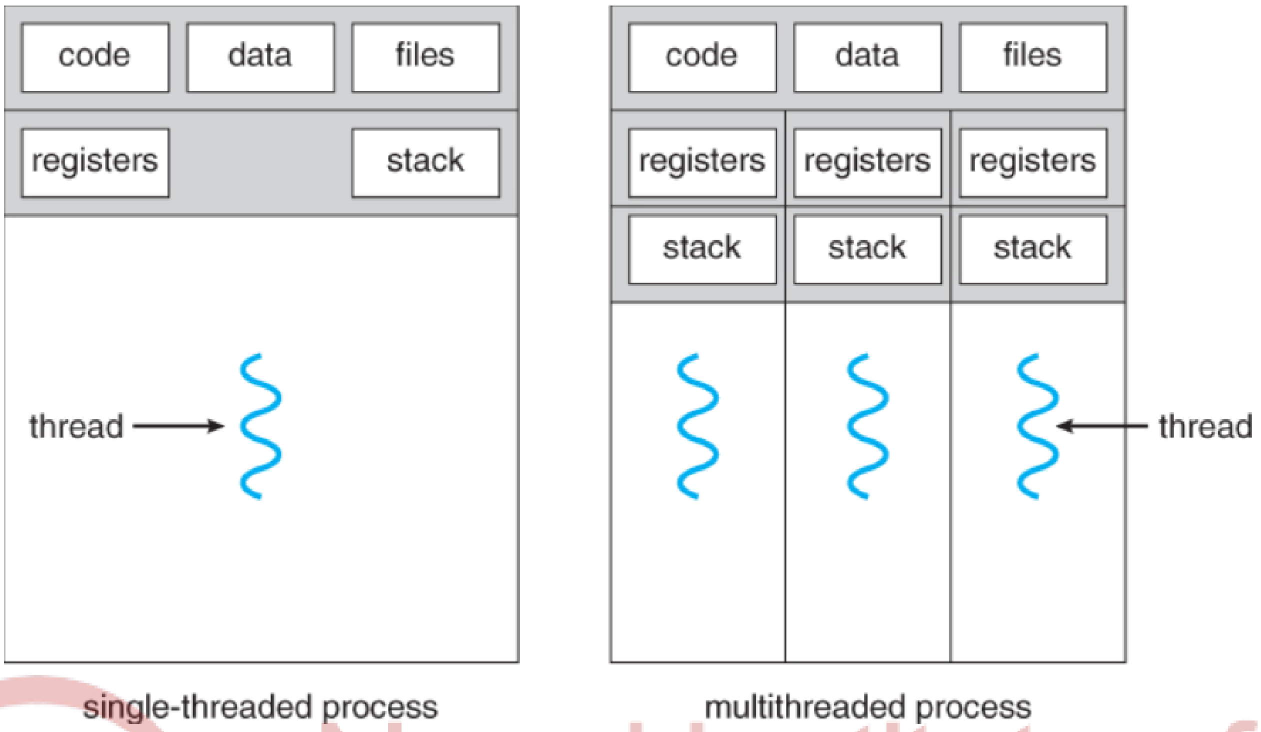
**Process:**

- A process is defined as an entity which represents the basic unit of work to be implemented in the system i.e. a process is a program in execution.
- The execution of a process must progress in a sequential fashion.

| **Program** | **Process** |
|---|---|
| 1. It is a set of instructions. | 1. It is a program in execution. |
| 2. It is a passive entity. | 2. It is an active entity. |
| 3. It has longer lifespan. | 3. It has limited lifespan. |
| 4. A program needs memory space on disk to store all instructions. | 4. A process contains many resources like a memory address, disk, printer, etc. |
| 5. Program is loaded into secondary storage device. | 5. Process is loaded into main memory. |
| 6. It is a static object existing in a file form. | 6. It is a dynamic object (i.e., program in execution) |
| 7. No such duplication is needed. | 7. New processes require duplication of parent process. |

**Threads:**

- Thread is a single sequential flow of execution of tasks of a process. It is also known as thread of execution or thread of control.
- It is the fundamental unit of CPU utilization consisting of a program counter, stack and set of registers.
- Threads have some of the properties of processes, they are sometimes called lightweight processes.
- Each thread belongs to exactly one process and outside a process no threads exist.
- The process can be split down into so many threads. **For example**, in a browser, many tabs can be viewed as threads. MS Word uses many threads - formatting text from one thread, processing input from another thread, etc.
- Like a traditional process i.e., process with one thread, a thread can be in any of several states (Running, Blocked, Ready or terminated).



**Similarities between process and thread**

- Like processes, threads share CPU and only one thread active (running) at a time.
- Like processes, threads within processes execute sequentially.
- Like processes, thread can create children.
- Like process, if one thread is blocked, another thread can run

**Difference between Process and thread**

| **Process** | **Thread** |
|---|---|
| 1. heavy weight | 1. light weight |
| 2. It takes more time to terminate. | 2. It takes less time to terminate. |
| 3. More resources are required. | 3. Less resources are required. |
| 4. More time required for creation. | 4. Less time required for creation. |
| 5. System call is involved in process. | 5. System call not involved in threads. |
| 6. If one server process is blocked then other server processes cannot execute until the first process is unblocked. | 6. If one thread is blocked and waiting then the second thread in the same task can run. |
| 7. Slow Execution | 7. Fast Execution |
| 8. Doesn't share memory (loosely coupled) | 8. Shares memories and files (tightly coupled) |
| 9. Takes more time to switch between processes. | 9. Takes less time to switch between threads. |

**Principle of concurrency**

- Concurrency is the execution of a set of multiple instruction sequences at the same time. This occurs when there are several process threads running in parallel.
- Process that coexist on the memory at a given time are called concurrent process. The concurrent process may either be independent or cooperating.
- Today's technology, like multi-core processors and parallel processing, allows multiple processes and threads to be executed simultaneously. Multiple processes and threads can access the same memory space, the same declared variable in code, or even read or write to the same file.
- The amount of time it takes for a process to execute is not easily calculated, so we are unable to predict which process will complete first, thereby allowing us to implement algorithms to deal with the issues that concurrency creates. The amount of time a process takes to complete depends on the following:
  - The activities of other processes
  - The way operating system handles interrupts
  - The scheduling policies of the operating system

**Problems in Concurrency:**

- **Sharing global resources:** Sharing of global resources safely is difficult. If two processes both make use of a global variable and both make changes to the variables value, then the order in which various changes take place are executed is critical.
- **Optimal allocation of resources:** It is difficult for the operating system to manage the allocation of resources optimally.
- **Locating programming errors:** It's difficult to spot a programming error because reports are usually repeatable due to the varying states of shared components each time the code is executed.
- **Locking the channel:** It may be inefficient for the operating system to simply lock the resource and prevent its use by other processes.

**Race Condition**

- A race condition is an undesirable situation that occurs when a device or system attempts to perform two or more operations at the same time, but because of the nature of the device or system, the operations must be done in the proper sequence to be done correctly.
- When two or more processes are reading or writing some shared data and the final result depends on who runs precisely when, are called race conditions.

**Example:**

- Consider two cooperating processes P1 and P2 that updates the balance of account in a bank.

| **Process P1** | **Process P2** |
|---|---|
| Read Balance | Read Balance |

| | |
|---|---|
| Balance= Balance + 1000 | Balance= Balance -400 |

**Table: Code Segment for processes P1 and P2**

- Suppose that the balance is initially 5000, then after the execution of both P1 and P2, it should be 5600. The correct result is achieved if P1 and P2 execute one by one in any order either P1 is followed by P2 or P2 followed by P1.
- However if the instructions of P1 and P2 are interleaved arbitrarily, the balance may not be 5600 after the execution of both P1 and P2. One possible interleaving sequence for the executions of instructions of P1 and P2 is given in below table.

| **Process P1** | **Process P2** | **Balance** |
|---|---|---|
| Read balance | | 5000 |
| | Read balance | 5000 |
| Balance= Balance + 1000 | | 6000 |
| | Balance = Balance -400 | 4600 |

- The above interleaved sequence results in an inconsistent balance that is 4600. If the order of the last two instructions is interchanged, the balance would be 6000 (again, inconsistent).

