Chapter 9: Main Memory

Calculation Problems

Operating System Concepts, 10th Edition
Silberschatz, Galvin, Gagne

Contents

1 Relocation Register: Logical to Physical Address

Problem 1 — MMU with Relocation Register (Slide 13)

The MMU uses a **relocation register** to translate logical addresses to physical addresses. The formula is:

$$\text{physical address} = \text{logical address} + \text{relocation register}$$

Given:

- Logical address generated by CPU: **346**
- Relocation register value: **14000**

Calculate the physical address.

Solution:

$$\text{physical address} = 346 + 14000 = \mathbf{14346}$$

Instructor Note

The user program only ever sees logical address 346. If the OS later moves this process to a different region of memory, it simply updates the relocation register — say, to 20000 — and the same logical address 346 would now map to physical address $346 + 20000 = 20346$. The user program does not need to change at all.

2 Base/Limit Register Protection and Translation

Problem 2 — Hardware Support for Relocation and Limit Registers (Slide 18)

The hardware uses a **limit register** for bounds checking and a **relocation register** for address translation. The process is:

1. Check: is logical address < limit register? If not, trap.
2. If valid: physical address = relocation register + logical address.

Given:

- Logical address generated by CPU: **1200**
- Limit register: **2000**
- Relocation register: **100000**

(a) Is the access valid?
(b) If valid, what is the physical address?
(c) What happens if the logical address were 2500 instead?

Solution:
(a) Check: $1200 < 2000$? Yes $\Rightarrow$ access is **valid**.
(b) physical address $= 100000 + 1200 = \mathbf{101200}$
(c) Check: $2500 < 2000$? No, $2500 \geq 2000 \Rightarrow$ the comparator triggers a **trap to the OS** (addressing error). Access denied.

Instructor Note

The limit register protects against out-of-bounds access; the relocation register translates from logical to physical. The two work together: limit check first, then relocation.

3 Paging Address Translation

Problem 3 — Paging: Splitting Logical Addresses (Slides 25–27)

Background — Deriving the formula:
Suppose the logical address space is 2^m bytes and the page size is 2^n bytes.

- **Offset bits:** n bits (to address every byte within a page).
- **Page number bits:** $m - n$ bits.
- **Number of pages:** 2^{m-n}.
- **Physical address formula:** physical address $= \text{frame}(p) \times 2^n + d$

Given:

- $n = 2$ (offset bits), so page size $= 2^2 = 4$ bytes.
- $m = 4$ (total address bits), so logical address space $= 2^4 = 16$ bytes.
- Page number bits $= m - n = 4 - 2 = 2$ bits, so $2^2 = 4$ pages (pages 0–3).
- Physical memory $= 32$ bytes $= 8$ frames of 4 bytes each.
- Page table: page 0 $\rightarrow$ frame 5, page 1 $\rightarrow$ frame 6, page 2 $\rightarrow$ frame 1, page 3 $\rightarrow$ frame 2.

Translate the following logical addresses to physical addresses.

Solution:

Example 1: Logical address **0** in binary is `00 00`. $p = 0$, $d = 0$. Page 0 $\rightarrow$ frame 5.

$$\text{physical address} = 5 \times 4 + 0 = \mathbf{20}$$

Example 2: Logical address **3** in binary is `00 11`. $p = 0$, $d = 3$. Page 0 $\rightarrow$ frame 5.

$$\text{physical address} = 5 \times 4 + 3 = \mathbf{23}$$

Example 3: Logical address **4** in binary is `01 00`. $p = 1$, $d = 0$. Page 1 $\rightarrow$ frame 6.

$$\text{physical address} = 6 \times 4 + 0 = \mathbf{24}$$

Example 4: Logical address **13** in binary is `11 01`. $p = 3$, $d = 1$. Page 3 $\rightarrow$ frame 2.

$$\text{physical address} = 2 \times 4 + 1 = \mathbf{9}$$

Instructor Note

Consecutive logical addresses 0–3 all fall in page 0 and map to physical addresses 20–23 (frame 5). But logical address 4 jumps to frame 6 (physical address 24). This demonstrates how paging allows noncontiguous allocation. The offset d passes through unchanged — only the page number is replaced by the frame number.

4 Internal Fragmentation Calculation

Problem 4 — Calculating Internal Fragmentation (Slide 28)

Given: page size = 2,048 bytes, process size = 72,766 bytes.
Calculate the internal fragmentation.

