OPERATING SYSTEMS

MODULE-4

FILE SYSTEM INTERFACE

Memory-Management Strategies: Introduction, Contiguous memory allocation, Paging, Structure of the Page Table, Swapping.
Virtual Memory Management: Introduction, Demand paging, Copy-on-write, Page replacement, Allocation of frames, Thrashing.
Storage Management: Overview of Mass Storage Structure, HDD Scheduling.

INTRODUCTION

Memory is central to the operation of a modern computer system. Memory consists of a large array of bytes, each with its own address. **Main Memory** refers to a physical memory that is the internal memory of the computer. The computer is able to change only data that is in the **Main Memory**.

Sometimes a certain part or routine of the program is loaded into the main memory only when it is called by the program, this mechanism is called **Dynamic Loading**. An address generated by the CPU is commonly referred to as a **logical address.** Whereas an address seen by the memory unit that is, the one loaded into the memory-address register of the memory is commonly referred to as a **physical address.**

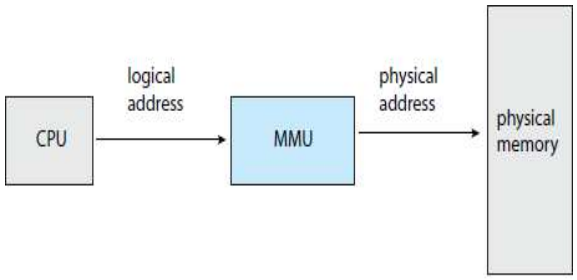


Figure Memory management unit (MMU).

Memory Management is the process of coordinating and controlling the memory in a computer, assigning portions known as blocks to various running programs to optimize the overall performance of the system. It is the most important function of an operating system that manages primary memory.

CONTIGUOUS MEMORY ALLOCATION

The main memory must accommodate both the operating system and the various user processes. We therefore need to allocate main memory in the most efficient way possible. We usually want several user processes to reside in memory at the same time. We therefore need to consider how to allocate available memory to the processes that are waiting to be brought into memory.

In **contiguous memory allocation, each process is contained in a single section of memory** that is contiguous to the section containing the next process. Whenever there is a **request by the user process** for the memory then a **single section of the contiguous memory block** is given to that process according to its requirement.

In order to allocate contiguous space to user processes the memory can be divided either in the

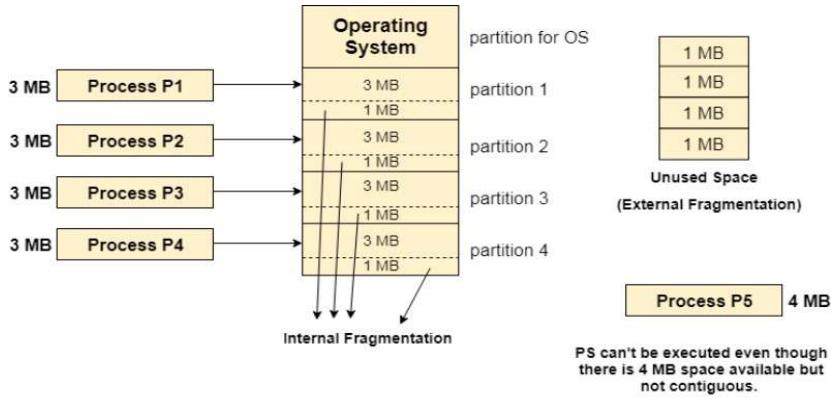
- **Fixed-sized Partition**
- **Variable-sized Partition**

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 as indicated by the name of the technique and it cannot be changed. The operating system always resides in the first partition while the other partitions can be used to store user processes.

In fixed partitioning,

- The partitions cannot overlap.
- A process must be contiguously present in a partition for the execution.



Fixed Partitioning

(Contiguous memory allocation)

There are various cons of using this technique

- **Internal Fragmentation**
- **External Fragmentation**
- **Limitation on the size of the process**
- **Degreeof multiprogramming is less**

Internal Fragmentation

If the size of the process is lesser then the total size of the partition then some size of the partition get wasted and remain unused. This is wastage of the memory and called internal fragmentation. As shown in the figure, the 4 MB partition is used to load only 3 MB process and the remaining 1 MB got wasted.

External Fragmentation

The total unused space of various partitions cannot be used to load the processes even though there is space available but not in the contiguous form. In the example the remaining 1 MB space of each partition cannot be used as a unit to store a 4 MB process. Despite of the fact that the sufficient space is available to load the process, process will not be loaded.

Limitation on the size of the process

If the process size is larger than the size of maximum sized partition then that process cannot be loaded into the memory. Therefore, a **limitation can be imposed** on the process size that is it cannot be larger than the size of the largest partition.

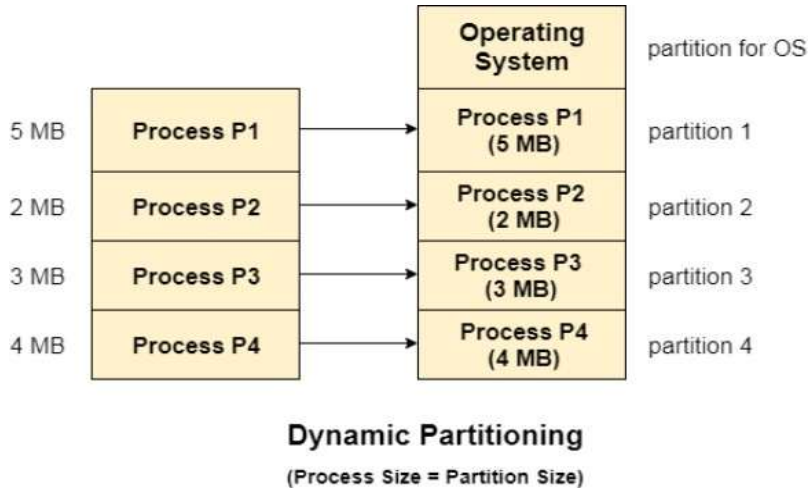
Degree of multiprogramming is less

By Degree of multi programming, we simply mean the **maximum number of processes** that can be loaded into the memory at the same time. In fixed partitioning, the degree of multiprogramming is **fixed** and very less due to the fact that the size of the partition cannot be varied according to the size of processes.

VARIABLE-SIZED PARTITION

This technique is also known as **Dynamic partitioning**. Dynamic partitioning tries to overcome the problems caused by fixed partitioning. In this technique, the partition size is not declared initially. It is declared at the time of process loading. The first partition is reserved for the operating system. The remaining space is divided into parts. The size of each partition will be equal to the size of the process. The partition size varies

according to the need of the process so that the internal fragmentation can be avoided.



Dynamic Partitioning

(Process Size = Partition Size)

Advantages of Variable Partitioning over fixed partitioning

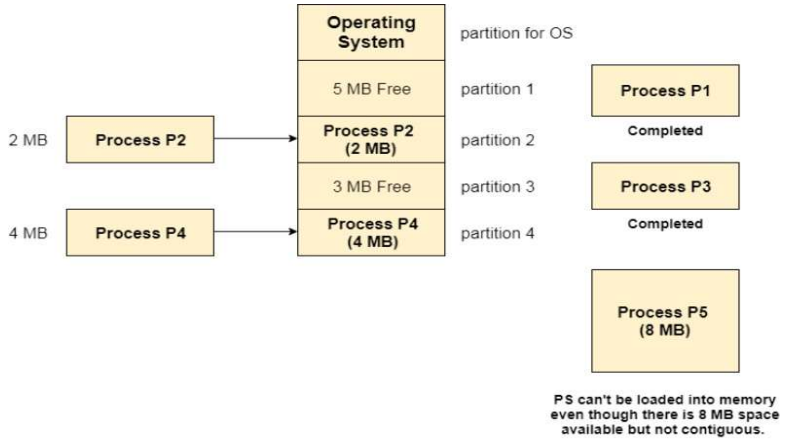
- No Internal Fragmentation
- No limitation on the size of the process
- Degree of multiprogramming is dynamic

Disadvantages of Variable Partitioning

- External Fragmentation
- Complex Memory Allocation

External Fragmentation

Absence of internal fragmentation doesn't mean that there will not be external fragmentation. The rule says that the process must be contiguously present in the main memory to get executed.



External Fragmentation in Dynamic Partitioning

Complex Memory Allocation

In Fixed partitioning, the list of partitions is made once and will never change. But in dynamic partitioning, the allocation and de-allocation is very complex since the partition size will be varied every time when it is assigned to a new process. OS has to keep track of all the partitions.

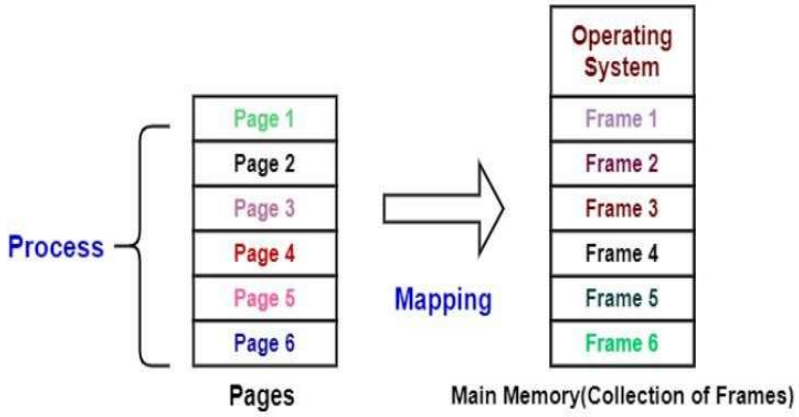
PAGING

Paging is a memory management scheme that permits a process's **physical address space to be non- contiguous.** Paging avoids **external fragmentation** and the associated **need for compaction**, two problems that affect contiguous memory allocation. Paging is implemented through cooperation between the **operating system** and the **computer hardware**.

The basic method for implementing paging involves

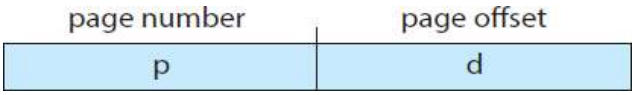
- Breaking **physical memory** into fixed-sized blocks called **frames** and
- Breaking **logical memory** into blocks of the same size called **pages**

When a process is to be executed, its pages are loaded into any available memory frames from their source.



Every address generated by the CPU is divided into two parts:

- **a page number (p)**
- **a page offset (d)**



The page table contains the base address of each frame in physical memory. The offset is the location in the frame being referenced. Thus, the base address of the frame is combined with the page offset to define the physical memory address. The page number is used as an index into a per-process page table.

