Memory Virtualization

Chris Zambrana

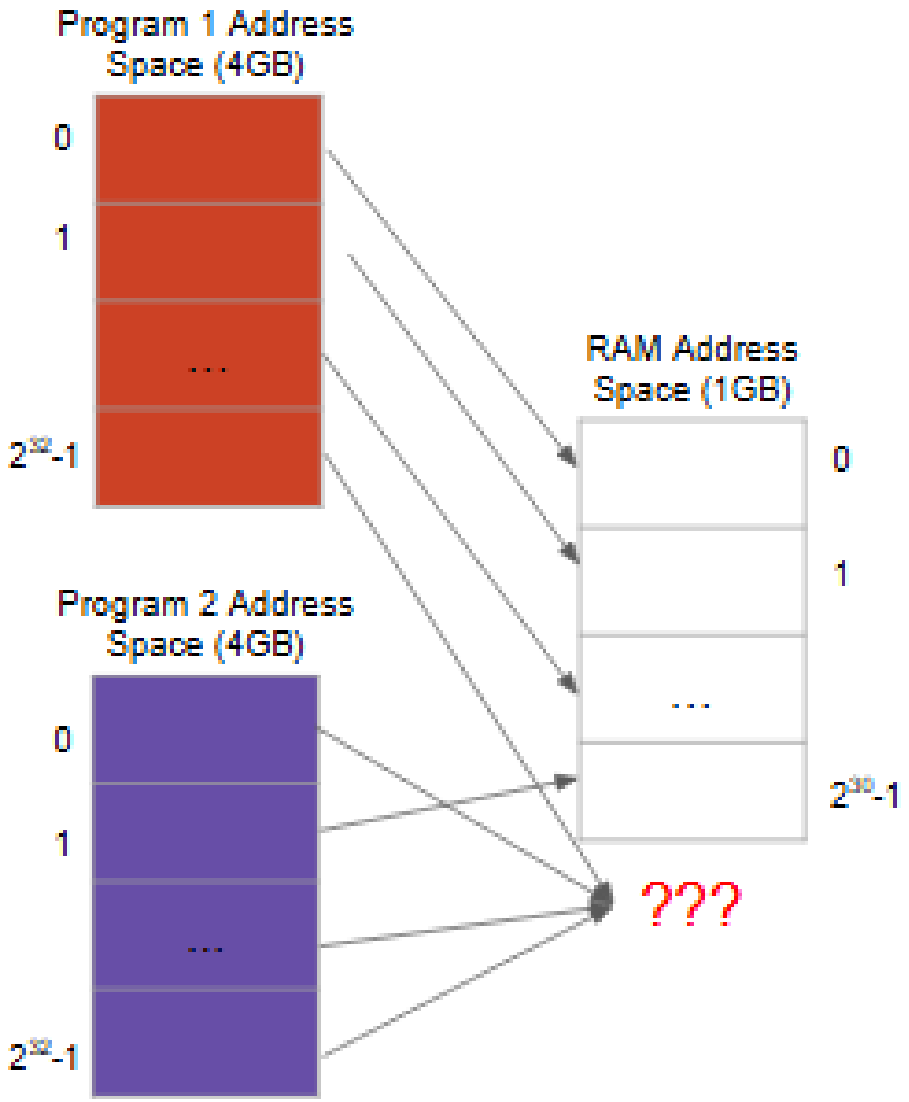
Cal Poly Pomona
ECE 4310

The 3 Memory Issues

1. **Insufficient amount of RAM** to run a single process or multiple processes
2. **Memory Fragmentation**
3. **Security for processes** to ensure they don't write over one another and corrupt data

