Unit-3

Memory Management

The memory of a computer system is shared by a large number of processes, so memory management has traditionally been a very important task of an operating system. Memories keep becoming cheaper and larger every year; however, the pressure on memory as an OS resource persists because both the size of processes and the number of processes that an operating system has to service at any time also keep growing. The basic issues in memory management are efficient use of memory, protection of memory allocated to a process against illegal accesses by other processes, performance of individual processes, and performance of the system.

❖ Managing the Memory Hierarchy

A memory hierarchy comprises cache memories like the L1 and L3 caches, the memory management unit (MMU), memory, and a disk. Its purpose is to create an illusion of a fast and large memory at a low cost.

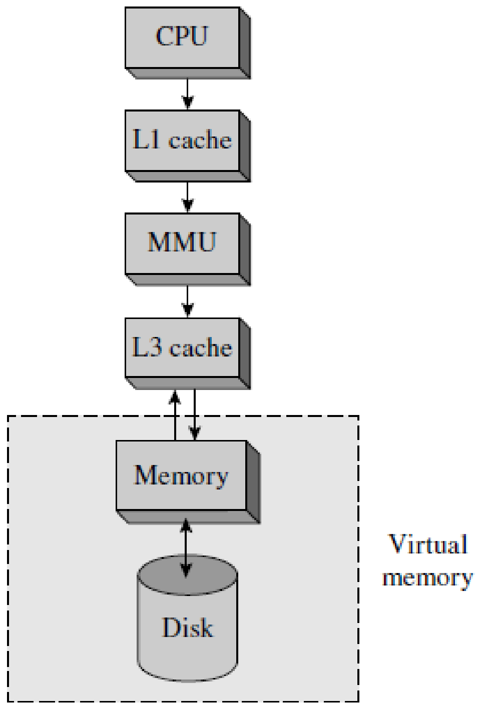


Figure 11.1 Managing the memory hierarchy.

The upper half of Figure 11.1 illustrates the memory hierarchy. The CPU refers to the fastest memory, the cache, when it needs to access an instruction or data. If the required instruction or data is not available in the cache, it is fetched from the next lower level in the memory hierarchy, which could be a slower cache or the random access memory (RAM), simply called memory in this book. If the required instruction or data is also not available in the next lower level memory, it is fetched there from a still lower level, and so on. Performance of a process depends on the hit ratios in various levels of the memory hierarchy, where the hit ratio in a level indicates what fraction of instructions or data bytes that were looked for in that level were actually present in it.

The caches are managed entirely in the hardware. The kernel employs special techniques to provide high cache hit ratios for a process. Memory is managed jointly by the kernel and the run-time library of the programming language in which the code of the process is written. The kernel allocates memory to user processes. The primary performance concern in this function is accommodating more user processes in memory, so that both system performance and user service would improve. The kernel meets this concern through efficient reuse of memory when a process completes.

The disk in the virtual memory is managed entirely by the kernel; the kernel stores different parts of each process's address space on the disk in such a manner that they can be accessed efficiently. It contributes to good execution performance of processes in a virtual memory.

❖ Static And Dynamic Memory Allocation

Memory allocation is an aspect of a more general action in software operation known as binding. Two other actions related to a program—it's linking and loading—are also aspects of binding.

A binding for an attribute of an entity such as a function or a variable can be performed any time before the attribute is used. Different binding methods perform the binding at different times. The exact time at which binding is performed may determine the efficiency and flexibility with which the entity can be used. Broadly speaking, we can differentiate between early binding and late binding. Late binding is useful in cases where the OS or run-time library may have more information about an entity at a later time, using which it may be able to perform a better quality binding. For example, it may be able to achieve more efficient use of resources such as memory. Early and late binding are represented by the two fundamental binding methods of static and dynamic binding, respectively.

Definition: Static Binding-A binding performed before the execution of a program (or operation of a software system) is set in motion.

Definition: Dynamic Binding-A binding performed during the execution of a program (or operation of a software system).

Static memory allocation can be performed by a compiler, linker, or loader while a program is being readied for execution. Dynamic memory allocation is performed in a “lazy” manner during the execution of a program; memory is allocated to a function or a variable just before it is used for the first time.

Static memory allocation to a process is possible only if sizes of its data structures are known before its execution begins. If sizes are not known, they have to be guessed; wrong estimates can lead to wastage of memory and lack of flexibility. For example, consider an array whose size is not known during compilation. Memory is wasted if we overestimate the array’s size, whereas the process may not be able to operate correctly if we underestimate its size. Dynamic memory allocation can avoid both these problems by allocating a memory area whose size matches the actual size of the array, which would be known by the time the allocation is performed. It can even permit the array size to vary during operation of the process. However, dynamic memory allocation incurs the overhead of memory allocation actions performed during operation of a process.

Operating systems choose static and dynamic memory allocation under different circumstances to obtain the best combination of execution efficiency and memory efficiency. When sufficient information about memory requirements is available a priori, the kernel or the run-time library makes memory allocation decisions statically, which provides execution efficiency. When little information is available a priori, the memory allocation decisions are made dynamically, which incurs higher overhead but ensures efficient use of memory. In other situations, the available information is used to make some decisions concerning memory allocation statically, so that the overhead of dynamic memory allocation can be reduced.

❖ Memory Allocation To A Process

➢ *Stacks and Heaps*

The compiler of a programming language generates code for a program and allocates its static data. It creates an object module for the program. The linker links the program with library functions and the run-time support of the programming language, prepares a ready-to-execute form of the program, and stores it in a file. The program size information is recorded in the directory entry of the file. The run-time support allocates two kinds of data during execution of the program. The first kind of data includes variables whose scope is associated with functions, procedures, or blocks, in a program and parameters of function or procedure calls. This data is allocated when a function, procedure or block is entered and is deallocated when it is exited. Because of the last-in, first-out nature of the allocation/deallocation, the data is allocated on the stack. The second kind of data is dynamically created by a program through language features like the new statement of Pascal, C++, or Java, or the malloc, calloc statements of C. We refer to such data as program-controlled dynamic data (PCD data). The PCD data is allocated by using a data structure called a heap.

Stack: In a stack, allocations and deallocations are performed in a last-in, first out (LIFO) manner in response to push and pop operations, respectively. We assume each entry in the stack to be of some standard size, say, l bytes. Only the last entry of the stack is accessible at any time. A contiguous area of memory is reserved for the stack. A pointer called the stack base (SB) points to the first entry of the stack, while a pointer called the top of stack (TOS) points to the last entry allocated in the stack. We will use the convention that a stack grows toward the lower end of memory.

