Computer Organization & Architecture

Cybersecurity Department

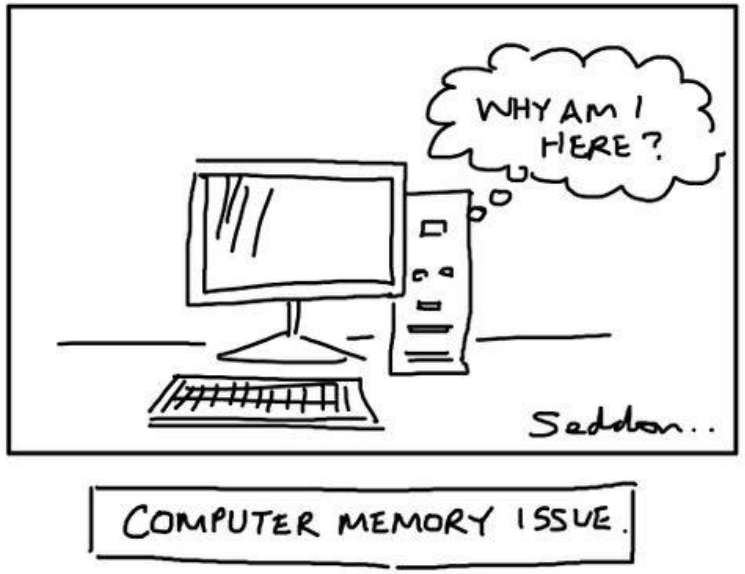
Course Code: ***CBS219***

Lecture 6: **Memory Organization,** ***Cache and Main Memory Architecture***

Halal Abdulrahman Ahmed

Lecture Outline

- Cache Memory Basics
- How Cache Works Internally (Hit & Miss)
- Cache Lines, Mapping & Replacement Policies
- Write Policies (Write-through & Write-back)
- Introduction to Virtual Memory
- Pages, Frames, and Page Tables
- Page Faults and Demand Paging
- TLB and Page Replacement
- Thrashing & Real OS Examples



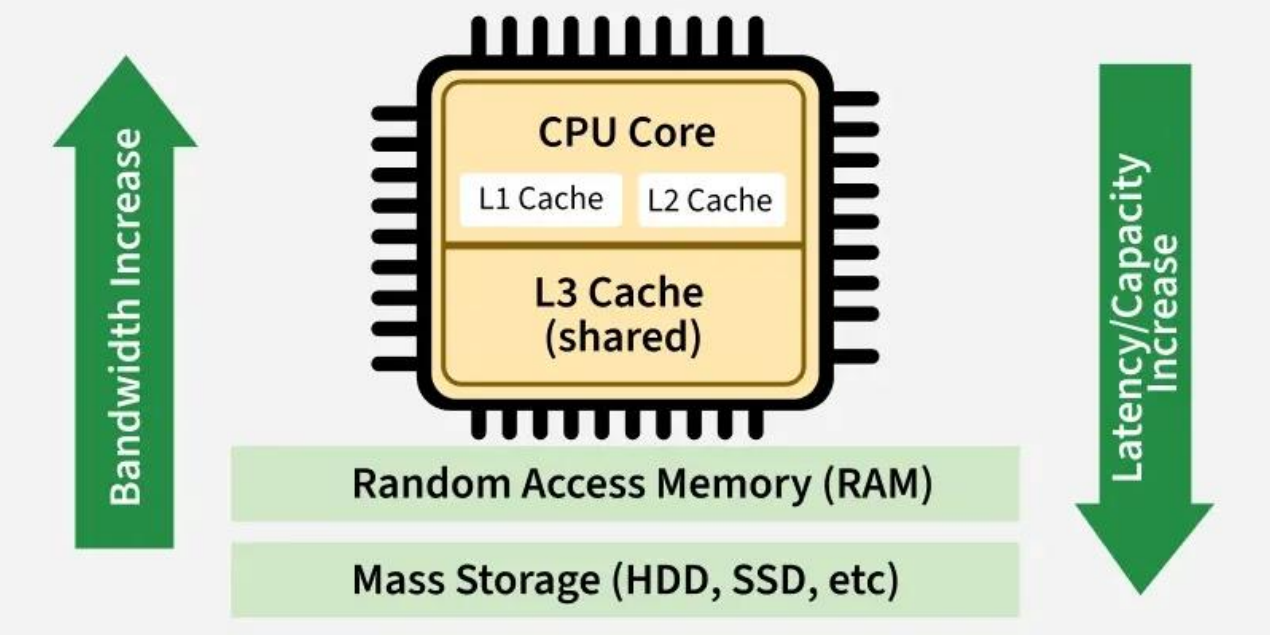
Lecture Outcomes

By the end of this lecture, students will be able to:

- Describe how cache improves CPU performance.
- Explain cache hits, misses, and mapping techniques.
- Understand virtual memory and why modern systems use it.
- Identify pages, frames, and the role of the page table.
- Explain page faults, TLB, and page replacement behavior.

Cache Memory

Cache memory is a small, fast storage space within a computer. It holds duplicates of data from commonly accessed locations in the main memory. The CPU contains several separate caches that store both instructions and data.