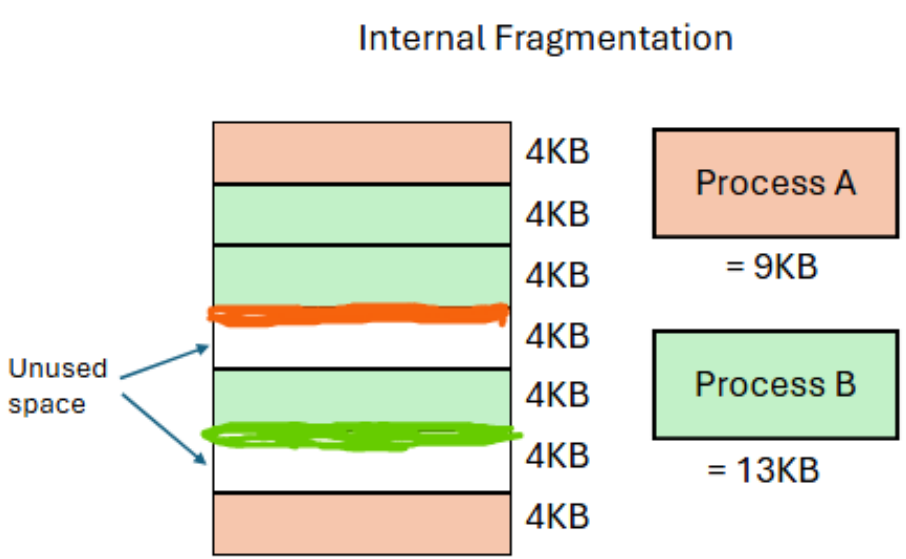
1. Insufficient RAM Space

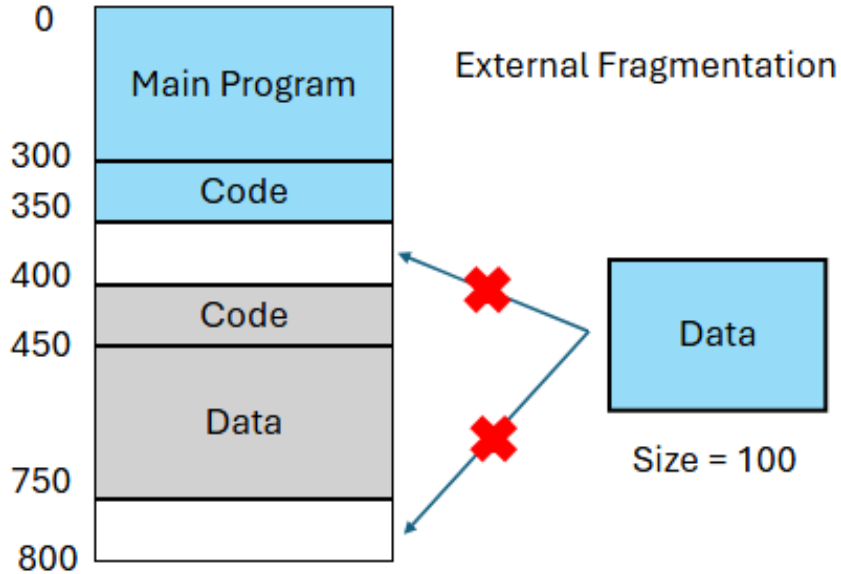
When there isn't enough RAM to supply the demand, the processes will crash which causes problems for the users. The overlying cause for this issue is users will always demand more processes to be run. Adding on to this, processes will always demand more resources from RAM to meet the demands of the user. Thus, **there is never enough RAM to satisfy an unbounded demand**.



2. Memory Fragmentation

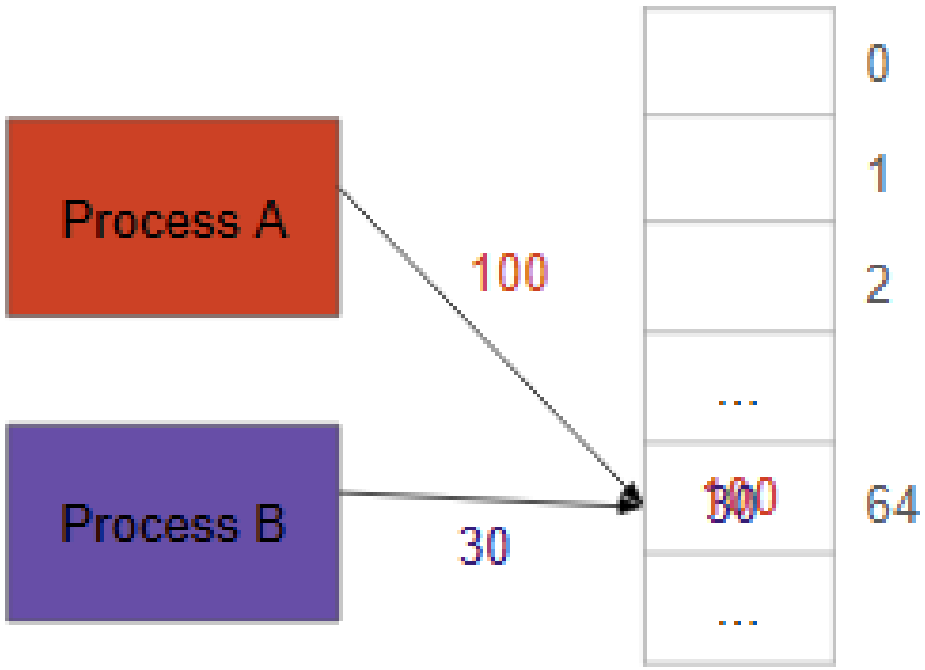
1. **Internal Fragmentation** — Allocated memory is larger than the memory that the process actually needs, leaving unused space.
2. **External Fragmentation** — Free memory is scattered in small blocks throughout the system. Even if the total free memory is enough for a large allocation, it cannot be used because the space isn't contiguous.





3. Process Security

As we've said before, each process can access any 32-bit address and each process thinks it has the entire memory to itself. This creates the possibility of **processes accessing the same address**, which causes the data in the address space to be **unreliable and corrupt**.



The Solution: Virtual Memory

Virtual memory is a system that gives each process the **illusion of a large, continuous block of memory**, even though the actual physical memory (RAM) may be **smaller, fragmented**, or even **partially stored on disk**. It is primarily managed by the operating system (OS) and Memory Management Unit (MMU), that allows:

- **Process isolation**
- **Efficient memory use**
- **Programs to run even if there's not enough physical RAM**

Memory Management Unit (MMU)

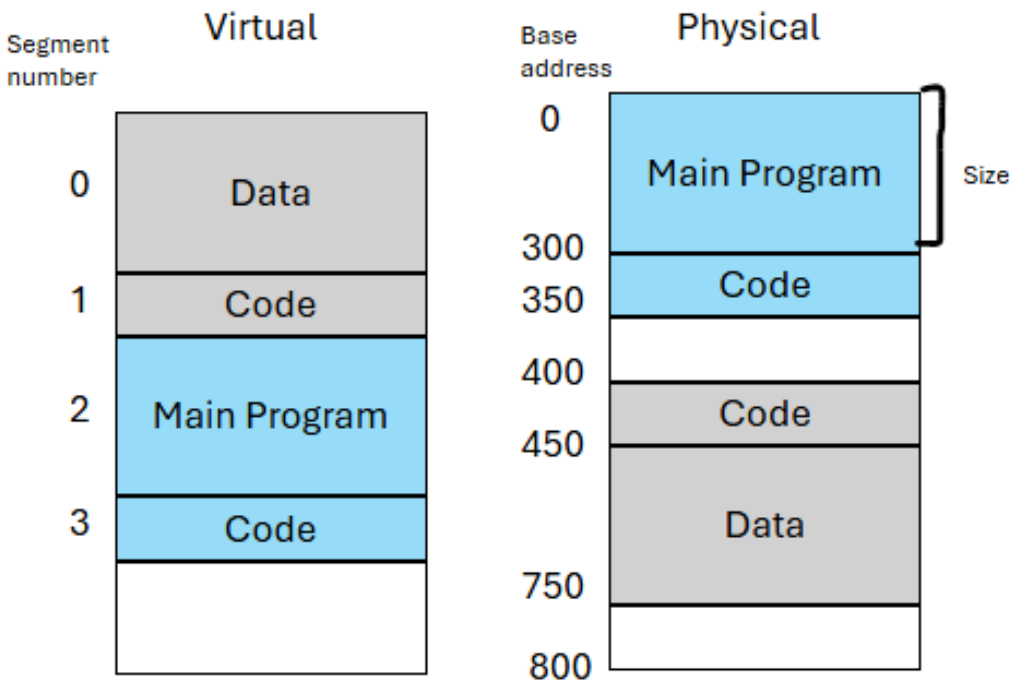
The **Memory Management Unit (MMU)** is a component of the CPU that automatically translates virtual addresses into physical addresses during every memory access. It's the **bridge between software and hardware memory**, handling the logic of where data really lives in RAM.