During execution of a program, a stack is used to support function calls. The group of stack entries that pertain to one function call is called a stack frame; it is also called an activation record in compiler terminology. A stack frame is pushed on the stack when a function is called. To start with, the stack frame contains either addresses or values of the function's parameters, and the return address, i.e., the address of the instruction to which control should be returned after completing the function's execution. During execution of the function, the run-time support of the programming language in which the program is coded creates local data of the function within the stack frame. At the end of the function's execution, the entire stack frame is popped off the stack and the return address contained in it is used to pass control back to the calling program.

Two provisions are made to facilitate use of stack frames: The first entry in a stack frame is a pointer to the previous stack frame on the stack. This entry facilitates popping off of a stack frame. A pointer called the frame base (FB) is used to point to the start of the topmost stack frame in the stack. It helps in accessing various stack entries in the stack frame.

Heap: A heap permits allocation and deallocation of memory in a random order. An allocation request by a process returns with a pointer to the allocated memory area in the heap, and the process accesses the allocated memory area through this pointer. A deallocation request must present a pointer to the memory area to be deallocated. The use of a heap to manage the PCD data of a process. As "holes" develop in the memory allocation as data structures are created and freed. The heap allocator has to reuse such free memory areas while meeting future demands for memory.

➢ ***Memory Fragmentation***

Memory Fragmentation - The existence of unusable areas in the memory of a computer system.

Table 11.3 describes two forms of memory fragmentation. External fragmentation occurs when a memory area remains unused because it is too small to be allocated. Internal fragmentation occurs when some of the memory allocated to a process remains unused, which happens if a process is allocated more memory than it needs. We would have internal fragmentation if an allocator were to allocate, say, 100 bytes of memory when a process requests 50 bytes; this would happen if an allocator dealt exclusively with memory blocks of a few standard sizes to limit its overhead.

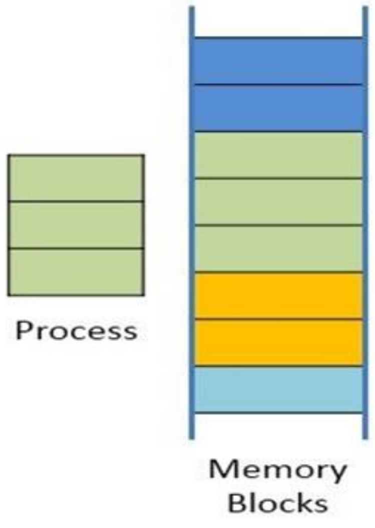
Table 11.3 Forms of Memory Fragmentation

Form of fragmentation	Description
External fragmentation	Some area of memory is too small to be allocated.
Internal fragmentation	More memory is allocated than requested by a process, hence some of the allocated memory remains unused.

❖ Contiguous Memory Allocation

In contiguous memory allocation, all the available memory space remains together in one place. It means freely available memory partitions are not scattered here and there across the whole memory space. In the contiguous memory allocation, both the operating system and the user must reside in the main memory. The main memory is divided into two portions one portion is for the operating and other is for the user program.

In the contiguous memory allocation when any user process request for the memory a single section of the contiguous memory block is given to that process according to its need. We can achieve contiguous memory allocation by dividing memory into the fixed-sized partition. A single process is allocated in that fixed sized single partition. But this will increase the degree of multiprogramming means more than one process in the main memory that bounds the number of fixed partition done in memory. Internal fragmentation increases because of the contiguous memory allocation.



Contiguous Memory Allocation

The memory can be divided either in the fixed-sized partition or in the variable-sized partition in order to allocate contiguous space to user processes.

- ***Fixed sized partition***

This technique is also known as static partitioning. In this scheme, the system divides the memory into fixed-size partitions. The partitions may or may not be the same size. The size of each partition is fixed.

In this partition scheme, each partition may contain exactly one process. There is a problem that this technique will limit the degree of multiprogramming because the number of partitions will basically decide the number of processes.

Whenever any process terminates then the partition becomes available for another process.

Advantage:

- This scheme is simple and is easy to implement
- It supports multiprogramming as multiple processes can be stored inside the main memory.
- Management is easy using this scheme

Disadvantage:

- **Internal Fragmentation**
  Suppose the size of the process is lesser than the size of the partition in that case some size of the partition gets wasted and remains unused. This wastage inside the memory is generally termed as internal fragmentation.

- **Limitation on the size of the process**
  If in a case size of a process is more than that of a maximum-sized partition then that process cannot be loaded into the memory. Due to this, a condition is imposed on the size of the process and it is: the size of the process cannot be larger than the size of the largest partition.

- ***Variable size partition***

This scheme is also known as dynamic partitioning and is came into existence to overcome the drawback i.e. internal fragmentation that is caused by static partitioning. In this partitioning, scheme allocation is done dynamically.

The size of the partition is not declared initially. Whenever any process arrives, a partition of size equal to the size of the process is created and then allocated to the process. Thus the size of each partition is equal to the size of the process.

As partition size varies according to the need of the process so in this partition scheme there is no internal fragmentation.

Advantage:

- **No Internal Fragmentation:** As in this partition scheme space in the main memory is allocated strictly according to the requirement of the process thus there is no chance of internal fragmentation. Also, there will be no unused space left in the partition.
- **Degree of Multiprogramming is Dynamic:** As there is no internal fragmentation in this partition scheme due to which there is no unused space in the memory. Thus more processes can be loaded into the memory at the same time.
- **No Limitation on the Size of Process:** In this partition scheme as the partition is allocated to the process dynamically thus the size of the process cannot be restricted because the partition size is decided according to the process size.

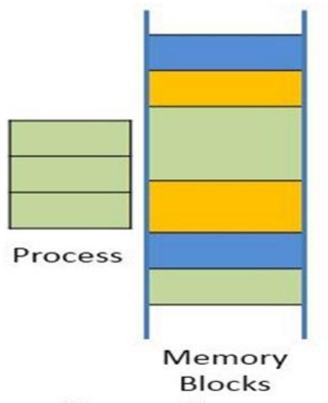
Disadvantage:

- **Difficult Implementation** The implementation of this partition scheme is difficult as compared to the Fixed Partitioning scheme as it involves the allocation of memory at run-time rather than during the system configuration. As we know that OS keeps the track of all the partitions but here allocation and deallocation are done very frequently and partition size will be changed at each time so it will be difficult for the operating system to manage everything.

