# Section-II (Weightage – 70%, Minimum Teaching Hours -28)

**Memory Management – Concept of Fragmentation, Swapping, Paging, Segmentation.**

**Virtual memory-Demand Paging, Page replacement algorithm, Thrashing.**

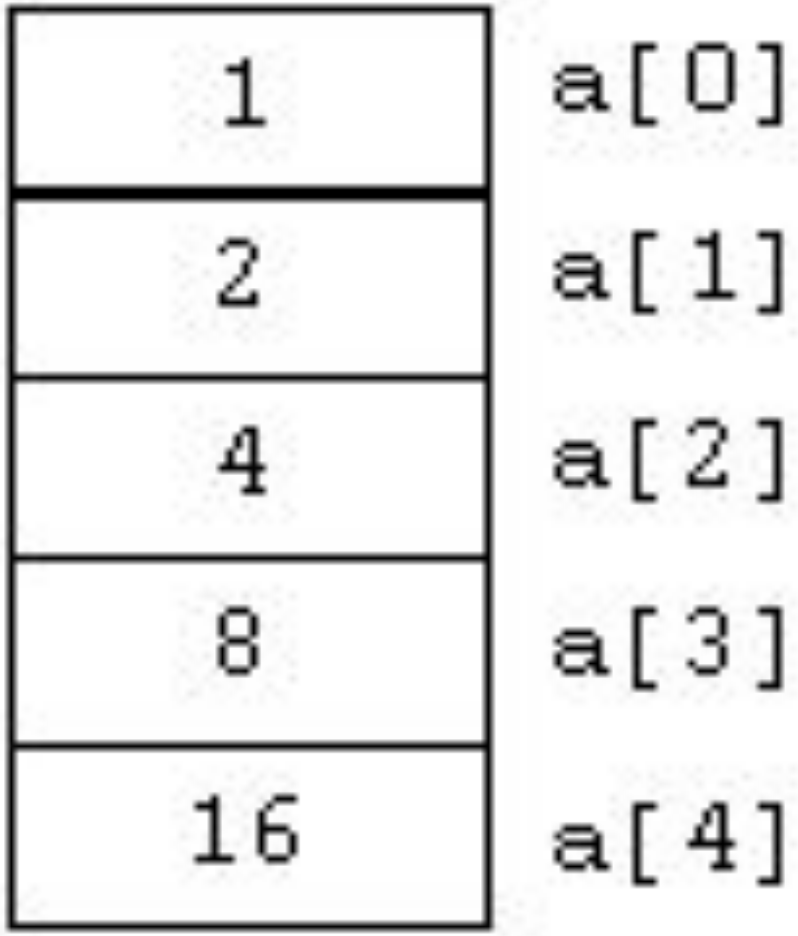
# Memory Management

- Background
- Swapping
- Contiguous Memory Allocation
- Paging
- Structure of the Page Table
- Segmentation

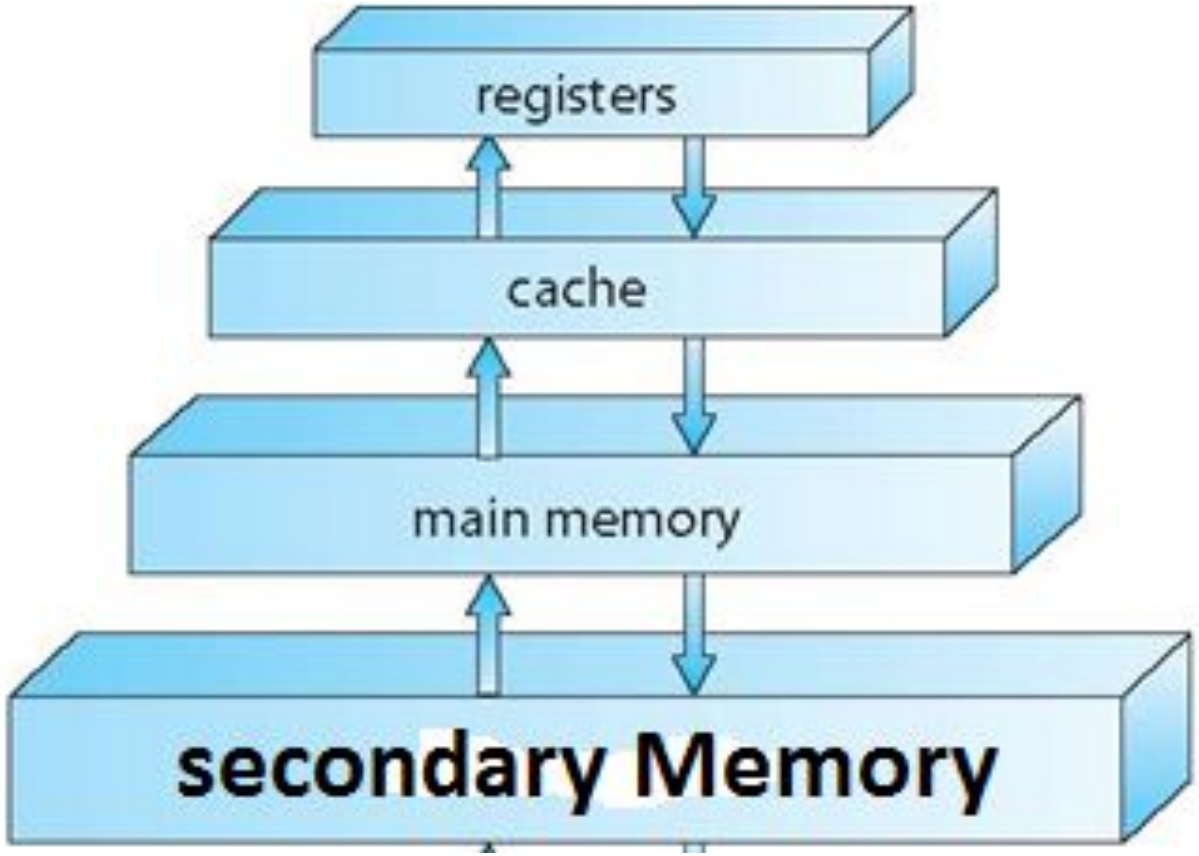
# Background

- Memory is an array of bytes, each with its own address.

- During execution of a process all the required programs and data must be available in memory.
- As available memory is small than required , memory management is main issue in multiprogramming environment.
- **Two concerns of memory management**
  - **Speed: Data movement through bus**
  - **Protection**
    - required to ensure correct operation
      - Protection of OS from user processes
      - Protection of one user process from another