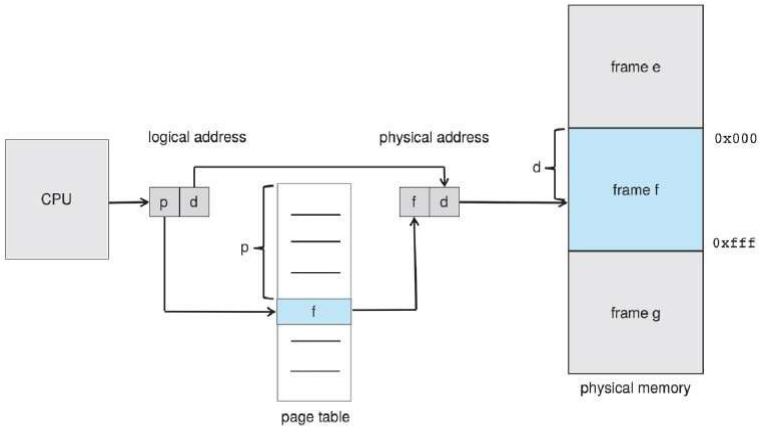


Figure Paging hardware.

The paging model of memory is shown in Figure

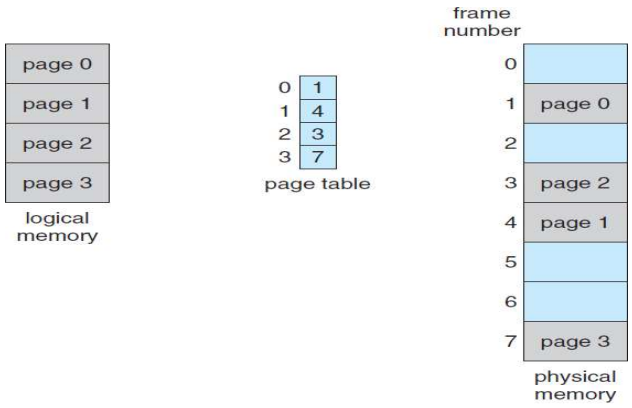


Figure Paging model of logical and physical memory.

The steps taken by the MMU to **translate a logical address** generated by the CPU **to a physical address:**

1. Extract the page number p and use it as an index into the page table.

2. Extract the corresponding frame number f from the page table.

3. Replace the page number p in the logical address with the frame number f.

As the offset d does not change, it is not replaced, and the frame number and offset now comprise the physical address.

For Example,

Using a page size of 4 bytes and a physical memory of 32 bytes (8 pages), we show how the programmer's view of memory can be mapped into physical memory. Logical address 0 is page 0, offset 0. Indexing into the page table, we find that page 0 is in frame

5. Thus, logical address 0 maps to physical address 20 [= (5 × 4) +0]. Logical address 3 (page 0, offset 3) maps to physical address 23 [= (5 × 4) +3].

Logical address 4 is page 1, offset 0; according to the page table, page 1 is mapped to frame 6. Thus, logical address 4 maps to physical address 24 [= (6× 4) + 0]. Logical address 13 maps to physical address 9.

As page tables are per-process data structures, a pointer to the page table is stored with the other register values in the process control block of each process. When the CPU scheduler selects a process for execution, it must reload the user registers and the appropriate hardware page-table values from the stored user page table.

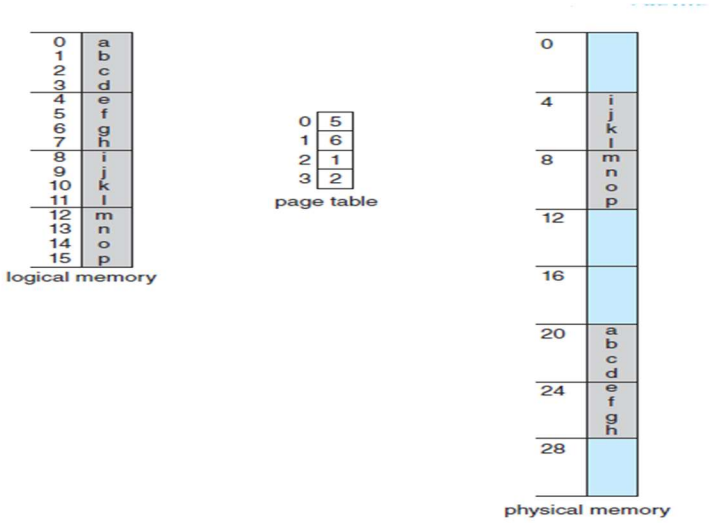


Figure Paging example for a 32-byte memory with 4-byte pages.

The hardware implementation of the page table can be done in several ways. In the simplest case, the page table is implemented as a set of **dedicated high-speed hardware registers**, which makes the page-address translation very efficient. However, this approach **increases context-switch time**, as each one of these registers must be exchanged during a context switch.

For contemporary CPU's, the use of fast registers to implement the page table is not feasible. Rather, the page table is kept in main memory, and a **page-table base register (PTBR)** points to the page table. Changing page tables requires changing only this one register, substantially reducing context-switch time. With the above scheme, two memory accesses are needed in order to access a byte(one for the page-table entry and one for byte). Thus memory access is slower by a factor of 2 and in most cases, this scheme slowed by a factor of 2.

There is the standard solution for the above problem that is to use a special, small, and fast-lookup hardware cache that is commonly known as **Translation of look-aside buffer(TLB).** TLB is associative and high-speed memory. Each entry in the TLB mainly

consists of two parts: **a key(that is the tag) and a value.** When associative memory is presented with an item, then the item is compared with all keys simultaneously. In case if the item is found then the corresponding value is returned.

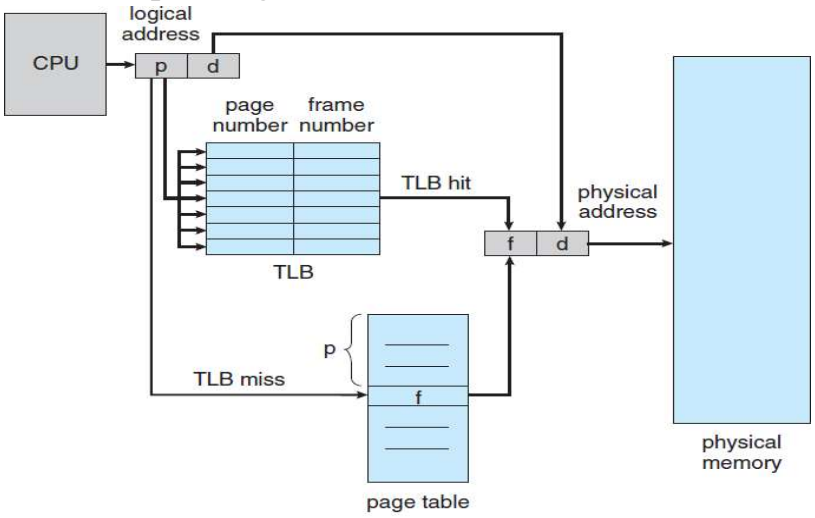


Figure Paging hardware with TLB.

The search with TLB is fast though the hardware is expensive. The number of entries in the TLB is small and generally lies in between 64 and 1024. The TLB contains only a few of the page-table entries. Whenever the logical address is generated by the CPU then its page number is presented to the TLB. If the page number is found, then its frame number is immediately available and is used in order to access the memory. The above whole task may take less than 10 percent longer than would if an unmapped memory reference were used. In case if the page number is not in the TLB (which is known as **TLB miss**), then a memory reference to the Page table must be made. When the frame number is obtained it can be used to access the memory. Additionally, page number and frame number is added to the TLB so that they will be found quickly on the next reference. In case if the **TLB is already full of entries** then the operating system must select **one for replacement.**

STRUCTURE OF PAGE TABLE

Some of the characteristics of the Page Table are as follows:

- It is stored in the main memory.
- Generally the Number of entries in the page table = the Number of Pages in which the process is divided.
- **PTBR** means page table base register and it is basically used to hold the base address for the page table of the current process.
- Each process has its own independent

page table. The most common techniques for

structuring the page table,

- **Hierarchical Paging**
- **Hashed Page Tables**
- **Inverted Page Tables**

HIERARCHICAL PAGING

Most modern computer systems support a large logical address space (232 to 264). In such an environment, the page table itself becomes excessively large. There might be a case where the page table is too big to fit in a contiguous space. One simple solution to this problem is to divide the page table into smaller pieces. In this type of Paging the logical address space is broke up into Multiple page tables.

Hierarchical Paging is one of the simplest techniques and for this purpose, a **two-level page table** and
three-level page table can be used.

Two-level paging algorithm

For example, consider again the system with a 32-bit logical address space and a page size of 4 KB. A logical address is divided into a page number consisting of 20 bits and a page offset consisting of 12 bits. Because we page the page table, the page number is further divided into a 10-bit page number and a 10-bit page offset.

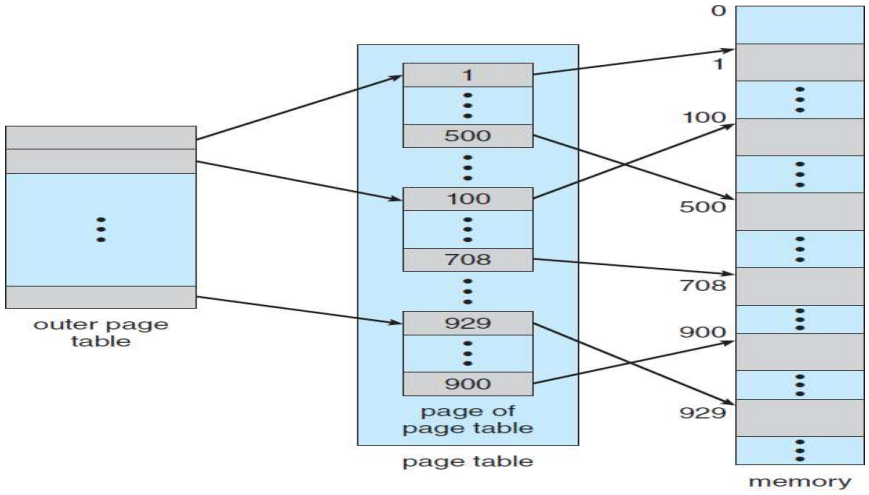
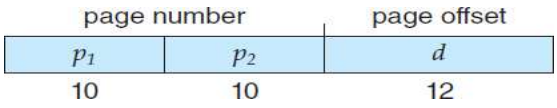


Figure A two-level page-table scheme.