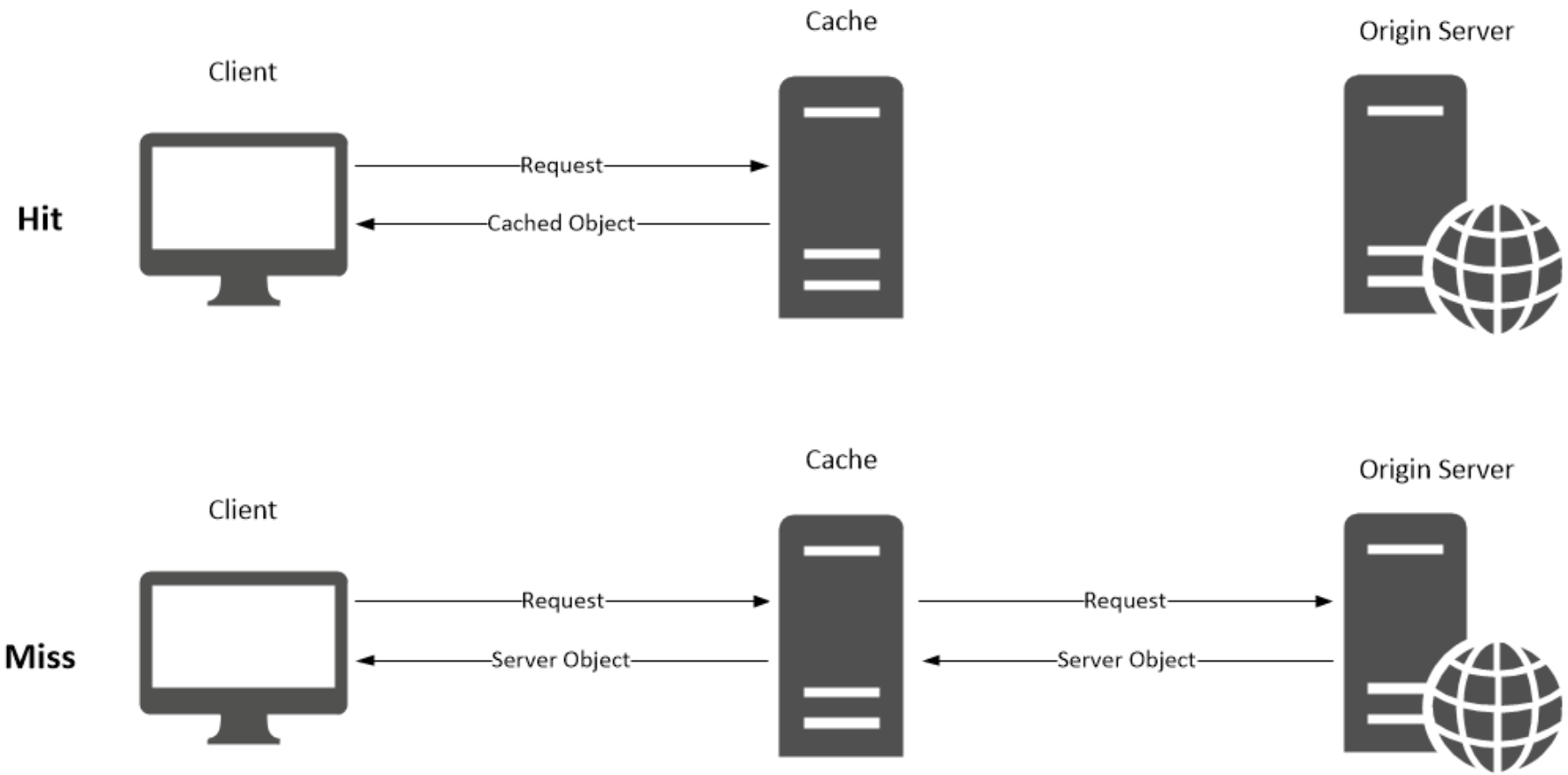


How Cache Works Internally?

When the CPU needs to access data:

- **Cache Lookup:** The CPU sends the memory address to the cache controller.
- **Tag Comparison:** Cache checks whether this address is stored in one of its cache lines.
- **Cache Hit:** Data is found → delivered immediately to CPU in a few CPU cycles.
- **Cache Miss:** Data not found → CPU fetches from RAM and also stores a copy inside cache for next time.
- **Replacement (if needed):** If cache is full, one of its lines must be replaced.

Cache's ability to store recently accessed data allows it to anticipate future requests and reduce delays.


Cache
Origin Server
Client
Request
Hit
Cached Object
Cache
Origin Server
Client
Request
Request
Miss
Server Object
Server Object

Cache Hit and Cache Miss

- **Cache Hit:** When the CPU finds the required data in the cache memory, allowing for quick access. On searching in the cache if data is found, a cache hit has occurred.
- **Cache Hit Example:**

You open Chrome → The tabs you recently used reload quickly because the data was cached.

- **Cache Miss:** When the required data is not found in the cache, forcing the CPU to retrieve it from the slower main memory. On searching in the cache if data is not found, a cache miss has occurred
- **Cache Miss Example:**

You open a program after many days → The system loads it from disk into RAM → takes longer.

