**Doesn’t cover everything, just what I thought was important.**

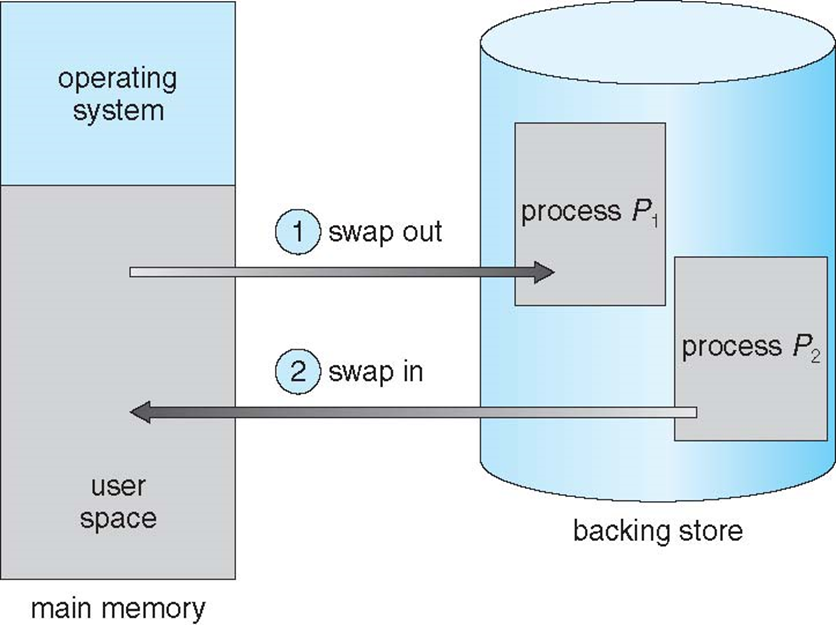
[**https://www.youtube.com/playlist?list=PLxCzCOWd7aiGz9donHRrE9I3Mwn6XdP8p**](https://www.youtube.com/playlist?list=PLxCzCOWd7aiGz9donHRrE9I3Mwn6XdP8p)

**L-5 in the playlist.**

**ALSO LEARN FIRST FIT BEST FIT WORST FIT ALGOS AND SUMS**

Swapping:

Swapping is the act of swapping a process temporarily to a secondary memory from the main memory. This occurs when RAM(main memory) is full, and inactive or lower-priority processes are “swapped out” to storage to allow higher-priority or new processes to execute. When the swapped-out process needs to run again, it is "swapped back" into main memory, replacing another process if necessary. Swapping helps improve multitasking efficiency but can slow down system performance if used excessively.



This diagram illustrates the process of swapping in an operating system:

1. **Swap Out**: When memory is full, a process is selected to be moved out of main memory (RAM) to a secondary storage area called the "backing store." This frees up space in main memory for other processes.
2. **Swap In**: When the swapped-out process is needed again, it is loaded back into main memory from the backing store, potentially replacing another process that can be swapped out.

The **Operating System (OS)** manages this swap process, deciding which processes to swap in and out based on scheduling policies and system load.

The time taken to swap out a process is significant and is proportional to the amount of memory being swapped. It is called context switch time.

## Continuous Memory Allocation

In contiguous memory allocation, each process is contained in a single contiguous section of memory.

Two types:

Fixed Partitioning

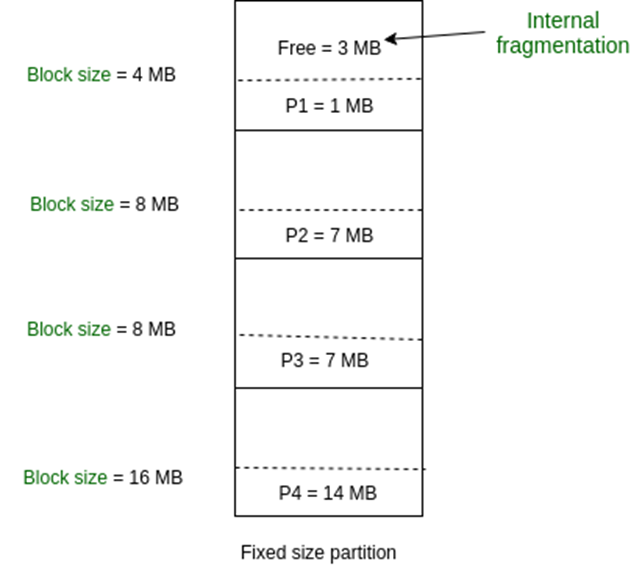
Variable Partitioning.

1. Fixed Partitioning

No of Partitions are fixed

In this partitioning, the number of partitions (non-overlapping) in RAM is fixed but the size of each partition may or may not be the same.

Each partition is dedicated to a specific process until it terminates or releases the partition.