Thus, a logical address is as follows where

p1 is an index into the **outer page table** and **p2 is the displacement** within the **page of the inner page table.**



The address-translation method for this architecture

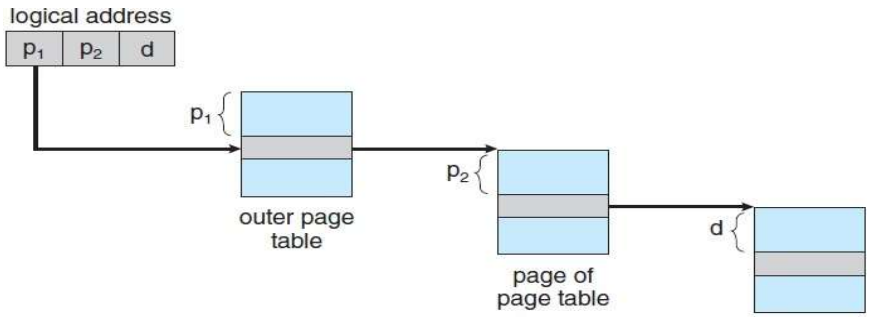
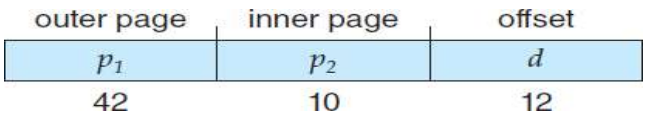


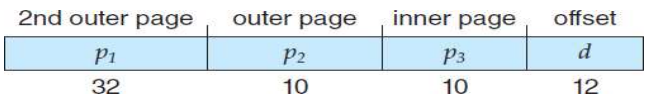
Figure Address translation for a two-level 32-bit paging architecture.

Three-level paging algorithm

For a system with a 64-bit logical address space, a two-level paging scheme is no longer appropriate. Let us suppose that the page size, in this case, is 4KB. If in this case, we will use the two-page level scheme then the addresses will look like this:



Thus in order to avoid such a large table, there is a solution and that is to divide the outer page table, and then it will result in a **Three-level page table**



HASHED PAGE TABLE

One approach for handling address spaces larger than 32 bits is to use a hashed page table, with the **hash value** being the virtual page number**.** Each entry in the hash table contains a **linked list** of elements that hash to the same location to handle **collisions.**

Each element consists of three fields:

- The virtual page number
- The value of the mapped page frame
- A pointer to the next element in the linked list

The algorithm works as follows:

- The virtual page number in the virtual address is hashed into the hash table.
- The virtual page number is compared with field 1 in the first element in the linked list.
- If there is a match, the corresponding page frame (field 2) is used to form the desired physical address.
- If there is no match, subsequent entries in the linked list are searched for a matching virtual page number.

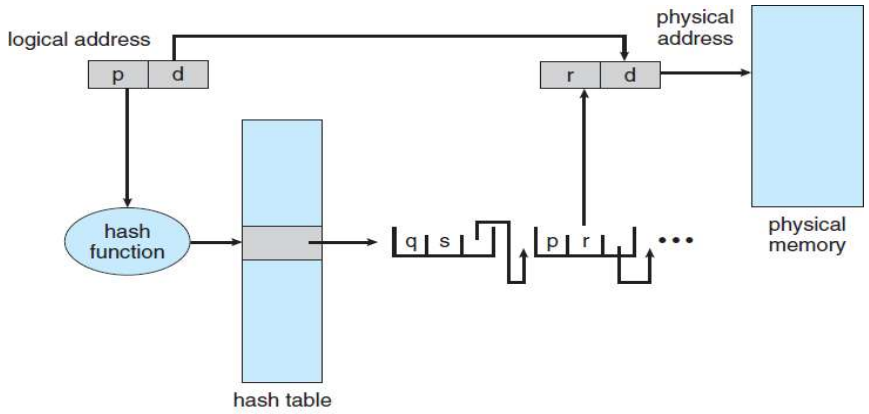


Figure Hashed page table.

INVERTED PAGE TABLES

Usually, each process has an associated page table. The page table has one entry for each page that the process is using. Since the table is sorted by virtual address, the operating system is able to calculate where in the table the associated physical address entry is located and to use that value directly One of the drawbacks of this method is that each page table may consist of millions of entries. These tables may consume large amounts of physical memory just to keep track of how other physical memory is being used.

To solve this problem, we can use an **inverted page table.** An inverted page table has one entry for each real page (or frame) of memory. Each entry consists of the virtual address of the page stored in that real memory location, with information about the process that owns the page. Thus, only one page table is in the system, and it has only one entry for each page of physical memory. Inverted page tables often require that an address-space identifier be stored in each entry of the page table, since the table usually contains several different address spaces mapping physical memory. Storing the address-space identifier ensures that a logical page for a particular process is mapped to the corresponding physical page frame.

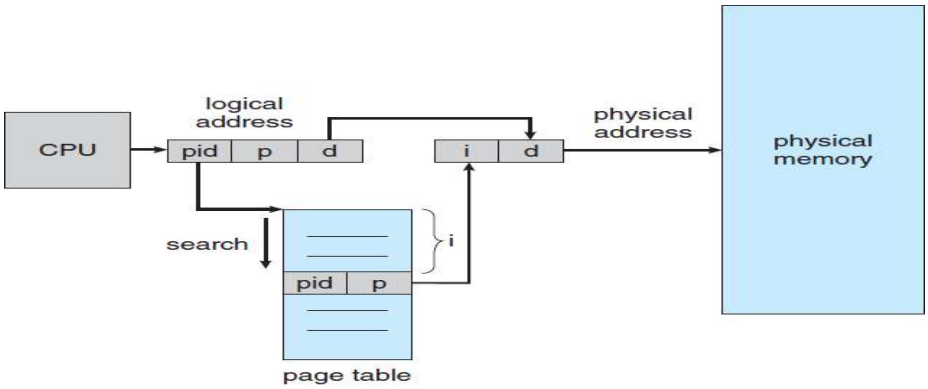


Figure Inverted page table.

SWAPPING

Process instructions and the data they operate on must be in memory to be executed. However, a process, or a portion of a process, can be **swapped** temporarily out of memory to a **backing store** and then brought back into memory for continued execution.

Swapping involves moving entire processes between main memory and a **backing store.** It will be later brought back into the memory for continue execution. **Backing store** is a hard disk or some other secondary storage device that should be big enough in order to accommodate copies of all memory images for all users.

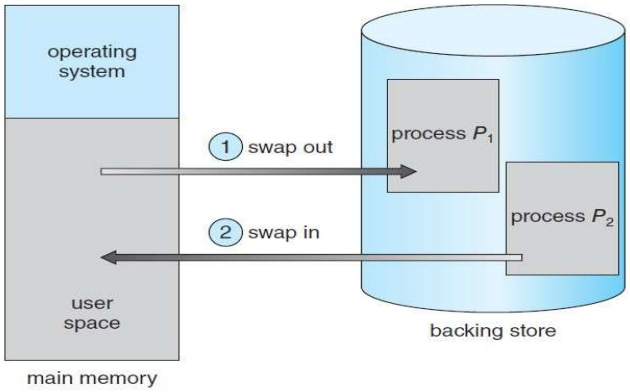


Figure Standard swapping of two processes using a disk as a backing store.

Swapping a process temporarily out of main memory to secondary storage and make memory available to other processes is known as **Swap-out** operation. At some later time, the system swaps back the process from the secondary storage to main memory is known as **Swap-in** operation. Performance is usually affected by swapping process but it helps in running multiple and big processes in parallel.

Major benefits of swapping are

- It offers a higher degree of multiprogramming

- Allows dynamic relocation
- It helps to get better utilization of memory
- Minimum wastage of CPU time

VIRTUAL MEMORY

Virtual memory is a technique that allows the execution of processes that are not in the **physical memory completely**. User can load the **bigger size processes** than the available main memory by having the **illusion** that the memory is available to load the process. Instead of loading one big process in the main memory, the Operating System loads the **different parts of more than one process** in the main memory.

In real scenarios, most processes never need all their pages at once, for the following reasons :

- **Error handling code** is not needed unless that specific error occurs, some of which are quite rare.
- **Arrays** are often over-sized for worst-case scenarios, and only a small fraction of the arrays are actually used in practice.
- Certain **features** of certain programs are rarely used.

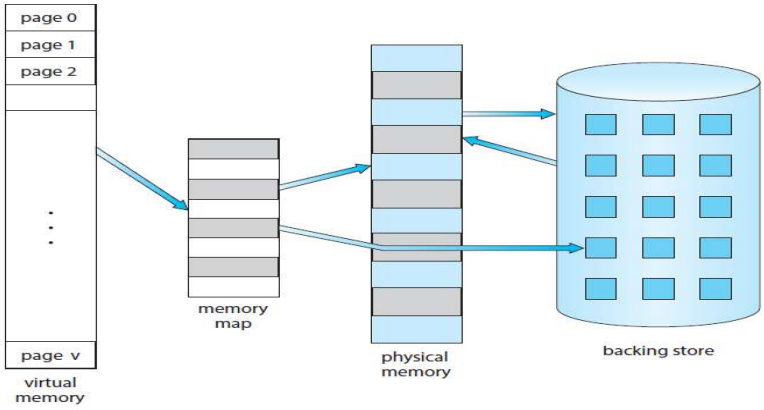


Figure Diagram showing virtual memory that is larger than physical memory.

The main visible advantage of this scheme is that programs can be larger than physical memory. Therefore, instead of loading one long process in the main memory, the OS loads the various parts of more than one process in the main memory. Virtual memory is mostly implemented with **demand paging**.

DEMAND PAGING

Consider how an executable program might be loaded from secondary storage into memory. One option is to load the entire program in physical memory at program execution time. However, a problem with this approach is that we may not initially need

the entire program in memory.

Suppose a program starts with a list of available options from which the user is to select. Loading the entire program into memory results in loading the executable code for ***all*** options, regardless of whether or not an option is ultimately selected by the user. An alternative strategy is to load pages only as they are needed. This technique is known as **demand paging** and is commonly used in virtual memory systems.

With demand-paged virtual memory, **pages are loaded only when they are demanded** during program execution. Demand paging explains one of the primary benefits of virtual memory **by loading only the portions of programs** that are needed, memory is used more efficiently. The general concept behind demand paging, is to load a page in memory only when it is needed. As a result, while a process is executing, some pages will be in memory, and some will be in secondary storage. Thus, we need some form of hardware support to distinguish between the two. The valid– invalid bit scheme can be used for this purpose.