Cache Stores Two Things

Cache stores:

- **Instructions** (what the program wants to do)
- **Data** (numbers, variables, arrays etc.)

Some CPUs have separate caches:

- **Instruction Cache (I-Cache)**
- **Data Cache (D-Cache)**

Cache Blocks (Cache Lines)

- Cache does not copy memory one byte at a time. It loads **chunks** called **cache blocks** or **cache lines**.

Example:

If block size is 64 bytes and CPU asks for one byte, cache loads **all 64 bytes** from RAM. This improves performance due to **spatial locality**.

Mapping Methods

- Cache must decide **where to store memory blocks**. There are 3 mapping methods:

1. Direct Mapping

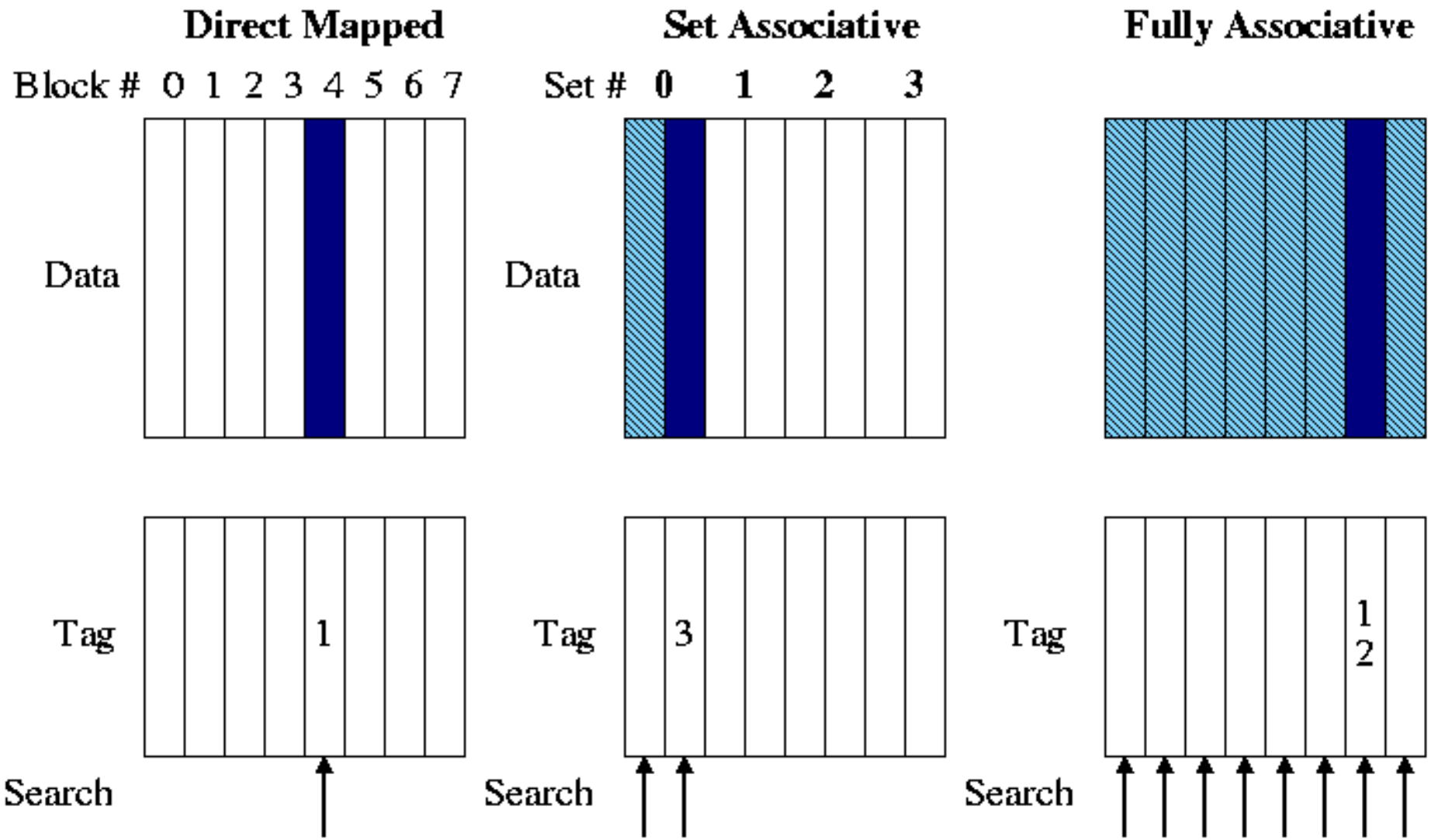
- Each memory block maps to **exactly one** cache line. Fast but can cause conflicts.

2. Fully Associative Mapping

- Any block can go **anywhere in cache**. Flexible but expensive to implement.

3. Set-Associative Mapping

- Middle approach: Cache is divided into sets. Each block maps to one set, but can go into **any line in that set**. Most CPUs use this method.


Direct Mapped
Block # 0 1 2 3 4 5 6 7
Data
Tag
1
Search
Set Associative
Set # 0 1 2 3
Data
Tag
3
Search
Fully Associative
Tag
1
2
Search

Replacement Policies

- When cache is full and new data must be loaded, cache must choose what to remove. Common policies:
- **LRU (Least Recently Used)**

Remove the data that was used longest ago.