❖ Non-contiguous memory allocation

In the non-contiguous memory allocation the available free memory space are scattered here and there and all the free memory space is not at one place. So this is time-consuming.

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. This technique of non-contiguous memory allocation reduces the wastage of memory which leads to internal and external fragmentation. This utilizes all the free memory space which is created by a different process.



Noncontiguous Memory Allocation

There are two fundamental approaches to implementing noncontiguous memory allocation:

- Paging
- Segmentation

In **paging**, each process consists of fixed-size components called pages. The size of a page is defined by the hardware of a computer, and demarcation of pages is implicit in it. The memory can accommodate an integral number of pages. It is partitioned into memory areas that have the same size as a page, and each of these memory areas is considered separately for allocation to a page. This way, any free memory area is exactly the same size as a page, so external fragmentation does not arise in the system. Internal fragmentation can arise because the last page of a process is allocated a page-size memory area even if it is smaller than a page in size.

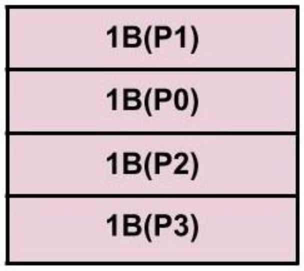
In ***segmentation***, a programmer identifies components called segments in a process. A segment is a logical entity in a program, e.g., a set of functions, data structures, or objects. Segmentation facilitates sharing of code, data, and program modules between processes. However, segments have different sizes, so the kernel has to use memory reuse techniques such as first-fit or best-fit allocation. Consequently, external fragmentation can arise.

A hybrid approach called *segmentation with paging* combines the features of both segmentation and paging. It facilitates sharing of code, data, and program modules between processes without incurring external fragmentation; however, internal fragmentation occurs as in paging.

➢ Paging

Paging is a non-contiguous memory allocation technique in which secondary memory and the main memory is divided into equal size partitions. The partitions of the secondary memory are called pages while the partitions of the main memory are called frames. They are divided into equal size partitions to have maximum utilization of the main memory and avoid external fragmentation.

Example: Process P having process size as 4B, page size as 1B. Therefore there will be four pages(P0, P1, P2, P3) each of size 1B. Also, when this process goes into the main memory for execution then depending upon the availability, it may be stored in non-contiguous fashion in the main memory frame as shown below:



Main Memory

Translation of logical Address into physical Address

As a CPU always generates a logical address and we need a physical address for accessing the main memory. This mapping is done by the MMU (memory management Unit) with the help of the page table.

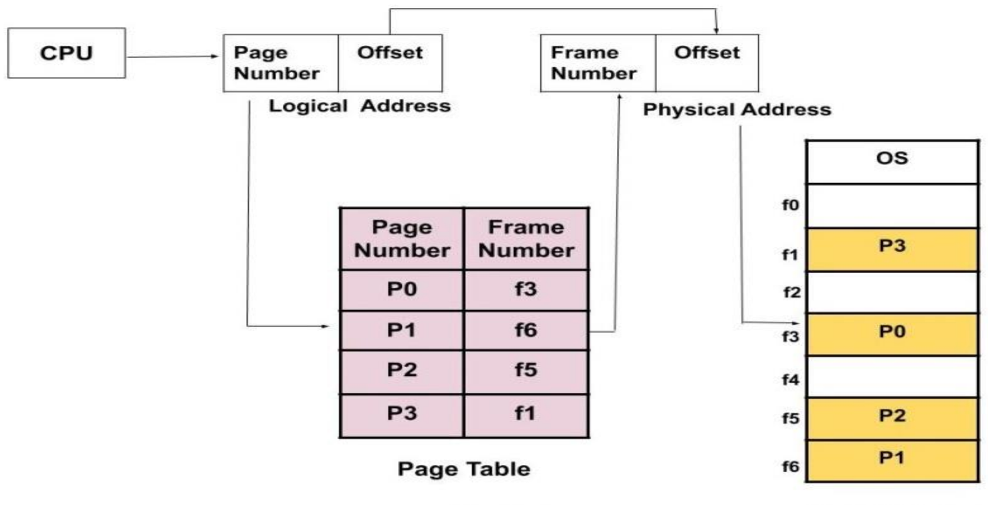
- Logical Address: The logical address consists of two parts page number and page offset.

1. *Page Number:* It tells the exact page of the process which the CPU wants to access.
2. *Page Offset:* It tells the exact word on that page which the CPU wants to read.

- Physical Address: The physical address consists of two parts frame number and page offset.

1. *Frame Number:* It tells the exact frame where the page is stored in physical memory.
2. *Page Offset:* It tells the exact word on that page which the CPU wants to read. It requires no translation as the page size is the same as the frame size so the place of the word which CPU wants access will not change.

- Page table: A page stable contains the frame number corresponding to the page number of some specific process. So, each process will have its own page table. A register called Page Table Base Register (PTBR) which holds the base value of the page table.



Paging

The CPU generates the logical address which contains the page number and the page offset. The PTBR register contains the address of the page table. Now, the page table helps in determining the frame number corresponding to the page number. Now, with the help of frame number and the page offset the physical address is determined and the page is accessed in the main memory.

Advantages of Paging

1. There is no external fragmentation as it allows us to store the data in a non-contiguous way.
2. Swapping is easy between equal-sized pages and frames.

Disadvantages of Paging

1. As the size of the frame is fixed, so it may suffer from internal fragmentation. It may happen that the process is too small and it may not acquire the entire frame size.
2. The access time increases because of paging as the main memory has to be now accessed two times. First, we need to access the page table which is also stored in the main memory and second, combine the frame number with the page offset and then get the physical address of the page which is again stored in the main memory.
3. For every process, we have an independent page table and maintaining the page table is extra overhead.