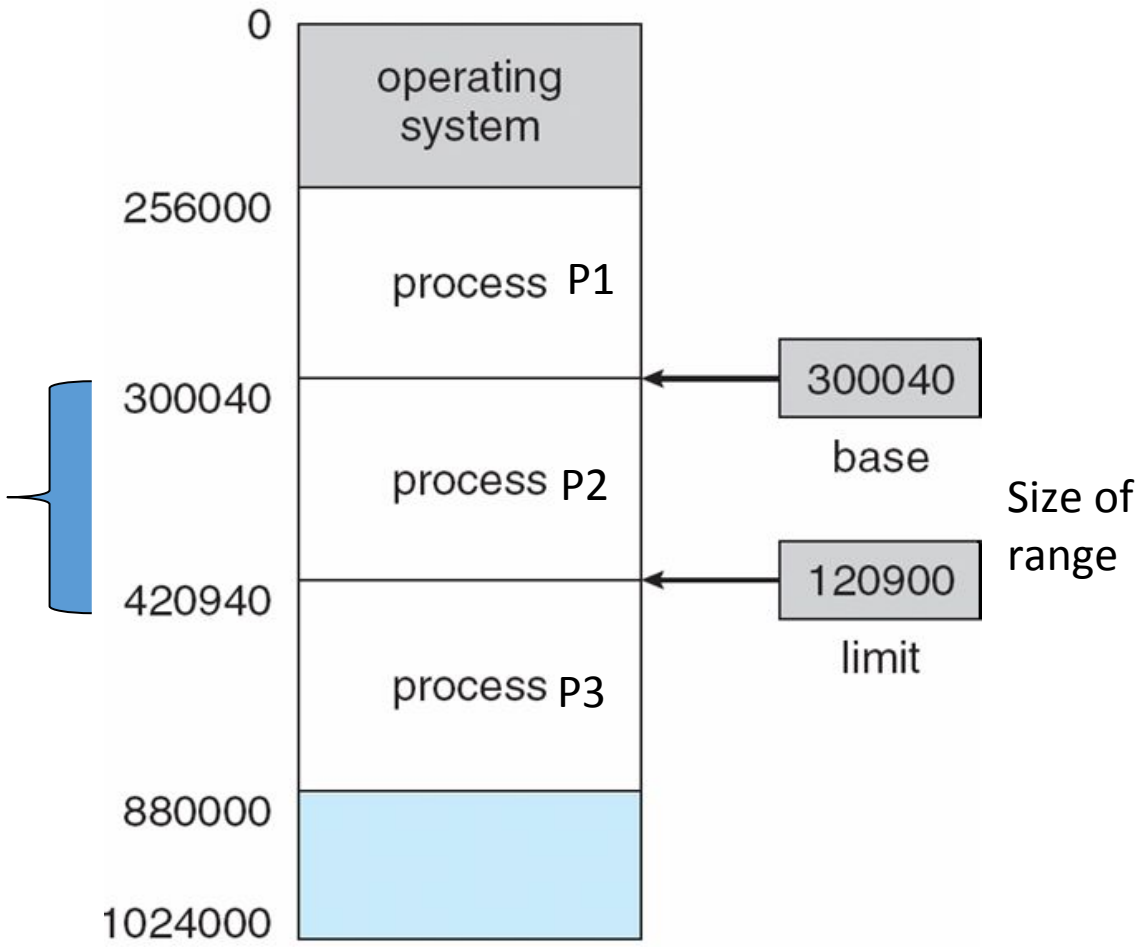


# Memory Hierarchy

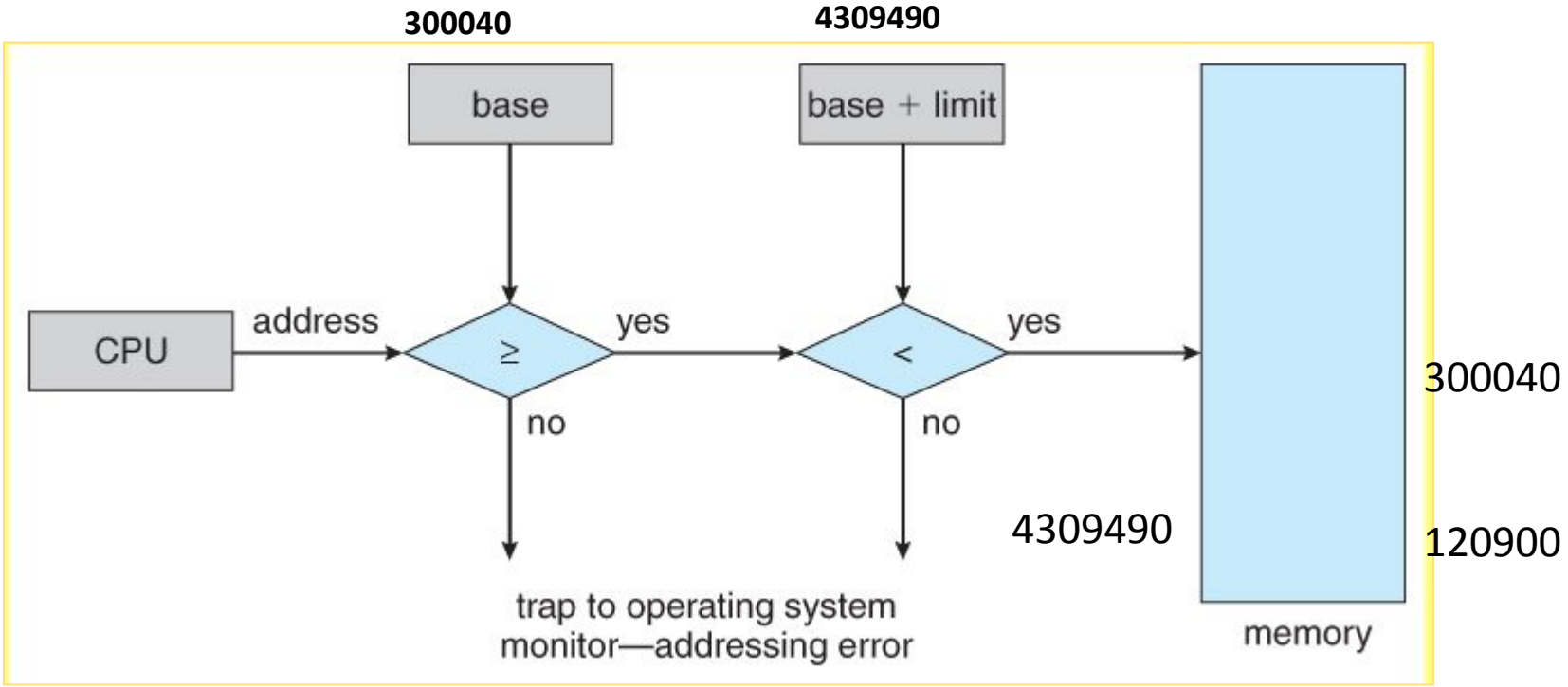


# Background

- Multiprogramming Environment
- A pair of **base** and **limit** registers define the logical address space