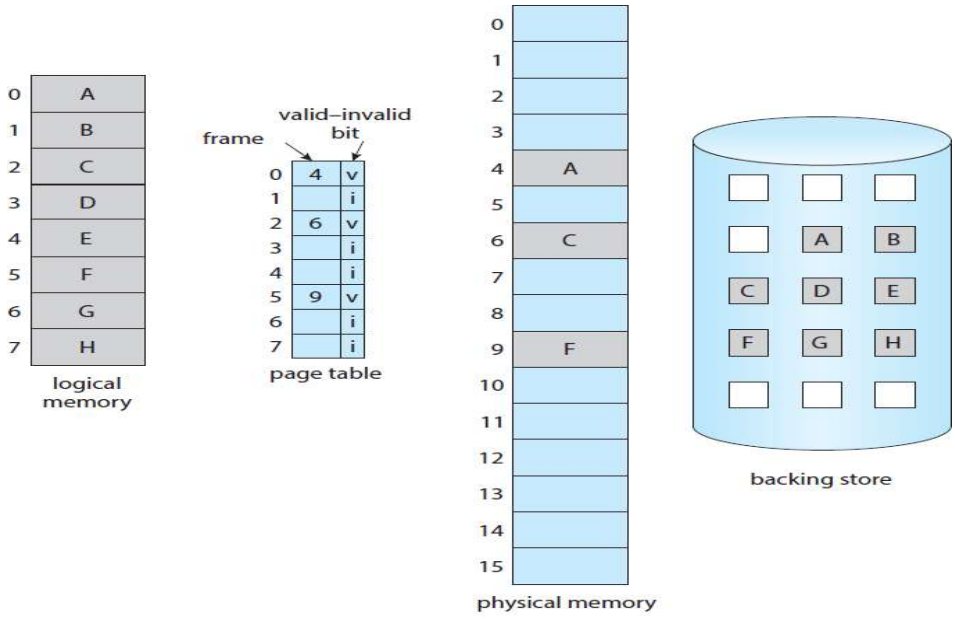


Figure 10.4 Page table when some pages are not in main memory.

When the bit is set to **valid**, the **associated page is both legal and in memory.** If the bit is set to **invalid,** the page **either is not valid or is valid but is currently in secondary storage.** The page-table entry for a page that is brought into memory is set as usual, but the page-table entry for a page that is not currently in memory is simply marked invalid.

But 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

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 secondary storage operation to **read the desired page into the newly allocated frame**.
5. When the storage 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.

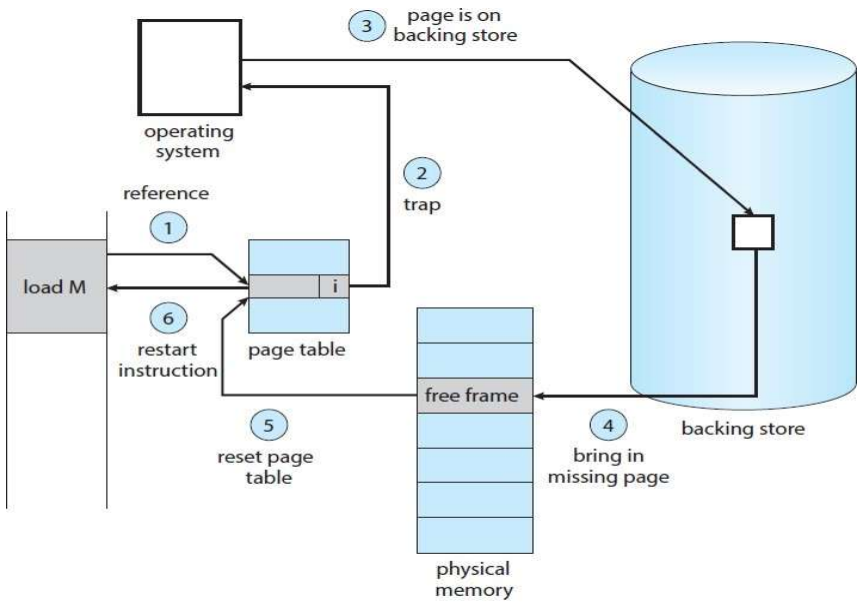


Figure 10.5 Steps in handling a page fault.

In the **extreme case**, we can start executing a process with **no pages in memory.** When the operating system sets the instruction pointer to the first instruction of the process, which is on a non-memory-resident page, the **process immediately faults for the page.** After this page is brought into memory, the process continues to execute, faulting as necessary until

every page that it needs is in memory. At that point, it can execute with no more faults. This scheme is **pure demand paging: never bring a page into memory until it is required.**

COPY-ON-WRITE

Copy-on-Write (CoW) plays a crucial role in **virtual memory management**, enhancing efficiency by deferring memory duplication until it's absolutely necessary. Virtual memory systems use CoW to optimize memory allocation and sharing between processes, reducing overhead while maintaining data integrity.

How CoW Works in Virtual Memory Management

1. **Shared Memory Pages:**
   - When a new process is created (e.g., using fork()), the operating system doesn't immediately copy the memory pages of the parent process. Instead:
     - The parent and child processes share the same physical memory pages.
     - These pages are marked as **read-only** in the page tables of both processes.
2. **Page Fault on Write:**
   - If either process tries to **write** to a shared page:
     - The CPU triggers a **page fault** because the page is marked as read-only.
     - The operating system's page fault handler steps in, allocates a new physical page, and copies the contents of the original page into it.
     - The process that initiated the write operation gets the new writable page, while the other process continues using the original read-only page.
3. **Efficient Reads:**
   - As long as the memory is only read, the shared pages remain intact, avoiding unnecessary duplication.

Steps in CoW in Virtual Memory Systems

1. **Mark Pages as Read-Only:**
   - Initially, the memory pages are shared between processes and marked read-only in their respective page tables.
2. **Trigger Page Fault:**
   - When a process writes to a shared page, the CPU raises a **page fault** because of the read-only protection.
3. **Duplicate the Page:**
   - The OS duplicates the memory page to create a private copy for the writing process.
   - The page table of the writing process is updated to map the new physical page, now marked as writable.
4. **Update TLB and Cache:**
   - The Translation Lookaside Buffer (TLB) and caches are updated to reflect the changes in memory mappings.

Key Benefits of CoW in Virtual Memory

1. **Reduced Memory Usage:**
   - Pages are duplicated only when necessary, saving memory in cases where shared memory pages remain unchanged.
2. **Fast Process Creation:**
   - In fork(), the child process shares the parent's memory space, making process creation faster and lightweight.
3. **Optimized Resource Utilization:**
   - CoW ensures that physical memory is used efficiently, especially in systems where processes often share common data.

Applications in Virtual Memory

1. **Process Management:**
   - CoW is heavily used during process creation (fork()) in Unix-like operating systems.
   - It allows the child process to share the parent's memory until one of them modifies the data.
2. **Shared Libraries:**
   - Processes can share read-only memory pages containing common libraries. If a process modifies the data, a private copy is created.
3. **Memory-Mapped Files:**
   - CoW is used for memory-mapped files to enable processes to share file-backed memory regions while allowing modifications.
4. **Snapshots and Checkpoints:**
   - Some virtual memory systems use CoW for creating snapshots of memory states, especially in virtualized environments or during debugging.

Challenges of CoW in Virtual Memory

1. **Page Fault Overhead:**
   - While CoW optimizes memory usage, the page fault mechanism introduces latency when a write operation occurs.
2. **Complexity:**
   - Implementing CoW in virtual memory systems requires careful management of page tables, TLB entries, and synchronization.
3. **Write-Heavy Workloads:**
   - In scenarios with frequent writes, CoW may lead to excessive page copying, reducing its benefits.

Example: CoW in Linux Virtual Memory

1. **Fork Mechanism:**
   - In Linux, when fork() is called, all memory pages are marked read-only, and the child process shares the parent's pages.
   - On a write, the do_wp_page() function in the Linux kernel handles the page fault, creating a new page and updating the page tables.
2. **Memory Map (/proc):**
   - You can observe shared memory regions using the /proc/[pid]/maps file, which lists the virtual memory regions of a process.
3. **Performance Optimization:**
   - Linux also uses CoW for file-backed memory regions, enabling efficient file sharing across processes.

Summary

In virtual memory management, **Copy-on-Write** optimizes memory usage by delaying memory duplication until a write operation occurs. It enhances the efficiency of process creation, shared memory management, and memory-mapped files. However, its effectiveness depends on the workload, with read-heavy tasks benefiting the most. By combining CoW with virtual memory techniques, operating systems achieve a balance between performance and resource efficiency.

PAGE REPLACEMENT

In Demand Paging, only certain pages of a process are loaded initially into the memory. This allows us to get more processes into memory at the same time. But what happens when a process requests for more pages and no free memory is available to bring them in.

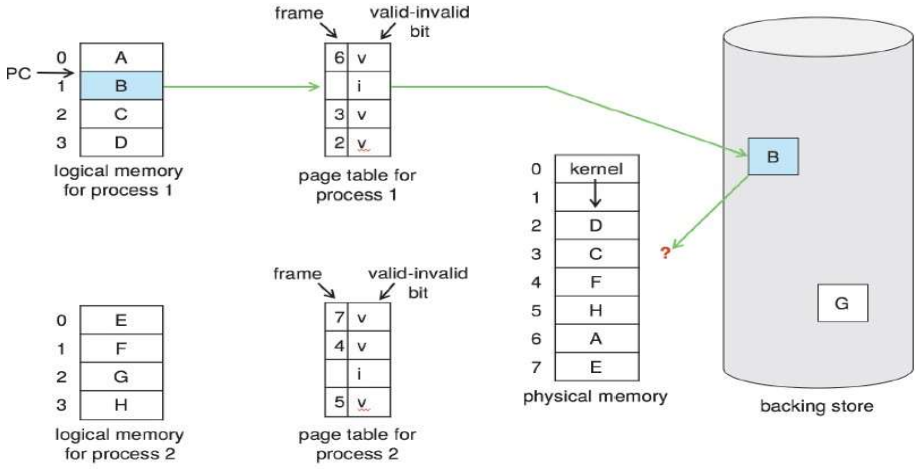


Figure Need for page replacement.

Following steps can be taken to deal with this problem :

- Put the **process in the wait queue**, until any other process finishes its execution thereby freeing frames.
- **Remove some other process completely** from the memory to free frames.
- **Find some pages that are not being used** right now, move them to the

disk to get free frames. This technique is called **Page replacement** and is most

commonly used.

BASIC 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.
- 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 secondary storage.
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 secondary storage (if necessary); 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 process from where the page fault occurred.

Notice that, if no frames are free, two page transfers (one for the page-out and one for the page-in) are required. This situation effectively doubles the page-fault service time and increases the effective access time accordingly.

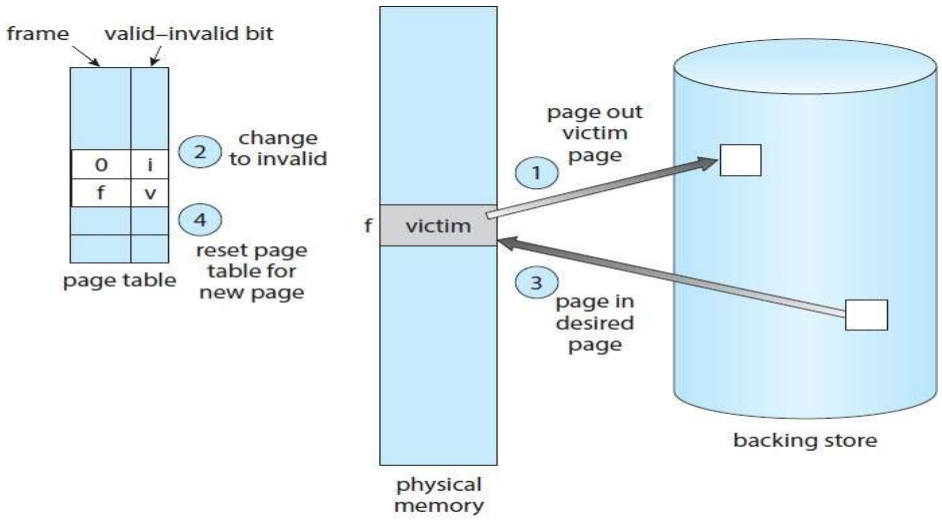


Figure Page replacement.

