CHAPTER 4

Memory Management

V.1. Introduction

Memory is the second most important resource after the processor in a computer. Although there are several memory elements like registers, cache, Read Only Memory (ROM), Random Access Memory (RAM), and secondary memory (hard disk), in a computer, **RAM** is known as the **main memory**.

Although registers and cache memory are closer to the processor that a processor can directly access, they are very expensive and thus of very small capacity. They cannot accommodate either the operating system or other programs that are executed by the processor. Main memory is the furthest memory unit from a processor that it can directly access which can accommodate the OS as well as user applications during their execution.

The OS kernel remains loaded in the low memory region of the memory as long as the system runs. Once loaded, the OS divides the memory into two parts: kernel space for storing the OS; and the user space for storing the application processes. In a single-programming system, the OS remains in the kernel space and only one application program can reside in the user space.

But in today's multiprogramming environment, the OS needs to further divide the user space so that several user programs can coexist in the memory. How many programs can be accommodated in the memory decides the **degree of multiprogramming**.

As opposed to the CPU, which only one process (or thread) owns it at any given time, numerous processes and/or threads may coexist in main memory at any point of time. All these processes must share the main memory. If the mechanisms for managing this sharing are inefficient, the computer will perform poorly, no matter how powerful its processor. On the other hand, the faster the CPU, the faster your system reacts, so it's up to the operating system to organize it in the best possible way to get the maximum performance out of it.

To maximize CPU performance in a multi-programmed system, a number of questions need to be asked concerning :

- How to share data between processes?
- How can memory space be protected for each process?
- How and by whom is memory space allocated to processes?
- How and when is space freed up by a process restored?

To meet these requirements, the OS needs a special memory management module called the **Memory Management Unit (MMU)**.

In this chapter, we'll see that there are several memory management schemes, with their advantages and disadvantages.

V.2. Memory Hierarchy Design and its Characteristics

A memory can be represented as a storage **cabinet** made up of different **drawers**. Each drawer represents a **memory location**, which can contain a single element: **data**. As the number of memory locations can be very large, it is necessary to be able to identify them by a number. This **number** is called an "**address**". Each piece of data can then be accessed via its address (see figure 5.1).

Address	Memory Location
7 = 111	⋯
6 = 110	⋯
5 = 101	⋯
4 = 100	⋯
3 = 011	⋯
2 = 010	⋯
1 = 001	⋯
0 = 000	0001 1010

Figure 5.1 : Memory Organization.

With an **n-bit** address, a maximum of 2^n memory locations can be referenced.

In computer system design, the concept of memory hierarchy is an improvement in the organization of computer memory so as to minimize memory access time. Memory hierarchy was developed based on a software program's behavior known as reference locality. The figure below describes the different levels of the memory hierarchy:

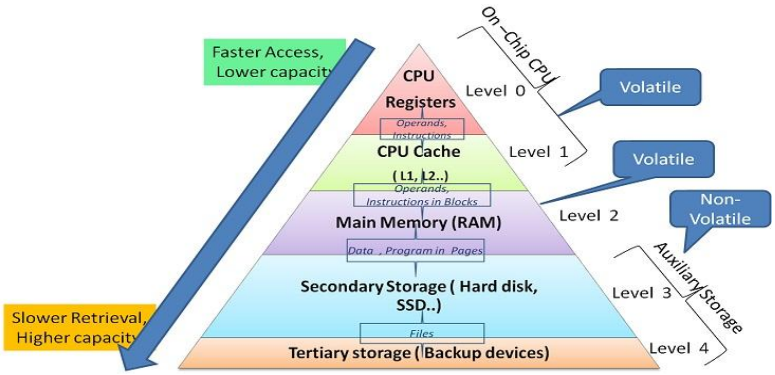


Figure 5.2 : Memory Hierarchy.

Higher levels are expensive but fast. As we move down the hierarchy, cost per bit decreases while access time increases. In addition to the speed and cost of the various storage systems, there is also the problem of storage volatility. Volatile storage loses its contents when the device's power supply is cut off.

Generally speaking, memory contains two main types of information:

- Instructions (processing information: commands): these dictate the processing of tasks in the form of elementary orders or commands,
- Operands (processed information): these are the data on which the processing dictated by the instructions is carried out.

On the other hand, two types of operation can be performed on memory words: Read and Write:

- To read memory: an address must be selected from the address bus and the control bus must trigger the operation. The data at the selected address is transferred to the data bus.
- To write memory: an address must be selected from the address bus and the control bus must initiate the operation. Data at the selected address is replaced by data on the data bus.

V.3. Main Memory Manager Objectives

The Main Memory Manager is a very important OS module whose function is to manage access to the Main Memory. It has the following objectives:

- **Relocation** is the process of **assigning load-time or run-time memory addresses to a program's instructions and data** so that it can be executed from any location in physical memory, rather than a fixed one.
  It allows the operating system to move programs in memory without affecting their execution.
- **Allocation** is the process of **assigning memory space** to programs and data so they can be loaded and executed. It ensures that **each process gets the memory it needs** without overlapping with others.
- **Protection** is the process of **preventing unauthorized access** to a program's or process's memory. It ensures that **one process cannot interfere** with the memory or data of another, maintaining system stability and security.
- **Sharing** is the process of **allowing multiple processes to access the same memory space** safely and efficiently. It enables **resource sharing**, such as shared libraries or data, without compromising protection.
- **Logical Organization** refers to the way **programs and data are structured in memory** using logical segments like code, data, and stack. It helps in **modular programming, protection, and efficient memory use**.
- **Physical Organization** refers to the actual hardware setup and physical components of a computer system or data storage system. It includes the layout and arrangement of devices such as processors, memory units, storage drives, and input/output devices. This organization affects how efficiently data is processed and accessed at the hardware level.

These objectives must be achieved in an efficient way, by increasing the performance of the computer system and transparency for the user.

V.4. Logical Versus Physical Address Space

➔ **Logical address space:** An address generated by the CPU is called a **"logical address"**. It's also known as a **virtual address**. The logical address space can be defined as the **process size**. A logical address can be **modified**.

➔ **Physical address space:** An address seen by the memory unit (i.e. the one loaded into the memory address register) is commonly referred to as a **"physical address"**. A physical address is also called a **real address**. The physical address always remains **constant**.

➔ The set of all physical addresses corresponding to these logical addresses is called the physical address space.

➔ The run-time mapping from virtual to physical addresses is done by a hardware device called the **Memory-Management Unit (MMU)**. A physical address is calculated **MMU**.

At runtime, it is necessary to convert logical addresses into physical addresses. For example, imagine a system where a physical address is obtained by adding the base address contained in a register to each logical address.

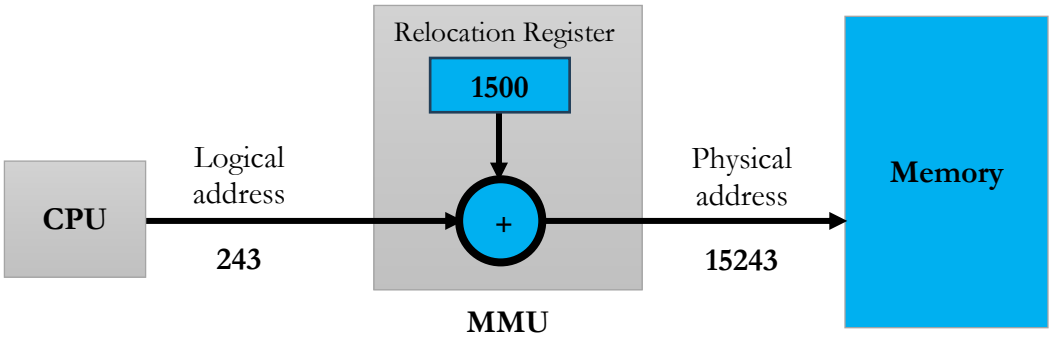


Figure 5.3 : Dynamic Relocation Using a Relocation Register.

The base register is now called a relocation register. The value in the relocation register is added to every address generated by a user process at the time the address is sent to memory (Figure 5.3). For example, if the base is at **15000**, then an attempt by the user to address location **0** is dynamically relocated to location **15000**; an access to location **243** is mapped to location **15346**.

It should be noted that the user program never sees the real physical addresses, it only manipulates logical addresses. As we have two different type of addresses **Logical address** in the range (**0** to **Max**) and **Physical addresses** in the range(**R** to **R + Max**) where **R** is the value of relocation register. The user generates only **logical addresses** and thinks that the process runs in location to **0** to **Max**. As it is clear from the above text that user program supplies only logical addresses, these **logical addresses** must be mapped to **physical address** before they are used.

V.5. Memory Management in Single-Programmed Systems

In a single-programmed system, the memory is subdivided into two contiguous partitions,

- One for the resident operating system, with its various modules (the interrupt vector, ...), often placed in low memory.
- The other reserved for the active user (it is totally at his disposal).

The simplest possible memory management scheme is to run just one program at a time, sharing the memory between that program and the operating system.

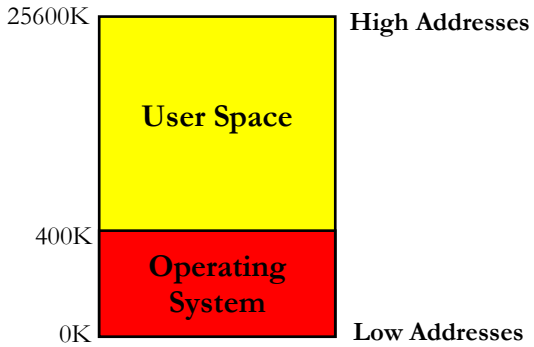


Figure 5.4 : Memory Organization in a Single-Programmed System.

When the system is organized in this way, only one process at a time can be running. As soon as the user types a command, the operating system copies the requested program from disk to memory and executes it. When the process finishes, the operating system displays a prompt character and waits for a new command. When it receives the command, it loads a new program into memory, overwriting the first one.

The allocation algorithm can be described by the flowchart below:

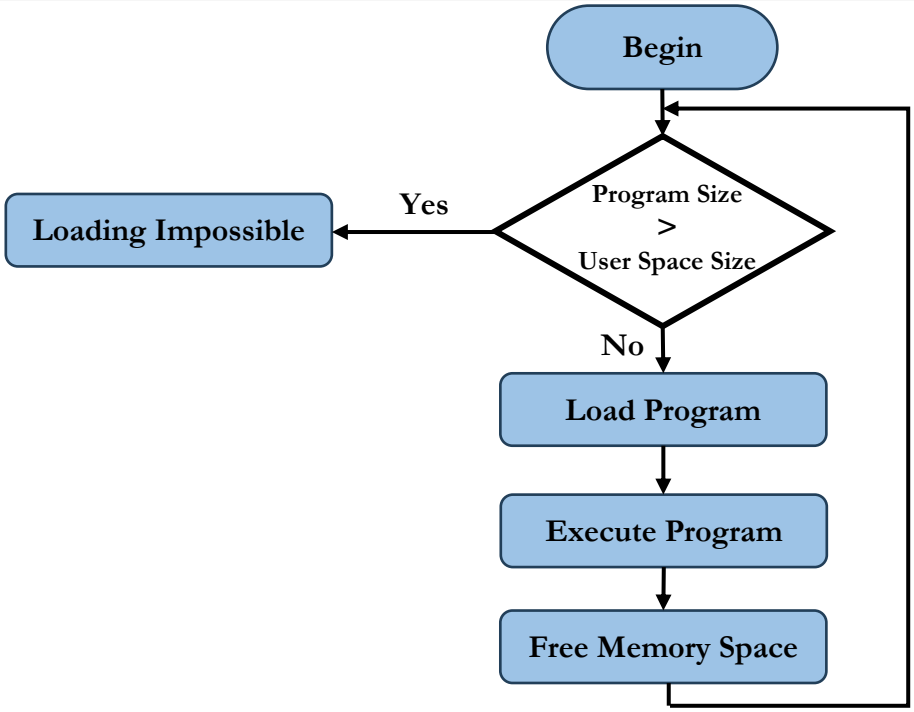


Figure 5.5 : Memory Allocation Algorithm in a Single-Programmed System.

