Distributed Systems

Christian J. Rudder

January 2025

Contents

This page is left intentionally blank.

Big thanks to **Professor Ioannis Liagouris**
and **Dr. Anna Arpaci-Dusseau** for teaching CS351: Distributed Systems
at Boston University [5].

All illustration contain original assets.

Disclaimer: These notes are my personal understanding and interpretation of the course material. They are not officially endorsed by the instructor or the university. Please use them as a supplementary resource and refer to the official course materials for accurate information.

Prerequisites

This text assumes the reader has a basic understanding of computer science and programming. It will also assume they are somewhat familiar with computer architecture and operating systems at a high level. The text will review these concepts briefly for completeness, but it will not try to teach them from scratch or provide a full understanding of these topics.

The main focus will be on distributed systems, and will touch on:

- **Concurrency and Parallelism**
  - Concurrency, Parallelism, Threads
- **Consistency and Fault Tolerance**
  - Consistency, Fault-tolerance, Atomicity
- **Distributed Systems and Coordination**
  - Asynchrony, Coordination, Logical Time, Snapshots
- **Consensus Algorithms**
  - Raft, Paxos, Consensus
- **Replication and Data Management**
  - Replication, Sharding, Cluster
- **Protocols and Computing Models**
  - RPC, 2PC, Broadcast
- **Technologies and Tools**
  - MapReduce, Spanner, Dynamo, GFS, TLA+, Golang

— 1 —

Introduction

1.1 Virtual Memory*

1.1.1 Problem Space

One may skip this section if they are already familiar with pages, or have reached the section detailing TLBs. The rest is offered for completeness. Virtual memory solves three problems [3]:

- **Not enough memory, Memory fragmentation, and Security**

Definition 1.1: Not Enough Memory

Back then, computer memory was expensive, and many computers had very little memory (e.g., 4–1 GiB or even less). CPUs could only support up to 4 GiB of memory, as CPUs were 32-bit (2^{32} addresses $= 2^{32} bytes = 4GiB$). On the other hand, 64-bit CPUs can support up to 2^{64} addresses = 16 million TB of memory.

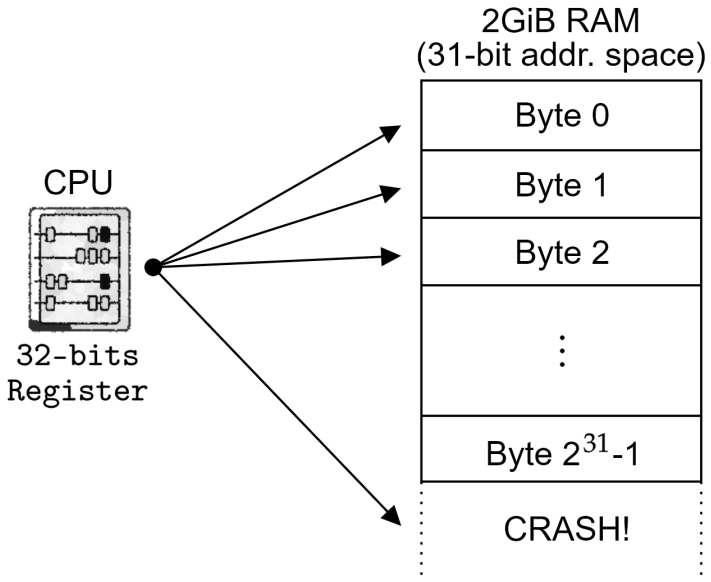

Figure 1.1: A 32-bit CPU accessing 2 GiBs of RAM, where a crash happens when trying to access beyond the 31-bit address space.

The next problem deals with multiple processes allocating and deallocating memory:

Definition 1.2: Memory Fragmentation

Memory can be thought of as a big array, where each cell is a resource a program can use. We want memory usage to be contiguous (i.e., no gaps or holes). So say we have an array

$$[O, O, O, O]$$

Where O represents free space in our array, each cell 1 GiB of space. If we have processes A and B take 1 and 2 GiBs respectively, we might have a memory layout of:

$$[A, B, B, O,]$$

If we then free process A, we might have a memory layout of:

$$[O, B, B, O]$$

Now, if another process C needs 2 GiBs of memory, it will not be able to find a contiguous space of 2 GiBs, even though we have 2 GiBs of free space. This is called **memory fragmentation**.