We can reduce this overhead by using a **modify bit (or dirty bit).** When this scheme is used, each page or frame has a modify bit associated with it in the hardware. The modify bit for a page is set by the hardware whenever any byte in the page is written into, indicating that the page has been modified. When we select a page for replacement, we examine it's modify bit. If the **bit is set** we know that the page has been modified since it was read in from secondary storage. In this case, we must **write the page to storage.** 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 storage**: it is already there.

There are many different page-replacement algorithms. They are

- **FIFO page replacement**
- **Optimal Page Replacement**
- **Least Recently Used (LRU) Page Replacement**
- **Least Frequently Used (LFU) page Replacement**

We evaluate an algorithm by running it on a particular string of memory references and computing the number of page faults. The string of memory references is called a **reference string.** To determine the number of page faults for a particular reference string and page-replacement algorithm, we also need to know the **number of page frames** available.

To reduce the number of data, we use two facts

- First, for a given page size we need to consider only the page number, rather than the entire address.

- Second, if we have a reference to a page ***p***, then any references to page ***p*** that immediately follow will never cause a page fault.
- Page ***p*** will be in memory after the first reference, so the immediately following references will not fault.

PAGE REPLACEMENT ALGORITHMS

FIFO PAGE REPLACEMENT

The simplest page-replacement algorithm is a first-in, first-out (FIFO) algorithm. A FIFO replacement algorithm associates with each page the time when that page was brought into memory. When a page must be replaced, the oldest page is chosen. We can create a FIFO queue to hold all pages in memory. We replace the page at the head of the queue. When a page is brought into memory, we insert it at the tail of the queue.

Consider the reference string for a memory with three frames.

7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1

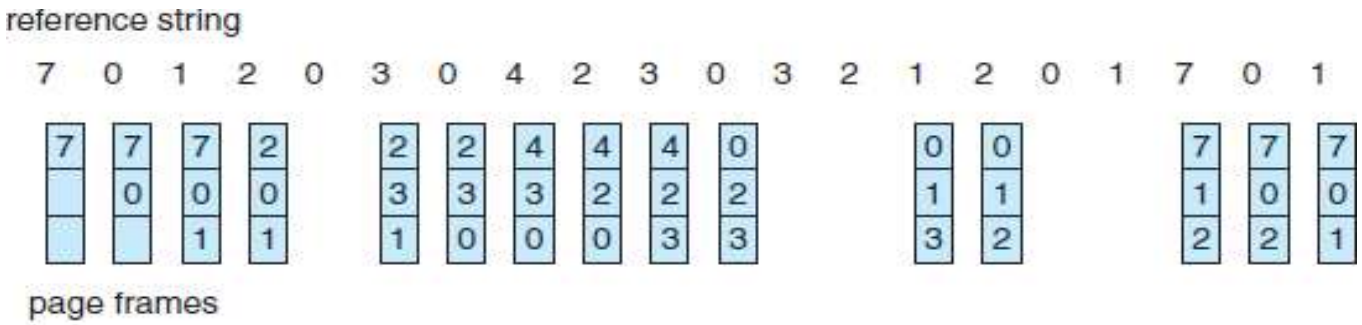


Figure 10.12 FIFO page-replacement algorithm.

- **Total number of reference strings** = 20
- **Total number of page faults or page misses** = 15
- **Total number of page hits = Total number of reference strings - Total number of page faults**

  = 20 – 15

  = 5
- **Page fault probability = Total number of page faults / Total number of reference strings**

  = 15/20

  = 0.75

- **Page fault percentage = Tot number of page faults / Tot number of reference strings * 100**

  = 0.75*100

  = 75%

The FIFO page-replacement algorithm is easy to understand and program but its performance is not always good. Notice that, even if we select for replacement a page that is in active use, everything still works correctly. After we replace an active page with a new one, a fault occurs almost immediately to retrieve the active page. Some other page must be replaced to bring the active page back into memory. Thus, a bad replacement choice increases the page-fault rate and slows process execution.

To illustrate the problems that are possible with a FIFO page-replacement algorithm, consider the following reference string: **1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5**

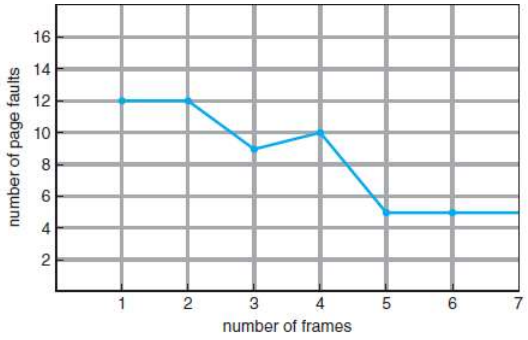


Figure Page-fault curve for FIFO replacement on a reference string.

Figure shows the curve of page faults for this reference string versus the number of available frames. Notice that the number of faults for four frames (ten) is ***greater*** than the number of faults for three frames (nine)! This most unexpected result is known as **Belady's anomaly**: for some page-replacement algorithms, the page-fault rate may ***increase*** as the number of allocated frames increases.

OPTIMAL PAGE REPLACEMENT

Belady's anomaly means for some page-replacement algorithms, the page-fault rate may increase as the number of allocated frames increases. We would expect that giving more memory to a process would improve its performance. In some early research, investigators noticed that this assumption was not always true. Belady's anomaly was discovered as a result.

One result of the discovery of Belady's anomaly was the search for an **optimal page-replacement algorithm.** The algorithm that has the lowest page-fault rate of all algorithms and will never suffer from Belady's anomaly. Such an algorithm does exist and has been called OPT or MIN. It is simply **Replace the page that will not be used**

for the longest period of time.

For example, Consider the reference string for a memory with three frames.

7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1

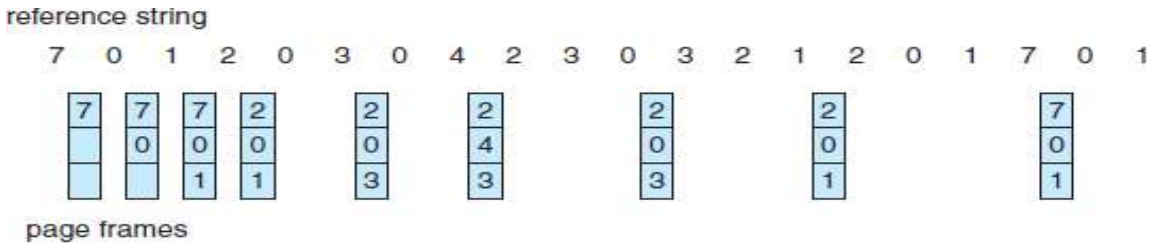


Figure Optimal page-replacement algorithm.

As shown in figure the optimal page-replacement algorithm would yield nine page faults. The first three references cause faults that fill the three empty frames. The reference to page 2 replaces page 7, because page 7 will not be used until reference 18, whereas page 0 will be used at 5, and page 1 at 14. The reference to page 3 replaces page 1, as page 1 will be the last of the three pages in memory to be referenced again. With only nine page faults, optimal replacement is much better than a FIFO algorithm, which results in fifteen faults.

With only nine page faults, optimal replacement is much better than a FIFO algorithm, which results in fifteen faults. In fact, no replacement algorithm can process this reference string in three frames with fewer than nine faults. Unfortunately, the optimal page-replacement algorithm is difficult to implement, because it requires future knowledge of the reference string. As a result, the optimal algorithm is used mainly for comparison studies.

LRU PAGE REPLACEMENT

If the optimal algorithm is not feasible, perhaps an approximation of the optimal algorithm is possible. The key distinction between the FIFO and OPT algorithms is that the FIFO algorithm uses the time when a page was brought into memory, whereas the OPT algorithm uses the time when a page is to be used. If we use the recent past as an approximation of the near future, then we can replace the page that has not been used for the longest period of time. This approach is the Least Recently Used (LRU) Algorithm. LRU replacement associates with each page the time of that page's last use.

For example, consider the reference string for a memory with three frames.

7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1

The result of applying LRU replacement to our example reference string is

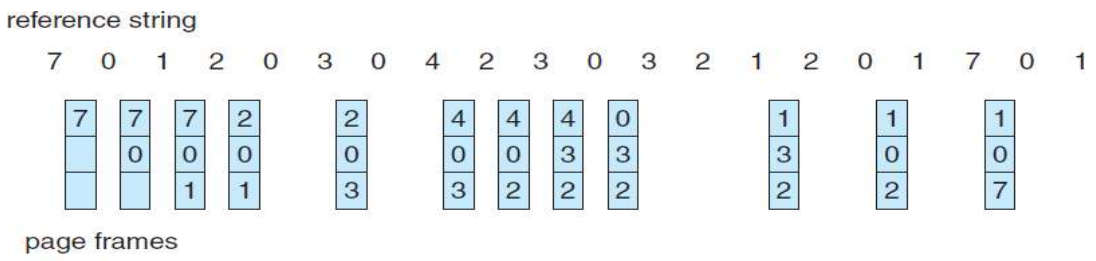


Figure LRU page-replacement algorithm.

The LRU algorithm produces twelve faults. Notice that the first five faults are the same as those for optimal replacement. When the reference to page 4 occurs, however, LRU replacement sees that, of the three frames in memory, page 2 was used least recently. Thus, the LRU algorithm replaces page 2, not knowing that page 2 is about to be used.When it then faults for page 2, the LRU algorithm replaces page 3, since it is now the least recently used of the three pages in memory. Despite these problems, LRU replacement with twelve faults is much better than FIFO replacement with fifteen.

Despite these problems, LRU replacement with **twelve faults** is much better than FIFO replacement with fifteen. The LRU policy is often used as a page-replacement algorithm and **is considered to be good.** The major problem is **how to implement LRU replacement.** An LRU page-replacement algorithm may require substantial **hardware assistance.** The problem is to **determine an order** for the frames defined by the time of last use.

Two implementations are feasible:

- **COUNTERS**
- **STACK**

COUNTERS

In the simplest case, we associate **with each page-table entry a time-of-use field** and **add to the CPU a logical clock or counter.** The clock is **incremented** for every memory reference. Whenever a reference to a page is made, the contents of the clock register are copied to the time-of-use field in the page-table entry for that page. In this way, we always have the "time" of the last reference to each page. We **replace the page with the smallest time value**. This scheme requires **a search of the page table** to find the LRU page and **a write to memory** (to the time-of-use field in the page table) **for each memory access**. The **times must also be maintained** when page tables are changed (due to CPU scheduling). **Overflow of the clock** must be considered.