The major disadvantage of this strategy is the under-utilization (i.e. poor use) of memory, in the sense that programs (generally) only partially occupy user space. Furthermore, the size of user programs is limited by the size of the user space.

V.5.1. Issues

In mono-programming, if the active process, the only one in Main Memory, performs a Blocking I/O operation. The CPU is **idle** for the duration of the I/O, even if the I/O system is equipped with DMA (Direct Memory Access)!

If we accept single-process execution, we cannot allow the CPU to be idle for a long time. We need to find a way to fill this idle time.

V.5.2. Swapping Technique

In a single-programmed system, if a process occupies memory space, it will keep it until it terminates. To alleviate this problem, a temporary release of the occupied space gives waiting processes on disk the opportunity to occupy main memory.

The Swapping principle is based on the state of a process: if process P1 blocks due to I/O, another process P2 is loaded to run (see figure 5.6).

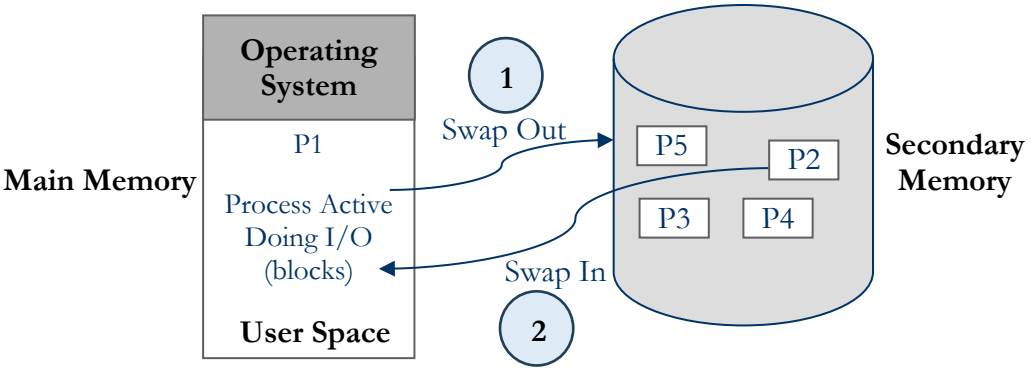


Figure 5.6 : Swapping Technique

A process must be in memory to be executed. A process, however, can be **swapped** temporarily out of memory to a **backing store** and then brought back into memory for continued execution (Figure 5.6). Swapping makes it possible for the total physical address space of all processes to exceed the real physical memory of the system, thus increasing the degree of multiprogramming in a system.

V.5.3. Overlays Technique

The size of a program can exceed the size of the Main Memory. To overcome this limitation in a single-programmed system, the programmer must divide his program at design time into a set of modules and load them dynamically at runtime into the Main Memory, so that he keeps only the modules he actually needs. Each new module loaded takes the place of the module to be unloaded.

The idea behind overlays is to only load the necessary parts of a program into memory at a given time, freeing up memory for other tasks. The unused portions of the program are kept on disk or other storage, and are loaded into memory as needed. This allows programs to be larger than the available memory, but still run smoothly.

The concept of overlays is that whenever a process is running it will not use the complete program at the same time, it will use only some part of it. Then overlays concept says that whatever part you required, you load it and once the part is done, then you just unload it, means just pull it back and get the new part you required and run it.

V.6. Memory Management in Multi-Programmed Systems

Multi-programming enables several processes to be executed at the same time. This optimizes processor utilization by reducing I/O waits. This technique requires the presence of several processes in memory. Memory is therefore shared between the operating system and several processes. However, the following problem arises:

How can memory be organized in such a way as to enable several processes to coexist efficiently, while ensuring process protection?

There are two main approaches:

1) Contiguous allocation approach is essentially a method in which a single section/part of contiguous memory is allocated to a process or file that needs it. As a result, all available memory space resides in the same place together, meaning that available (free/unused) memory partitions are not randomly distributed here and there throughout the memory space.
A program: a set of contiguous memory words that are inseparable (unbreakable).
There are two main types of technique in this approach:
- Fixed (or Static) Partitioning.
- Variable (or Dynamic) Partitioning.

2) Non-contiguous allocation approach is basically a method unlike the contiguous allocation method, allocates the memory space present in different locations to the process according to its needs. As all available memory space is in a distributed model, freely available memory space is also scattered here and there.
A program: a set of noncontiguous memory words that are separable (breakable).

This approach mainly involves the following techniques:

- Pagination,
- Segmentation
- Paginated Segmentation.

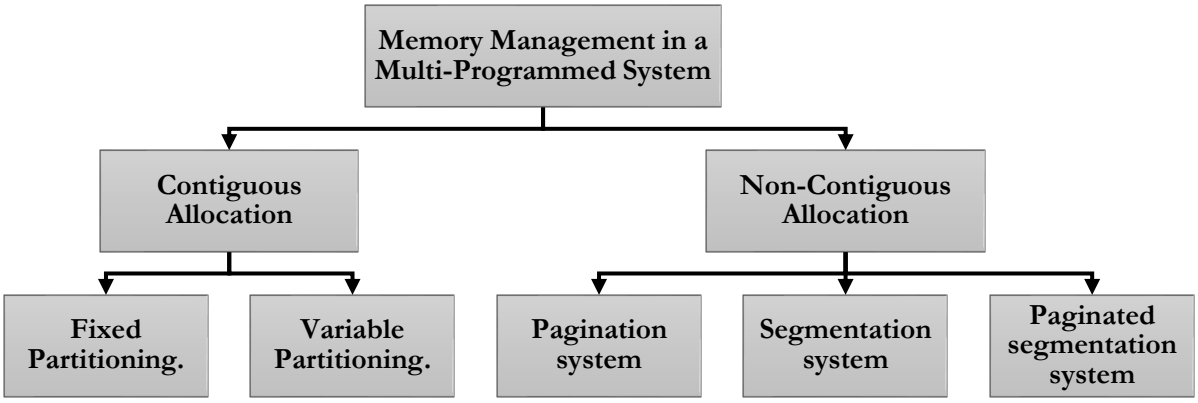


Figure 5.7 : Approaches to Memory Management in Multi-Programmed Systems.

V.6.1. Managing Free Memory

The system keeps tracks of the free disk blocks for allocating space to files when they are created. Also, to reuse the space released from deleting the files, free space management becomes crucial. The system maintains a free space list which keeps track of the disk blocks that are not allocated to some file or directory. When memory is assigned dynamically, the operating system must manage it. In general terms, there are two ways to keep track of memory usage: bitmaps and free lists

V.6.1.1. Bitmap or Bit Vector

The main memory is divided into a set of allocation blocks ranging from a few bytes to a few KB. To manage these blocks, each one is assigned a bit indicating its availability:

The bit can take two values: **0** and **1**:

- **1** indicates the allocation block (memory unit) is **occupied**;
- **0** indicates the allocation block (memory unit) is **free**.

All these bits make up what is known as the **bit table** or **mapping table**. This table is stored in main memory in the (protected) system area. By increasing the size of the allocation unit, we reduce the size of the bit table, but lose a lot of memory space (internal fragmentation).

Disadvantages

- The main problem with it is that when it has been decided to bring a **k** unit process into memory, the memory manager must search the bitmap to find a run of **k** consecutive **0** bits in the map.
- This technique is rarely used, as the search method is slow.

V.6.1.2. Linked Lists

Another way of keeping track of memory is to maintain a linked list of allocated and free memory segments, where a segment is either a process or a hole between two processes. **A segment is a set of consecutive allocation units.** An element of the list is made up of four fields indicating :

- The free or busy state of the segment (H : Hole or P : Process).
- Segment start address.
- Segment length.
- The pointer to the next element in the list.

In short, linked lists are a faster solution than the previous one for allocation, but slower for release.

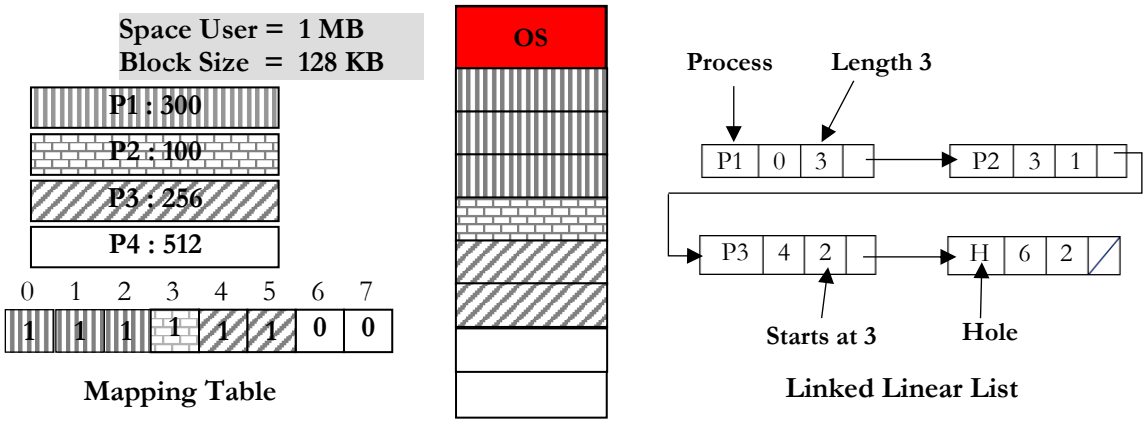


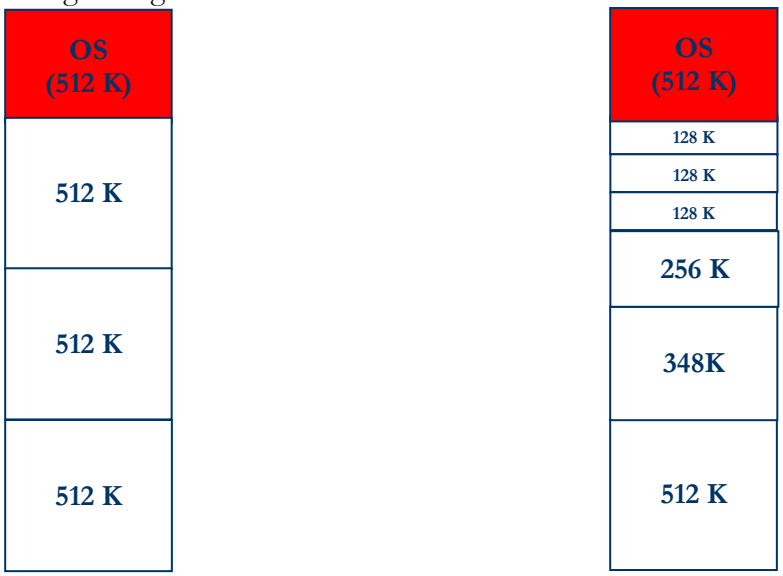
Figure 5.8 : Main Memory State Represented by Bitmap and Linked Linear List.