Note

The MMU also controls **caches, TLB**, and provides a layer of protection with it being able to call **page faults** and control permissions for different address spaces, but all of these will be topics we will cover later on.

Segmentation

Segmentation is a memory management scheme where a process's memory is divided into variable-sized segments, based on logical divisions. Each segment in physical memory is defined by a **base address** (where the segment begins) and a **size** (indicates the segment's limits). Additionally, a **virtual address** is made up of a **segment number** (the segment it wants to access) and an **offset** (where in the segment we want to access).

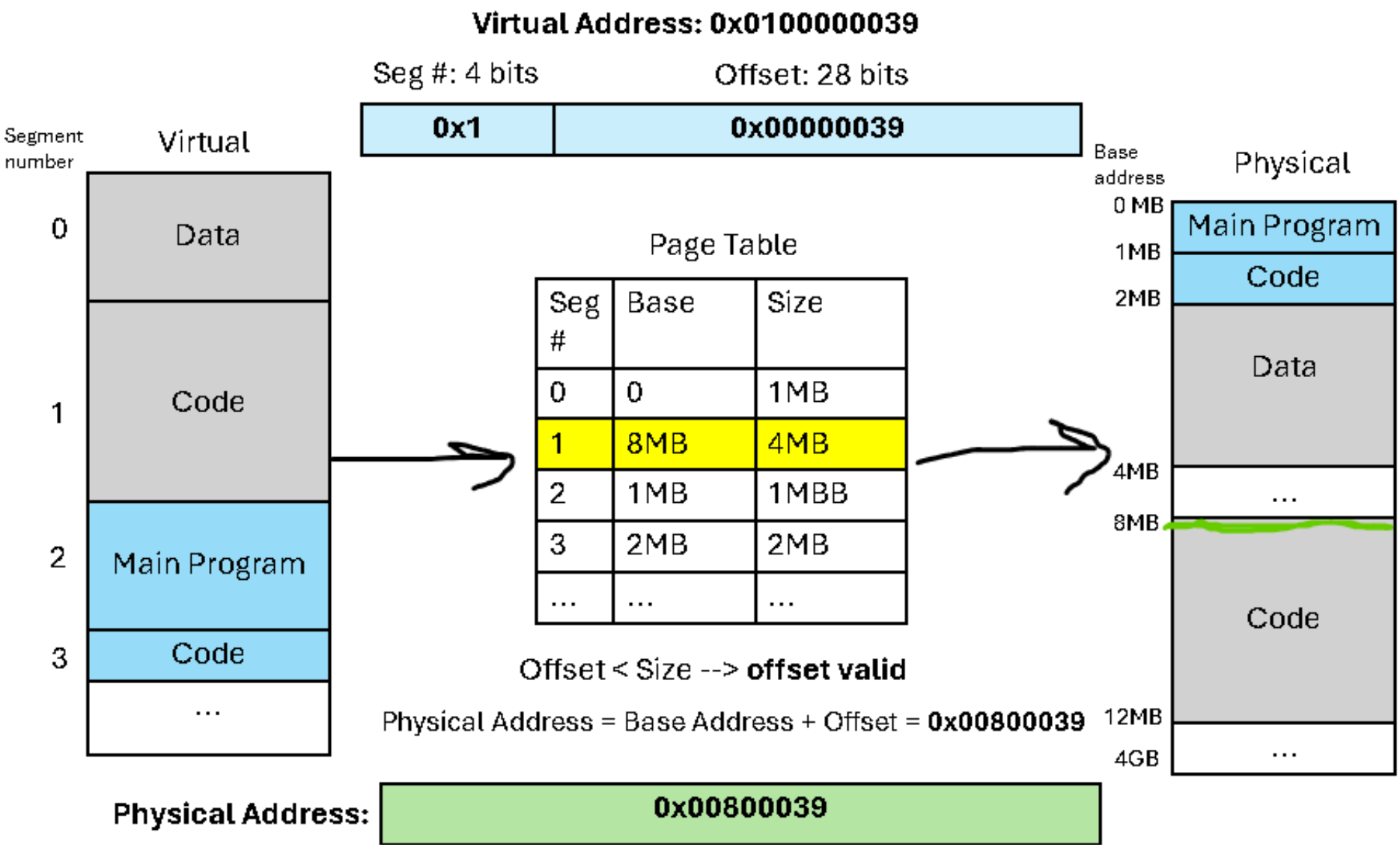


How Does Segmentation Work?