# Hardware address protection with base and limit registers

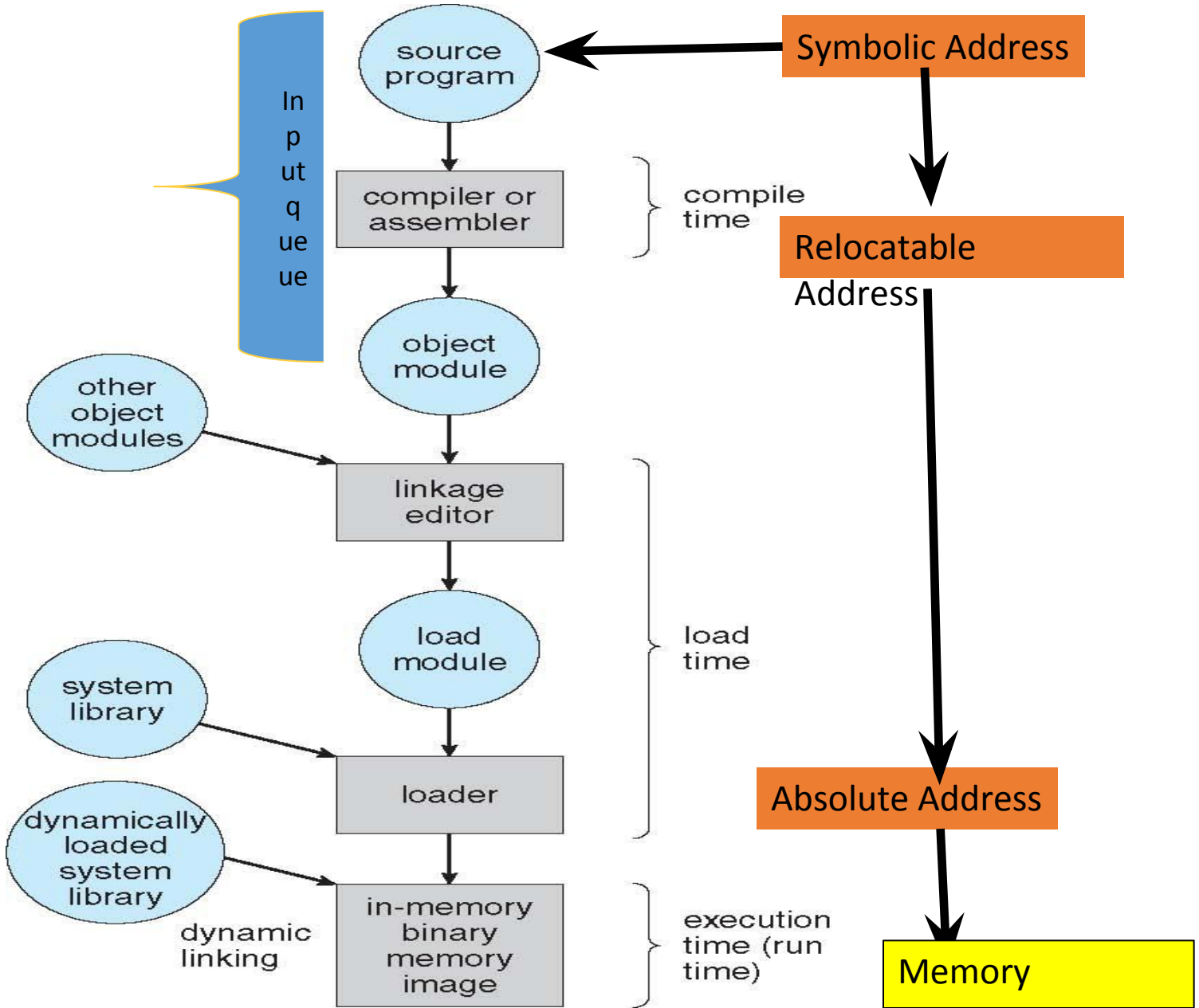



- ***Every*** memory access made by a user process is checked against these two registers, and if a memory access is attempted outside the valid range, then a fatal error is generated.
  - Kernel has access to OS memory & users memory OS loads program into users memory, log errors.

# Address Binding

- The process may be moved between disk and memory during its execution.
- 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.
- As the process is executed, it accesses instructions and data from memory.
- Most systems allow a user process to reside in any part of the physical memory.
- Thus, although the address space of the computer starts at 00000, the first address of the user process need not be 00000. This approach affects the addresses that the user program can use.
- In most cases, a user program will go through several steps-some of which may be optional-before being executed (Figure 8.3).
- Addresses in the source program are generally symbolic (such as count).
- A compiler will typically bind these symbolic addresses to relocatable addresses (such as "14 bytes from the beginning of this module").
- The linkage editor or loader will in turn bind the relocatable addresses to absolute addresses (such as 74014).
- Each binding is a mapping from one address space to another.

# Multistep Processing of a User Program



# Binding of Instructions and Data to Memory

- Address binding of instructions and data to memory addresses can happen at three different stages
  - Compile time: If memory location known a priori, absolute code can be generated;
  - If you know at compile time, where the process will reside in memory, then absolute code can be generated.
  - Load time: Compiler must generate relocatable code if memory location is not known at compile time. Binding is delayed until load time.
  - If it is not known at compile time where the process will reside in memory, then the compiler must generate relocatable code. Final binding is delayed until load time.
  - Execution time: Binding delayed until run time if the process can be moved during its execution from one memory segment to another. Need hardware support for address maps (e.g., base and limit registers)
  - If the process can be moved during its execution from one memory segment to another, then binding must be delayed until run time.

# Logical vs. Physical Address Space

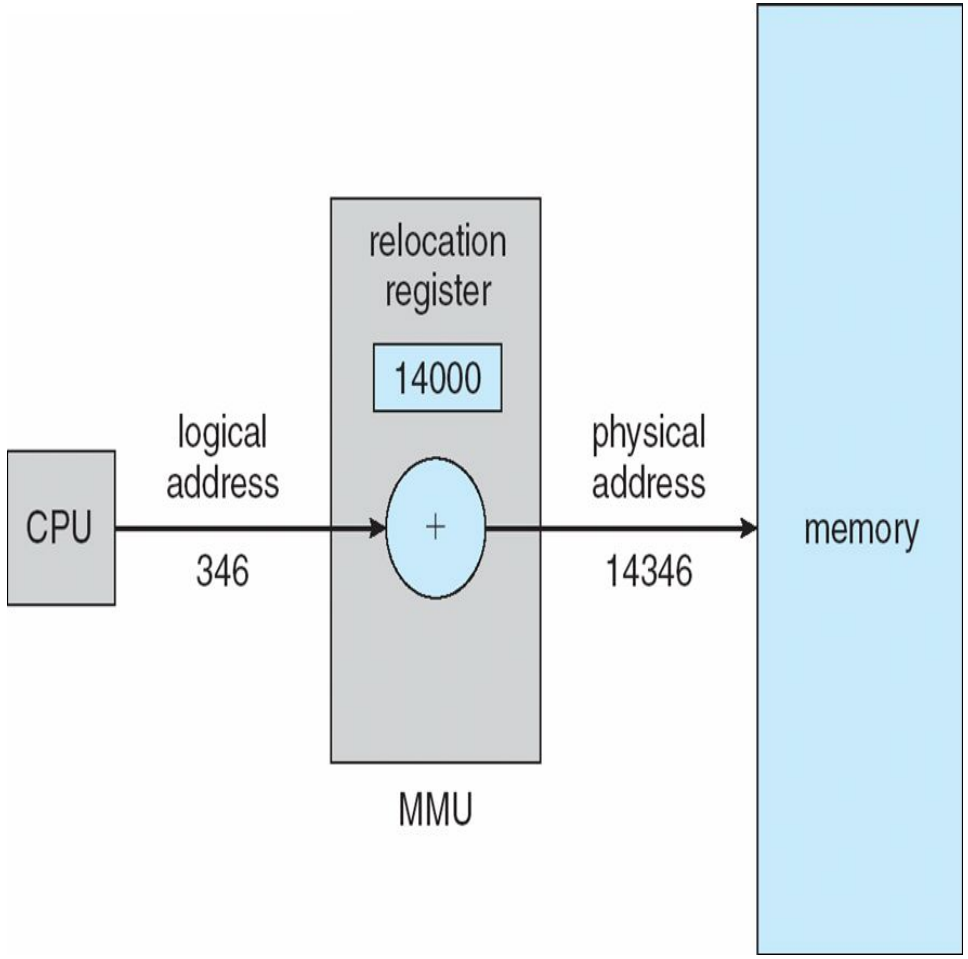
- The concept of a logical address space that is bound to a separate **physical address space** is central to proper memory management
  - **Logical address** – generated by the CPU; also referred to as **virtual address**
  - **Physical address** – address seen by the memory unit
- Logical and physical 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
- **Logical address space** is the set of all logical addresses generated by a program
- **Physical address space** is the set of all physical addresses generated by a program

