UNIT-3
MEMORY MANAGEMENT AND SECONDARY STORAGE

- **MEMORY MANAGEMENT:**

  Swapping, Contiguous Allocation, Paging, Structure of the page table, Segmentation with Paging.

- **VIRTUAL MEMORY:**

  Demand Paging, Page Replacement Algorithms, Copy-on-Write, Thrashing.

- **SECONDARY STORAGE STRUCTURE:**

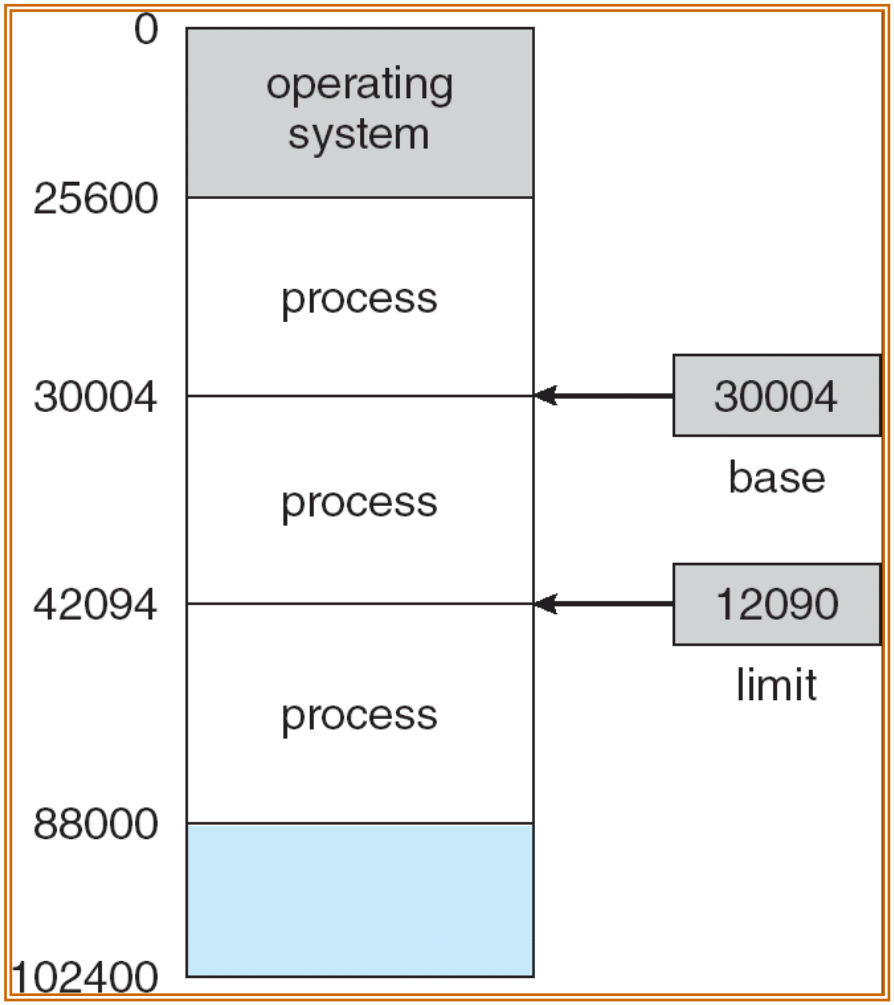
  Overview of Mass Storage Structure, Disk Structure, Disk Scheduling, Disk Management.

Basic H/W:

Main memory and the registers built into the processor itself are the only storage that the CPU can access directly.

The processor normally needs to stall, since it does not have the data required to complete the instruction that it is executing. This situation is intolerable because of the frequency of memory accesses.

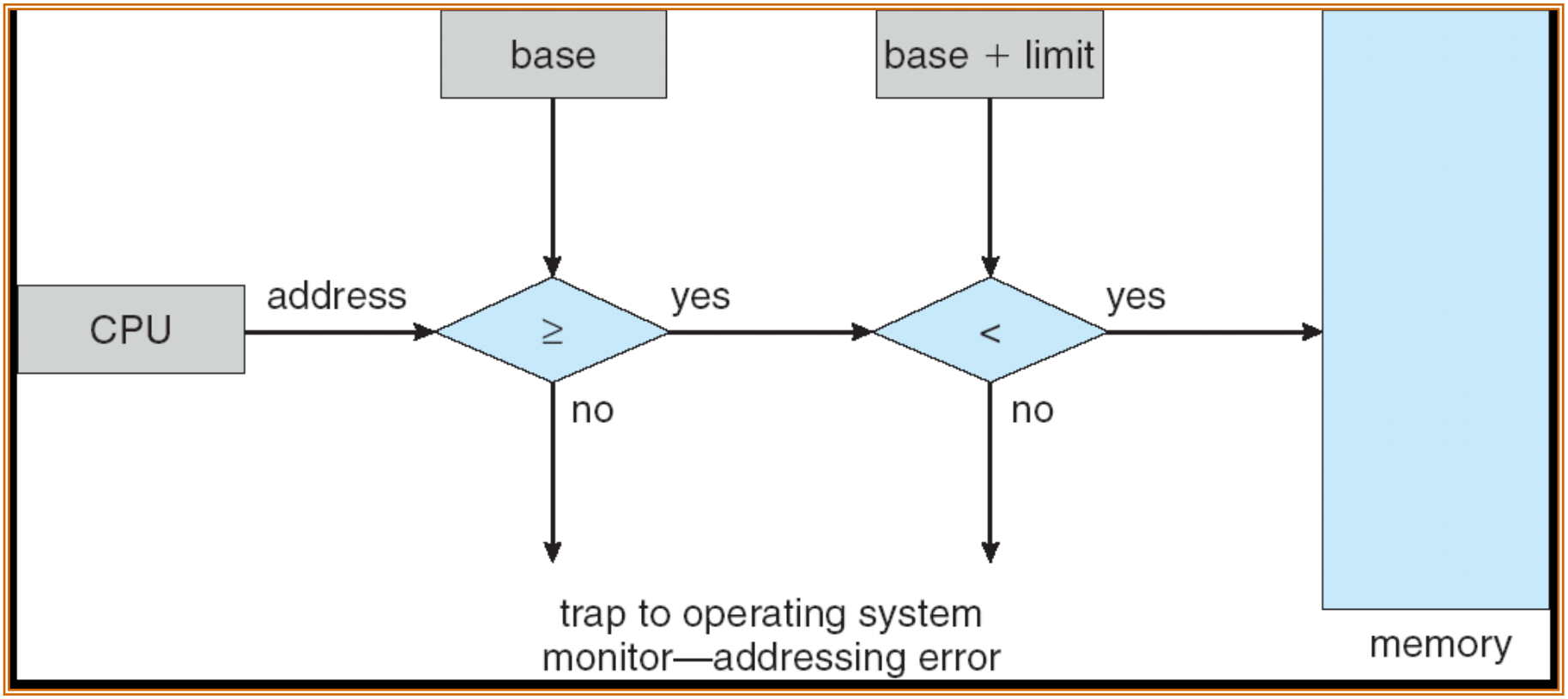
The remedy is to add fast memory between the CPU and main memory. A memory buffer used to accommodate a speed differential, called a Cache.



A Base and a Limit Register define a Logical Address Space

- The Base Register holds the smallest legal physical memory address.
- The Limit Register specifies the size of the range.
- For example, if the base register holds 300040 and the limit register is 120900, then the program can legally access all addresses from 300040 through 420939 (inclusive).
- The base and limit registers can be loaded only by the operating system- executed only in kernel mode.

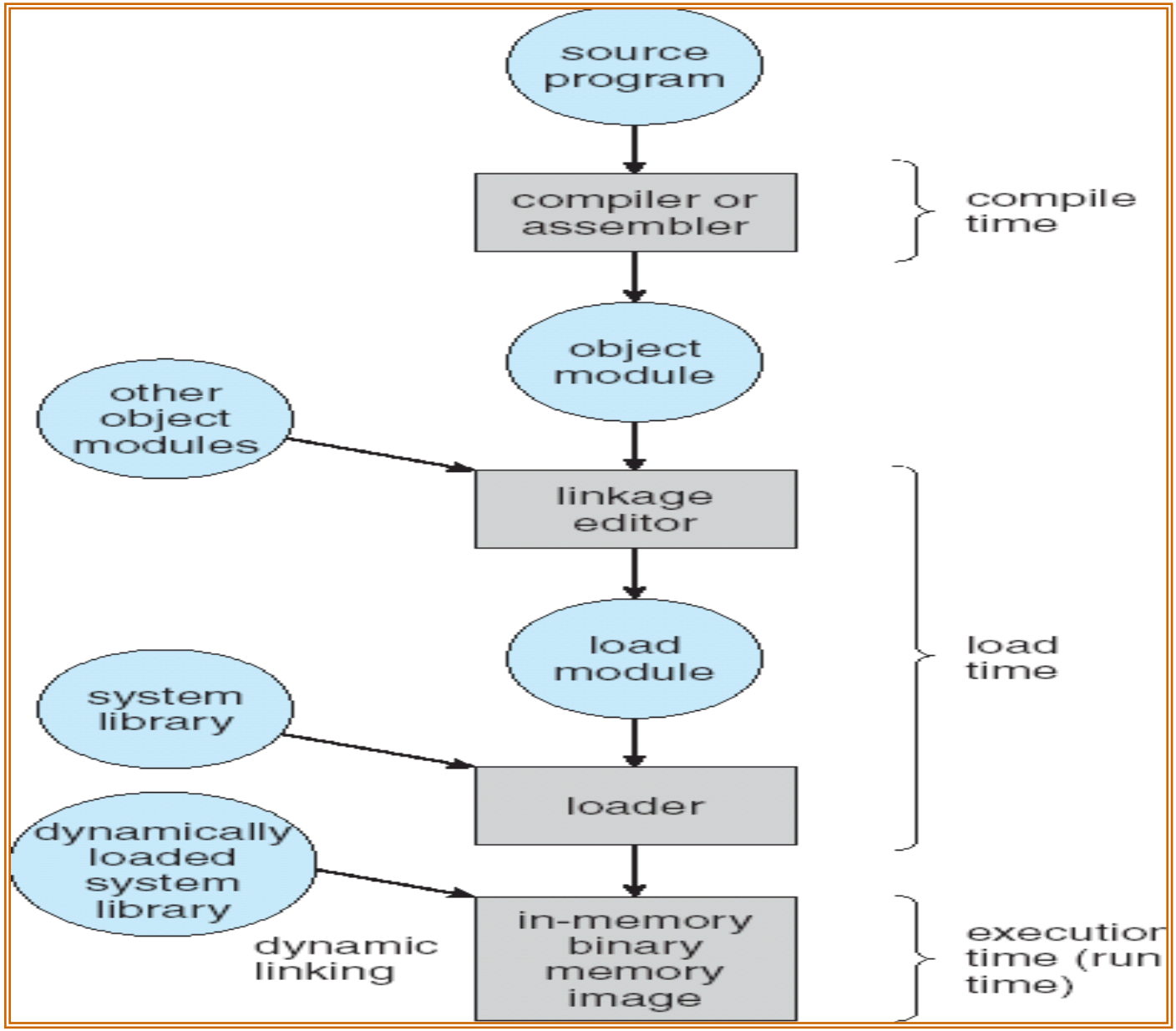
Hardware address protection with Base and Limit Registers



Address Binding

- A program resides on a disk as a binary executable file.
- The processes on the disk that are waiting to be brought into memory for execution form the Input Queue.
- The normal procedure is to select one of the processes in the input queue and to load that process into memory.
- **Compile time-** where the process will reside in memory, then absolute code can be generated.
- **Load time.** If it is not known at compile time where the process will reside in memory, then the compiler must generate Relocatable Code.
- **Execution time-** If the process can be moved during its execution from one memory segment to another, then binding must be delayed until run time.

Multistep processing of a user program.



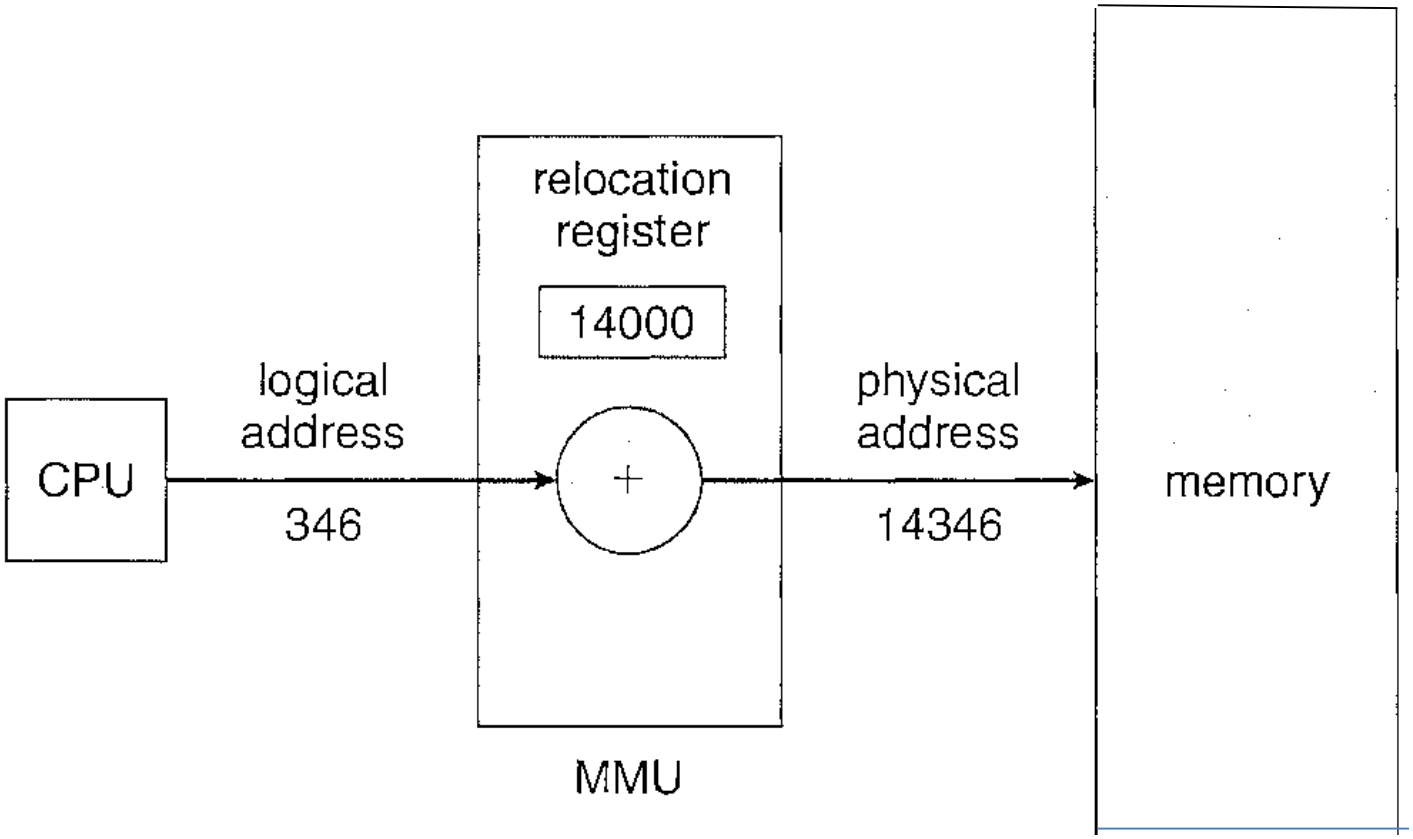
Logical vs. Physical Address Space

- 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 **Memory Address Register** the of the memory-is commonly referred to as a **Physical Address.**
- The compile-time and load-time address-binding methods generate identical logical and physical addresses.
- The set of all logical addresses generated by a program is a logical address space.
- The set of all physical addresses corresponding to these logical addresses is a physical address space.
- The run-time mapping from virtual to physical addresses is done by a hardware device called the Memory Management Unit.

Logical vs. Physical Address Space

- The concept of a logical address space that is bound to a separate physical address space is central to proper memory management.
  - Logical address – generated by the CPU; also referred to as Virtual Address.
  - Physical address – address seen by the memory unit.
- Logical and physical addresses are the same in compile-time and load-time address-binding schemes;
- logical (virtual) and physical addresses differ in execution-time address-binding scheme.

Dynamic relocation using a relocation register.



The user program never sees the real physical addresses.

The program can create a pointer to location 346, store it in memory, manipulate it, and compare it with other addresses-all as the number 346.

The user program deals with logical addresses.

- The memory-mapping hardware converts logical addresses into physical addresses.
- Two different types of addresses: logical addresses (in the range 0 to max) and physical addresses (in the range R+ 0 to R+ max for a base value R).
- The user generates only logical addresses and thinks that the process runs in locations 0 to *max.*
- However, these logical addresses must be mapped to physical addresses before they are used.
- To obtain better memory-space utilization- use **dynamic Loading.**
- With dynamic loading, a routine is not loaded until it is called.
- All routines are kept on disk in a relocatable load format.
- The main program is loaded into memory and is executed.

- The advantage of dynamic loading is that an unused routine is never loaded. This method is particularly useful when large amounts of code are needed to handle infrequently occurring cases, such as error routines. Dynamic loading does not require special support from the operating system.

Memory-Management Unit (MMU)

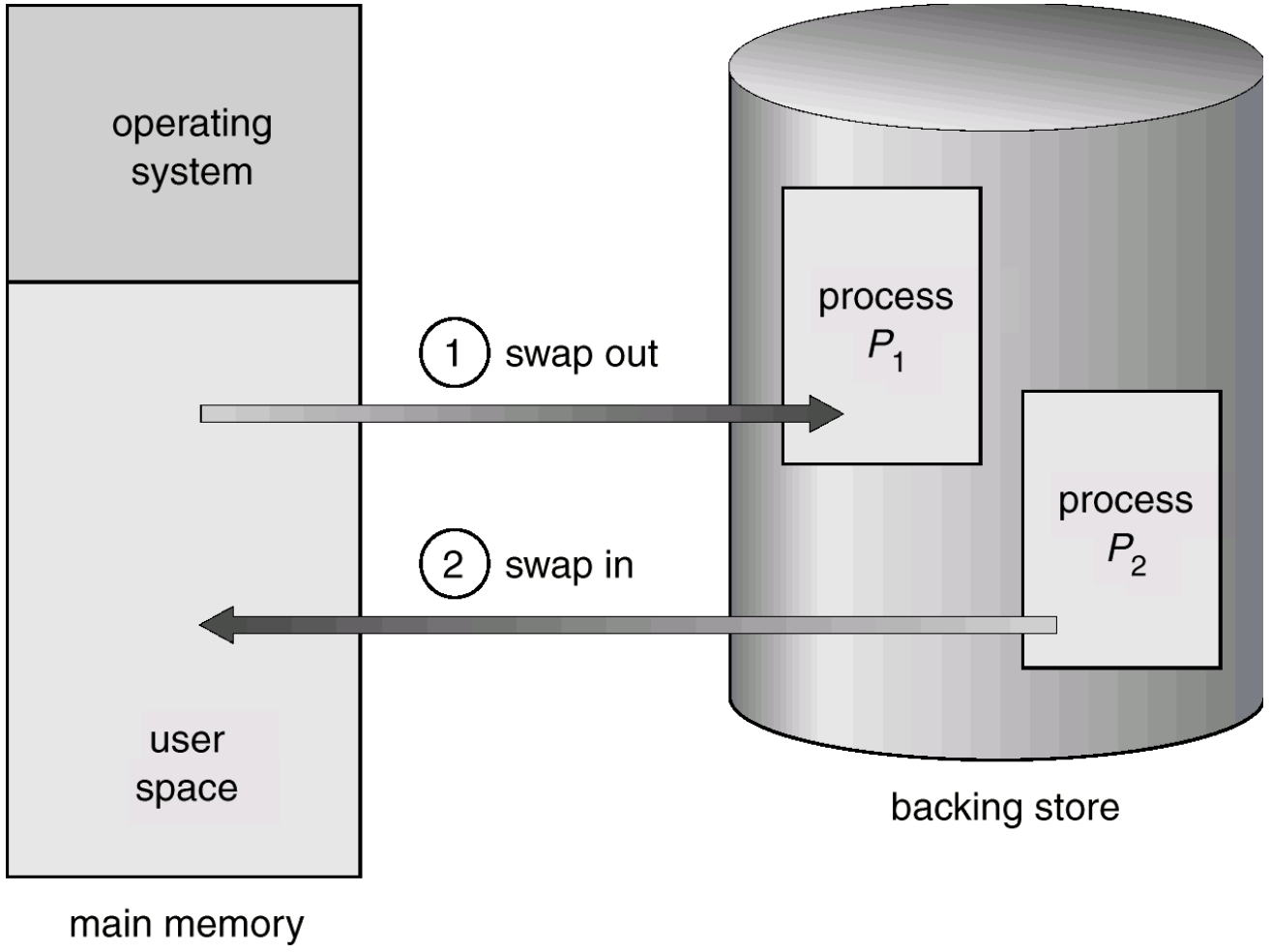
- Hardware device that maps virtual to physical address.
- In MMU scheme, the value in the relocation register is added to every address generated by a user process at the time it is sent to memory.
- The user program deals with *logical* addresses; it never sees the *real* physical addresses.

1. Swapping

- A process can be swapped temporarily out of memory to a backing store, and then brought back into memory for continued execution.

  Example: Round Robbin CPU Scheduling Algorithm
- Backing store – is a fast disk large enough to accommodate copies of all memory images for all users; must provide direct access to these memory images.
- Roll out, Roll in – swapping variant used for priority-based scheduling algorithms; lower-priority process is swapped out so higher-priority process can be loaded and executed.
- Major part of swap time is transfer time; total transfer time is directly proportional to the amount of memory swapped.
- Modified versions of swapping are found on many systems, i.e., UNIX and Microsoft Windows.

Schematic View of Swapping



- To get an idea of the context-switch time, let us assume that the user process is 100 MB in size and the backing store is a standard hard disk with a transfer rate of 50MB per second.
- The actual transfer of the 100-MB process to or from main memory takes: 100MB/50MB per second= 2 seconds.
- Assuming an average latency of 8 milliseconds, the swap time is 2008 milliseconds. Since we must both swap out and swap in, the total swap time is about 4016 milliseconds.
- Notice that the major part of the swap time is transfer time.
- The total transfer time is directly proportional to the amount of memory swapped. If we have a computer system with 4 GB of main memory and a resident operating system taking 1 GB, the maximum size of the user process is 3GB. However, many user processes may be much smaller than this-say, 100 MB.
- A 100 MB process could be swapped out in 2 seconds, compared with the 60 seconds required for swapping 3 GB.

2. Contiguous Allocation

1. Memory Mapping and Protection
2. Memory Allocation
3. Fragmentation

Main memory usually into two partitions:

1. Resident operating system, usually held in low memory with interrupt vector.
2. User processes then held in high memory.

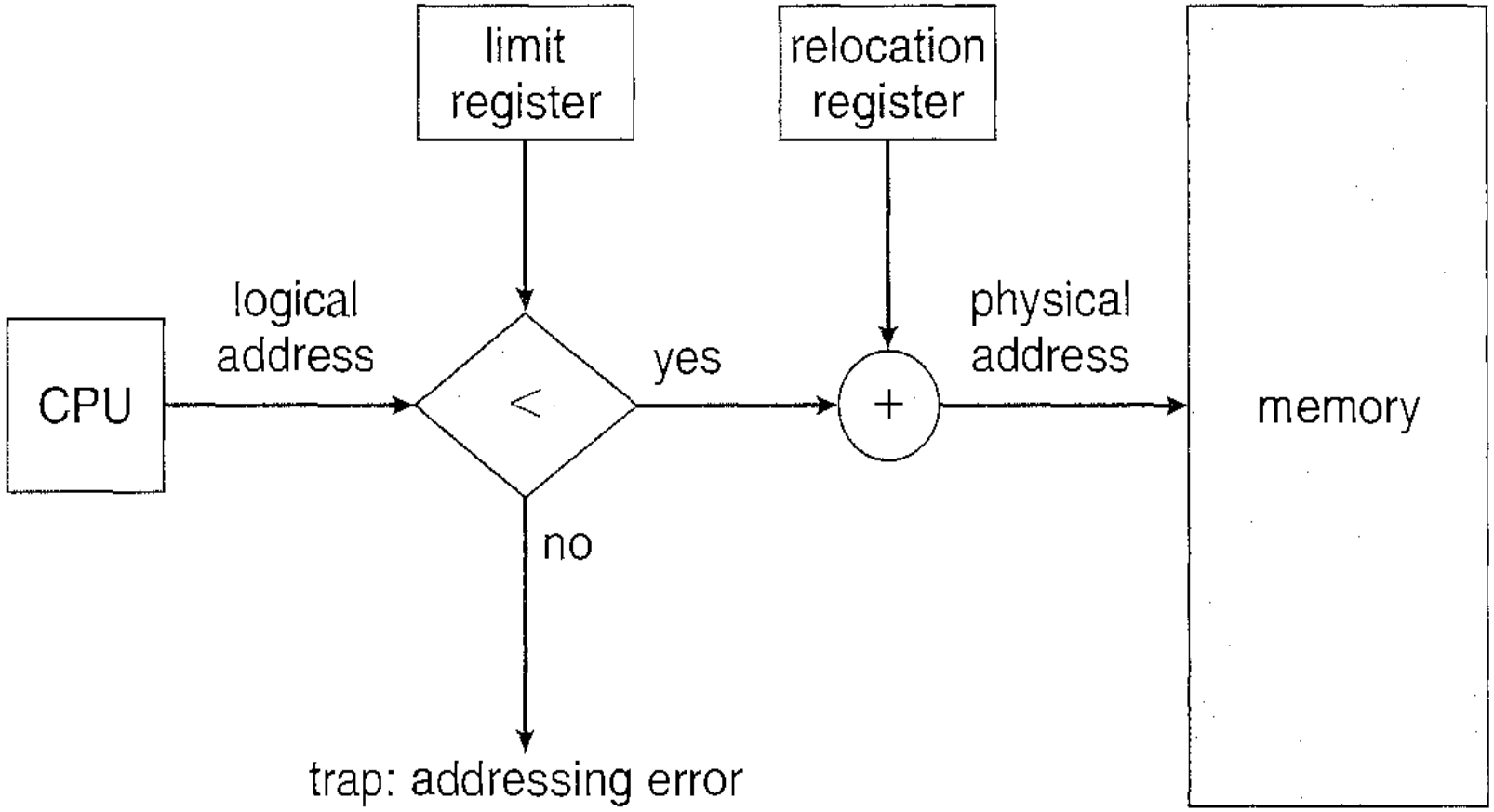
- In contiguous memory allocation, each process is contained in a single contiguous section of memory.
- The relocation register contains the value of the smallest physical address; the limit register contains the range of logical addresses (for example, relocation= 100040 and limit= 74600).
- With relocation and limit registers, each logical address must be less than the limit register; the MMU maps the logical address dynamically by adding the value in the relocation register. This mapped address is sent to memory.

