**Background:**

* A program must be loaded from disk into memory to be executed within a process.
* The CPU can only directly access the main memory and registers.
* The memory unit sees a stream of addresses and either read or write requests.
* Accessing data in registers is very fast (one CPU clock cycle or less).
* Memory management is about efficiently managing processes in main memory.
* The goal of memory management is to efficiently utilize the memory

**1. CPU's Direct Access to Memory**

* The core of your computer (the CPU) can only work directly with information that's in two places:
  + **Main Memory:** This is the computer's primary working memory (RAM).
  + **Registers:** These are small, very fast storage locations built right into the CPU itself.
* Think of it this way: the CPU has instructions to grab data from main memory or registers, but it doesn't have instructions to directly grab data from your hard drive or SSD.
* So, if a program or data is on the disk and the CPU needs it, the computer *must* first copy that program or data from the disk into main memory.

**2. Speed Difference: Registers vs. Main Memory**

* Accessing data in CPU registers is incredibly fast. It can happen within a single "tick" of the CPU's internal clock. CPUs are designed to get information from their registers almost instantly.
* Accessing main memory is much slower. When the CPU needs data from main memory, it has to send a request over the "memory bus" (a communication pathway). This process takes many CPU clock cycles.
* Because main memory is slower, the CPU often has to "stall" or wait. It's like waiting for someone to hand you a tool before you can continue building something. This waiting is a problem because CPUs are designed to process information very quickly.

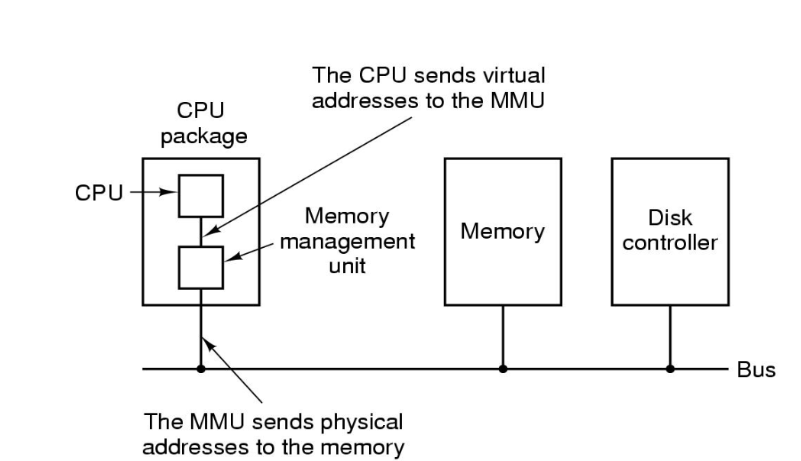
**3. The Cache Solution**

* To reduce the CPU's waiting time, computers use a special, fast memory called "cache."
* The cache is a smaller, faster memory that sits between the CPU and main memory. It stores frequently used data so the CPU can access it quickly.
* When the CPU needs data, it first checks the cache. If the data is there (a "**cache hit**"), the CPU gets it very fast. If not (a "**cache miss**"), the CPU retrieves it from main memory (which is slower) and also copies it into the cache for future use.
* The cache is managed by **hardware**, so this speeding up of memory access happens automatically without the operating system having to do much.
* In some advanced CPUs, if one thread of execution has to wait for memory, the CPU can switch to working on a different thread to stay busy.

**4. Memory Protection**

* It's crucial to protect different parts of memory.
  + The operating system (which controls the computer) needs to be protected from user programs. If a user program could accidentally or intentionally overwrite the OS, the system would crash.
  + User programs also need to be protected from each other**. If one program could freely access another program's memory,** it could lead to data corruption or security breaches.
* This **protection** is mainly handled by the computer's hardware (**MMU**). It's more efficient for the hardware to enforce these **rules** than for the operating system to check **every single memory access.**

==========================================================================

**CPU Package:** This box represents the physical chip that contains the Central Processing Unit (CPU) and, in this diagram, the Memory Management Unit (MMU) as well.

**CPU**: The core of the computer that executes instructions. It generates virtual addresses. Think of virtual addresses as logical addresses used by the programs. Each program thinks it has its own continuous block of memory starting from address zero.

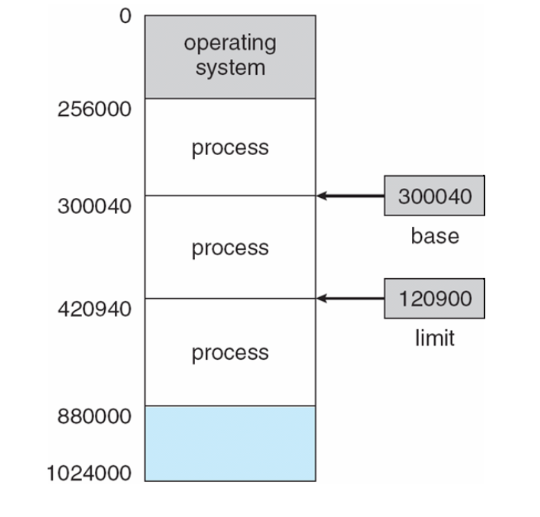
**Memory Management Unit (MMU):** This is a hardware component that is integrated in the CPU chip. Its main job is to handle memory access requested by the CPU, as shown by the arrows, by performing the translation between virtual addresses and physical addresses.

**The CPU sends virtual addresses to the MMU:** The arrow shows the flow of a virtual address from the CPU to the MMU when the CPU needs to access memory.

**The MMU sends physical addresses to the memory:** The arrow shows the MMU taking the virtual address, looking up the corresponding physical address (using tables stored in memory), and then sending this physical address to the actual Memory (RAM). Physical addresses are the actual locations in the computer's main memory.

**Memory**: This represents the computer's main memory (RAM). It receives physical addresses from the MMU and uses these addresses to access the requested data.

**Disk Controller:** This is responsible for managing the communication between the main memory and the disk (hard drive or SSD). It's shown for context but isn't directly involved in the virtual-to-physical address translation.

**Bus**: This is the electrical pathway that connects different components of the computer, allowing them to communicate with each other.

**Base & Limit Register:**

Processes should not be able to access memory locations of other processes without permission.

**Base** and **limit** registers define the logical address space for a process.

The CPU checks every memory access in user mode to ensure it's within the base and limit for that process.

To protect process from each other: each process has its own memory space

The diagram shows an example:

The operating system is in **low** **memory**.

Processes are in **higher** **memory**.

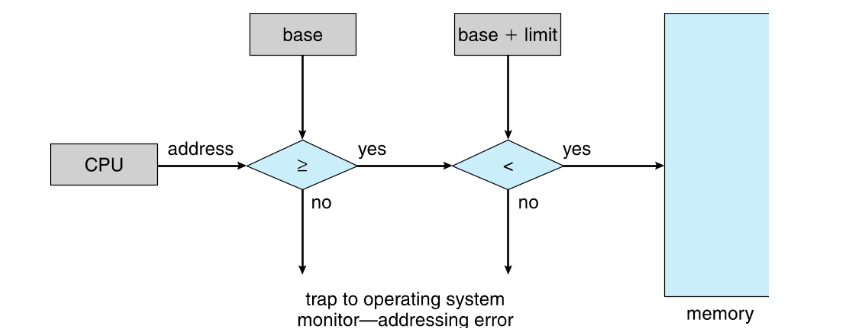
The **base register** (**300040** in the example) holds the starting address of a process's memory.

The **limit register** (**120900** in the example) specifies the size of the process's memory

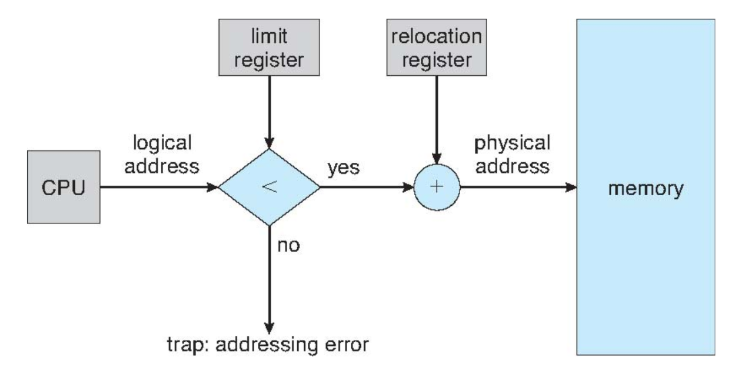
Base is the starting point of the process whereas limit is the total size allocated to that particular process, the process can’t go beyond that size or limit.

**Hardware Address Protection:**

**1. Without MMU:**



**2. With MMU**



It emphasizes that the CPU must check every memory access to ensure it's within the base and limit.

**The diagram shows the comparison:**

* The CPU generates an address.
* The address is checked to see if it's greater than or equal to the base register. (greater than the earlier process)
* If it is, it's then checked to see if it's less than the "base + limit".   (less than the next process)
* If either check fails, it's a memory access error, and a trap (interruption) occurs, sending control to the operating system.

**Logical vs Physical Address Space:**



A close up of a text

AI-generated content may be incorrect.

A diagram of a medical address

AI-generated content may be incorrect.