# Memory-Management Unit (MMU)

- Hardware device that at run time maps virtual to physical address
- To start, consider simple scheme where the value in the relocation register is added to every address generated by a user process at the time it is sent to memory
  - Base register now called **relocation register**
- The user program deals with *logical* addresses; it never sees the *real* physical addresses
  - Execution-time binding occurs when reference is made to location in memory
  - Logical address bound to physical addresses

# Dynamic relocation using a relocation register

- Routine is not loaded until it is called
- Better memory-space utilization; unused routine is never loaded
- All routines kept on disk in relocatable load format
- Useful when large amounts of code are needed to handle infrequently occurring cases
- No special support from the operating system is required
  - Implemented through program design
  - OS can help by providing libraries to implement dynamic loading



# Dynamic loading: Background

- Program consist of structures like procedures, functions, error routines etc.
- Normal practice is to load complete program n the memory.
- But all these structures are not needed at same time.
- Wastage of memory space.
- **Required routines are kept in relocatable load format.**
- **Only main program is loaded in memory.**
- **When routine is called, it is initially searched in the memory. If not available, that routine is loaded in memory.**
- Routine is **not loaded until it is called**
- User should know require system library files

# Dynamic loading (Advantages)

- Better memory-space utilization; unused routine is never loaded
- Useful when large amounts of code are needed to handle infrequently occurring cases
- No special support from the operating system is required .
- Implemented through program design

# Swapping: Background

- In **contiguous allocation** while executing a process, **entire process** must be available in memory.

- But we want to execute **multiple processes**, and if **no sufficient memory** is available to load it in memory, swapping can be used.

# Swapping:

- A process must be in memory to be executed. A process, however, can be swapped temporarily out of memory to a backing store and then brought back into memory for continued execution.
- For example, assume a multiprogramming environment with a round-robin CPU-scheduling algorithm.
- When a quantum expires, the memory manager will start to swap out the process that just finished and to swap another process into the memory space that has been freed (Figure 8.5).
- In the meantime, the CPU scheduler will allocate a time slice to some other process in memory.
- When each process finishes its quantum, it will be swapped with another process.
- Ideally, the memory manager can swap processes fast enough that some processes will be in memory, ready to execute, when the CPU scheduler wants to reschedule the CPU.
- In addition, the quantum must be large enough to allow reasonable amounts of computing to be done between swaps.

# Schematic View of Swapping

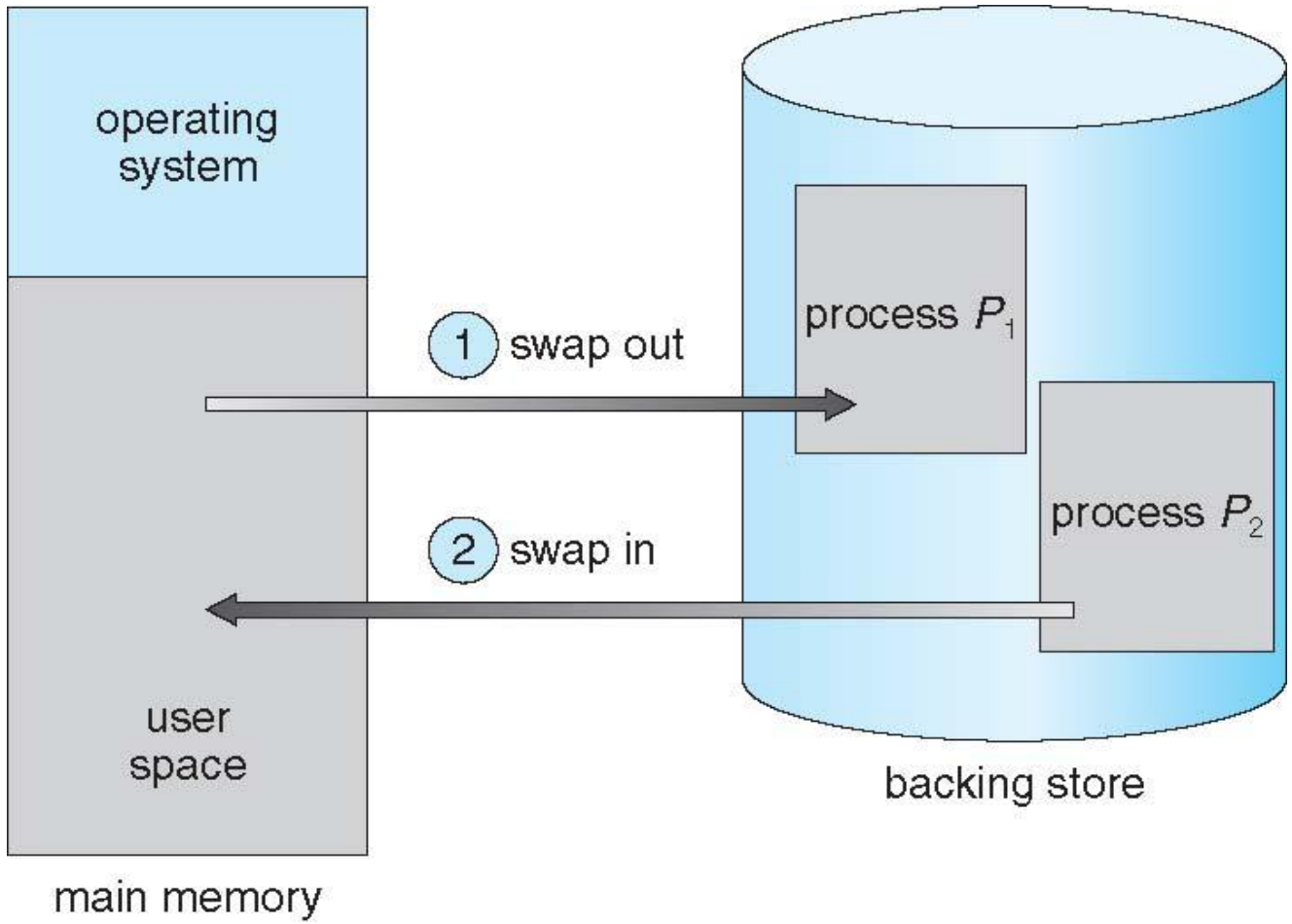



fig 8.5 e.g. Round Robin scheduling, priority scheduling
Location of swapped process: same/different Ready queue