STACK

Another approach to implementing LRU replacement is to keep a stack of page numbers. Whenever a page is referenced, it is removed from the stack and put on the top. In this way, the most recently used page is always at the top of the stack, and the least recently used page is always at the bottom. It is best to implement this approach by using a **doubly linked list** with **a head pointer** and **a tail pointer.**

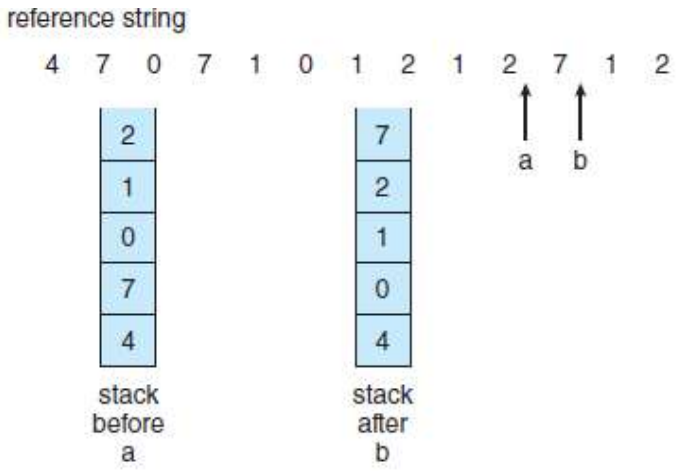
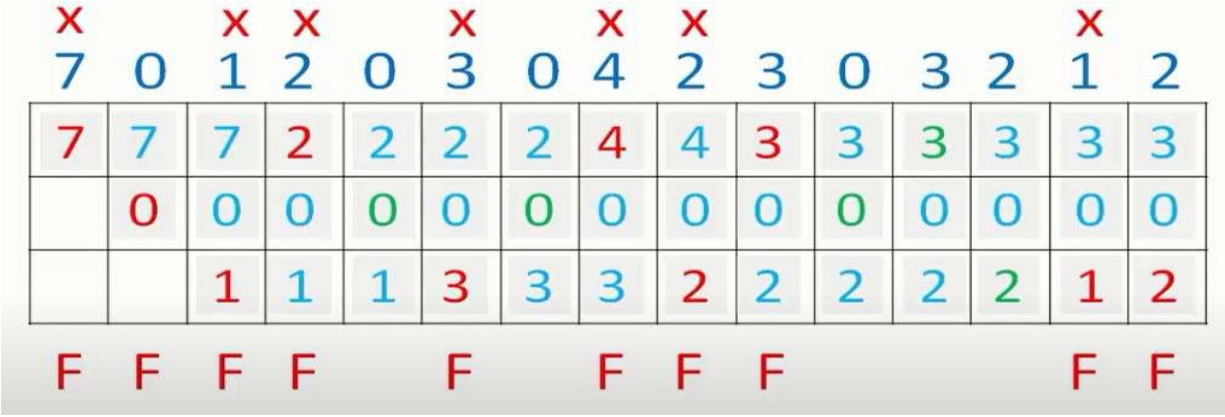


Figure Use of a stack to record the most recent page references.

LFU PAGE REPLACEMENT

The **Least Frequently Used (LFU)** page-replacement algorithm requires that the **page with the smallest count be replaced.** The reason for this selection is that an actively used page should have a large reference count. A problem arises, however, when a page is used heavily during the initial phase of a process but then is never used again. Since it was used heavily, it has a large count and remains in memory even though it is no longer needed.



ALLOCATION OF FRAMES

The main memory of the system is divided into frames. The OS has to allocate a sufficient number of frames for each process. How do we allocate the fixed amount of free memory among the various processes? If we have 93 free frames and two processes, how many frames does each process get?

Consider a simple case of a system with 128 frames. The operating system may take 35, leaving 93 frames for the user process. Under pure demand paging, all 93 frames would initially be put on the free-frame list. When a user process started execution, it would generate a sequence of page faults. The first 93 page faults would all get free frames from the free-frame list. When the free-frame list was exhausted, a page-replacement algorithm would be used to select one of the 93 in-memory pages to be replaced with the 94th, and so on. When the process terminated, the 93 frames would once again be placed on the free-frame list.

MINIMUM NUMBER OF FRAMES

Our strategies for the allocation of frames are constrained in various ways.

- We cannot allocate more than the total number of available frames.
- We must also allocate at least a minimum number of frames.

One reason for allocating at least a minimum number of frames involves **performance.** Obviously, as the number of frames allocated to each process **decreases,** the page-fault rate **increases**, slowing process execution. **The minimum number of frames per process is defined by the architecture**, **the maximum number is defined by the amount of available physical memory.** In between, we are still left with significant choice in frame allocation.

ALLOCATION ALGORITHMS

- **EQUAL ALLOCATION**
- **PROPOTIONAL ALLOCATION**
- **PRIORITY ALLOCATION**

EQUAL ALLOCATION

The easiest way to split *m* frames among *n* processes is to give everyone an equal share, ***m/n*** **frames.** For instance, if there are 93 frames and 5 processes, each process will get 18 frames. The 3 leftover frames can be used as a free-frame buffer pool. This scheme is called **equal allocation.**

PROPOTIONAL ALLOCATION

An alternative is to recognize that various processes will need differing amounts of memory. Consider a system with a 1-KB frame size. If a small student process of 10 KB and an interactive database of 127 KB are the only two processes running in a system with 62 free frames, it does not make much sense to give each process 31 frames. The student process does not need more than 10 frames, so the other 21 are, strictly speaking, wasted. To solve this problem, we can use **proportional allocation,** in which we allocate available memory to each process according to its **size.**

Let the size of the virtual memory for process p_i be s_i, and define $S = \Sigma s_i$.

Then, if the total number of available frames is m, we allocate a_i frames to process p_i, where a_i is approximately

$$a_i = s_i/S \times m$$

With proportional allocation, we would split 62 frames between two processes, one of 10 pages and one of 127 pages, by allocating 4 frames and 57 frames, respectively, since

$$10/137 \times 62 \approx 4$$

$$127/137 \times 62 \approx 57$$

In this way, both processes share the available frames according to their "needs," rather than equally.

PRIORITY ALLOCATION

With either equal or proportional allocation, a high-priority **process is treated the same** as a low- priority process. However, we may want to give the **high-priority process more memory to speed its execution**, to the detriment of low-priority processes. One solution is to use a proportional allocation scheme wherein the ratio of frames depends not on the relative sizes of processes but rather on the **priorities of processes or on a combination of size and priority.**

GLOBAL VERSUS LOCAL ALLOCATION

Another important factor in the way frames are allocated to the various processes is **page replacement.** With **multiple processes competing for frames**, we can classify page-replacement algorithms into two broad categories:

- **Global replacement**
- **Local replacement**

Global Replacement allows a process **to select a replacement frame from the set of all frames**, even if that frame is currently allocated to some other process; that is, **one process can take a frame from another. Local Replacement** requires that **each process select from only its own set of allocated frames.**

THRASHING

Consider what occurs if a process does not have "**enough" frames** i.e., it does not have the minimum number of frames it needs to support pages in the working set. The process will quickly page-fault. At this point, it must replace some page. However, since all its pages are in active use, it must replace a page that will be needed again right away. Consequently, it **quickly faults again**, and again, and again, replacing pages that it must bring back in immediately. This **high paging** activity is called **thrashing.**

A process is thrashing if it is spending **more time paging than executing.** As you might expect, thrashing results in **severe performance problems.**

This phenomenon is illustrated in figure, in which CPU utilization is plotted against the degree of multiprogramming.

- As the degree of multiprogramming increases, CPU utilization also increases, although more slowly, until a maximum is reached.
- If the degree of multiprogramming is increased further, thrashing sets in, and CPU utilization drops sharply.
- At this point, to increase CPU utilization and stop thrashing, we must decrease the degree of multiprogramming.

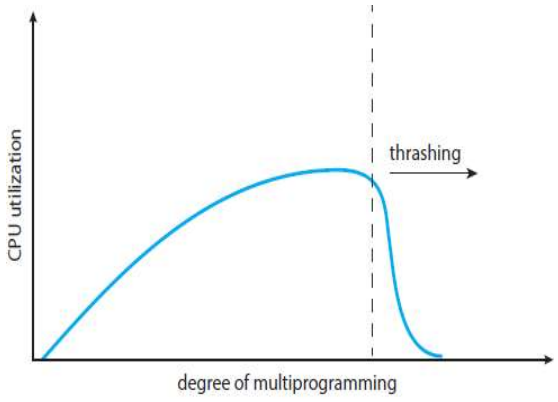


Figure Thrashing.

EFFECT OF THRASHING

Whenever thrashing starts, operating system tries to apply either

- **Global Page Replacement Algorithm**
- **Local Page Replacement Algorithm**

Global Page Replacement

Since global page replacement can access to bring any page, it tries to bring more

pages whenever thrashing found. But what actually will happen is, due to this, no process gets enough frames and by result thrashing will be increase more and more. So global page replacement algorithm is not suitable when thrashing happens.

Local Page Replacement

Unlike global page replacement algorithm, local page replacement will select pages which only belongs to that process. So there is a chance to reduce the thrashing. But it is proven that there are many disadvantages if we use local page replacement. So local page replacement is just alternative than global page replacement in thrashing scenario.

Two techniques are present to handle thrashing :

- **Working Set Model**
- **Page Fault Frequency**

Working Set Model

Locality is the page used recently can be used again and also the pages which are nearby this page will also be used. Working set means set of pages in the most recent Δ time. The page which completed its Δ amount of time in working set automatically dropped from it. To find out the number of frames required by a process we can use the locality model. The working set model is based on the same locality model concept and defines a set of referenced pages with the localities to which they belong and this set is called a working set window.

The most important factor for the success of this model is the length of the working set window known as Delta Δ.

We have to choose an optimal length such that

- it is neither too small to hold all of the pages of the current locality nor
- too large to include pages that don’t belong

to the locality. For the last page reference let us

find the current working set:

Taking a fairly large Delta, Δ as 9, the working set will check the recent 9 references that have been page faulted. Those are 5,7,2,4,4,3,2,3,1, thereby the working set becomes WS= [5,7,2,4,3,1] where we have pages 5 and 7 which are hardly referenced again.

Again, taking some small Δ=2, we get WS=[1,3]. We miss frequently visiting pages like 4 and 2. The optimal value of Δ can be 5 referencing 4,3,2,3,1, WS=[2,4,3,1]. As you see, the current working set is different for different localities or page references and has to be evaluated after each page reference for the current process. The challenge in this model is keeping track of the constantly changing current working sets for each process by keeping track of pages that have been referenced so far within the delta.