* The concept of a logical address space separate from a physical address space is crucial for memory management.
* Logical Address (Virtual Address): Generated by the CPU.
* Physical Address: The address seen by the memory unit.
* Logical and physical addresses are the same in compile-time and load-time address binding.
* They differ in execution-time address binding.
* Logical address space is the set of all logical addresses generated by a program.
* Physical address space is the set of all physical addresses corresponding to these logical addresses.

**Memory-Management Unit (MMU):**

* The **MMU** is a hardware component that maps virtual addresses to physical addresses at **runtime**.
* The Memory Management Unit is a combination of 2 registers -
  + - Base Register (Relocation Register)
  + Limit Register.
* It involves adding (sum) a "**relocation/base register**" value to every **logical/virtual address** generated by a user process during the execution.
* Relocation Register is a hardware register that stores a number called the **base address** (used to relocate a process’s starting address).
* It **replaces** the older concept of a "**base register**."
* The **base** register is now called the **relocation register**.
* The user program works with *logical addresses* and never sees the actual *physical addresses*.
* Execution-time binding occurs when a memory location is *referenced*.
* Logical addresses are bound to physical addresses.
* If the address is outside the bounds defined by the limit register, the MMU detects a **segmentation fault** or **memory access violation** and signals the CPU (which then typically triggers an operating system exception or error).

A screenshot of a computer program

AI-generated content may be incorrect.

A black background with white text

AI-generated content may be incorrect.A black screen with white text

AI-generated content may be incorrect.A screenshot of a computer

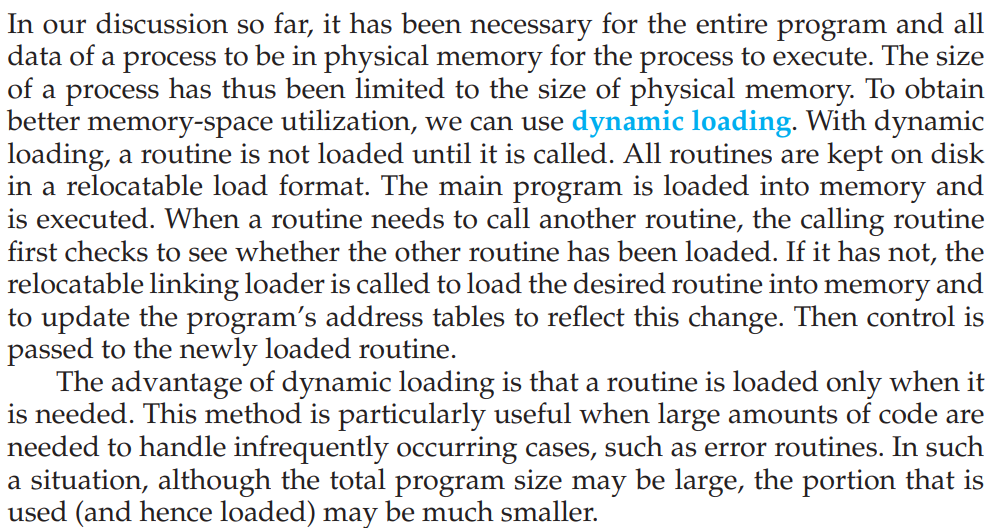
AI-generated content may be incorrect.

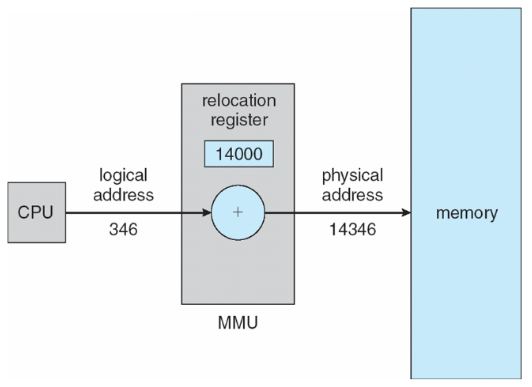
**Dynamic Relocation using Relocation Register:**

**Dynamic relocation** is a memory management technique where the actual physical address of a program's instructions and data is determined *at runtime* when the program is loaded into memory or even during its execution. This contrasts with static relocation, where addresses are fixed at compile or link time.

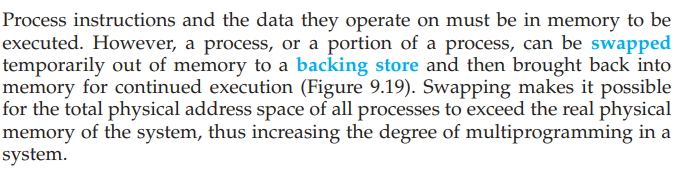
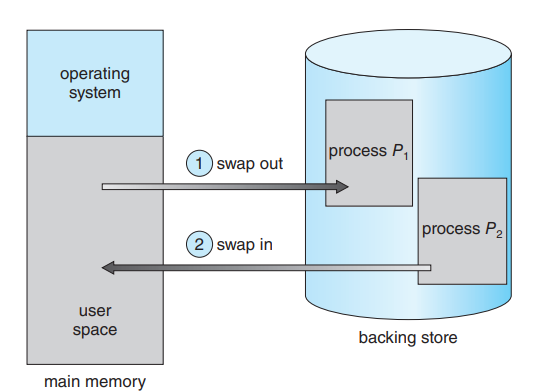
**Hardware Support:** Dynamic relocation heavily relies on hardware support from the CPU and the Memory Management Unit (MMU).

* A routine (a piece of code) is not loaded into memory until it is called.
* This improves memory utilization because unused routines are never loaded.
* All routines are kept on disk in a relocatable load format.
* This is useful for large programs that have code to handle infrequent events.
* No special operating system support is required; it can be implemented through program design.
* The OS can help by providing libraries for **dynamic loading.**





**Swapping:**



A close up of a text

AI-generated content may be incorrect.

**Increasing the Degree of Multiprogramming:** The primary goal of swapping is to enable a higher degree of multiprogramming. By temporarily removing **inactive** or **lower-priority** processes from memory, the system can load and run more processes concurrently than would be possible if all processes had to reside in memory entirely. The total virtual memory space used by all processes can exceed the actual physical memory available.

**Roll Out, Roll In:** This is a specific variant of swapping often used in priority-based scheduling. A lower-priority process is "rolled out" (swapped out) to free up memory for a higher-priority process to be "rolled in" (swapped in) and executed. Once the higher-priority process finishes, the lower-priority process can be rolled back in.

**Does the swapped-out process need to swap back into the same physical addresses?**

The answer to this question **depends on the address binding method** used by the operating system:

* **Static Address Binding:** If a process's addresses are bound to specific physical memory locations at compile or load time (static relocation), then the process **would ideally need to be swapped back into the same physical addresses**. If those addresses are occupied by another process in the meantime, swapping back becomes problematic or impossible without significant memory reorganization. This rigidity is a major disadvantage of **static address binding** in a swapping environment.
* **Dynamic Address Binding (using a relocation register or similar mechanism):** If the system uses dynamic relocation, where logical addresses are translated to physical addresses at runtime using a base register, then the swapped-out process **does not need to be swapped back into the same physical addresses**. When the process is swapped back in, the operating system can load it into any available block of physical memory and update the process's base register accordingly. Modern systems heavily rely on **dynamic address binding**, which makes swapping much more flexible.

**Pending I/O to/from Process Memory Space:**

This is a critical constraint on swapping. If a process has pending I/O operations (data transfers in progress between the process's memory space and an I/O device), **it cannot be safely swapped out**. Here's why:

* The I/O operation is likely using Direct Memory Access (DMA), where the I/O device directly reads from or writes to the physical memory addresses allocated to the process.
* If the process is swapped out, the physical memory it was using might be allocated to another process.
* If the I/O operation completes while the original process is swapped out, the data transfer would occur to the wrong memory locations, leading to data corruption and system errors.

There are a couple of ways operating systems handle this:

* **Prevent Swapping of Processes with Pending I/O:** The simplest approach is to prevent a process from being swapped out if it has any pending I/O operations. The OS keeps track of the I/O status of processes and makes swapping decisions accordingly.
* **Always Transfer I/O to Kernel Space (Double Buffering):** A more complex but safer approach is to always transfer I/O data to kernel space first, and then after the process is swapped in copy it to the user process's memory space. This is known as **double buffering**.
  + When a process initiates an output operation, the data is first copied to a kernel buffer. The I/O device then reads from the kernel buffer. The process can be swapped out after the data is in the kernel buffer. When the process is swapped back in, the data can be copied from the kernel buffer to the process's memory.
  + Similarly, for input operations, the I/O device writes data to a kernel buffer. The process is swapped in, and the data is then copied from the kernel buffer to the process's memory space.
  + **Overhead:** Double buffering adds overhead due to the extra data copy between user space and kernel space.

**Modern Swapping Strategies:**

* **Swapping Only When Free Memory is Extremely Low:** Swapping is typically disabled under normal memory conditions. It is often activated only when the system is experiencing severe memory pressure (i.e., the amount of free physical memory falls below a certain threshold). This acts as a last resort to prevent the system from crashing or becoming completely unresponsive due to memory exhaustion.
* **Swapping Out Idle or Less Active Processes:** Modern swapping mechanisms often prioritize swapping out processes that have been idle for a long time or have a lower priority, as these are less likely to be needed immediately.
* **Swap Partitions/Files:** Operating systems typically designate a special partition on the hard disk (swap partition) or a dedicated file (swap file) to serve as the backing store.
* **Memory Compression:** Some modern systems employ memory compression techniques as a first line of defense against memory pressure. Instead of immediately swapping out, they try to compress less frequently used memory pages to free up space. Swapping is used only if compression is insufficient.