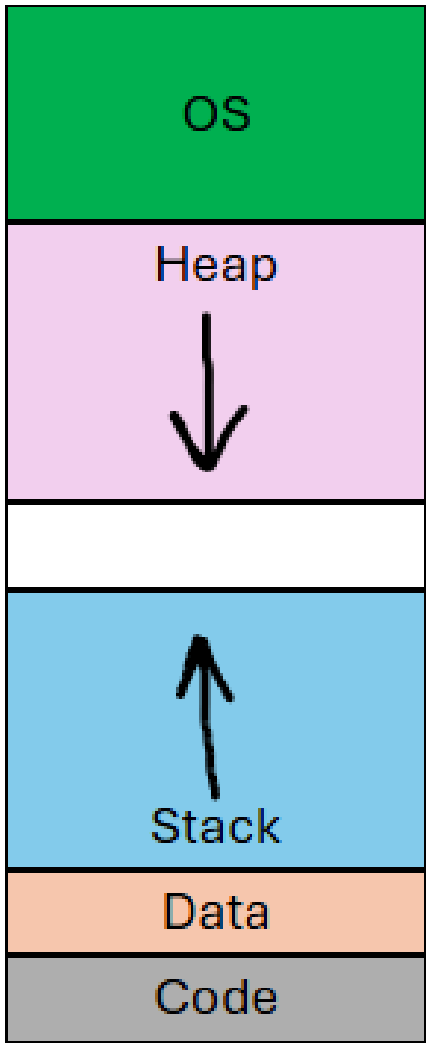
The **segment table** is a data structure used by the MMU to map a virtual segment to its corresponding physical segment. It uses the **segment number** as an index to look up a **segment table entry (STE)**, which contains the base address and size.

1. The MMU uses the segment number to look up the **base address and size** of the physical segment in the segment table.
2. The MMU checks whether the **offset falls within the segment's limit** to ensure the access is valid:
   - If the offset is within bounds, the MMU computes the physical address as: **Physical Address = Base Address + Offset**.
   - If it's out of bounds, a **segmentation fault** occurs.

Example


Virtual Address: 0x0100000039
Seg #: 4 bits
Offset: 28 bits
0x1
0x00000039
Segment number
Virtual
0
Data
1
Code
2
Main Program
3
Code
...
Page Table
Seg #
Base
Size
0
0
1MB
1
8MB
4MB
2
1MB
1MBB
3
2MB
2MB
...
...
...
Offset < Size --> offset valid
Physical Address = Base Address + Offset = 0x00800039
Base address
Physical
0 MB
1MB
2MB
4MB
8MB
12MB
4GB
Main Program
Code
Data
...
Code
...
Physical Address:
0x00800039

Four Different Types of Segments



1. **Code** – Contains the executable instructions of a program.
2. **Data** – Stores global and static variables
3. **Heap** – Used for dynamically allocated memory and grows upward.
4. **Stack** – Used for function call management (local variables, return addresses, function parameters) and grows downward

Drawbacks

External Fragmentation

Complex Allocation

More complex logic is required because the OS has to search for contiguous blocks that are large enough to allocate in RAM or the OS may even have to reorganize the entire RAM in order to create contiguous blocks of free space.

Limited Growth Flexibility

Even though segments can grow through the stack or heap, it won't even matter if there isn't enough space for the segment to expand.

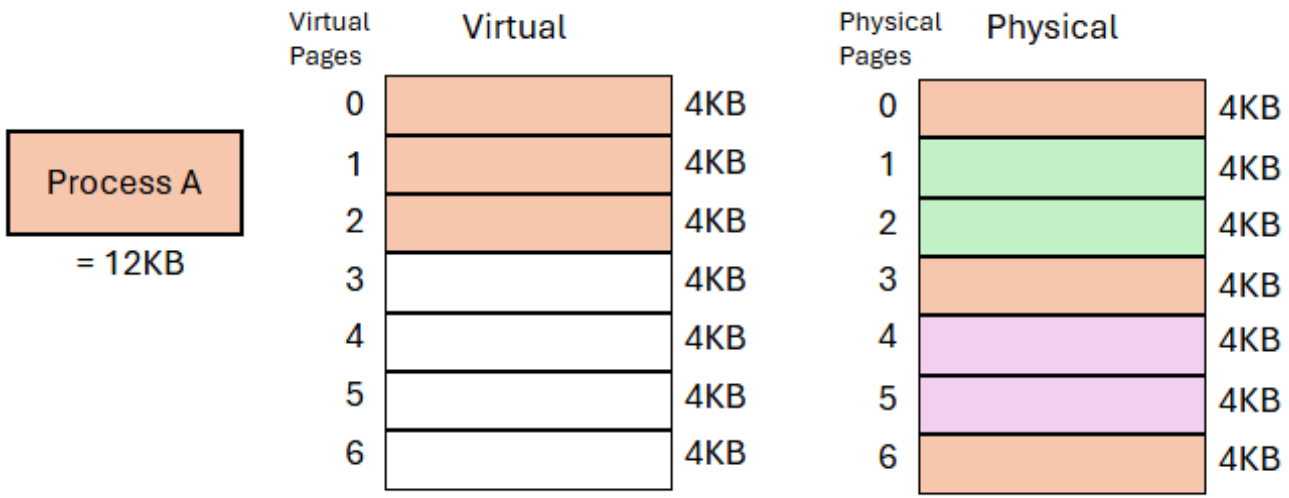
Disk Limitations

Less suitable for utilizing disk storage since this too would be organized in segments and would end up having the same issues as the RAM.