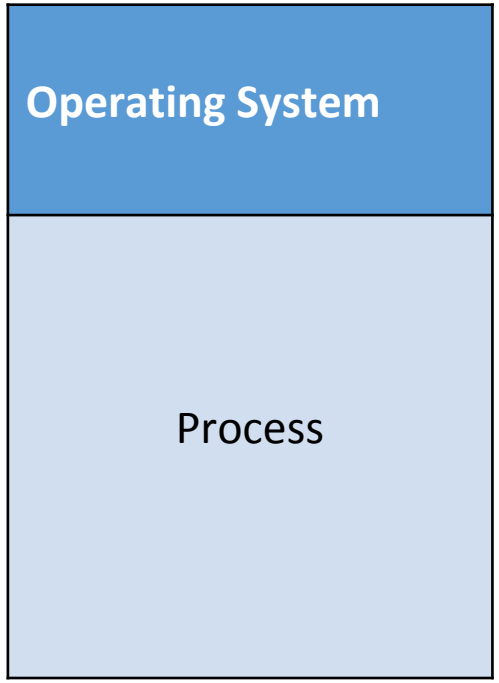
# Swapping:

- A variant of this swapping policy is used for priority-based scheduling algorithms. If a higher-priority process arrives and wants service, the memory manager can swap out the lower-priority process and then load and execute the higher-priority process.
- When the higher-priority process finishes, the lower-priority process can be swapped back in and continued. This variant of swapping is sometimes called roll out,roll in.
- Major part of swap time is transfer time; total transfer time is directly proportional to the amount of memory swapped
- System maintains a **ready queue** of ready-to-run processes which have memory images on disk
- Does the swapped out process need to swap back in to same physical addresses?
- Depends on address binding method
  - Plus consider pending I/O to / from process memory space
- Modified versions of swapping are found on many systems (i.e., UNIX, Linux, and Windows)
  - Swapping normally disabled
  - Started if more than threshold amount of memory allocated
  - Disabled again once memory demand reduced below threshold

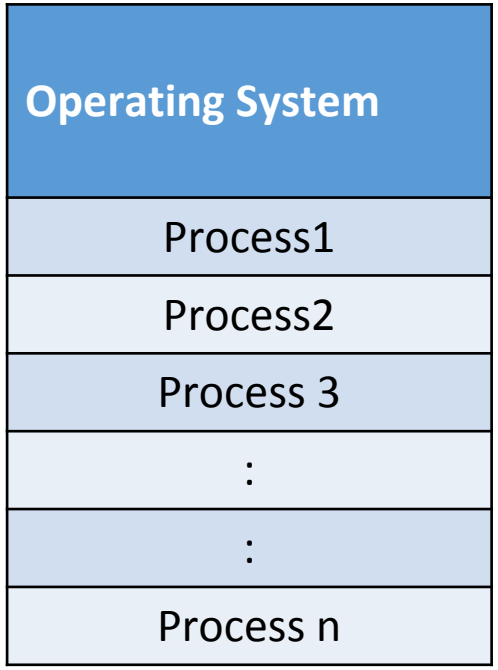
# Swapping Issues

- Context switch **overhead**
- **More time for swapping and Less execution**
- Process must be **idle (even i/o operation will not be allowed)**
- **Seek time can be reduced using swap space.**

# Contiguous Memory Allocation

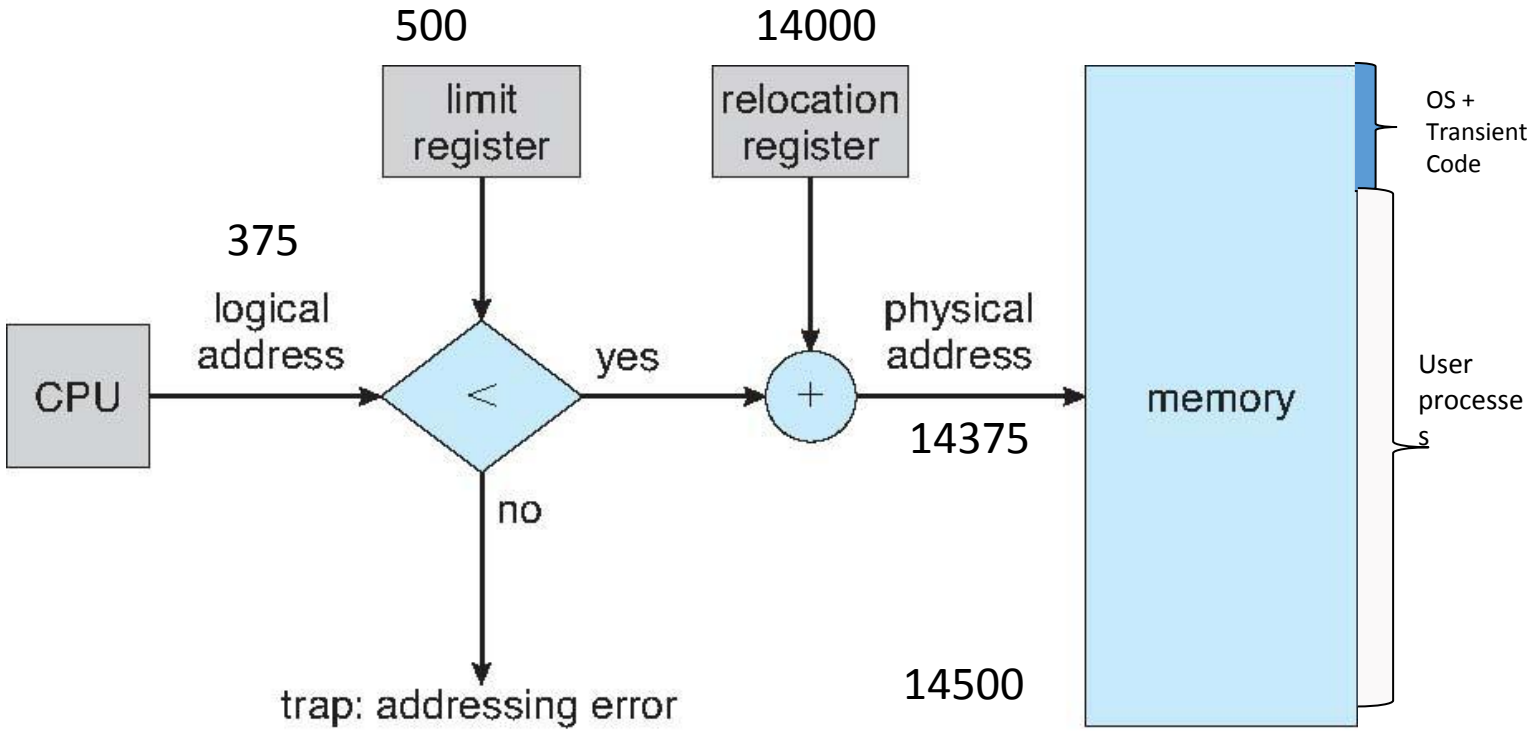



Main Memory




Contiguous memory allocation
Each process is in single contiguous section

# Memory Mapping & Protection




Hardware Support for Relocation and Limit Registers

**Relocation Register:**

Protection

Dynamic address binding

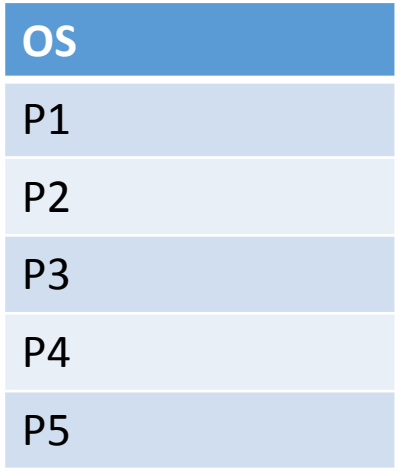
Change OS space dynamically.