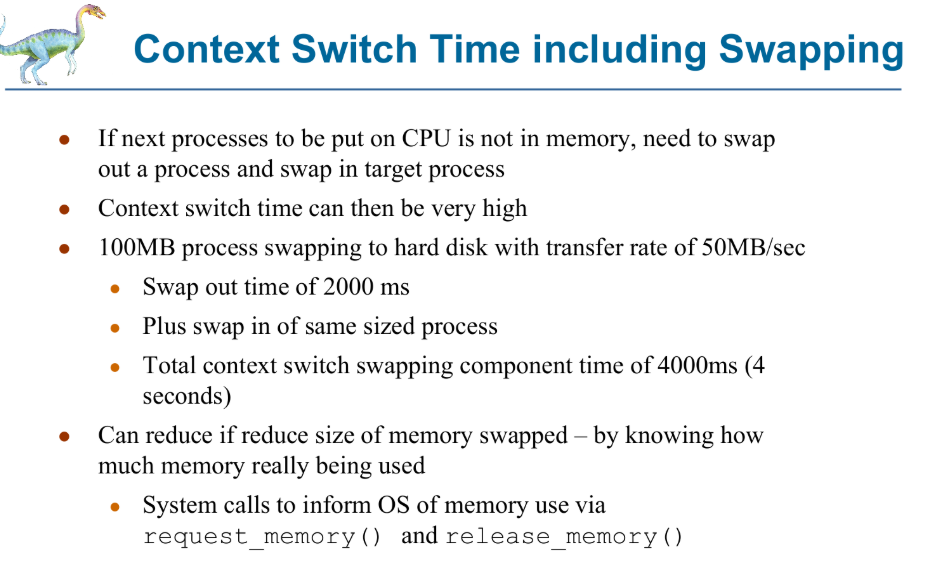
**Swap Time and Performance Implications:**

* **Major Part of Swap Time is Transfer Time:** The dominant factor in the time it takes to swap a process is the **transfer time** between main memory and the backing store.
* **Proportional to Amount of Memory Swapped:** The total transfer time is directly proportional to the size of the memory image being swapped. Larger processes take significantly longer to swap.
* **Example:** A 100MB process being swapped to a hard disk with a transfer rate of 50MB/sec would take:
  + **Swap out time** = 100 MB / 50 MB/sec = 2 seconds (2000 ms)
  + **Swap in time** (assuming the same size) = 100 MB / 50 MB/sec = 2 seconds (2000 ms)
  + **Total context switch swapping component time** = 4 seconds (4000 ms). This is a very significant delay compared to a regular context switch between processes that are already in memory (which typically takes microseconds).

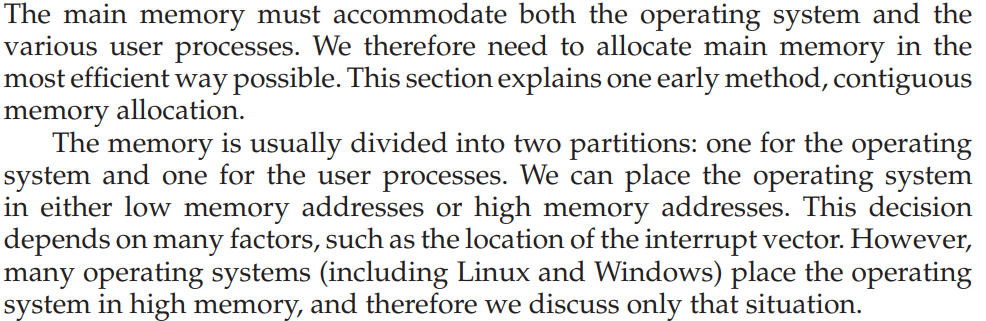
**Reducing Swap Time:**

To mitigate the performance impact of swapping, operating systems try to:

* **Minimize the amount of memory swapped:** This can be achieved by being selective about which processes are swapped and by tracking how much of a process's allocated memory is actually being used.
* **Use faster backing stores:** Solid-state drives (SSDs) offer much higher transfer rates than traditional hard disk drives, significantly reducing swap times.
* **Employ memory compression:** As mentioned earlier, compressing memory can reduce the amount of data that needs to be swapped to disk.



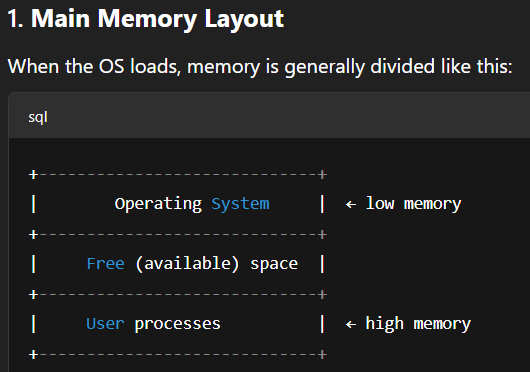
**Contiguous Memory Allocation:**

****

**A close up of text

AI-generated content may be incorrect.**



****

**Contiguous Memory**: The memory is usually divided into two partitions - one for the **operating system** and one for the **user processes**.

**The Basic Idea**

* When a computer runs programs, it needs to store those programs and their data in its main memory (RAM).
* One way to manage this is to divide the memory into chunks (contiguous section of memory), and each chunk can hold one program.
* These chunks can be different sizes, which is why it's called "**variable-partition**".

**How It Works**

* The operating system keeps track of which parts of memory are free and which are being used.
* When the computer starts, all of the memory is free. It's like one big empty space. This free space is called a "hole".
* As programs start, they are put into these "holes." So, the memory gets filled up with programs.
* Over time, programs finish and are removed from memory. This creates new "holes". Now you have a mix of programs and empty spaces scattered throughout the memory.

**Dealing with Limited Memory**

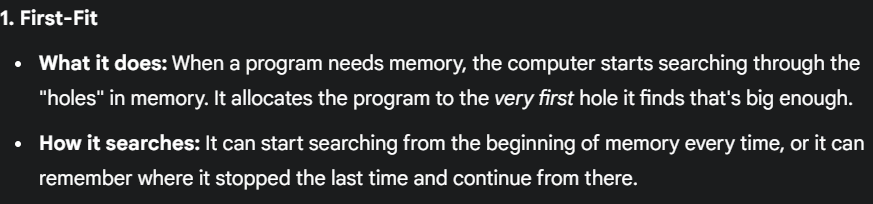
* What if a new program wants to start, but there's not enough free space?
  + The computer could just say "no" and give an error.
  + Or, it could make the program wait in line. When another program finishes and frees up space, the waiting program can then be loaded.

**Keeping Memory Organized**

* When a program needs memory, the computer looks for a free "hole" that's big enough.
* If the hole is bigger than what the program needs, the hole is split. One part is used for the program, and the other part stays free to be used later.
* When a program is done, it frees up its memory, creating a new "hole".
* If this new "hole" is next to other free "holes," they are combined into one bigger "hole" to avoid having lots of tiny, unusable free spaces.

****

**Memory Allocation Strategies:**

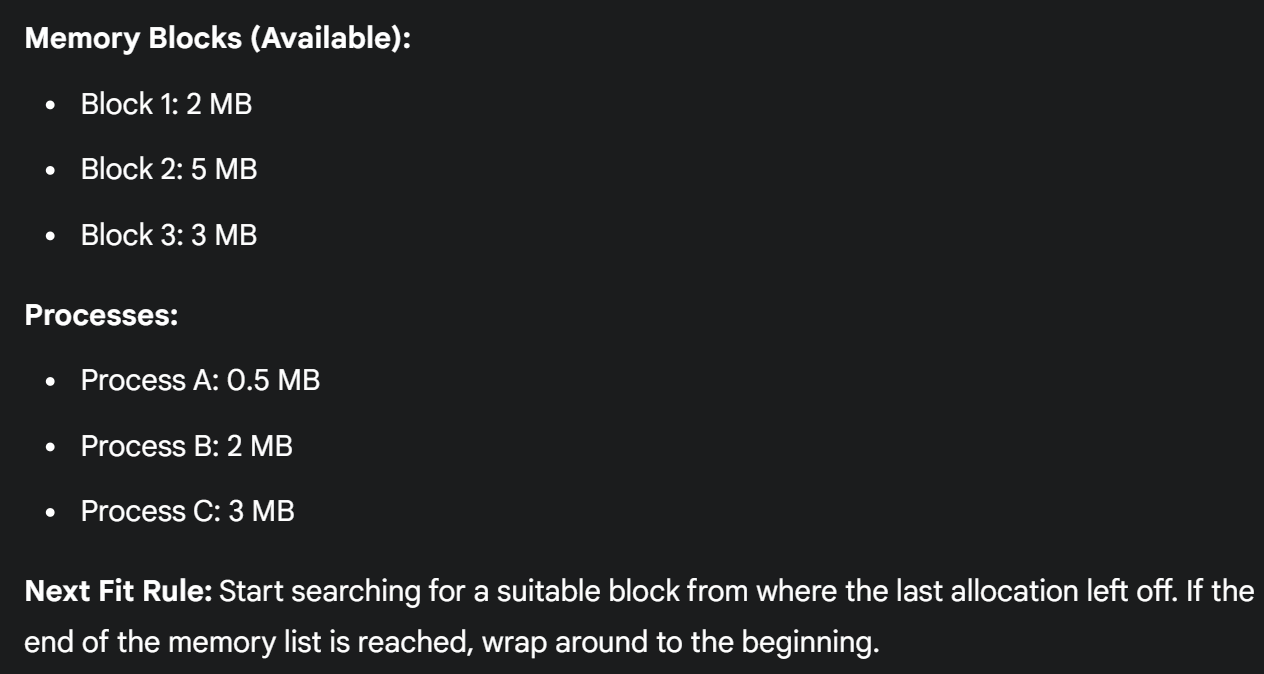


A black background with white text

AI-generated content may be incorrect. A black background with white text

AI-generated content may be incorrect.