Note that race condition is a term for a situation where several processes sharing some data execute concurrently, and where the result of execution depends on the order in which the shared data is accessed by the processes.

**Mutual Exclusion:**

- To avoid race conditions, we need to ensure mutual exclusion. That means if a process P1 is manipulating shared data, no other cooperating process should be allowed to manipulate it until P1 finishes with it. In the above example, since mutual exclusion was not ensured while accessing the shared data, it yielded an inconsistent result.
- Mutual exclusion is the way of making sure that if one process is executing critical section, other processes will be excluded from executing critical section.
- It is the responsibility of operating system to assure that no more than one process is in its critical section simultaneously.

**Critical Section Problem:**

- In concurrent programming, concurrent accesses to shared resources can lead to unexpected or erroneous behavior, so parts of the program where the shared resource is accessed need to be protected in ways that avoid the concurrent access. This protected section is the critical section or critical region. It cannot be executed by more than one process at a time.

**Four conditions to avoid critical region problem:**

1. No two processes may be simultaneously inside their critical regions.
2. No assumptions may be made about speeds or the number of CPUs.
3. No process running outside its critical region may block other processes.
4. No process should have to wait forever to enter its critical region.

**Example:**

Here process A enters its critical region at time T1. A little later, at time T2 process B attempts to enter its critical region but fails because another process is already in its critical region and we allow only one at a time. Consequently, B is temporarily suspended until time T3 when A leaves its critical region, allowing B to enter immediately. Eventually B leaves (at T4) and we are back to the original situation with no processes in their critical regions.

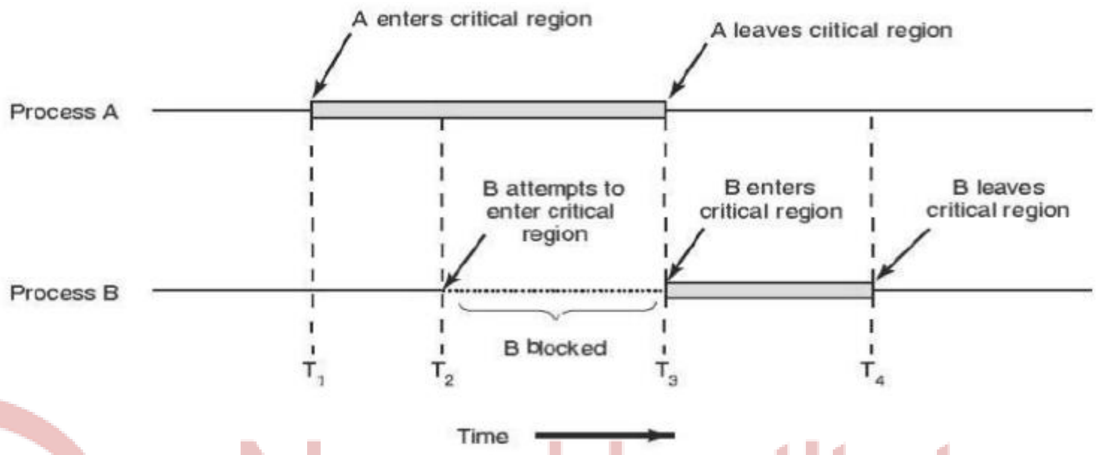


**Figure 1. Mutual exclusion using critical regions.**

**Semaphores**

- Semaphore is a simply a variable. This variable is used to solve critical section problem and to achieve process synchronization in the multiprocessing environment.
- It was proposed by Edsger Dijkstra. It is a technique to manage concurrent processes by using a simple integer value known as semaphore.
- The two most common kinds of semaphores are counting semaphores and binary semaphores. Counting semaphore can take non-negative integer values and Binary semaphore can take the value 0 & 1 only.
- A semaphore can only be accessed using the following operations:
  - **wait ()** :- called when a process wants access to a resource
  - **signal ():-** called when a process is done using a resource

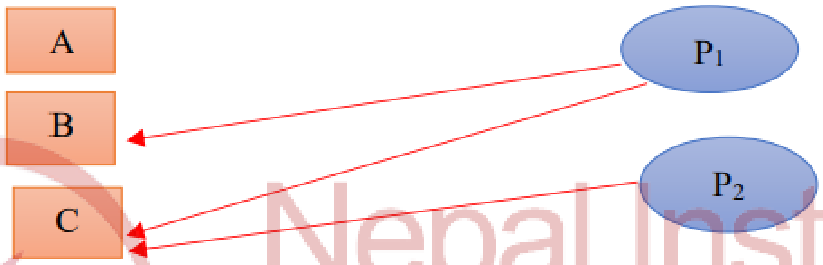
**Mutex:**

- When the semaphore's ability to count is not needed, a simplified version of the semaphore, called a mutex, is sometimes used.
- Mutexes are good only for managing mutual exclusion to some shared resource or piece of code. They are easy and efficient to implement, which makes them especially useful in thread packages that are implemented entirely in user space.
- A mutex is a variable that can be in one of two states: unlocked or locked.

- Consequently, only 1 bit is required to represent it, but in practice an integer often is used, with 0 meaning unlocked and all other values meaning locked.
- Two procedures are used with mutexes.
- When a thread (or process) needs access to a critical region, it calls mutex-lock. If the mutex is currently unlocked (meaning that the critical region is available), the call succeeds and the calling thread is free to enter the critical region.
- On the other hand, if the mutex is already locked, the calling thread is blocked until the thread in the critical region is finished and calls mutex-unlock. If multiple threads are blocked on the mutex, one of them is chosen at random and allowed to acquire the lock.

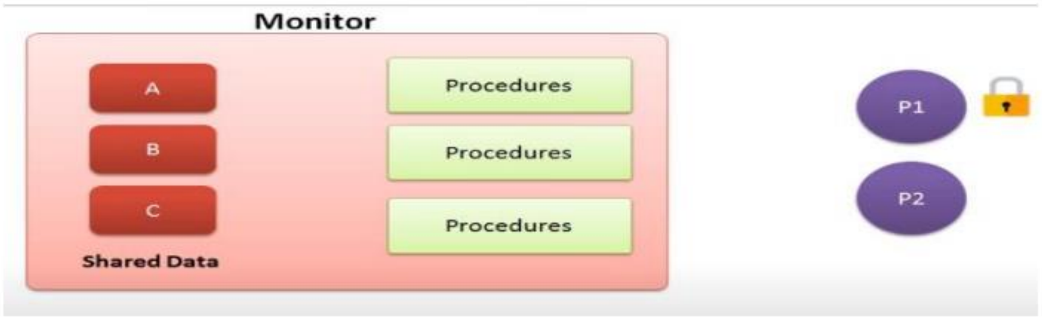
**Monitor:**

- When multiple process access to shared data simultaneously, it leads to the problem of Race Condition.



- Monitor is programming language construct that control access to shared data.
- It is a collection of procedures, variables, and data structures that are all grouped together in a special kind of module or package.
- Monitor is a module that encapsulate:
  - Shared data structures.
    - Procedures that operates on shared data structure and
  - Synchronization between concurrent procedure invocations.
- In Monitor when the process wants to access shared data, they cannot access directly. Process have to call procedure and those procedure in turn will allow access to shared data.
- Only one process can enter into monitor at a time.

**Example:**



- Here when process P1 calls the procedure in order to access shared data, at first it is checked if any other process is in monitor or not.
- If no other process is in the monitor then P1 acquires lock and enters the monitor.