# Contiguous Allocation

- Main memory usually into two partitions:
  - Resident operating system, usually held in low memory with interrupt vector
  - User processes that held in high memory
- **Relocation registers** used to **protect** user processes from each other, and from **changing** operating-system code and data
  - Base register contains value of smallest physical address
  - Limit register contains range of logical addresses – each logical address must be less than the limit register
  - MMU maps logical address *dynamically*

# Contiguous Allocation

- Two Partitioning Methodologies:
  - Fixed size partition
  - Variable size partition

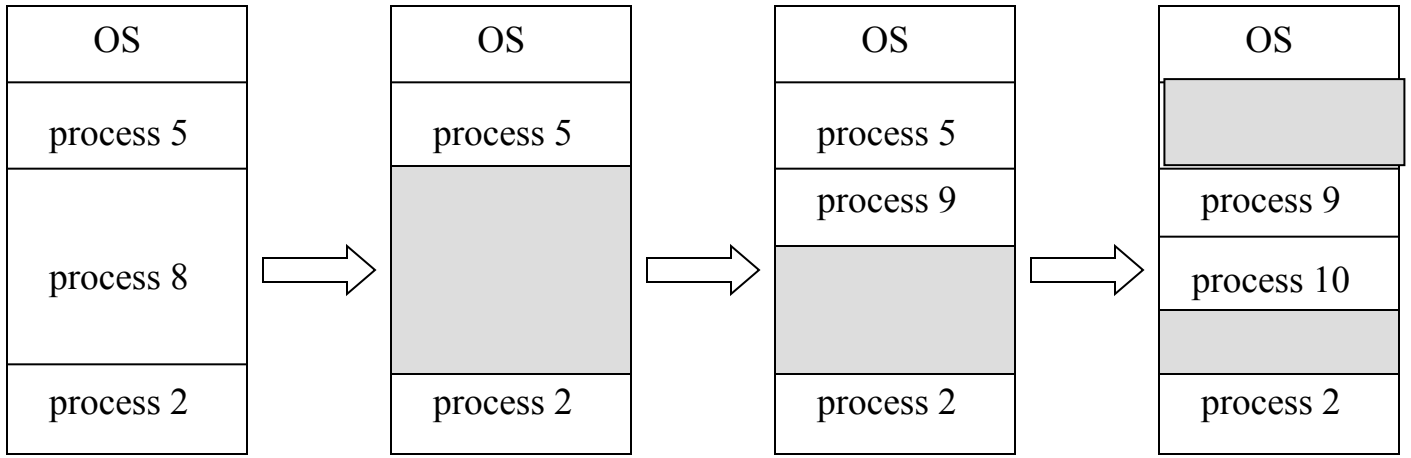



Fixed Size Partition


Variable Size Partition

# Contiguous Allocation (Cont.)

- Multiple-partition allocation
  - 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)**



# Dynamic Storage-Allocation Problem

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

- **First-fit**: Allocate the *first* hole that is big enough
- **Best-fit**: Allocate the *smallest* hole that is big enough; must search entire list, unless ordered by size
  - Produces the smallest leftover hole
- **Worst-fit**: Allocate the *largest* hole;
  - must also search entire list
  - Produces the largest leftover hole

First-fit and best-fit better than worst-fit in terms of speed and storage utilization (according to simulations)

# Solution

212 KB, 417 KB, 112 KB and 426 KB

Given memory partitions of 100 KB, 500 KB, 200 KB, 300 KB and 600 KB (in order), how would each of the first-fit, best-fit and worst-fit algorithms place processes of 212 KB, 417 KB, 112 KB and 426 KB (in that order) ? Which algorithm makes the most efficient use of memory?

| 100 | |
|---|---|
| 500 | 500-212=288K |
| 200 | |
| 300 | |
| 600 | 600-417=388 |

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

Best-Fit:
212K is put in 300K partition.
417K is put in 500K partition.
112K is put in 200K partition.
426K is put in 600K partition.

Worst-Fit:
212K is put in 600K partition.
417K is put in 500K partition.
112K is put in 388K partition.
426K must wait.

- In this example, Best-Fit turns out to be the best.

# Contiguous Allocation Drawbacks Fragmentation

- **External Fragmentation** – total memory space exists to satisfy a request, but it is not contiguous
- **Internal Fragmentation** – allocated memory may be slightly larger than requested memory; this size difference is memory internal to a partition, but not being used
- Reduce external fragmentation by
  - **non contiguous allocation / compaction**

**compaction**

- Shuffle memory contents to place all free memory together in one large block
- Compaction is possible *only* if relocation is dynamic, and is done at execution time

# Paging

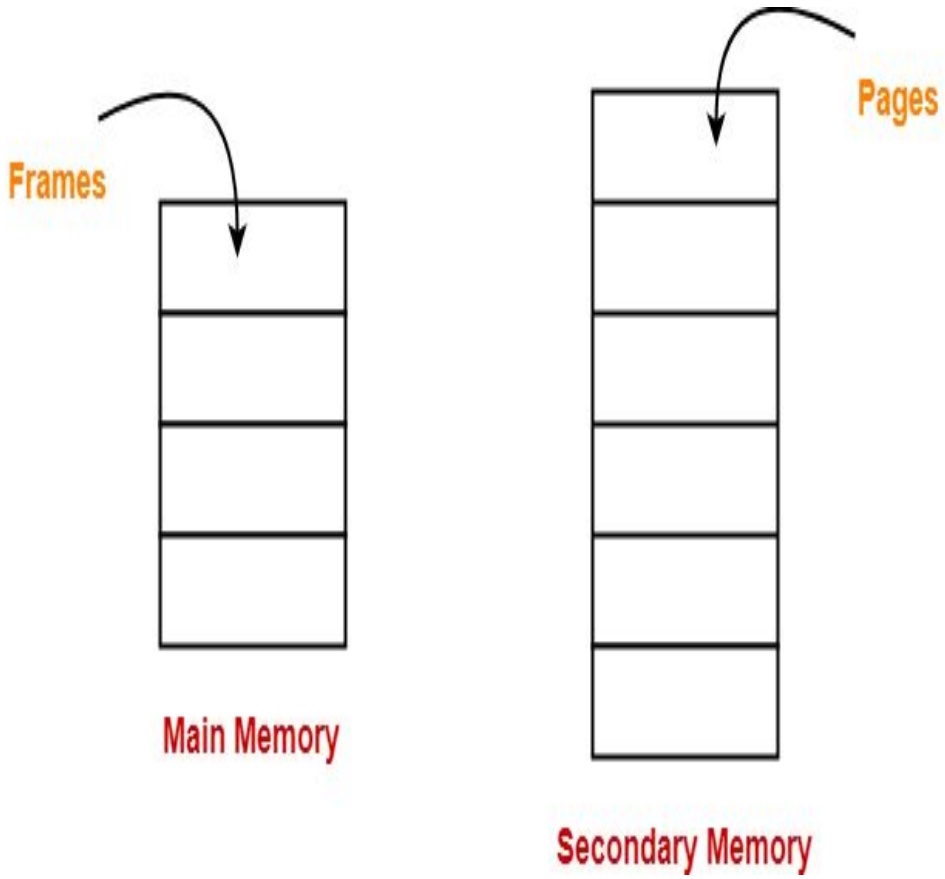
- Non contiguous allocation technique.
- Reduces external fragmentation.
- Logical Address or Virtual Address (represented in bits): An address generated by the CPU
- Logical Address Space or Virtual Address Space( represented in words or bytes): The set of all logical addresses generated by a program
- Physical Address (represented in bits): An address actually available on memory unit
- Physical Address Space (represented in words or bytes): The set of all physical addresses corresponding to the logical addresses