**4. Next-fit:** Allocate the next available hole that is big enough.

A black background with white text

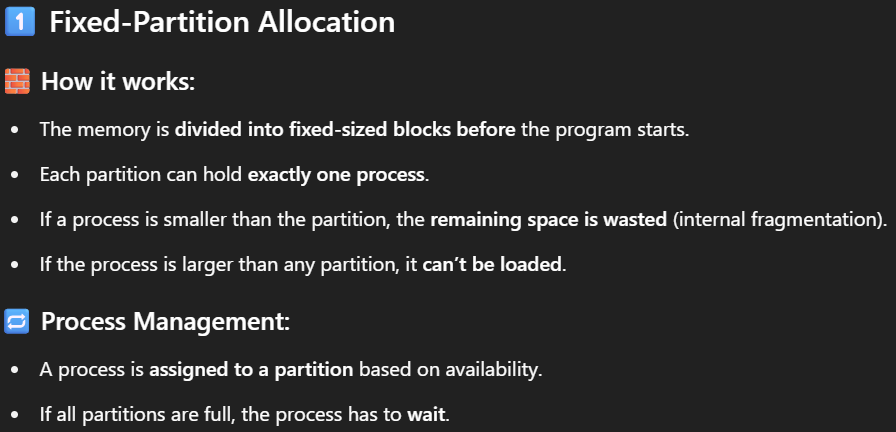
AI-generated content may be incorrect.

**Multiple Partition Allocation:**

There are two types of **contiguous memory allocation**:

**1). Fixed Partition Allocation:** The memory is pre-divided.

**2). Variable/Dynamic Partition Allocation:** The memory is **not** pre-divided.

****

**A diagram of process and process

AI-generated content may be incorrect.1.1** **Fixed Equal Partitions:**

Main memory is divided into a **fixed number of partitions**, and all partitions are of the **same size**. When a process arrives and needs memory, it is allocated the **first available partition** that can accommodate it.

* hardware requirement: base/relocation register, limit register
* physical address = logical address + base register
* base register loaded by OS when it switches to a process
* **Simple to Implement:** The memory management is straightforward. The OS just needs to keep track of which partitions are free and which are allocated.

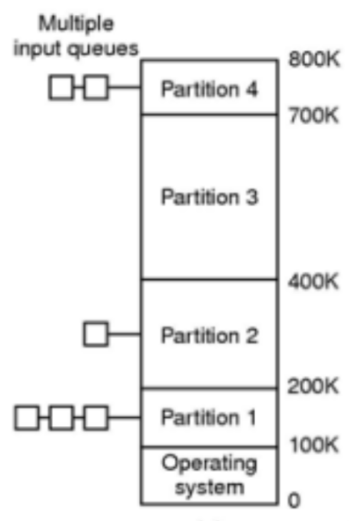
All partitions are of equal size, small processes waste space, large processes don’t fit in. (Solution: Fixed Unequal Partitions)

**Disadvantages:**

* **Internal Fragmentation:** This is a significant issue. Processes rarely fit the partition size exactly. Any unused space within a partition allocated to a process is wasted and cannot be used by other processes. If partition sizes are large, internal fragmentation can be substantial.
* **Limitation on Process Size:** Processes larger than the fixed partition size cannot be loaded. This severely restricts the types of applications that can run.
* **Fixed Degree of Multiprogramming:** The number of partitions directly limits the number of processes that can reside in memory simultaneously.

**1.2 Fixed Unequal Partitions:**

When a process arrives, the OS tries to allocate it to the **smallest available partition** (**Best-Fit**) that is large enough to accommodate it. If no partition is large enough, the process must wait.

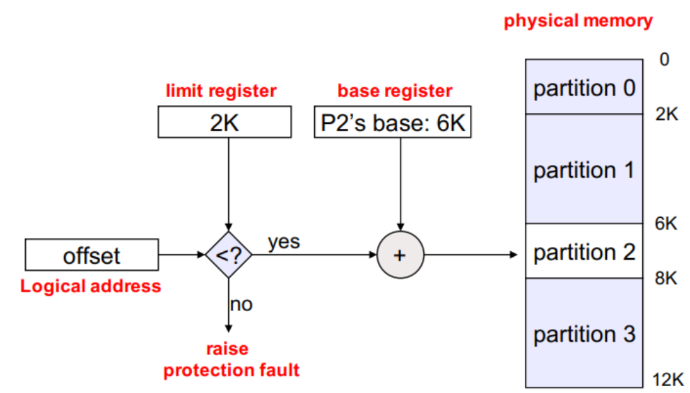
* ****Physical memory is broken into fixed but unequal partitions
* Multiple Queues: Place process in queue for smallest partition that fits in.
* **Improved Memory Utilization (compared to equal partitions):** By having partitions of different sizes, the system can potentially accommodate a wider range of process sizes more efficiently.

**Disadvantages:**

* **Internal Fragmentation:** Still a problem, although potentially less severe than with equal-sized partitions if the partition sizes are chosen thoughtfully.
* **Fixed Degree of Multiprogramming:** The number of partitions still limits the maximum number of processes in memory.
* **Allocation Strategy Complexity:** The OS needs a strategy (like best-fit among available partitions) to decide which partition to allocate, adding a bit more complexity than the simple first-available approach of equal partitions.

**Mechanism for Fixed Partitions (Equal and Unequal)**

* Logical address is generated by the CPU.
* Suppose Process 2 resides in partition 2.
* Limit register for process 2 is 2k and Base = 6k.
* Offset is displacement inside the user process. E.g., 1,2,3,4,..upto max 2K for Process 2 in the below example.

****

**Variable/Dynamic Partition Allocation**

**A screenshot of a black and white screen

AI-generated content may be incorrect.**

Free holes next to each other combine to form a large block.

**A diagram of a system

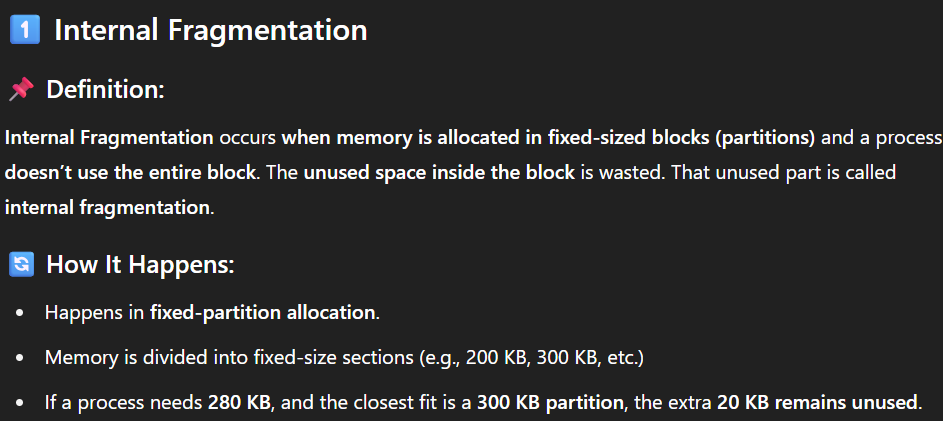
AI-generated content may be incorrect.**

**Fragmentation**

Fragmentation in memory management refers to the **waste of memory space** that occurs during the process of allocating memory to programs or processes. Even though there may be enough total free memory, it may not be usable efficiently due to how it is divided or left scattered.

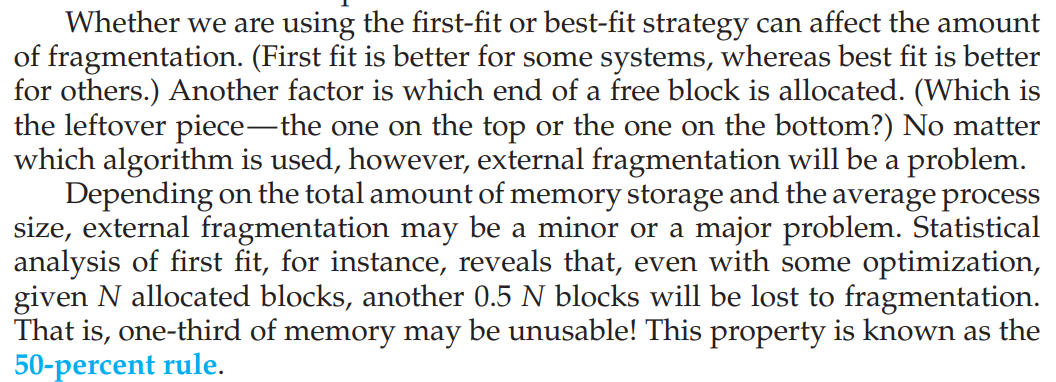
***Why Fragmentation Happens?***

When memory is allocated and deallocated repeatedly, the free memory gets broken into small pieces. This can lead to situations where a process cannot be allocated memory, even though there is enough memory in total, because the available memory is not contiguous.