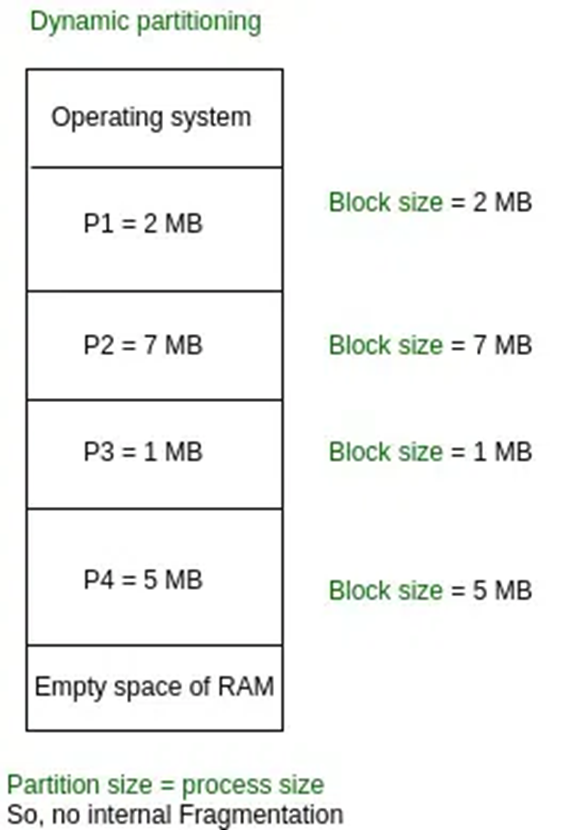


Disadvantage:

It leads to **internal fragmentation** which can leave some space inside a partition unused and wasted when the process’ memory is less than the size of the partition.

2. Variable Partitioning

In **Variable Partitioning**, memory is dynamically divided into partitions based on the size of each incoming process. Unlike fixed partitioning, where partitions are of fixed sizes, variable partitioning adapts to each process's memory needs, allowing for more efficient memory utilization.



It eliminates internal fragmentation as each process is dynamically assigned exactly the space it needs and thus no extra space is present in the partition. However, its disadvantage is that eventually when processes get swapped out of the main memory they leave behind holes. Even though enough memory may be available in total, there is no contiguous block available to accommodate a process. This is called external fragmentation.

Summary:

### **1. Fixed Partitioning**

* **Definition**: Memory is divided into fixed-size partitions, each with a pre-defined size.
* **Process Allocation**: Each partition can hold one process. If a process is smaller than the partition, the unused space remains empty, causing **internal fragmentation** (wasted memory within a partition).
* **Advantages**: Simple to implement and manage.
* **Disadvantages**: Limited flexibility. Large processes might not fit in available partitions even if there’s enough total free memory, leading to inefficient memory use.

### **2. Variable Partitioning**

* **Definition**: Memory is divided into partitions of varying sizes that are dynamically created to fit each process's memory requirements.
* **Process Allocation**: A process is allocated exactly the amount of memory it needs. New partitions are created as processes arrive, and memory is freed when processes exit.
* **Advantages**: Reduces internal fragmentation, as partitions match process sizes.
* **Disadvantages**: Leads to **external fragmentation** (scattered free memory that may be too small to accommodate new processes). Compaction (rearranging memory) may be needed to optimize usage but this takes time.

## Fragmentation

### **1. Internal Fragmentation**

* **Definition**: Internal fragmentation occurs when a fixed-size memory block is allocated to a process, but the process doesn’t use the entire block. The unused memory within the allocated block is wasted.
* **Cause**: Fixed-size allocation (e.g., in fixed partitioning or paging). Since a process might not perfectly match the block size, extra space inside the block goes unused.
* **Example**: If a 100 KB memory block is allocated to a process that needs only 80 KB, 20 KB of memory is wasted within that block.
* **Impact**: Memory is wasted inside allocated partitions or blocks, which reduces the effective memory utilization but doesn’t create gaps between blocks.

### **2. External Fragmentation**

* **Definition**: External fragmentation occurs when free memory is split into small, non-contiguous blocks scattered throughout memory. Even though enough total free memory exists, a process may not be allocated memory because the free blocks aren’t contiguous.
* **Cause**: Dynamic allocation of varying-sized blocks (e.g., in variable partitioning). As processes are loaded and removed, free memory becomes scattered, leaving small gaps between allocated blocks.
* **Example**: Suppose there’s 1000 KB of free memory divided into blocks of 200 KB, 300 KB, and 500 KB, scattered across memory. A new process requiring 600 KB cannot be accommodated, even though there’s a total of 1000 KB free, because no single block is large enough.
* **Impact**: Memory is wasted as free blocks become fragmented, requiring techniques like **compaction** to rearrange memory and create larger contiguous blocks.

## Non contiguous memory allocation

**Non-Contiguous Memory Allocation** is a memory management scheme where a process's memory is divided into multiple blocks that don’t have to be stored in a contiguous (continuous) sequence in physical memory. Instead, the OS can place parts of a process in different memory locations, improving flexibility and utilization of available memory. This method is commonly implemented using techniques like **paging** and **segmentation**.

Paging and Segmentation are the two ways that allow a process’s physical address space to be non-contiguous. It has the advantage of reducing memory wastage but it increases the overheads due to address translation. It slows the execution of the memory because time is consumed in address translation.

**Logical Address**:

* **Definition**: The address generated by the CPU during program execution, also called the virtual address.
* **Usage**: Part of a process's virtual memory space; translated by the OS to a physical address.
* **Translation**: Converted to a physical address by the **Memory Management Unit (MMU)**.