# Paging

- In Operating Systems, Paging is a storage mechanism used to retrieve processes from the secondary storage into the main memory in the form of pages.
- The main idea behind the paging is to divide each process in the form of pages. The main memory will also be divided in the form of frames.
- One page of the process is to be stored in one of the frames of the memory. The pages can be stored at the different locations of the memory but the priority is always to find the contiguous frames or holes.
- Pages of the process are brought into the main memory only when they are required otherwise they reside in the secondary storage.
- Paging is a memory-management scheme that permits the physical address space of a process to be non contiguous.
- Paging eliminates the need for contiguous allocation of physical memory.
- Paging avoids external fragmentation and the need for compaction.

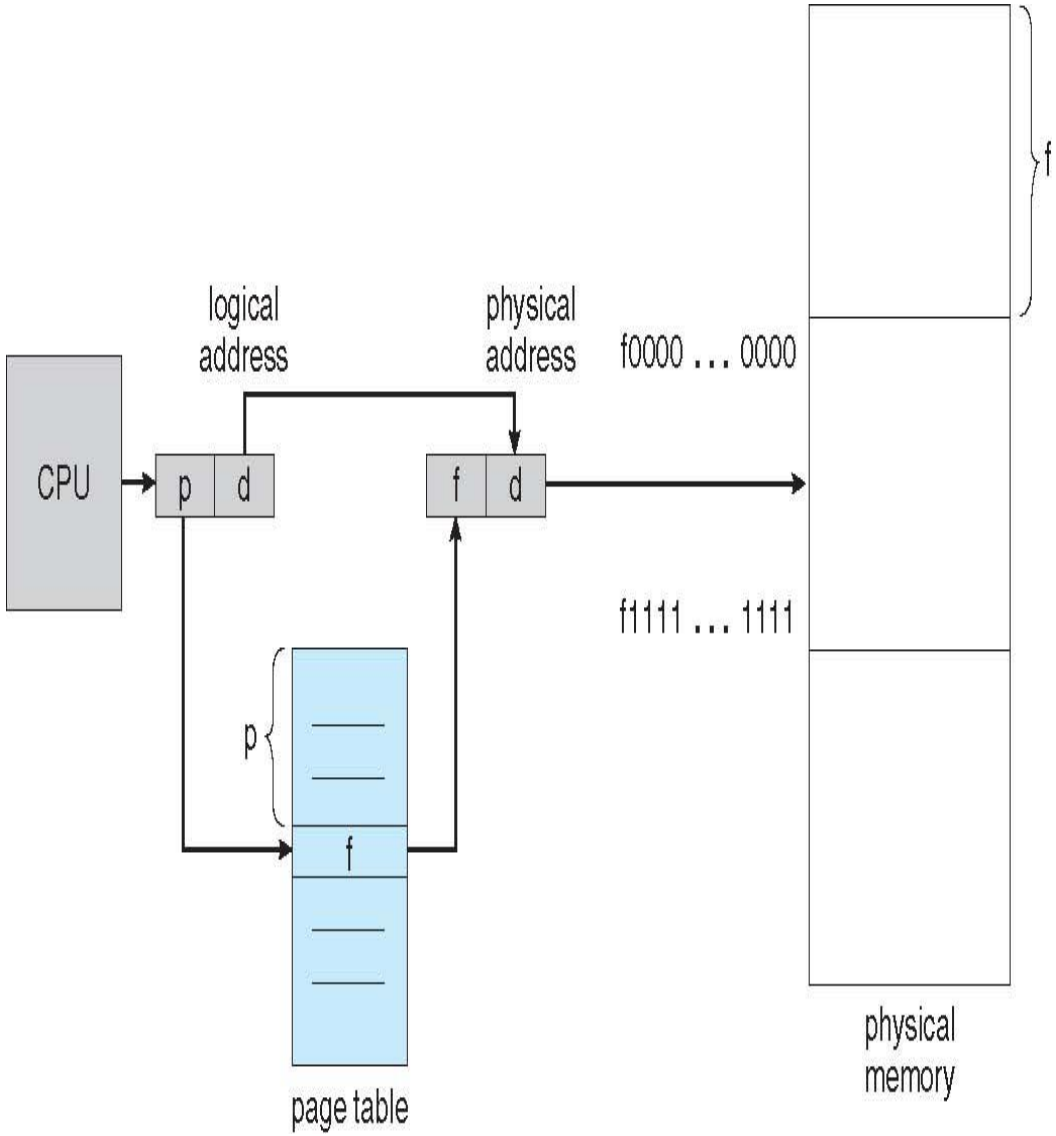
# Paging

- It also solves the considerable problem of fitting memory chunks of varying sizes onto the backing store;
- The basic method for implementing paging involves:
- Breaking physical memory into fixed-sized blocks called frames
- 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 (a file system or the backing store).
- The backing store is divided into fixed-sized blocks that are of the same size as the memory frames.