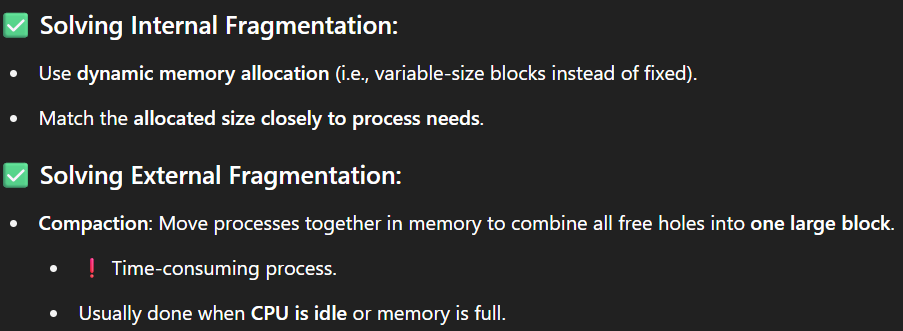


A black screen with white text

AI-generated content may be incorrect.



**A screen shot of a screen

AI-generated content may be incorrect.Compaction 🡪**

A process connot be moved while its performing an I/O job.

**Solution 1: Latch Job in Memory:** One way to address this is to **"latch"** (or pin) the memory of a process that is actively involved in I/O. This means that the memory region of such a process is marked as non-movable and will not be part of the compaction process until the I/O operation is complete. This preserves the integrity of the data transfer.

**Solution 2: Do I/O Only into OS Buffers:** A more robust approach is to ensure that all I/O operations are performed through operating system buffers. When a process wants to read data from a device, the data is first transferred to a kernel buffer. The OS then copies the data from the kernel buffer to the process's memory. Similarly, for writing, the data is first copied from the process's memory to a kernel buffer, and then the OS initiates the transfer from the kernel buffer to the device. This indirection allows the process's memory to be moved during compaction, as the actual I/O operation is happening to stable kernel buffers. However, this introduces overhead due to the extra data copy.

**Segmentation** and **Paging** are the primary and most effective solutions to the problems caused by **external fragmentation** in memory management.

**Non-Contiguous Memory Allocation**: The key innovation of both segmentation and paging is that they permit the **logical address space** of each process to be **non-contiguous** **in physical memory**.

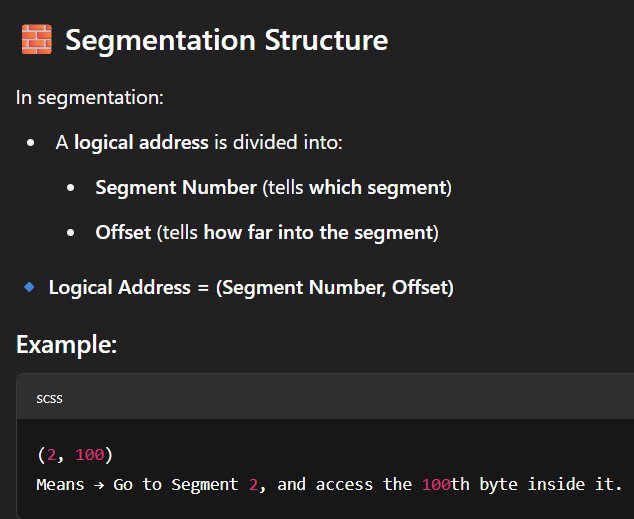
**Segmentation:**

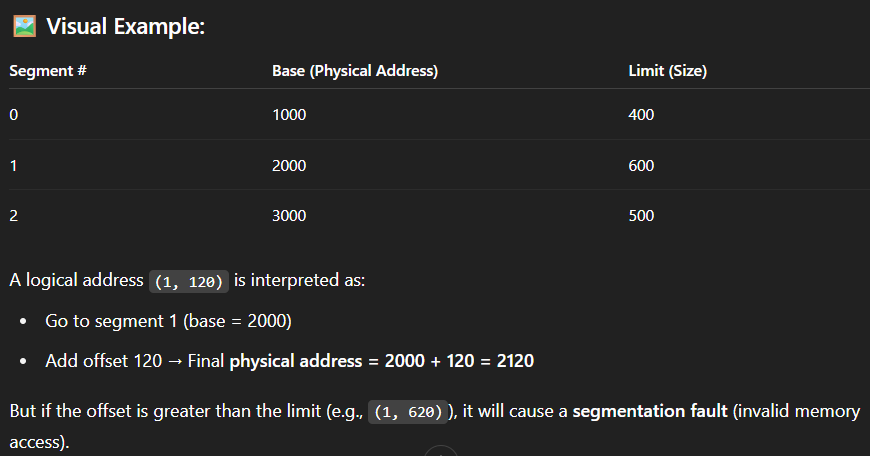
* **Logical Units:** Segmentation divides a process's logical address space into logical units called segments (e.g., code segment, data segment, stack segment). These **variable-sized** segments are typically contiguous in the logical address space but can be placed in non-contiguous physical memory.
* **Segment Table:** The operating system maintains a **segment table** for each process. This table maps each logical segment to its corresponding physical memory location (base address) and stores base and limit registers for each segment.
* **Allocation:** When a segment needs to be loaded into memory, the OS finds a contiguous block of physical memory large enough for that specific segment. These physical blocks for different segments of the same process do not need to be adjacent.
* **Address Translation:** The logical address generated by the CPU consists of a **segment number** and an **offset** within that segment. The **MMU** uses the segment table to find the physical base address of the segment and adds the offset to it to get the physical address.
  + Code, Data, Stack, Functions, Arrays, Object, Local & global variables, Common block, Symbol table. Each segment has a name (or number) and a length.
* **Impact on External Fragmentation:** Segmentation reduces external fragmentation compared to pure contiguous allocation because the OS is looking for smaller, contiguous blocks for individual segments rather than one large block for the entire process. While external fragmentation can still occur (as free memory might be broken into blocks smaller than some segments), it's generally less severe. However, segmentation can still suffer from external fragmentation if segments are of **varying and arbitrary sizes**.

***Why Segmentation?***

Programs naturally consist of different parts that don't need to be stored together in physical memory. Instead of treating the whole program as one big block, segmentation allows the operating system to handle each logical part (segment) separately

A black background with white text

AI-generated content may be incorrect.



A screenshot of a computer

AI-generated content may be incorrect.

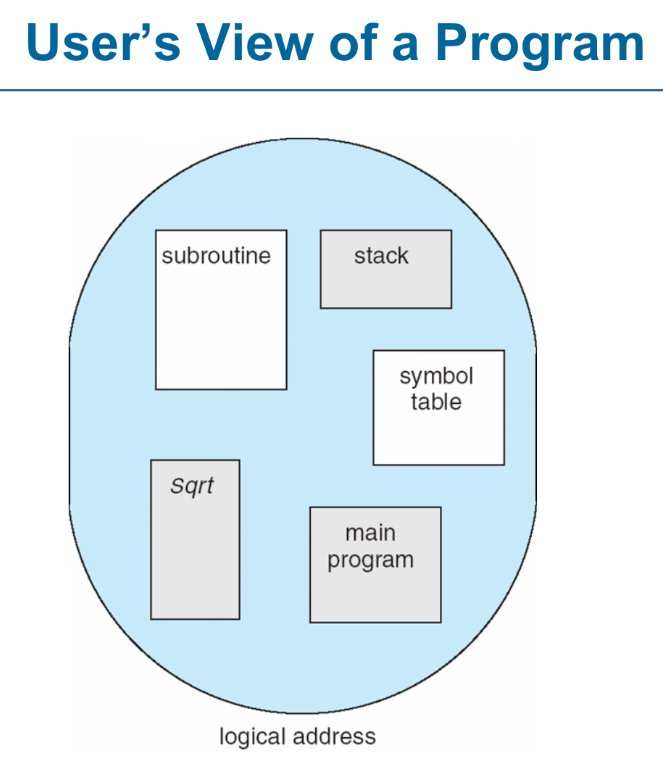
A diagram of a computer program

AI-generated content may be incorrect.

**S =** segment number (s < STLR)

**D =** offset (d < limit)

* **Segment-Table Base Register (STBR):** This is a hardware register that holds the **starting physical address** of the **segment table** in main memory for the **currently running process**. Each process has its own segment table (or a set of segment tables). When the CPU generates a logical address that includes a segment number, the MMU uses the STBR to locate the correct segment table in physical memory. **segment number (s)** must be **greater than or equal** to the value in **STBR.**
* **Segment-Table Length Register (STLR):** This hardware register stores the **length** of the segment table for the **current process**, typically indicating the **number of segments** the process is using. This is crucial for **memory protection**. Before accessing a segment, the MMU checks if the **segment number (s)** provided in the logical address is **less than the value in the STLR**. If s >= STLR, it indicates an attempt to access a segment that the process doesn't own or that is outside its allocated range, resulting in a **segmentation fault** (a type of memory access violation).

 A diagram of a diagram of a diagram

AI-generated content may be incorrect.