- When process P2 calls the procedure, then it gets block and will be placed in the queue of monitor.
- In monitor Conditional variable provides the synchronization between the processes.
- Three operations can be performed on conditional variables:
  - Wait ()
  - Signal ()
  - Broadcast ()
- **wait ():** When a process call wait () operation, that process is placed on a queue to enter to the monitor.
- **signal ():** When a process wants to exit from monitor it calls signal () operation.
- When a process calls a signal () operation, it causes one of the waiting processes in the queue to enter monitor.
- **broadcast ():** It signals all the waiting processes in the queue to wait.

| Monitors | Semaphore |
|---|---|
| We can use condition variables only in the monitors. | In semaphore, we can use condition variables anywhere in the program, but we cannot use conditions variables in a semaphore. |
| In monitors, wait always block the caller. | In semaphore, wait does not always block the caller. |
| The monitors are comprised of the shared variables and the procedures which operate the shared variable. | The semaphore S value means the number of shared resources that are present in the system. |
| Condition variables are present in the monitor. | Condition variables are not present in the semaphore. |

## Classical Problems of Synchronization

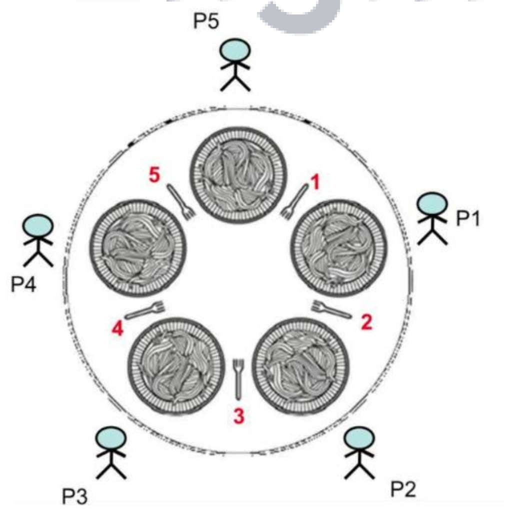
### Producer Consumer Problem

- The Producer-Consumer problem is a classical problem. It is used for multi-process synchronization, which means synchronization between more than one processes.
- In this problem, we have one producer and one consumer. The producer is the one who produces something, and the consumer is the one who consumes something, produced by the producer. The producer and consumer both share the common memory buffer, and the memory buffer is of fixed-size.
- Let us consider the producer-consumer problem where two processes share a common, fixed-size buffer.

- One of them, the producer, puts information into the buffer, and the other one, the consumer, takes it out.
- Trouble arises when the producer wants to put a new item in the buffer, but it is already full. The solution is for the producer to go to sleep, to be awakened when the consumer has removed one or more items.
- Similarly, if the consumer wants to remove an item from the buffer and sees that the buffer is empty, it goes to sleep until the producer puts something in the buffer and wakes it up.
- This approach sounds simple enough, but it leads to the same kinds of race conditions.
- To keep track of the number of items in the buffer, we will need a variable, count. If the maximum number of items the buffer can hold is N, then:
  - The producer's code will first test to see if count is N. If it is, the producer will go to sleep; if it is not, the producer will add an item and increment count.
  - Similarly, the consumer's code first test count to see if it is 0. If it is, go to sleep; if it is nonzero, remove an item and decrement the counter. Each of the processes also tests to see if the other should be awakened, and if so, wakes it up.

**Dining Philosophers Problem**

- In 1965, Dijkstra posed and solved a synchronization problem he called the dining philosophers problem.
- The problem can be stated quite simply as follows:
  - Five philosophers are seated around a circular table.
  - Each philosopher has a plate of spaghetti.
  - The spaghetti is so slippery that a philosopher needs two forks to eat it. Between each pair of plates is one fork.



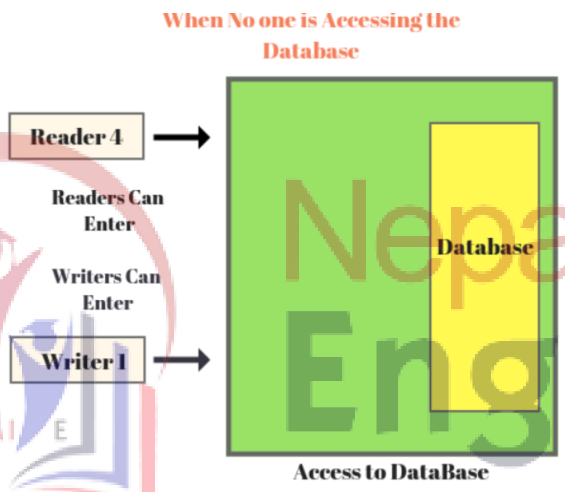
- The life of a philosopher consists of alternate periods of eating and thinking.
- When a philosopher gets hungry, she tries to acquire her left and right forks, one at a time, in either order. If successful in acquiring two forks, she eats for a while, then puts down the forks, and continues to think.

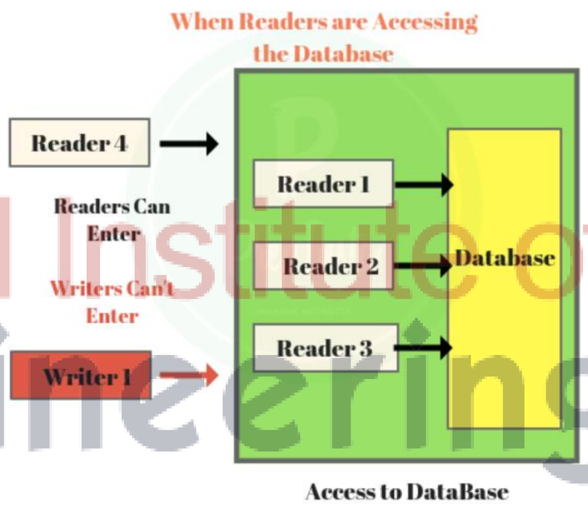
**Reader writer problems**

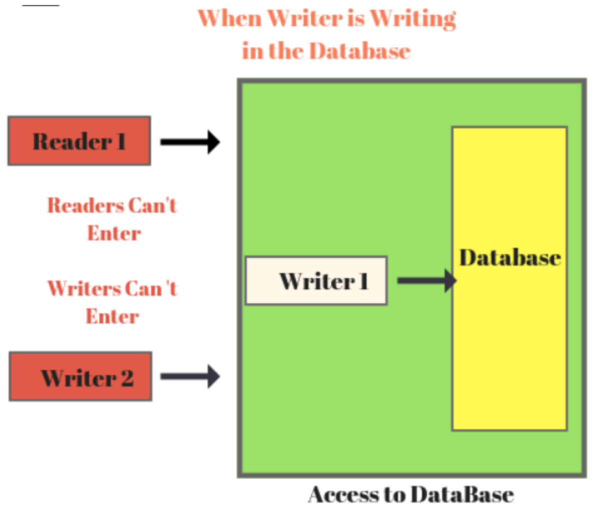
- Consider a situation where we have a file shared between many people.
  - If one of the people tries editing the file, no other person should be reading or writing at the same time, otherwise changes will not be visible to him/her.
  - However if some person is reading the file, then others may read it at the same time.
  - Precisely in OS we call this situation as the **readers-writers problem.**

**Problem parameters:**

- One set of data is shared among a number of processes
- Once a writer is ready, it pe rforms its write. Only one writer may write at a time
- If a process is writing, no other process can read it
- If at least one reader is reading, no other process can write
- Readers may not write and only read







**Sleeping Barber Problem**

- Dijkstra introduced the sleeping barber problem in 1965. This problem is based on a hypothetical scenario where there is a barbershop with one barber. The barbershop is divided into two rooms, the waiting room, and the workroom. The waiting room has n chairs for waiting customers, and the workroom only has a barber chair.
- Now, if there is no customer, then the barber sleeps in his own chair(barber chair). Whenever a customer arrives, he has to wake up the barber to get his haircut. If there are multiple customers and the barber is cutting a customer's hair, then the remaining customers wait in the waiting room with "n" chairs(if there are empty chairs), or they leave if there are no empty chairs in the waiting room.
- The sleeping barber problem may lead to a race condition. This problem has occurred because of the actions of both barber and customer.