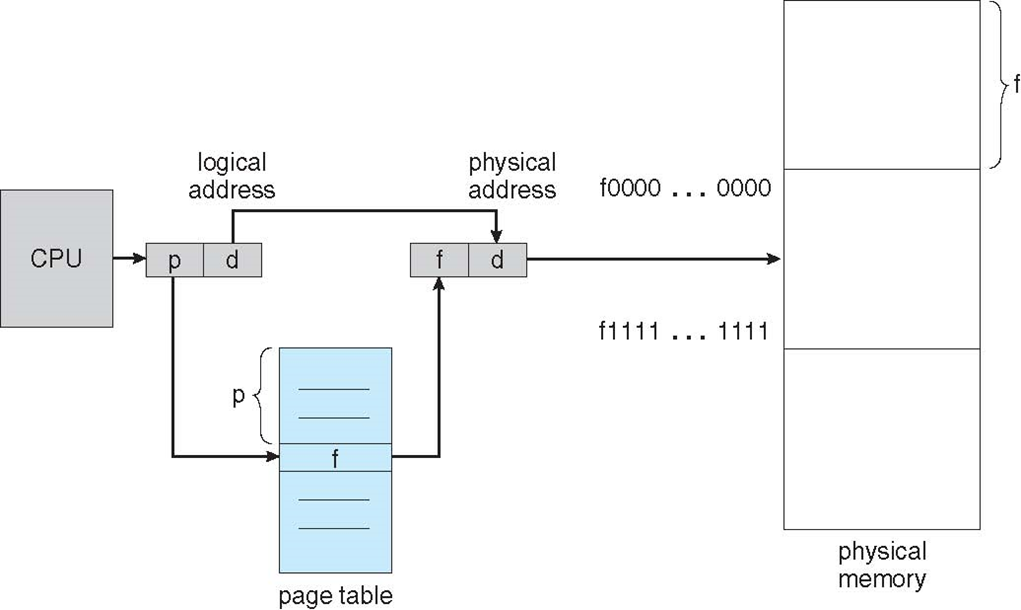
**Physical Address**:

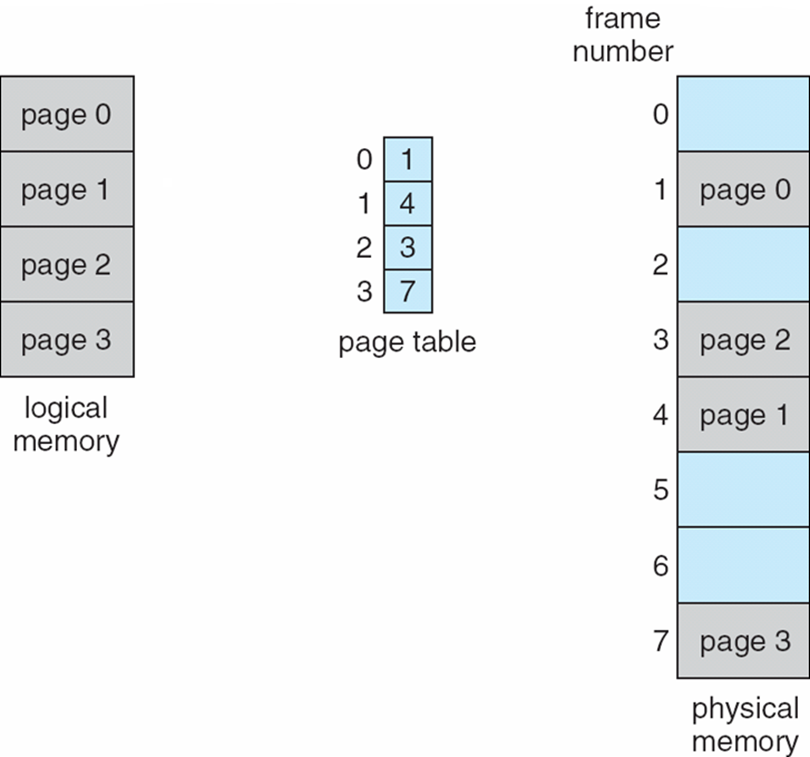
* **Definition**: The actual address in main memory (RAM) used by the hardware to access data.
* **Usage**: Directly accessed by the memory unit to retrieve data.
* **Translation**: Result of translating a logical address, enabling the CPU to access physical memory.

### Paging

Paging is a memory management scheme that eliminates the need for contiguous allocation of physical memory. It divides the physical memory into fixed-size blocks called "frames" (Size is power of 2, between 512 bytes and 16 Mbytes) and divides logical memory into blocks of the same size, called "pages." The operating system maintains a page table to map virtual addresses to physical addresses. When a process accesses memory, the system translates its virtual address to the corresponding physical address using the page table. This allows efficient memory usage and supports virtual memory by enabling processes to use more memory than physically available.

To run a program of size N pages, the system must locate N free frames in physical memory and load the program into those frames. The operating system keeps track of all available free frames. A page table is then set up to map the program's logical addresses to the corresponding physical addresses.