1. Memory Mapping and Protection:

Single-partition allocation

- Relocation-register scheme used to protect user processes from each other, and from changing operating-system code and data.
  - Relocation register contains value of smallest physical address.
  - Limit Register contains range of logical addresses – each logical address must be less than the limit register.
- The MMU maps the logical address dynamically by adding the value in the relocation register. This mapped address is sent to memory.

Hardware support for Relocation and Limit Registers.



- When the CPU scheduler selects a process for execution, the dispatcher loads the relocation and limit registers with the correct values as part of the context switch.
- Because every address generated by a CPU is checked against these registers, we can protect both the operating system and the other users' programs and data from being modified by this running process.
- The relocation-register scheme provides an effective way to allow the operating system's size to change dynamically.

Example, the operating system contains code and buffer space for device drivers.

- If a device driver (or other operating-system service)
- is not commonly used then we don't want *to keep the code and data in memory.* Such code is called transient operating-system code; it comes and goes as needed.

Contiguous Allocation: 2. Memory Allocation

- One of the simplest methods for allocating memory is to **divide memory into several fixed-sized partitions**. Each partition may contain exactly one process.

Multiple-partition allocation:

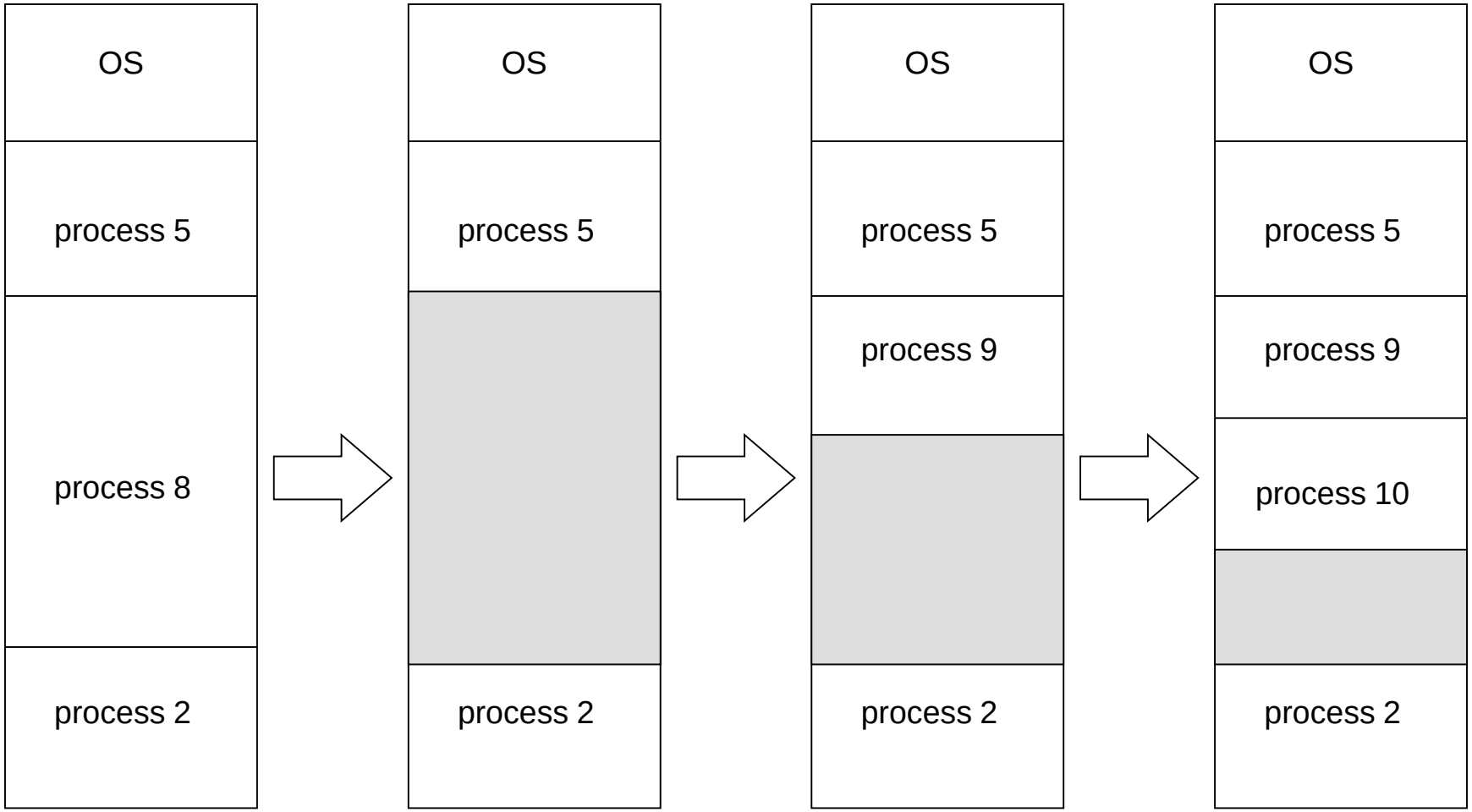
- **Fixed Partition Scheme:** when a partition is free, a process is selected from the input queue and is loaded into the free partition.

Hole – block of available memory; holes of various size are scattered throughout memory.

When a process arrives, it is allocated memory from a hole large enough to accommodate it.

Operating system maintains information about:

a) allocated partitions b) free partitions (hole)


OS
process 5
process 8
process 2
OS
process 5
process 2
OS
process 5
process 9
process 2
OS
process 5
process 9
process 10
process 2

Dynamic Storage-Allocation Problem

How to satisfy a request of size *n* from a list of free holes.
Solutions are

- **First-fit**: Allocate the first hole that is big enough. Searching can start either at the beginning of the set of holes or at the location where the previous first-fit search ended. We can stop searching as soon as we find a free hole that is large enough.
- **Best-fit**: Allocate the smallest hole that is big enough. We must search the entire list, unless the list is ordered by size. This strategy produces the smallest leftover hole.
- **Worst-fit**: Allocate the largest hole; must also search entire list. Produces the largest leftover hole.
- Both the first-fit and best-fit strategies for memory allocation suffer from **external fragmentation. (dynamic allocation)**

First-fit and best-fit better than worst-fit in terms of speed and storage utilization.

Memory Fragmentation

- External fragmentation – total memory space exists to satisfy a request, but it is not contiguous. Dynamic Partitioning.
- Internal fragmentation – allocated memory may be slightly larger than requested memory; this size difference is memory internal to a partition, but not being used. Fixed Partitioning.
- Reduce external fragmentation by compaction technique.
  - Shuffle memory contents to place all free memory together in one large block.
- If relocation is static and is done at assembly or load time, compaction cannot be done;
- compaction is possible only if relocation is dynamic and is done at execution.
- Another possible solution to the external-fragmentation problem is to permit the logical address space of the processes to be noncontiguous, thus allowing a process to be allocated physical memory wherever the latter is available.

Contiguous Memory Allocation

Fixed sized partition

- In the fixed sized partition the system divides memory into fixed size partition (may or may not be of the same size) here entire partition is allowed to a process and if there is some wastage inside the partition is allocated to a process and if there is some wastage inside the partition then it is called internal fragmentation.

Advantage: Management or book keeping is easy.

Disadvantage: Internal fragmentation

Variable size partition

- In the variable size partition, the memory is treated as one unit and space allocated to a process is exactly the same as required and the leftover space can be reused again.

Advantage: There is no internal fragmentation.

Disadvantage: Management is very difficult as memory is becoming purely fragmented after some time.

Non-contiguous memory allocation

1. **Paging:** A non-contiguous policy with a fixed size partition is called paging.

Advantages: It is independent of external fragmentation.

Disadvantages:

- It makes the translation very slow as main memory access two times.
- A page table is a burden over the system which occupies considerable space.

2. Segmentation

3. Segmentation with paging

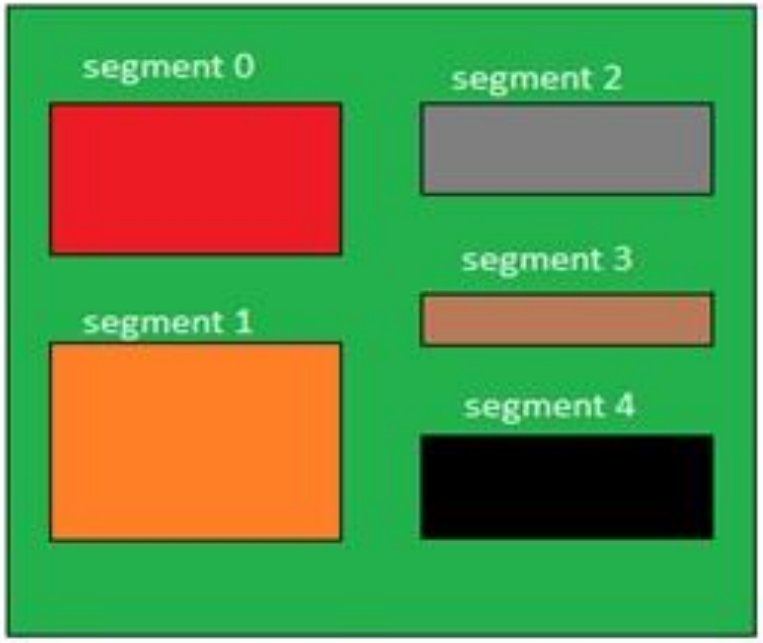
2. **Segmentation** is a programmer view of the memory where instead of dividing a process into equal size partition we divided according to program into partition called segments.

- The translation is the same as paging but paging segmentation is independent of internal fragmentation but suffers from external fragmentation. Reason of external fragmentation is program can be divided into segments but segment must be contiguous in nature.

3. Segmentation with paging

- In segmentation with paging, we take advantages of both segmentation as well as paging. It is a kind of multilevel paging but in multilevel paging, we divide a page table into equal size partition but here in segmentation with paging, we divide it according to segments. All the properties are the same as that of paging because segments are divided into pages.

Logical View of Segmentation

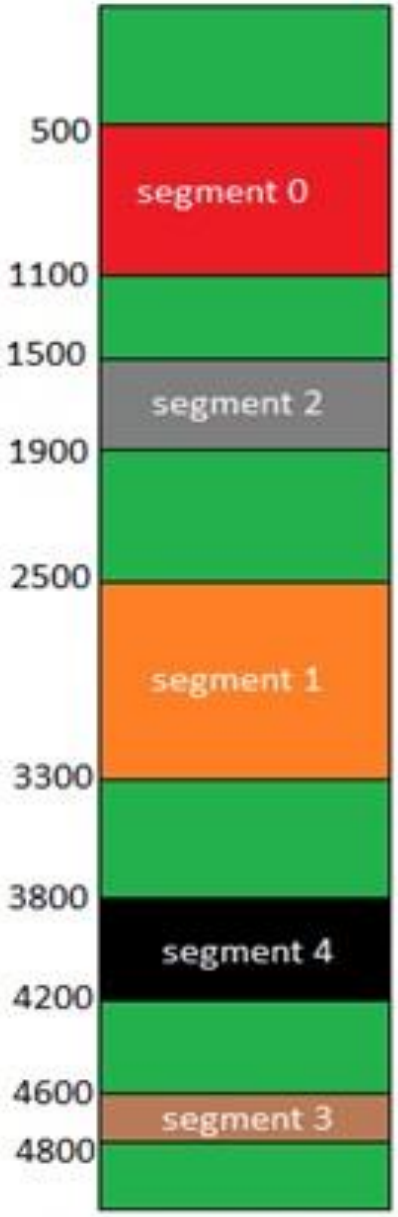


Logical Address Space

Segment Number

	base address	Limit
0	500	600
1	2500	800
2	1500	400
3	4600	200
4	3800	400

Segment Table



Physical Address Space

3. Paging

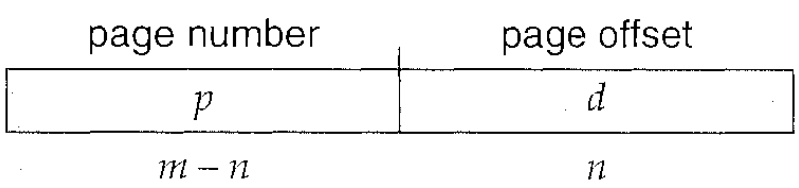
- Paging is a **memory-management scheme** that permits the physical address space of a process to be noncontiguous.
- Paging avoids external fragmentation and the need for compaction.
- Paging solves the considerable problem of fitting memory chunks of varying sizes onto the backing store.
- The backing store has the same fragmentation problems discussed in connection with main memory, but **access is much slower,** so compaction is impossible. Because of its advantages over earlier methods, paging in its various forms is used in most operating systems.
- **Paging** implementation closely by **integrating the hardware and operating system,** especially on 64-bit microprocessors.

- Logical address space of a process can be noncontiguous; process is allocated physical memory whenever the latter (last/final) is available.

The **basic method** for implementing paging involves :

- Divide **physical memory** into fixed-sized blocks called **frames** (size is power of 2, between 512 bytes and 8192 bytes).
- Divide **logical memory** into blocks of same size called **pages**.
- Keep track of all free frames.
- To run a program of size n pages, need to find *n* free frames and load program.
- Set up a page table to translate logical to physical addresses.
- Internal fragmentation.

Address Translation Architecture: Paging hardware.



Page number (p)

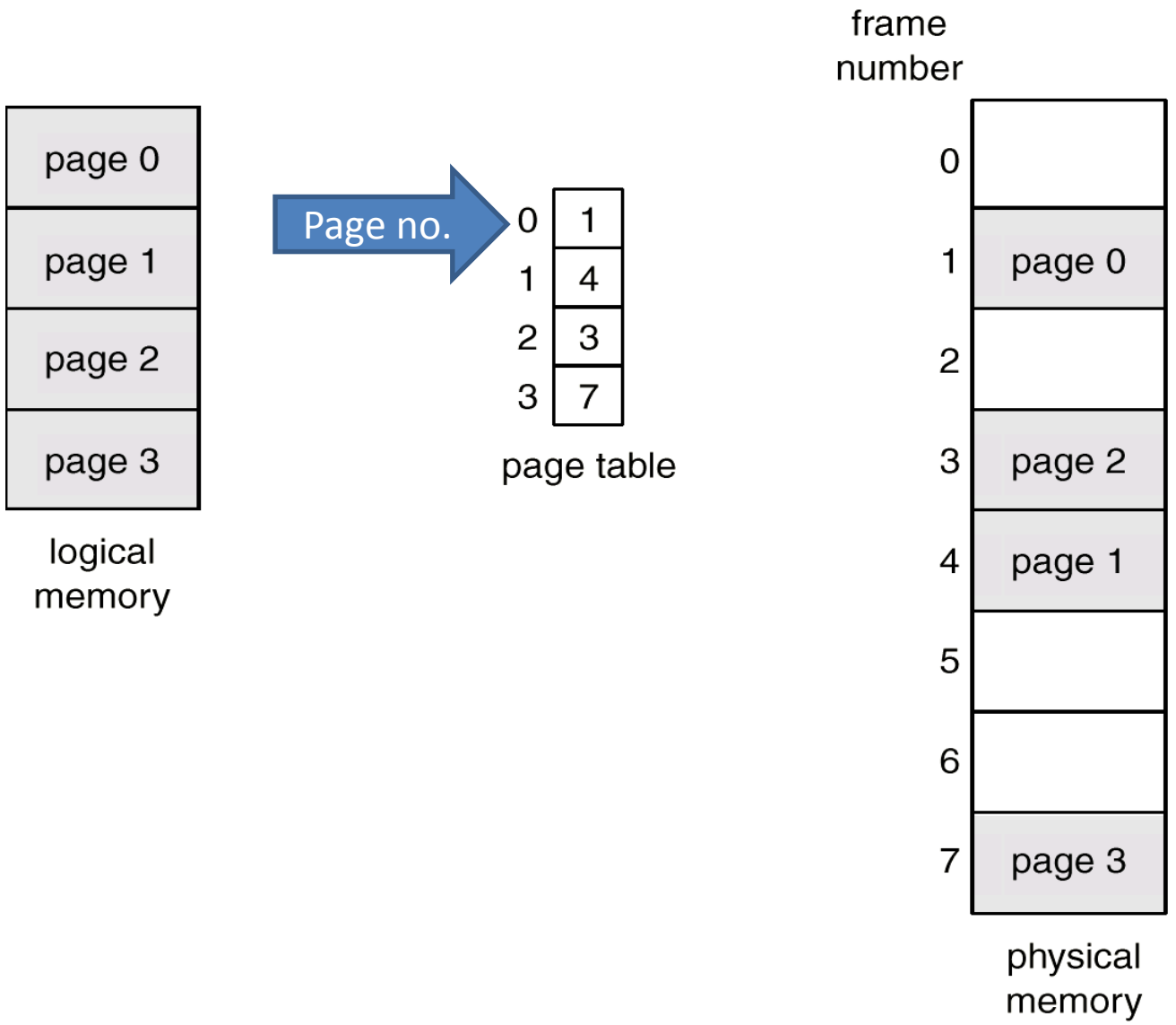
Page offset (d)

Address Translation Scheme

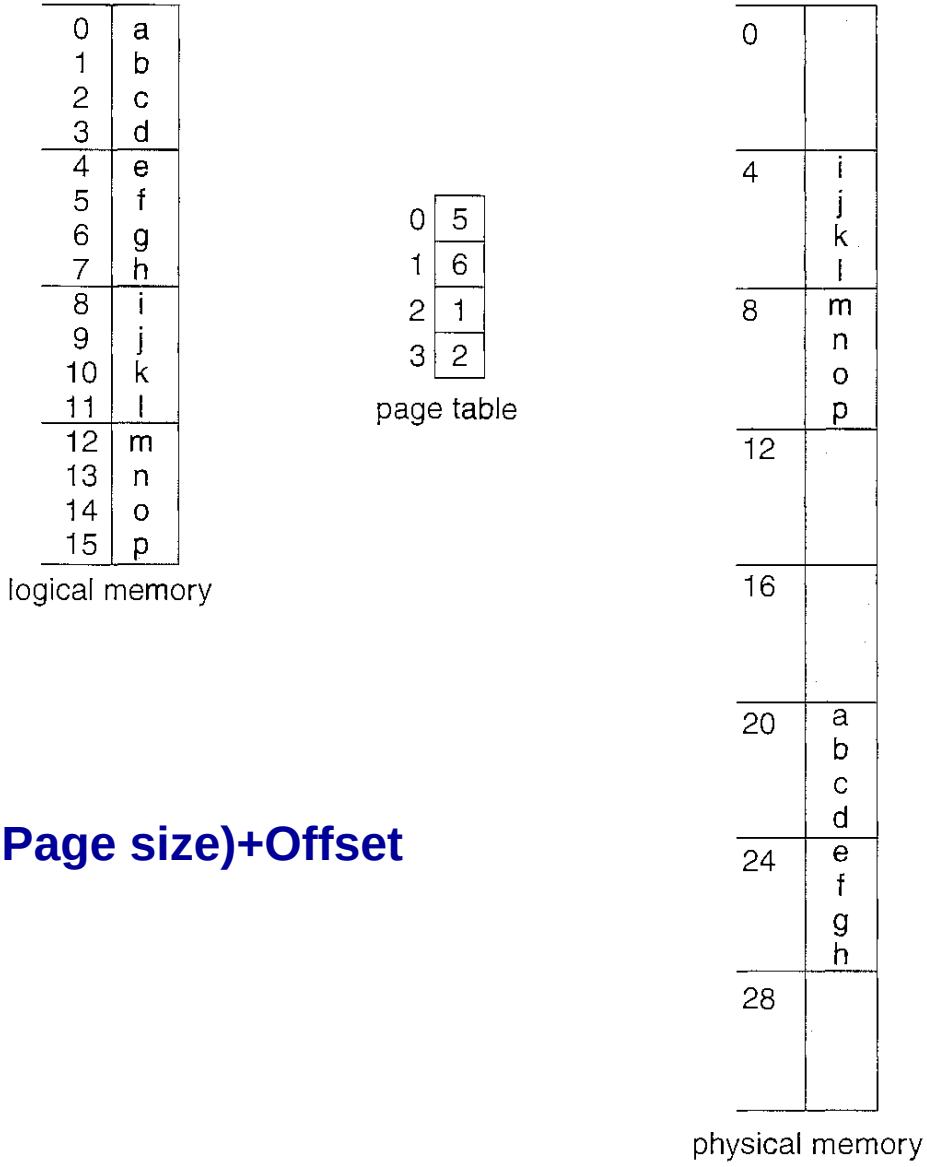
Every address generated by the CPU is divided into two parts: a **page number (p)** and **a page offset (d).**

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

Paging Example: Paging model of logical and physical memory

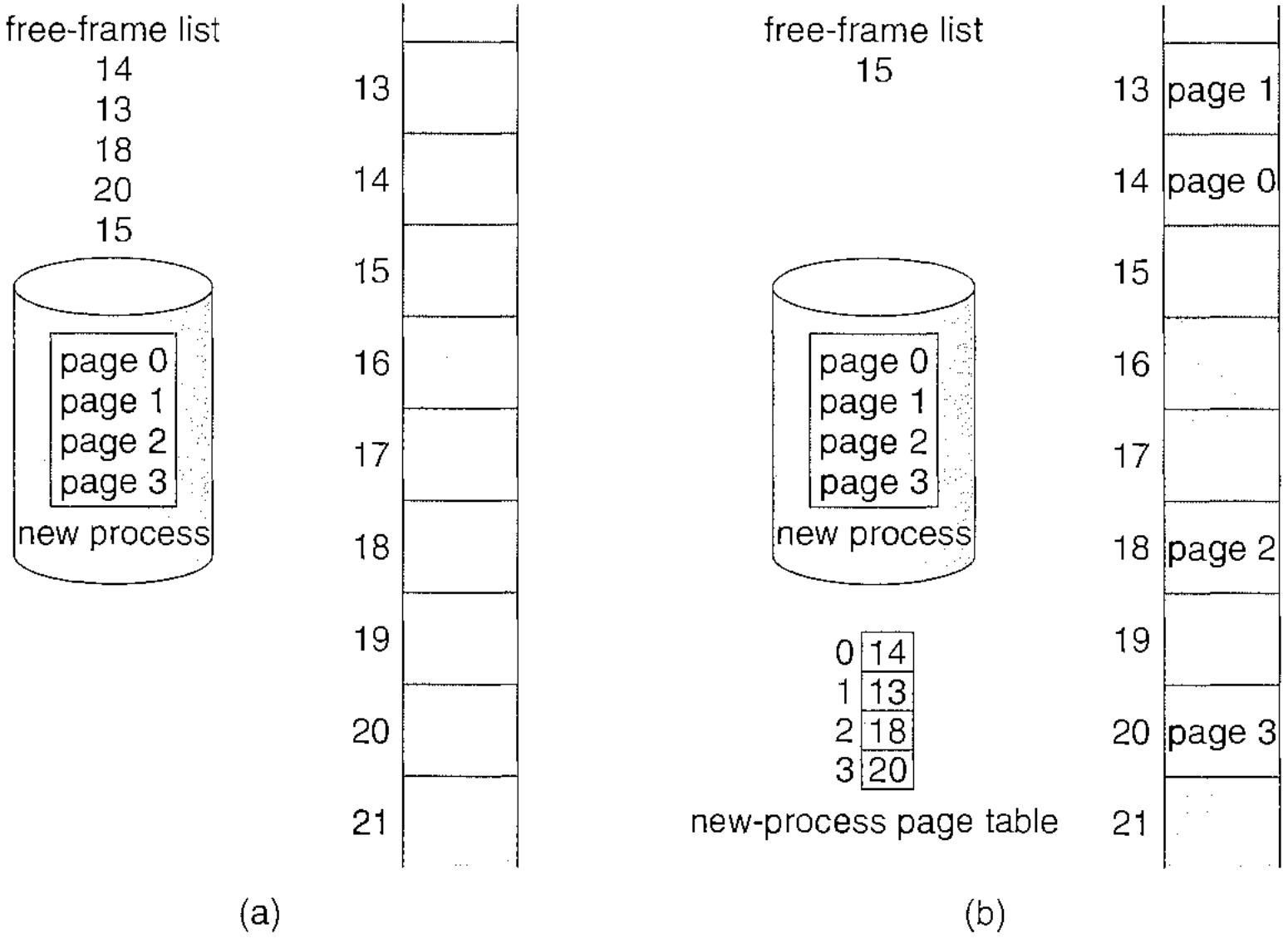


Paging example for a 32-byte memory with 4-byte pages.



Physical address= (frame no. X Page size)+Offset

Free frames (a) before allocation and (b) after allocation



The size of a page is typically a power of 2, varying between 512 bytes and 16 MB per page, depending on the computer architecture.