Now finally we have the problem of protecting memory from other processes:

Definition 1.3: Memory Security

In a multi-process system, processes may have collisions when trying to access the same memory space. For example, if process A is a weather service and process B is some finance service, we don't want the weather service to overwrite the same memory space where the finance service is storing critical data. This is called **memory security**.

So in theory we want to give each process it s own portion of memory, to solve overlapping access:

Definition 1.4: Virtual Memory

To solve process memory collisions, we give each process its own fictional view of memory, called **virtual memory**. Though for this to work, each virtual view is mapped to an actual place in the original memory we call **physical memory**.

Virtual and physical addresses are the cells spaces themselves.

Consider the figure below showing how virtual memory works in theory:

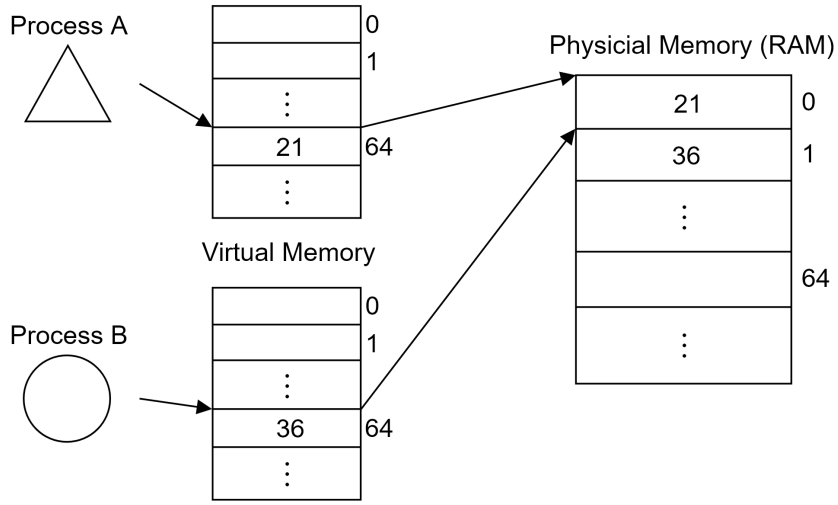

Figure 1.2: Processes A and B write to memory cell 64 in their view of memory, but in reality they map to different physical memory cells (0 and 1 respectively).

A quick aside:

Theorem 1.1: Physical Memory the CPU Accesses

In reality the CPU can access the physical memory of many other devices (e.g., hard drives, SSDs, etc.). In addition, the OS takes up some of the physical memory as well. The rest is left to programs to use. The program allocated space is the memory we refer to going forward as the **physical memory**.

Virtual memory solves the three problems we mentioned before:

Definition 1.5: Virtual Memory & Not Enough Memory (Swapping)

The physical memory can be much smaller than what a program thinks it has in virtual memory. When a program tries to access memory it does not have, the OS will **swap** physical memory to external storage to free up space. This means, while a program is not using a portion of memory at a given time, the OS can swap it in and out depending on system needs.

Memory that is swapped out is called **swap-memory**. Every time a we try to access such absent memory in our mappings, it's called a **page fault** (more on this later).

Definition 1.6: Virtual Memory & Fragmentation

Virtual memory allows programs to think they have contiguous memory. This simplifies their memory management, while in reality the OS manages the split memory mappings.