#### Address Translation

Address translation in paging involves converting a program's **logical address** to a **physical address** using a page table. The logical address is divided into two parts: the **page number** and the **offset**. The page number is used to index the page table, which contains the physical address of the corresponding memory frame. The offset remains unchanged, as it specifies the exact location within the frame.

The translation process works as follows:

1. Extract the page number from the logical address.
2. Use the page number to look up the corresponding frame in the page table.
3. Combine the frame's physical address with the offset to form the final physical address.

This process allows efficient memory access and supports the use of virtual memory.

Address generated by CPU is divided into:

Page number (p) – used as an index into a page table

The Page table contains the base address of each page in physical memory

Page offset (d) – combined with base address to define the physical memory address that is sent to the memory unit

**Translation Lookaside Buffer (TLB)**

A **Translation Lookaside Buffer (TLB)** is a small, high-speed cache used to store recent translations of logical addresses to physical addresses. When a logical address is translated, the system first checks the TLB for the corresponding physical address. If the translation is found (a **TLB hit**), the physical address is used directly, speeding up memory access. If the translation is not found (a **TLB miss**), the system accesses the page table to retrieve the physical address and then updates the TLB with the new translation for future reference.

The TLB reduces the time spent on address translation by avoiding the need to access the page table for every memory reference, improving performance.

#### Hardware Implementation of Page Tables

In hardware implementation of page tables, several approaches are used depending on the size of the page table and system constraints.

1. **Registers for Small Page Tables**: For small page tables (e.g., with 256 entries), the page table can be implemented using dedicated registers with high-speed logic for efficient address translation. This approach is feasible because the small size allows fast access, but it is limited in scalability.
2. **Page Table in Main Memory**: For larger page tables (e.g., 1 million entries), it becomes impractical to store them in registers. In these cases, the page table is stored in **main memory**. The **Page Table Base Register (PTBR)** holds the starting address of the page table in memory, while the **Page Table Length Register (PTLR)** indicates the table's size.
3. **Memory Accesses**: With page tables in main memory, every data or instruction access requires two memory accesses: one to retrieve the page table entry and another to access the actual data or instruction. This introduces overhead and can slow down performance.
4. **Translation Lookaside Buffers (TLB)**: To mitigate the performance cost of these two memory accesses, **TLBs** (Translation Lookaside Buffers) are used. TLBs are small, fast caches that store recent page table entries. When a logical address is accessed, the TLB is first checked. If the entry is found (a **TLB hit**), the physical address is quickly obtained. If not (a **TLB miss**), the system looks up the page table in memory, and the TLB is updated with the new entry for future accesses.

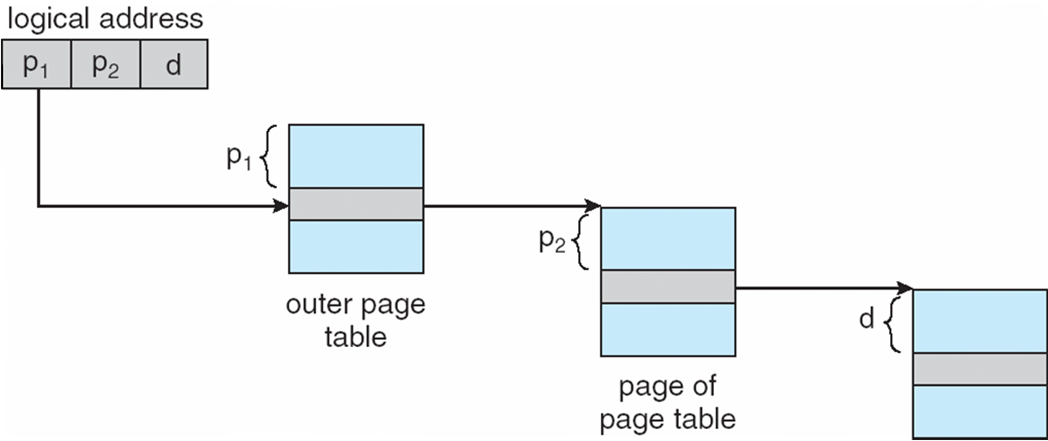
### Types of Paging

To address the problem of large page tables, especially in systems with large address spaces, various methods have been developed to divide the page table into smaller, more manageable pieces. These approaches help improve the efficiency of memory usage and address translation. Three common techniques are **Hierarchical Paging**, **Hashed Page Tables**, and **Inverted Page Tables**.

### **1. Hierarchical Paging:**

Hierarchical paging is a method where the page table is divided into multiple levels. Instead of having a single large page table, this approach breaks it down into smaller tables organized in a hierarchy. Each level of the hierarchy handles a different portion of the address space.

* **Multi-level Page Tables**: For example, in a two-level page table, the first level points to second-level tables, and each second-level table maps to physical frames. This reduces the size of each individual table and allows the system to handle larger address spaces without creating a massive, contiguous page table.
* **Benefits**:
  + It minimizes memory waste by allocating only as many page table entries as needed at each level.
  + It helps scale efficiently to larger address spaces.
* **Drawbacks**:
  + While it reduces the size of individual page tables, there is still some overhead involved in multiple lookups (one for each level of the page table).