Paging

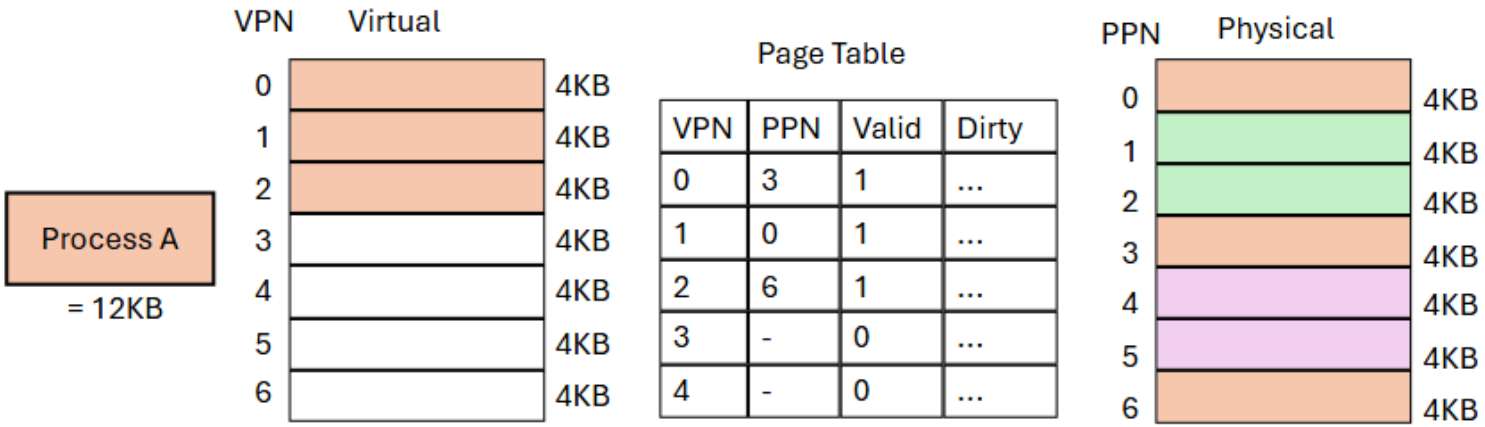
Paging is a memory management scheme that eliminates the need for contiguous memory allocation required with segmentation by dividing both physical and virtual memory into fixed-size blocks called **pages**.

- **Typical page size = 4KB**
- **Virtual and physical pages are the same size**
- **Pages can occupy a whole process, multiple processes, or parts of a process** depending on their sizes



Page Table

Each page within virtual memory will be mapped to a page within physical memory and a data structure called a **page table** is what holds all the **page table entries (PTE)**, which contain the mappings between each virtual and physical memory address and some **flags**, such as **valid (present)** or **dirty**. These flags serve different purposes: valid indicates if the entry is usable, dirty indicates the page has been written to, etc. The page table is stored within physical memory and whenever a virtual address is given, the MMU breaks it down and uses the page table to do the address translation.



How Does it All Work?

The virtual address is made up of a **virtual page number (VPN)** (the page in virtual memory we want to access) and an **offset** (what specific byte we want to access). The physical address is composed the **PPN (Physical Page Number)** which indicates the page in physical memory we want to access and the offset. The process of paging require only two simple steps:

1. The MMU gets the PPN by using the VPN to index into the **page table**, which stores virtual-to-physical page mappings
2. The offset is directly copied from the virtual address to the physical address.

Note

The **page sizes are the same in both virtual and physical memory**, so the **offset remains unchanged during translation**. Only the **page number is translated via the page table**.

Example

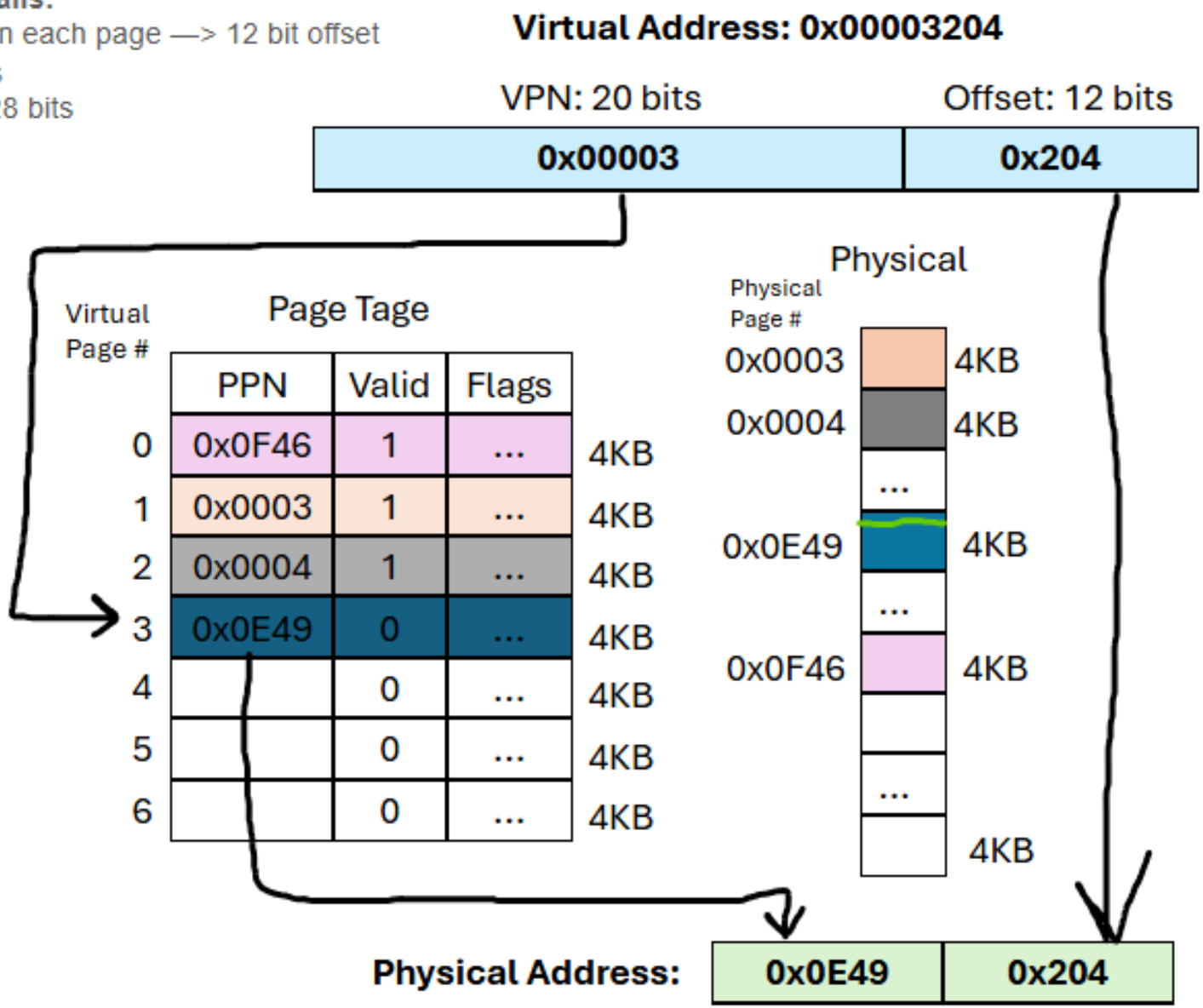
32-bit Virtual Address Details:

4KB pages = 2^{12} bytes within each page —> 12 bit offset

VPN size = 32 - 12 = 20 bits

Physical Address Space = 28 bits

- 12 bit offset & 16 bit PPN



Page Faults

A **page fault** occurs when a process tries to access a virtual page whose PTE indicates that the page is not currently in physical memory (RAM). In this case, the MMU triggers a **page fault exception** and control is passed to the OS's **page fault handler**, which is a special function in the OS kernel.

Note

Page faults typically occur when the page is stored on disk, but they can also trigger for **invalid memory accesses** or **page table mappings that haven't been updated**. Depending on what triggered the page fault, the OS either **loads the page**, **updates the page table**, or **terminates the process**.

Drawbacks

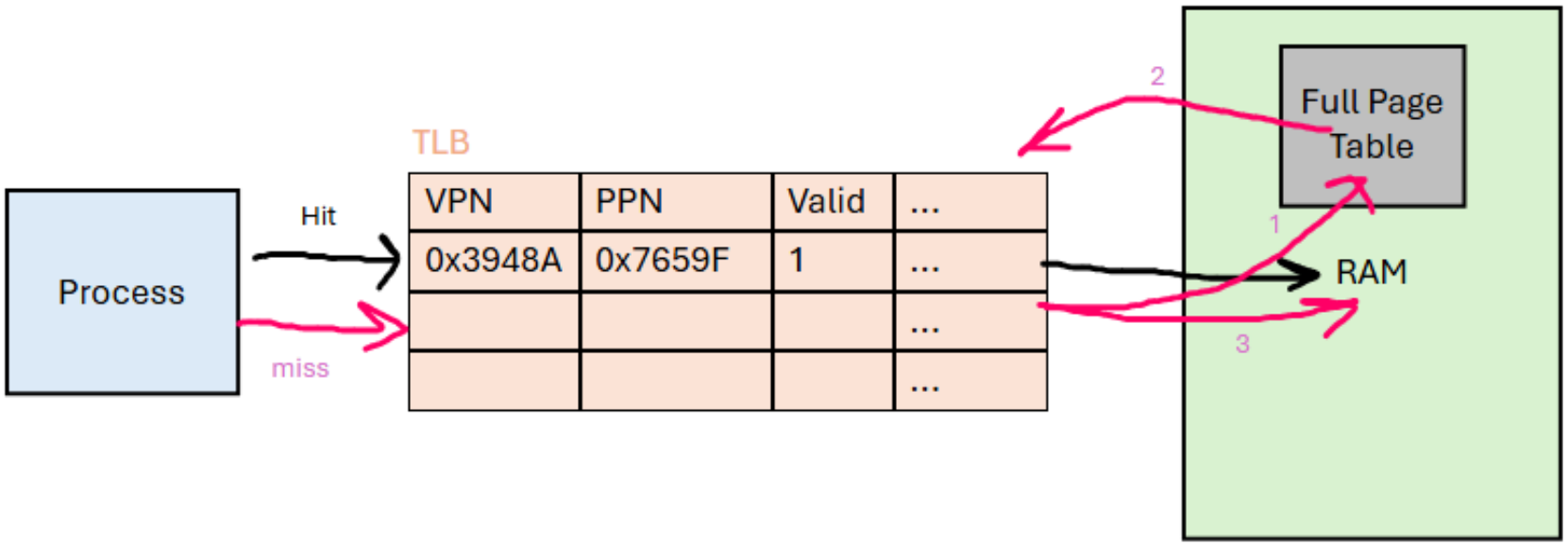
1. **Single-level page tables can be extremely large**, requiring a significant amount of physical memory just to store the page table entries.
2. **Internal Fragmentation**
3. Although it's possible, **moving the entire page table to disk is not practical**. It's slower than accessing RAM and introduces complexity, as there's no built-in mechanism to track where the page table is stored when it's not in RAM.

Note

Each process requires its own page table, which all reside in the physical memory. With multiple processes running, these drawbacks become significant concerns.

Caches

A **cache** is a **small, fast memory** that stores copies of **frequently accessed data** to speed up future access. The specific one we'll look into is a **Translation Lookaside Buffer (TLB)**, which is located within the **MMU** and it stores **recent virtual-to-physical page translations**.



Why are Caches Necessary?