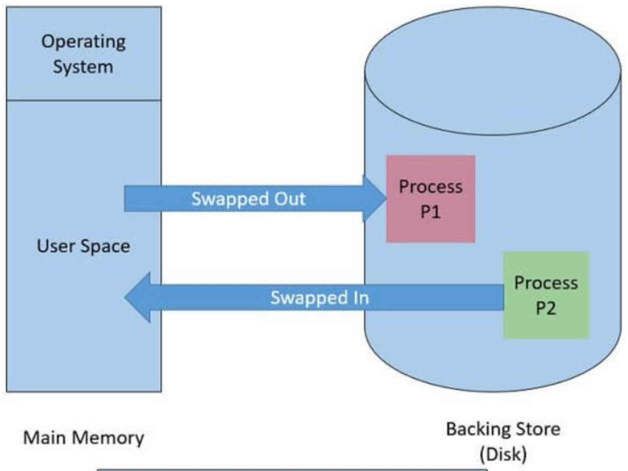
**Example to explain the problem:**

- Suppose a customer arrives and notices that the barber is busy cutting the hair of another customer, so he goes to the waiting room. While he is on his way to the waiting room, the barber finishes his job and sees the waiting room for other customers. But he (the barber) finds no one in the waiting room (as the customer has yet not arrived in the waiting room), so he sits down in his chair (barber chair) and sleeps. Now the barber is waiting for new customers to wake him up, and the customer is waiting as he thinks the barber is busy.
- Here, both of them are waiting for each other, which leads to race conditions.

Solution of sleeping barber problem
The following solution uses three semaphores, one for customers (for counts of waiting for customers), one for barber (a binary semaphore denoting the state of the barber, i.e., 0 for idle and 1 for busy), and a mutual exclusion semaphore, mutex for seats.

**Swapping:**

- Any process which needs to be executed should be kept in main memory.
- However, if the process in the main memory is not getting executed for some reason such as waiting for an event to occur or waiting for an IO, then such process can be moved back to backing store (swap out) and any process in the backing store which is ready to be executed can be loaded into the main memory (swap in).
- This process of swapping-out and swapping-in of processes between main memory and backing store in order to reduce the CPU idle time is called as swapping.
- The phenomenon of moving process from main memory to secondary memory is called **swapping out**. While the phenomenon of moving process from secondary memory back to primary memory is called **swapping in**.



Swapping of Processes

**Advantages of Swapping:**

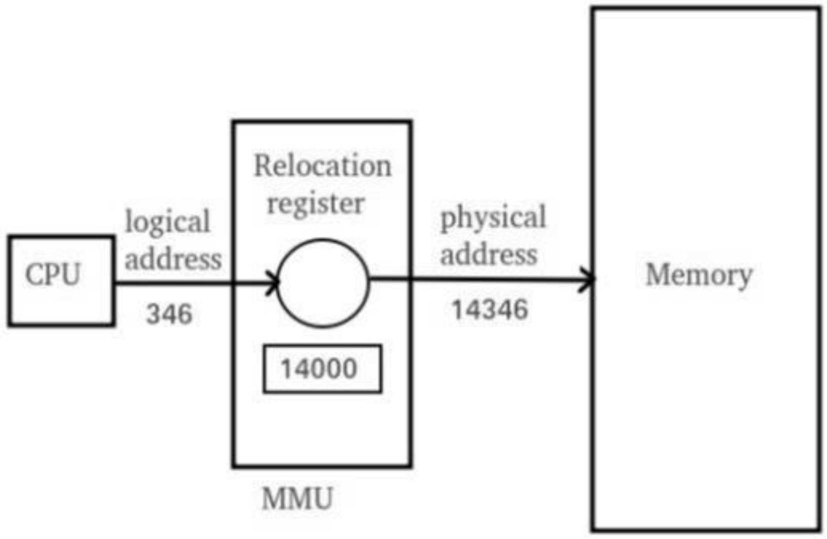
- Provides higher degree of multi programming, dynamic relocation greater memory utilization, less wastage of CPU time, superior performance and priority-based scheduling.

**Problems with swapping**

- Swapping with variable sized partitions creates holes.
- ‘Holes’ can be simply defined as areas of unused memory that exist between partitions. These holes are usually too small and too scattered to be of any use at all. Their size is often the criteria for not being able to hold programs that are too large to fit into it.

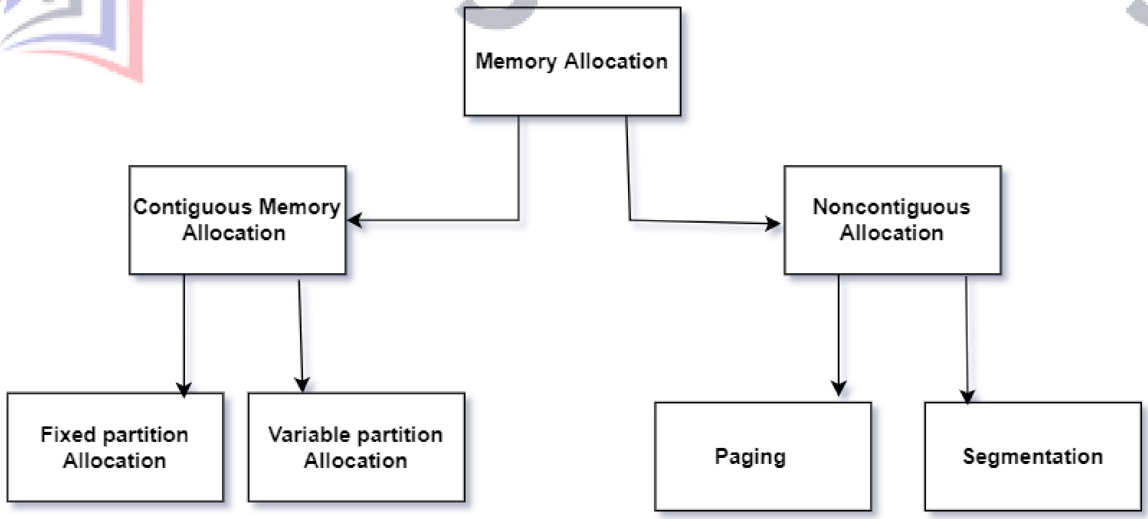
| Logical Address | Physical Address |
|---|---|
| 1. Address generated by CPU during program execution. | 1. refers to the physical location in the memory unit. |
| 2. User programs deals with logical address directly. | 2. User program never sees the physical address. |
| 3. It doesn’t exist physically and also called virtual address. | 3. It is a real location that exists in the memory unit. |
| 4. Logical address is erased when the system is rebooted. | 4. Physical address is not affected when the system is rebooted. |

**Mapping Virtual Addresses to Physical Addresses**



- The CPU generates the logical address (here, 346).
- The MMU will generate the base address (here, 14000) which is stored in the Relocation Register.
- The value of Relocation Register (here, 14000) is added to the logical address to get the physical address i.e. 14000+346= 14346 (Physical Address).

**Memory Allocation Techniques**


Memory Allocation
Contiguous Memory Allocation
Noncontiguous Allocation
Fixed partition Allocation
Variable partition Allocation
Paging
Segmentation

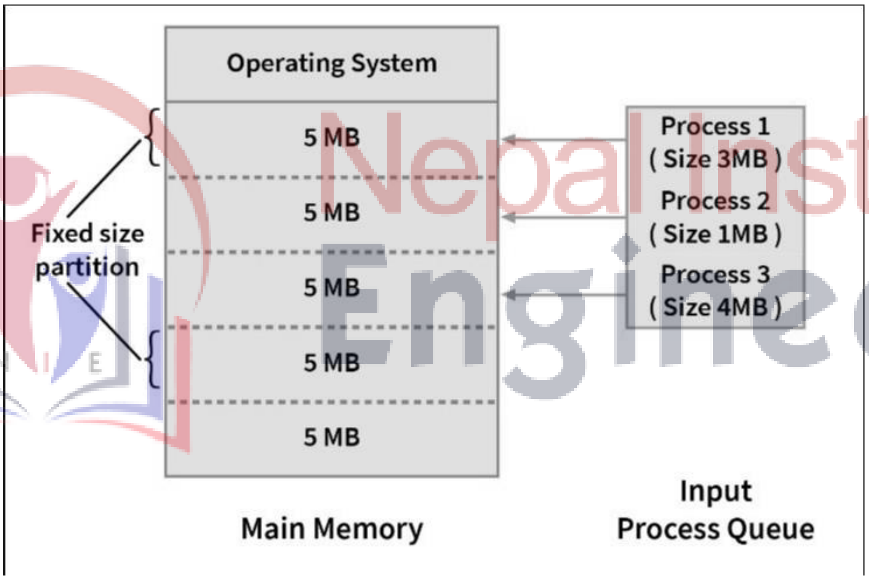
**Contiguous Memory Allocation**

- Contiguous memory allocation is a classical memory allocation model that assigns a process consecutive memory block.