- **FIFO (First In First Out)**

Remove oldest data.

- **Random**

Remove a random block.

Write Policies

When CPU writes data, cache must update memory. It uses two main policies:

Write-Through

- Write to cache AND RAM at same time
- Safer, slower

Write-Back

- Write only to cache
- RAM updated later
- Faster, used widely

Virtual Memory

"You forgot our anniversary?! I knew I shouldn't have married someone that runs out of memory so quick."

Introduction to Virtual Memory

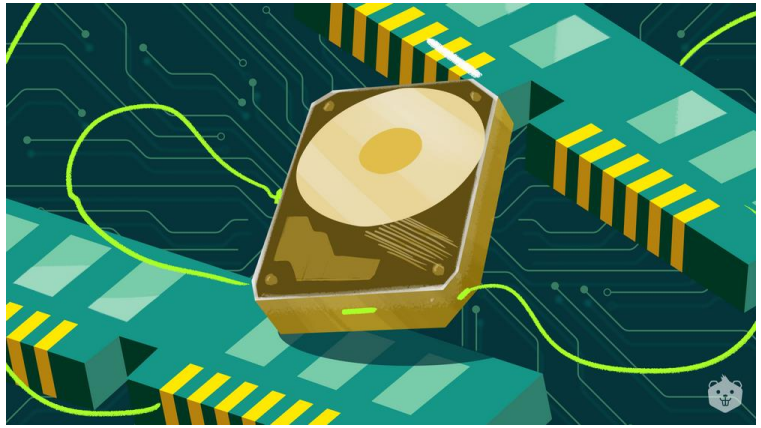
Before we start understanding virtual memory, imagine this situation:

You have a small desk to study on. The desk is your **RAM**.

Your backpack is the **hard disk or SSD**

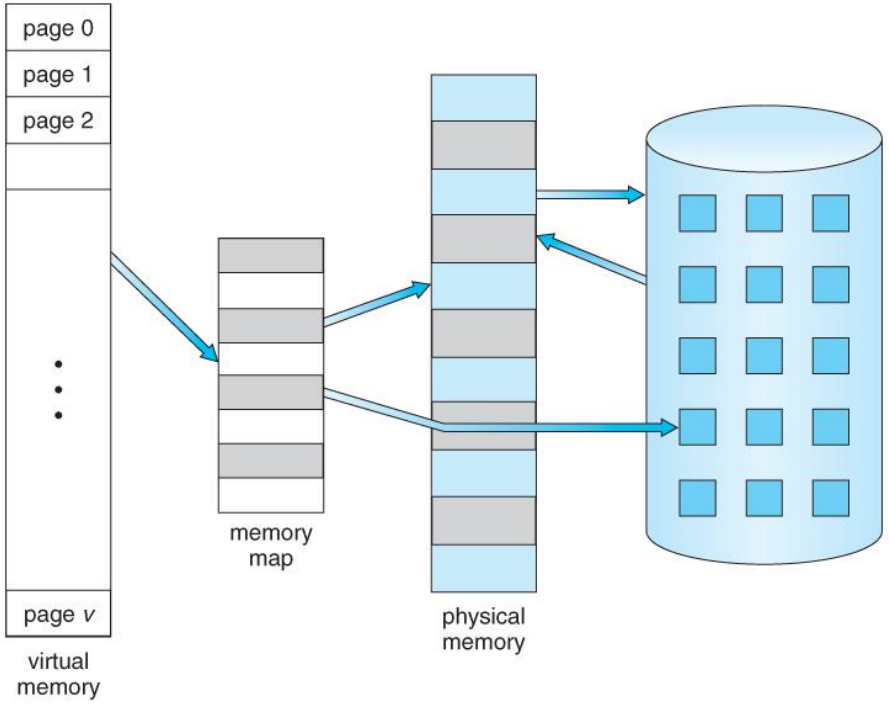
Your books are your **programs**

- Your desk is small, so you cannot place all your books at the same time. Instead, you put *only the book you are reading now* on the desk. That book represents the **page currently loaded in RAM**. The rest of the books remain in your backpack until you need them. Virtual memory works exactly like this.
- Your RAM is small, but your programs are large. So the operating system keeps only the parts of the program needed right now in RAM. Everything else waits on the disk until you need it.
- This clever trick allows modern computers to run large programs even with limited physical memory.



What is Virtual Memory?

- Virtual Memory is a memory management technique that allows programs to use more memory than physically available by storing inactive parts of memory on disk.



What “Virtual” Really Means

- “Virtual” means something that appears to be there, but not fully real. Virtual memory gives the program the **illusion** that it has access to a huge, continuous block of memory even if your computer has very little RAM.
- The program does **not** access RAM directly. Instead, it uses **virtual addresses**, and the system translates them into **physical addresses**.
- This translation happens automatically, every nanosecond, for every memory access, using a hardware component called the **Memory Management Unit (MMU)**.

Why Virtual Memory is Necessary?