**Paging:**

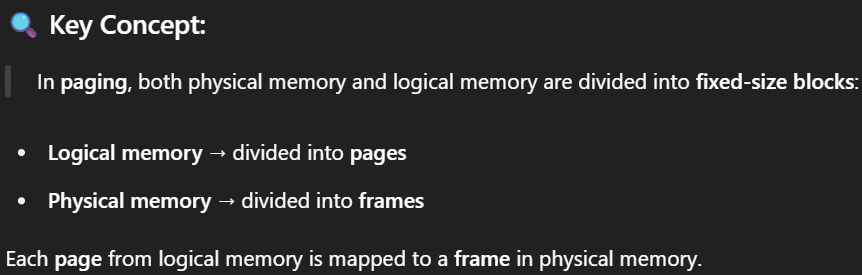
* **Fixed-Size Units:** Paging takes a different approach by dividing both the logical address space of a process and the physical memory into fixed-size units called **pages** and **frames** (page frames), respectively. the **page size and the frame size are always equal.**
* **Page Table:** Each process has a **page table** that maps its logical pages to physical frames. A single process can have its pages scattered across multiple non-contiguous frames in physical memory. Its logical space is contiguous.
* **Allocation:** When a page of a process needs to be loaded into memory, the OS finds any available physical frame and loads the page into it. The page table is updated to record this mapping.
* **Address Translation:** The logical address consists of a page number and an offset within the page. The MMU uses the page number to index into the page table, which provides the corresponding physical frame base address. The base address of the frame is then combined with the offset to generate the physical address.
* **Impact on External Fragmentation:** Paging is highly effective at eliminating external fragmentation (theoretically). Since physical memory is divided into small, fixed-size frames, any free frame can be used to hold any page of any process. There's no issue of finding a contiguous block of a specific arbitrary size. The main form of fragmentation in paging is **internal fragmentation**, which occurs when a process doesn't use the entire last page allocated to it. However, since page sizes are typically small (e.g., 4KB), this internal fragmentation is usually minimal compared to the external fragmentation that can occur with **contiguous allocation or segmentation**.

***Idea behind Paging***

**a**. If we have only two small non-contiguous free holes in the memory, say 1KB each.

**b**. If OS wants to allocate RAM to a process of 2KB, in contiguous allocation, it is not possible, as we must have contiguous memory space available of 2KB. (**External Fragmentation**)

**c**. What if we divide the process into 1KB-1KB blocks?

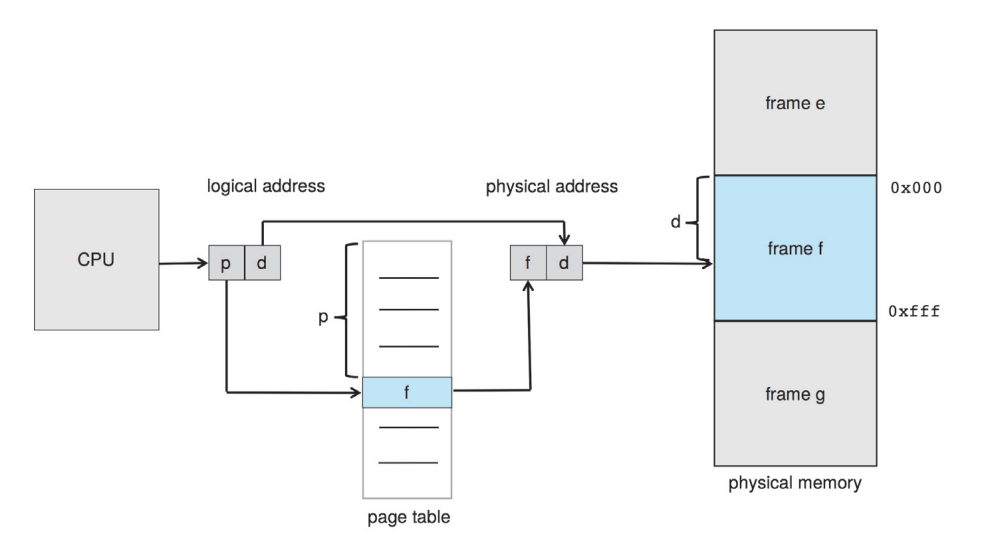


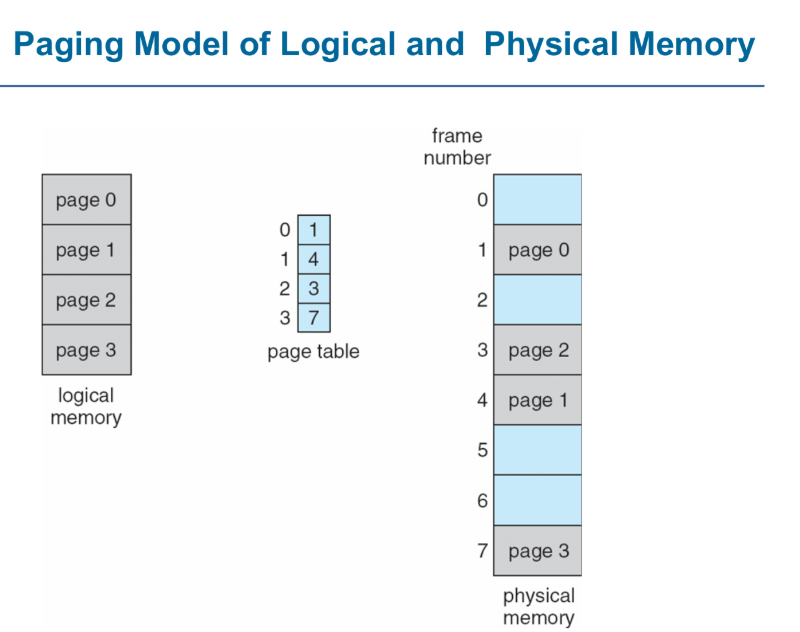
A screenshot of a computer

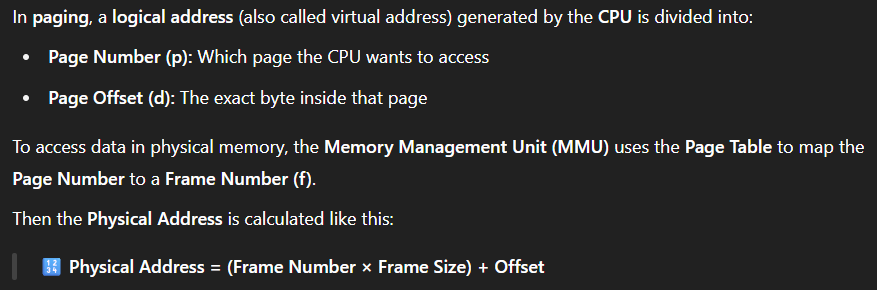
AI-generated content may be incorrect.

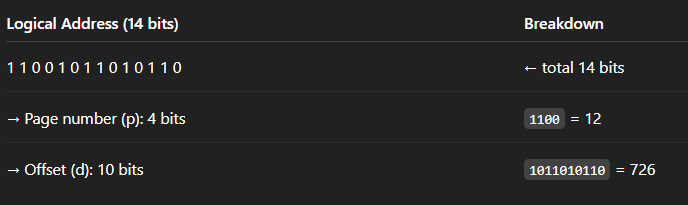
A black background with white text

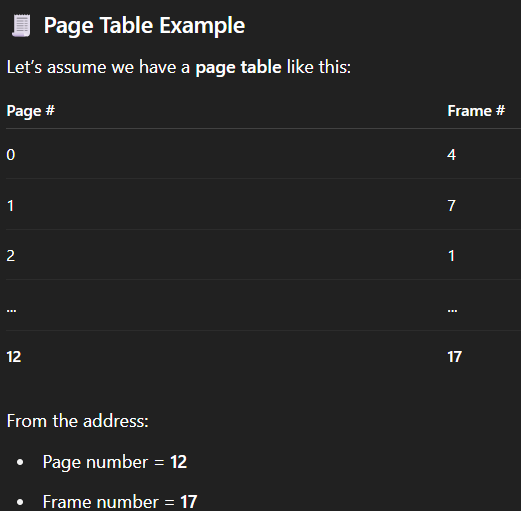
AI-generated content may be incorrect.





**Address Calculation in Paging:**



 A black background with white text

AI-generated content may be incorrect.

A black background with white text

AI-generated content may be incorrect.A screenshot of a computer

AI-generated content may be incorrect.A screenshot of a computer

AI-generated content may be incorrect.

**Practice Questions:**

***Q1*: Suppose that we have free segments of sizes: 6, 17, 25, 14, and 19. Place a program with size 13kB in the free segment using first-fit, best-fit and worst fit.**

**Answer:  
1 First Fit Strategy:**In first fit we scan the memory given and find the very first from the beginning which is large enough to fit the program:  
So from the given memory sizes, **6KB** is too **small**, but right after it **17KB** is **large** enough to **fit** the program with **13KB**. So, the program is successfully placed in **17KB** memory segment

Internal Fragmentation = 17KB-13KB = 4KB.