➢ Segmentation

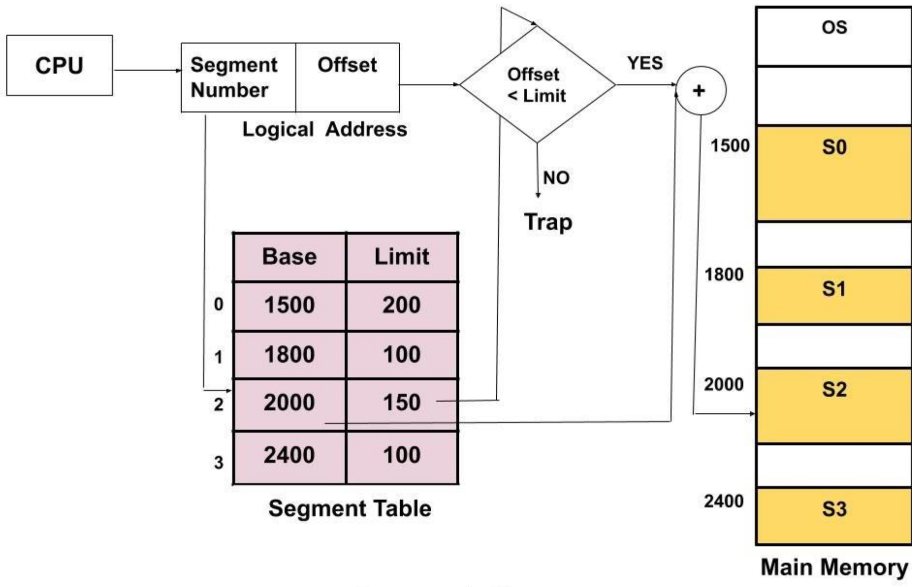
As a CPU always generates a logical address and needs a physical address for accessing the main memory. This mapping is done by the MMU (memory management Unit) with the help of the segment table.

- Logical Address: The logical address consists of two parts segment number and page offset.

1. Segment Number: It tells the specific segment of the process from which the CPU wants to read the data.
2. Segment Offset: It tells the exact word in that segment which the CPU wants to read.

- Physical Address: The physical address is obtained by adding the base address of the segment to the segment offset.

- Segment table: A segment table stores the base address of each segment in the main memory. It has two parts i.e. Base and Limit. Here, base indicates the base address or starting address of the segment in the main memory. Limit tells the size of that segment. A register called Segment Table Base Register (STBR) which holds the base value of the segment table. The segment table is also stored in the main memory itself.



Segmentation

The CPU generates the logical address which contains the segment number and the segment offset. STBR register contains the address of the segment table. Now, the segment table helps in determining the base address of the segment corresponding to the page number. Now, the segment offset is compared with the limit corresponding to the Base. If the segment offset is greater than the limit then it is an invalid address. This is because the CPU is trying to access a word in the segment and this value is greater than the size of the segment itself which is not possible. If the segment offset is less than or equal to the limit then only the request is accepted. The physical address is generated by adding the base address of the segment to the segment offset.

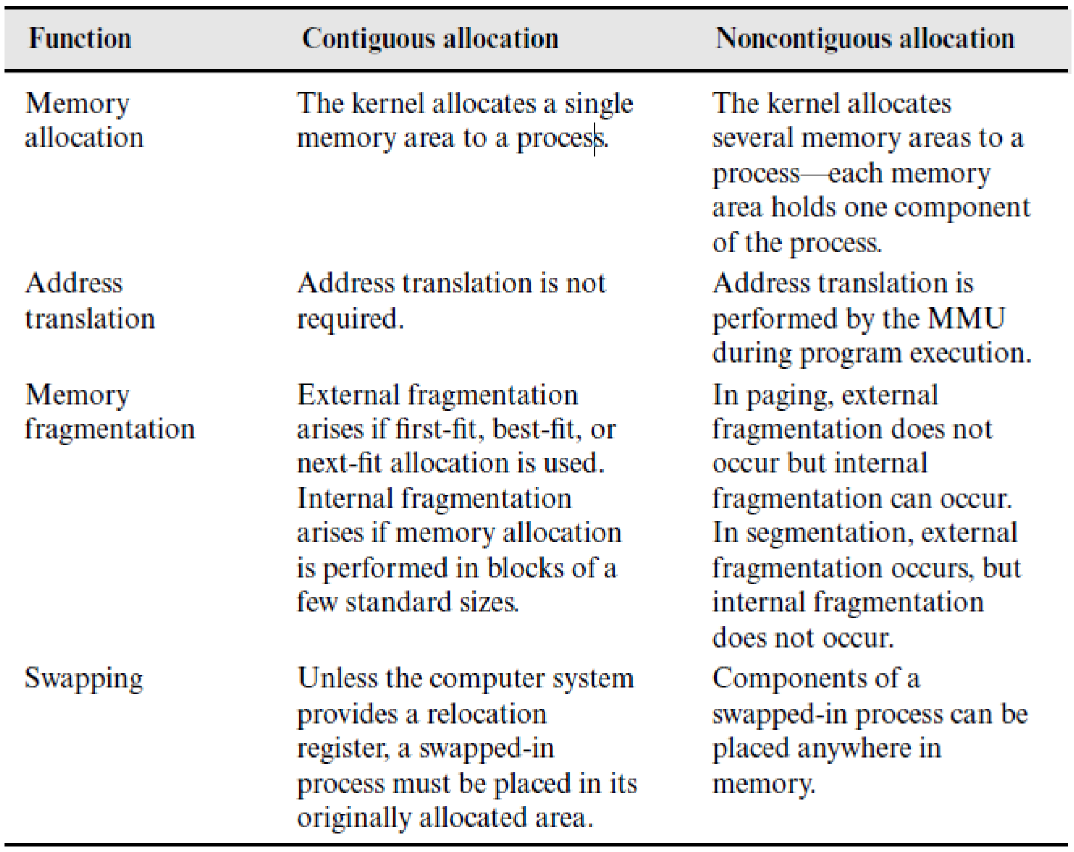
Advantages of Segmentation

1. The size of the segment table is less compared to the size of the page table.
2. There is no internal fragmentation.

Disadvantages of Segmentation

1. When the processes are loaded and removed (during swapping) from the main memory then free memory spaces are broken into smaller pieces and this causes external fragmentation.
2. Here also the time to access the data increases as due to segmentation the main memory has to be now accessed two times. First, we need to access the segment table which is also stored in the main memory and second, combine the base address of the segment with the segment offset and then get the physical address which is again stored in the main memory.

➢ Comparison of Contiguous and Noncontiguous Memory Allocation



Function	Contiguous allocation	Noncontiguous allocation
Memory allocation	The kernel allocates a single memory area to a process.	The kernel allocates several memory areas to a process—each memory area holds one component of the process.
Address translation	Address translation is not required.	Address translation is performed by the MMU during program execution.
Memory fragmentation	External fragmentation arises if first-fit, best-fit, or next-fit allocation is used. Internal fragmentation arises if memory allocation is performed in blocks of a few standard sizes.	In paging, external fragmentation does not occur but internal fragmentation can occur. In segmentation, external fragmentation occurs, but internal fragmentation does not occur.
Swapping	Unless the computer system provides a relocation register, a swapped-in process must be placed in its originally allocated area.	Components of a swapped-in process can be placed anywhere in memory.