Page Fault Frequency

The working-set model is successful but it seems a clumsy way to control thrashing. A strategy that uses the page-fault frequency (PFF) takes a more direct approach. Thrashing has a high page-fault rate. Thus, we want to control the page-fault rate. When it is too high, we know that the process needs more frames. Conversely, if the page fault rate is too low, then the process may have too many frames. We can establish upper and lower bounds on the desired page- fault rate.

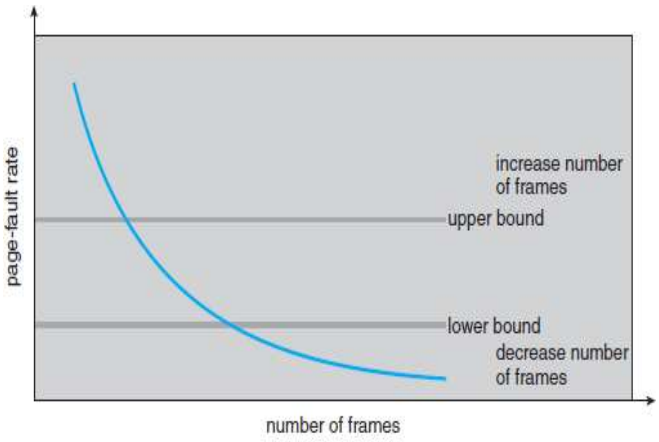


Figure Page-fault frequency.

If the actual page-fault rate exceeds the upper limit, we allocate the process another frame. If the page- fault rate falls below the lower limit, we remove a frame from the process. Thus, we can directly measure and control the page fault rate to prevent thrashing.

OVERVIEW OF MASS STORAGE STRUCTURE

The main mass-storage system in modern computers is **secondary storage.** The bulk of secondary storage for modern computers is provided by

- **Hard Disk Drives (HDD)**
- **Non-Volatile Memory (NVM) Devices**

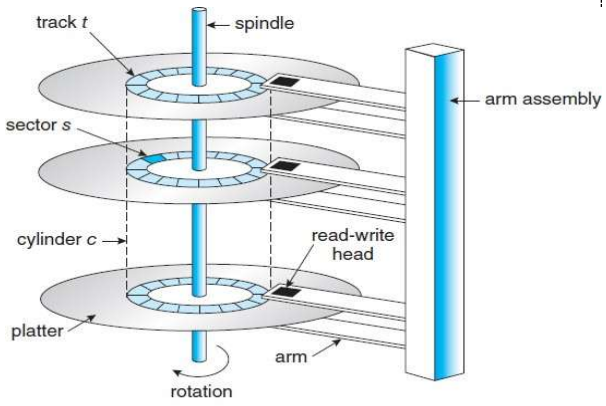


Figure HDD moving-head disk mechanism.

Hard Disk Drive Disk Structure

Conceptually, HDDs are relatively simple figure. Each disk **platter** has a flat circular shape, like a CD. The **two surfaces** of a platter are covered with a **magnetic material.** We **store** information by recording it **magnetically** on the platters. We **read** information by **detecting the magnetic pattern** on the platters. A read– write head "flies" just above each surface of every platter. The heads are attached to a **disk arm** that moves all the heads as a unit.
The surface of a platter is logically divided
into circular **tracks,** which are **subdivided into sectors.**

The set of tracks at a given arm position make up a **cylinder.** There may be thousands of **concentric cylinders** in a disk drive, and **each track** may contain **hundreds of sectors**. Each **sector has a fixed size** and is the **smallest unit of transfer.** The storage capacity of common disk drives is measured in **gigabytes** and **terabytes**. A disk drive motor spins it at high speed. Most drives rotate **60 to 250** times per second, specified in terms of rotations per minute (**RPM).**

Some drives power down when not in use and spin up upon receiving an I/O request. Rotation speed relates to transfer rates. The **transfer rate** is the rate at which data flow between the **drive and the computer.** Another performance aspect is the **positioning time, or random- access time.**

It consists of two parts:

- the time necessary to move the disk arm to the desired cylinder, called the **seek time**
- the time necessary for the desired sector to rotate to the disk head, called the **rotational latency.**

The disk head flies on an extremely thin cushion of air or another gas, such as helium. There is a danger that the head will make contact with the disk surface. Although the disk platters are coated with a thin protective layer, the head will sometimes damage the magnetic surface. This accident is called a **head crash.** A head crash normally cannot be repaired. The entire disk must be replaced, and the data on the disk are lost unless they were backed up to other storage or RAID protected.

Non-Volatile Memory Devices

Flash-memory-based NVM is frequently used in a disk-drive-like container, in which case it is called a **Solid-State Disk (SSD).** In other instances**,** it takes the form of a **USB drive** (also known as a thumb drive or flash drive) or a **DRAM stick.** It is also **surface-mounted onto motherboards** as the main storage in devices like smartphones. NVM

devices can be more reliable than HDDs because they have **no moving parts** and can be **faster** because they have no seek time or rotational latency they consume **less power**

On the negative side, they are **more expensive** per megabyte than traditional hard disks and have **less capacity** than the larger hard disks. Over time, however, the capacity of NVM devices has increased and their price has dropped more quickly, so their use is increasing dramatically.

Figure A 3.5-inch SSD circuit board.

HDD SCHEDULING

As we know, a process needs two type of time, **CPU time** and **I/O time.** For I/O, it requests the Operating system to **access the disk**. We use disk scheduling to schedule the Input/output requests that arrive for the disk. Disk Scheduling is also called **Input/output scheduling.**

Seek Time

Seek time is the time taken in locating the disk arm to a specified track where the read/write request will be satisfied.

Rotational Latency

It is the time taken by the desired sector to rotate itself to the position from where it can access the R/W heads.

Transfer Time

It is the time taken to transfer the data.

Disk Access Time

Disk access time is given as,

Disk Access Time = Rotational Latency + Seek Time + Transfer

Time Disk Response Time

It is the average of time spent by each request waiting for the IO operation.

Whenever a process needs I/O to or from the drive, it **issues a system call** to the operating system.

The request specifies several pieces of information:

- Whether this operation is input or output
- The open file handle indicating the file to operate on
- What the memory address for the transfer is
- The amount of data to transfer

If the desired **drive and controller are available**, the request can be **serviced immediately.** If the **drive or controller is busy**, any new requests for service will be **placed in the queue** of pending requests for that drive. For a **multiprogramming system** with many processes, the device queue may often have **several pending requests.** The existence of a queue of requests to a device that can have its **performance optimized**. It can be optimized by **avoiding head seeks,** allows device drivers a chance to improve performance via **queue ordering.** There are different types of disk scheduling algorithms each contains its own benefits and drawbacks.

Various disk scheduling algorithms are

- **FCFS Disk Scheduling Algorithm**
- **SSTF (Shortest Seek Time First) Disk Scheduling Algorithm**
- **SCAN Disk Scheduling Algorithm**
- **C-SCAN Disk Scheduling Algorithm**
- **LOOK Disk Scheduling Algorithm**
- **C-LOOK Disk Scheduling**

Algorithm FCFS Disk Scheduling

Algorithm

The simplest form of disk scheduling is, of course, the first-come, first-served (FCFS) algorithm (or FIFO). As the name suggests, this algorithm entertains requests in

the order they arrive in the disk queue. The algorithm looks very fair and there is no starvation (all requests are serviced sequentially) but generally, it does not provide the fastest service.

Consider, for example, a disk queue with requests for I/O to blocks on cylinders 98, 183, 37, 122, 14, 124, 65, 67, in that order. in that order. If the disk head is initially at cylinder 53, it
will first move from 53 to 98, then to 183, 37, 122, 14, 124, 65, and finally to 67, for a total head movement of 640 cylinders. This schedule is diagrammed in figure

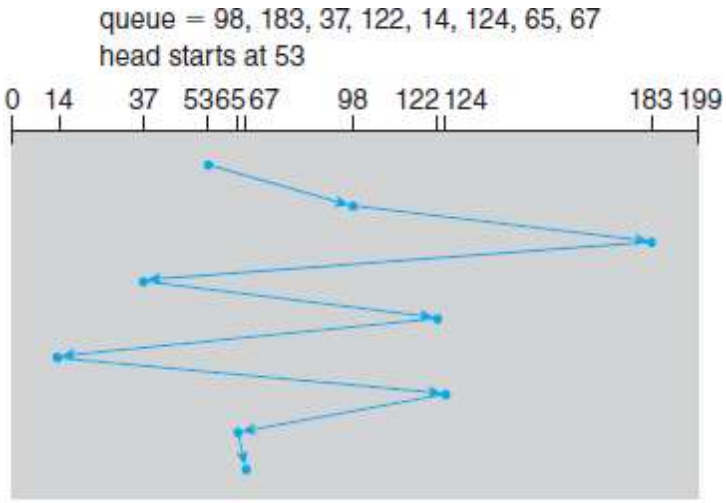


Figure FCFS disk scheduling.

- Number of cylinders moved by the head

= (98-53)+(183-98)+(183-37)+(122-37)+(122-14)+

(124-14)+(124-65)+(67-65)

= 45+85+146+85+108+110+59+2

= 640 cylinders

The wild swing from 122 to 14 and then back to 124 illustrates the problem with this schedule. If the requests for cylinders 37 and 14 could be serviced together, before or after the requests for 122 and 124, the total head movement could be decreased substantially, and performance could be thereby improved.

SSTF Disk Scheduling Algorithm

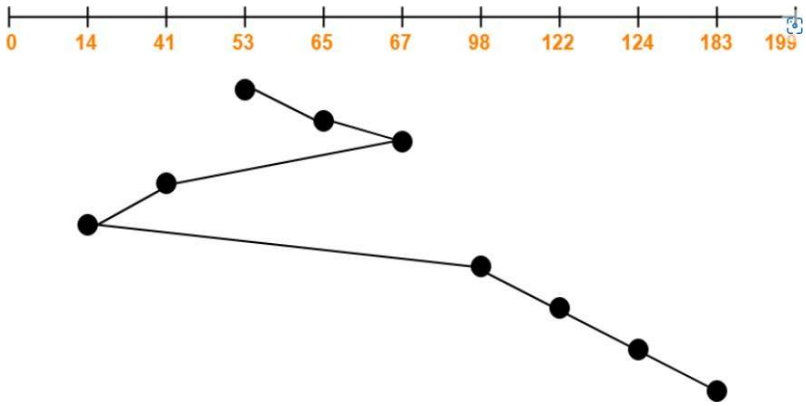
Shortest seek time first (SSTF) algorithm selects the disk I/O request which

requires the least disk arm movement from its current position regardless of the direction. It reduces the total seek time as compared to FCFS. It allows the head to move to the closest track in the service queue.

Consider a disk queue with requests for I/O to blocks on cylinders

98, 183, 41, 122, 14, 124, 65, 67.

The head is initially at cylinder number 53 moving towards larger cylinder numbers on its servicing pass.