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

Using a page size of 4 bytes and a physical memory of 32 bytes (8 pages)

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 x 4) + 0).

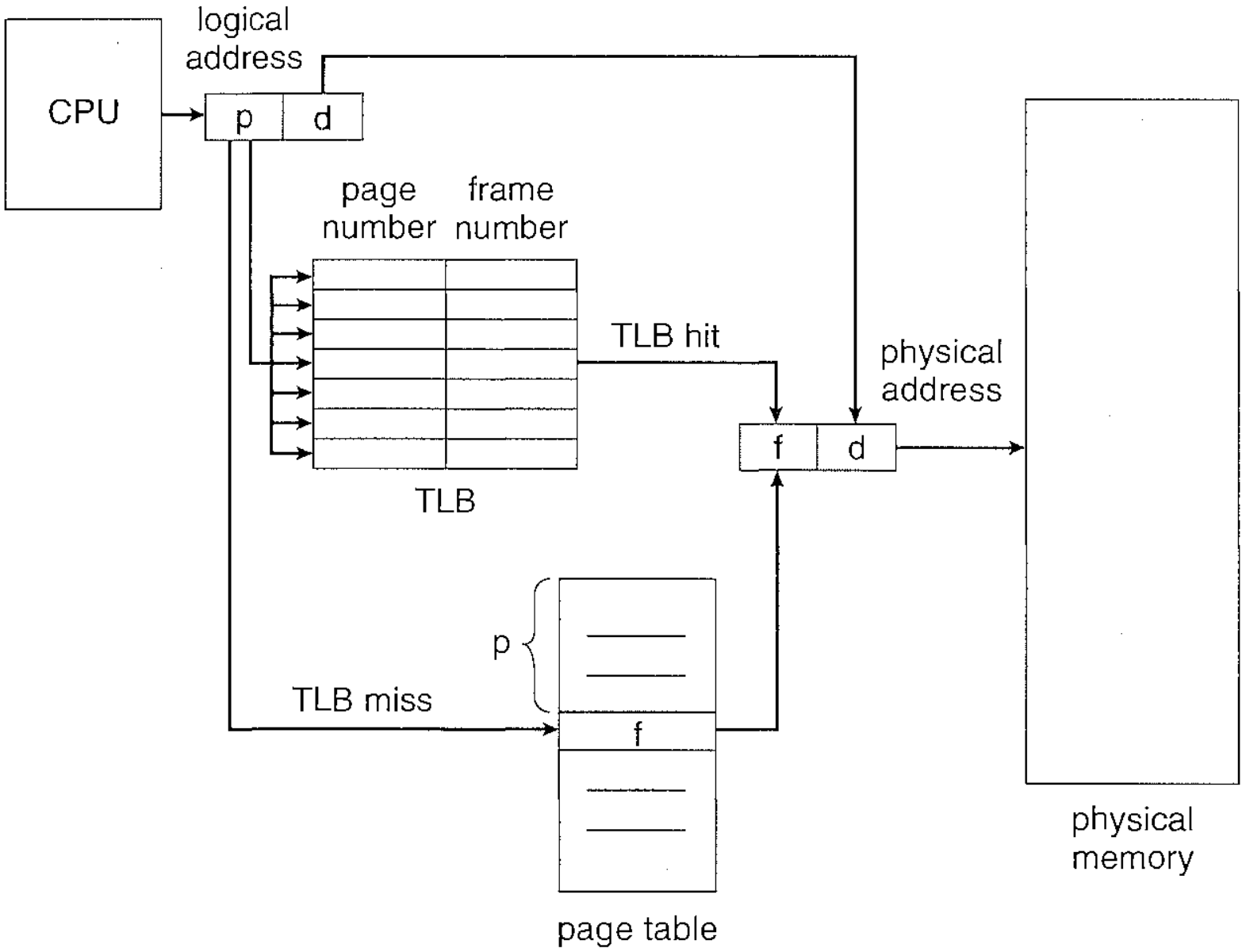
- Logical address 3 (page 0, offset 3) maps to physical address 23(= *(5x4) + 3).*

Logical address 4 (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 (= (6x4) + 0).

Paging itself is a form of dynamic relocation.

- In Paging Scheme- **No External Fragmentation**, but some internal fragmentation.
- The memory access is slowed by a factor of 2. This delay would be intolerable under most circumstances.
- The standard solution to this problem is to use a special, small, fast lookup hardware cache, called a **Translation Look-aside Buffer (TLB).**
- The TLB is associative, high-speed memory. Each entry in the TLB consists of two parts: a key (or tag) and a value.
- In addition to them the TLB has the page number and frame number to the TLB, so that they will be found quickly on the next reference.
- The percentage of times that a particular page number is found in the TLB is called the **hit ratio.**
- If the page number is not in the TLB known as a **TLB miss.**
- If the **TLB** is already **full** of entries, the **O S** must select one for replacement. **Replacement policies** range from **Least Recently Used (LRU) to random**.

Paging hardware with TLB.



- The percentage of times that a particular page number is found in the TLB is called the Hit Ratio.
- An 80-percent hit ratio, for example, means that we **find the desired page number in the TLB** 80 percent of the time. If it takes 20 nanoseconds to search the TLB and 100 nanoseconds to access memory, then a mapped-memory access takes 120 nanoseconds when the page number is in the TLB.
- **If we fail to find the page number** in the TLB (20 nanoseconds), then we must first access memory for the page table and frame number (100 nanoseconds) and then access the desired byte in memory (100 nanoseconds), for a total of 220 nanoseconds.
- To find the effective we weight the case by its probability:

Effective Access Time = 0.80 x 120 + 0.20 x 220

= 140 nanoseconds.

- In this example, we suffer a 40-percent slowdown in memory-access time (from 100 to 140 nanoseconds).
- **For a 98-percent hit ratio, we have**

Effective Access Time = 0.98 x 120 + 0.02 x 220

= 122 nanoseconds.

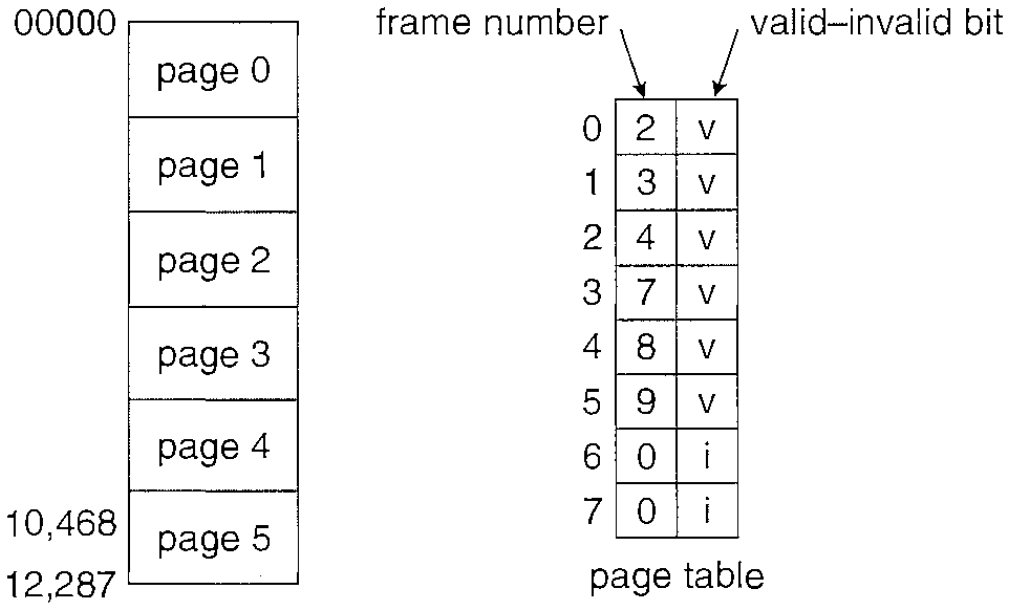
- This increased hit rate produces only a 22 percent slowdown in access time.

Protection

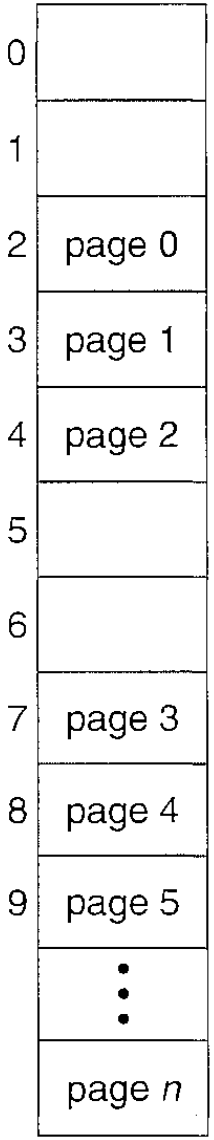
- Memory protection in a paged environment is accomplished by protection bits associated with each frame.
- Normally, these bits are kept in the page table.
- One bit can define a page to be read-write or read-only. Every reference to memory goes through the page table to find the correct frame number.
- We can create hardware to provide read-only, read-write, or execute-only protection; or, by providing separate protection bits for each kind of access, we can allow any combination of these accesses.

- One additional bit is generally attached to each entry in the page table: a valid- invalid bit.
- When this bit is set to "valid," the associated page is in the process's logical address space and is thus a legal (or valid) page.
- When the bit is set to "invalid,'" the page is not in the process's logical address space.
- This scheme has created a problem. Because the program extends to only address 10468, any reference beyond that address is illegal.
- Rarely does a process use all its address range. So It would be wasteful in these cases to create a page table with entries for every page in the address range.
- Some systems provide hardware, in the form of a Page-table Length Register (PTLR), to indicate the size of the page table.

- A system with a 14 bit address space(0-16383)
- Program that should use only 0-10468
- Page size of 2 KB



Valid (v) or invalid (i) bit in a page table



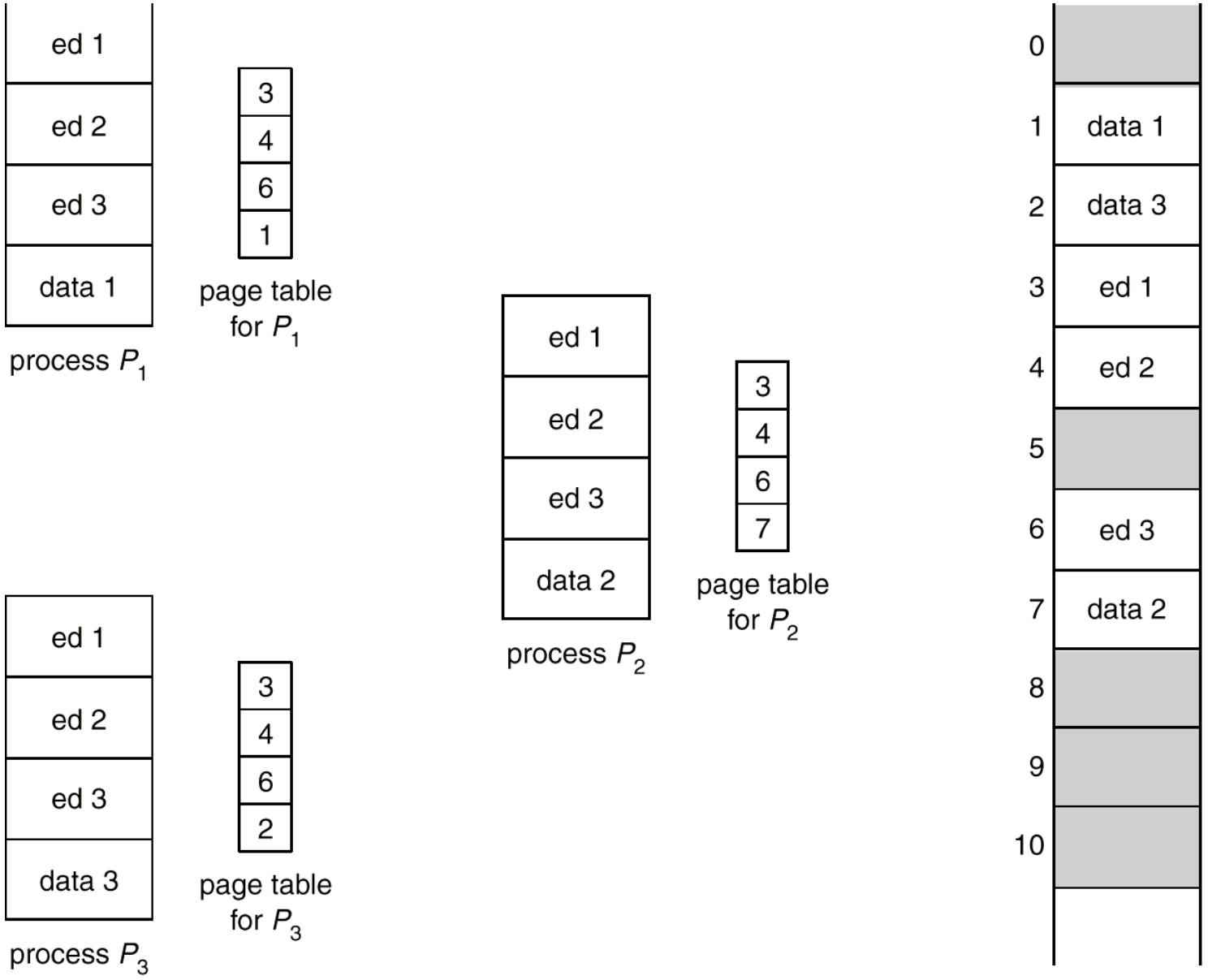
Shared Pages

- An advantage of paging is the possibility of sharing common code. This consideration is particularly important in a time-sharing environment.
- **Reentrant code** or **pure code** is non-self-modifying code; it never changes during execution. Thus, two or more processes can execute the same code at the same time.
- Each process has its own copy of registers and data storage to hold the data for the process's execution.

Example:

- If the text editor consists of 150 KB of code and 50 KB of data space, we need 8,000 KB to support the 40 users.
- Only one copy of the editor need be kept in physical memory. Each user‘s page table maps onto the same physical copy of the editor, but data pages are mapped onto different frames. Thus, to support 40 users, we need only one copy of the editor (150 KB), plus 40 copies of the 50 KB of data space per user. The total space required is now 2,150 KB instead of 8,000 KB- a significant savings.

- Other **heavily used programs** can also be shared - compilers, window systems, run-time libraries, database systems, and so on.
- To be sharable, the code must be reentrant.
- The read-only nature of shared code should not be left to the correctness of the code; the operating system should enforce this property.
- **Interprocess Communication.**
- Some operating systems implement shared memory using shared pages.
- Organizing memory according to pages provides numerous benefits in addition to allowing several processes to share the same physical pages.



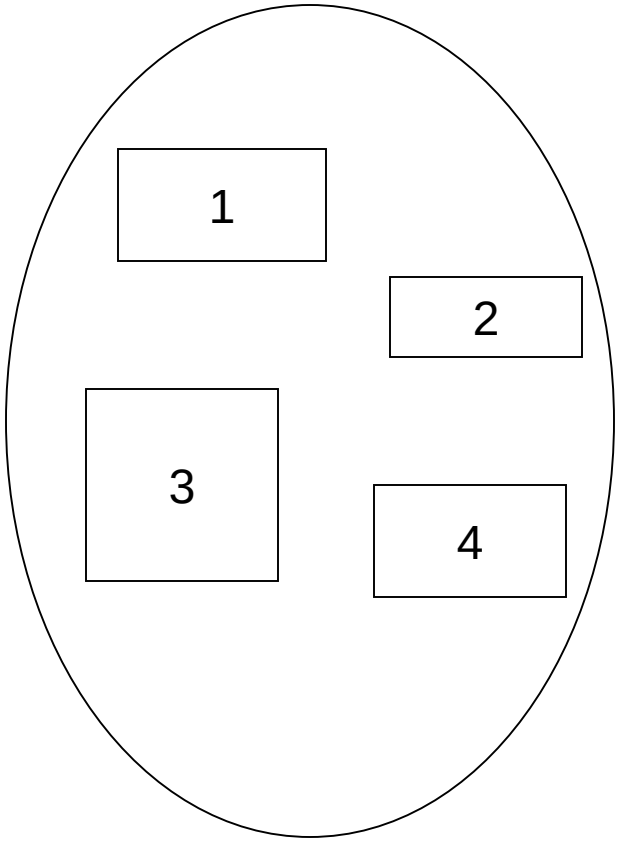
Sharing of Code in a Paging Environment

SEGMNETATION

- Segmentation- Non contiguous storage allocation.
- Paging- fixed Size, Physical Memory.
- Segmentation- Variable Size, Logical Memory.
- Memory-management scheme that supports user view of memory.
- A logical address space is a collection of segments.
- Each Logical address has a segment name and an offset.
- A segment is a logical unit such as:

  main program, procedure, function, local variables, global variables, common block, stack, symbol table, arrays

Logical View of Segmentation



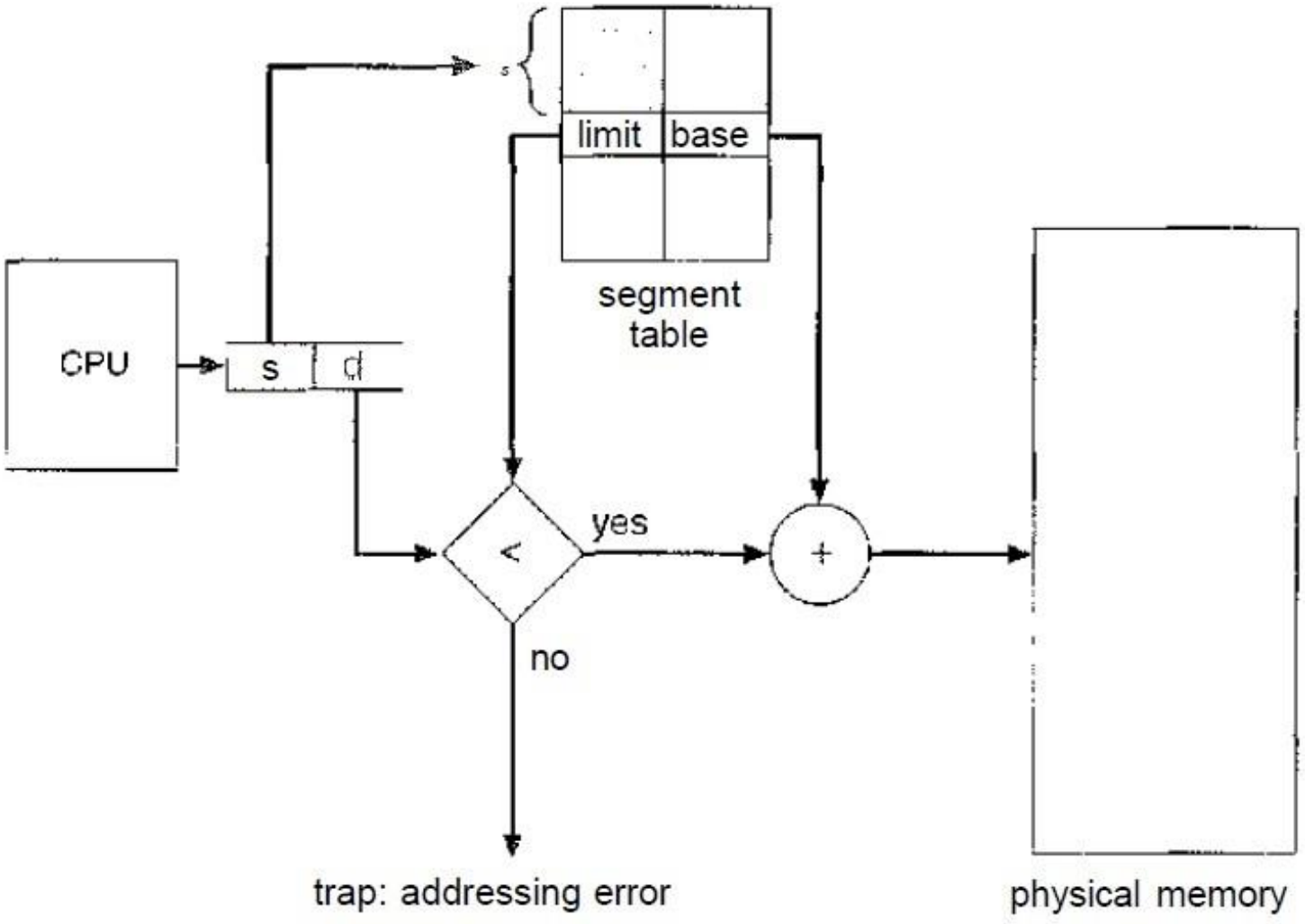
user space **physical memory space**

- A '**C**' compiler might create separate segments for the following:

1. The code
2. Global variables
3. The heap, from which memory is allocated
4. The stacks used, by each thread
5. The standard C library

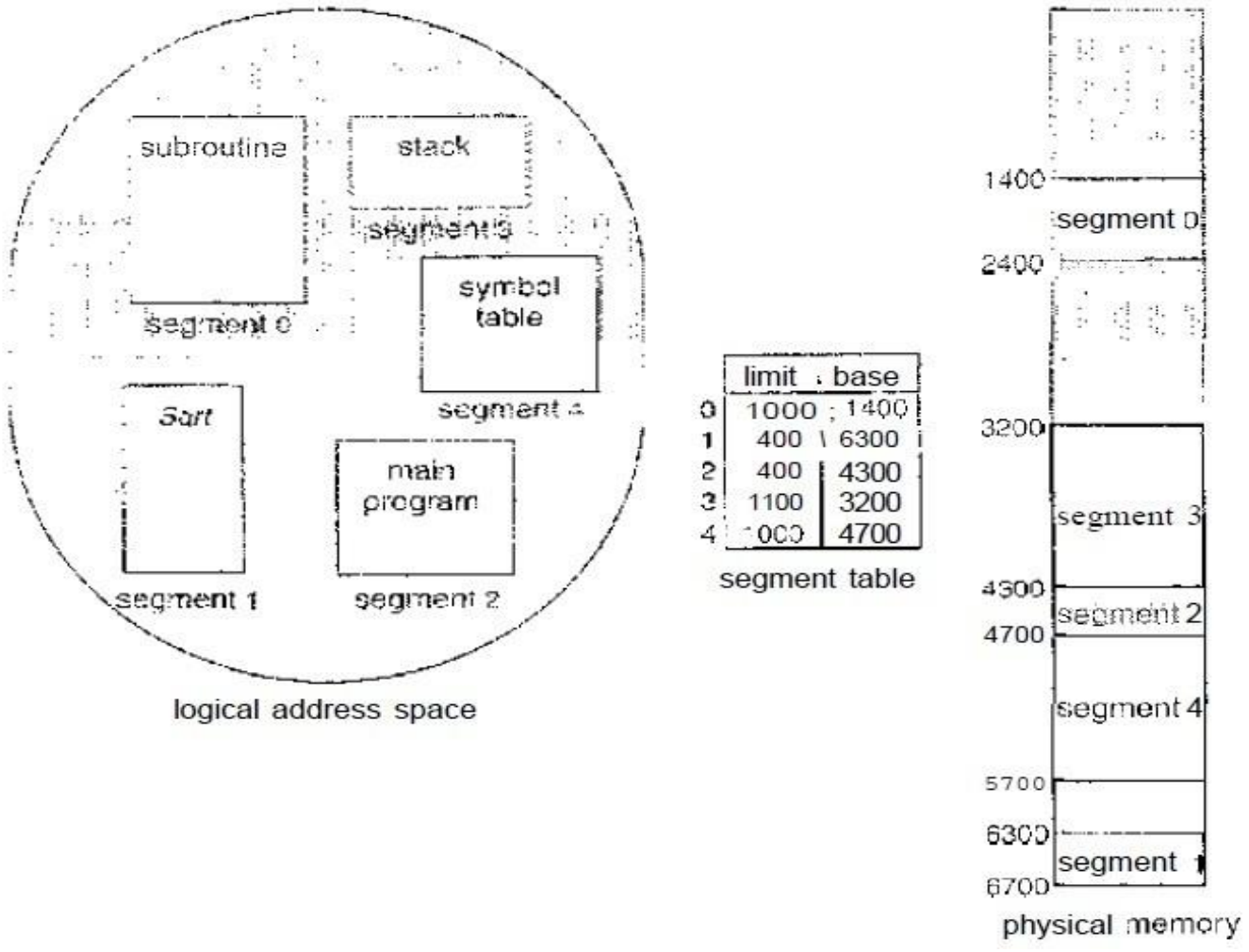
- The C Libraries that are linked in during compile time might be assigned separate segments.
- The loader would take all these segments and assign them segment numbers.

Segmentation hardware.



- Logical address consists of a two tuple:
  <segment-number, offset>
- Each entry in the segment table has a *segment base* and *a segment limit*.
- *The segment base contains the starting* physical address where the segment resides in memory.
- The segment limit specifies the length of the segment.
- The offset *d of* the logical address must be between 0 and the segment limit.
- If it is not, we trap to the operating system (logical addressing attempt beyond, end of segment).
- When an offset is legal, it is added to the segment base to produce the address in physical memory of the desired byte.
- The segment table is thus essentially an array of base-limit register pairs.