**2 Best Fit Strategy:**

In best fit, we search all free segments and find the **smallest** one that is still big enough to hold the program,

So, from the given segment sizes we can see that the **smallest** yet **enough** to hold the program is of size **14KB.**

Internal Fragmentation = 14KB-13KB = 1KB. (least leftover)

**3. Worst Fit Strategy:**

In worst fit, we look for the **largest** **segment** available in our entire list of segment size, So in the given list we can see that **25** is the largest segment, so we will choose it to hold our program with **13KB** of size:  
Internal Fragmentation = 25KB-13KB = 12KB.

A black screen with white text

AI-generated content may be incorrect.

A black screen with white text

AI-generated content may be incorrect.

A screenshot of a computer program

AI-generated content may be incorrect.

A screenshot of a computer

AI-generated content may be incorrect.

***Q2:* Consider a user program of logical address of size 6 pages and page size is 4 bytes. The physical address contains 300 frames. The user program consists of 22 instructions a, b, c, . . . u, v. Each instruction takes 1byte. Assume at that time the free frames are 7, 26, 52, 20, 55, 6, 18, 21, 70, and 90. Find the following?**

**A) Draw the logical and physical maps and page tables?**

**B) Allocate each page in the corresponding frame?**

**C) Find the physical addresses for the instructions m, d, v, r?**

**D) Calculate the fragmentation if it exists.**

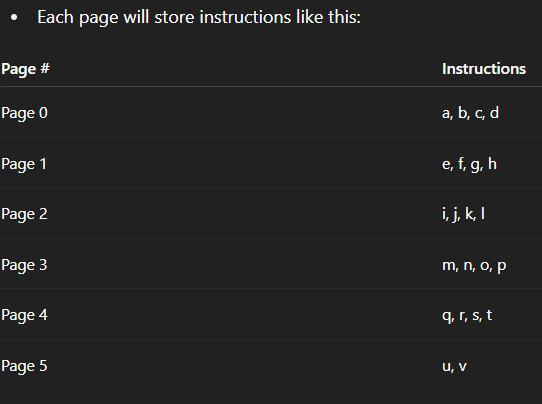
**Answer:**

**A) Draw the Logical Map, Physical Map, and Page Table**

**Given:**

* Logical address space (program) = **6 pages**
* Page size = **4 bytes**
* Physical memory = **300 frames**
* Instructions: **a to v** = 22 instructions  
  (Each instruction = 1 byte)
* Free physical frames available =  
  [7, 26, 52, 20, 55, 6, 18, 21, 70, 90]  
  (We only need 6 frames for 6 pages)

**Step 1: Logical Memory Map**

* Total instructions = 22
* Page size = 4 bytes (each page holds 4 instructions)
* So, number of pages = **ceil(22/4)** = 6 Pages
* 

**Step 2: Page Table**

Assign **the first 6 free frames** from the list:

Free frames: [7, 26, 52, 20, 55, 6]

A black background with white lines

AI-generated content may be incorrect.

**Step 3: Physical Memory Map**

Each frame can hold 4 instructions, so:

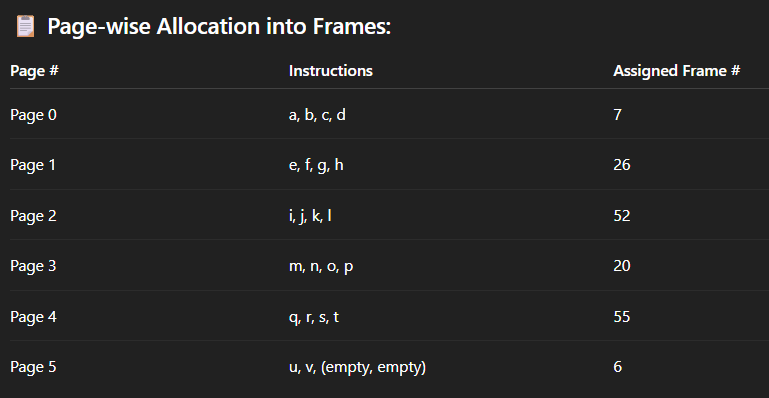
A screenshot of a black screen

AI-generated content may be incorrect.

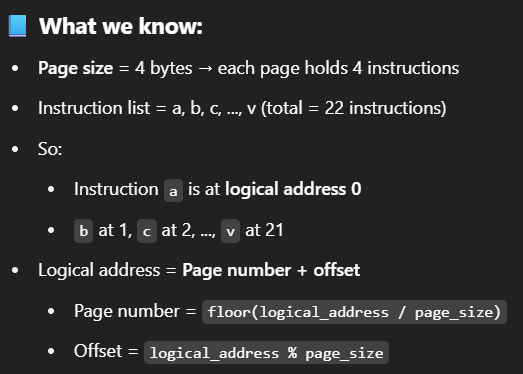
**B) Allocate each page in the corresponding frame**

In this part, we’re simply **mapping each logical page to a physical frame**, and placing the program’s **instructions (a to v)** into these frames.

We have already determined:

* Each **page** holds **4 bytes** (i.e., 4 instructions).
* The program has 22 instructions (a to v).
* So we need 6 pages.
* From the list of free frames, we are using these **first 6**: Free frames = [7, 26, 52, 20, 55, 6]
* 

**C) Find the Physical Addresses for Instructions m, d, v, r**



**Page Table:**

A screenshot of a black screen

AI-generated content may be incorrect.

**A black background with white text

AI-generated content may be incorrect.🔎 Now let’s find for each instruction:**

**A black background with white text

AI-generated content may be incorrect.**A black background with white text

AI-generated content may be incorrect.

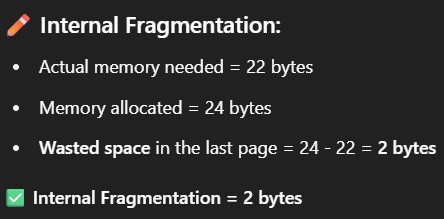
A black background with white text

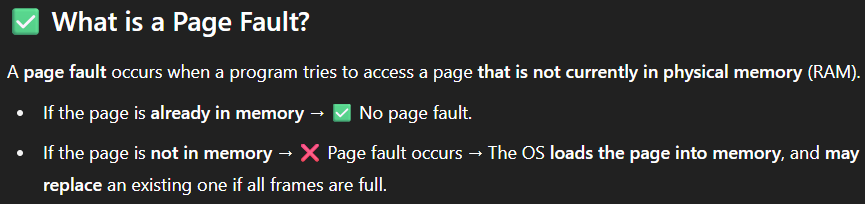
AI-generated content may be incorrect.

**A screenshot of a computer

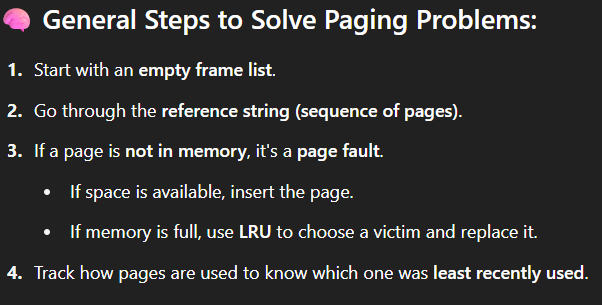
AI-generated content may be incorrect.**

**D) Calculate the Fragmentation**

****No External Fragmentation Exists (because its not possible in paging).







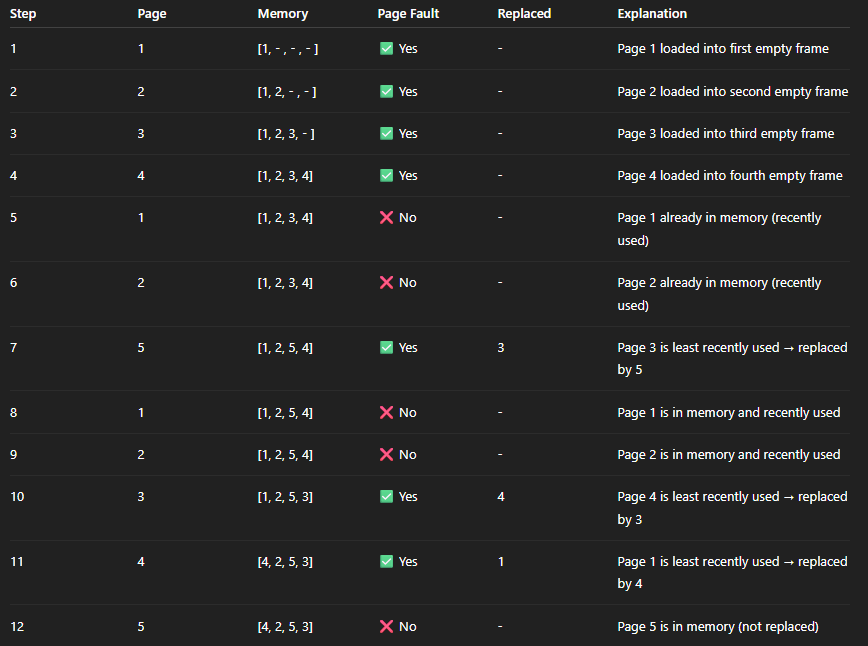
**Q3: Consider a computer system with a physical memory consisting of 4 frames, initially empty. The system uses the LRU (Least Recently Used) algorithm for page replacement. The following page reference string is given: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5. Tasks: 1. Use the LRU algorithm to show the order in which pages are replaced and calculate the total number of page faults.**