V.6.2. Contiguous Allocation

This strategy represents a simple technique for implementing multiprogramming. The main memory is divided into separate regions or memory partitions; each partition has its own address space. Memory partitioning can be static (fixed) or dynamic (variable). Each process is loaded completely into memory. The executable contains relative addresses, and the actual addresses are determined at the time of loading.

V.6.2.1. Fixed Partitioning

This solution involves dividing the memory into fixed partitions, not necessarily of equal size, when the system is initialized. The partitions can be of either equal or unequal size (see figure 5.9). On the other hand, the number of these partitions is prefixed (by the system or, possibly, by the constructor) and their sizes are different, but also prefixed. One of the problems in designing such a system is determining the right number and size.



(a) Equal-size partitions **(b) Unequal-size partitions**

Figure 5.9 : Example of Fixed Partitioning of a 2-Mbyte Memory.

Unequal partitioning is more flexible than equal partitioning but requires comparatively more management effort from the OS. Nevertheless, fixed partitioning is relatively simple, requiring minimal management overhead and processing.

A) Partition Table

Partitions are managed by the system using a partition table.

The operating system maintains a **Partition Description Table (PDT)** showing which parts of memory are available and which are occupied.

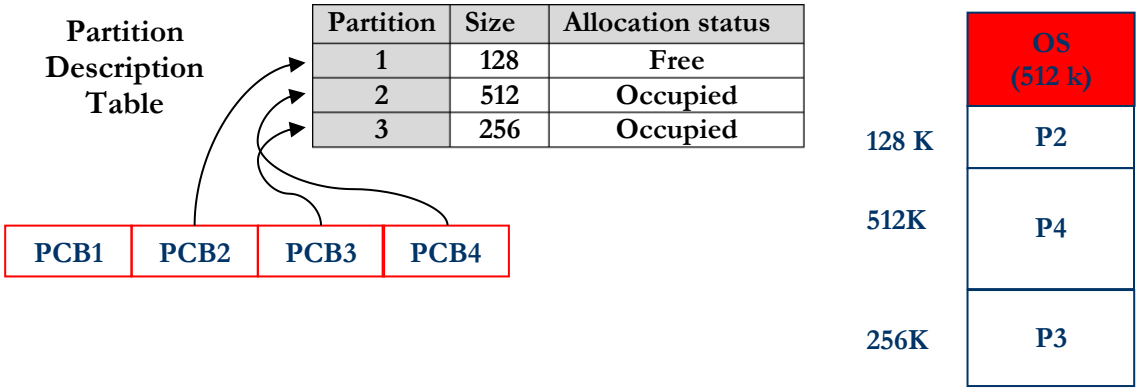


Figure 5.10 : Contiguous Allocation Scheme for Multiple Fixed Partitioning Using PDT.

B) Queue Ready

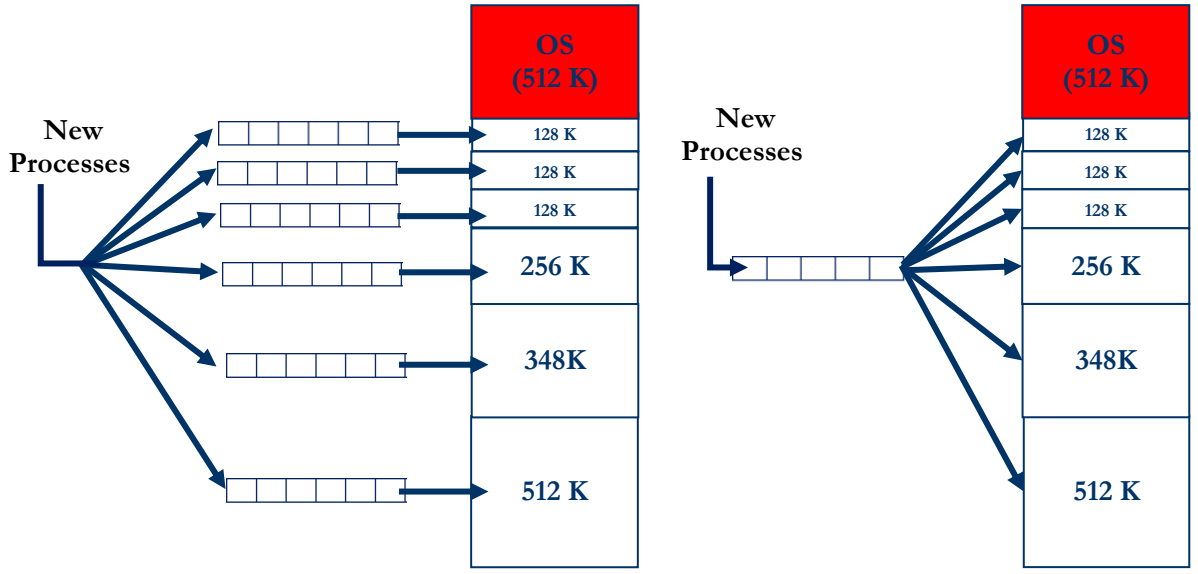
Programs that have not been able to fit into memory are placed in a queue (a single queue or a queue per partition).

- **In the case of a queue per partition** : each new process is placed in the queue of the smallest partition that can contain it (see figure 5.11a). This can lead to a process being kept waiting in one queue, while another partition that can contain it is free.
- **The alternative to this approach is to use a single queue**: as soon as a partition becomes free, the system places the first process in the queue that can fit (see figure 5.11b). In this approach, there are different strategies for allocating a free partition to a waiting process:
  - ✓ As soon as a partition becomes available, it is allocated to the first process in the queue that can fit on it. The disadvantage is that a large partition can be assigned to a small process, wasting a lot of space.
  - ✓ As soon as a partition becomes available, it is allocated to the largest process in the queue that can fit in it. The disadvantage is that small processes are penalized.

Hence, fixed partitioning contiguous memory allocation is not seen in modern systems. IBM OS/MFT (Multiprogramming with a Fixed number of Tasks) - an early mainframe OS had implementation of fixed partitioning.

Disadvantages

- **Internal fragmentation**: When a process does not occupy all the space in the partition. The remaining space is unusable.
- Queue management problems (overloading, Wasting memory, etc.).
- The degree of multi-programming is limited by the number of partitions.



(a) One Queue per partition **(b) Single Queue**

Figure 5.11 : Memory Assignment for Fixed Partitioning

Example 1 :

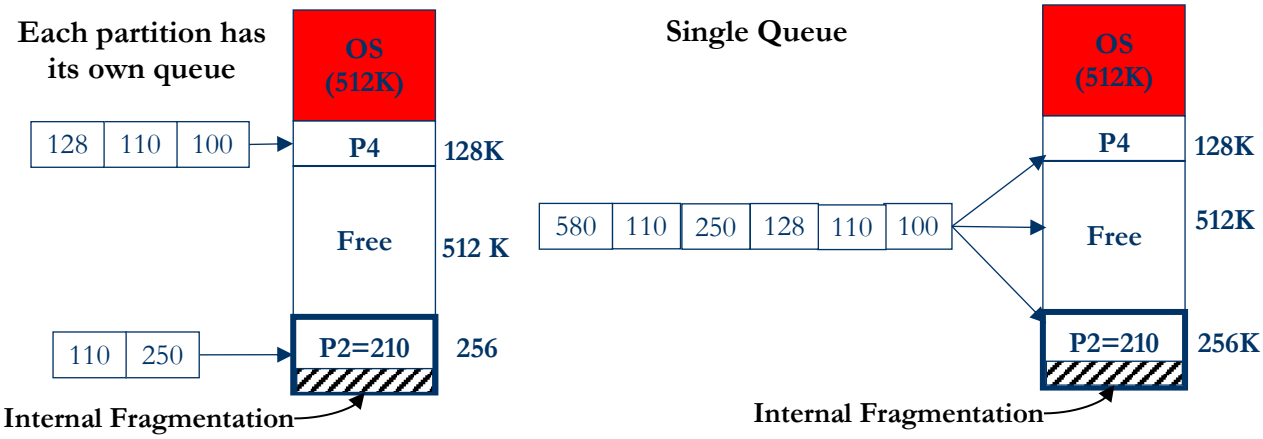


Figure 5.12 : Example of Fixed Partitioning With Internal Fragmentation

V.6.2.2. Variable (or Dynamic) Partitioning

Wasted Memory and internal fragmentation in fixed-partition systems led to the development of variable partitions. In this case, memory is allocated dynamically (variable partitions), according to process demand and load time. **Each program is allocated a partition exactly equal to its size**. When a program finishes execution, its partition is recovered by the system and allocated to another program, either fully or partially, depending on demand. We are no longer limited by partitions that are too large or too small, as with fixed partitions. This improved use of main memory requires a more complex allocation and liberation (release) mechanism; for example, a list of available memory spaces must be maintained.

As this example shows, this method starts out well, but eventually it leads to a situation in which there are a lot of small holes in memory. As time goes on, memory becomes more and more fragmented, and memory utilization declines. This phenomenon is referred to as **external fragmentation**, indicating that the memory that is external to all partitions becomes increasingly fragmented. This is in contrast to internal fragmentation, referred to earlier.

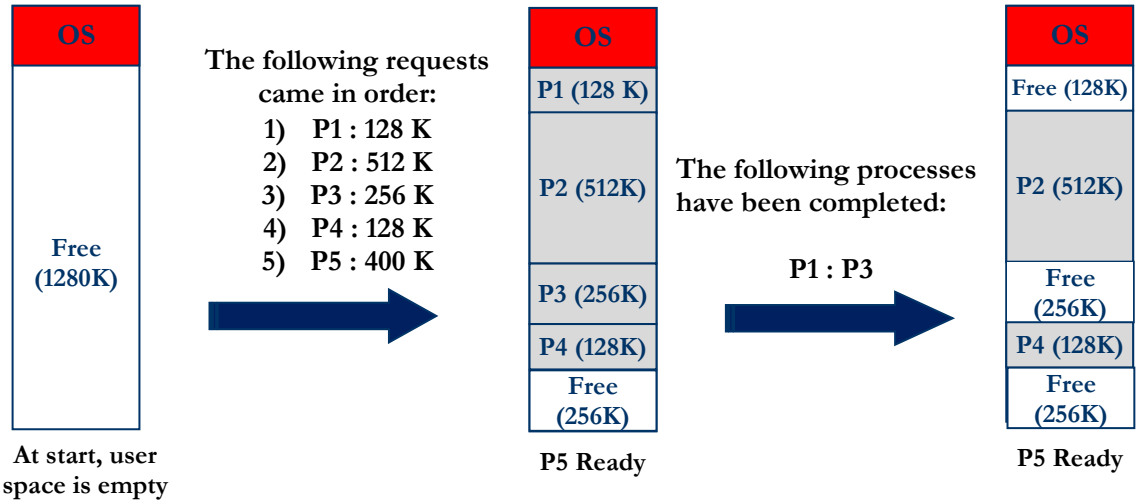


Figure 4.13 : Example of Variable Partitioning With External Fragmentation.

Disadvantages

- **External fragmentation:** When a process cannot find a suitable partition size when the sum of free partitions is sufficient.