❖ Kernel Memory Allocation

The kernel creates and destroys data structures at a high rate during its operation. These are mostly control blocks that control the allocation and use of resources in the system. Some familiar control blocks are the process control block (PCB) created for every process and the event control block (ECB) created whenever the occurrence of an event is anticipated. The sizes of control blocks are known in the design stage of an OS. This prior knowledge helps make kernel memory allocation simple and efficient— memory that is released when one control block is destroyed can be reused when a similar control block is created. To realize this benefit, a separate free list can be maintained for each type of control block.

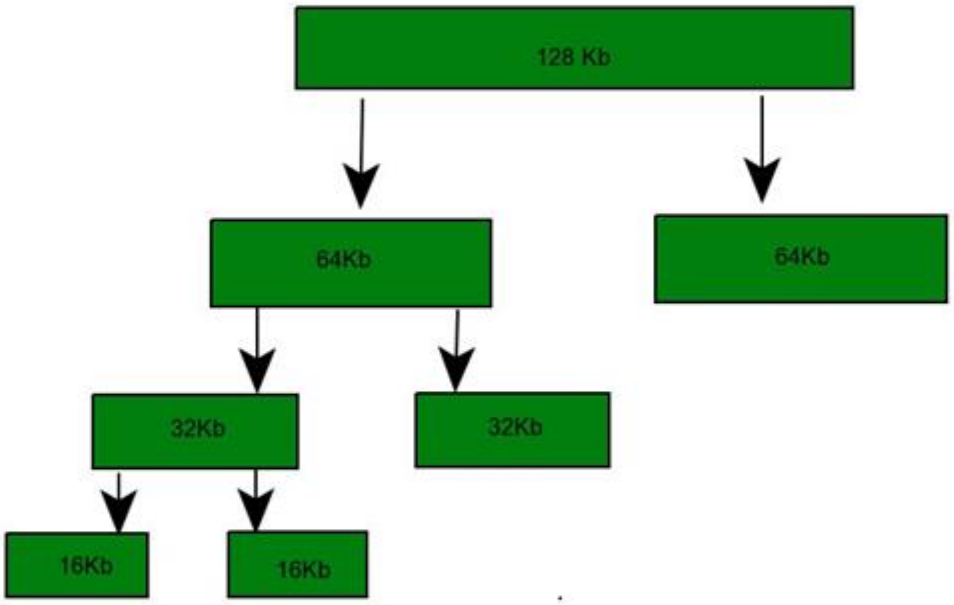
Kernels of modern operating systems use noncontiguous memory allocation with paging to satisfy their own memory requirements, and make special efforts to use each page effectively. Three of the leading memory allocators are:

- Lazy buddy allocator or Buddy allocator
- Slab allocator

Buddy allocation system

Buddy allocation system is an algorithm in which a larger memory block is divided into small parts to satisfy the request. This algorithm is used to give best fit. The two smaller parts of block are of equal size and called as buddies. In the same manner one of the two buddies will further divide into smaller parts until the request is fulfilled. Benefit of this technique is that the two buddies can combine to form the block of larger size according to the memory request.

Example – If the request of 25Kb is made then block of size 32Kb is allocated.



Advantages –

- In comparison to other simpler techniques such as dynamic allocation, the buddy memory system has little external fragmentation.
- The buddy memory allocation system is implemented with the use of a binary tree to represent used or unused split memory blocks.
- The buddy system is very fast to allocate or deallocate memory.
- In buddy systems, the cost to allocate and free a block of memory is low compared to that of best-fit or first-fit algorithms.
- Other advantage is coalescing.
- Address calculation is easy.

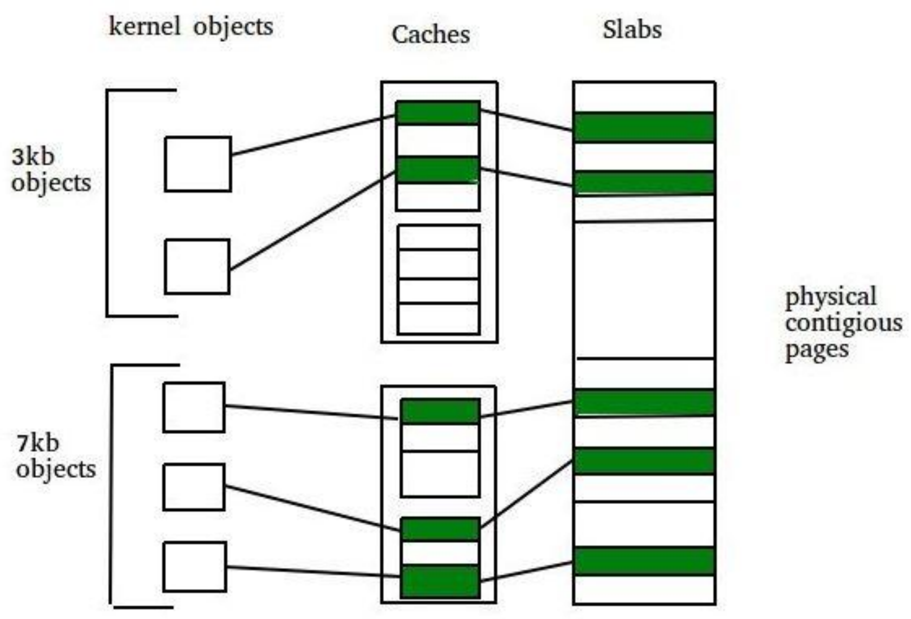
Disadvantage –
The main drawback in buddy system is internal fragmentation as larger block of memory is acquired then required. For example if a 36 kb request is made then it can only be satisfied by 64 kb segment and remaining memory is wasted.

Slab Allocation

A second strategy for allocating kernel memory is known as slab allocation. It eliminates fragmentation caused by allocations and deallocations. This method is used to retain allocated memory that contains a data object of a certain type for reuse upon subsequent allocations of objects of the same type. In slab allocation memory chunks suitable to fit data objects of certain type or size are preallocated. Cache does not free the space immediately after use although it keeps track of data which are required frequently so that whenever request is made the data will reach very fast. Two terms required are:

- Slab – A slab is made up of one or more physically contiguous pages. The slab is the actual container of data associated with objects of the specific kind of the containing cache.
- Cache – Cache represents a small amount of very fast memory. A cache consists of one or more slabs. There is a single cache for each unique kernel data structure.