In a single-level paging system, every memory access requires the MMU to read the **page table from RAM** to complete the virtual-to-physical address translation. This adds **extra memory accesses** before the actual data can be fetched. If a page is accessed multiple times, this **repeated translation becomes costly**. To avoid this overhead, systems use **caches to store recent page table lookups**, allowing much faster translation.

Note

Although this scenario might not seem too bad, it's important to consider **modern systems run multiple processes concurrently** and will frequently be accessing memory. To make things worse, these systems also use **multi-level pages**, so it takes a **significant amount of time** for a process to access the physical memory.

How does a TLB work?

When the CPU issues a **virtual address**, the **MMU first checks the TLB** to see if it already contains the **virtual-to-physical address mapping**. One of two things can happen:

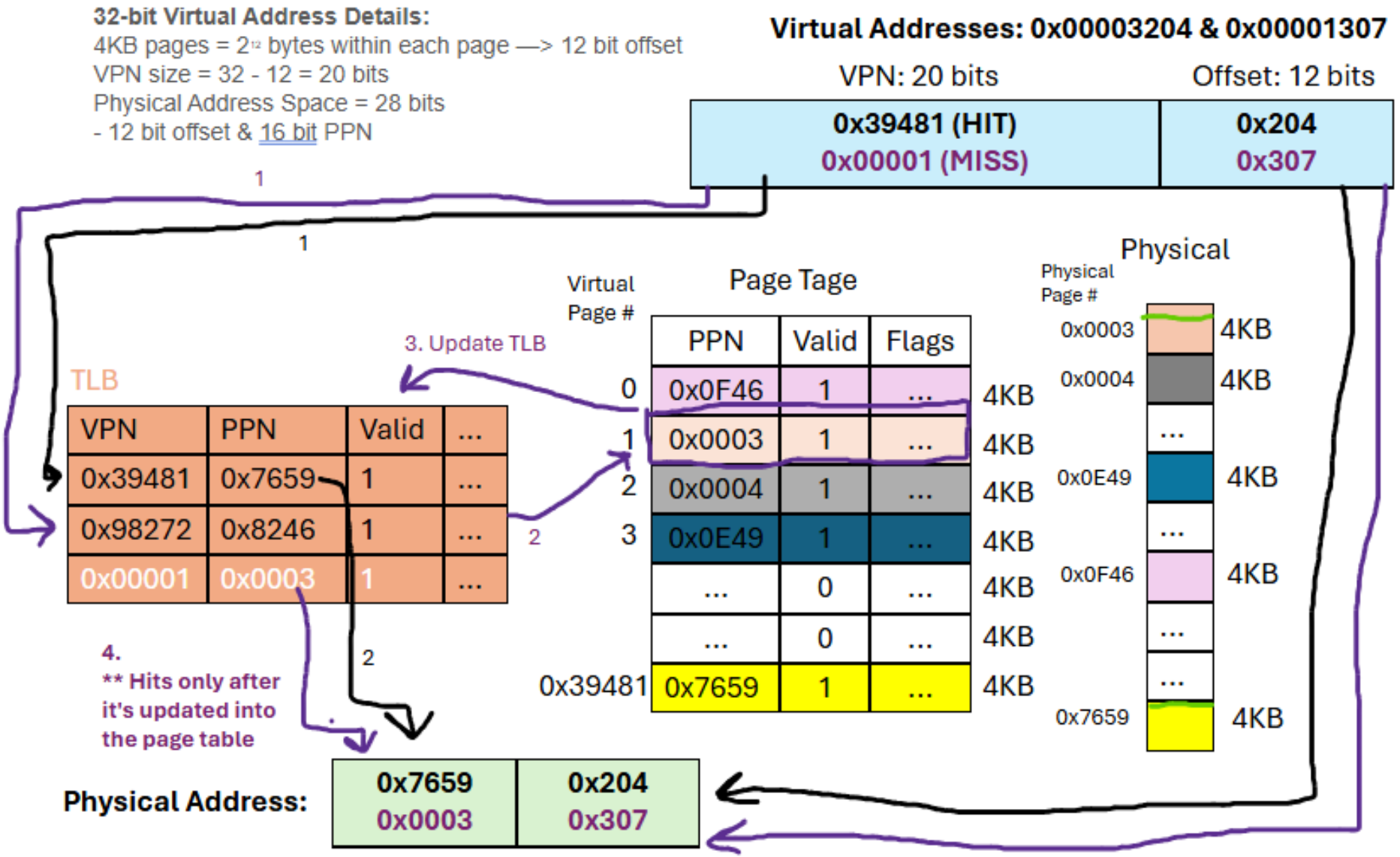
1: TLB Hit

TLB Hit: the mapping is found in the TLB, allowing the MMU to translate the virtual address into a physical address quickly and grant the process access to the physical memory.