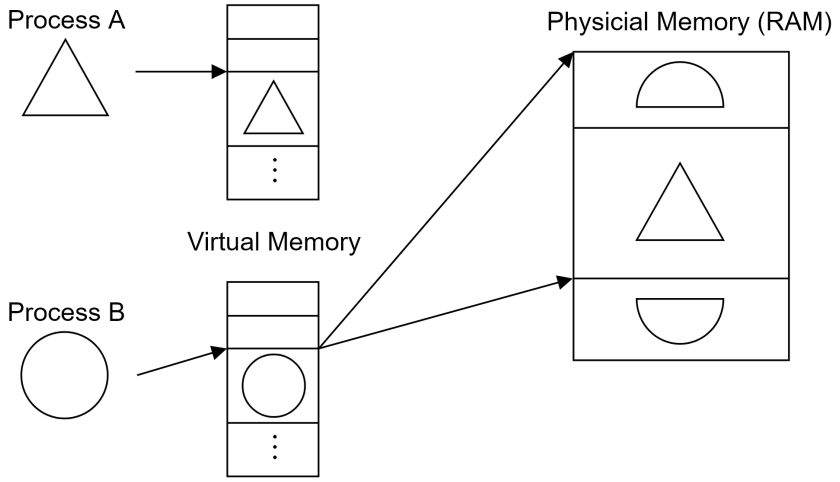

Figure 1.3: Here two processes A and B are using the same physical memory space. Both think they have contiguous memory, but in reality, B is split into two parts in the physical memory.

Definition 1.7: Virtual Memory & Security

Memory cells are collision safe as memory mappings are not shared in physical memory; However, this would be inefficient if say A and B depend on a secondary process C. In this case, A and B may share the same mapping to C's memory.

1.1.2 Virtual Memory Implementation (Page Tables)

We discuss the mapping mechanism further in linking virtual to physical memory.

Definition 1.8: Page Tables

The OS keeps a **page table**, where each entry is a mapping of a virtual memory cell to a physical one, called a **Page Table Entry (PTE)**. CPUs work with words (32 bits = 4 bytes), so the page table has one entry for every word (address) in the virtual memory space.

We make the following observation:

Theorem 1.2: Theoretical Page Table Space Complexity

If our CPU 32-bits, then there are 2^{32} addresses. CPUs speak in words, hence we'd need 2^{30} words, and thus ≈ 1 billion PTEs (4 GiB per table). This is too much memory to practically implement for each process.

We solve the above problem, via the following strategy:

Definition 1.9: Memory Chunking (Pages)

Instead of mapping each address to a PTE—which would be too large. We chunk the memory into **pages** reducing the number of PTEs needed. Typically, page sizes are some power of 2, most commonly 4 KiB (4096 bytes), meaning 1024 words per page. This reduces our page table requirement from 4 GiB to 4 MiB (1 Million PTEs). Despite moving 4 KiBs of data at a time, this works well in practice, as adjacent memory is often accessed together.

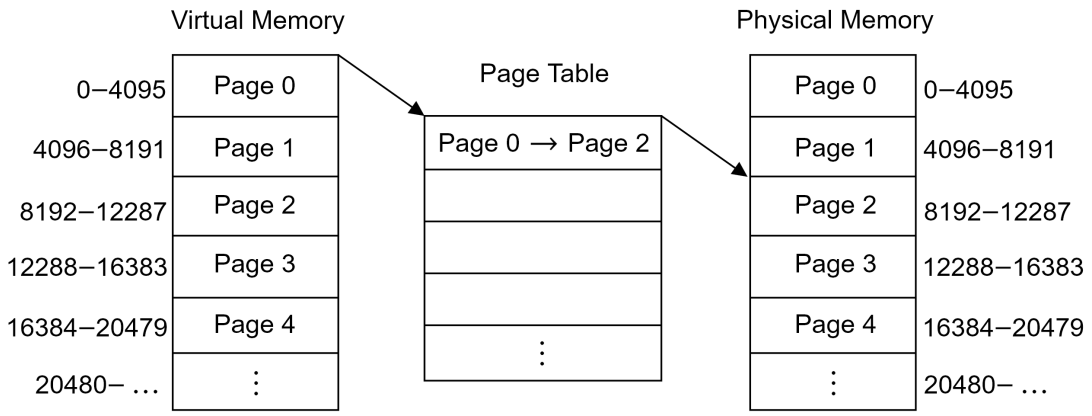

Figure 1.4: A 4 MiB page table, each PTE 4 KiB (4096 bytes, 1024 words), showing mappings.

One must ask themselves:

- What is the relationship of how the Page Numbers are separated.
- Can we determine the page number given a specific address?
- How might we convert a virtual to a physical address given some address?