Implementation –
The slab allocation algorithm uses caches to store kernel objects. When a cache is created a number of objects which are initially marked as free are allocated to the cache. The number of objects in the cache depends on size of the associated slab. *Example* – A 12 kb slab (made up of three contiguous 4 kb pages) could store six 2 kb objects. Initially all objects in the cache are marked as free. When a new object for a kernel data structure is needed, the allocator can assign any free object from the cache to satisfy the request. The object assigned from the cache is marked as used.



In linux, a slab may in one of three possible states:

1. Full – All objects in the slab are marked as used
2. Empty – All objects in the slab are marked as free
3. Partial – The slab consists of both

The slab allocator first attempts to satisfy the request with a free object in a partial slab. If none exists, a free object is assigned from an empty slab. If no empty slabs are available, a new slab is allocated from contiguous physical pages and assigned to a cache.

Advantages

- No memory is wasted due to fragmentation because each unique kernel data structure has an associated cache.
- Memory request can be satisfied quickly.
- The slab allocating scheme is particularly effective for managing when objects are frequently allocated or deallocated. The act of allocating and releasing memory can be a time consuming process. However, objects are created in advance and thus can be quickly allocated from the cache. When the kernel has finished with an object and releases it, it is marked as free and return to its cache, thus making it immediately available for subsequent request from the kernel.

❖ VIRTUAL MEMORY

Virtual memory is a part of the memory hierarchy that consists of memory and a disk. In accordance with the principle of memory hierarchies described in Chapter 2, only some portions of the address space of a process—that is, of its code and data—exist in memory at any time; other portions of its address space reside on disk and are loaded into memory when needed during operation of the process. The kernel employs virtual memory to reduce the memory commitment to a process so that it can service a large number of processes concurrently, and to handle processes whose address space is larger than the size of memory.

Virtual memory is implemented through the noncontiguous memory allocation model and comprises both hardware components and a software component called a virtual memory manager. The hardware components speed up address translation and help the virtual memory manager perform its tasks more effectively. The virtual memory manager decides which portions of a process address space should be in memory at any time.

Performance of virtual memory depends on the rate at which portions of a process address space have to be loaded in memory from a disk and removed from memory to make space for new portions. According to the law of locality of reference, a process is likely to access recently referenced portions of its address space again. The virtual memory manager ensures good performance of virtual memory by allocating an adequate amount of memory to a process and employing a replacement algorithm to remove a portion that has not been referenced recently.

❖ Virtual Memory Basics

Users always want more from a computer system—more resources and more services. The need for more resources is satisfied either by obtaining more efficient use of existing resources, or by creating an illusion that more resources exist in the system. A virtual memory is what its name indicates—it is an illusion of a memory that is larger than the real memory, i.e., RAM, of the computer system. This illusion is a part of a user's abstract view of memory. A user or his application program sees only the virtual memory. The kernel implements the illusion through a combination of hardware and software means. We refer to real memory simply as memory. We refer to the software component of virtual memory as a virtual memory manager.

The illusion of memory larger than the system's memory crops up any time a process whose size exceeds the size of memory is initiated. The process is able to operate because it is kept in its entirety on a disk and only its required portions are loaded in memory at any time. The basis of virtual memory is the noncontiguous memory allocation model described in memory management. The address space of each process is assumed to consist of portions called components. The portions can be loaded into nonadjacent areas of memory. The address of each operand or instruction in the code of a process is a logical address of the form ($comp_i$, $byte_i$). The memory management unit (MMU) translates it into the address in memory where the operand or instruction actually resides.

Use of the noncontiguous memory allocation model reduces memory fragmentation, since a free area of memory can be reused even if it is not large enough to hold the entire address space of a process. More user processes can be accommodated in memory this way, which benefits both users and the OS. The kernel carries this idea further—even processes that can fit in memory are not loaded fully into memory. This strategy reduces the amount of memory that is allocated to each process, thus further increasing the number of processes that can be in operation at the same time.

Figure 12.1 shows a schematic diagram of a virtual memory. The logical address space of the process shown consists of five components. Three of these components are presently in memory. Information about the memory areas where these components exist is maintained in a data structure of the virtual memory manager. This information is used by the MMU during address translation. When an instruction in the process refers to a data item or instruction that is not in memory, the component containing it is loaded from the disk. Occasionally, the virtual memory manager removes some components from memory to make room for other components.

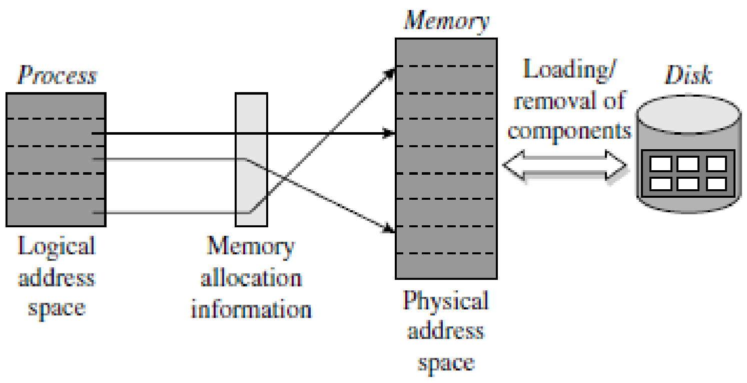


Figure 12.1 Overview of virtual memory.

The arrangement shown in Figure 12.1 is a memory hierarchy. The hierarchy consists of the system's memory and a disk. Memory is fast, but small in size. The disk is slow, but has a much larger capacity. The MMU and the virtual memory manager together manage the memory hierarchy, so that the current instruction in a process finds its operands in memory.