2: TLB Miss

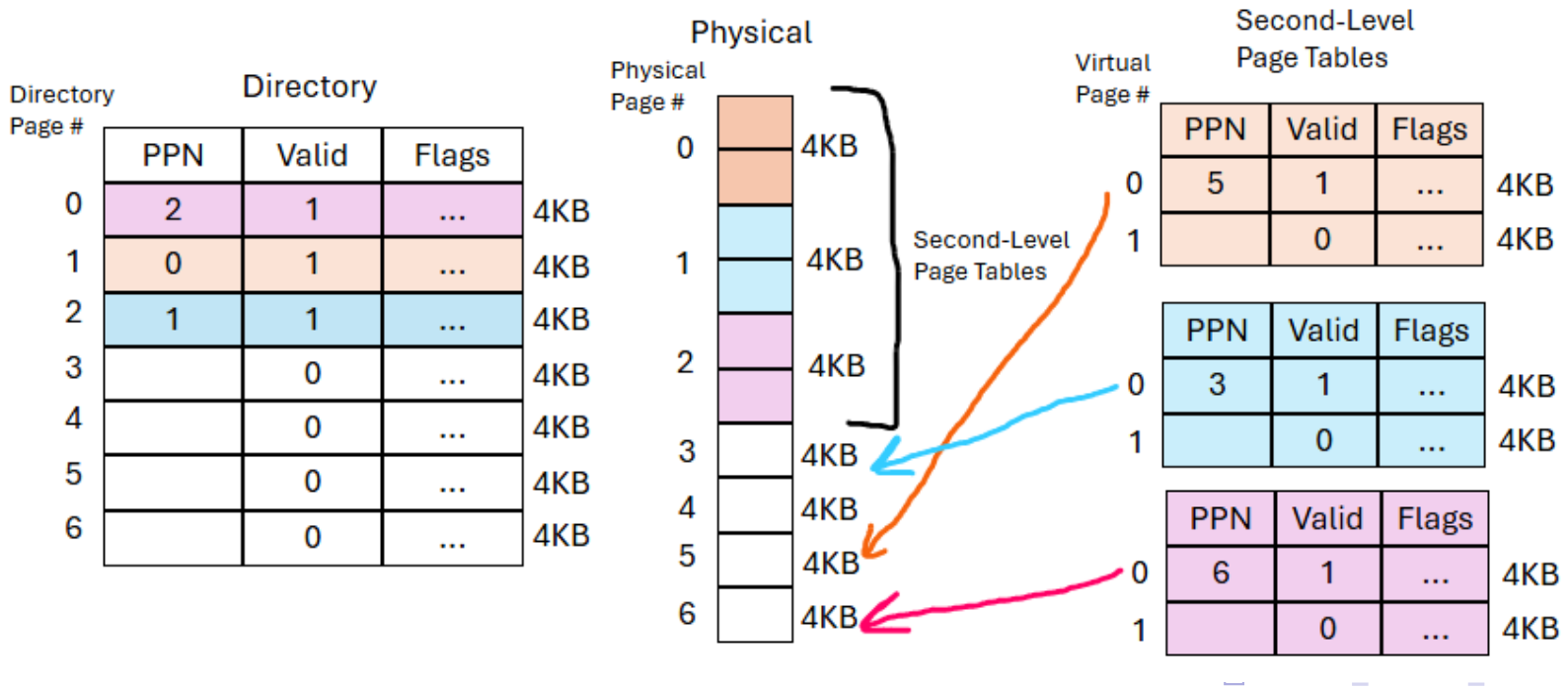
TLB Miss: the mapping is not in the TLB, so the MMU must walk the **page table** to find the corresponding physical address. Once found, the TLB is updated with the new mapping so that **future accesses to the same page are faster**.

Example


32-bit Virtual Address Details:
4KB pages = 2^12 bytes within each page —> 12 bit offset
VPN size = 32 - 12 = 20 bits
Physical Address Space = 28 bits
- 12 bit offset & 16 bit PPN
Virtual Addresses: 0x00003204 & 0x00001307
VPN: 20 bits
Offset: 12 bits
0x39481 (HIT)
0x00001 (MISS)
0x204
0x307
1
1
TLB
VPN
PPN
Valid
...
0x39481
0x7659
1
...
0x98272
0x8246
1
...
0x00001
0x0003
1
...
3. Update TLB
2
2
Virtual Page #
Page Tage
PPN
Valid
Flags
0
0x0F46
1
...
4KB
1
0x0003
1
...
4KB
2
0x0004
1
...
4KB
3
0x0E49
1
...
4KB
...
0
...
4KB
...
0
...
4KB
0x39481
0x7659
1
...
4KB
Physical
Physical Page #
0x0003
4KB
0x0004
4KB
...
0x0E49
4KB
...
0x0F46
4KB
...
...
0x7659
4KB
4.
** Hits only after it's updated into the page table
Physical Address:
0x7659
0x0003
0x204
0x307

Multi-Level Paging

Multi-level paging is a memory management technique that replaces a **single, large flat page table** with a **hierarchy of smaller page tables**. Instead of storing one long list of PTEs, the system **breaks the page table into multiple levels**. The **top/first level table** (**page directory**) points to **lower-level page tables** and these contain the actual mappings to **physical page numbers**.



Why is Multi-Level Paging Necessary?

Multi-level paging is necessary because it **addresses the major drawbacks of single-level paging**, particularly the issue of **memory waste**.

1st

Multi-level paging allows the system to **only allocate page tables when they are actually needed**, which makes it **much more memory-efficient**, especially for processes that **use only a small portion of their virtual address space**.

2nd

Multi-level paging **reduces the memory footprint of page tables in RAM** since **lower-level page tables that are not actively needed can be paged out to disk** and their locations can be tracked via entries in page directory.

How does Multi-Level Paging Work?

Multi-Level paging works similarly to single-level paging, except the **virtual address is split into three components**: the **directory page number (DPN)**, the **VPN**, and **offset**. The DPN usually consists of the most significant bits and is the **index into the page directory**. The **page directory entry (PDE)** at the DPN index points to a **second-level page table**, which is used just like in single-level paging to find the **PPN**

Note

A PDE can map to a page table within disk, which would cause the MMU to **trigger a page fault**. Page faults in multi-level paging **function the same as single-level page faults**, but instead of **loading a single page into RAM** we are **loading entire page tables**.

Example

32-bit Virtual Address Details:
4KB pages = 2^{12} bytes within each page → 12 bit offset
4B PTEs → Size of page tables = 2^{12} B / 2^2B per entry = 2^{10} entries
DPN & VPN size = 10 bits
Physical Address Space = 28 bits --> 12 bit offset & 16 bit PPN