We discuss the following on the next page.

Example 1.1: Converting Virtual to Physical Memory (intuition)

We may find conversions from virtual to physical memory via the following formula:

$$\text{PA} = \underbrace{(\text{VA } \% \text{ Page Size})}_{\text{offset}} + \underbrace{(\text{Page Size} \cdot \text{Physical Page Number})}_{\text{Physical Page starting index}}$$

Where % is modulo and Page Number $= \left\lfloor \frac{\text{Addr.}}{\text{Page Size}} \right\rfloor$. **However,** this is not the most efficient way, and is strictly for educational/intuition building purposes.

Given the Figure (1.4), $104 \to 8296$ from $104 + 4096 * 2 = 8296$. ■

Theorem 1.3: Converting Virtual to Physical Memory

Addresses are binary numbers, typically represents as hexadecimal (base 16) numbers.

The first 12 bits of an address is the **offset**. The remaining higher-order bits yields the **page number**. To find the physical address, take the last 12 bits of the address, index the higher-order bits into the page table, and append the offset to the physical page number.

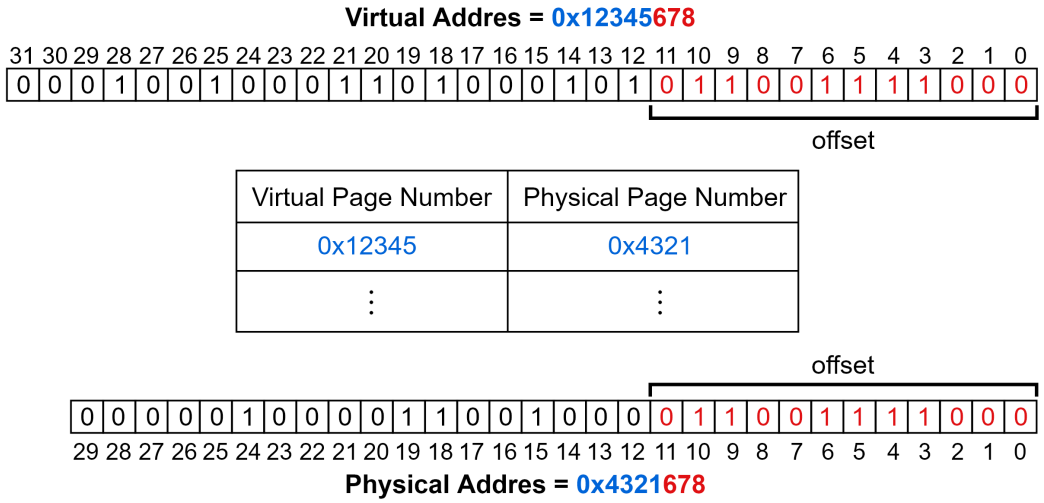

Figure 1.5: The conversion of 0x123456 (32-bit, 4 GiB) → 0x4321678 (30-bit, 1 GiB). Recall that if a accessing a page that is not in memory, causes a page fault, from which the OS with begin to swap memory with external storage devices.

1.1.3 Page Faults & Translation Lookaside Buffer (TLB)

So far we have discussed how the mapping system of virtual memory works, but now we pivot to how the OS handles swapping data in and out of memory on page faults.

Definition 1.10: Swapping on Page Faults

A **page fault** occurs when a program tries to access a page that does not have a mapping in the page table. The OS then picks the **least recently used (LRU)** page in the page table, and swaps it out to external storage. A page is **dirty** if it has been written to. In such case, we write back its contents to external storage before swapping. If the page is not dirty, we may discard it on swap.

