concepts

1.1 Physical vs. Virtual Addresses

Definition 1.1: *A **physical address** is the actual address of a storage cell in main memory. The physical address space is determined by the total memory size installed in the system.*

Example 1.1: *With 8 GB of RAM, we need 33 bits for physical addressing:*

$$8\ GB = 8 \times 2^{30}\ bytes = 2^{33}\ bytes \tag{1}$$

This explains why 32-bit operating systems can only utilize 4 GB of RAM.

1.2 Virtual Memory Management

Key Concept 1.1: *Since processes are unaware of other processes' memory usage, the OS provides each process with a **virtual address space**. The Memory Management Unit (MMU) handles the mapping between virtual memory (VM) and physical memory (PM).*

1.3 Paging Fundamentals

Definition 1.2: *The logical address space is partitioned into fixed-size blocks called **pages**, while corresponding physical memory blocks are called **frames**. The mapping between pages and frames is stored in a **page table**, where each entry is a **Page Table Entry (PTE)**.*

***Note:** CPU-generated addresses are divided into a page number p (used as an index into the page table) and a page offset d (displacement within the page).*

As the number of processes increases, the percentage of memory devoted to page tables grows significantly. Hierarchical paging addresses this scalability issue.

2 Hierarchical Paging

Hierarchical paging reduces memory overhead by organizing page tables in a tree-like structure, where higher-level tables point to lower-level tables, and only the top-level table needs to remain in memory at all times.

2.1 Two-Level Hierarchical Paging

Example 2.1 (Two-Level Paging System)**:** *Consider a system with the following specifications:*

- *32-bit virtual address space*
- *Page size: 4 KB (2^{12} bytes)*
- *Page table entry size: 4 bytes*

2.1.1 Address Structure

The 32-bit virtual address is partitioned as follows:

10 bits	10 bits	12 bits
P1 (Page Directory)	P2 (Page Table)	Page Offset

2.1.2 Translation Process

The address translation occurs in three steps:

1. **Page Directory Access:**
   - First 10 bits (P1) index into the page directory
   - Maximum $2^{10} = 1024$ entries
   - Each entry points to a second-level page table
2. **Page Table Access:**
   - Next 10 bits (P2) index into the selected page table
   - Each page table contains 1024 entries
   - Each entry points to a physical frame
3. **Physical Address Formation:**
   - Last 12 bits specify the byte offset within the page
   - $2^{12} = 4096$ bytes per page

2.2 Four-Level Hierarchical Paging

For larger address spaces, more levels are required. Modern 64-bit systems often use four-level paging.

Example 2.2 (48-bit Address Space with 4-Level Paging)**:** *System specifications:*

- *48-bit virtual address space*
- *Page size: 4 KB (2^{12} bytes)*
- *Page table entry size: 8 bytes*

2.2.1 Design Constraints and Calculations

Constraint: Each page table must fit exactly within one page.

$$\text{Entries per page table} = \frac{\text{Page size}}{\text{PTE size}} \tag{2}$$

$$= \frac{4096 \text{ bytes}}{8 \text{ bytes}} = 512 \text{ entries} \tag{3}$$

Bits needed per level:

$$2^n = 512 \implies n = 9 \text{ bits per level} \tag{4}$$

Number of levels required:

$$\text{Non-offset bits} = 48 - 12 = 36 \text{ bits} \tag{5}$$

$$\text{Number of levels} = \frac{36}{9} = 4 \text{ levels} \tag{6}$$

2.2.2 Address Structure

9 bits	**9 bits**	**9 bits**	**9 bits**	**12 bits**
PML4	PDPT	PD	PT	Offset
Level 4	Level 3	Level 2	Level 1	Page Offset

3 Alternative Approach: Inverted Page Tables

Traditional page tables scale poorly with virtual address space size. Inverted page tables offer a fundamentally different approach to address translation.

3.1 Concept and Motivation

Key Concept 3.1: *An **inverted page table** maintains exactly one entry per physical frame, shared among all processes. Each entry contains the virtual page number and process ID of the page currently stored in that frame.*

Definition 3.1: *Entry i in an inverted page table specifies which virtual page is currently stored in physical frame i, creating a one-to-one mapping between physical frames and table entries.*

3.2 Advantages and Trade-offs

Advantages:

- Memory usage scales with physical memory size, not virtual address space
- Single table shared among all processes
- Eliminates per-process page table overhead

Disadvantages:

- Requires linear search through all entries: $O(n)$ lookup time vs. $O(1)$ in traditional paging
- Cannot map to non-existent physical addresses (no swap space mapping)
- Severe cache pollution from linear searches