### **2. Hashed Page Tables:**

A **hashed page table** is a technique used to efficiently manage large virtual address spaces in systems with sparse memory usage. The idea is to use a **hash table** to store the mappings between virtual pages and physical frames, rather than maintaining a large, contiguous page table. This approach is especially useful when the virtual address space is large but only a small portion of it is used, as it reduces memory overhead and improves lookup speed.

#### Structure of a Hashed Page Table

1. **Hash Table**: The virtual page number (VPN) from the logical address is hashed into a specific index of a hash table. The hash table provides a quick way to look up entries, and each index of the hash table holds a **linked list** of entries.
2. **Linked List**: Each entry in the hash table is a linked list node, and multiple logical addresses can hash to the same location. This is called a **collision**. To handle this, the linked list stores multiple elements (if necessary), each containing:
   * **Virtual page number**: The virtual page number that is mapped to a physical frame.
   * **Mapped page frame**: The physical frame number corresponding to the virtual page.
   * **Pointer to the next element**: This pointer links to the next entry in the list, allowing the system to traverse through entries that hash to the same location.

#### Algorithm of Hashed Page Table

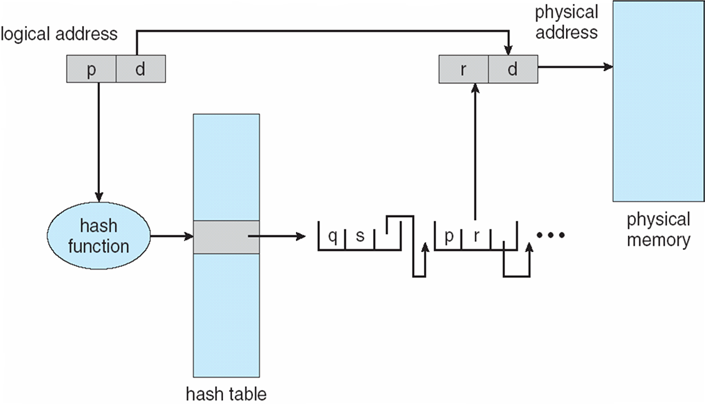
1. **Hashing**: When a program accesses memory, the **virtual page number** (VPN) from the logical address is extracted. A **hash function** is then applied to this VPN, which computes an index in the hash table.
2. **Searching the Linked List**: Once the hash table index is found, the system checks the first element of the linked list at that index.
   * If the **virtual page number** of the first element matches the requested **VPN**, then the **mapped page frame** (the second field) is used to compute the corresponding physical address.
   * If the first entry does not match, the system moves to the **next element** in the linked list and compares the virtual page number with each successive element.
3. **Page Frame Mapping**: Once a match is found, the **physical address** is formed by combining the **mapped page frame** from the matched entry with the **offset** from the logical address. This gives the final physical address for the memory access.
4. **Collision Handling**: If there is no match in the linked list, the system either returns an error (indicating that the page is not in memory) or loads the page from disk, updating the hash table with the new mapping. If necessary, the new entry is appended to the linked list.

#### Benefits of Hashed Page Tables

* **Efficient Lookup**: Hashing enables quick lookup of page mappings, especially when the virtual address space is sparsely populated.
* **Memory Savings**: The size of the hash table can be dynamically adjusted based on the number of entries, reducing memory overhead compared to large, continuous page tables.
* **Collision Handling**: Linked lists provide a way to handle collisions (when multiple virtual pages hash to the same index), ensuring that even with collisions, page lookups remain functional.

#### Drawbacks

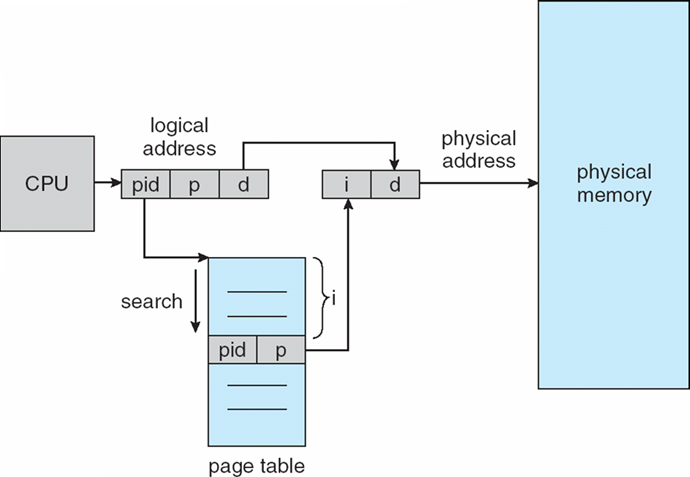
* **Hash Collisions**: Hash collisions may still occur, and handling them through linked lists introduces some extra overhead.
* **Search Time**: If there are many collisions, searching the linked list could take longer than desired, particularly if the linked list becomes long.
* **Complexity**: The hash function and collision resolution methods must be carefully designed to minimize search times and ensure efficient memory usage.



### **3. Inverted Page Tables:**

Inverted page tables are a different approach where, instead of having one entry for each page in the virtual address space, the page table has one entry for each frame in the physical memory. This design is essentially the inverse of traditional page tables.