Solution:
Step 1: How many full pages does the process need?

$$\left\lfloor \frac{72{,}766}{2{,}048} \right\rfloor = \lfloor 35.53 \rfloor = 35 \text{ full pages}$$

Step 2: How many bytes remain after filling 35 pages?

$$72{,}766 - 35 \times 2{,}048 = 72{,}766 - 71{,}680 = 1{,}086 \text{ bytes}$$

Step 3: These 1,086 bytes need one more page (the 36th page). That page is 2,048 bytes, but we only use 1,086. The wasted space is:

$$\text{internal fragmentation} = 2{,}048 - 1{,}086 = \mathbf{962} \text{ bytes}$$

General analysis:
- **Worst case:** process needs just 1 byte beyond a page boundary $\Rightarrow$ waste = page size − 1.
- **Best case:** process size is exact multiple of page size $\Rightarrow$ waste = 0.
- **Average case:** $\frac{1}{2} \times$ page size.

Instructor Note

Trade-off: smaller pages reduce internal fragmentation but increase the number of page table entries (more memory overhead). For example, a 4 GB address space with 4 KB pages needs $2^{20} \approx 1$ million entries; with 4 MB pages, only $2^{10} = 1{,}024$ entries.

5 Effective Access Time (EAT) with TLB

Problem 5 — Effective Access Time (Slide 34)

Definitions:

- α = TLB hit ratio (probability of finding the page number in the TLB).
- t_m = memory access time.
- TLB lookup time is negligible (done in hardware, in parallel).

Two cases on every memory reference:

- **TLB hit** (probability α): access memory once for data. Time $= t_m$.
- **TLB miss** (probability $1-\alpha$): access memory once for page table + once for data. Time $= 2t_m$.

General formula:

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

Example 1: $t_m = 10$ ns, $\alpha = 80\%$:

$$\begin{aligned}\text{EAT} &= 0.80 \times 10 + 0.20 \times 2 \times 10\\ &= 8 + 4 = \mathbf{12} \text{ ns}\end{aligned}$$

Slowdown $= \dfrac{12-10}{10} = 20\%$.

Example 2: $t_m = 10$ ns, $\alpha = 99\%$:

$$\begin{aligned}\text{EAT} &= 0.99 \times 10 + 0.01 \times 20\\ &= 9.9 + 0.2 = \mathbf{10.1} \text{ ns}\end{aligned}$$

Slowdown $= \dfrac{10.1-10}{10} = 1\%$.

Instructor Note

With a realistic TLB hit ratio of 99% or higher, the overhead of paging is negligible. The TLB is the single most important piece of hardware for making paging practical.

6 EAT for Multi-Level Paging

Problem 6 — EAT with Two-Level Paging (Slide 42)

With two-level paging, a TLB miss requires **3 memory accesses** (outer page table → inner page table → data) instead of 2.

Formula for two-level paging:

$$\text{EAT} = \alpha \cdot t_m + (1 - \alpha) \cdot 3t_m$$

Given: $\alpha = 99\%$, $t_m = t_m$.

Solution:

$$\text{EAT} = 0.99 \times t_m + 0.01 \times 3t_m = 0.99\,t_m + 0.03\,t_m = \mathbf{1.02\,t_m}$$

Overhead is only 2% — the TLB makes the extra levels almost free.

Instructor Note

Without a TLB, every memory reference in two-level paging costs 3 memory accesses. The TLB is absolutely critical for multi-level paging to be practical.

7 Two-Level Page Table Structure

Problem 7 — Two-Level Paging Derivation (Slide 41)

Given: 32-bit machine, page size = 4 KB.
Derive the two-level page table structure.

Solution:
Step 1: Page size = 4 KB = 2^{12} bytes ⇒ offset $d = 12$ bits.
Page number = $32 - 12 = 20$ bits. Page table has $2^{20} \approx 1$ million entries.
Step 2: Page the page table itself. Each entry is 4 bytes. One page (2^{12} bytes) holds:

$$\frac{2^{12}}{4} = 2^{10} = 1{,}024 \text{ entries}$$

Step 3: Split the 20-bit page number:

- $p_1 = 10$ bits — index into the **outer page table** ($2^{10} = 1{,}024$ choices).
- $p_2 = 10$ bits — index **within** that page of the inner page table ($2^{10} = 1{,}024$ entries per page).

Result: The 32-bit logical address is structured as:

$$\underbrace{p_1}_{\text{10 bits}} \;\Big|\; \underbrace{p_2}_{\text{10 bits}} \;\Big|\; \underbrace{d}_{\text{12 bits}}$$

Instructor Note

The outer page table has only 1,024 entries (fits in one page). This is called a forward-mapped page table. For 64-bit systems, even two-level paging is not sufficient — the outer table would still be 16 TB. This is why x86-64 uses 4-level paging with only 48 bits implemented.