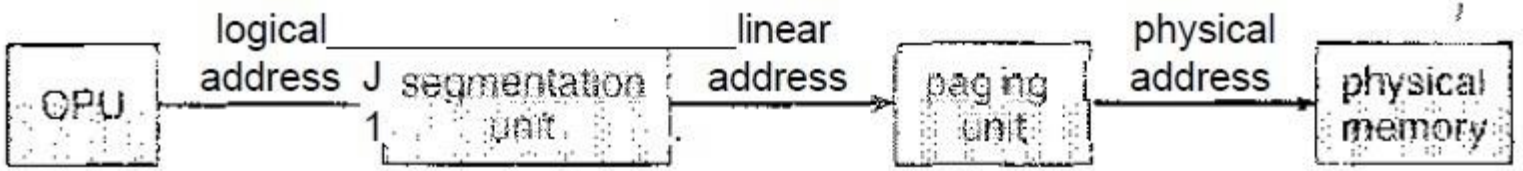
Example of segmentation



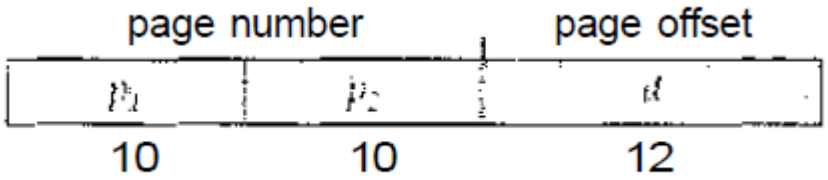
A reference to segment 3 byte 852, is mapped to: 3200 (the base of segment 3) + 852 = 4052.

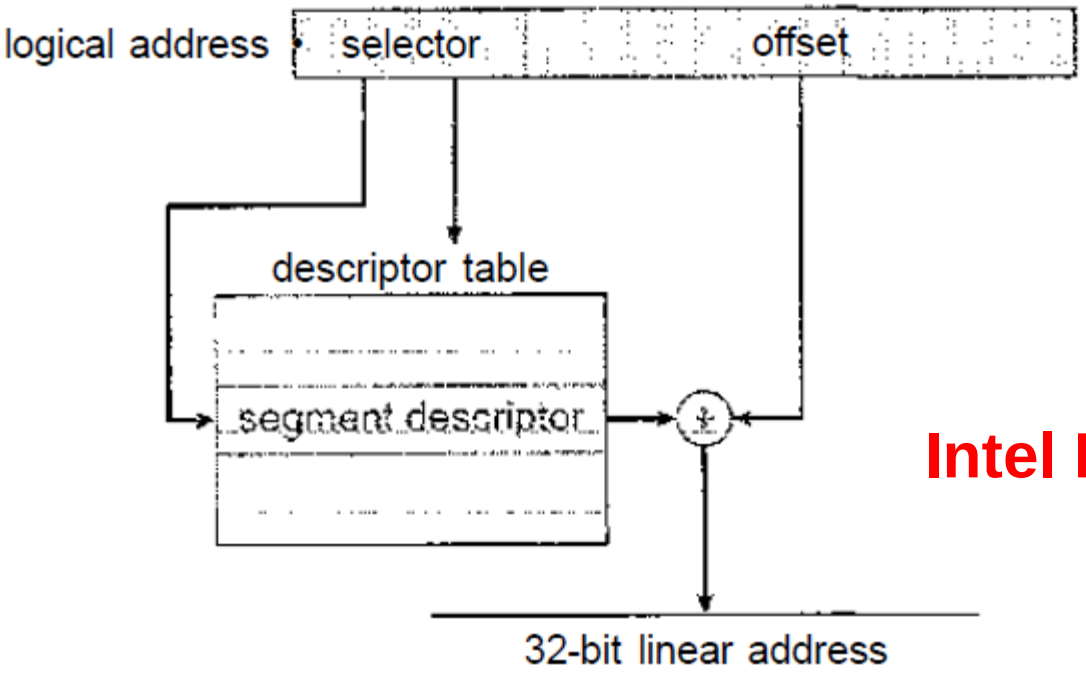
Segmentation with Paging Example: Pentium

Logical to physical address translation in the Pentium



Pentium Paging





Intel Pentium Segmentation.

Consider the following segment table:

Segment	Base	Length
0	219	600
1	2300	14
2	90	100
3	1327	580
4	1952	96

What are the physical addresses for the following logical addresses?

a. 0,430

b. 1,10

c. 2,500

d. 3,400

e. 4,112

Solved Question On Segmentation: SId

Consider the following Segment table:

Segment	Base	Length
0	219	600
1	2300	14
2	90	100
3	1327	580
4	1952	96

What are the physical addresses for the following logical addresses?

i) 0|430 → s=0, d=430 → = B+d (d<Length)
430 < 600 → 219+430 = 649

iii) 1|10 → s=1, d=10 = 2300+10 = 2310

iii) 2|550 → d>Length (illegal)

iv) 3|400 → 1327 + 400 = 1727.

iv) 4|112 → d>L = illegal.

Virtual Memory(VM)

Objectives:

- To describe the benefits of a virtual memory system.
- To explain the concepts of demand paging, page-replacement algorithms, and allocation of page frames.

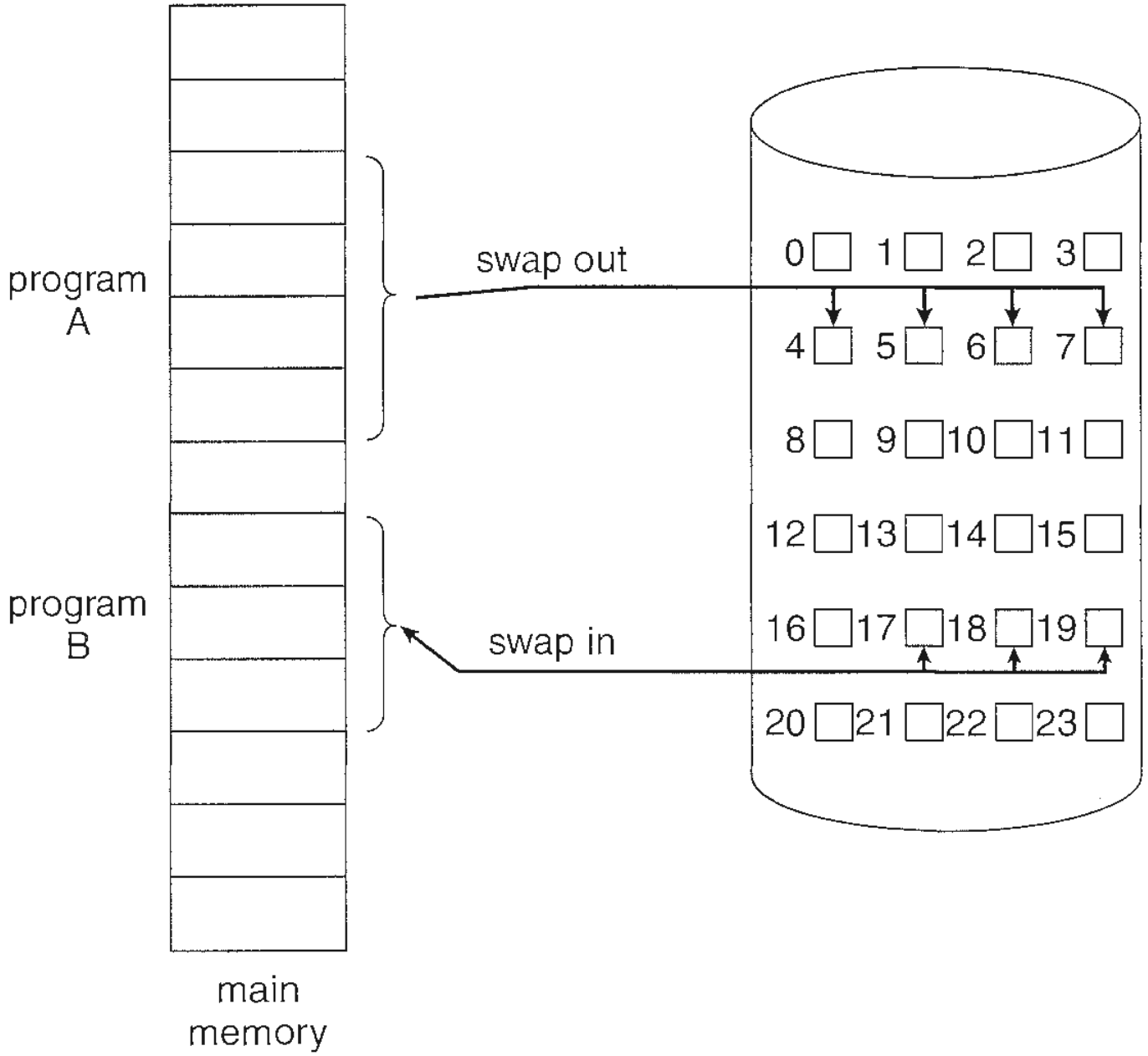
Virtual memory is a technique that allows the execution of processes that are not completely in memory.

- One major advantage of VM is that programs can be larger than physical memory.
- VM frees programmers from the concerns of memory-storage limitations.
- VM also allows processes to share files easily and to implement shared memory.
- VM is not easy to implement.

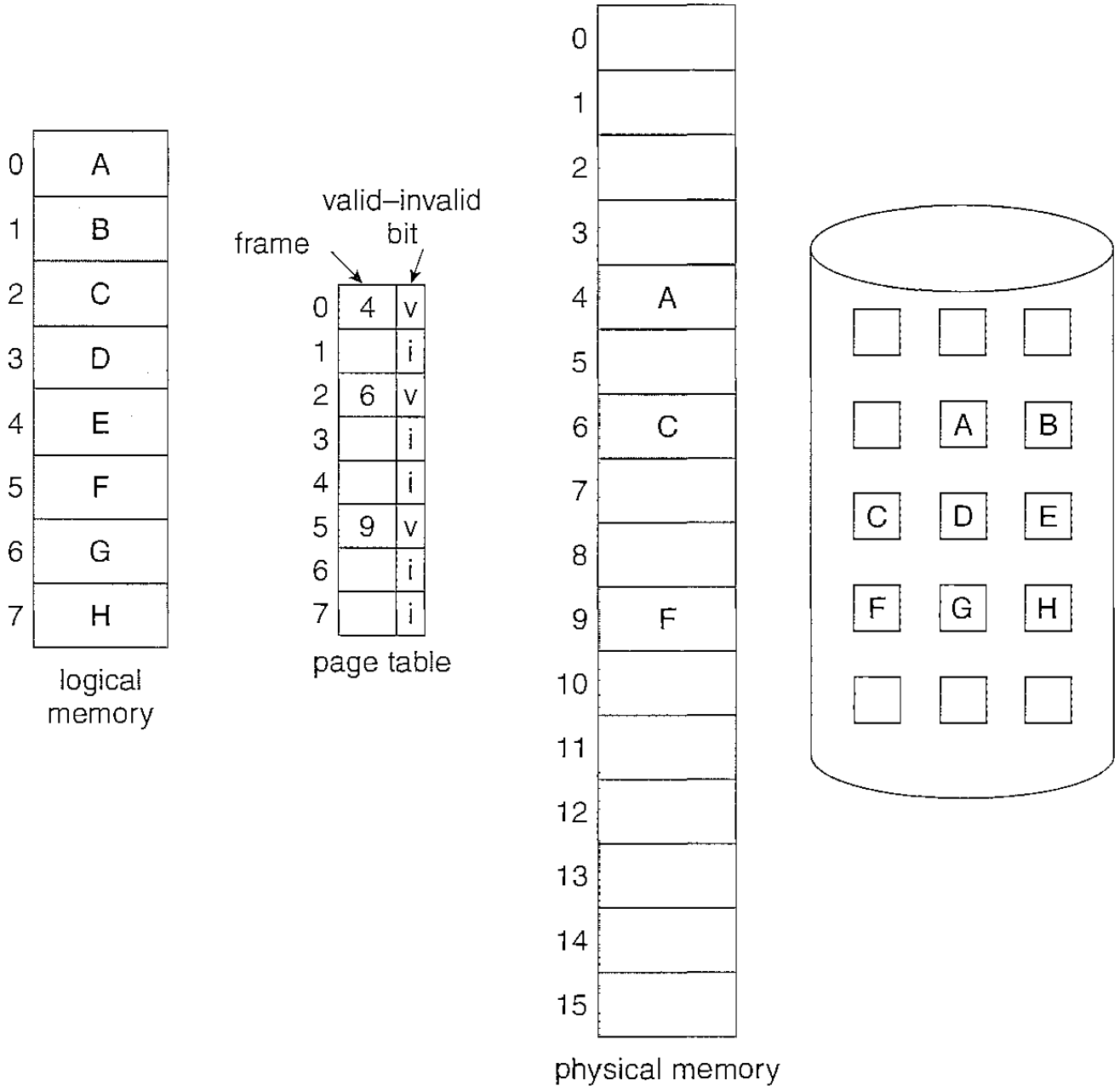
Demand Paging

- How an executable program might be loaded from disk 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.*
- An alternative strategy is to initially 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 only loaded when they are demanded during program execution; pages that are never accessed are thus never loaded into physical memory.
- A demand-paging system is similar to a paging system with swapping.
- Rather than swapping the entire process into memory, however, we use a **lazy swapper.**
- A **lazy swapper** never swaps a page into memory unless that page will be needed.

Transfer of a paged memory to contiguous disk space

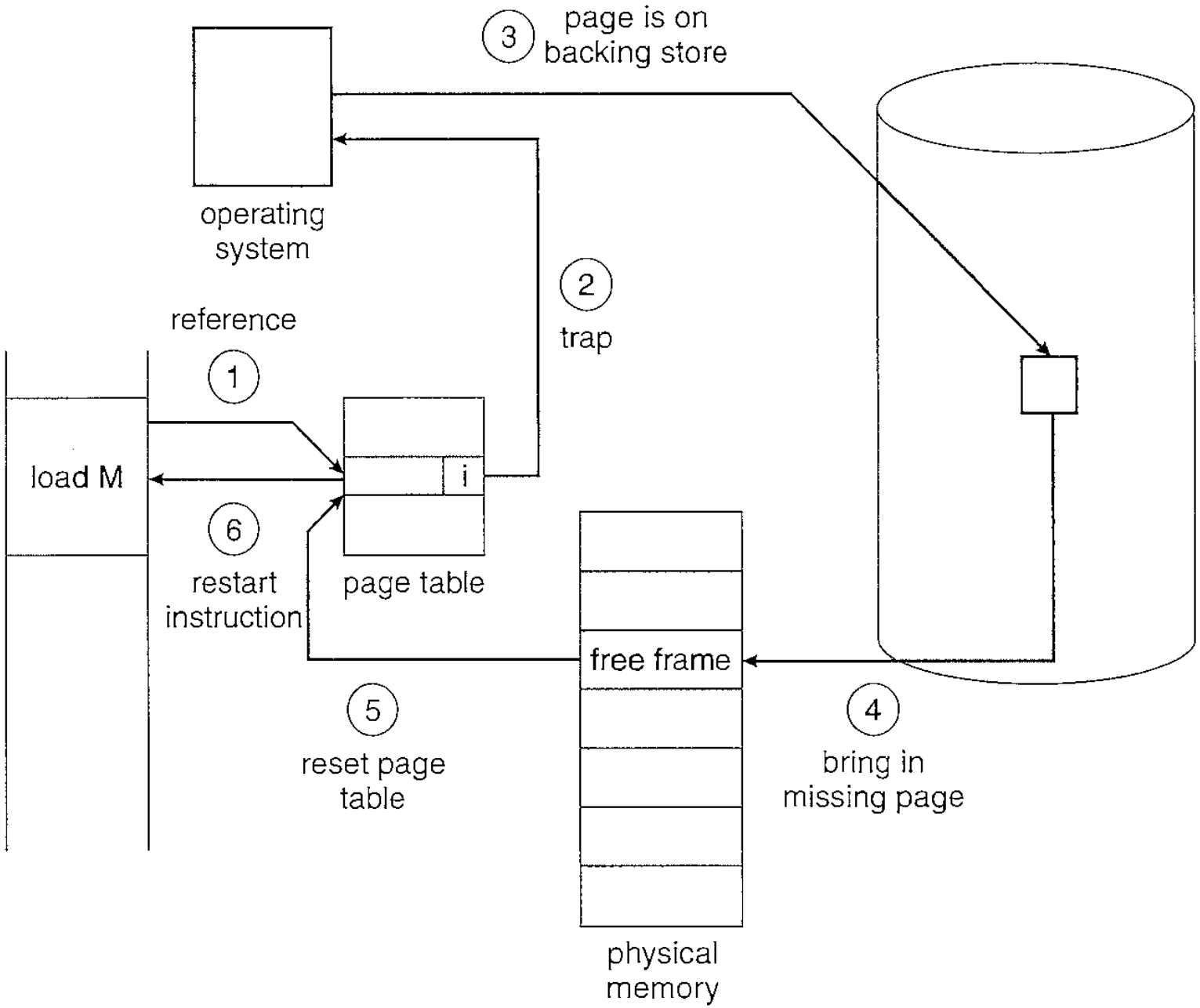


- A swapper manipulates entire processes whereas a pager is concerned with the individual pages of a process.
- Use *pager, rather than swapper, in* connection with demand paging.
- When a process is to be swapped in, the pager guesses which pages will be used before the process is swapped out again.
- Instead of swapping in a whole process, the **pager** brings only those necessary pages into memory.
- Thus, it avoids reading into memory pages that will not be used anyway, decreasing the swap time and the amount of physical memory needed.



Page table when some pages are not in main memory.

Steps in Handling a Page Fault



The procedure for handling the page fault is straightforward:

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

Performance of Demand Paging

- The effective access time for a demand-paged memory.
- To compute the effective access time, we must know how much time is needed to service a page fault.

A page fault causes the following sequence to occur:

1. Trap to the operating system.
2. Save the user registers and process state.
3. Determine that the interrupt was a page fault. '
4. Check that the page reference was legal and determine the location of the page on the disk.
5. Issue a read from the disk to a free frame:
   a. Wait in a queue for this device until the read request is serviced.
   b. Wait for the device seek and /or latency time.
   c. Begin the transfer of the page to a free frame.
6. While waiting, allocate the CPU to some other user (CPU scheduling, optional).
7. Receive an interrupt from the disk I/O subsystem (I/O completed).
8. Save the registers and process state for the other user (if step 6 is executed).
9. Determine that the interrupt was from the disk.
10. Correct the page table and other tables to show that the desired page is now in memory.
11. Wait for the CPU to be allocated to this process again.
12. Restore the user registers, process state, and new page table, and then resume the interrupted instruction.

- Let p be the probability of a page fault (0 <= p<= 1). We would expect p to be close to zero-that is, we would expect to have only a few page faults.
- The effective access time= (1 - p) x ma + p x page fault time.

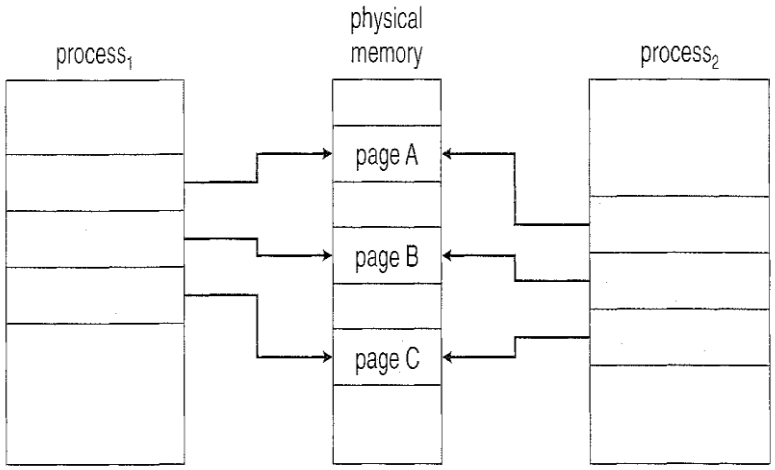
Memory Access Time- ma

In any case, we are faced with three major components of the page-fault service time:

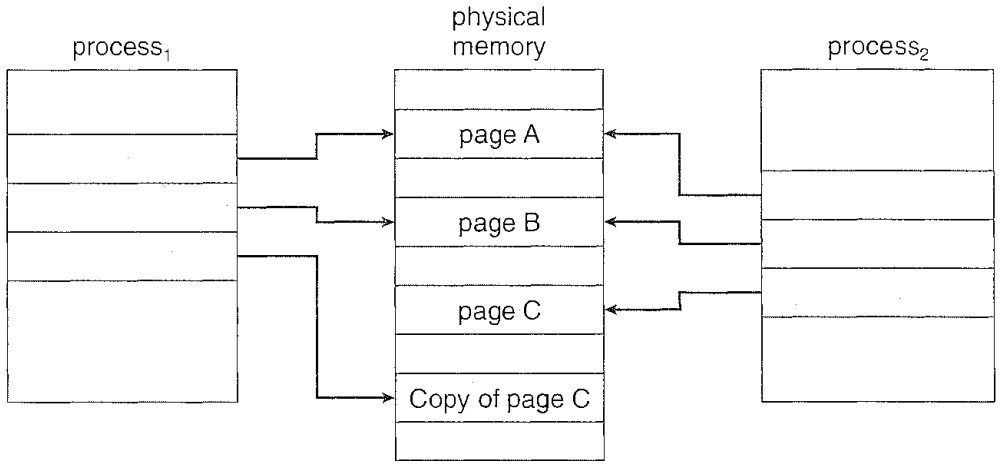
1. Service the page-fault interrupt.
2. Read in the page.
3. Restart the process.

Copy-on-Write

- The fork() system call creates a child process that is a duplicate of its parent.
- Traditionally, **fork()** worked by creating a copy of the parent‘s address space for the child, duplicating the pages belonging to the parent.
- However, considering that many child processes invoke the **exec()** system call immediately after creation, the copying of the parent's address space may be unnecessary.
- which works by allowing the parent and child processes initially to share the same pages.
- These shared pages are marked as copy-on-write pages, meaning that if either process writes to a shared page, a copy of the shared page is created.



Before process 1 modifies page C



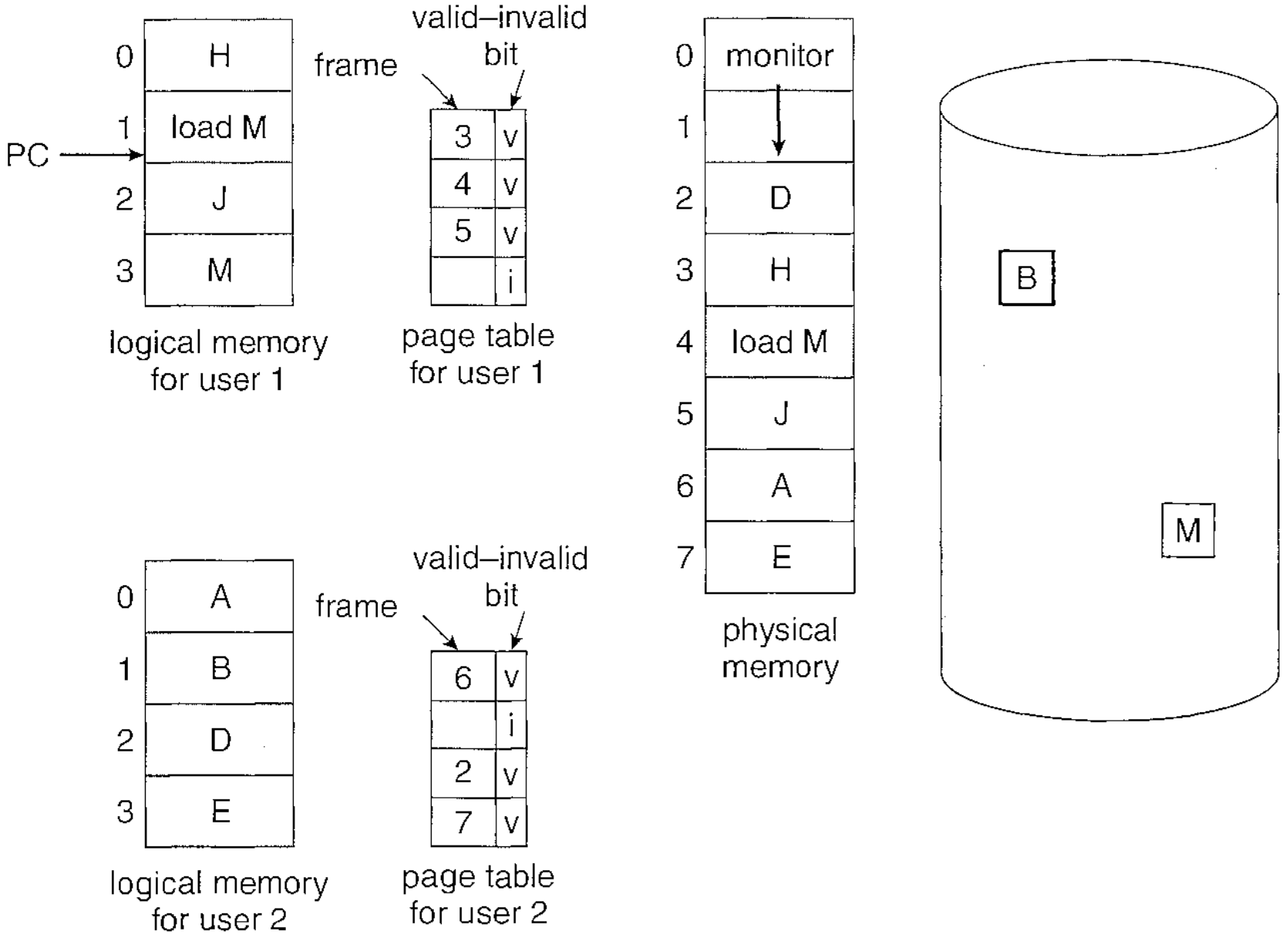
After process 1 modifies page C

Assume that the child process attempts to modify a page containing portions of the stack, with the pages set to be copy-on-write.
The operating system will create a copy of this page, mapping it to the address space of the child process. The child process will then modify its copied page and not the page belonging to the parent process.

- **Copy-on-write** is a common technique used by several operating systems, including Windows XP, Linux, and Solaris.
- Many operating systems provide a pool of free pages for such requests.
- These free pages are typically allocated when the stack or heap for a process must expand or when there are

  copy-on-write pages to be managed.
- Operating systems typically allocate these pages using a technique known as zero-fill-on-demand.
- **Zero-fill-on-demand** pages have been zeroed-out before being allocated, thus erasing the previous contents.
- With vfork()-Virtual Memory Fork, the parent process is suspended, and the child process uses the address space of the parent.
- Therefore, vf ork () must be used with caution to ensure that the child process does not modify the address space of the parent.