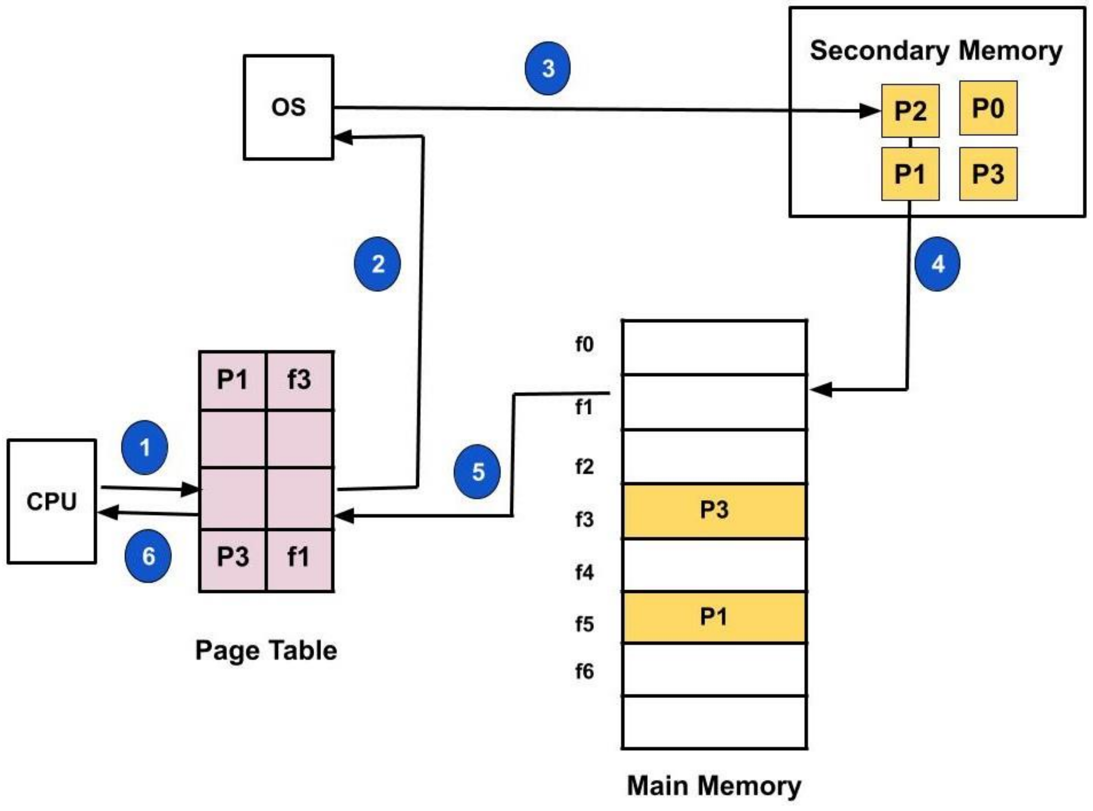
Definition: Virtual Memory **-** A memory hierarchy, consisting of a computer system's memory and a disk, that enables a process to operate with only some portions of its address space in memory.

❖ Demand Paging

According to concepts of virtual memory, in order to execute any process, it not necessary that the whole process should present in the main memory at the given time. The process can also be executed if only some pages are present in the main memory at any given time. But, how can we decide beforehand which page should be present in the main memory at a particular time and which should not be there? To resolve this problem Demand paging concept came into play.

Demand paging is a technique used in virtual memory systems where the pages are brought in the main memory only when required or demanded by the CPU. Hence, it is also named as *lazy swapper* because the swapping of pages is done only when required by the CPU.

For example: Suppose, to execute a process P having four pages as P0, P1, P2, and P3. Currently, in the page table, there are only page P1 and P3.



1. Now, if the CPU wants to access page P2 of a process P, first it will search the page in the page table.
2. As the page table does not contain this page so it will be a trap or page fault. As soon as the trap is generated and context switching happens and the control goes to the operating system.
3. The OS system will put the process in a waiting/ blocked state. The OS system will now search that page in the backing store or secondary memory.
4. The OS will then read the page from the backing store and load it to the main memory.
5. Next, the OS system will update the page table entry accordingly.
6. Finally, the control is taken back from the OS and the execution of the process is resumed.

Hence whenever a page fault occurs these steps are followed by the operating system and the required page is brought into memory.

Advantages

- It increases the degree of multiprogramming as many processes can be present in the main memory at the same time.
- There is a more efficient use of memory as processes having size more than the size of the main memory can also be executed using this mechanism because we are not loading the whole page at a time.

Disadvantages

- The amount of processor overhead and the number of tables used for handling the page faults is greater than in simple page management techniques.

❖ Page Replacement Algorithm

Page replacement becomes necessary when a page fault occurs and there are no free page frames in memory. However, another page fault would arise if the replaced page is referenced again. Hence it is important to replace a page that is not likely to be referenced in the immediate future. Page Replacement Algorithm decides which page to remove, also called swap out when a new page needs to be loaded into the main memory. Page Replacement happens when a requested page is not present in the main memory and the available space is not sufficient for allocation to the requested page.

When the page that was selected for replacement was paged out, and referenced again, it has to read in from disk, and this requires for I/O completion. This process determines the quality of the page replacement algorithm: the lesser the time waiting for page-ins, the better is the algorithm. A page replacement algorithm tries to select which pages should be replaced so as to minimize the total number of page misses. There are many different page replacement algorithms. These algorithms are evaluated by running them on a particular string of memory reference and computing the number of page faults. The fewer is the page faults the better is the algorithm for that situation.

*** If a process requests for page and that page is found in the main memory then it is called **page hit**, otherwise **page miss** or **page fault**.*

Some Page Replacement Algorithms:

- First In First Out (FIFO)
- Least Recently Used (LRU)
- Optimal Page Replacement

➢ First In First Out (FIFO)

This is the simplest page replacement algorithm. In this algorithm, the OS maintains a queue that keeps track of all the pages in memory, with the oldest page at the front and the most recent page at the back. When there is a need for page replacement, the FIFO algorithm, swaps out the page at the front of the queue, that is the page which has been in the memory for the longest time.

For Example: Consider the page reference string of size 12: 1, 2, 3, 4, 5, 1, 3, 1, 6, 3, 2, 3 with frame size 4(i.e. maximum 4 pages in a frame).

1	2	3	4	5	1	3	1	6	3	2	3

1	1	1	1	5	5	5	5	5	5	2	2
	2	2	2	2	1	1	1	1	1	1	1
		3	3	3	3	3	3	6	6	6	6
			4	4	4	4	4	4	3	3	3

M	M	M	M	M	M	H	H	M	M	M	H

M = Miss
H = Hit

- Initially, all 4 slots are empty, so when 1, 2, 3, 4 came they are allocated to the empty slots in order of their arrival. This is page fault as 1, 2, 3, 4 are not available in memory.
- When 5 comes, it is not available in memory so page fault occurs and it replaces the oldest page in memory, i.e., 1.
- When 1 comes, it is not available in memory so page fault occurs and it replaces the oldest page in memory, i.e., 2.
- When 3, 1 comes, it is available in the memory, i.e., Page Hit, so no replacement occurs.
- When 6 comes, it is not available in memory so page fault occurs and it replaces the oldest page in memory, i.e., 3.
- When 3 comes, it is not available in memory so page fault occurs and it replaces the oldest page in memory, i.e., 4.
- When 2 comes, it is not available in memory so page fault occurs and it replaces the oldest page in memory, i.e., 5.
- When 3 comes, it is available in the memory, i.e., Page Hit, so no replacement occurs.