How can the issue of external fragmentation be effectively managed, as it keeps on increasing with continuous arrival of newer processes? Two popular solutions applied are:

- **Compaction**
- **Placement algorithms.**

A) Compaction

One technique for overcoming external fragmentation is compaction: From time to time, the operating system shifts the processes so that they are contiguous and so that all of the free memory is together in one block.

The difficulty with compaction is that it is a time-consuming procedure and wasteful of processor time. Note that compaction implies the need for a dynamic relocation capability. That is, it must be possible to move a program from one region to another in main memory without invalidating the memory references in the program.

On the other hand, the compaction operation is performed when a program requesting execution cannot find a large enough partition, but its size is smaller than the existing external fragmentation.

Example 2:

In the example below (see figure 5.14), the number of free zones is **4**. None of these zones satisfies one of the three waiting processes, but their sum satisfies all three.

So, it is in our interest to pick up the free space. This is achieved by moving the loaded programs to form a single free zone. This operation is known as **Garbage Collection**.

B) Fragmentation

The disadvantage of contiguous memory allocation is fragmentation. There may be enough free memory to load a process, but no partition is of sufficient size. There are two types of fragmentation, namely, internal fragmentation and External fragmentation.

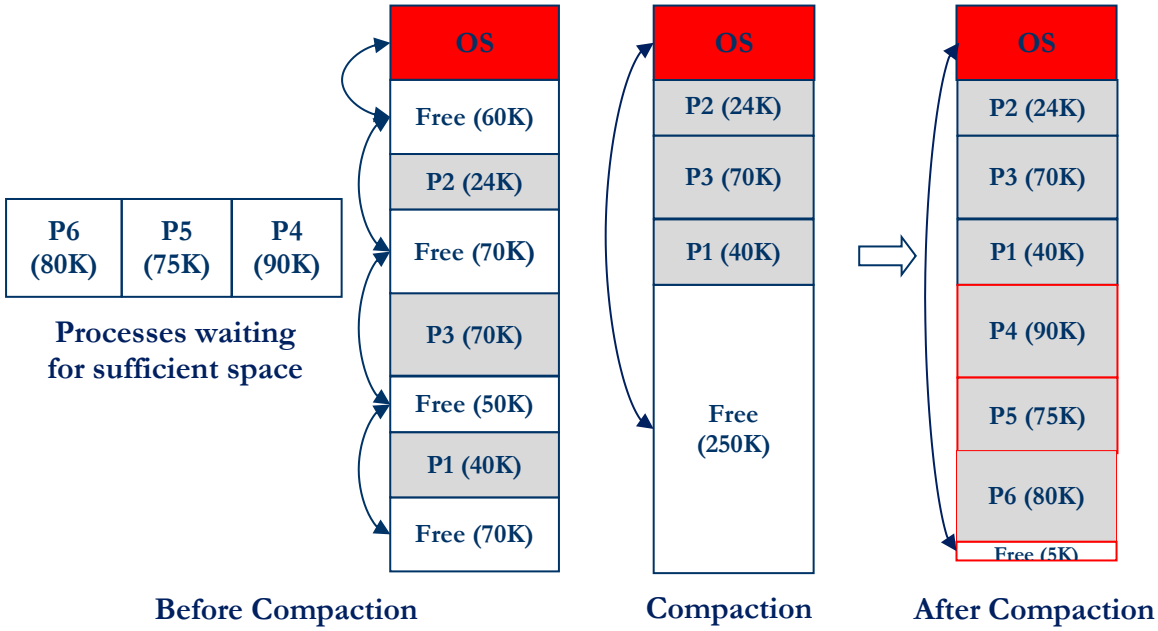


Figure 5.14 : Example of Compaction.

- **Internal fragmentation:** Allocated memory may be slightly larger than required memory. This difference is called internal fragmentation - memory that is internal to a partition but not used.
- **External fragmentation:** occurs when there is sufficient total memory space to satisfy a request, but it is not contiguous; memory is fragmented into a large number of small holes (i.e. free blocks) where a program cannot be loaded into any of these holes.

C) Placement Algorithms

Because memory compaction is time consuming. Another way out to minimize the fragmentation is done while placing new processes into the **available holes.** The OS maintains a list of available holes with their sizes when processes leave the memory. Before a new process is allocated space, the search is made on the list to find the most appropriate hole.

There are different algorithms for allocating holes to the requesting processes. The aim of all these algorithms is to maximize the memory space occupied, in other words, to reduce the probability of situations where a process cannot be served, even if there is enough memory. There are three main allocation strategies:

- **First fit**. Allocate the first hole that is big enough. Searching can start either at the beginning of the set of holes or at the location where the previous first-fit search ended. We can stop searching as soon as we find a free hole that is large enough.
- **Best fit**. Allocate the smallest hole that is big enough. We must search the entire list, unless the list is ordered by size. This strategy produces the smallest leftover hole.
- **Worst fit**. Allocate the largest hole. Again, we must search the entire list, unless it is sorted by size. This strategy produces the largest leftover hole, which may be more useful than the smaller leftover hole from a best-fit approach.

Simulations have shown that both first fit and best fit are better than worst fit in terms of decreasing time and storage utilization. Neither first fit nor best fit is clearly better than the other in terms of storage utilization, but first fit is generally faster.

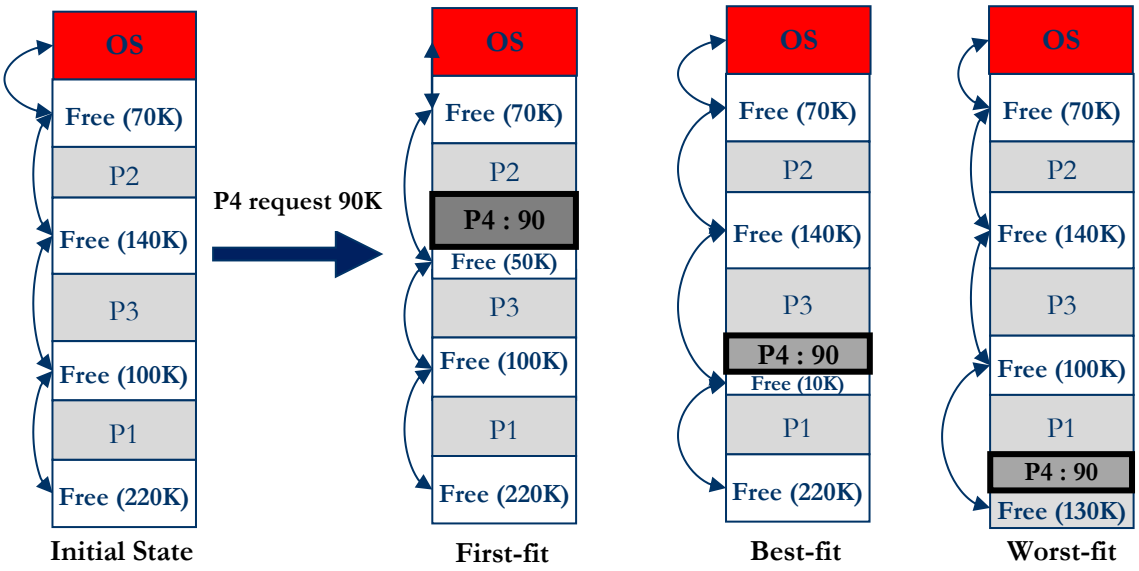


Figure 5.15 : Example Illustrating Placement Algorithms.

D) Buddy System

Fixed partitioning limits the number of active processes depending on the pre-decided partitions. Dynamic partitioning is a complex scheme to manage and requires time-consuming compaction. A compromise between the two is the buddy system.

Memory is dynamically divided in multiples of 2^k words ($L \leq k \leq U \;/\; L, k, U \in \mathbb{N}$) , where :

- 2^L = smallest size block that is allocated
- 2^U = largest possible block-size that is allocated; generally, 2^U is the size of the entire memory available for allocation

To begin, the entire space available for allocation is treated as a single block of size 2^U.

- **If** a request of size S such that $2^{U-1} < S \leq 2^U$ is made, the entire block is allocated to it.
- **Otherwise**, the block is **split into two equal buddies** of size 2^{U-1}.
  - ✓ **If** $2^{U-2} < S \leq 2^{U-1}$, then the request is allocated to one of the two buddies.
  - ✓ **Otherwise**, one of the buddies is split in half again.
- This process continues until the smallest block greater than or equal to S is generated and allocated to the request.
- At any time, the buddy system maintains a list of holes (unallocated blocks) of each size 2^i.
- A hole may be removed from the $(i + 1)$ list by splitting it in half to create two buddies of size 2^i in the i list.
- Whenever a pair of buddies on the i list both become unallocated, they are removed from that list and coalesced into a single block on the $(i + 1)$ list.
- Presented with a request for an allocation of size k such that $2^{i-1} < k \leq 2^i$, the following recursive algorithm **`get_hole`** is used to find a hole of size 2^i:

```
void get_hole(int i) {
   if (i == (U + 1)) <failure>;
   if (<i_list empty>) {
         get_hole(i + 1);
         <split hole into buddies>;
         <put buddies on i_list>;
   }
   <take first hole on i_list>;
}
```

Disadvantages

- Even though the buddy system minimizes external fragmentation to some extent, internal fragmentation is very much there as we have to allocate space in the size of 2^i words, whereas need may be much less.
- The major drawback of this method is its internal fragmentation. A process requiring $2^n + 1$ (e.g. **257**) will have a block of 2^{n+1} (e.g. **512**).
- Also, two empty blocks can only be merged if they are next to each other. Otherwise, compaction is necessary.

Example 2:

Figure 5.16 gives an example using a 1-Mbyte initial block. Initially, the memory is empty.

	Memory (1 Mbyte)						Free Zones
Initially	1 M (1024K)						1
Request (A) 100K	A=128K	128K		256K	512K		3
Request (B) 240K	A=128K	64	64	B = 256K	512K		3
Request (C) 64K	A=128K	C	64	B = 256K	512K		2
Request (D) 256K	A=128K	C	64	B = 256K	D = 256K	256K	2
Release B	A=128K	C	64	256K	D = 256K	256K	3
Release A	128K	C	64	256K	D = 256K	256K	4
Request (E) 75K	E=128K	C	64	256K	D = 256K	256K	3
Release C	E=128K	128K		256K	D = 256K	256K	3
Release E	512K				D = 256K	256K	2
Release D	1 M (1024K)						1

Figure 5.16 : Example of Buddy System

V.6.3. Non-Contiguous Allocation

Previous allocation techniques, from the contiguous allocation approach, have several limitations:

1) At allocation time, the manager must find sufficient contiguous free space to accommodate the program. Even if there are free fragments whose sum is sufficient, contiguous allocation refuses to accommodate it. As the size of a program increases, the chances of finding such space decrease.

2) Frequent external fragmentation due to space allocation and deallocation. Compacting used to be the solution, but it's no longer the best one!

3) Programs whose size exceeds main memory are impossible to run.

To solve these problems, the memory manager **needs to change its view of the unbreakable (undivided) nature of a program**. Fragments of free space scattered throughout the main memory can be exploited if we consider the program to be an entity that can be split into several unbreakable chunks, each of which can occupy a free zone that is sufficient for it independently of the others.

The main aim of non-contiguous allocation is to be able to load a process while making the most of all memory holes.