- Computers today run many programs at once browsers, email, Zoom, games, editors, etc. If the computer required every program to be fully loaded into RAM, we would need machines with massive memory 128 GB or more just to open common applications. That is not practical. Most laptops have between 4 - 16 GB of RAM.
- Virtual memory solves this by **pretending** the system has more memory than it physically has. It allows each program to run independently, without worrying about how much RAM is available. It also isolates programs from each other, so one program cannot accidentally read another program's data.
- For example, if Microsoft Word crashes, it does not destroy the memory used by Chrome or Zoom. That protection comes from virtual memory.

How Virtual Memory Works?

- When a program is running, it does not use physical RAM directly. It uses something called **virtual addresses**. These virtual addresses must be converted into physical addresses that point into actual RAM. This conversion is done by the MMU.
- Not every part of the program is loaded into RAM. Only the parts needed right now are kept inside RAM. The rest of the program stays on disk in a special area called **swap space**.
- Whenever the CPU needs something that is not in RAM, the system performs a **page fault**, which means: “The page you want is not in RAM; I must get it from disk.”

The operating system will then:

- Pause the running program,
- Go to the disk,
- Bring the missing part into RAM,
- Update the mapping,
- Resume the program as if nothing happened.

To the program, everything looks smooth.

To the hardware, a lot of work is happening behind the scenes.

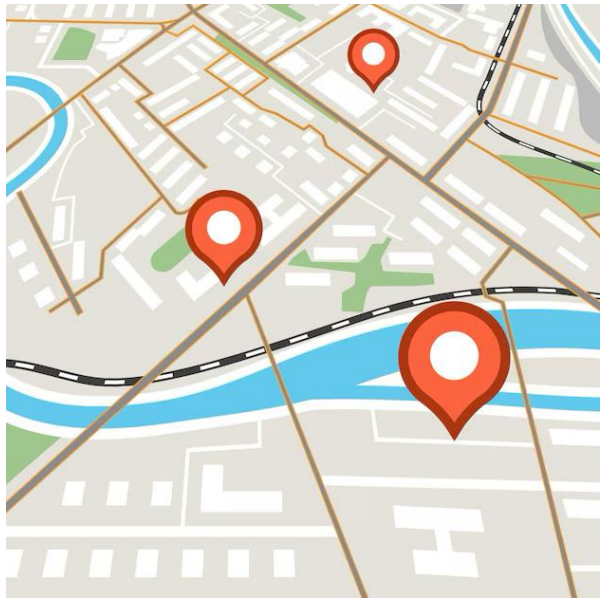
Pages and Frames

- To manage memory easily, both virtual memory and physical memory are divided into small equal-sized blocks. Virtual memory is divided into **pages**. Physical memory (RAM) is divided into **frames**.
- Think of virtual pages as chapters of a big book. Think of physical frames as shelves in a limited-size bookcase. Your book has hundreds of chapters (pages), but your bookcase can only hold a few of them. You take out the chapters you need right now, and when you finish them, you replace them with others. That's exactly how virtual memory works.

- If your RAM has space for 300 frames and your program has 2000 pages, only the pages needed now will be loaded. When you need page number 500, the OS will find a free frame and load it. If there is no free frame, it will choose an existing one to remove and put the new page in its place.

What Is a Page Table?

Because pages move in and out of RAM, the CPU needs a map to know where each virtual page is located. That map is called the **page table**. Without a page table, the CPU would have no idea where to look for the data.