Total Page Fault = 9
Page Fault ratio = 9/12 i.e. total miss/total possible cases

Advantages:

- Simple and easy to implement.
- Low overhead.

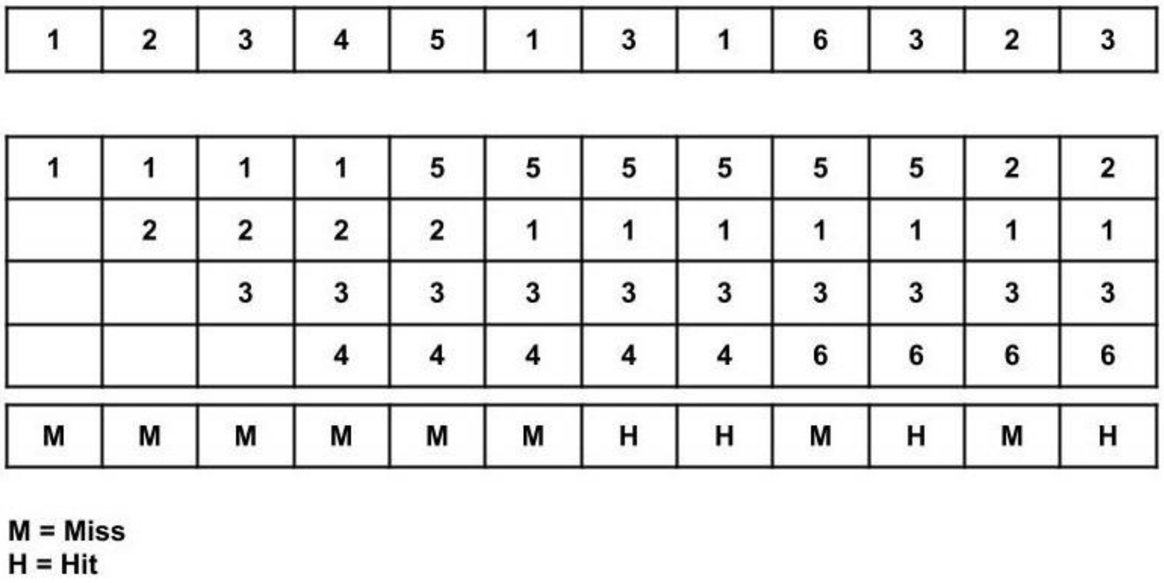
Disadvantages:

- Poor performance.
- Doesn't consider the frequency of use or last used time, simply replaces the oldest page.
- Suffers from Belady's Anomaly(i.e. more page faults when we increase the number of page frames).

➢ Least Recently Used (LRU)

Least Recently Used page replacement algorithm keeps track of page usage over a short period of time. It works on the idea that the pages that have been most heavily used in the past are most likely to be used heavily in the future too. In LRU, whenever page replacement happens, the page which has not been used for the longest amount of time is replaced.

For Example



1	2	3	4	5	1	3	1	6	3	2	3

1	1	1	1	5	5	5	5	5	5	2	2
	2	2	2	2	1	1	1	1	1	1	1
		3	3	3	3	3	3	3	3	3	3
			4	4	4	4	4	6	6	6	6

M	M	M	M	M	M	H	H	M	H	M	H

M = Miss
H = Hit

- Initially, all 4 slots are empty, so when 1, 2, 3, 4 came they are allocated to the empty slots in order of their arrival. This is page fault as 1, 2, 3, 4 are not available in memory.
- When 5 comes, it is not available in memory so page fault occurs and it replaces 1 which is the least recently used page.
- When 1 comes, it is not available in memory so page fault occurs and it replaces 2.
- When 3,1 comes, it is available in the memory, i.e., Page Hit, so no replacement occurs.
- When 6 comes, it is not available in memory so page fault occurs and it replaces 4.
- When 3 comes, it is available in the memory, i.e., Page Hit, so no replacement occurs.
- When 2 comes, it is not available in memory so page fault occurs and it replaces 5.
- When 3 comes, it is available in the memory, i.e., Page Hit, so no replacement occurs.

Total Page Fault = 8
Page Fault ratio = 8/12

Advantages

- Efficient.
- Doesn't suffer from Belady's Anomaly.

Disadvantages

- Complex Implementation.
- Expensive.
- Requires hardware support.

➢ Optimal Page Replacement

Optimal Page Replacement algorithm is the best page replacement algorithm as it gives the least number of page faults. In this algorithm, pages are replaced which would not be used for the longest duration of time in the future, i.e., the pages in the memory which are going to be referred farthest in the future are replaced.

This algorithm was introduced long back and is difficult to implement because it requires future knowledge of the program behavior. However, it is possible to implement optimal page replacement on the second run by using the page reference information collected on the first run.

For Example

1	2	3	4	5	1	3	1	6	3	2	3

1	1	1	1	1	1	1	1	6	6	6	6
	2	2	2	2	2	2	2	2	2	2	2
		3	3	3	3	3	3	3	3	3	3
			4	5	5	5	5	5	5	5	5

M	M	M	M	M	H	H	H	M	H	H	H

M = Miss
H = Hit

- Initially, all 4 slots are empty, so when 1, 2, 3, 4 came they are allocated to the empty slots in order of their arrival. This is page fault as 1, 2, 3, 4 are not available in memory.
- When 5 comes, it is not available in memory so page fault occurs and it replaces 4 which is going to be used farthest in the future among 1, 2, 3, 4.
- When 1, 3, 1 comes, they are available in the memory, i.e., Page Hit, so no replacement occurs.
- When 6 comes, it is not available in memory so page fault occurs and it replaces 1.
- When 3, 2, 3 comes, it is available in the memory, i.e., Page Hit, so no replacement occurs.

Total Page Fault = 6

Page Fault ratio = 6/12

Advantages

- Easy to Implement.
- Simple data structures are used.
- Highly efficient.

Disadvantages

- Requires future knowledge of the program.
- Time-consuming.