3.3 Address Translation Process

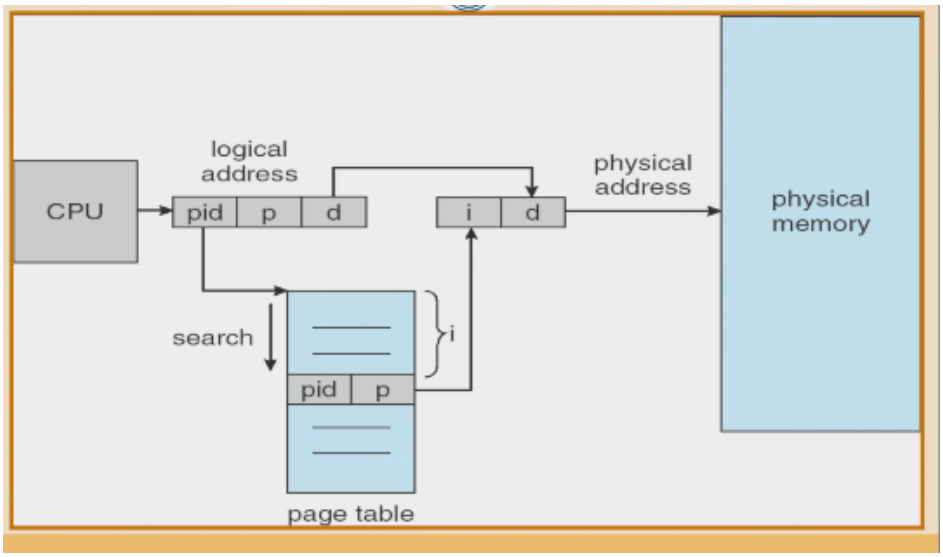

Figure 1: Inverted Page Table Architecture

The translation process involves three steps:

3.3.1 Step 1: Address Generation

The CPU generates a logical address containing:

- **pid**: Process identifier
- **p**: Virtual page number
- **d**: Page offset

3.3.2 Step 2: Table Search

Unlike direct indexing in traditional page tables:

1. Linear search through inverted page table entries
2. Each entry format: `[pid | p]`
3. Search for matching process ID and virtual page number
4. Continue until match found or table exhausted

3.3.3 Step 3: Address Formation

Upon finding a match at index i:

- Frame number = i
- Physical address = `[frame number | d]`

- Page offset remains unchanged

Note: *The linear search creates significant cache pollution, loading thousands of page table entries into CPU cache and evicting useful program data, creating a double performance penalty.*

4 Translation Lookaside Buffer (TLB)

The Translation Lookaside Buffer addresses the fundamental performance challenge of virtual memory systems: the multiple memory accesses required for address translation.

4.1 The Performance Problem

Key Concept 4.1: *Every memory access in a paged system traditionally requires multiple physical memory accesses:*

- ***Single-level paging****: 2 accesses (page table + data)*
- ***Multi-level paging****: 3-5 accesses (multiple page table levels + data)*

4.2 TLB Architecture and Operation

Definition 4.1: *The* ***Translation Lookaside Buffer (TLB)*** *is an associative cache that stores recently used virtual-to-physical address translations, implemented in dedicated hardware on the CPU chip.*

4.2.1 Hardware Implementation Advantages

The TLB's efficiency stems from its hardware-based design:

- **Associative Memory**: Uses content-addressable memory (CAM) that searches all entries in parallel
- **On-Chip Location**: Resides on the CPU chip, eliminating memory bus latency
- **Parallel Search**: Can examine all 64+ entries simultaneously, not sequentially
- **Dedicated Circuits**: Search logic implemented in electronic circuits, not software

4.3 Complete Address Translation Process

The address translation process always begins with a TLB lookup, regardless of the underlying paging scheme.

4.3.1 Step 1: TLB Consultation

For any virtual address containing page number p and offset d:

- The TLB is searched for a cached translation of page p
- This lookup occurs using parallel associative search
- Hardware determines hit or miss in a single cycle

4.3.2 Step 2A: TLB Hit (Optimized Path)

When the translation is found in the TLB:

1. Frame number is retrieved directly from the TLB entry
2. Physical address is formed: `[frame number | d]`
3. Memory access proceeds immediately
4. **Total memory accesses: 1**

4.3.3 Step 2B: TLB Miss (Page Table Walk)

When the translation is not cached in the TLB, the system performs the complete page table lookup:

For Single-Level Paging:

1. Access page table entry for page p
2. Extract frame number from PTE
3. Form physical address: `[frame number | d]`
4. Access target data/instruction
5. Update TLB with the new translation
6. **Total memory accesses: 2**