A **program is divided into chunks** to enable each chunk to be allocated separately. The chunks are much smaller than the whole program and therefore allow more efficient use of memory. Consequently, smaller holes can be used more easily. Depending on the size of the chunks, two strategies can be distinguished:

- If the program chunks are of fixed and equal size, this is known as a **Paging**.
- If the program chunks are of variable size, this is referred to as a **Segmentation**.
- These two techniques can be combined, in which case we speak of the **Paged Segmentation** technique, which is the combination of paging and segmentation at the same time.

V.6.3.1. Paging

1) Principle of operation

In the paging mechanism, the program address space is divided into fixed-sized chunks (blocks) called **PAGES**. This space is called the logical program space. In turn, the physical memory (Main Memory) space is itself divided into fixed-sized chunks (blocks) called **FRAMES** or **PAGE FRAMES** (see figure 5.17).

The pages and frames are always the **same size.** This size is defined by the hardware and the target OS. Generally, it is power of **2**, between **512 Bytes** and **8192 Bytes**, depending on the computer architecture. We show in this section that the wasted space in memory for each process is due to internal fragmentation consisting of only a fraction of the last page of a process. There is no external fragmentation.

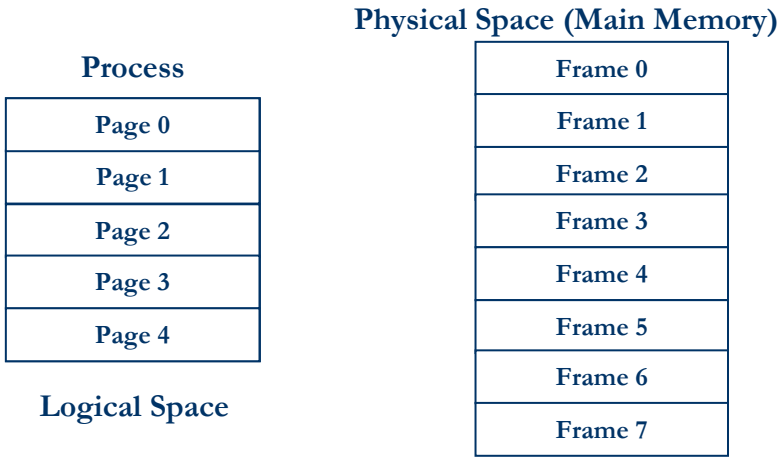


Figure 5.17 : Logical Space vs. Physical Space In Paging

Thus, a program of size **L** will be divided into a number of pages '**P**' such that '**P**' is the first integer verifying the relationship: $L >= n \times page\_size$.

- When a program is loaded, each page is loaded into any free slot in main memory. The set of slots housing the individual pages is not necessarily contiguous.
- In a pure paging system, a program can only be loaded if there is a number of free slots equal to the number of pages in the program.
- If the size **L** of a program is not a multiple of the size of a page, then the last page in the program's logical space will not be completely filled. This empty space is only on the last page, causing internal fragmentation.
- There is no external fragmentation in the pagination either.

2) Paging Hardware

The association of a logical page with a physical page is described in a table called the **Page Table**. Translating logical addresses into physical addresses is the responsibility of the **MMU**. The hardware support for paging is illustrated in Figure 5.18.

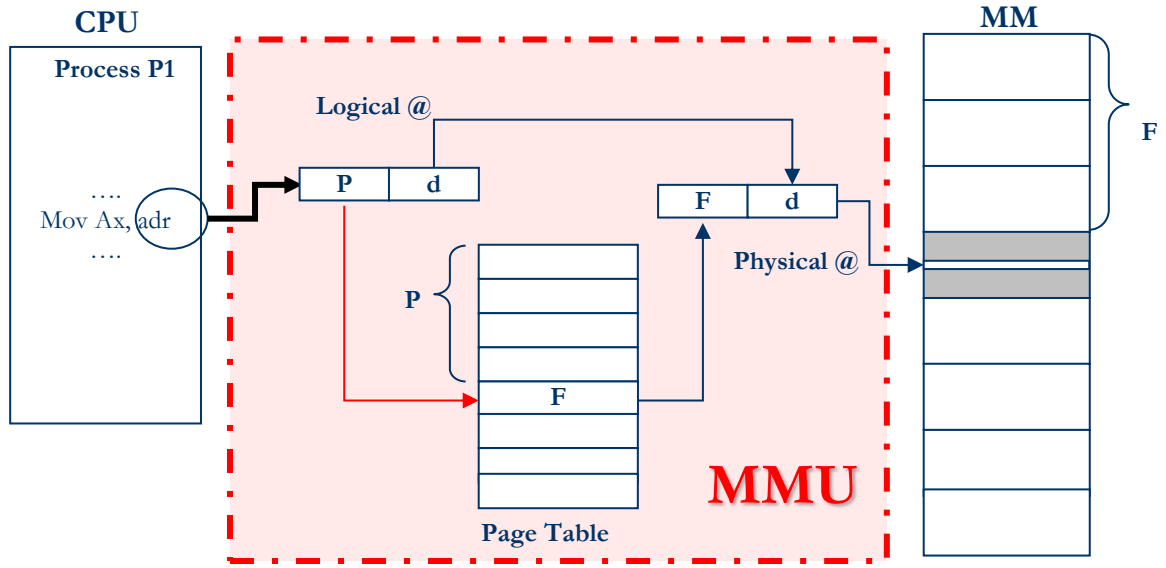


Figure 5.18 : Paging Hardware.

3) Logical-to Physical Address Translation for Paging

Every (logical) address generated by the CPU is divided into two parts: a **page number (P)** and a **page offset (d)**.

- A page number (**P**) which is used as an index into a **page table**.
- The **page table** contains the base address of each page in physical memory.
- A **page offset** (**d** : page displacement) which is combined with the **page base** address to define the physical memory address that is sent to the memory unit.

The paging model of memory is shown in Figure 8.19.

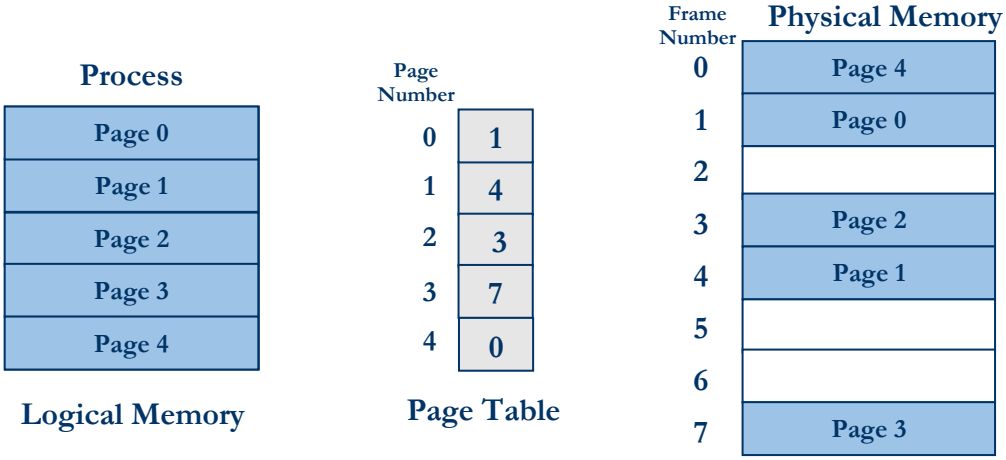


Figure 5.19 : Paging Model of Logical and Physical Memory.

If the size of the logical address space is 2^m, and a page size is 2^n bytes, then the high-order $m - n$ bits of a logical address designate the page number, and the n low-order bits designate the page offset. Thus, the logical address is as follows:

where P is an index into the page table and d is the displacement within the page.

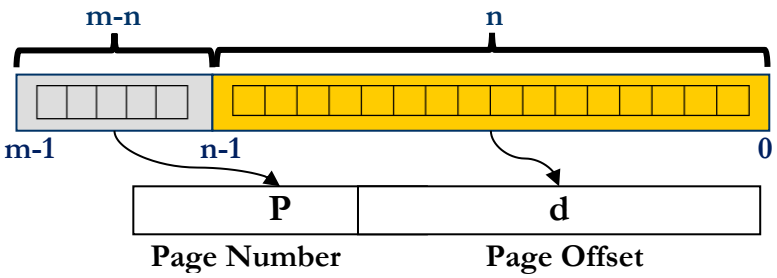


Figure 5.20 : The Paged Address

A logical address is referred in terms of page number and offset within the page as a tuple $< page\#, offset >$. So, if T is the size of a page and U is a logical address, then the paged address $< P, d >$ is deduced from the following formulas:

- $P = U \; Div \; T$ (where, **Div** is the integer division)
- $d = U \; Mod \; T$ (where **Mod** is the remainder of the integer division)

4) Get the physical address?

The physical address corresponding to a logical address $Logical @ = < P, d >$ is obtained by replacing the page number P by the frame number F, or in practice its layout address, containing this page. The offset (displacement) d in the page is the same in the frame, since both have the same size. The frame number or address is obtained by indexing the page table of the active process with the value P. The corresponding frame returns F, or the equivalent address.

Note : At runtime, and for a given processor, only one page table is active: that corresponding to the process currently being executed. Each context switching operation involves changing the active page table at processor level.

Example 3:

An address $AB = 1010\;1011$ on 8 bits, $m = 8 \; bits$. We also have pages of size $2^6 \; words$. So, the number of pages is $2^{m-n} = 2^{8-6} = 4$.

Then, the logical address becomes a paged address $< P, d >$ as follows:

- $P = 171 \; Div \; 64 = 2$, ➔ The page number is **2**
- $d = 171 \; Mod \; 64 = 43$, ➔ **43** is the offset in page number **2**.

AB @ = 1010 1011 ➔ AB @ = <2 , 43>

V.6.3.2. Segmentation

1) Principle of operation

The way in which a program is divided up, as presented in a pagination system, does not correspond to the user's vision and way of thinking. The latter sees a program as a collection of logical units: code, data, stack, main program, procedures, and DLL, etc., generally referred to as **Segments**.

- Paging does not project this aspect of reasoning.
- Main Memory is not broken down into blocks as in the case of paging.

The address space of a program is divided into several chunks of different sizes, but whose logical structure corresponds to a user viewpoint. This user view is not reflected in paging memory. Each segment of a program can be accommodated in a memory space independent of the other segments.

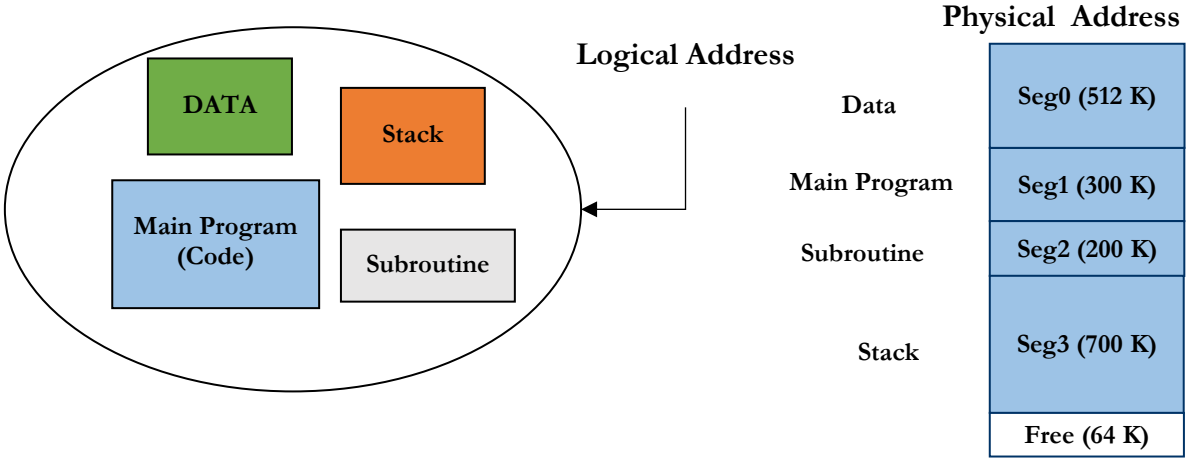


Figure 5.21 : Programmer's View of a Program & Segmentation.