- As the name implies, we allocate contiguous blocks of memory to each process using this technique. So, whenever a process wants to enter the main memory, we allocate a continuous segment from the totally empty space to the process based on its size.
- The various memory management scheme that are based in this approach are
    i. Fixed length Partitioning
    ii. Variable length partitioning

**i. Fixed length Partitioning:**

- In this type of contiguous memory allocation technique, each process is allotted a fixed size continuous block in the main memory.
- That means there will be continuous blocks of fixed size into which the complete memory will be divided, and each time a process comes in, it will be allotted one of the free blocks.
- Because irrespective of the size of the process, each is allotted a block of the same size memory space.
- This technique is also called **static partitioning**.



- In the diagram above, we have 3 processes in the input queue that have to be allotted space in the memory.
- As we are following the fixed size partition technique, the memory has fixed-sized blocks.
- The first process, which is of size 3MB is also allotted a 5MB block, and the second process, which is of size 1MB, is also allotted a 5MB block, and the 4MB process is also allotted a 5MB block. So, the process size doesn't matter. Each is allotted the same fixed-size memory block.

*Note: The technique of having fixed partitions is no longer in use.*

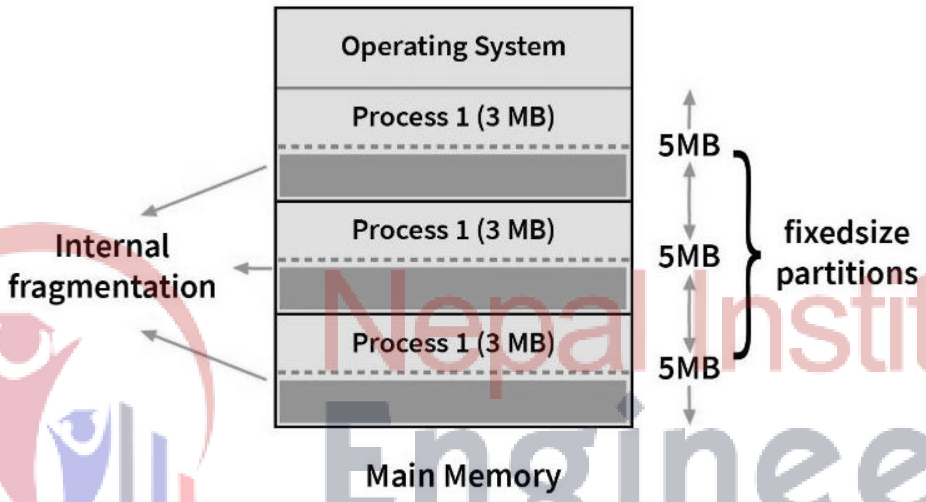
**Advantages of fixed length partitioning:**

- Easy to implement. It simply requires putting a process into certain partition without focusing on the emergence of Internal and External Fragmentation.

- It is easy to keep track of how many blocks of memory are left, which in turn decides how many more processes can be given space in the memory.

**Disadvantages of fixed length partitioning:**

- As the size of the blocks is fixed, we will not be able to allot space to a process that has a greater size than the block.
- **Internal fragmentation**:
  - The situation when there is wasted space inside a partition when the amount of data allocated in that partition is smaller than the partition size.
  - For e.g.: the size of partition is 8 Mbytes and the process is of 2 Mbytes but still it is given the whole 8 Mbytes block in this technique.
  - It leads to space wastage.

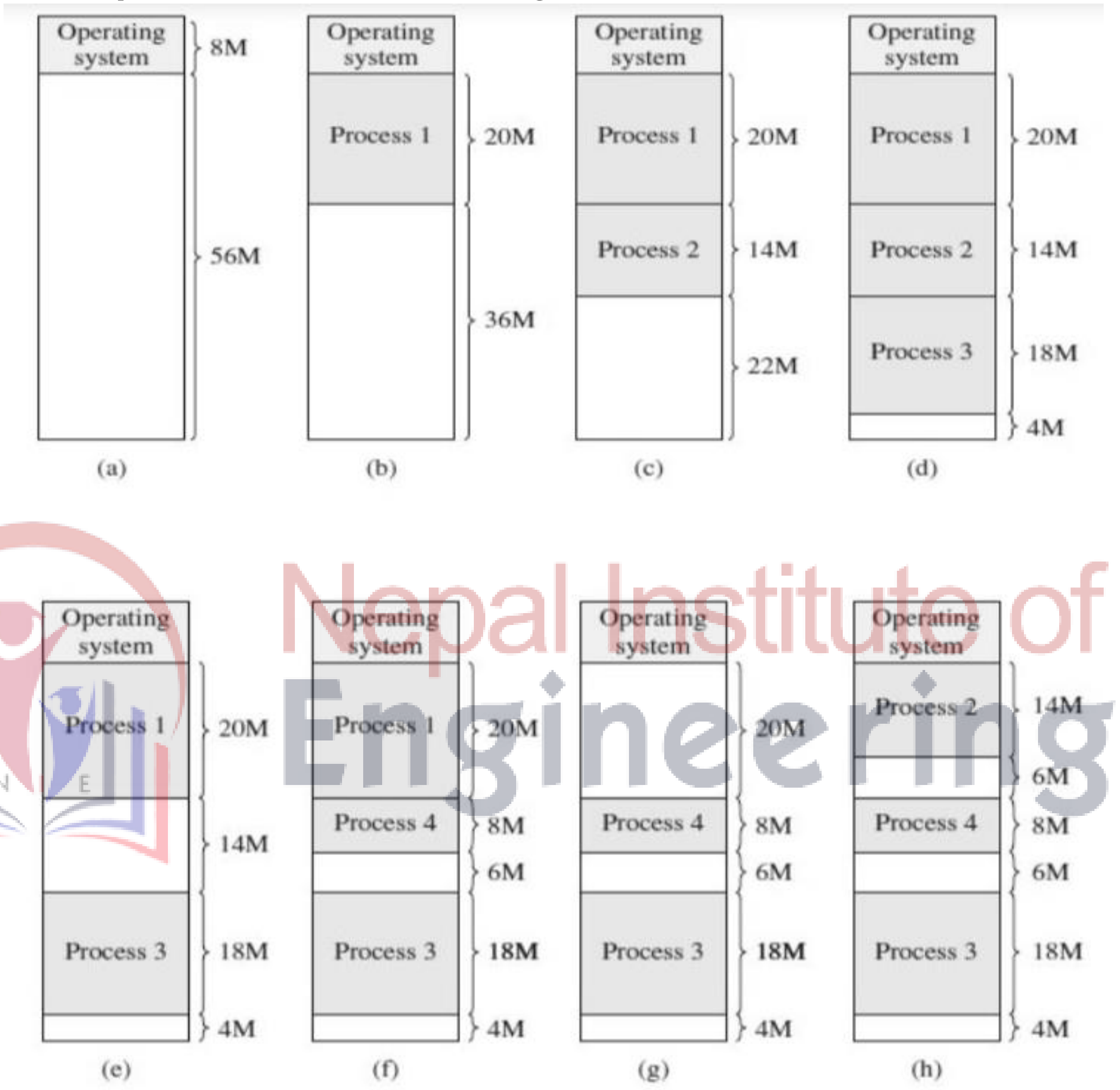


**ii. Variable length partitioning:**

- To overcome some of the difficulties with fixed partitioning, an approach known as dynamic or variable partitioning was developed.
- In this type of contiguous memory allocation technique, no fixed blocks or partitions are made in the memory.
- Instead, each process is allotted a variable-sized block depending upon its requirements. That means, whenever a new process wants some space in the memory, if available, this amount of space is allotted to it.
- Hence, the size of each block depends on the size and requirements of the process which occupies it.