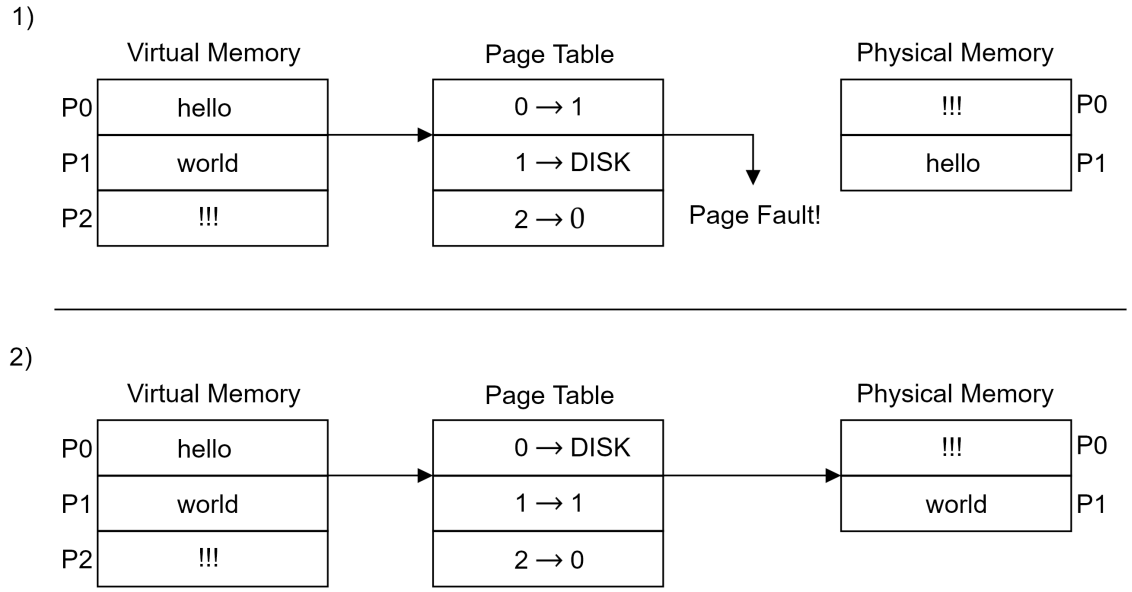

Figure 1.6: 1) A page fault occurs when trying to access page 1. 2) The OS has swapped out page 0 to external storage (DISK), and now page 1 can be accessed. Note: This diagram is for illustrative purposes, virtual memory does not contain data, only addresses binding to physical memory.

Definition 1.11: Direct Memory Access (DMA)

Page faults are expensive, as swapping data with I/O devices may take some time. Modern CPUs have a **Direct Memory Access (DMA)** module, which allows I/O devices to access memory directly, while the CPU completes other tasks.

Still our routine of mapping is expensive in it of itself.

Definition 1.12: Translation Lookaside Buffer (TLB)

The mapping process includes three steps:

1. Index the page table: Access Memory (RAM).
2. Convert Addresses: Computation.
3. Interact: Access Memory (RAM).

To speed this up, we create a small cache of the most recent Page Table Lookups, called the **Translation Lookaside Buffer (TLB)**. If a page in not in the TLB, we must preform the translation then load it into the TLB for future use.

Every successful TLB lookup is called a **hit**, while a failed lookup is called a **miss**. The **hit time** and **miss time** are the time it takes to access the TLB and page table respectively (measured in CPU clock cycles). The **hit rate** is the ratio of hits to the total number of lookups.

Modern CPU architectures usually have two TLBs, one for instructions (**ITLB**) and one for data (**DTLB**).