* **Structure**: Each entry in the inverted page table maps a physical frame to the virtual address space that is currently occupying that frame. The key difference from traditional paging is that it is indexed by the physical frame rather than the virtual page.
* **Benefits**:
  + It reduces the size of the page table because the number of entries is equal to the number of physical frames, not the number of virtual pages. For example, if there are 1 million physical frames, the page table will have only 1 million entries, regardless of the virtual address space size.
* **Drawbacks**:
  + To translate a logical address to a physical address, the system must search the inverted page table to find the mapping for the given virtual page, which can be slower than direct indexing in traditional page tables.
  + This search is often done using a hash table to speed up lookups, but it still introduces some overhead.
  + The inverted page table requires additional processing when a new page is loaded, as it may need to search for a free physical frame and update the table.



### **Summary:**

* **Hierarchical Paging**: Breaks the page table into multiple levels to manage large address spaces efficiently, reducing memory overhead.
* **Hashed Page Tables**: Uses hashing to map logical pages to physical frames, improving look-up efficiency, especially in sparse address spaces.
* **Inverted Page Tables**: Maps physical frames to virtual pages, reducing the page table size, but requiring additional search mechanisms for address translation.

Each of these techniques aims to manage memory efficiently in systems with large address spaces, each with its own trade-offs in terms of complexity, lookup time, and memory usage.

### Segmentation

**Segmentation** is a memory management scheme that divides a program's memory into **segments**, each of which is a logical unit such as a function, array, stack, or data structure. Unlike paging, which divides memory into fixed-size blocks (pages), segmentation divides memory into variable-sized segments that reflect the logical structure of a program.

### **Key Concepts of Segmentation:**

1. **Segments**: In segmentation, the memory is divided into segments, where each segment is a continuous block of memory. Segments are typically of different sizes, depending on the program's needs. Common types of segments include:
   * **Code Segment**: Contains executable instructions.
   * **Data Segment**: Stores global and static variables.
   * **Stack Segment**: Used for function call stack frames and local variables.
   * **Heap Segment**: Used for dynamic memory allocation.
2. **Logical Address**: A **logical address** (or **virtual address**) in a segmented system consists of two parts:
   * **Segment Number**: Identifies which segment of the program the address refers to.
   * **Offset**: Specifies the position of the data within the segment (similar to the page offset in paging).
3. **Segment Table**: Each process has a **segment table**, which keeps track of the physical memory locations of all segments. The segment table contains:
   * **Base Address**: The starting address of each segment in physical memory.
   * **Limit**: The size of the segment (i.e., the maximum valid offset for that segment).
4. The segment table is indexed by the segment number, and each entry maps a segment to its corresponding physical memory location.

### **Address Translation in Segmentation:**

To translate a logical address into a physical address, the system uses the segment table. The translation process is as follows:

1. **Extract the Segment Number**: The segment number is obtained from the higher part of the logical address.
2. **Lookup the Segment Table**: The segment number is used to look up the segment table, which provides the base address (starting address) of the segment in physical memory and its size (limit).
3. **Calculate the Physical Address**: The physical address is calculated by adding the **offset** (the lower part of the logical address) to the **base address** obtained from the segment table entry.
4. **Check for Bounds**: The offset is checked against the segment's **limit** to ensure that it does not exceed the segment's size. If the offset is out of bounds, a segmentation fault occurs.

### **Benefits of Segmentation:**

* **Logical Structure**: Segmentation reflects the logical structure of a program, which is more intuitive for both the programmer and the operating system. Each segment corresponds to a logical component (e.g., code, data, stack).
* **Dynamic Size**: Segments can vary in size based on the program's needs, unlike fixed-size pages in paging, which may waste space or require extra fragmentation.
* **Flexibility**: It allows easier sharing of memory, as individual segments (like the code segment) can be shared between processes.

### **Drawbacks of Segmentation:**

* **Fragmentation**: Segmentation can lead to **external fragmentation** (unused gaps between segments in physical memory). This can be problematic as the program grows or segments are allocated and deallocated dynamically.
* **Complex Address Translation**: Unlike paging, which uses simple fixed-size blocks, segmentation requires managing multiple sizes of segments and performing additional checks, which can increase complexity.
* **Segment Table Management**: The segment table must track each segment’s base address and limit, which requires more memory and processing.

Differences

| **Aspect** | **Paging** | **Segmentation** |
| --- | --- | --- |
| **Memory Division** | Divides memory into **fixed-size pages** (e.g., 4 KB). | Divides memory into **variable-sized segments** (e.g., code, data, stack). |
| **Page Size / Segment Size** | Fixed-size pages. | Variable-size segments depending on logical structure (e.g., functions, arrays). |
| **Address Translation** | Translates logical address using **page number** and **page offset**. | Translates logical address using **segment number** and **offset**. |
| **Address Structure** | Logical address = **(Page Number, Offset)**. | Logical address = **(Segment Number, Offset)**. |
| **Memory Allocation** | No fragmentation within pages (but can have **internal fragmentation**). | May suffer from **external fragmentation** due to variable-sized segments. |
| **Address Space Representation** | Flat address space, the page table maps virtual pages to physical frames. | Segmented address space, the segment table maps logical segments to physical memory locations. |
| **Page Table / Segment Table** | Uses a **page table** to map pages to frames in physical memory. | Uses a **segment table** to map segments to physical memory locations. |
| **Translation Speed** | Simple, fixed translation (page lookup + offset). | More complex, as it involves segment lookup and size validation. |
| **Memory Usage Efficiency** | **Internal fragmentation** (unused space within pages). | **External fragmentation** (unused gaps between segments). |
| **Access to Logical Units** | No clear logical separation between code, data, etc. | Reflects logical program structure (code, data, stack, etc.). |
| **Protection** | Protection is applied at the page level (read/write/execute). | Protection is applied at the segment level (read/write/execute). |
| **Hardware Complexity** | Requires hardware to manage the page table and perform address translation. | Requires hardware for segment table lookup and bounds checking. |