- Initially, when there is no process in memory, the whole memory is available for allocation and is considered as single large partition (a hole).
- Whenever process request for memory, the hole large enough to accommodate that process is allocated. The rest of memory is available to other process. As soon as process terminates, the memory occupied by it is deallocated and can be used for other process.
- With dynamic partitioning, the partitions are of variable length and number according to incoming process and Main Memory’s size.

- When a process is brought into main memory, it is allocated exactly as much memory as it requires and no more as shown in the figure below.



**Advantages of Variable length partition**

**1. No Internal Fragmentation:**

- In variable Partitioning, space in main memory is allocated strictly according to the need of process, hence there is no case of internal fragmentation. There will be no unused space left in the partition.

**2. No restriction on Degree of Multiprogramming:**

- More number of processes can be accommodated due to absence of internal fragmentation. A process can be loaded until the memory is full.

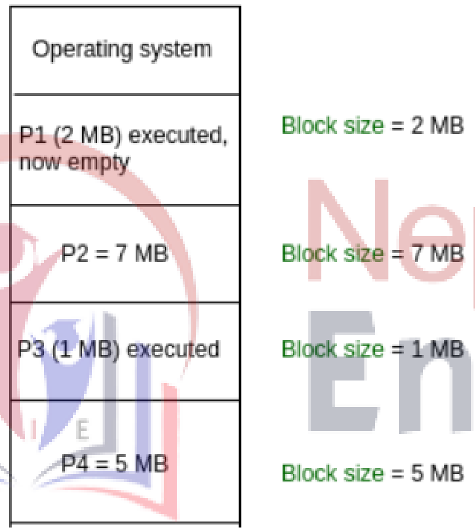
**3. No Limitation on the size of the process:**

- In Fixed partitioning, the process with the size greater than the size of the largest partition could not be loaded and process cannot be divided as it is invalid in contiguous allocation technique. Here, in variable partitioning, the process size can't be restricted since the partition size is decided according to the process size.

**Disadvantages of Variable length**

- It is difficult to keep track of processes and the remaining space in the memory.

- **External Fragmentation (checker boarding):**
    - A form of fragmentation that arises when there is enough memory available to allocate for the process but that available memory is not contiguous.

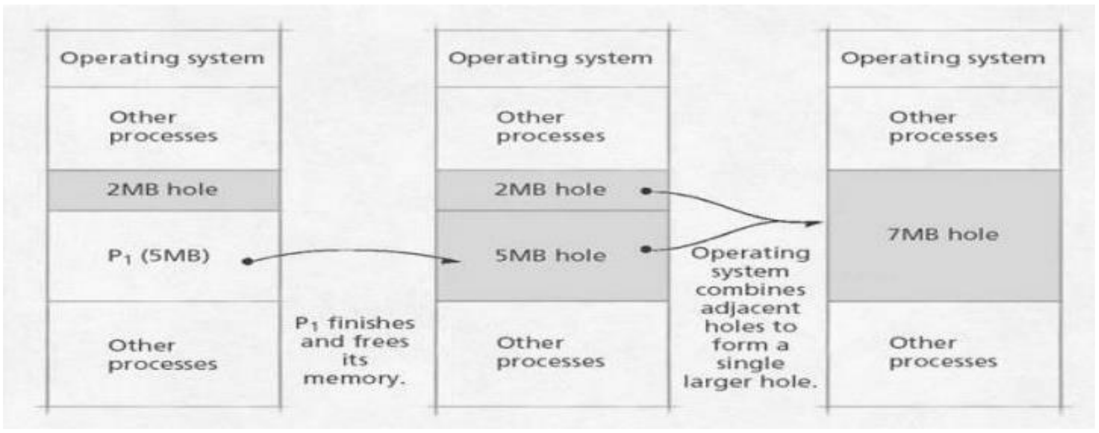


- Here, if P5 with size 3 MB arrives, it cannot be allocated though there is enough space because free memory is scattered and not available as single unit.

**Solution to Fragmentation**

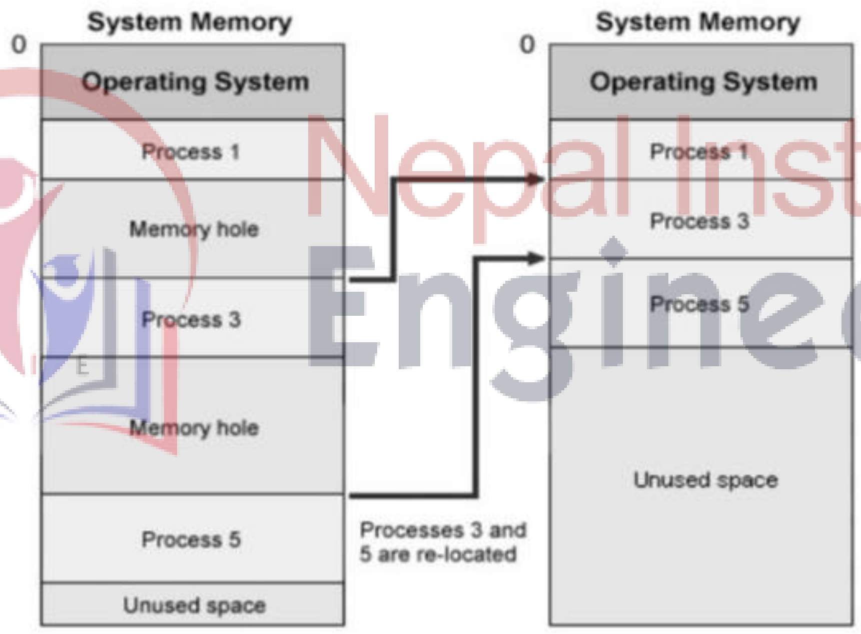
**Coalescing:**

- Coalescing is a part of memory management in which two adjacent free blocks of computer memory are merged.



**Compaction**:

- Compaction is the process of combining of holes together so that they can be big enough to accommodate a process. This can be referred to as memory compaction.
- For the very reason that memory compaction is a time-consuming process



**Partition Selection Algorithms**

- Whenever a process arrives and there are various holes enough to accommodate it, the operating system may use one of the following algorithms to select a partition for the process.

**1. First Fit Algorithm**:

- In this algorithm, the OS scans the free storage list and allocates the first hole that is large enough to accommodate that process.
- This algorithm is fast because very little search is involved, compared to other algorithms.

**2. Best Fit Algorithm:**

- In this algorithm, the OS scans the free storage list and allocates the smallest hole whose size is larger than or equal to, the size of process.

- It is slower than first fit algorithm.
- Best fit algorithm is very effective in reducing wastage of memory.

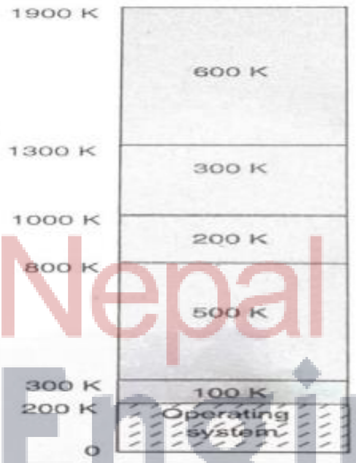
**3. Worst fit:**

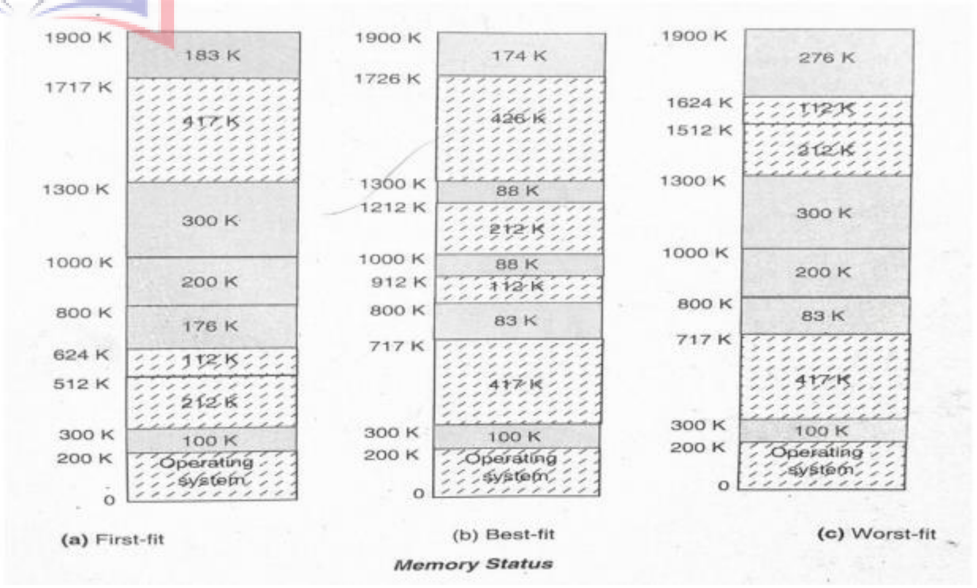
- In this algorithm OS scans the entire free storage list and allocates the largest hole to that process.
- Worst fit allocation is not very effective in reducing the wastage of memory.