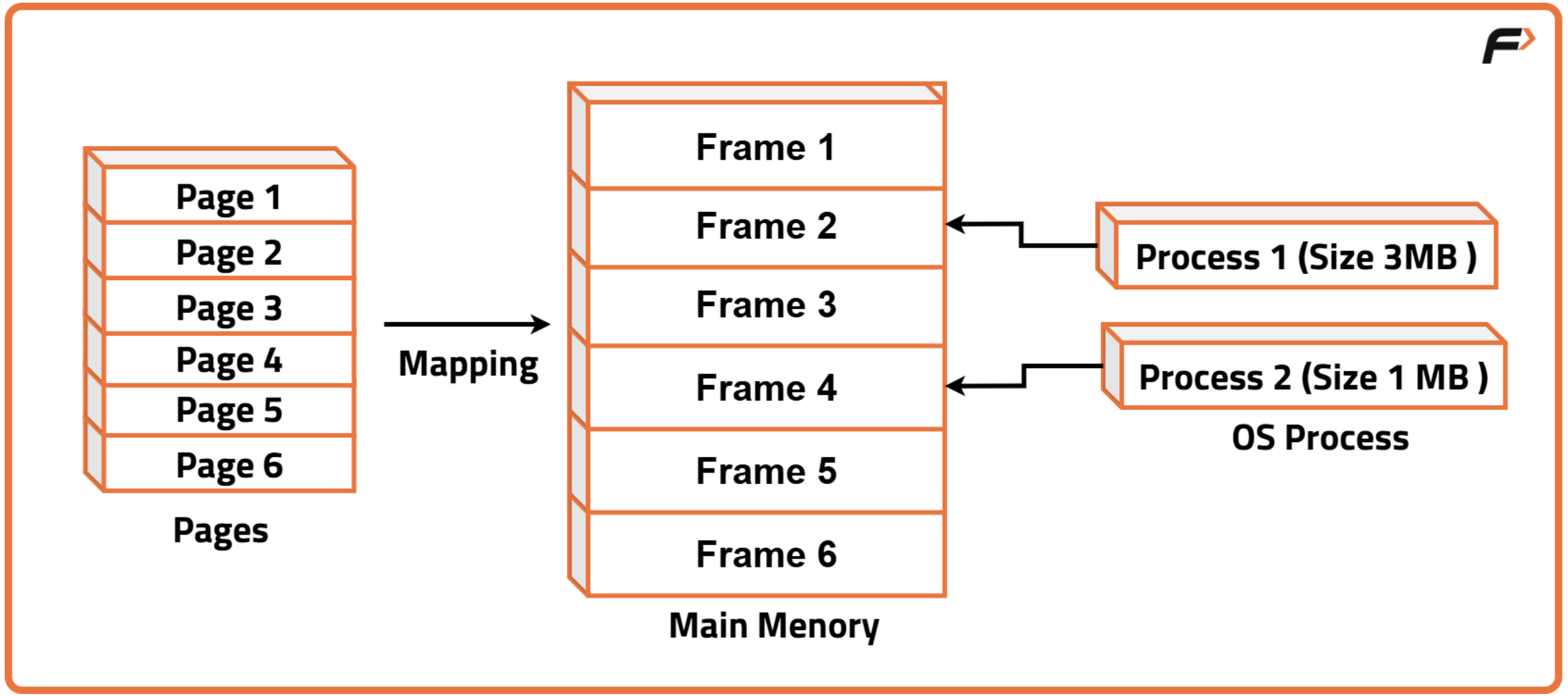


The page table does not store the actual data. It only tells the CPU:

- Whether the page is currently in RAM
- Which frame contains it
- Whether it has been modified
- Whether the process is allowed to read or write it