Segmentation is a memory management strategy that reproduces the memory breakdown as logically described by the user. In a segmented memory, each logical unit of a user program is stored in a memory block, called a "**segment**", within which addresses are relative to the start of the segment. These segments are of different sizes. A program will therefore be made up of a set of code and data segments, which can be dispersed in main memory. Segmentation facilitates linker, as well as the sharing of data or code segments between processes.

Unlike the paging scheme, segmentation **eliminates internal fragmentation**, as the memory space allocated to a segment is exactly the same size as the segment itself. However, **external fragmentation may occur** due to the dynamic allocation of memory space.

Each segment has a name and a length. The addresses specify both the segment name and the offset within the segment. As with paging, a logical address using segmentation consists of two parts, in this case a **Segment number** and an **offset**.

2) Segmentation Hardware

This mapping is performed by a **segment table**. Each entry in the segment table has a **segment base** and a **segment limit**. The segment base contains the starting physical address where the segment resides in memory, and the segment limit specifies the length of the segment. The hardware support for segmentation is illustrated in Figure 5.22.

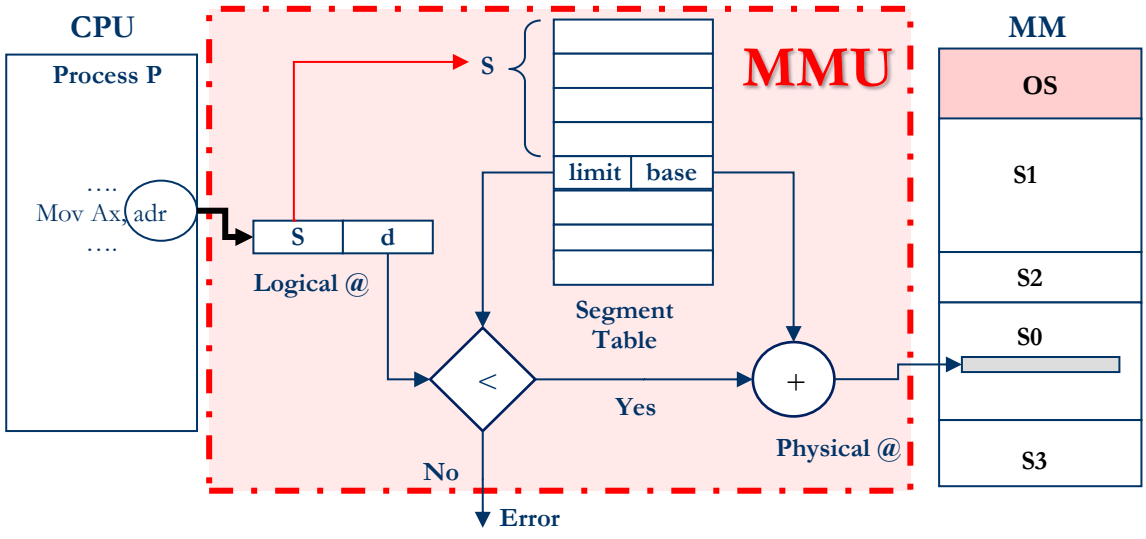


Figure 5.22 : Segmentation Hardware.

3) Logical-to Physical Address Translation for Segmentation

A logical address consists of two parts: a **segment number (S)**, and an **offset** in this segment **(d)**.

- The **segment number** (**S**) is used as an index to the **segment table**.
- The **offset** (**d**) of the logical address must be between **0** and the segment **limit**.
- The **segment table** is thus essentially an array of **base–limit** register pairs.
- The **segment base** contains the starting physical address where the segment resides in memory, and the **segment limit** specifies the length of the segment.

The segmentation model of memory is shown in Figure 8.21.

4) Get the physical address?

In a segmented system, a logical address referenced by an instruction is in the form of a pair $\boldsymbol{Logical}\ @ = < \boldsymbol{S}, \boldsymbol{d} >$ such that $\boldsymbol{S}$: segment number, and $\boldsymbol{d}$ displacement within the segment.

The memory manager needs to keep track of the memory location in which segment S is hosted. To do this, it builds a structure, usually software-based, for each process, called a segment table, in which it maps each segment loaded into the main memory (MM) to its location address (in the MM) and its size.

The offset $\boldsymbol{d}$ in the segment must be between 0 and the segment limit $\boldsymbol{d} \in [\boldsymbol{0}, \boldsymbol{L}[$, If it is not, we trap to the operating system (logical addressing attempt beyond end of segment).

- **When an offset is legal**, in which case the physical address is calculated by adding the value of $\boldsymbol{d}$ to the segment base.
- **Otherwise**, an addressing error is generated.

Example 4:

On a system using simple segmentation, calculate the physical address of each logical address, using the segment table below. Assume that the addresses are decimal instead of binary.

Segment	Base	Limit
0	1100	500
1	2500	1000
2	200	600
3	4000	1200

Logical Adress	Adresse physique
<0, 300>	300<500, so, **Phy @=1100+300=1400**
<2, 800>	800>600, so**, Addressing error**
<1, 600>	600<1000, so, **Phy @=2500+600=3100**
<3, 1100>	1100<1200, so, **Phy @=4000+1100=5100**
<1, 1111>	1111>1000, so**, Addressing error**

V.6.3.3. Paged Segmentation

Segment size can be large, which can result in long loading times. Paged segmentation, combining segmentation and paging, can be a solution. In this technique, **programs are divided into segments and each segment into pages.** This technique was invented for the Multicast system.

Each process is assigned a segment table and a page table. So, each segment address is not a memory address, but an address in the page table of the segment.

A logical address $< \boldsymbol{S}, \boldsymbol{D} >$, with $\boldsymbol{S}$ a segment number and $\boldsymbol{D}$ an offset (displacement) **within the segment**, is transformed into $< \boldsymbol{S}, \boldsymbol{P}, \boldsymbol{d} >$, where $\boldsymbol{S}$ a segment number, $\boldsymbol{P}$ is a page number and $\boldsymbol{d}$ an offset (displacement) **within page** $\boldsymbol{P}$. Each address then becomes a triplet $< \boldsymbol{S}, \boldsymbol{P}, \boldsymbol{d} >$.

The hardware support for paged segmentation is illustrated in Figure 5.23.

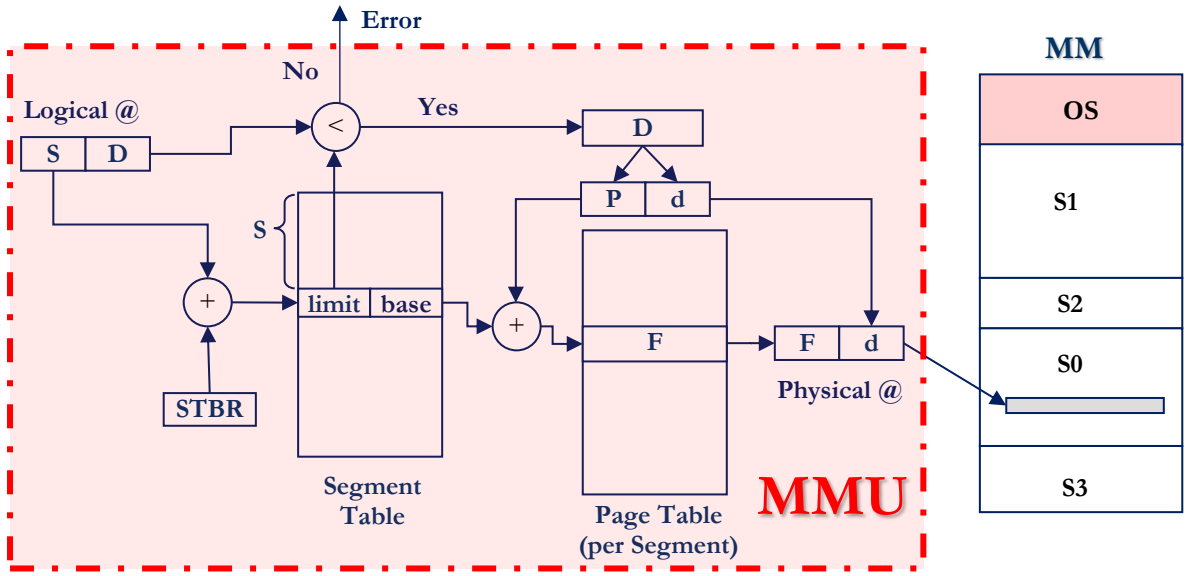


Figure 5.23 : Paged Segmentation Hardware.

This technique eliminates external fragmentation, it also introduces internal fragmentation.

V.7. Virtual Memory

Previous memory management policies all have disadvantages. Because of this, they are seldom used as a general policy in contemporary operating systems. These disadvantages are:

1) There is some portion of the main memory that is unused in the form of external fragmentation (none of the free partitions are large enough to hold a process), or the total size of all free page frames is short of holding a complete program.
2) The maximum size of every program and its data is limited to the size of user main memory. **Longer programs are never executed**.
3) When a program comes into main memory, it remains there until termination even though part of the program, and/or data and results, might have been **unused for a long time** (only one page or segment is active at any given time). The space that is **occupied by these parts could have been used to load new programs**.
4) The belief that we could still increase the average number of live processes and improve the overall system efficiency, if better main memory management policies are used.

The solution to all aforementioned deficiencies is virtual memory.

V.7.1. Basics of Virtual Memory

Virtual memory is a technique that allows processes to be executed that may not be completely in memory. It is based on the Main Memory (MM) virtualization technique (like a virtual CPU, virtual printer, etc.). The principle is to let processes see that they have a theoretically **infinite amount of memory**. Part of the program code is loaded into the MM, while the other part is stored in a memory extension of the MM. On the other hand, virtual memory provides an extremely large address space, whereas physical memory is limited. This is made possible by using auxiliary memory as a workspace for loading and unloading individual pages by the OS.

To achieve this, programs in Secondary Memory (SM) are divided into a number of chunks, each of which is smaller than the physical size of the MM. At runtime, a few chunks are loaded; as soon

as another is required, it is loaded into the MM, while, if memory space is insufficient, another is output. In this way, the OS considers the MS to be a logical extension of the MM. In practice, only part of the MS, known as the swap zone or MV, is extended to the MM.

In this way, a requesting process will be loaded entirely into the new memory (MM + Swap). Only part of its code is loaded into the MM. We emphasize that the MM for user processes is the one formed by the real MM + the swap zone. Exchanges between the two are transparent to the user.