**Answer:**

**Given:**

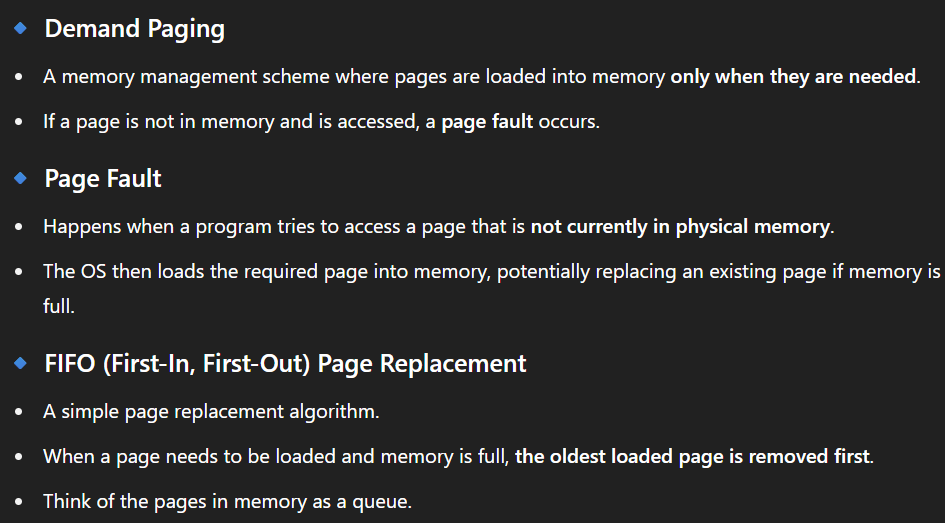
* Total **frames** = 4
* **Page reference string**:  
  1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

**LRU Algorithm**



**Total Page Faults:** 7

**Pages Replaced:** 3 (Step 7), 4 (Step 10), 1 (Step 11)



**Q4: Consider a computer system that uses demand paging with a fixed-size page table. The system has a physical memory of 32 frames, and each frame can hold one page. The system also has a virtual memory of 128 pages. The system uses a FIFO page replacement algorithm. Initially, the page table is empty, and the system is running a program that accesses the following pages in sequence: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 16, 17, 18, 19. Assume that each page access incurs a page fault if the page is not already in memory. Calculate the total number of page faults that occur during the execution of this program and illustrate the contents of the page table and the frames at each step of the process.**

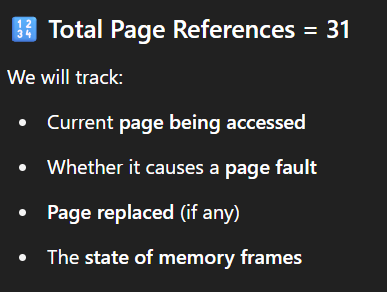
**Answer:**

**Given Parameters:**

* **Physical memory = 32 frames**
* **Virtual memory = 128 pages**
* **Page reference string** (sequence of page accesses):

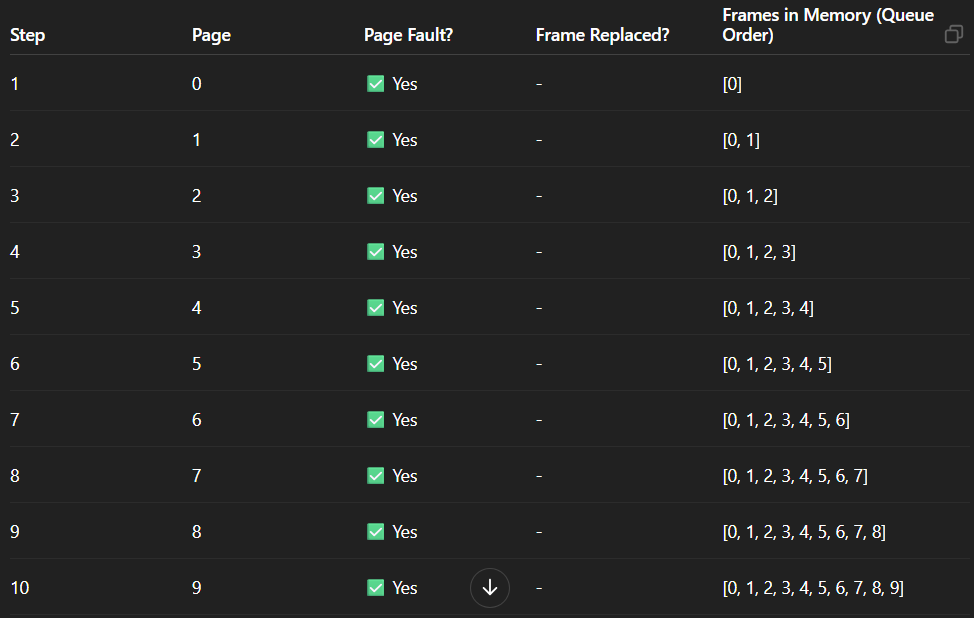
0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15,

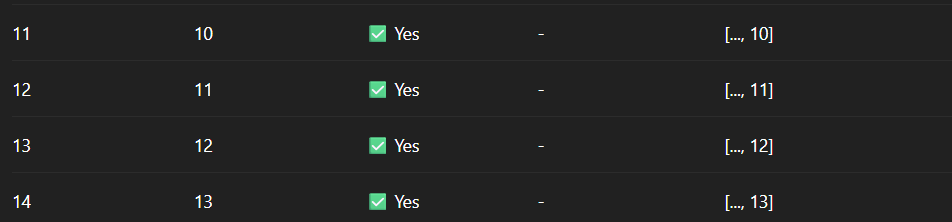
0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 16, 17, 18, 19



A black screen with white text

AI-generated content may be incorrect.



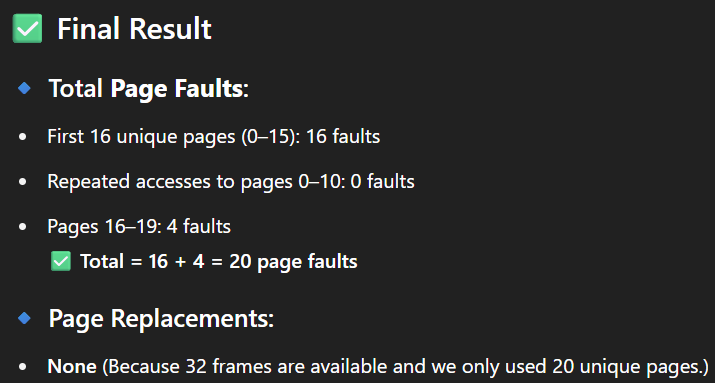


A screenshot of a computer

AI-generated content may be incorrect.

A screenshot of a computer

AI-generated content may be incorrect.



**Associative Memory/Content-Addressable Memory (CAM)**

Unlike main memory (RAM) where you provide an *address* to retrieve data, **associative memory** allows you to perform context-based search where you provide *content* (a piece of data or a "key") and it searches its entire memory in parallel to find a match. If a match is found, it returns the *address* (or addresses) where that content is stored. This parallel search capability makes it incredibly fast for lookup operations.

**Main Memory (RAM) vs. Associative Memory:**

* **RAM:** Address-based access. You tell it "give me the data at address X." It's cheaper and has higher density, making it suitable for large general-purpose storage.
* **Associative Memory (CAM):** Content-based access. You tell it "find me where data Y is stored." It's significantly more expensive per bit due to the additional comparison circuitry in each cell, but much faster for searches.

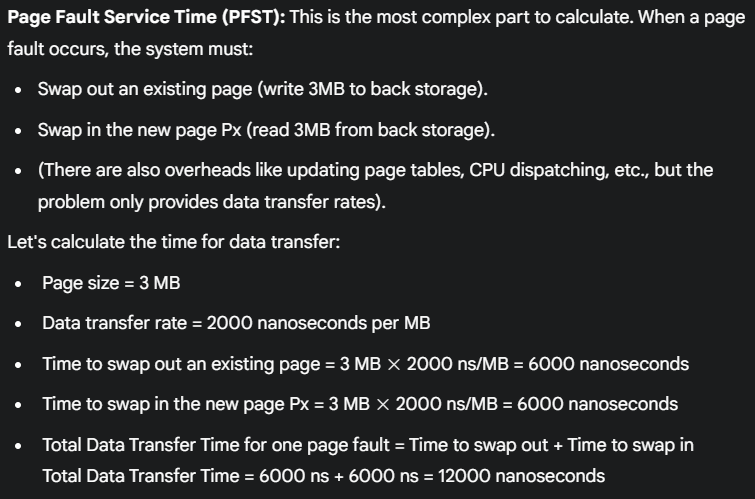
**Translation Lookaside Buffers (TLBs)**

This is its most common and vital application in memory management. In virtual memory systems, the TLB is a small, fast cache that stores recent virtual-to-physical address translations. When the CPU needs to translate a virtual address, it first checks the TLB. The **virtual page number** acts as the "content" to be searched in the TLB's associative memory. If a match (a TLB hit) is found, the corresponding physical frame number is retrieved very quickly, avoiding a slower lookup in the main memory-based page table.

**Effective Access Time (EAT)**

The EAT formula, when considering page faults, is generally:

**EAT=(1−Page Fault Rate)×Memory Access Time + Page Fault Rate×Page Fault Service Time**



A black background with white text

AI-generated content may be incorrect.

A black screen with white text

AI-generated content may be incorrect.

**For all the general parameters:**