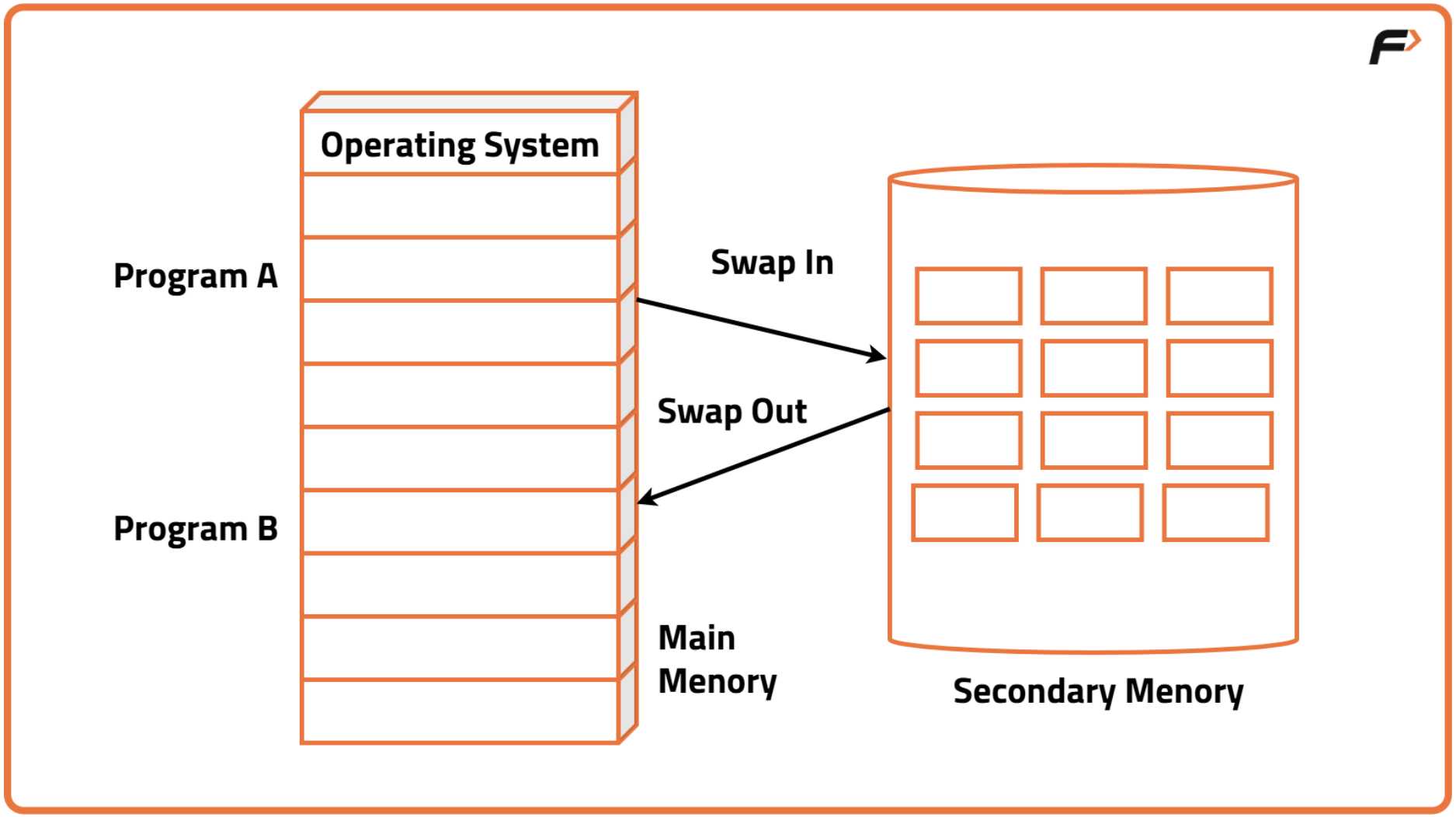
Page Fault

A page fault is an exception that occurs when a program accesses a memory page that is not currently present in physical memory (RAM). In modern operating systems that use virtual memory management, memory is divided into fixed-size pages, and only the pages that are actively being used by a process are kept in physical memory. When a program attempts to access a page that is not in RAM, a page fault occurs, triggering the operating system to load the required page from secondary storage (such as a hard disk or SSD) into physical memory.


Page 1
Page 2
Page 3
Page 4
Page 5
Page 6
Pages
Mapping
Frame 1
Frame 2
Frame 3
Frame 4
Frame 5
Frame 6
Main Menory
Process 1 (Size 3MB)
Process 2 (Size 1 MB)
OS Process

Page Faults (cont.)

Page faults are a fundamental part of virtual memory systems, allowing programs to address more memory than is physically available. They enable efficient use of memory resources by swapping pages in and out of physical memory as needed, based on the demands of running processes. While page faults incur some overhead due to the need to access slower secondary storage, they are essential for providing the illusion of a larger memory space to processes than actually exists in physical memory.


Operating System
Program A
Program B
Swap In
Swap Out
Main
Menory
Secondary Menory

Page Faults

- A page fault is not an error. It is simply a signal that the page you need isn’t in RAM. The delay you feel when switching between heavy applications is often caused by page faults.

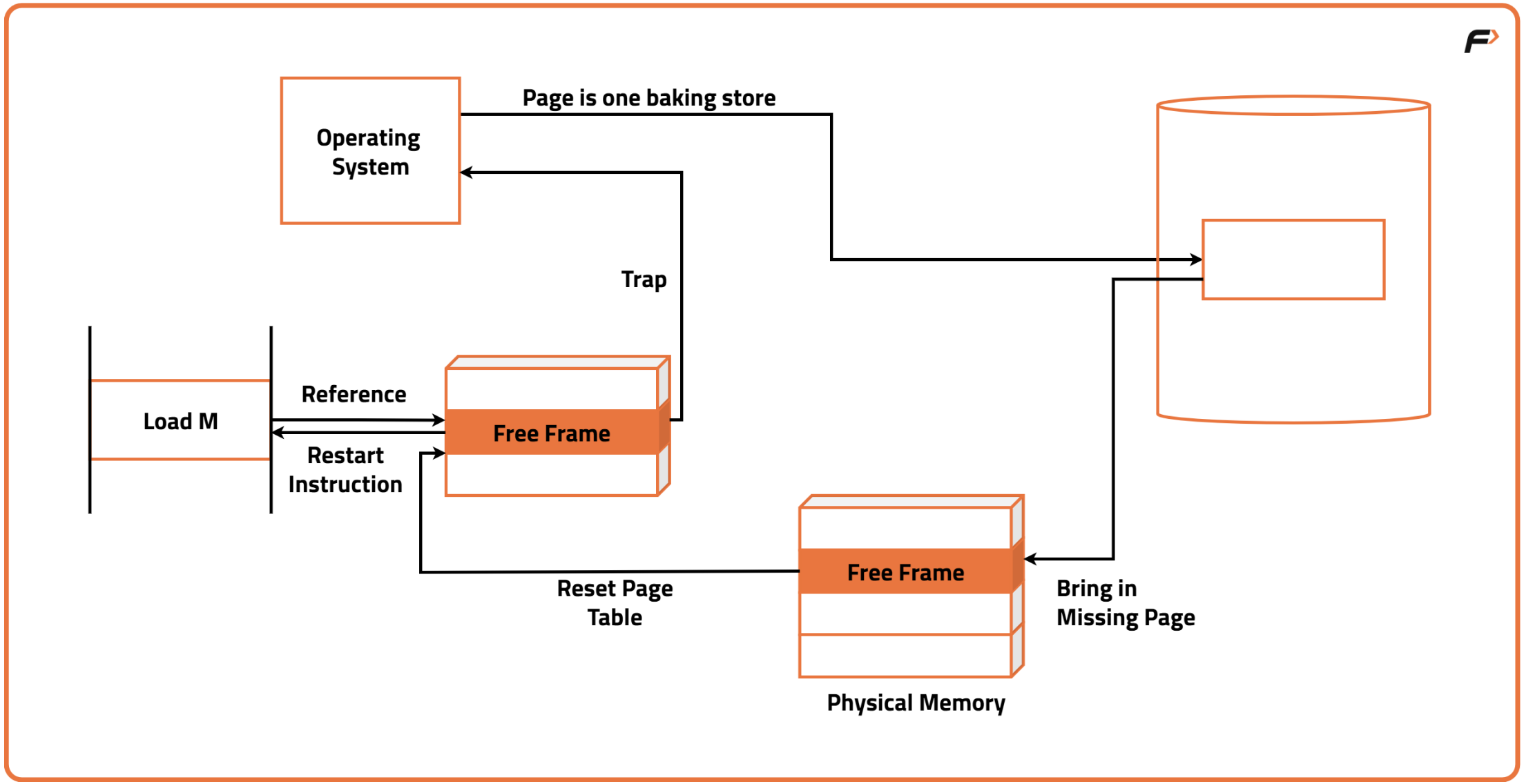
When the computer receives a page fault:

- It pauses the program.
- It finds the missing page on disk.
- It loads it into a frame in RAM.
- It updates the page table.
- It resumes the program.