### **Physical vs. Logical Addresses**

| **Aspect** | **Physical Address** | **Logical Address** |
| --- | --- | --- |
| **Definition** | The actual address in **physical memory** (RAM). | The address used by a process in its **virtual address space**. |
| **Used By** | Accessed by the **CPU** to fetch data or instructions from memory. | Used by programs to refer to locations in virtual memory. |
| **Translation** | Translated from logical address by hardware (page table, segment table). | Translated to physical address through page or segment table. |
| **Visibility** | Visible to the operating system and hardware (real address). | Visible only to the program; the OS and hardware handle translation. |
| **Context** | The physical memory of the computer. | The **virtual memory** allocated to a process. |
| **Size** | Fixed size depending on the system’s memory (e.g., 32-bit or 64-bit). | Same size as physical address, but only represents a virtual space for a process. |
| **Role in Memory Management** | Represents the actual memory locations where data resides. | Represents logical view used by processes, which may not match physical memory. |

# 5.2

### **1. Demand Paging**

**Demand paging** is a type of virtual memory system in which pages of a process are loaded into memory only when they are needed, rather than loading the entire process at once. This helps in efficient memory usage.

* **How it works**:
  + When a process is executed, only the initial pages of the process are loaded into memory.
  + If the process tries to access a page that is not currently in memory (a **page fault** occurs), the operating system loads the required page from the disk into RAM.
  + The page is brought into memory "on demand," hence the term **demand paging**.
* **Advantages**:
  + **Efficient memory use**: Only the pages that are actually needed are loaded into memory.
  + **Faster startup**: The program can start running without having to wait for the entire process to be loaded into memory.
* **Disadvantages**:
  + **Page faults**: If the needed pages are not in memory, there is a delay caused by loading pages from the disk, leading to overhead.
  + **Increased latency**: The disk I/O involved in fetching pages can lead to slower performance, especially with frequent page faults.

### **2. Page Replacement**

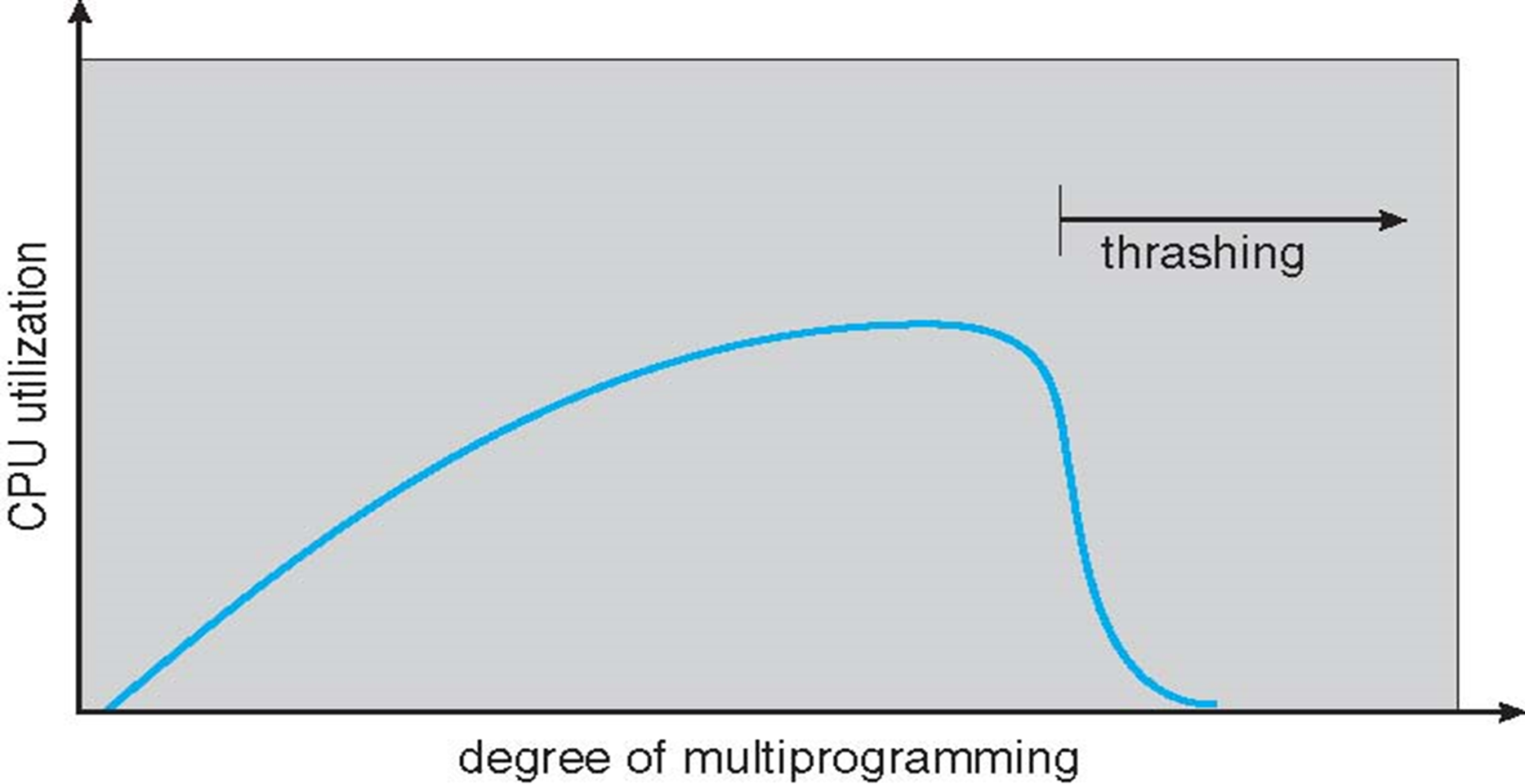
Page replacement refers to the process of managing which pages should be swapped out of memory when the system runs out of space in physical memory, and a new page needs to be loaded. This is necessary when the system has more processes or data than can fit in physical RAM.