For Multi-Level Paging:

1. Traverse each level of the page table hierarchy
2. Access each intermediate page table in sequence
3. Extract frame number from the final PTE
4. Form physical address: `[frame number | d]`
5. Access target data/instruction
6. Update TLB with the new translation
7. **Total memory accesses: 3-5 (depending on hierarchy depth)**

Key Concept 4.2: *The TLB serves as a learning cache: each miss triggers a complete page table walk, but the resulting translation is cached to accelerate future accesses to the same page.*

4.4 Effective Access Time Analysis

Definition 4.2: ***Effective Access Time (EAT)*** *represents the average time cost of accessing a page, accounting for both TLB hits and misses with their respective probabilities.*

4.4.1 Mathematical Framework

Let:

- h = TLB hit ratio (probability of finding translation in TLB)
- t_{tlb} = TLB access time
- t_{mem} = Memory access time
- t_{pt} = Page table access time (multiple memory accesses for multi-level)

General EAT Formula:

$$\text{EAT} = h \times (t_{tlb} + t_{mem}) + (1 - h) \times (t_{tlb} + t_{pt} + t_{mem}) \tag{7}$$

For single-level paging where $t_{pt} = t_{mem}$:

$$\begin{aligned} \text{EAT} &= h \times (t_{tlb} + t_{mem}) + (1 - h) \times (t_{tlb} + 2t_{mem}) && (8)\\ &= t_{tlb} + t_{mem} + (1 - h) \times t_{mem} && (9)\\ &= t_{tlb} + t_{mem}(2 - h) && (10) \end{aligned}$$

Example 4.1 (EAT Calculation)**:** *Consider a system with:*

- *TLB access time: 1 ns*
- *Memory access time: 100 ns*
- *TLB hit ratio: 95%*

$$EAT = 1 + 100(2 - 0.95) = 1 + 105 = 106 \ ns \tag{11}$$

Without TLB: $2 \times 100 = 200$ *ns*
Performance improvement: 47% reduction in access time

Note: *The hardware-based parallel search capability explains why TLB consultation is always the preferred first step in address translation, despite needing to search through multiple entries.*

5 Copy-On-Write (CoW)

Copy-On-Write is a crucial optimization technique that dramatically improves process creation efficiency and memory utilization.

5.1 Concept and Motivation

Key Concept 5.1: *Copy-On-Write defers the actual copying of memory pages until a write operation occurs, allowing multiple processes to safely share read-only copies of the same physical memory.*

This approach addresses the inefficiency of traditional process creation, where entire address spaces are copied immediately upon `fork()`, often unnecessarily.

5.2 Traditional Fork() vs. Copy-On-Write

Traditional Approach Problems:

- Immediate copying of all parent process pages
- Many pages may never be modified by either process
- Time-consuming for large address spaces
- Significant memory overhead for duplicate pages

Copy-On-Write Solution: CoW optimizes process creation by initially sharing physical memory between parent and child, copying only when necessary.

5.3 CoW Implementation Mechanism

1. **Initial Setup (fork() call):**
   - Child process receives an identical virtual address space
   - Both parent and child page tables point to the same physical frames
   - All shared pages are marked as **read-only** in both page tables
   - OS maintains reference counts for each physical frame
2. **Read Operations:**
   - Both processes can read from shared pages without restriction
   - No copying occurs during read-only access
   - Memory usage remains minimal
3. **Write Operations (Copy Trigger):**
   - When any process attempts to write to a shared page:
     (a) Hardware detects write to read-only page
     (b) Page fault exception is triggered
     (c) OS page fault handler intervenes
     (d) OS checks frame reference count
     (e) If reference count > 1: allocate new frame and copy content

(f) If reference count = 1: mark page as writable (no copy needed)
(g) Update writing process's page table
(h) Write operation retries and succeeds

5.4 Reference Counting and Shared Pages

Key Concept 5.2: *The key principle: A page can only be written to directly if exactly one process is using it (reference count = 1). Multiple processes sharing a page require copying before any write can proceed.*

Scenarios:

- **Reference count = 1**: Direct write allowed, no copying
- **Reference count > 1**: Copy required before write

Example 5.1 (CoW with Multiple Processes)**:** *Parent + Child A + Child B all share Frame 10:*

Initial state: *All point to Frame 10 (reference count = 3)*
Child A writes:

- *Child A gets new Frame 25*
- *Parent + Child B still share Frame 10 (reference count = 2)*

Parent writes:

- *Parent gets new Frame 30*
- *Child B alone has Frame 10 (reference count = 1)*

Child B writes:

- *Direct write to Frame 10 (no copy needed)*