Need for page replacement



Basic Page Replacement

- If no frame is free, we find one that is not currently being used and free it.

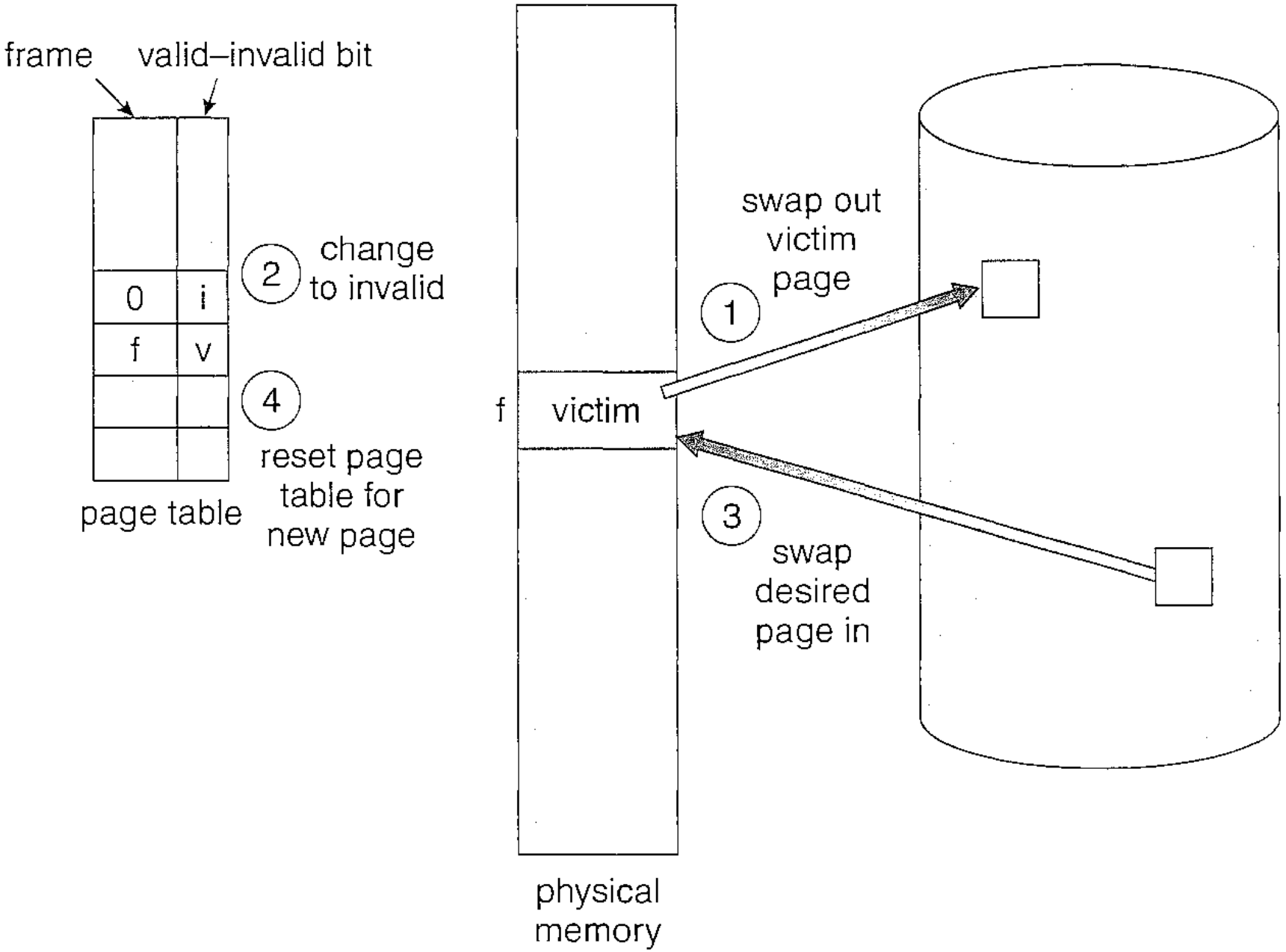
We modify the page-fault service routine to include page replacement:

1. Find the location of the desired page on the disk.
2. Find a free frame:
   a. If there is a free frame, use it.
   b. If there is no free frame, use a page-replacement algorithm to select a victim frame.
   c. Write the victim frame to the disk; change the page and frame tables accordingly.
3. Read the desired page into the newly freed frame; change the page and frame tables.
4. Restart the user process.

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

- We can reduce this overhead by using a **modify bit (or dirty bit).**
- The modify bit for a page is set by the hardware whenever any word or byte in the page is written into, indicating that the page has been modified.
- When we select a page for replacement, we examine its modify bit. If the bit is set, we know that the page has been modified since it was read in from the disk. In this case, we must write the page to the disk.
- If the modify bit is not set, however, the page has not been modified since it was read into memory. In this case, we need not write the memory page to the disk: it is already there. This technique also applies to read-only pages.

Page replacement



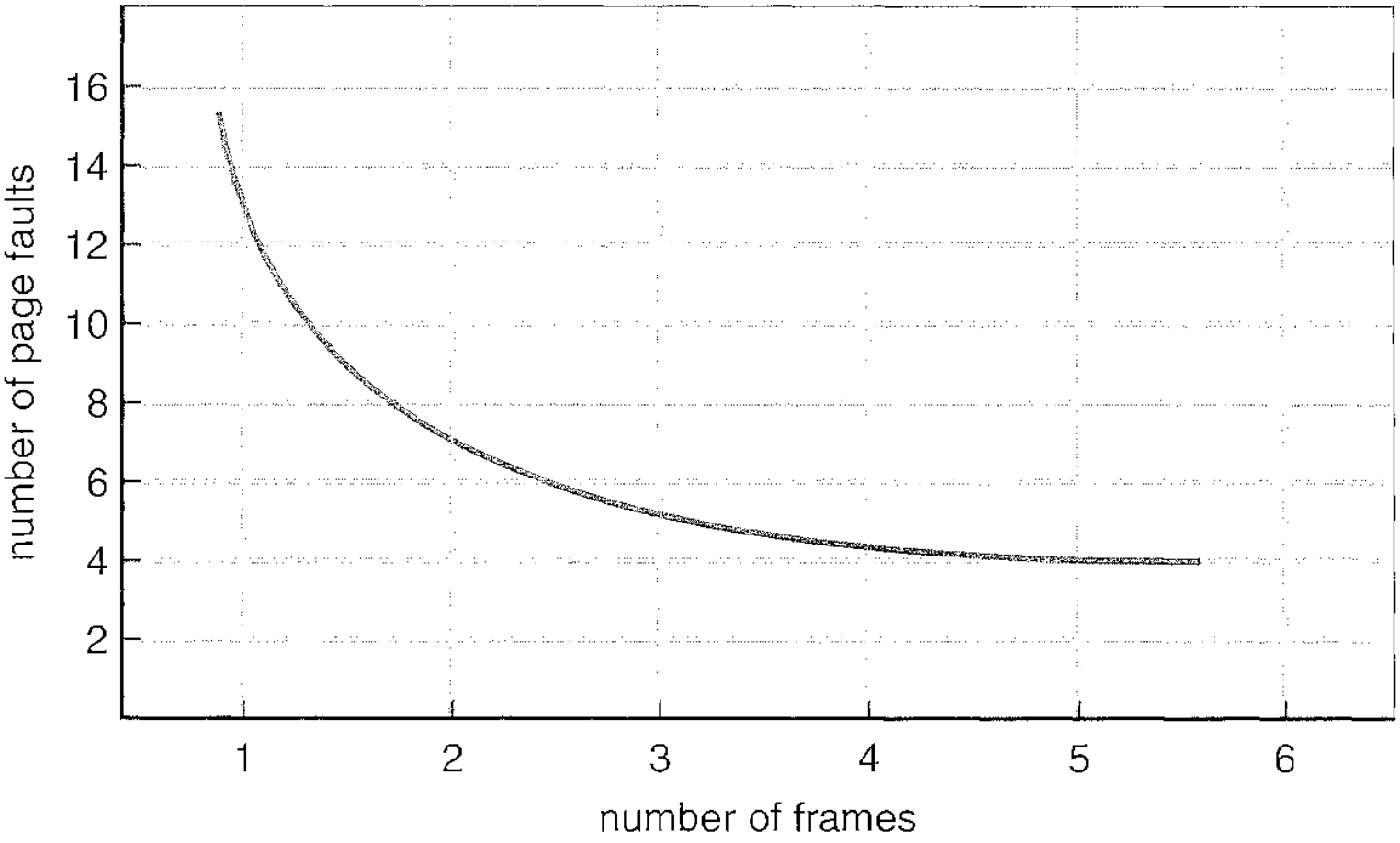
- Page replacement is basic to demand paging. It completes the separation between logical memory and physical memory.
- With demand paging, the size of the logical address space is no longer constrained by physical memory.

We must solve two major problems to implement demand paging: We must develop

1. **A frame-allocation algorithm** and
2. **A page-replacement algorithm.**

- If we have multiple processes in memory, we must decide how many frames to allocate to each process. Further, when page replacement is required, we must select the frames that are to be replaced.
- 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.**

Graph of page faults versus number of frames.

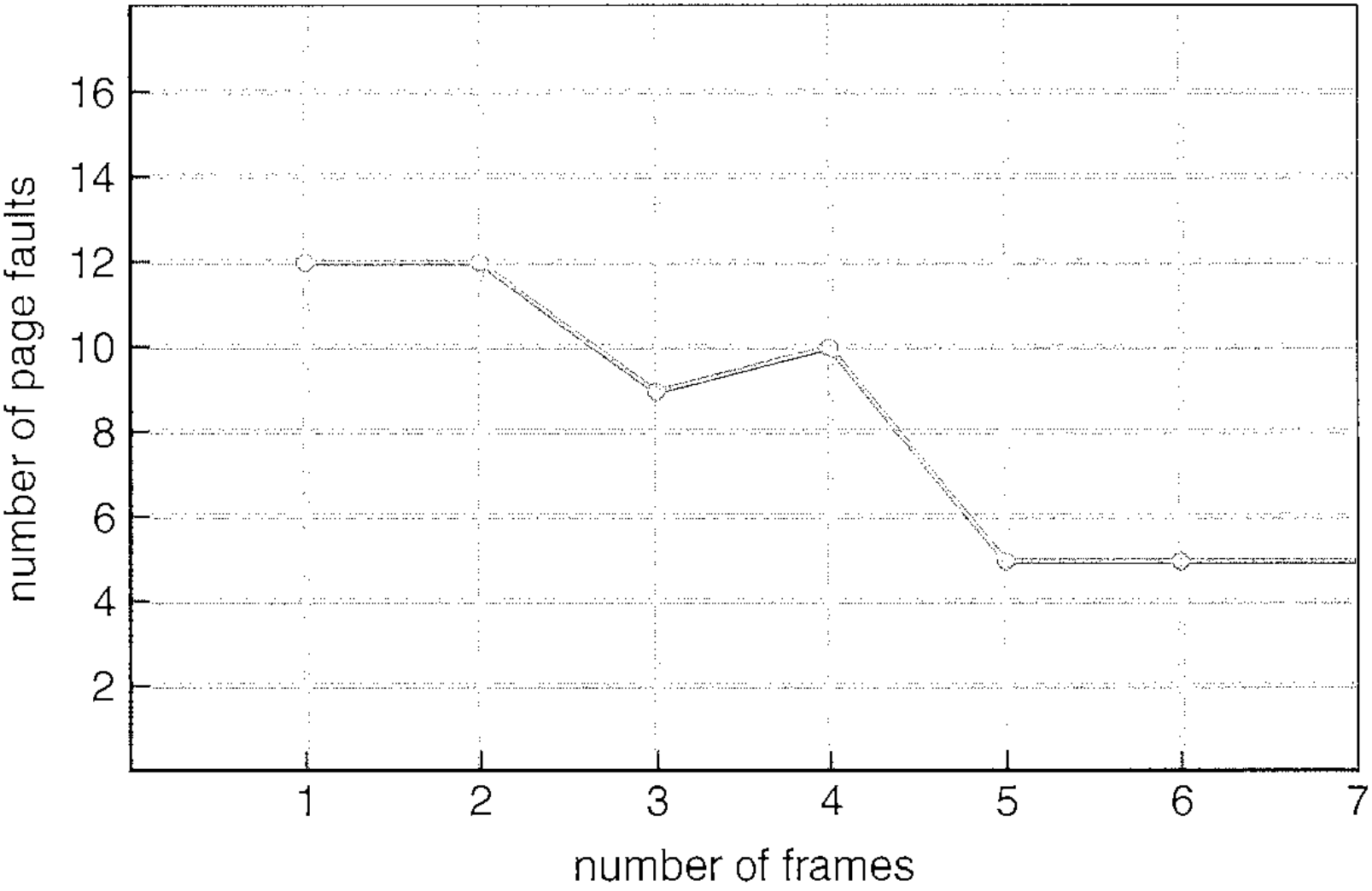


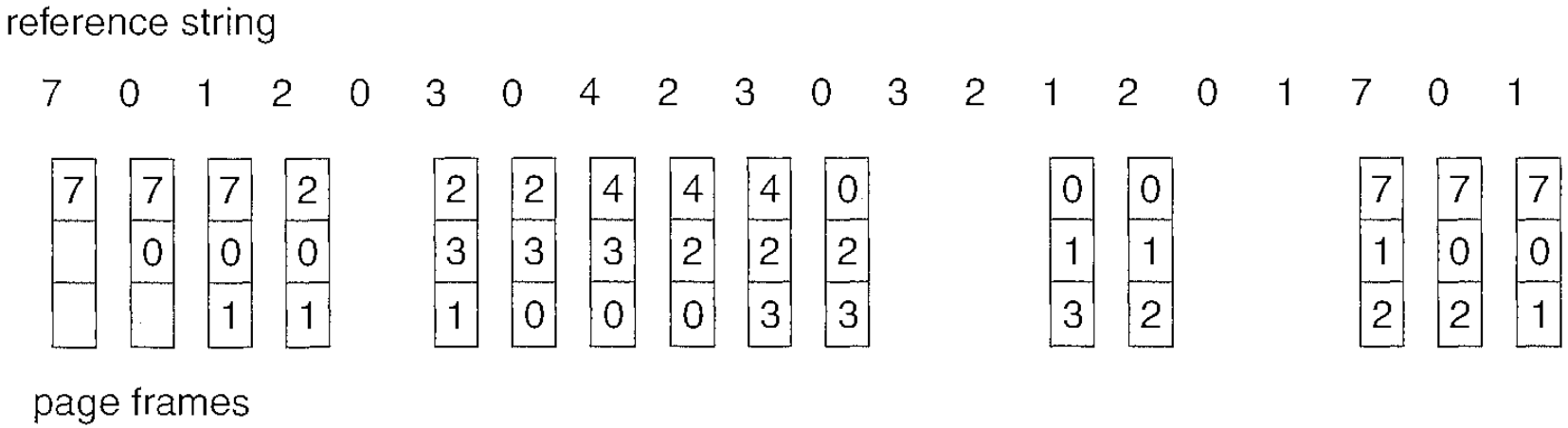
As the number of frames available increases, the number of page faults decreases.

FIFO Page Replacement

- The simplest page-replacement algorithm is a first-in, first-out (FIFO) algorithm.
- When a page must be replaced, the oldest page is chosen.
- 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.
- 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. (reference to graph)
- We would expect that giving more memory to a process would improve its performance.

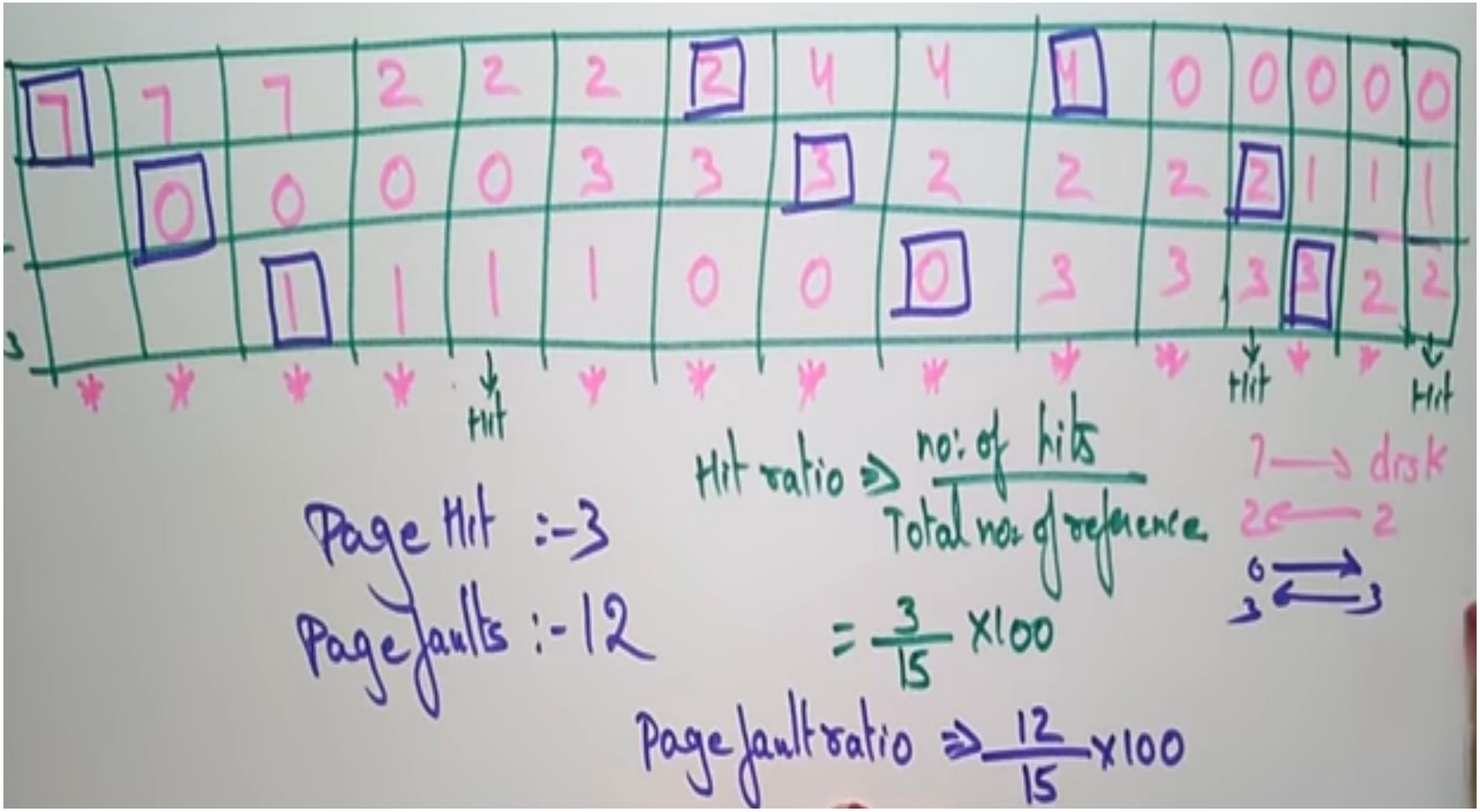
Page-fault curve for FIFO replacement on a reference string





- The FIFO page-replacement algorithm is easy to understand and program.
- However, its performance is not always good.
- On the one hand, the page replaced may be an initialization module that was used a long time ago and is no longer needed.
- On the other hand, it could contain a heavily used variable that was initialized early and is in constant use.

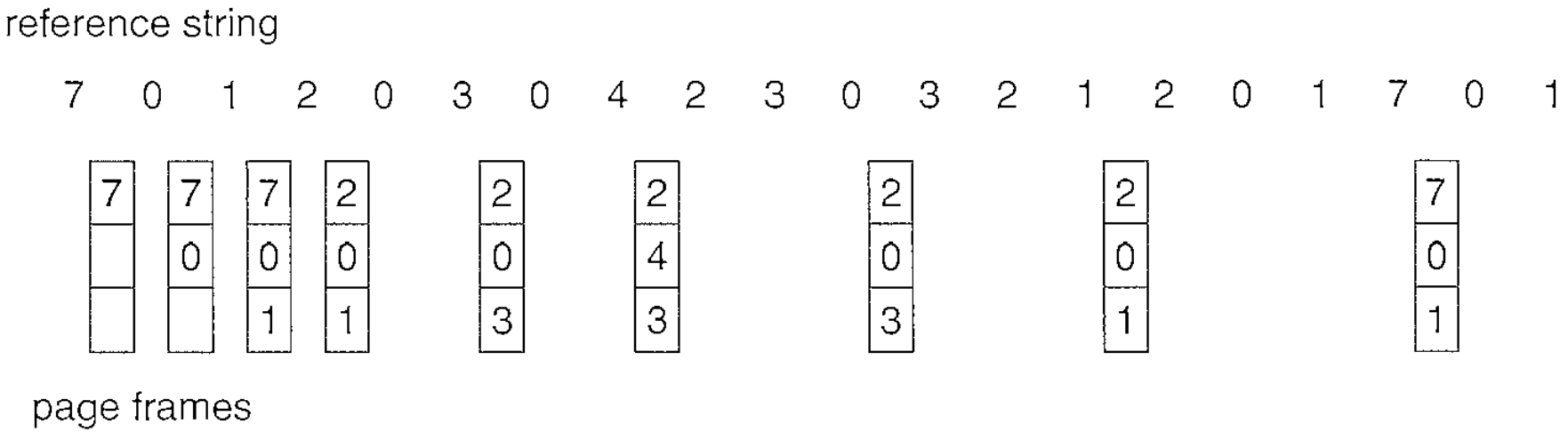
FIFO


7 7 7 2 2 2 2 4 4 1 0 0 0 0 0
0 0 0 0 3 3 3 2 2 2 2 1 1 1
1 1 1 1 0 0 0 3 3 3 3 2 2
Hit
Hit
Hit
Page Hit :-3
Page faults :-12
Hit ratio ⇒ no: of hits / Total no: of reference
= 3/15 x100
Page fault ratio ⇒ 12/15 x100
7 → disk
2 ← 2

Optimal Page Replacement

- An optimal page-replacement algorithm has the lowest page-fault rate of all algorithms and will never suffer from Belady‘s anomaly.
- **Replace the page that will not be used for the longest period of time.**
- Use of this page-replacement algorithm guarantees the lowest possible page fault rate for a fixed number of frames.
- The optimal page-replacement algorithm is difficult to implement, because it requires future knowledge of the reference string. (a similar situation with the SJF CPU-scheduling algorithm).

Optimal Page-replacement Algorithm.



The optimal page-replacement algorithm is a looking **forward** in time.

Replace the page that will not be used for the longest period of time.

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 (other than looking backward versus forward in time) 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.
- When a page must be replaced, LRU chooses the page that has not been used for the longest period of time.
- **The LRU page-replacement algorithm is a looking backward in time.**

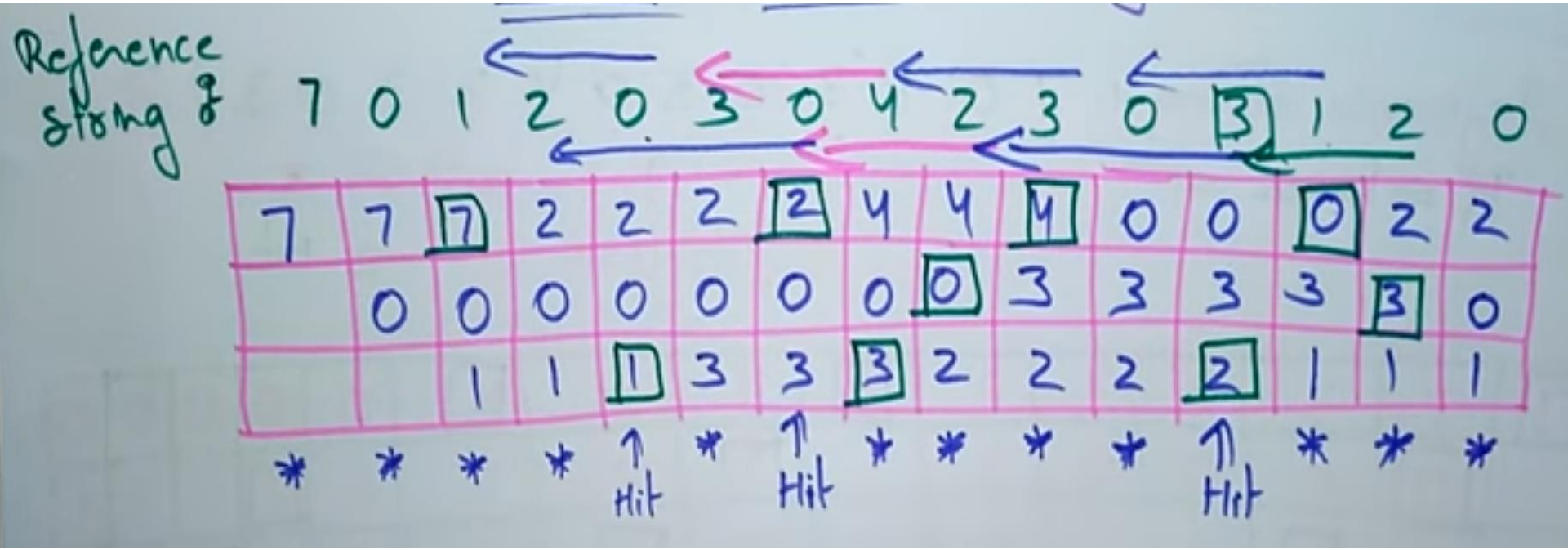
- The LRU policy is often used as a page-replacement algorithm and is considered to be good.
- The major problem is how to implement LRU replacement. An LRU page-replacement algorithm may require substantial hardware assistance.
- The problem is to determine an order for the frames defined by the time of last use.