**Example:**

1) Given memory partitions of 100K, 500K, 200K, 300K and 600K in order. How would each of the First Fit, Best Fit, and Worst Fit algorithms place the processes of 212K, 417K, 112K and 426K in order? Which algorithm makes the most efficient use of memory? Show the diagram of memory status in each case. Assume that OS resides in lower part of memory and occupies 200K.

**<u>Solution:</u>**





- In first fit process with size 426K has to wait.
- In best Fit all the processes are accommodated.

- In worst fit the process with size 426K has to wait.

1. First-fit:
   - 212K is put in 500K partition
   - 417K is put in 600K partition
   - 112K is put in 288K partition (new partition 288K = 500K - 212K)
   - 426K must wait

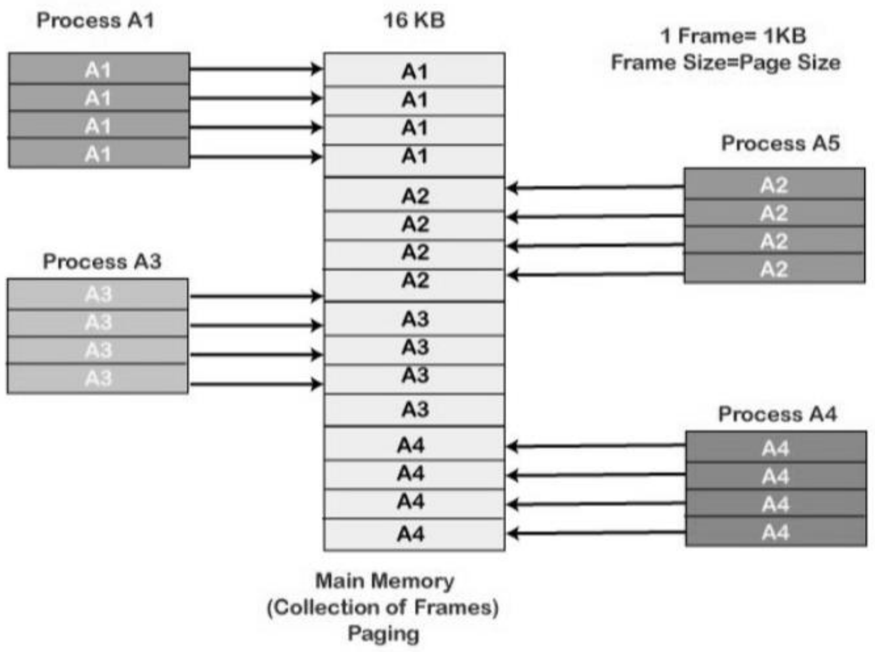
2. Best-fit:
   - 212K is put in 300K partition
   - 417K is put in 500K partition
   - 112K is put in 200K partition
   - 426K is put in 600K partition
3. Worst-fit:
   - 212K is put in 600K partition
   - 417K is put in 500K partition
   - 112K is put in 388K partition
   - 426K must wait

**Non-Contiguous Memory Allocation**

- It is the allocation approach where the parts of single process can occupy non-contiguous physical addresses.
- It allows a process to acquire the several memory blocks at the different location in the memory according to its requirement.
- It reduces the memory wastage caused due to internal and external fragmentation.
- In the non-contiguous memory allocation, a process will acquire the memory space but it is not at one place it is at the different locations according to the process requirement.
- The various memory management scheme based on con-contiguous allocation are:

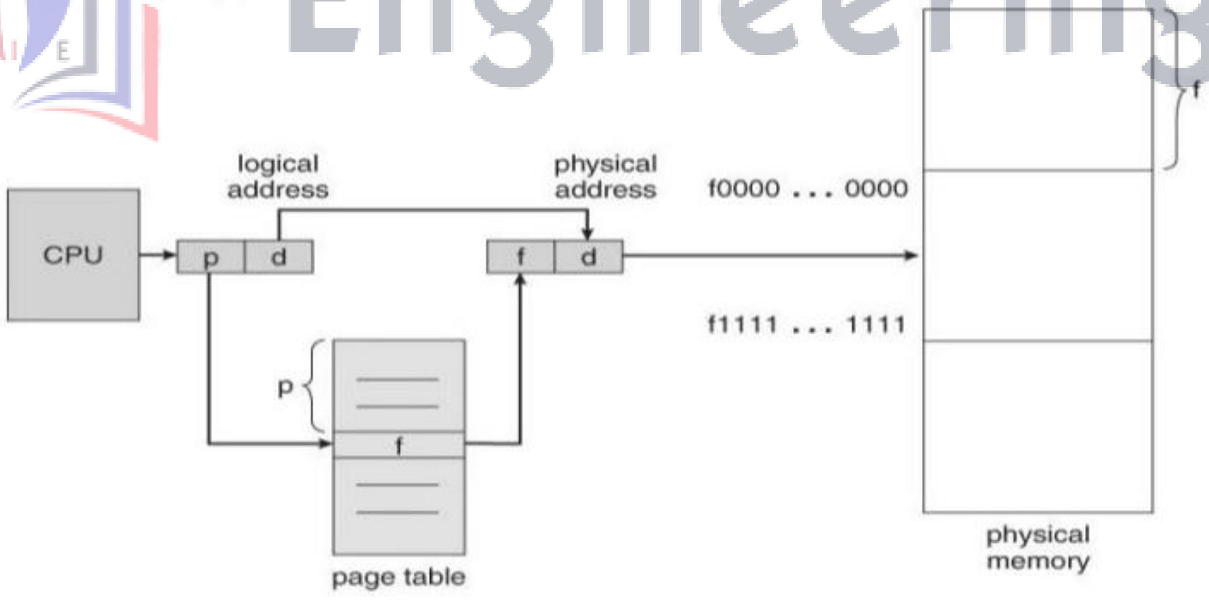
**1. Paging:**

- In paging, the physical memory is divided into fixed sized blocks called **page frame** and logical address is also divided into fixed size blocks called **pages** which are of same size as page frame.
- The size of a frame should be kept the same as that of a page to have maximum utilization of the main memory and to avoid external fragmentation.
- When a process is to be executed, its page can be loaded into any allocated frames (not necessarily contiguous) from the disk.



**Principal of operation**

- In paging, mapping of logical address to physical address is performed at page level.
- When CPU generates a logical address, it is divided into two parts:
  - A page number (P) and
  - Page offsets (d)



- Page address is called logical address and represented by page number and the offset. **Logical address=Page number + page offset**
- Frame address is called physical address and represented by a frame number and the offset. **Physical address = Frame number + page offset**

- Address translation is done using page table. OS maintains a page table for each process to keep track of which page frame is allocated to which page.
- Page table stores frame number allocated to each page and page number is used as index to page table.
- When CPU generates a logical address, that address is send to MMU (Memory Management Unit).
- The MMU uses the page number to find corresponding page frame number in page table.
- That page frame number is attached to higher-order end of page offset to generate physical address.

**Segmentation:**

- It is just like the Paging technique except the fact that in segmentation, the segments are of variable length but, in Paging, the pages are of fixed size.
- In segmentation, the memory is split into variable-length parts. Each part is known as segments.
- The information which is related to the segment is stored in a table which is called a segment table.
- Like paging, a logical address using segmentation consists of two parts:

a) Segment number (s).