Handling Page Faults

When a page fault occurs, the operating system intervenes to handle the exception and ensure that the required page is brought into physical memory. The steps involved in handling a page fault are:

- **Page Table Lookup:** The operating system consults the page table of the process to determine whether the accessed memory address corresponds to a valid page in physical memory or if it has been swapped out to disk.
- **Disk Access:** If the required page is not in physical memory, the operating system initiates a disk access to read the required page from secondary storage into a free page frame in physical memory.
- **Updating Page Table:** Once the required page is loaded into physical memory, the operating system updates the page table entry for the accessed memory address to indicate that the page is now resident in RAM.
- **Resuming Execution:** Finally, the operating system restarts the instruction that caused the page fault, allowing the process to continue its execution with the required page now available in physical memory.


Operating System
Page is one baking store
Trap
Load M
Reference
Restart Instruction
Free Frame
Free Frame
Reset Page Table
Bring in Missing Page
Physical Memory

Demand Paging

Most modern operating systems use demand paging, where only the pages of memory that are actively being used by a process are loaded into physical memory. As a result, when a process accesses a page that is not currently in RAM, a page fault occurs, and the required page is fetched from disk into memory.

If you open Microsoft Word, the system does not load the entire program into RAM. It loads only the **necessary pages**, such as:

- The toolbar
- The ribbon
- The document you are editing

As you scroll through the document, the OS loads the new pages you need and removes the old ones. This method is called **demand paging** because pages are loaded “on demand.”

The TLB: A Helper That Makes Everything Faster

- Because virtual-to-physical translation is slow, the CPU keeps a very small but very fast memory called the **Translation Lookaside Buffer (TLB)**.
- The TLB stores recent translations: “Virtual page 10 is in physical frame 48.”

When the CPU wants to access memory:

- It checks the TLB first.
- If the translation is there, everything is extremely fast.
- If not, the CPU must check the full page table (slower).
- A TLB miss is like forgetting where you put your keys, so you check the usual place first before searching the whole house.

Page Replacement: When RAM Is Full

When RAM becomes full and a new page is needed, the system must decide which page to remove. This decision process is guided by **page replacement algorithms**, like:

- Remove the page that has been unused the longest (LRU).
- Remove the oldest page (FIFO).
- Remove the least frequently used page (LFU).

Thrashing: The Worst Case Scenario

Thrashing happens when the system is swapping pages in and out of memory so frequently that there is no time left to actually execute the program.

- The system becomes very slow, the disk becomes extremely active, and nothing seems to work.
- Thrashing usually happens because:
- Too many programs are open
- RAM is too small
- Programs are switching between many pages rapidly

The solution is simple: close some programs or add more RAM.

Real Examples from Operating Systems

- Virtual memory is not just a theory, your real computer uses it every day. Every operating system (Windows, Linux, macOS) uses a part of the **hard disk or SSD** as an extension of the RAM. This disk area is where pages are temporarily stored when RAM is full.
- In Windows computers, when RAM becomes full, the system takes the pages that are not being used and stores them in a **special file** on the disk called: **pagefile.sys ,** this file acts like "extra memory."

Example:

- If your RAM is 8 GB but you need 12 GB to run your applications:
- 8 GB will go into RAM
- The extra **4 GB** will be stored temporarily in **pagefile.sys**
- Windows uses it automatically. You don't need to do anything.

Real Examples from Operating Systems (cont.)

- Linux uses something similar, but it has two methods: **Swap partition,** this is a dedicated part of the disk created when installing Linux. It is a separate area only for virtual memory.

Swap file

- A normal file (like Windows pagefile) used as virtual memory. Linux will use the swap area when RAM is full.

Example:

- If your Linux machine has 4 GB RAM and you open many apps, Linux might move inactive pages into the swap area to make space.

Real Examples from Operating Systems (cont.)

- macOS (MacBook) handles virtual memory automatically. It creates **multiple small swap files** when needed. You don’t see them unless you open a terminal they are hidden inside the system folder.

Example:

- If your Mac has 8 GB RAM and you open Chrome with many tabs, macOS creates swap files like:
- swapfile0
- swapfile1
- swapfile2
- Each of these files stores pages removed from RAM.
- You don’t manage it; macOS handles everything.

References

- Fox, C. (2022). Computer architecture: From the Stone Age to the quantum age. No Starch Press.
- Stallings, W. (2022). Computer Organization and Architecture: Designing for Performance (11th ed.). Pearson.
- Take U Forward. (n.d.). *Page fault*. Take U Forward. Retrieved November 27, 2025, from https://takeuforward.org/operating-system/page-fault


Any
Question