Two implementations are feasible:

1. **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.
2. **Stack approach** is used 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.

Like optimal replacement, LRU replacement does not suffer from Belady's anomaly.

LRU



Applications and Page Replacement

- **Database.**
- **Data warehouses** frequently perform massive sequential disk reads, followed by computations and writes. The LRU algorithm would be removing old pages and preserving new ones, while the application would more likely be reading older pages than newer ones (as it starts its sequential reads again). Here, MFU would actually be more efficient than LRU.

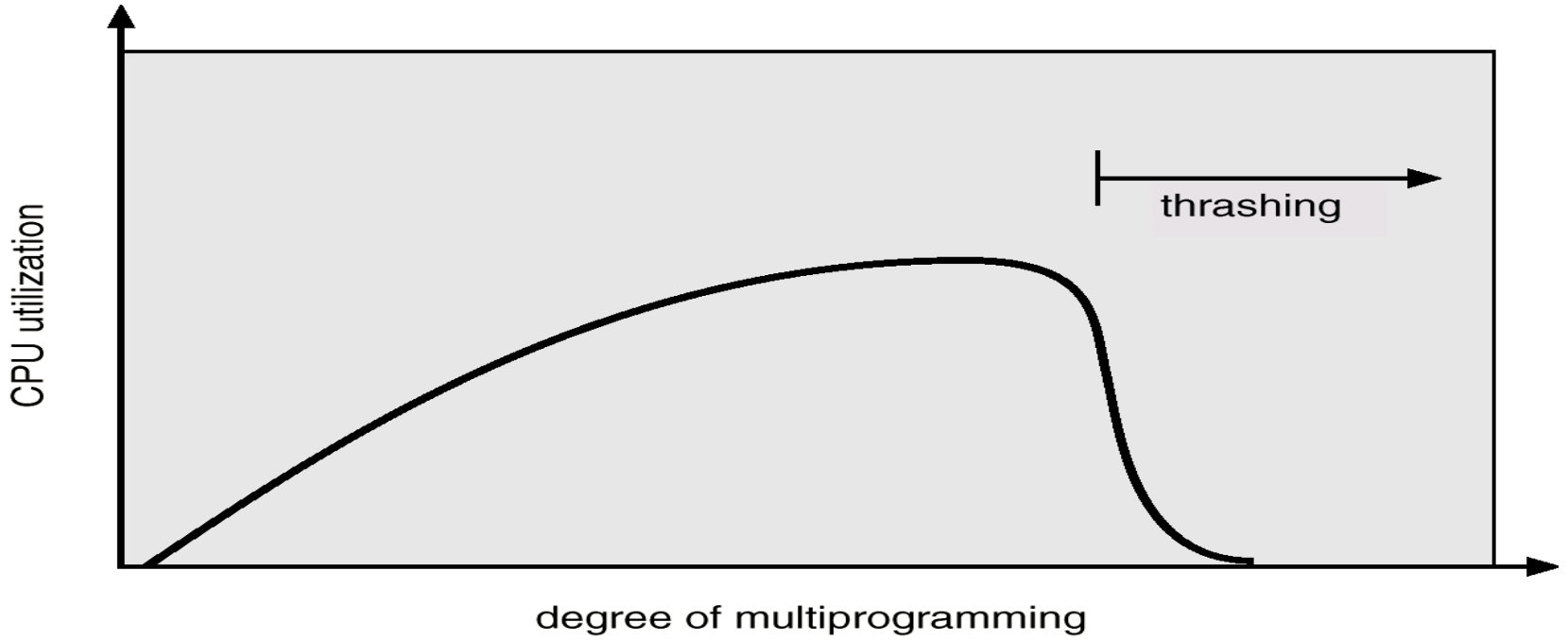
Thrashing

- In fact, look at any process that does not have "enough" frames. If the process does not have the number of frames it needs to support pages in active use, it will quickly page-fault.
- Quickly Page 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 on paging than executing.

Cause of Thrashing :

- Thrashing results in severe performance problems.
- The operating system monitors CPU utilization. If CPU utilization is too low, we increase the degree of multiprogramming by introducing a new process to the system.
- A global page-replacement algorithm is used; it replaces pages without regard to the process to which they belong.
- The CPU scheduler sees the decreasing CPU utilization and increases the degree of multiprogramming as a result.
- The new process tries to get started by taking frames from running processes, causing more page faults and a longer queue for the paging device.
- The page fault rate increases tremendously As a result, the effective memory-access time increases.
- No work is getting done, because the processes are spending all their time paging.

Thrashing.



- 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 even 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 multi programming.

- We can limit the effects of thrashing by using a **Local Replacement Algorithm** (or **priority replacement algorithm).** With local replacement, if one process starts thrashing, it cannot steal frames from another process and cause the latter to thrash as well.
- However, the problem is not entirely solved. If processes are thrashing, they will be in the queue for the paging device most of the time.
- The average service time for a page fault will increase because of the longer average queue for the paging device. Thus, the effective access time will increase even for a process that is not thrashing.

Summary

- If a process does not have “enough” pages, the page-fault rate is very high. This leads to:
  - low CPU utilization.
  - operating system thinks that it needs to increase the degree of multiprogramming.
  - another process added to the system.
- Thrashing - a process is busy swapping pages in and out.
- To prevent thrashing, we must provide a process with as many frames as it needs. But how do we know how many frames it "needs'?
- This approach defines the **locality model of process execution.**

Locality Model of Process Execution.

- The locality model states that, as a process executes, it moves from locality to locality.
- A locality is a set of pages that are actively used together.
- A program is generally composed of several different localities, which may overlap.
- **For example**, when a function is called, it defines a new locality.
- In this locality, memory references are made to the instructions of the function call, its local variables, and a subset of the global variables. When we exit the function, the process leaves this locality, since the local variables and instructions of the function are no longer in active use.
- Thus localities are defined by the program structure and its data structures.

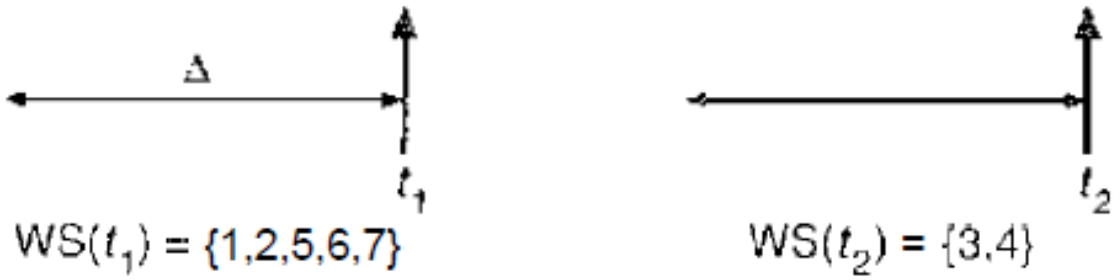
Working-Set Model

- The working-set model is based on the assumption of locality.
- This model uses a parameter, A, to define the working-set window.
- The idea is to examine the most recent A page references.
- The set of pages in the most recent A page references is the working set.
- If a page is in, active use, it will be in the working set. If it is no longer being used, it will drop from the working set A time units after its last reference.
- Thus, the working set is an approximation of the program's locality.
- This working-set strategy prevents thrashing while keeping the degree of multiprogramming as high as possible. Thus, it optimizes CPU utilization.

- The difficulty with the working-set model is keeping track of the working set.
- The working-set window is a moving window. At each memory reference, a new reference appears at one end and the oldest reference drops off the other end.
- A page is in the working set if it is referenced anywhere in the working-set window.

page reference table

. . . 2 6 1 5 7 7 7 7 5 1 6 2 3 4 1 2 3 4 4 4 3 4 3 4 4 4 1 3 2 3 4 4 4 3 4 4 4 . . .

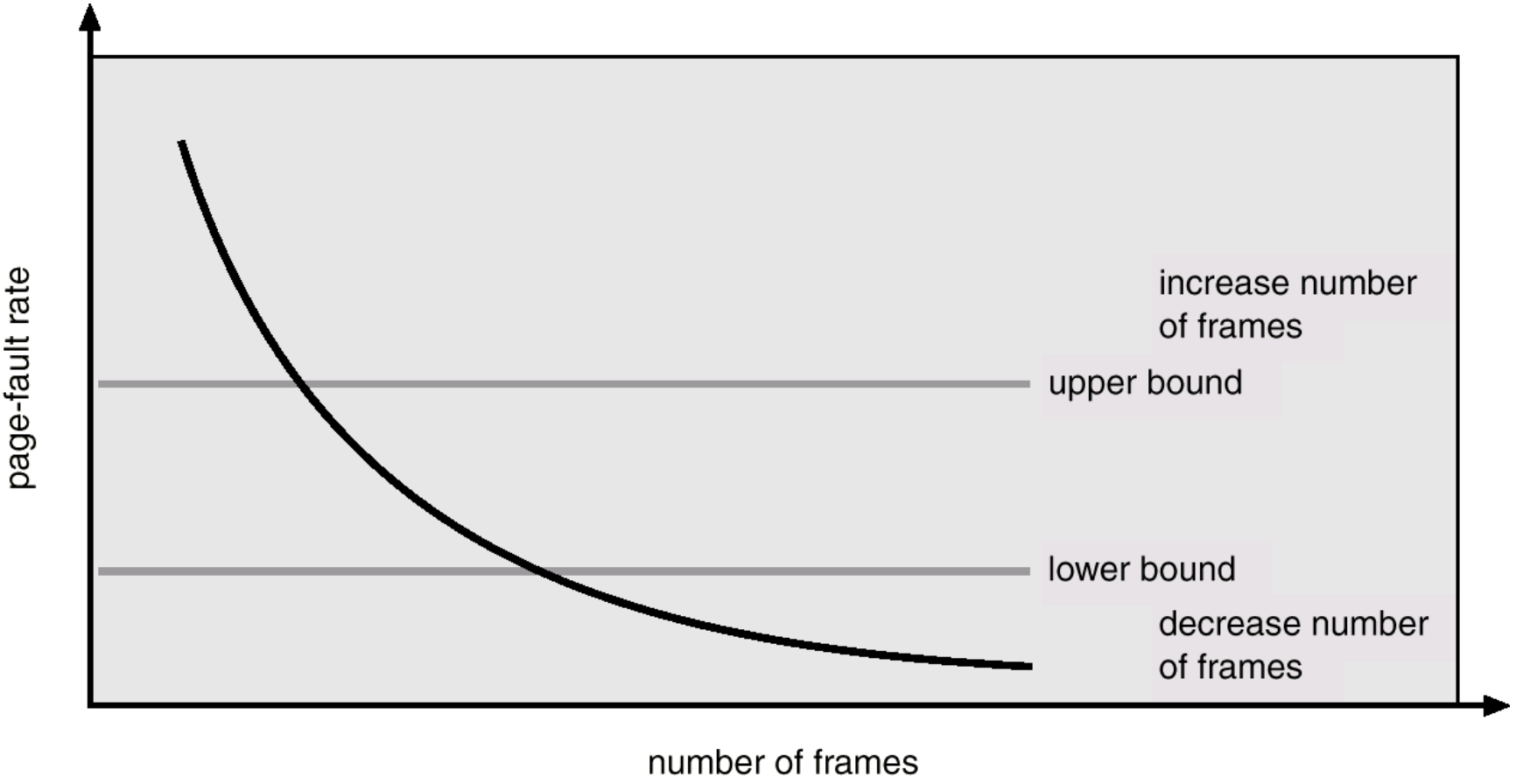


$WS(t_1) = \{1,2,5,6,7\}$ $WS(t_2) = \{3,4\}$

Page-Fault Frequency (PFF) Scheme

- The working-set model is successful, and knowledge of the working set can be useful for pre-paging, but it seems a clumsy way to control thrashing.
- A strategy that uses the **page-fault frequency (PFF)** takes a more direct approach.
- The specific problem is how to prevent thrashing.
- 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.
- On the other hand, 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.

- 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.
- As with the working-set strategy, we may have to suspend a process. If the page-fault rate increases and no free frames are available, we must select some process and suspend it.
- The freed frames are then distributed to processes with high page-fault rates.



Establish "acceptable" page-fault rate.

If actual rate too low, process loses frame.

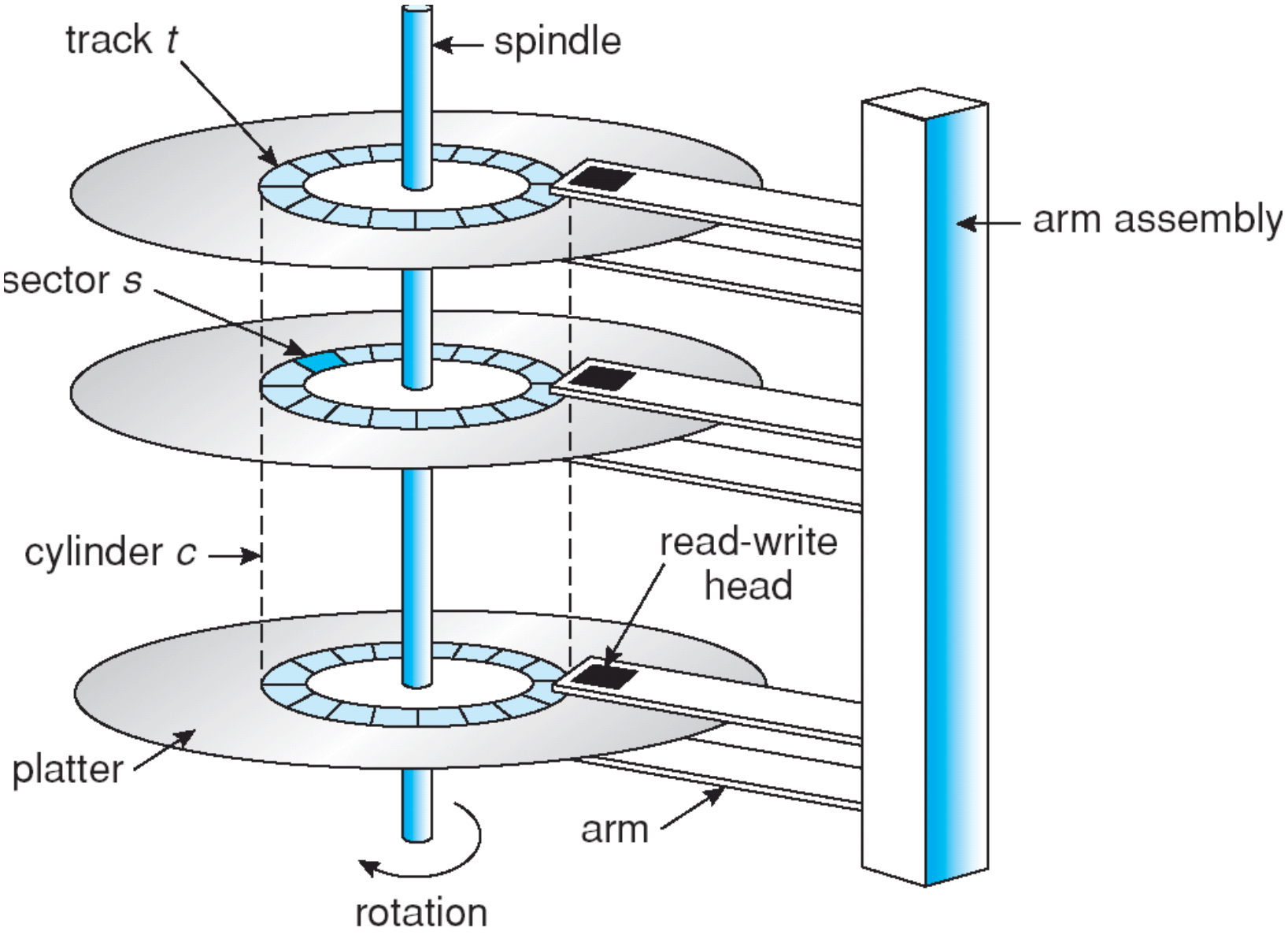
If actual rate too high, process gains frame.

SECONDARY STORAGE STRUCTURE

1. Overview of Mass Storage Structure
2. Disk Structure
3. Disk Scheduling
4. Disk Management

.

What Does The Disk Look Like?



- 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 that are at one arm position makes up a cylinder.
- There may be thousands of concentric cylinders in a disk drive, and each track may contain hundreds of sectors. The storage capacity of common disk drives is measured in gigabytes.
- The positioning time sometimes called the **random access time**, consists of the time necessary to move the disk arm to the desired cylinder, called the **seek time**, and the time necessary for the desired sector to rotate to the disk head, called the **rotational latency.**
- Typical disks can transfer several megabytes of data per second, and they seek times and rotational latencies of several milliseconds.

Overview of Mass Storage Structure

- **Magnetic disks** provide bulk of secondary storage of modern computers Drives rotate at 60 to 200 times per second.
- **Transfer rate** is rate at which data flow between drive and computer.
- **Positioning time (random-access time)** is time to move disk arm to desired cylinder (**seek time**) and time for desired sector to rotate under the disk head (**rotational latency**)- (**seek time+ rotational latency).**
- **Head crash:** The head will sometimes damage the magnetic surface. A head crash normally cannot be repaired; the entire disk must be replaced.
- **Drive attached to computer via I/O bus**
- Busses vary, including **EIDE, ATA, SATA, USB, SCSI-** Enhanced Integrated Drive Electronics (EIDE), Advanced Technology Attachment (ATA), Serial ATA (SATA), Universal Serial Bus (USB), Fiber Channel (FC), and SCSI buses.
  - Host controller in computer uses bus to talk to **disk controller** built into drive or storage array.

Overview of Mass Storage (Cont.)

- **Magnetic tape**
  - Was early secondary-storage medium.
  - Relatively permanent and holds large quantities of data.
  - Access time slow.
  - Random access ~1000 times slower than disk.
  - Mainly used for backup, storage of infrequently-used data, transfer medium between systems.
  - Kept in spool and wound or rewound past read-write head.
  - Once data under head, transfer rates comparable to disk.
  - 20-200GB typical storage.
  - **Common technologies** are 4mm, 8mm, 19mm, LTO-2 and SDLT
  - **Linear Tape-Open** (**LTO**) is a magnetic tape data storage technology originally developed in the late 1990s as an open standards alternative to the proprietary magnetic tape formats that were available at the time. Hewlett Packard Enterprise, IBM and Quantum control the **LTO Consortium**, which directs development and manages licensing and certification of media and mechanism manufacturers.

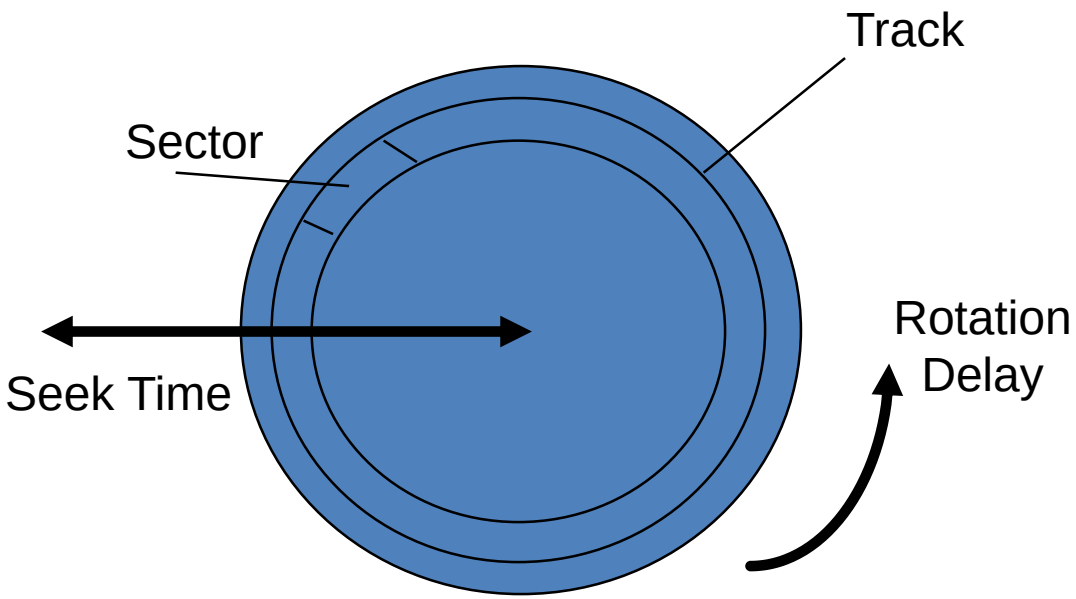

FUJIFILM
ULTRIUM 2
200GB/400GB
NATIVE/COMPRESSED

- **Digital Linear Tape** (**DLT**; previously called **CompacTape**) is a magnetic tape data storage technology developed by Digital Equipment Corporation (DEC) from 1984 onwards.
- In 1994, the technology was purchased by Quantum Corporation, who manufactures drives and licenses the technology and trademark.
- A variant with higher capacity is called **Super DLT (SDLT)**.
- The lower cost "value line" was initially manufactured by Benchmark Storage Innovations under license from Quantum. Quantum acquired Benchmark in 2002.


Quantum®
Super DLTtape™ I
0 Cm
1
2
3
4
5
6
7
8
9

Disk Overheads

- **To read from disk, we must specify:**
  - Cylinder, surface , sector, transfer size, memory address
- **Latency includes:**
  - Seek time: to get to the track
  - Latency time: to get to the sector (rotation delay)
  - Transfer time: get bits off the disk



Modern Disks

	Barracuda 180	Cheetah X15 36LP
Capacity	181GB	36.7GB
Disk/Heads	12/24	4/8
Cylinders	24,247	18,479
Sectors/track	~609	~485
Speed	7200 RPM	15000 RPM
Rotational latency (ms)	4.17	2.0
Avg seek (ms)	7.4	3.6
Track-2-track(ms)	0.8	0.3

50 Years Ago…

- On 13th September 1956, IBM 305 RAMAC computer system first to use disk storage
- 80000 times more data on the 8GB 1-inch drive in his right hand than on the 24-inch RAMAC one in his left…

Disks vs. Memory

- Smallest write: sector
- Atomic write = sector
- Random access: 5ms
  - not on a good curve
- Sequential access: 200MB/s
- Cost $.002MB
- Crash: doesn't matter ("non-volatile")

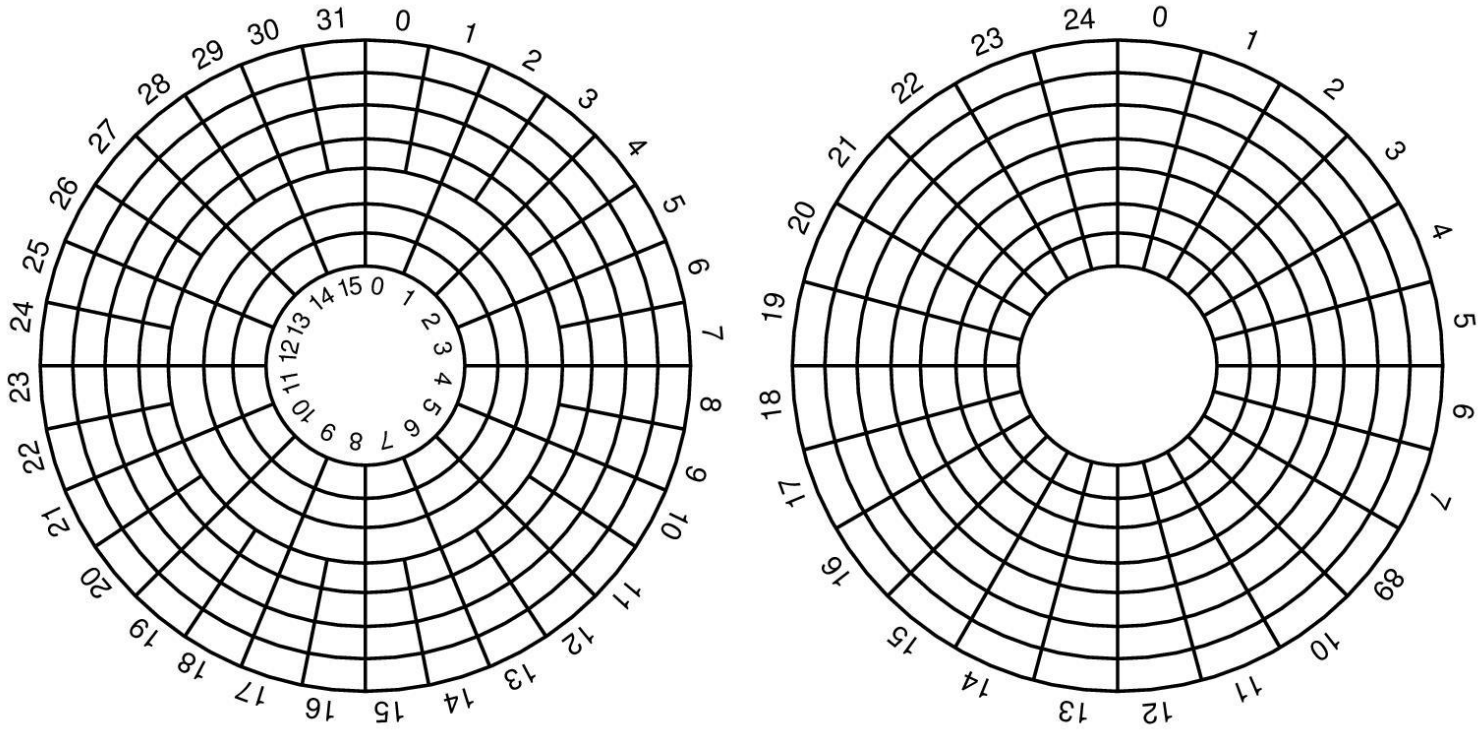
- (usually) bytes
- byte, word
- 50 ns
- faster all the time
- 200-1000MB/s
- $.10MB
- contents gone ("volatile")

Disk Structure

- Disk drives addressed as 1-dimensional arrays of *logical blocks*
  - the logical block is the smallest unit of transfer
  - *Logical block addressing*
- This array mapped sequentially onto disk sectors
  - Address 0 is 1st sector of 1st track of the outermost cylinder
  - Addresses incremented within track, then within tracks of the cylinder, then across cylinders, from innermost to outermost
- Translation is theoretically possible, but usually difficult
  - Some sectors might be defective
  - Number of sectors per track is not a constant

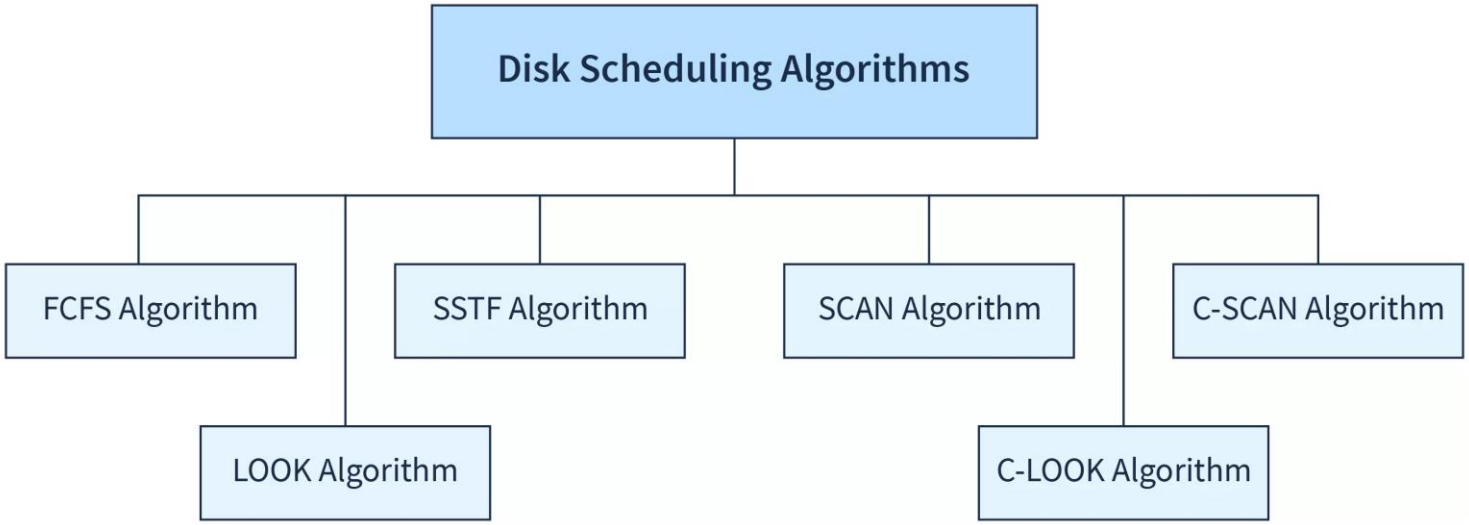
Non-Uniform # Sectors / Track

- Maintain same data rate with **Constant Linear Velocity**
- **Approaches:**
  - Reduce bit density per track for outer layers
  - Have more sectors per track on the outer layers (virtual geometry)



Disk Scheduling

- The operating system tries to use hardware efficiently
  - for disk drives $\Rightarrow$ having fast access time, disk bandwidth
- **Access time has two major components**
- The **seek time** is the time for the disk arm to move the heads to the cylinder containing the desired sector.
- The **rotational latency** is the additional time for the disk to rotate the desired sector to the disk head.
- Want to minimize seek time
- Seek time $\approx$ seek distance
- Disk bandwidth is total number of bytes transferred, divided by the total time between the first request for service and the completion of the last transfer.