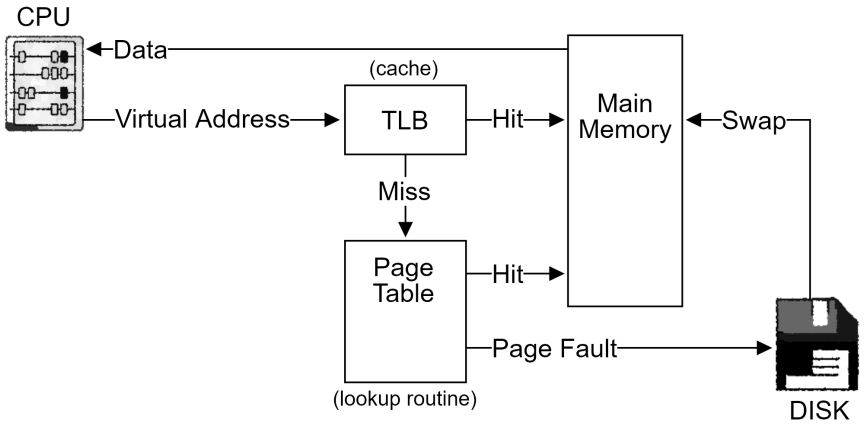

Figure 1.7: Demonstrating how a TLB lookup effects the memory access flow. CPU attempts to access memory providing the virtual address to the TLB. (Happy path) A hit occurs, memory is accessed and passed to the CPU. (Ok path) A TLB miss occurs, causing a page table lookup. A page table hit occurs, the mapping is cached and the memory is accessed. (Sad path) both a TLB and page table miss occurs, causing a page fault. The OS must decide which page to swap out. After swapping, the addressed is cached and the memory is accessed.

So far what we've been covering actually resides within the CPU architecture:

Definition 1.13: Memory Management Unit (MMU)

The **Memory Management Unit (MMU)** is a hardware component that manages the mapping of virtual to physical memory. It is responsible for translating virtual addresses to physical addresses, and it also handles page faults and TLB lookups.

1.1.4 Multi-level Page Tables

Let's define the problem space of multi-level page tables [1]:

Theorem 1.4: Single Level Page Table Cost

In a 32-bit system, each page table requires 4 MiB of memory. If we had 100 programs running, that's easily 400 MiB of memory.

Additionally, if we move page tables to disk (external storage), we lose any reference we had to them in an attempt to free up memory. So we need to keep some reference oracle in memory.

This highlights the need for a more efficient way

Definition 1.14: Multi-level Page Tables

Recall that in a 32-bit system, we have 4 GiB of memory. In single level page tables, we chunk our addresses into 4 KiBs (4096 bytes), and load all of them at once. That's

$$4 \text{ KiB} \cdot 1024 \text{ PTEs } = 4096 \text{ KiB} = 4 \text{ MiB}$$

We wish to offload data without losing references. To do this we allocate a single chunk of memory (4 KiB) to serve as an reference oracle, called the **page directory** or **1^{st} level page table**. Each entry in the page directory points to other chunks from which we are safe to load and unload out of memory. These referenced chunks are called the **2^{nd} level page tables**. Each entry in the 2^{nd} level page table points to a physical page.

Definition 1.15: Converting Virtual to Physical Memory (Multi-level)

In multi-level page tables, the first 12 bits are the **offset**, the next 10 bits are the index into the **second-level page table**, and the last 10 bits are the index into the **first-level page directory**. These tables are treated as arrays: indexing the first table yields the address of the second table, and indexing the second table yields the physical page number.

Recall, in single-level page tables the higher 20 bits are the page number and the index, which yields the physical page number. Multi-level breaks away from this, and instead as illustrated below:

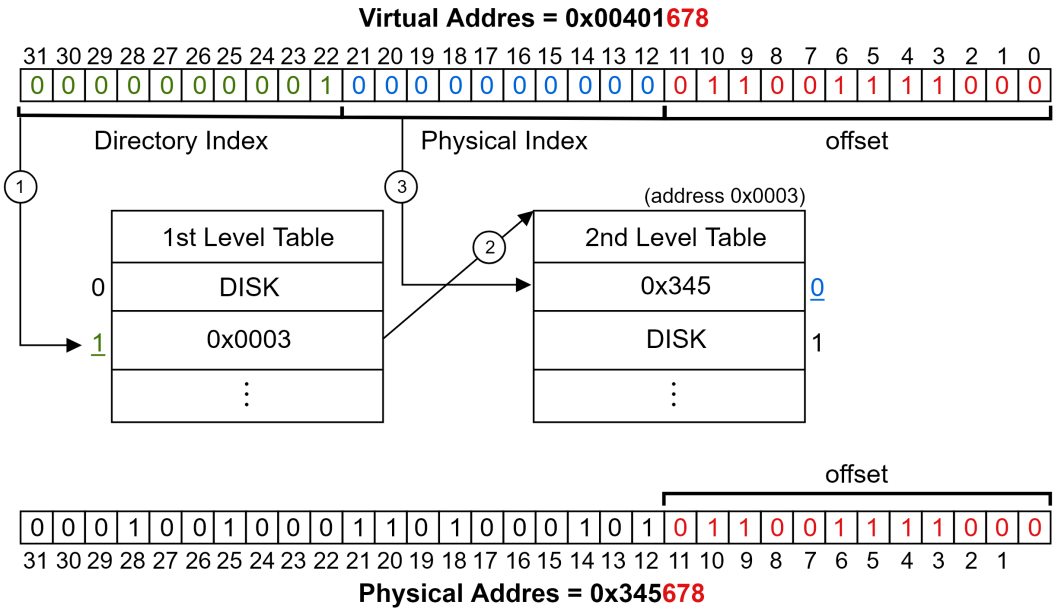

Figure 1.8: A multi-level page table, where the first level is a page directory, and the second level is a page table which points to physical pages.

Definition 1.16: Multi-level Page Scaling (64-bit)

Increasing the number of bits allows us to scale the number of levels in our page table. In 64-bit architectures using 4-level paging (e.g., x86-64), only the lower 48 bits are used. The virtual address is divided into five parts:

- 12 bits for the page offset
- 9 bits for the 4^{th}-level page table (PT)
- 9 bits for the 3^{rd}-level page directory (PD)
- 9 bits for the 2^{nd}-level page directory pointer table (PDPT)
- 9 bits for the 1^{st}-level page map level 4 (PML4)

Modern versions of Windows and Linux support 5 levels, which allows for a maximum of 129 PiB, while 4 gives us 256 TiB.

Finally we talk about the performance of our lookups:

Definition 1.17: Effective Memory Access Time (EMAT)

Effective Memory Access Time (EMAT) accounts for the time it takes to access memory in the presence of a Translation Lookaside Buffer (TLB), multi-level paging, and potential page faults. Given:

- t = TLB access time; m = Memory access time
- S = Page fault service time; n = Number of page levels
- h = TLB hit rate; p = Page hit rate (1 - page fault rate)

The EMAT is computed as:

$$\text{EMAT} = h(t+m) + (1-h)\,(t + p(n \cdot m) + (1-p)S)$$

Essentially [4],

```
EMAT=
TLB hit*(TLB access time + Memory access time)
+ TLB Miss*(TLB access time + PageHit*[n * memory access time]
+ PageMiss*PageFaultServiceTime)
```

Example 1.2: Three-Level Paging System

Suppose the system has:

- $n = 3$ page levels
- $t = 5$ ns (TLB lookup)
- $m = 100$ ns (memory access time)
- $\alpha = 0.80$ (TLB hit rate)

Then the EMAT is:

$$\begin{aligned}\text{EMAT} &= (5+100)\cdot 0.80 + (5+3\cdot 100+100)\cdot(1-0.80)\\ &= 105\cdot 0.80 + 405\cdot 0.20 = 84+81 = \boxed{165 \text{ ns}}\end{aligned}$$

For a better TLB hit rate $\alpha = 0.98$:

$$\text{EMAT} = 105\cdot 0.98 + 405\cdot 0.02 = 102.9 + 8.1 = \boxed{111 \text{ ns}} \qquad [2]$$

■

Bibliography

[1] Neso Academy. Virtual memory: Multilevel page tables, May 2020. Accessed: 2025-04-19.

[2] Yair Amir. Performance of multilevel paging. https://www.cs.jhu.edu/~yairamir/cs418/os5/sld035.htm, 2003. Slide 35, Operating Systems Course (CS 418), Johns Hopkins University.

[3] Computerphile. But, what is virtual memory?, 2023. Accessed: 2025-04-18.

[4] Ritwik (https://cs.stackexchange.com/users/58645/ritwik). calculate the effective (average) access time (e at) of this system. Computer Science Stack Exchange. URL:https://cs.stackexchange.com/q/69017 (version: 2019-04-27).

[5] Ioannis Liagouris. Cs351: Distributed systems. Lecture notes, Boston University, Spring Semester, 2025. Boston University, CS Department.