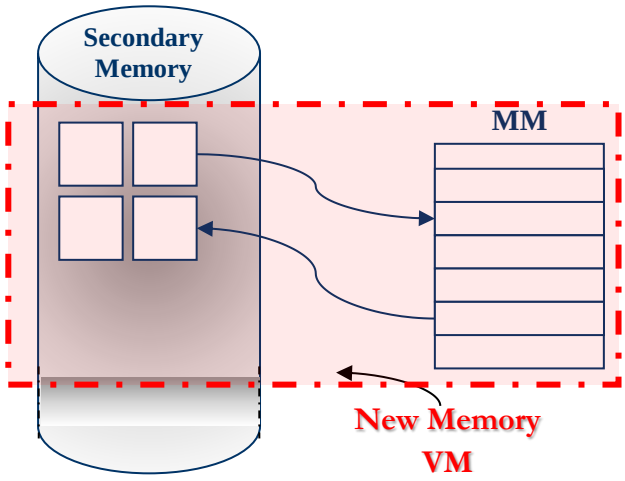


Figure 5.24 : Virtual Memory

V.7.2. Demand Paging

1) Basic Concepts

The principle of virtual memory is commonly implemented with On-Demand Paging, i.e. process pages are only loaded into Main Memory when the processor requests access to them. A demand paging is similar to a paging system with swapping

How can the OS (Using Hardware Support) distinguish between pages in memory and those on disk and detect their absence?

Using this technique, the OS has the means to distinguish between pages that are in memory, and those that are on disk. It uses an additional bit ***V*** in the page table, called the **validation bit** (Valid/invalid), to describe whether the page is loaded into memory or not.

- ***V*** = **1** ➔ The page is **loaded into the Main Memory**, access to it is **valid**
- ***V*** = **1** ➔ The page **is not loaded into the Main Memory**, access to it is **invalid**

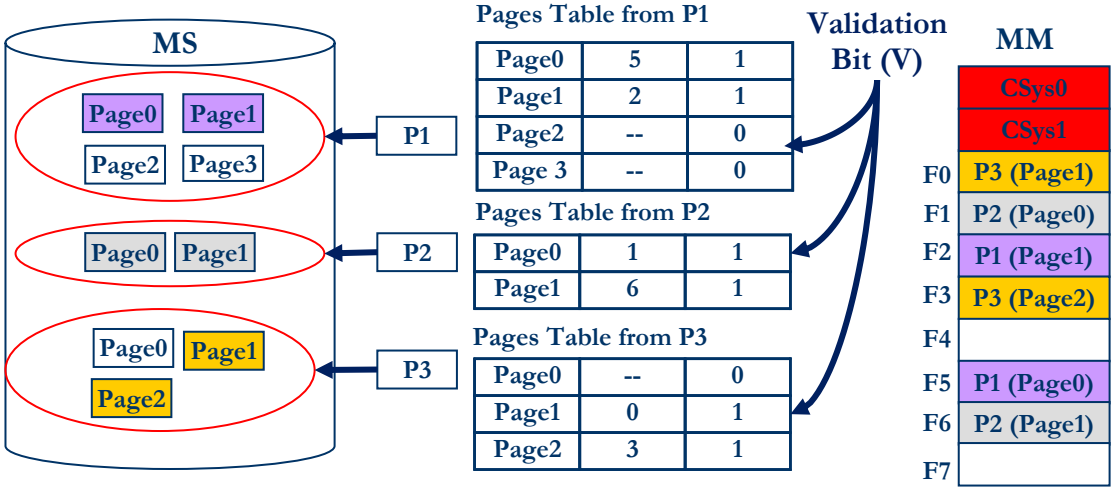


Figure 5.25 : Page Tables with Validation Bit

2) Demand Paging Hardware

Following a reference to any '**adr**' address, this is converted by the MMU into a $< P, d >$ pair.

- The ***P*** value is used to index the page table; before converting, we must first consult the **V** bit associated with it.
  - ✓ **If this bit is set to 1,** this means that the page has been loaded into Main Memory, so conversion can continue.
  - ✓ **Otherwise**, the MMU generates a diversion, indicating that the process wants to access a page that does not exist in the Main Memory. This is called a **page fault**.

The hardware support for demand paging is illustrated in Figure 5.26.

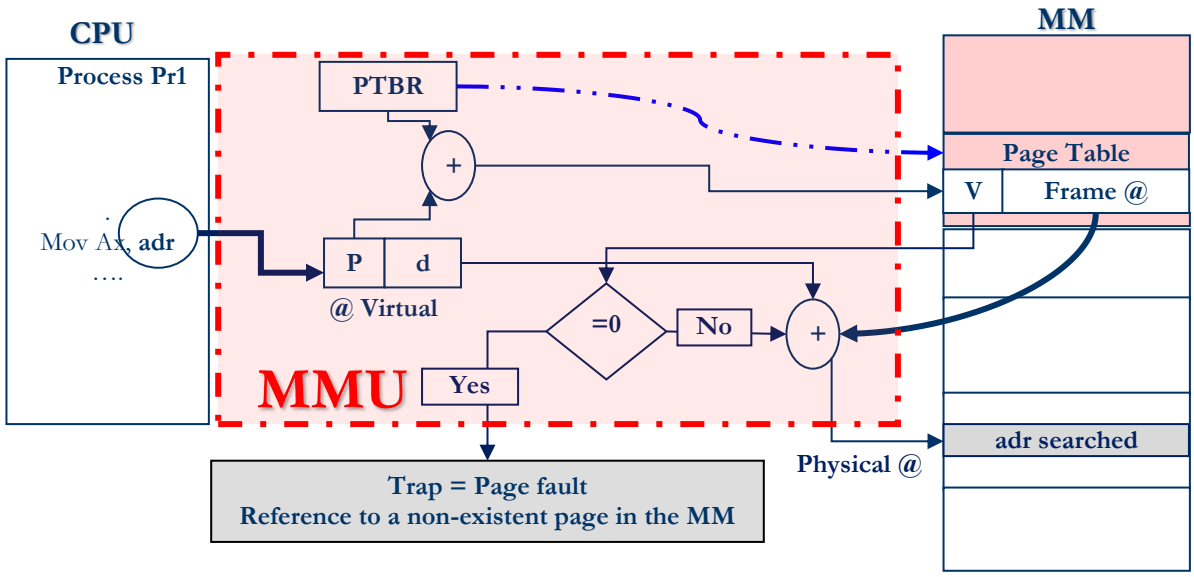


Figure 5.26 : Demand Paging Hardware.

3) Page Faults

What happens if the process tries to access a page that was not brought into memory?

Access to a page marked invalid causes a **Page Fault**. The paging hardware, in translating the address through the page table, will notice that the invalid bit is set, causing a trap to the operating system. This trap is the result of the operating system's failure to bring the desired page into memory.

The procedure for handling this page fault is straightforward (Figure 5.27):

1) We check an internal table (usually kept with the process control block) for this process to determine whether the reference was a valid or an invalid memory access.
2) If the reference was invalid, we terminate the process. If it was valid but we have not yet brought in that page, we now page it in.
3) We find a free frame (by taking one from the free-frame list, for example).
4) We schedule a disk operation to read the desired page into the newly allocated frame.
5) When the disk read is complete, we modify the internal table kept with the process and the page table to indicate that the page is now in memory.
6) We restart the instruction that was interrupted by the trap. The process can now access the page as though it had always been in memory.

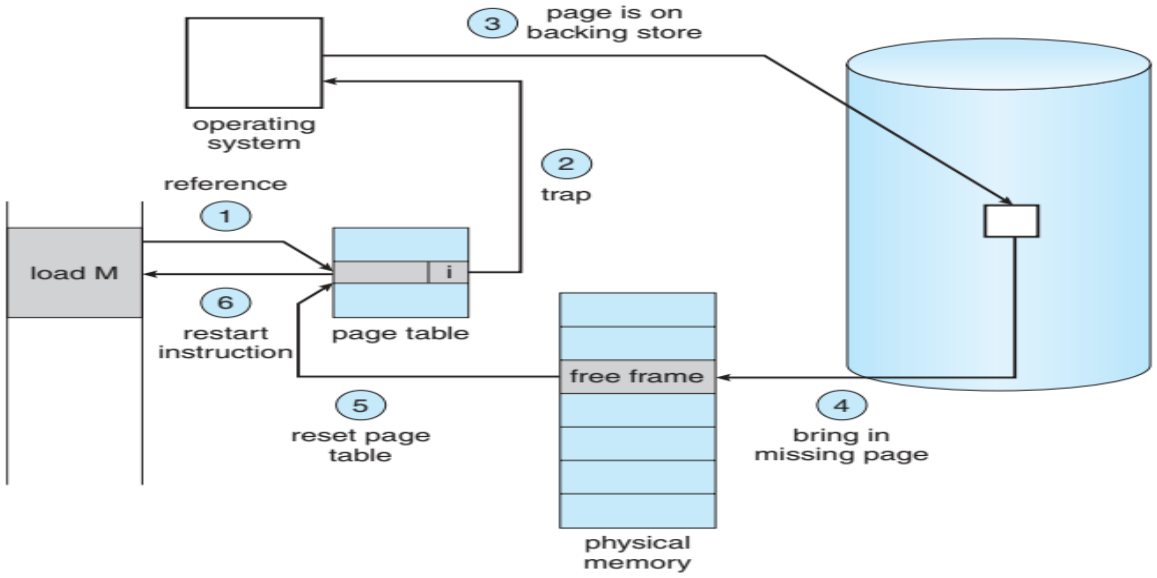


Figure 5.27 : Steps in Handling a Page Fault.

4) Performance of Demand Paging

Demand paging can significantly affect the performance of a computer system. To see why, let's compute the **effective access time** for a demand-paged memory. For most computer systems, the **memory-access time**, denoted $\boldsymbol{ma}$, ranges from **10** to **200 nanoseconds**.

- As long as we have no page faults, the effective access time is equal to the memory access time.
- If, however, a page fault occurs, we must first read the relevant page from disk and then access the desired word

Let $\boldsymbol{p}$ be the probability of a page fault $(\boldsymbol{0 \leq p \leq 1})$. We would expect $\boldsymbol{p}$ to be close to zero that is, we would expect to have only a few page faults. The effective access time is then

$$\boldsymbol{effective\ access\ time = (1 - p) \times ma + \rho \times page\ fault\ time.}$$

To compute the effective access time, we must know how much time is needed to service a page fault.

Example 5:

With an average page-fault service time of **8 milliseconds** and a memory access time of **200 nanoseconds**, the effective access time in nanoseconds is

$$\begin{aligned} \boldsymbol{effective\ access\ time} &= \boldsymbol{(1 - p) \times (200) + p\,(8\ milliseconds)} \\ &= \boldsymbol{(1 - p) \times 200 + p \times 8{,}000{,}000} \\ &= \boldsymbol{200 + 7{,}999{,}800 \times p} \end{aligned}$$

V.7.3. Page Replacement

Page replacement takes the following approach. If no frame is free, we find one that is not currently being used and free it. We can free a frame by writing its contents to swap space and changing the page table (and all other tables) to indicate that the page is no longer in memory (Figure 5.28). We can now use the freed frame to hold the page for which the process faulted. We modify the page-fault service routine to include page replacement:

1) Find the location of the desired page on the disk.

2) Find a free frame:

A) If there is a free frame, use it.

B) If there is no free frame, use a page-replacement algorithm to select a **victim frame**.

C) Write the victim frame to the disk; change the page and frame tables accordingly.

3) Read the desired page into the newly freed frame; change the page and frame tables.

4) Continue the user process from where the page fault occurred.