8 64-bit Page Table Size Analysis

Problem 8 — Why Two-Level Paging Fails for 64-bit (Slide 43)

Given: 64-bit addresses, page size = $4\,\text{KB} = 2^{12}$.
Show why two-level paging is insufficient.

Solution:
Page number bits $= 64 - 12 = 52$ bits $\Rightarrow$ page table has 2^{52} entries (≈ 4.5 quadrillion!).
Try two-level: Each inner page holds 2^{10} entries, so $p_2 = 10$ bits.
Outer page number $= 52 - 10 = 42$ bits.
Outer page table size $= 2^{42} \times 4 = 2^{44}$ bytes = **16** TB. Still enormous!
Try three-level: Split the 42-bit outer into two levels of 21 bits each.
Second outer table has $2^{21} \approx 2$ million entries — better but still very large, and now requires **4 memory accesses** per reference (without TLB).

Instructor Note

This is why 64-bit architectures like x86-64 use **4-level paging** and only implement 48 bits of the 64-bit address space in practice. Each additional level trades one more memory access for a dramatic reduction in individual table sizes.

9 Inverted Page Table Size

Problem 9 — Inverted Page Table Size (Slide 48)

Given: A system with 4 GB of RAM and 4 KB pages.
How many entries does the inverted page table have?

Solution:

$$\text{Number of physical frames} = \frac{4\,\text{GB}}{4\,\text{KB}} = \frac{2^{32}}{2^{12}} = 2^{20} \approx \mathbf{1} \text{ million entries}$$

Instructor Note

The inverted page table size is proportional to *physical memory*, not the virtual address space. The same 1 million entries apply regardless of whether the virtual address space is 32 bits or 64 bits. However, linear search is slow — in practice, a hash table is placed in front for $O(1)$ average lookup.

10 Context Switch Time with Swapping

Problem 10 — Context Switch Time Including Swapping (Slide 54)

Given:

- Process size = 100 MB.
- Disk transfer rate = 50 MB/s.

Calculate the total swap time for a context switch.

Solution:

Swap-out time (write current process to disk):

$$t_{\text{out}} = \frac{100\text{ MB}}{50\text{ MB/s}} = 2{,}000\text{ ms} = 2\text{ seconds}$$

Swap-in time (read new process from disk):

$$t_{\text{in}} = \frac{100\text{ MB}}{50\text{ MB/s}} = 2{,}000\text{ ms} = 2\text{ seconds}$$

Total swap component of context switch:

$$t_{\text{total}} = t_{\text{out}} + t_{\text{in}} = 2{,}000 + 2{,}000 = \mathbf{4{,}000}\text{ ms} = \mathbf{4}\text{ seconds}$$

Optimization: If the OS knows the process only uses 10 MB out of 100 MB:

$$t_{\text{total}} = \frac{10}{50} + \frac{10}{50} = 200 + 200 = \mathbf{400}\text{ ms}$$

Still slow, but 10× better.

Instructor Note

A typical context switch without swapping takes microseconds. 4 seconds is enormous! System calls like `request_memory()` and `release_memory()` help the OS know how much memory is actually in use, enabling the optimization above.

11 IA-32 PAE Address Structure

Problem 11 — Intel IA-32 Page Address Extensions (Slide 64)

Compare the address structures with and without PAE.

Without PAE (standard IA-32, 2-level):

$$\underbrace{\text{Dir}}_{10} \;\Big|\; \underbrace{\text{Table}}_{10} \;\Big|\; \underbrace{\text{Offset}}_{12} \quad = 32 \text{ bits} \Rightarrow 4\,\text{GB physical}$$

With PAE (3-level):

$$\underbrace{\text{PDPT}}_{2} \;\Big|\; \underbrace{\text{Dir}}_{9} \;\Big|\; \underbrace{\text{Table}}_{9} \;\Big|\; \underbrace{\text{Offset}}_{12} \quad = 32 \text{ bit virtual}$$

Key changes:

1. New top level: **Page Directory Pointer Table (PDPT)**, 2 bits $\Rightarrow$ 4 entries.
2. Page table entries expanded from 32 bits to 64 bits (wider frame numbers).
3. Physical address is now **36 bits**: $2^{36} = \mathbf{64}\,\text{GB}$ addressable physical memory.

Instructor Note

The virtual address is still 32 bits (application sees 4 GB), but page table entries can point to physical frames anywhere in 64 GB. The OS gives different processes access to different 4 GB windows of the larger physical memory.