Total head movements incurred while servicing these requests

= (65 – 53) + (67 – 65) + (67 – 41) + (41 – 14) + (98 – 14) + (122 – 98) + (124 – 122) +

(183 – 124)

= 12 + 2 + 26 + 27 + 84 + 24 + 2 + 59

= 236

SCAN Disk Scheduling Algorithm

In the **SCAN algorithm,** the disk arm starts at one end of the disk and moves toward the other end, servicing requests as it reaches each cylinder, until it gets to the other end of the disk. At the other end, the direction of head movement is reversed, and servicing continues. The head continuously scans back and forth across the disk.

Example: Before applying SCAN to schedule the requests on cylinders 98, 183, 37, 122, 14, 124, 65, and 67, we need to know the direction of head movement in addition to the head's current position. Assuming that the disk arm is moving toward 0 and that the initial head position is again 53, the head will next service 37 and then 14. At cylinder 0, the arm will reverse and will move toward the other end of the disk, servicing the

requests at 65, 67, 98, 122, 124, and 183

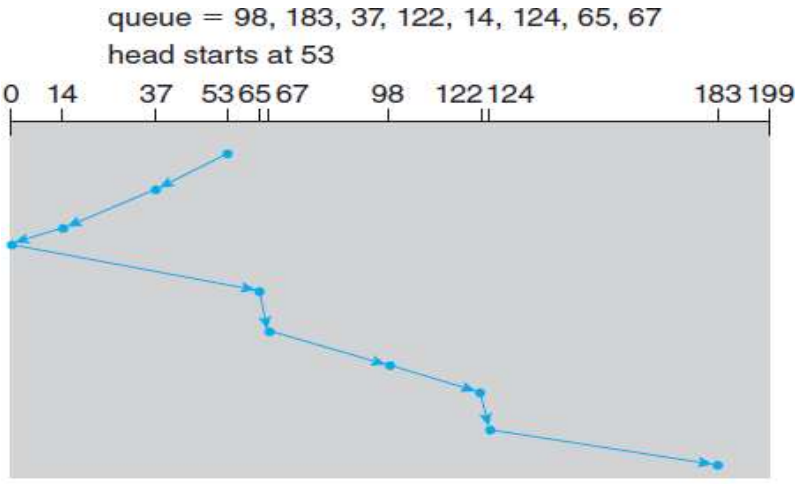


Figure SCAN disk scheduling.

CSCAN Disk Scheduling Algorithm

Circular SCAN (C-SCAN) scheduling is a variant of SCAN designed to provide a more uniform wait time. Like SCAN, C-SCAN moves the head from one end of the disk to the other, servicing requests along the way. When the head reaches the other end, however, it immediately returns to the beginning of the disk without servicing any requests on the return trip.

Example: Before applying C-SCAN to schedule the requests on cylinders 98, 183, 37, 122, 14, 124, 65, and 67 we need to know the direction of head movement in which the requests are scheduled. Assuming that the requests are scheduled when the disk arm is moving from 0 to 199 and that the initial head position is again 53, the request will be served.

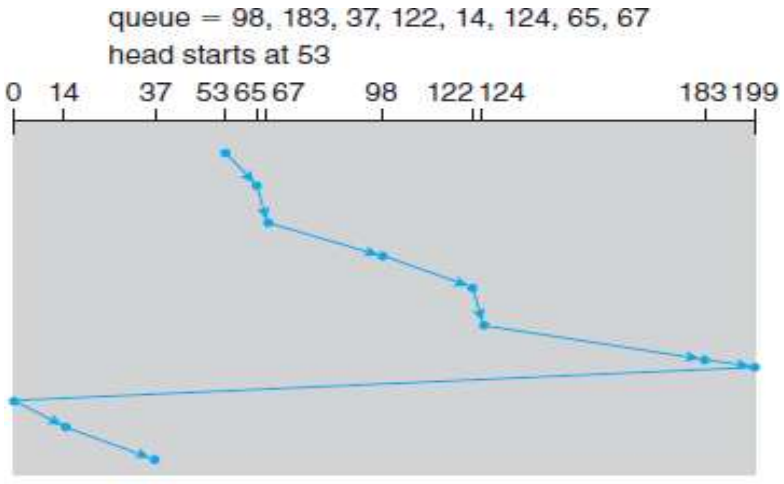
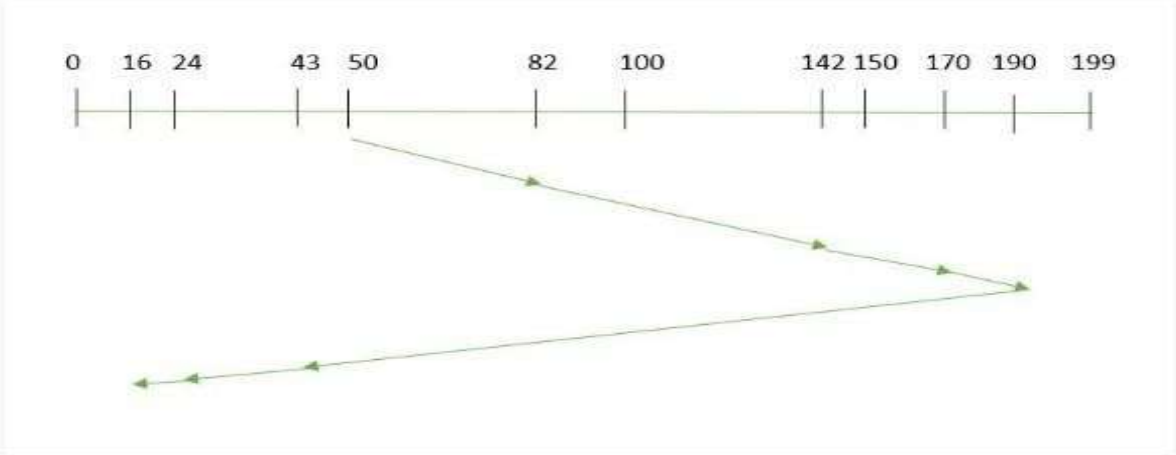


Figure C-SCAN disk scheduling.

LOOK Disk Scheduling Algorithm

LOOK is the advanced version of SCAN (elevator) disk scheduling algorithm which gives slightly better seek time than any other algorithm. The LOOK algorithm services request similarly as SCAN algorithm meanwhile it also **"looks"** ahead as if there are more tracks that are needed to be serviced in the same direction. If there are **no pending requests** in the moving direction the **head reverses the direction** and start servicing requests in the opposite direction. The main reason behind the better performance of LOOK algorithm in comparison to SCAN is because in this algorithm the head is not allowed to move till the end of the disk.



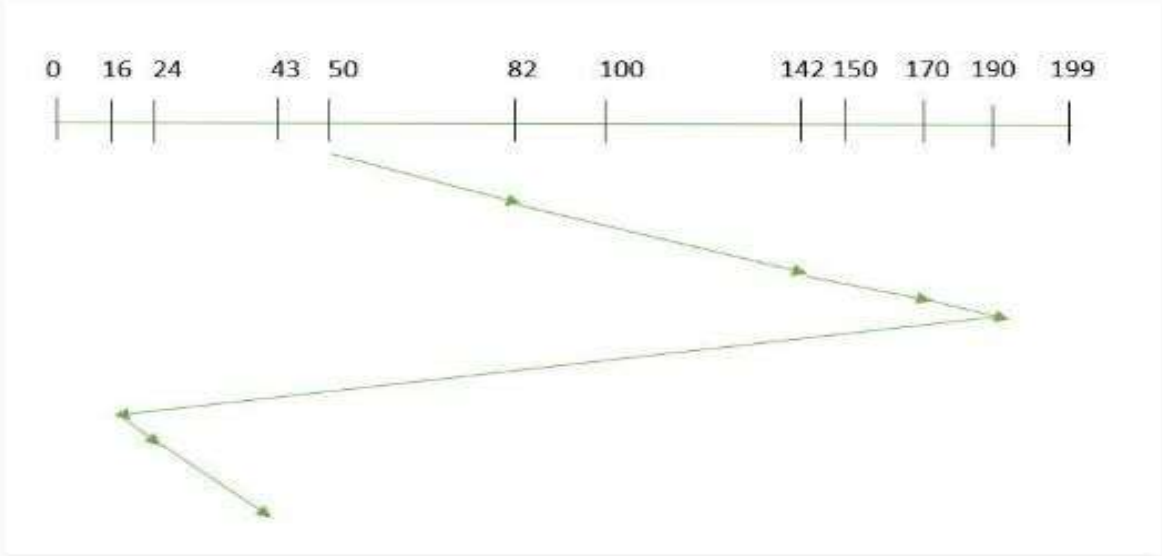
1.

So, the seek time is calculated as:

1. =(190-50)+(190-16)
   =314

CLOOK Disk Scheduling Algorithm

C-LOOK is an enhanced version of both **SCAN** as well as **LOOK** disk scheduling algorithms. This algorithm also uses the idea of wrapping the tracks as a circular cylinder as C- SCAN algorithm but the seek time is better than C-SCAN algorithm. We know that C-SCAN is used to avoid starvation and services all the requests more uniformly, the same goes for C- LOOK. In this algorithm, the head services requests only in one direction(either left or right) until all the requests in this direction are not serviced and then jumps back to the farthest request on the other direction and service the remaining requests. This gives a better uniform servicing as well as avoids wasting seek time for going till the end of the disk.



1.

So, the seek time is calculated as:

1. =(190-50)+(190-16)+(43-16)
   =341

COMPARISON OF VARIOUS DISK SCHEDULING ALGORITHMS

FCFS Scheduling

It is a fair scheduling policy. It prevents starvation and gives low overhead. This is acceptable when the load on a disk is light. It has extremely low throughput due to lengthy seeks.

SSTF Scheduling

SSTF Scheduling has higher throughput and lower response time than FCFS Policy. It is reasonable solution for batch processing system. Sometimes, it does not ensure fairness because with this scheduling starvation is possible. This policy is generally not acceptable for interactive systems. It leads to higher variances of response times.

SCAN Scheduling

It offers an improved variance of response time. The drawback of this scheduling policy is that it does not change the direction until edge of disk is reached. Starvation is still possible in this scheduling. Under a light load, SCAN policy is best.

C-SCAN Scheduling

It maintains high level of throughput while further limiting variance of response times by avoiding discrimination against the innermost and outermost cylinders.

LOOK Scheduling

The main advantage of this scheduling is that it only performs sweeps large enough to service all requests. It improves efficiency by avoiding unnecessary seek operation. It gives high throughput.

C-LOOK Scheduling

It gives lower variance of response time than LOOK, at the expense of throughput.

From various runs, it is concluded that Shortest seek time first has the minimum average head movement average head movement than all other five algorithms for the similar requests. Thus SSTF algorithm is a good criterion for selecting the request for I/O from the disk queue.

END OF MODULE-4