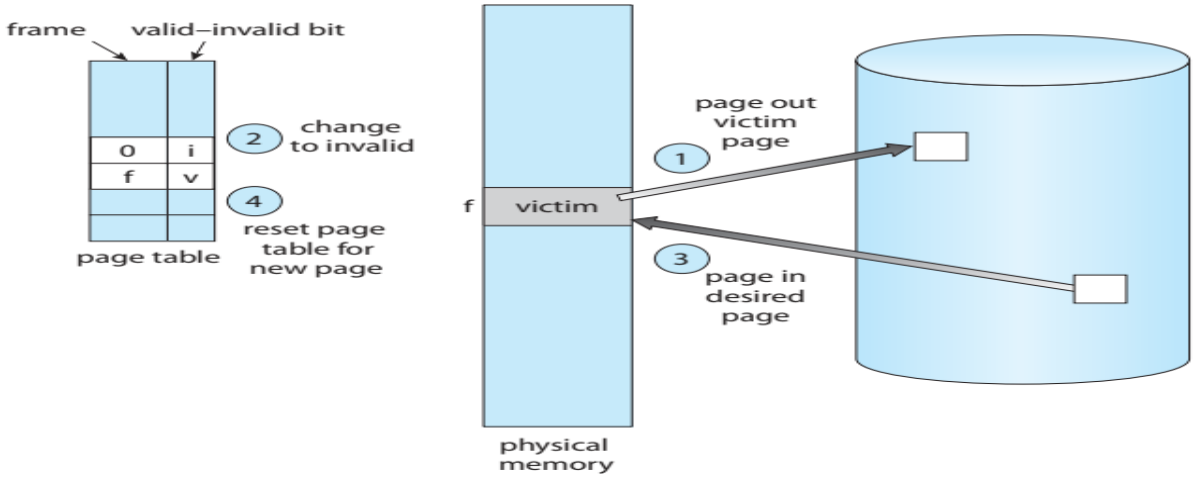


Figure 5.28 : Page Replacement.

Notice that, if no frames are free, **two-page transfers** (one out and one in) are required. This situation effectively doubles the page-fault service time and increases the effective access time accordingly. We can reduce this overhead by using a **modify bit** (or **dirty bit**). When we select a page for replacement, we examine its modify bit.

- If the bit is set, we know that the page has been modified since it was read in from the disk. In this case, we must write the page to the disk.
- If the modify bit is not set, however, the page has not been modified since it was read into memory. In this case, we need not write the memory page to the disk: it is already there.

V.7.4. Page Replacement Algorithms

There are several different page replacement algorithms. How do we select a particular replacement algorithm? In general, we want the one with the **lowest page-fault rate**. We evaluate an algorithm by running it on a particular string of memory references and calculating the number of page faults. The string of memory references is called a **reference string**. The future behavior of a program can be controlled by knowledge of this reference string.

In order to determine the number of page faults for a particular reference string and replacement algorithm, we also need to know the number of page frames available. Obviously, as the number of page frames increases, the number of page faults must decrease.

Example 6:

For this example, if we trace a particular process, we might record the following address sequence:

$$100, 210, 355, 120, 420, 110, 200, 550, 139, 201, 395, 404, 505.$$

At **100 bytes per page**, this sequence is reduced to the following reference string:

$$1, 2, 3, 1, 4, 1, 2, 5, 1, 2, 3, 4, 5.$$

What is the number of page faults with different replacement algorithms?

V.7.4.1. FIFO Page Replacement

Principle: Replace the old page loaded

Reason: It is highly likely that the old page is not frequently referenced as a recent page

Implementation: For each page in a page table, a field is initialized with the loading time.

Memory with 3 Frames

Reference String	1	2	3	1	4	1	2	5	1	2	3	4	5
Frame 1	1	1	1	1	4	4	4	5	5	5	5	5	5
Frame 2		2	2	2	2	1	1	1	1	1	3	3	3
Frame 3			3	3	3	3	2	2	2	2	2	4	4
Page Fault	X	X	X		X	X	X	X			X	X	

- **9-Page Faults** for a memory with **3 frames**
- You might think that if you increase the number of frames, the number of page faults will decrease!

We add a frame to the memory, and we obtain a

Memory with 4 Frames. And we can see :

Reference String	1	2	3	1	4	1	2	5	1	2	3	4	5
Frame 1	1	1	1	1	1	1	1	5	5	5	5	4	4
Frame 2		2	2	2	2	2	2	2	1	1	1	1	5
Frame 3			3	3	3	3	3	3	3	2	2	2	2
Frame 4					4	4	4	4	4	4	3	3	3
Page Fault	X	X	X		X			X	X	X	X	X	X

- The result is an increase in the number of page faults to **10** (Memory with 4 Frames).
- This phenomenon is known as the **Belady anomaly**.

V.7.4.2. Optimal Page Replacement

Principle: The victim page is the page that will not be referenced in the immediate future.

Reason: It's the most appropriate frame, because all the others will be referenced before it.

Implementation: How can we know that a page will be referenced in the future?

- Non-implementable algorithm, used as a reference for comparison with other algorithms.

Memory with 4 Frames

Reference String	1	2	3	1	4	1	2	5	1	2	3	4	5
Frame 1	1	1	1	1	1	1	1	1	1	1	1	4	4
Frame 2		2	2	2	2	2	2	2	2	2	2	2	2
Frame 3			3	3	3	3	3	3	3	3	3	3	3
Frame 4					4	4	4	5	5	5	5	5	5
Page Fault	X	X	X		X			X				X	

- **6-Page Faults** for a memory with **4 Frames**!

V.7.4.3. Least Recently Used (LRU) Page Replacement

Principle: The victim page will be the least referenced page in the near past.

Reason: The principle of spatial locality and temporal locality means that a recently used page has a high chance of being referenced in the very near future. So, the victim page should be an older page.

Implementation: A field for each entry in the page table indicating the time of the last reference to the page. Each time the page is referenced (read or write), this field is updated.

- The victim page is the one with the oldest reference time.
- This algorithm is a little cumbersome to implement. It's used in most cases, but with slightly lighter versions.

Memory with 4 Frames

Reference String	1	2	3	1	4	1	2	5	1	2	3	4	5
Frame 1	1	1	1	1	1	1	1	1	1	1	1	1	5
Frame 2		2	2	2	2	2	2	2	2	2	2	2	2
Frame 3			3	3	3	3	3	5	5	5	5	4	4
Frame 4					4	4	4	4	4	4	3	3	3
Page Fault	X	X	X		X			X			X	X	X

- **8-Page Faults** for a memory with **4 Frames**!

V.7.4.4. Least Frequently Used (LFU) Page Replacement

Principle: The victim page is the page that has been used least recently.

Reason: It is the least used page in the near past among the others, so we can assume that it has a low probability of being referenced in the near future.

Implementation: A field for each entry in the page table indicating the number of times this page has been referenced. The victim page is the one with the lowest value.

Memory with 4 Frames

Reference String	1	2	3	1	4	1	2	5	1	2	3	4	5
Frame 1	1	1	1	1	1	1	1	1	1	1	1	1	1
Frame 2		2	2	2	2	2	2	2	2	2	2	2	2
Frame 3			3	3	3	3	3	5	5	5	5	4	4
Frame 4					4	4	4	4	4	4	3	3	5
Page Fault	X	X	X		X			X			X	X	X

- **8-Page Faults** for a memory with **4 Frames**!

V.7.4.5. Most Frequently Used (MFU) Page Replacement

Principle: The victim page is the page that is frequently used.

Reason: A page that is not frequently used, but may be in the near future.

Implementation: A field for each entry in the page table indicating the number of times this page has been referenced. The victim page is the one with the highest value.

Memory with 4 Frames

Reference String	1	2	3	1	4	1	2	5	1	2	3	4	5
Frame 1	1	1	1	1	1	1	1	5	5	5	5	5	5
Frame 2		2	2	2	2	2	2	2	1	2	2	2	2
Frame 3			3	3	3	3	3	3	3	3	3	3	3
Frame 4					4	4	4	4	4	4	4	4	4
Page Fault	X	X	X		X			X	X	X			

- **7-Page Faults** for a memory with **4 Frames**!

V.7.4.6. Second-Chance Page Replacement

Principle: This is the FIFO algorithm with the following modifications: a victim page is chosen, if it is referenced it gets a second chance and another is sought.

Reason: To improve FIFO

Implementation: In addition to FIFO, a Reference bit is added, which is set to **1** each time the page is referenced. Once the page has been selected as the victim:

- If this bit is set to zero, the page will be the victim;
- Otherwise , the bit is reset to 0 and another, older page is searched for.
- This is an approximation of LRU

Memory with 4 Frames

Reference String	1	2	3	1	4	1	2	5	1	2	3	4	5
Frame 1	$\underline{1_1^+}$	1_1^+	1_1^+	1_1^+	1_1^+	1_1^+	1_1^+	$\underline{5_1}$	5_1	5_1	5_1^+	$\underline{4_1}$	4_1
Frame 2		$\underline{2_1}$	2_1	2_1	2_1	2_1	2_1	2_0^+	$\underline{1_1}$	1_1	1_1	1_0^+	$\underline{5_1}$
Frame 3			$\underline{3_1}$	3_1	3_1	3_1	3_1	3_0	3_0^+	$\underline{2_1}$	2_1	2_0	2_0^+
Frame 4					$\underline{4_1}$	4_1	4_1	4_0	4_0	4_0^+	$\underline{3_1}$	3_0	3^0
Page Fault	X	X	X		X			X	X	X	X	X	X

- **10-Page Faults** for a memory with **4 Frames**!

V.7.5. Thrashing

After completing initialization, most programs operate on a small number of code and data pages compared to the total memory the program requires. The page's most frequently accessed are called the **working set.**

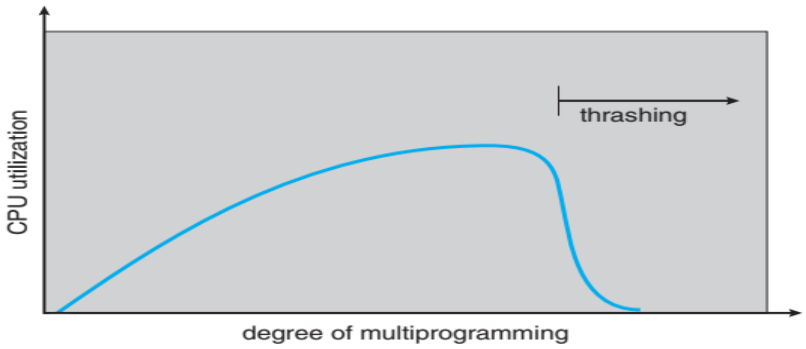


Figure 5.28 : Thrashing.

- When the working set is a small percentage of the system's total number of pages, virtual memory systems work most efficiently and an insignificant amount of computing is spent resolving page faults. As the working set grows, resolving page faults remains manageable until the growth reaches a **critical point**.
- Then faults go up dramatically and the time spent resolving them overwhelms time spent on the computing the program was written to do. This condition is referred to as **thrashing**.
- Consequently, if the processor spends most of its time suspending and resuming processes due to frequent page faults, then a thrashing condition occurs.
- Thrashing is a situation that must be avoided. The chance of thrashing is higher when main memory is not large enough for the number of processes, or active programs.
- Thrashing occurs on a program that works with huge data structures, as its large working set causes continual page faults that drastically slow down the system. Satisfying page faults may require freeing pages that will soon have to be re-read from disk.