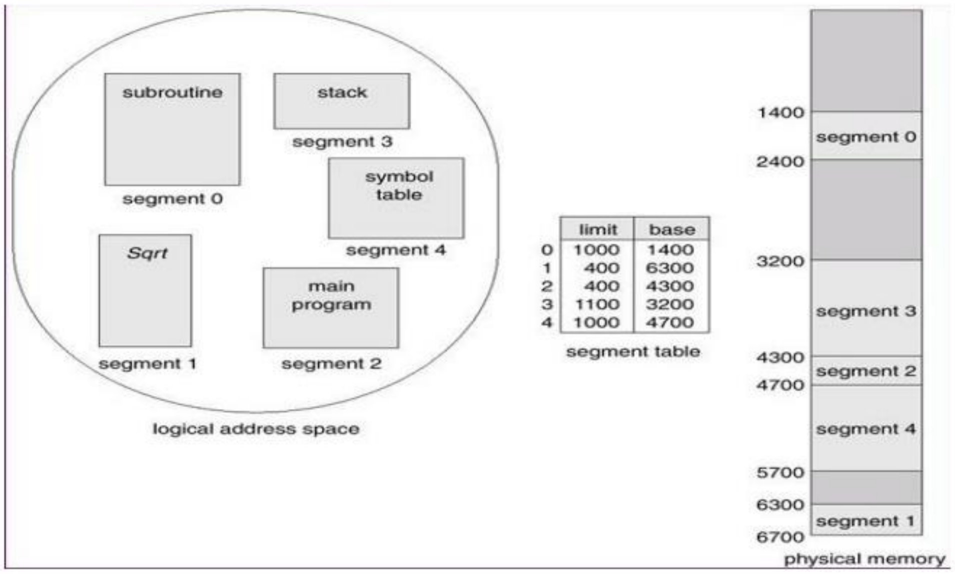
b) Offset (d).

- Because of the use of unequal size segments, segmentation is similar to dynamic partitioning.
- The difference is that the segments of a same program need not be contiguous
- Segmentation eliminates internal fragmentation but like dynamic partitioning, it suffers from external fragmentation.
- To keep tracks of each segment, a segment table is maintained by OS.
- Each entry in segment table consists of two fields:

a) **Segment base**: - specifies the starting address of segment in physical memory and

b) **Segment limit**: - specifies the length of segment.

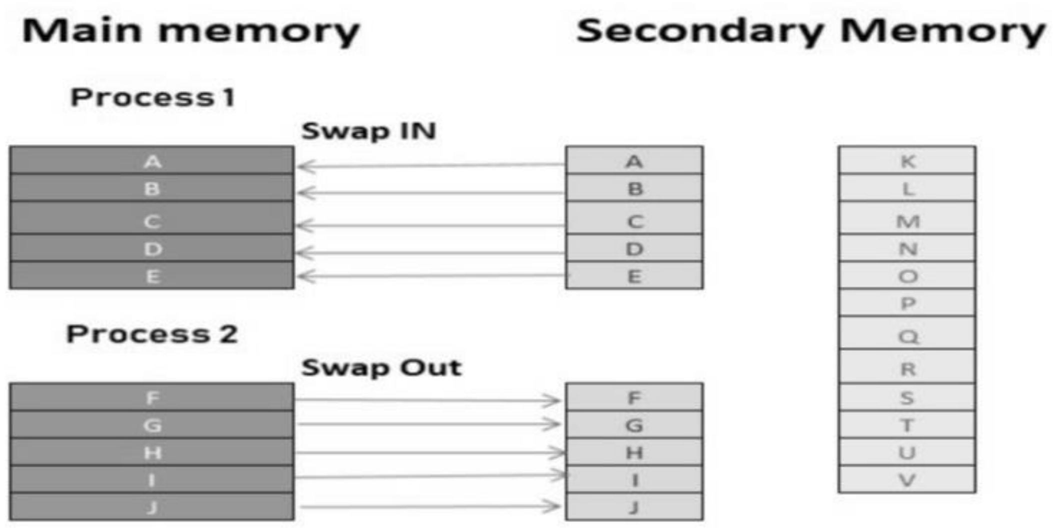
- The segment number is used as an index to the segment table.



- When CPU generates logical address, the address is sent to MMU.
- The MMU uses segment number of logical address as an index to the segment table.
- The offset is compared with segment limit and if it is greater, an invalid error is generated, otherwise the offset is added to the segment base to form the physical address that is sent to memory.

**Demand Paging**

- A demand paging mechanism is very much similar to a paging system with swapping where processes stored in the secondary memory and pages are loaded only on demand, not in advance.
- So, when a context switch occurs, the OS never copy any of the old program's pages from the disk or any of the new program's pages into the main memory. Instead, it will start executing the new program after loading the first page and fetches the program's pages, which are referenced.
- During the program execution, if the program references a page that may not be available in the main memory because it was swapped, then the processor considers it as an invalid memory reference. That's because the page fault and transfers send control back from the program to the OS, which demands to store page back into the memory.



**Advantages of Demand Paging**

- It reduces swap time, since only the required pages age swapped in instead of swapping whole process.
- It increases degree of multiprogramming by reducing the amount of physical memory required for a process.
- It minimizes the initial disk overhead as initially not all pages are to be read.
- It does not need extra hardware support.

**Page Replacement Algorithms**

- When a page fault occurs, the OS has to choose a page to remove from memory to make room for incoming page.
- If the page to be removed has been modified while in memory (check the modified bit), it must be rewritten to the disk to make the copy of the page in the disk up to date
- If the page has not been changed, the disk copy is up to date and no rewrite is needed, the page to be brought in simply replaces the old page in memory
- Choosing a page randomly to be removed might not be effective. We will discuss some algorithms in doing so.

**1. FIFO Page Replacement algorithm.**
**2. Optimal Page Replacement algorithm.**
**3. Least Recently Used Page Replacement algorithm.**
**4. Counting based page replacement algorithm.**
**a. Least frequently used page replacement algorithm.**
**b. Most frequently used page replacement algorithm.**
**5. The second chance page replacement algorithm.**
**6. The clock page replacement algorithm**

**Algorithms**

**1. First-in First-out (FIFO) Page Replacement:**

- FIFO is simplest page replacement algorithm.
- As name suggest, the first page loaded into the memory is the first page to be removed.
- In this algorithm, the operating system keeps track of all pages in the memory in a queue, (being the oldest page is in the front of the queue). When a page needs to be replaced page in the front of the queue is selected for removal.

**2. Optimal Page Replacement Algorithm:**

- It is the best possible page replacement algorithm, which is easy to describe but impossible to implement
- Replace page that will not be used for longest period of time.
- At the moment that page fault occurs, some set of pages are in the memory that will be referenced sometime in the future (i.e. after some number of instruction).

**3. Least Recently Used (LRU) Page Replacement**

- Use past knowledge rather than future.
- It is based on the assumption that the page that has been used in the last few instruction will probably be referenced in the next few instruction.
- Replace page that has not been referenced for longest time.

**4. Counting based page replacement algorithm**

We can keep a counter of the number of references that have been made to each page and develop the following two schemes:

**a) Least Frequently Used (LFU) page replacement Algorithm**

**b) Most frequently Used (MFU) page replacement Algorithm.**

**a) Least Frequently Used (LFU) page Replacement**

- Requires that the page with the smallest count be replaced.
- Page with smallest count/ frequency will be replaced.

**b) Most Frequently Used (MFU) page Replacement**

- Based on the argument that the page with the smallest count was probably just brought in and has yet to be used.

**Second Chance Page Replacement Algorithm**

- This is the modified version of FIFO and we do not replace the page as it came early, we also check the reference bit means how recently it is used.
- Drawback in FIFO is that even if page is in use, it may be replaced due to its arrival time.
- It uses reference bit (R).
- Initially reference bit for all pages are 0.
- As the page is referenced, bit is set to 1 to indicate that the page is in use.

- A page whose reference bit 1 will not be replaced and will be given second chance.