Disk Scheduling Algorithms
FCFS Algorithm
SSTF Algorithm
SCAN Algorithm
C-SCAN Algorithm
LOOK Algorithm
C-LOOK Algorithm

Disk Scheduling (Cont.)

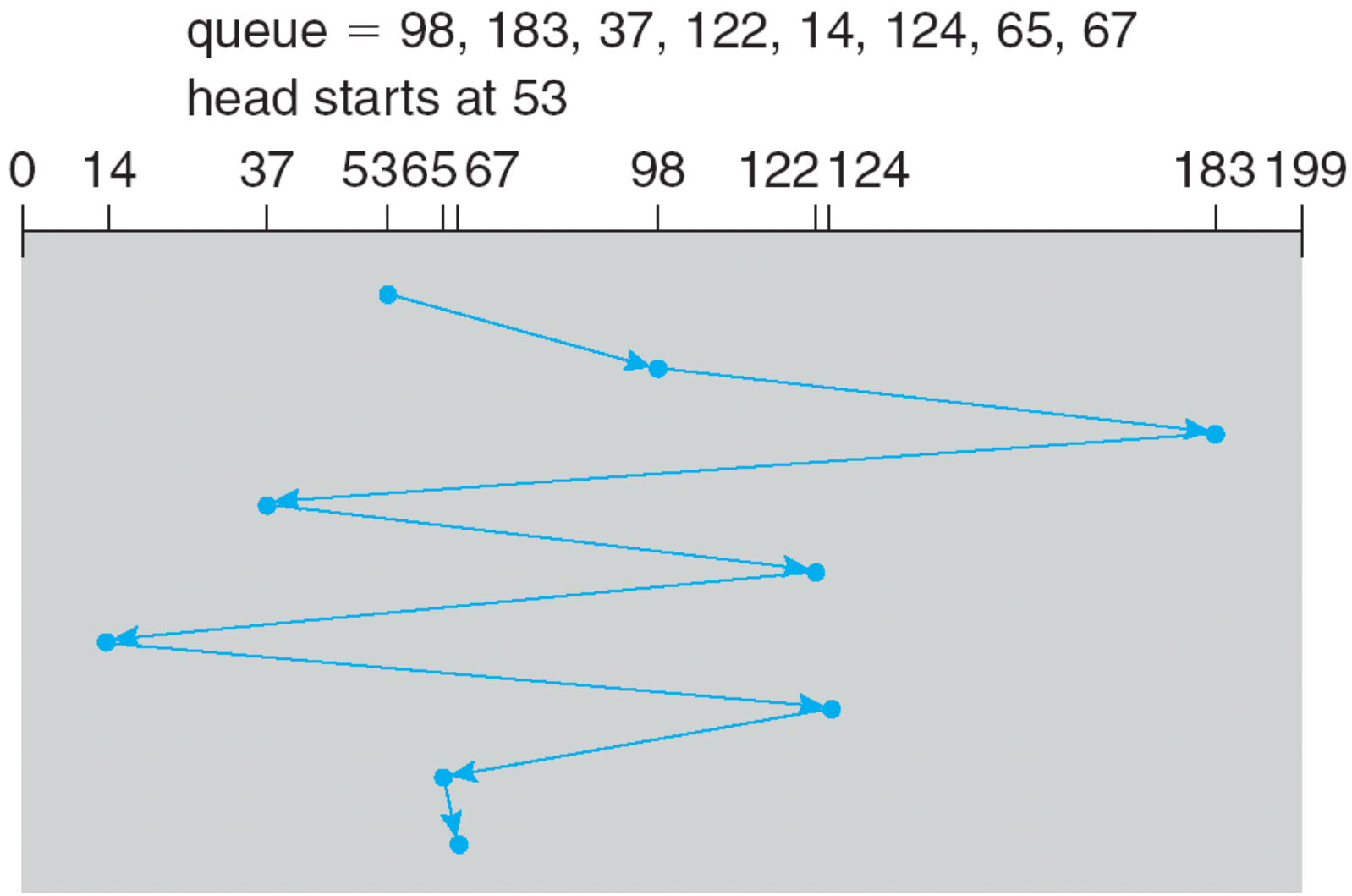
- Several algorithms exist to schedule the servicing of disk I/O requests.
- We illustrate them with a request queue (0-199).

  98, 183, 37, 122, 14, 124, 65, 67

  Head pointer starts at 53
- 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

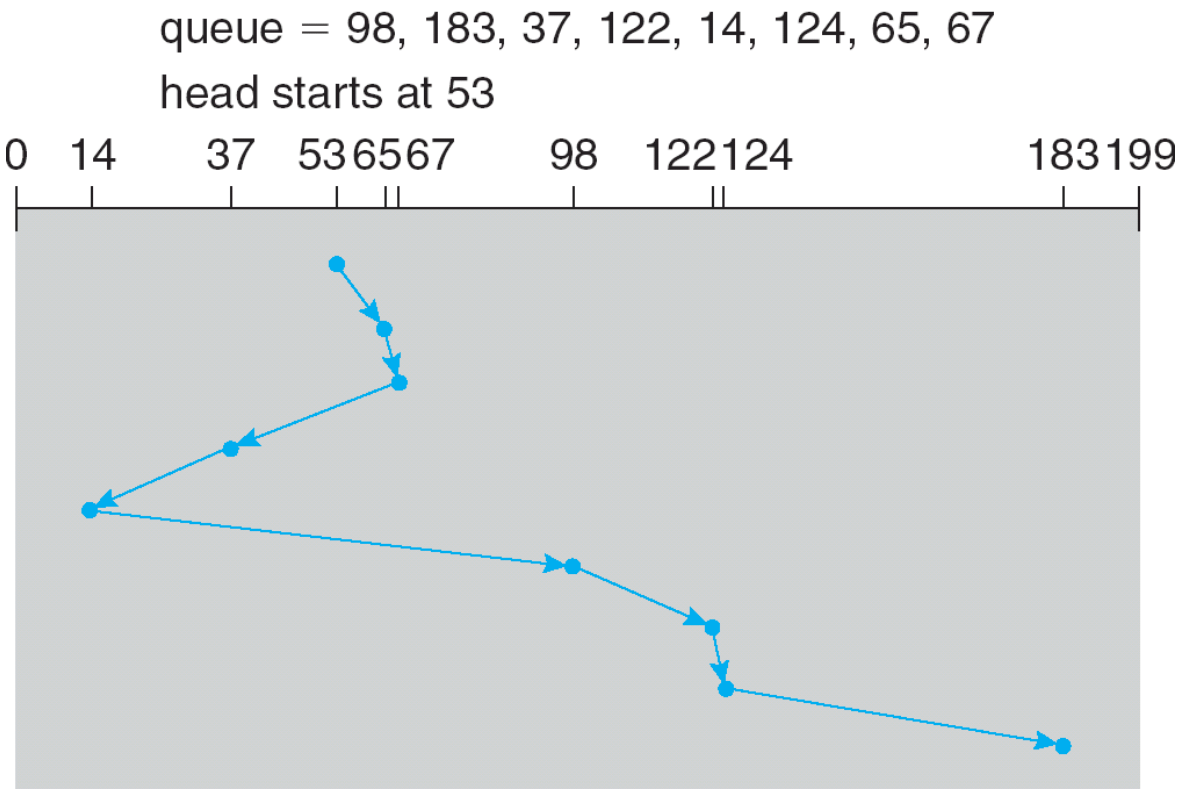
FCFS (First-Come First Serve)

Illustration shows total head total movement of 640 cylinders.



SSTF (Shortest Seek Time First)

- Selects request with minimum seek time from current head position.
- SSTF scheduling is a form of SJF scheduling.
  - may cause starvation of some requests.
- Illustration shows total head movement of 236 cylinders.



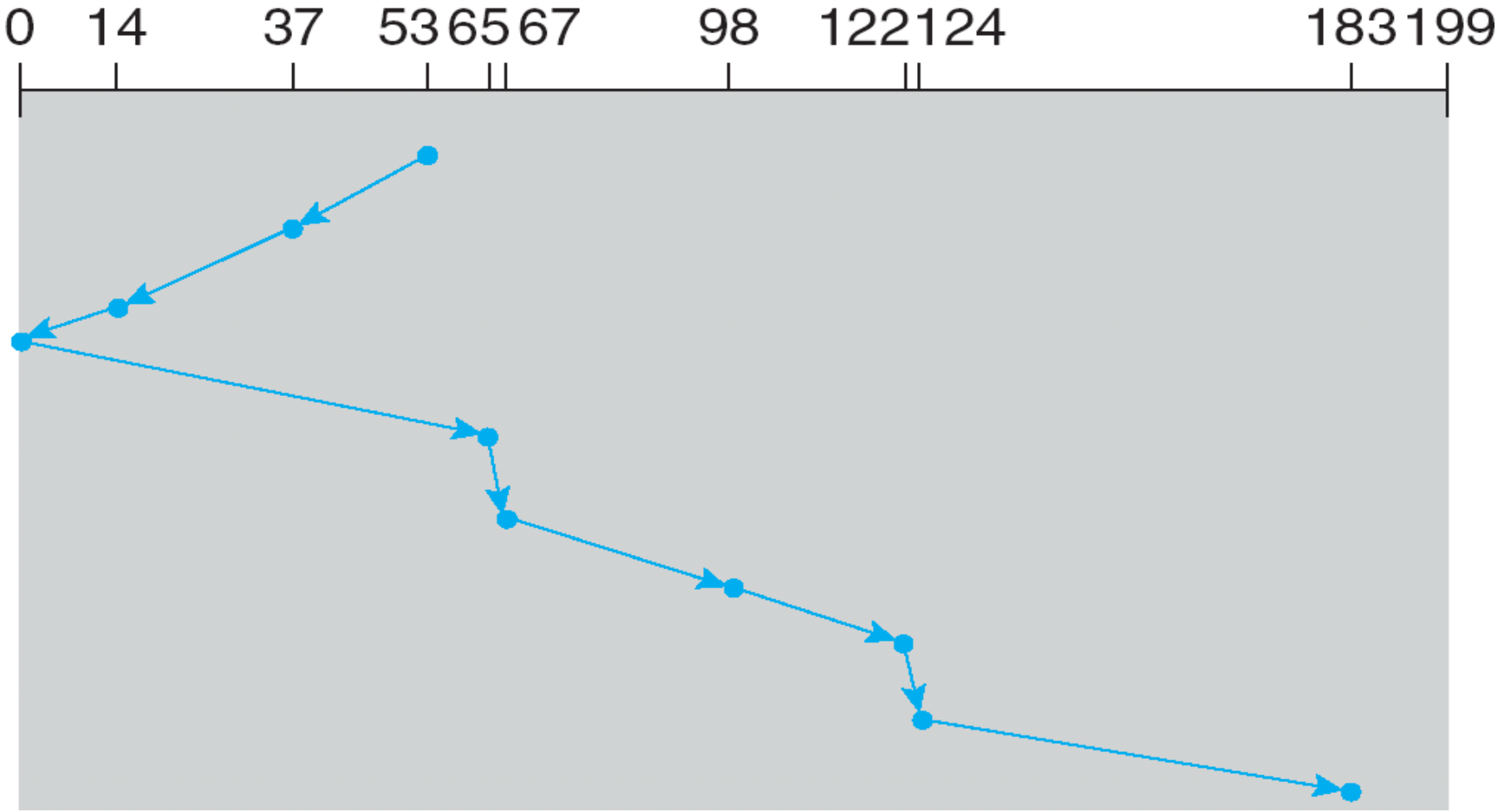
- Although the SSTF algorithm is a substantial improvement over the FCFS algorithm, it is not optimal.
- In the example, we can do better by moving the head from 53 to 37, even though the latter is not closest, and then to 14, before turning around to service 65, 67, 98, 122, 124, and 183.
- This strategy reduces the total head movement to 208 cylinders.

SCAN

- 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.
- Sometimes called the ***elevator algorithm***.
- Illustration shows total head movement of 208 cylinders.

queue = 98, 183, 37, 122, 14, 124, 65, 67

head starts at 53



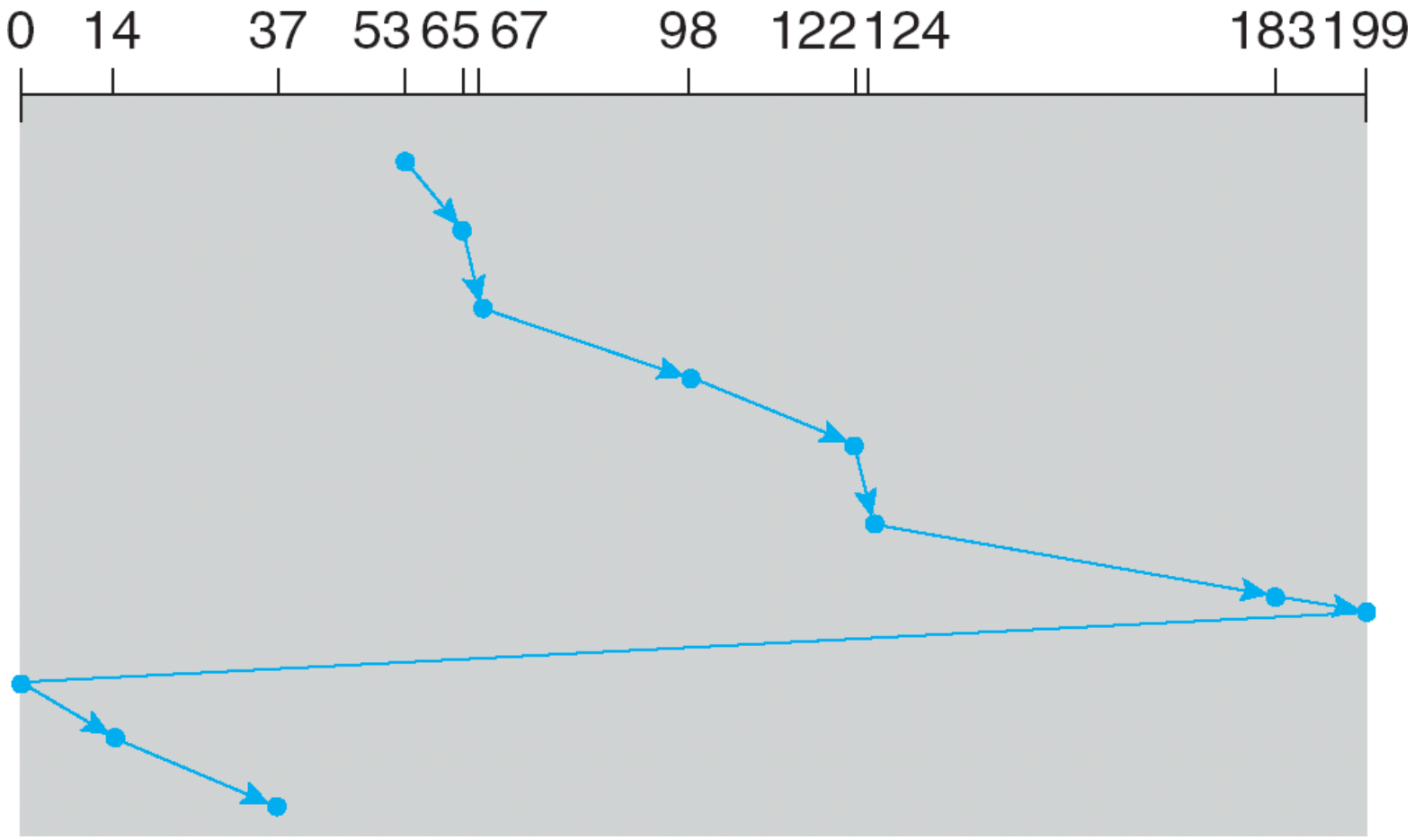
Disadvantage: The requests have also waited the longest time

C-SCAN

- **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.
- Treats the cylinders as a circular list.
  - that wraps around from the last cylinder to the first one.

queue = 98, 183, 37, 122, 14, 124, 65, 67

head starts at 53

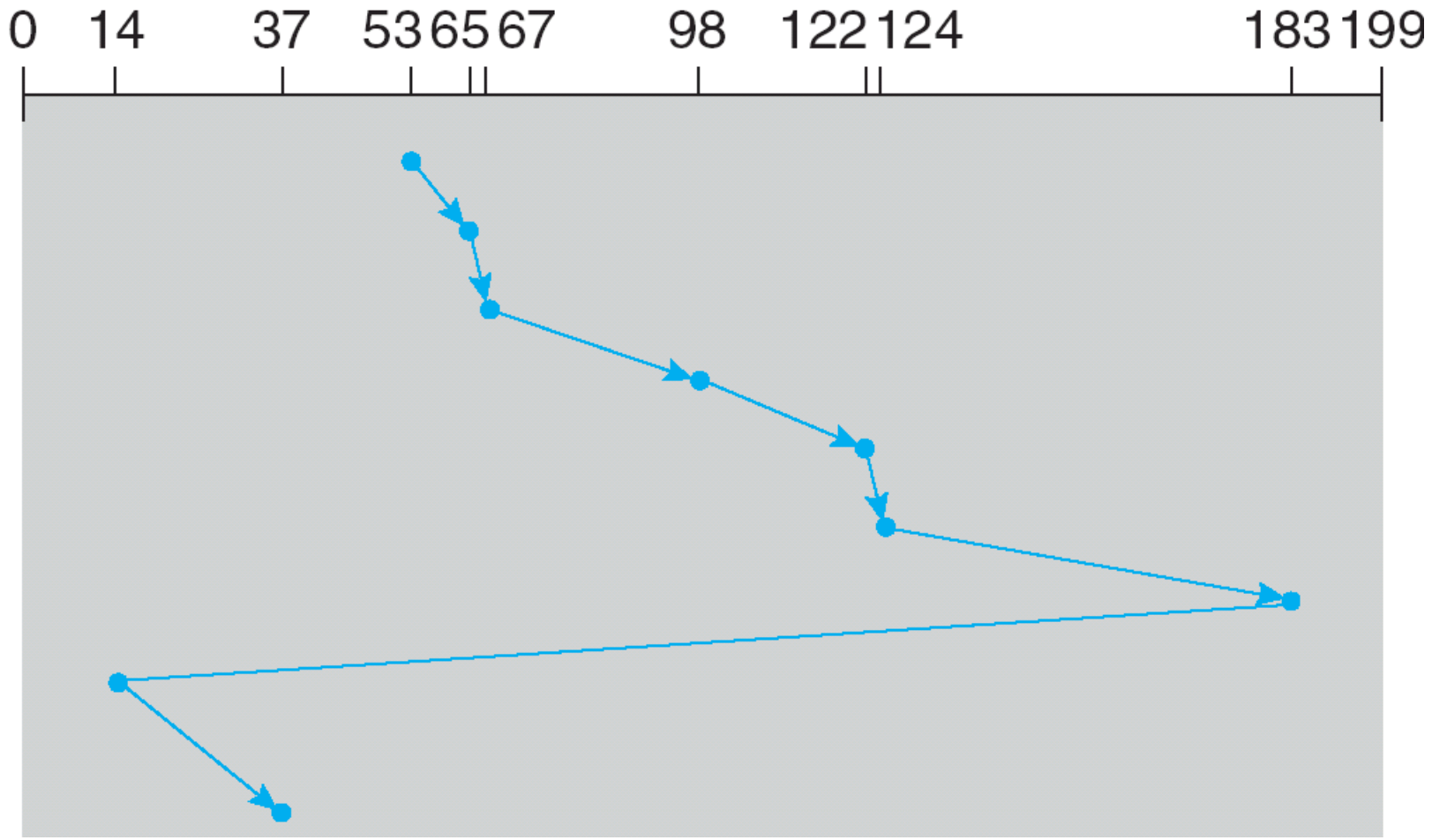


LOOK & C-LOOK

- Versions of SCAN and C-SCAN - **LOOK** and **C-LOOK scheduling,** because they look for a request before continuing to move in a given direction.
- Arm only goes as far as last request in each direction,
  - then reverses direction immediately,
  - without first going all the way to the end of the disk.

queue 98, 183, 37, 122, 14, 124, 65, 67

head starts at 53



Selecting a Good Algorithm

- SSTF is common and has a natural appeal
- SCAN and C-SCAN perform better under heavy load
- Performance depends on number and types of requests
- Requests for disk service can be influenced by the file-allocation method.
- Disk-scheduling algo should be a separate OS module
  - allowing it to be replaced with a different algorithm if necessary.
- Either SSTF or C-LOOK is a reasonable default algorithm.
- With the classical approach of disk scheduling algorithm, few algorithms like **SSTF and LOOK** will be the most efficient algorithm compared to FCFS, SCAN, C-SCAN and C-LOOK disk scheduling algorithm with respect to these parameters.

Disk Scheduling

- The operating system is responsible for using hardware efficiently — for the disk drives, this means having a fast access time and disk bandwidth
- Minimize seek time
- Seek time ≈ seek distance
- Disk **bandwidth** is the total number of bytes transferred, divided by the total time between the first request for service and the completion of the last transfer

Disk Scheduling (Cont.)