* **Page Replacement Algorithms**: These are strategies used by the operating system to decide which page to swap out. Common algorithms include:
  + **FIFO (First-In, First-Out)**: The oldest page in memory is swapped out first.
  + **LRU (Least Recently Used)**: The page that has not been used for the longest period is swapped out.
  + **Optimal**: The page that will not be used for the longest period in the future is swapped out. (This is theoretical as it requires knowledge of future accesses).
* **Page Fault Handling**: When a page fault occurs, the operating system must decide which page to replace. If the page is dirty (modified), it needs to be written back to disk before it’s replaced.

### **3. Thrashing**

**Thrashing** occurs when the operating system spends more time swapping pages in and out of memory than executing the process. This can happen when the system runs out of physical memory and continuously performs **page replacement** due to high page fault rates.

* **Causes of Thrashing**:
  + **Excessive page faults**: If processes access more pages than can fit in memory, page faults occur constantly.
  + **Insufficient memory**: If the total memory available is too small for the workload, thrashing is more likely.
* **Symptoms**:
  + The system becomes very slow, as most of the CPU time is spent on swapping pages in and out of memory.
  + The **CPU utilization** decreases because the system is waiting on page swaps.
* **Mitigation**:
  + **Increasing RAM**: More physical memory reduces the need for swapping.
  + **Working Set Model**: Keep track of the set of pages that a process is actively using, and ensure they remain in memory.
  + **Better Page Replacement Algorithms**: Using algorithms like **LRU** can minimize the number of page faults and reduce thrashing.



### **4. Allocating Kernel Memory**

**Kernel memory allocation** refers to the management of memory that is reserved for the operating system (OS) kernel and its operations.

* **How Kernel Memory is Allocated**:
  + **Fixed Size**: The OS may allocate kernel memory in a fixed-size manner, with pre-defined sizes for kernel data structures and buffers.
  + **Dynamic Allocation**: For more flexibility, kernel memory can be allocated dynamically to respond to changing needs during runtime (e.g., for device drivers, system calls, or managing buffers).
* **Allocation Strategies**:
  + **Slab Allocation**: This is a method where kernel memory is allocated in fixed-size chunks (slabs), with an associated cache to optimize memory usage. Each type of object in the kernel (e.g., a file descriptor) has its own cache to reduce fragmentation.
  + **Buddy System**: This is a memory allocation scheme where the memory is divided into blocks of various sizes, and pairs of blocks (buddies) can be merged or split as needed.
  + **Page-Based Allocation**: The kernel can allocate memory in page-sized blocks, similar to how user memory is allocated with paging.
* **Why Kernel Memory is Special**:
  + Kernel memory is **not swapped** to disk because it contains critical components of the operating system, like device drivers, system buffers, and process management structures. Losing kernel memory would lead to system crashes.
  + Kernel memory must be allocated efficiently to avoid running out of memory and causing system instability.

### 

| **Feature** | **Buddy System** | **Slab Allocation** |
| --- | --- | --- |
| **Memory Divisions** | Divides memory into blocks of powers of two. | Divides memory into slabs for specific objects. |
| **Allocation Unit** | Allocates memory in blocks that are powers of two. | Allocates memory for fixed-size objects. |
| **Fragmentation** | Can cause internal fragmentation. | Minimizes both internal and external fragmentation. |
| **Efficiency** | Simple but can be inefficient for small objects. | More efficient for managing fixed-size objects. |
| **Use Cases** | General-purpose memory allocation. | Object caches and kernel memory management. |
| **Complexity** | Simpler, with fewer metadata overheads. | More complex due to slab management and object caching. |