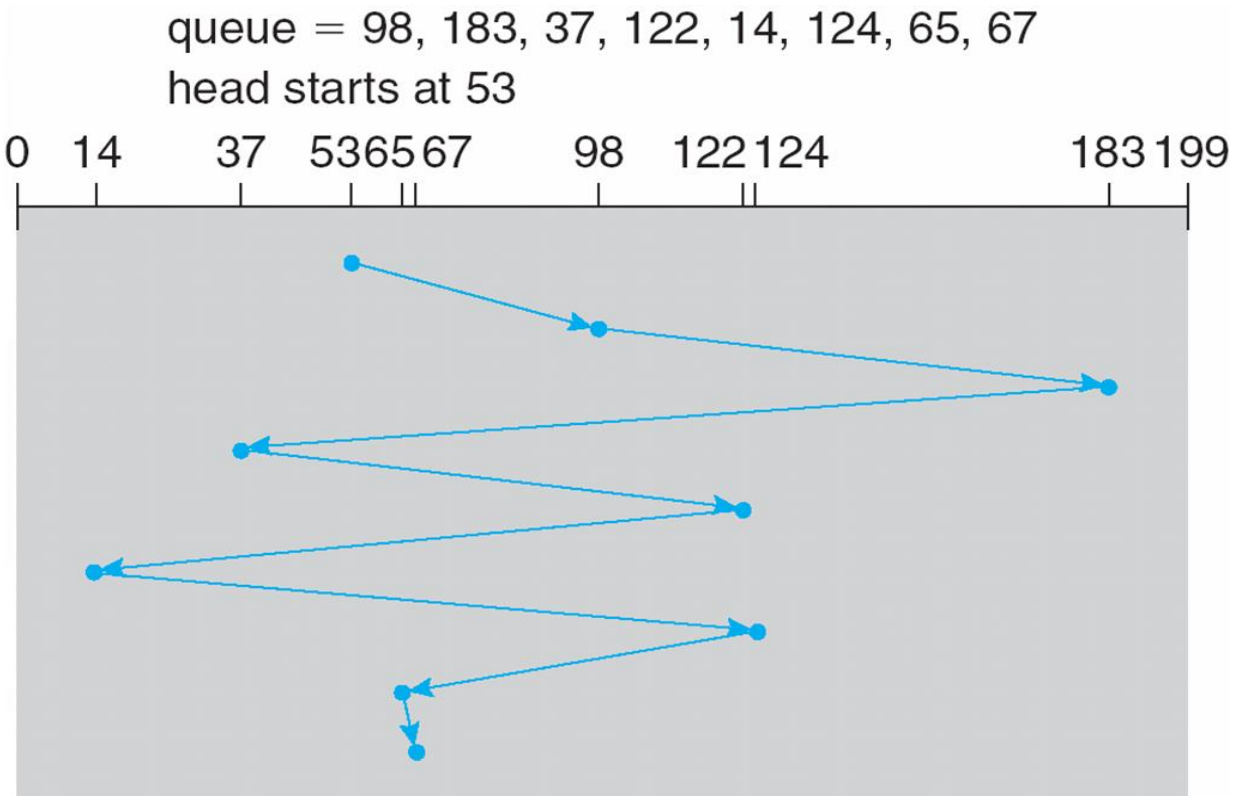
- There are many sources of disk I/O request
  - OS
  - System processes
  - Users processes
- I/O request includes input or output mode, disk address, memory address, number of sectors to transfer
- OS maintains queue of requests, per disk or device
- Idle disk can immediately work on I/O request, busy disk means work must queue
  - Optimization algorithms only make sense when a queue exists
- Note that drive controllers have small buffers and can manage a queue of I/O requests (of varying "depth")
- Several algorithms exist to schedule the servicing of disk I/O requests
- The analysis is true for one or many platters
- We illustrate scheduling algorithms with a request queue (0-199)

98, 183, 37, 122, 14, 124, 65, 67

Head pointer 53

FCFS

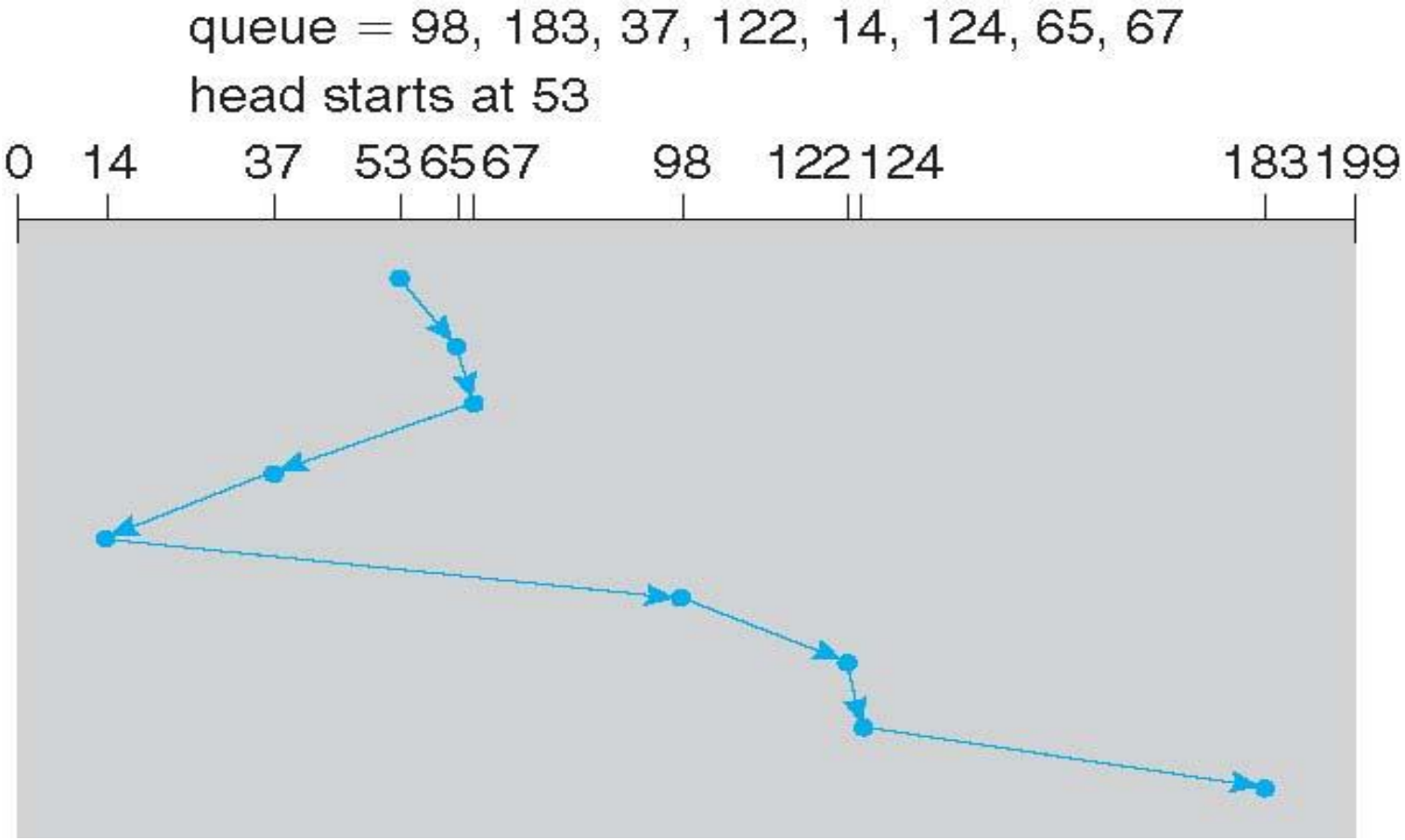
Illustration shows total head movement of 640 cylinders



SSTF

- Shortest Seek Time First selects the request with the minimum seek time from the current head position
- SSTF scheduling is a form of SJF scheduling; may cause starvation of some requests
- Illustration shows total head movement of 236 cylinders

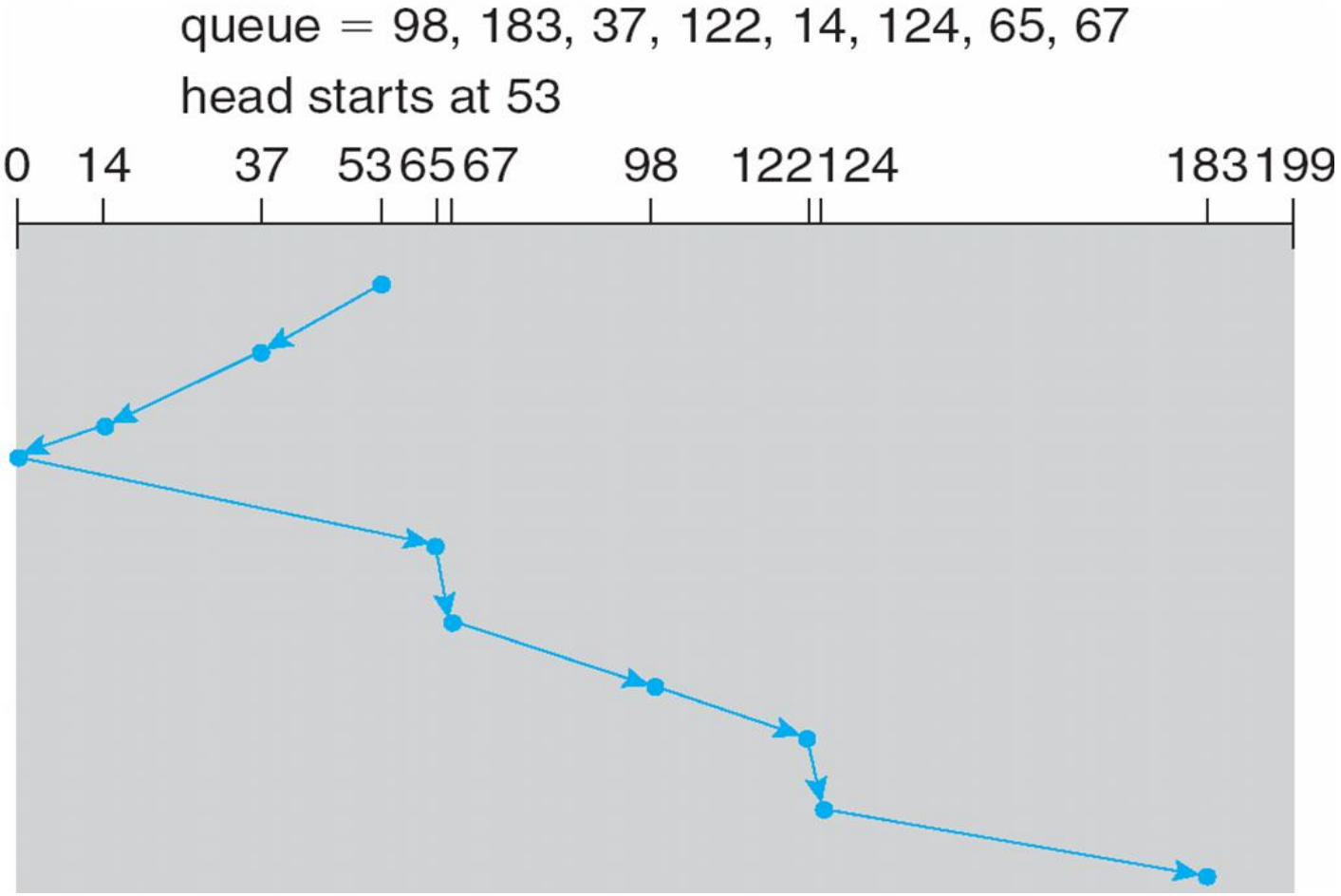
SSTF (Cont.)


queue = 98, 183, 37, 122, 14, 124, 65, 67
head starts at 53
0
14
37
53
65
67
98
122
124
183
199

SCAN

- The disk arm starts at one end of the disk, and moves toward the other end, servicing requests until it gets to the other end of the disk, where the head movement is reversed and servicing continues.
- **SCAN algorithm** Sometimes called the **elevator algorithm**
- Illustration shows total head movement of 208 cylinders
- But note that if requests are uniformly dense, largest density at other end of disk and those wait the longest

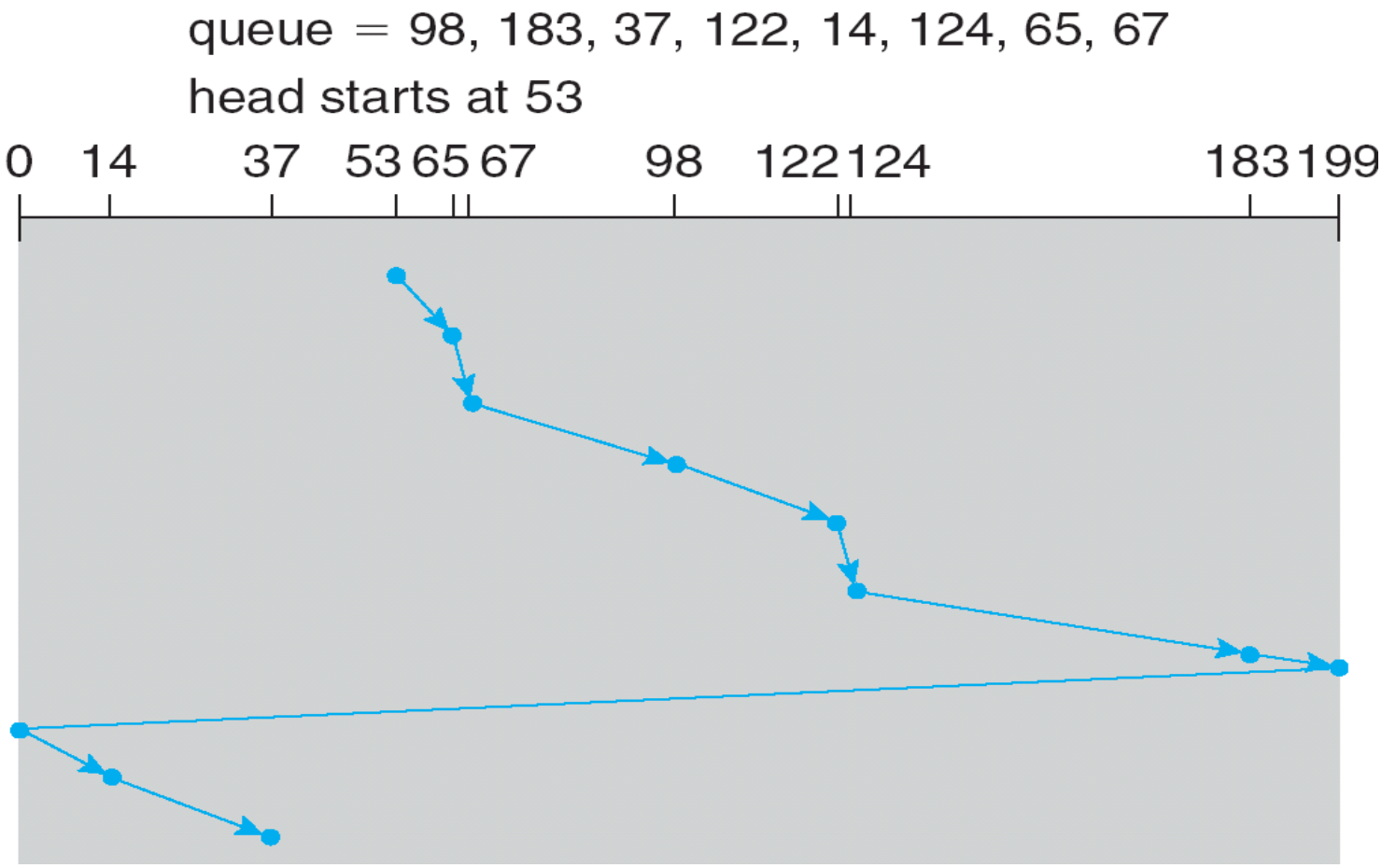
SCAN (Cont.)


queue = 98, 183, 37, 122, 14, 124, 65, 67
head starts at 53
0
14
37
53
65
67
98
122
124
183
199

C-SCAN

- Provides a more uniform wait time than SCAN
- The head moves from one end of the disk to the other, servicing requests as it goes
  - When it reaches the other end, however, it immediately returns to the beginning of the disk, without servicing any requests on the return trip
- Treats the cylinders as a circular list that wraps around from the last cylinder to the first one
- Total number of cylinders?

C-SCAN (Cont.)


queue = 98, 183, 37, 122, 14, 124, 65, 67
head starts at 53
0
14
37
53
65
67
98
122
124
183
199

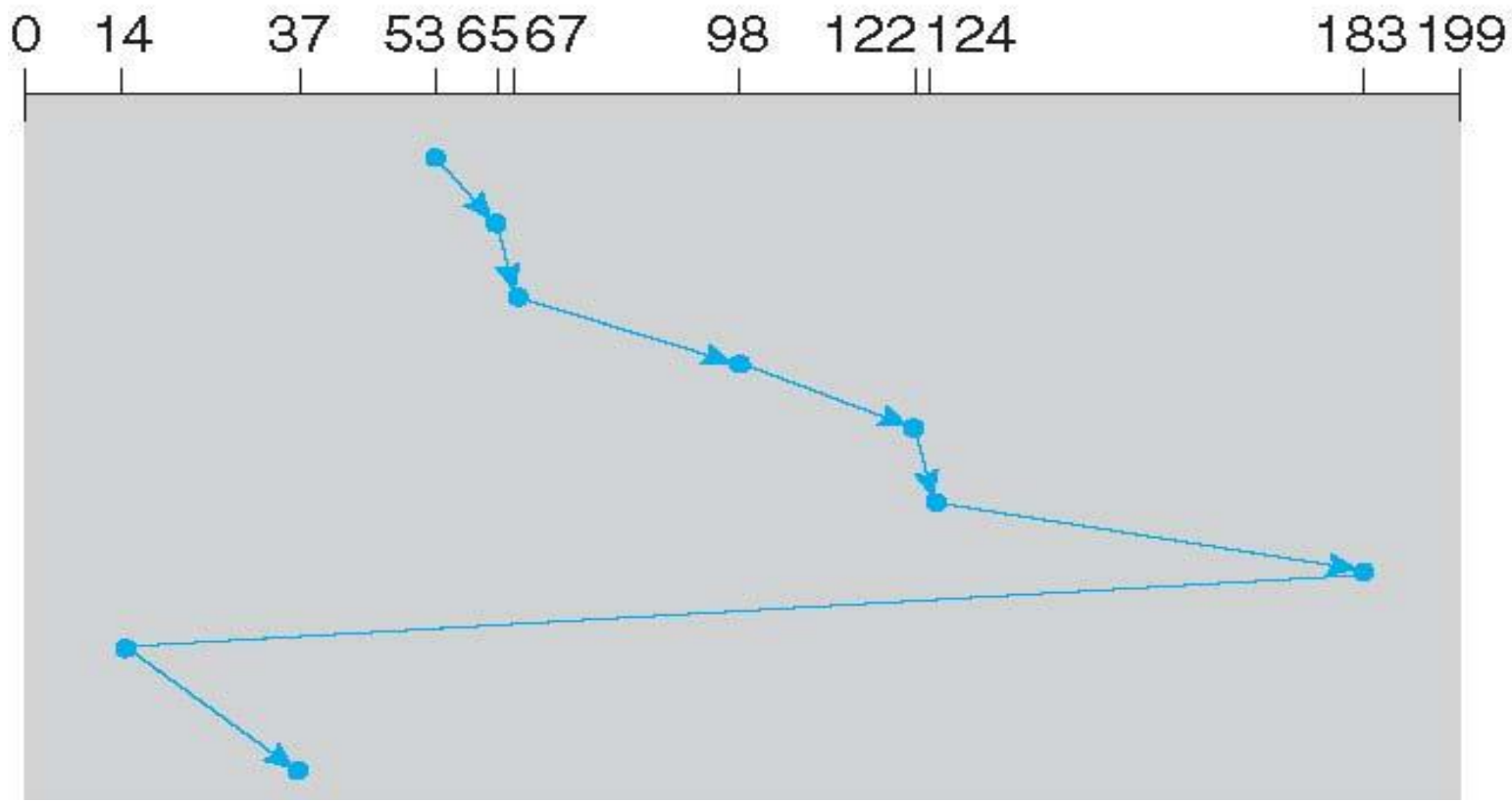
C-LOOK

- LOOK a version of SCAN, C-LOOK a version of C-SCAN
- Arm only goes as far as the last request in each direction, then reverses direction immediately, without first going all the way to the end of the disk
- Total number of cylinders?

C-LOOK (Cont.)

queue = 98, 183, 37, 122, 14, 124, 65, 67

head starts at 53



Selecting a Disk-Scheduling Algorithm

- SSTF is common and has a natural appeal
- SCAN and C-SCAN perform better for systems that place a heavy load on the disk
  - Less starvation
- Performance depends on the number and types of requests
- Requests for disk service can be influenced by the file-allocation method
  - And metadata layout
- The disk-scheduling algorithm should be written as a separate module of the operating system, allowing it to be replaced with a different algorithm if necessary
- Either SSTF or LOOK is a reasonable choice for the default algorithm

- What about rotational latency?
  - Difficult for OS to calculate

- How does disk-based queueing effect OS queue ordering efforts?

Disk management

Disk management of the operating system includes:

- Disk Format
- Booting from disk
- Bad block recovery

The low-level format or physical format:

- Divides the disk into sectors before storing data so that the disk controller can read and write Each sector can be:
- The header retains information, data, and error correction code (ECC) sectors of data, typically 512 bytes of data, but optional disks use the operating system's own data structures to preserve files using disks.

It is conducted in two stages:

1. Divide the disc into multiple cylinder groups. Each is treated as a logical disk.
2. Logical format or "Create File System". The OS stores the data structure of the first file system on the disk. Contains free space and allocated space.

- For efficiency, most file systems group blocks into clusters. Disk I / O runs in blocks. File I / O runs in a cluster.

Boot block:

- When the computer is turned on or restarted, the program stored in the initial bootstrap ROM finds the location of the OS kernel from the disk, loads the kernel into memory, and runs the OS. start.
- To change the bootstrap code, you need to change the ROM and hardware chip. Only a small bootstrap loader program is stored in ROM instead.
- The full bootstrap code is stored in the “boot block” of the disk.
- A disk with a boot partition is called a boot disk or system disk.

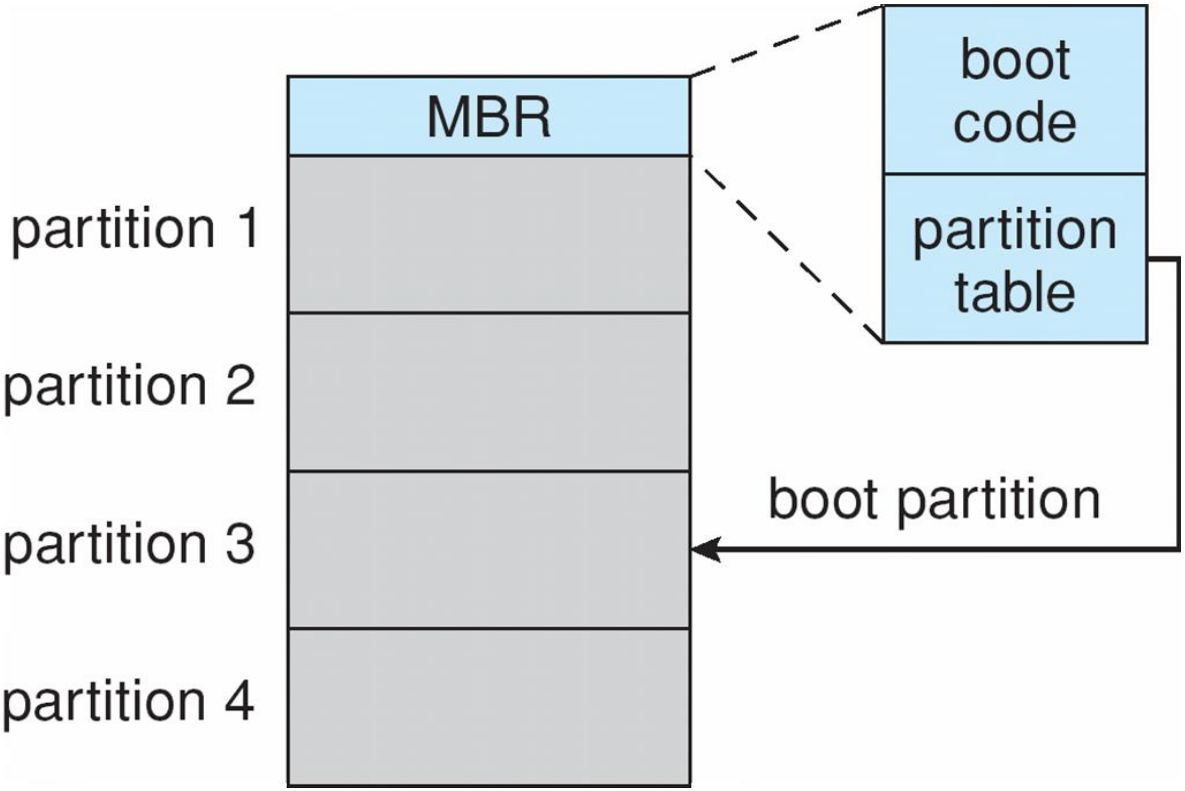
Bad Blocks:

- Disks are error-prone because moving parts have small tolerances.
- Most disks are even stuffed from the factory with bad blocks and are handled in a variety of ways.
- The controller maintains a list of bad blocks.
- The controller can instruct each bad sector to be logically replaced with one of the spare sectors. This scheme is known as sector sparing or transfer.
- A soft error triggers the data recovery process.
- However, unrecoverable hard errors may result in data loss and require manual intervention.

Disk Management

- **Low-level formatting**, or **physical formatting** — Dividing a disk into sectors that the disk controller can read and write
  - Each sector can hold header information, plus data, plus **Error Correction Code** (**ECC**)
  - Usually 512 bytes of data but can be selectable
- To use a disk to hold files, the operating system still needs to record its own data structures on the disk
  - **Partition** the disk into one or more groups of cylinders, each treated as a logical disk
  - **Logical formatting** or "making a file system"
  - To increase efficiency most file systems group blocks into **clusters**
    - Disk I/O done in blocks
    - File I/O done in clusters
- Raw disk access for apps that want to do their own block management, keep OS out of the way (databases for example)
- Boot block initializes system
  - The bootstrap is stored in ROM
  - **Bootstrap loader** program stored in boot blocks of boot partition
- Methods such as **sector sparing** used to handle bad blocks

Booting from a Disk in Windows



The **Master Boot Record** (MBR) is the information in the first sector of a hard disk or a removable drive. It identifies how and where the system's operating system (OS) is located in order to be booted (loaded) into the computer's main storage or random access memory (RAM).