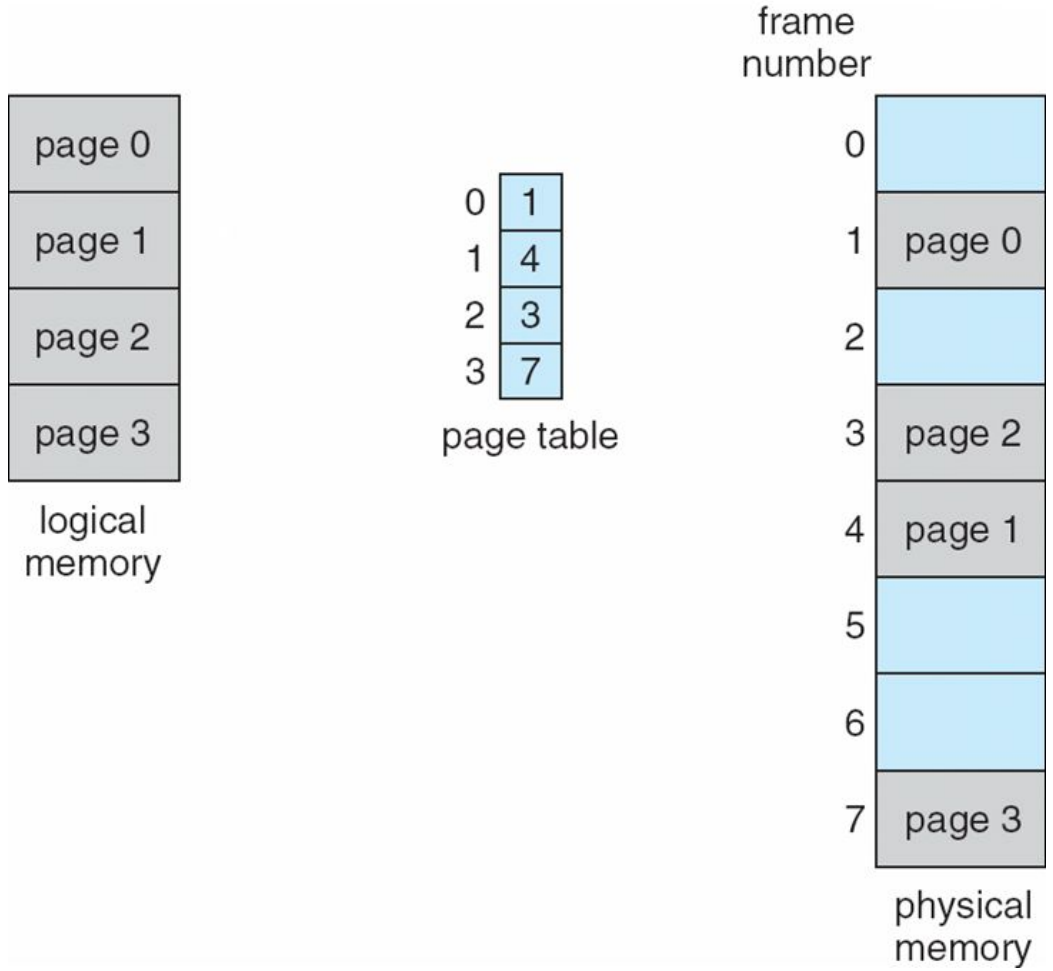
# Paging

- The hardware support for paging is illustrated in Figure 8.7.
- Every address generated the CPU is divided into two parts:
- A page number {p) and a page offset (d).
- The page number is used as an index into a page table.
- 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.



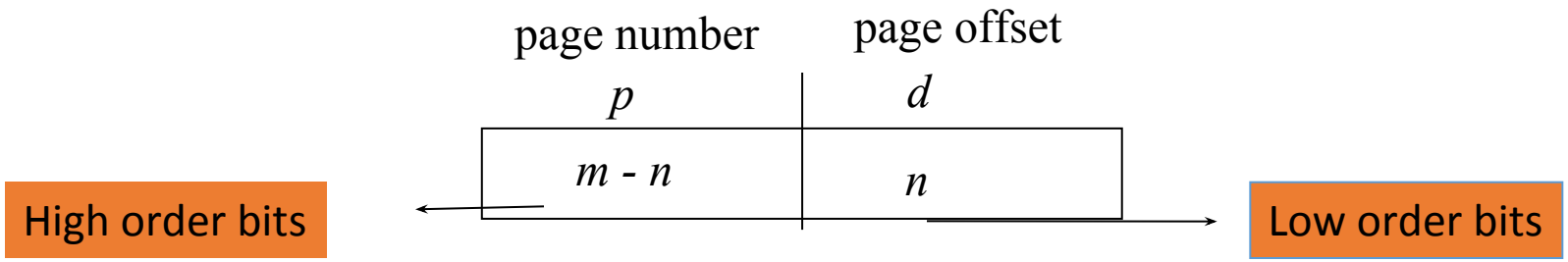
# Paging

- The paging model of memory is shown in Figure 8.8.
- The page size (like the frame size) is defined by the hardware.
- 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.
- The selection of a power of 2 as a page size makes the translation of a logical address into a page number and page offset particularly easy.
- If the size of the logical address space is $2^m$, and a page size is $2^n$ addressing units (bytes or words t then the high-order *m- n* bits of a logical address designate the page number, and the *n* low-order bits designate the page offset.

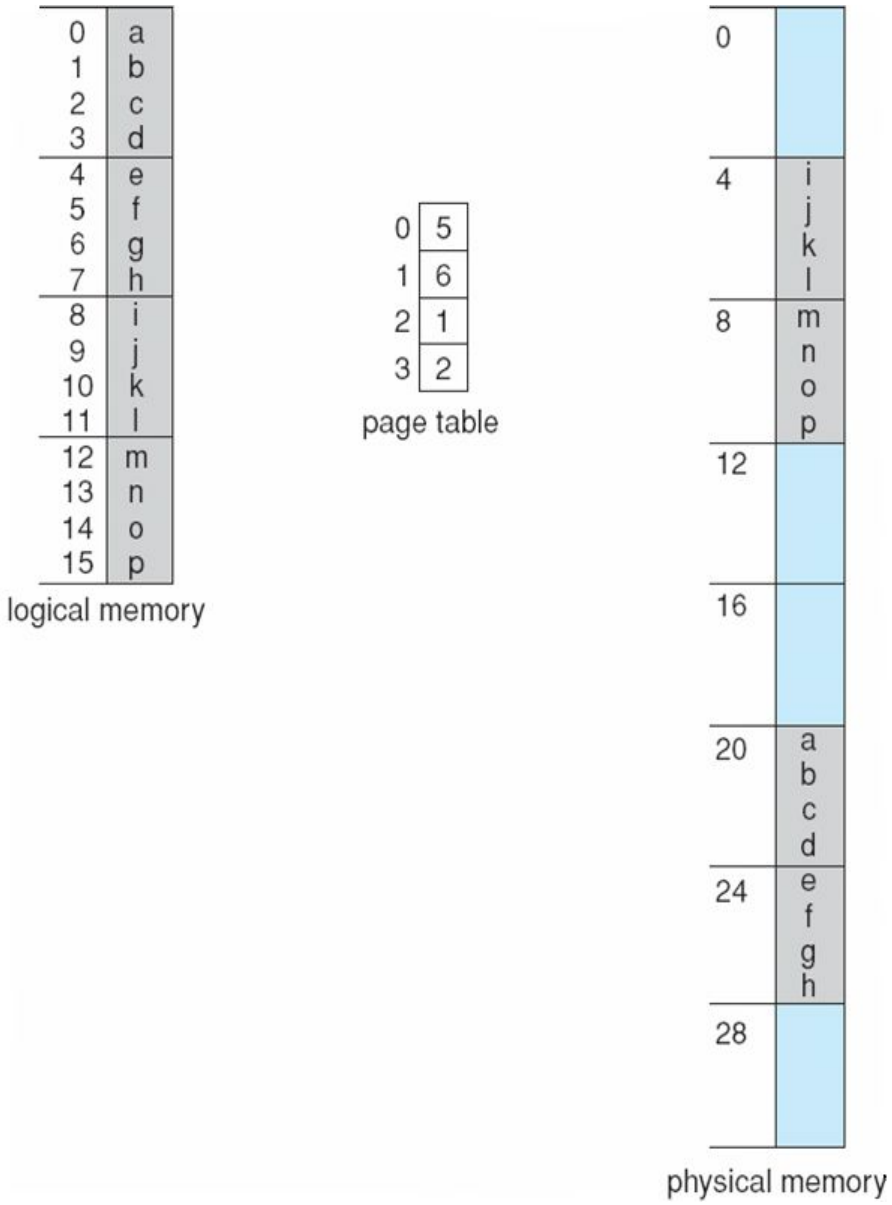


# Address Translation Scheme

- Address generated by CPU is divided into:
  - **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
  - **For given logical address space $2^m$ *and page size* $2^n$**
- Thus, the logical address is as follows:
- *p* is an index into the page table and *d* is the displacement within the page.
- As a concrete (although minuscule) example, consider the memory in Figure 8.9.



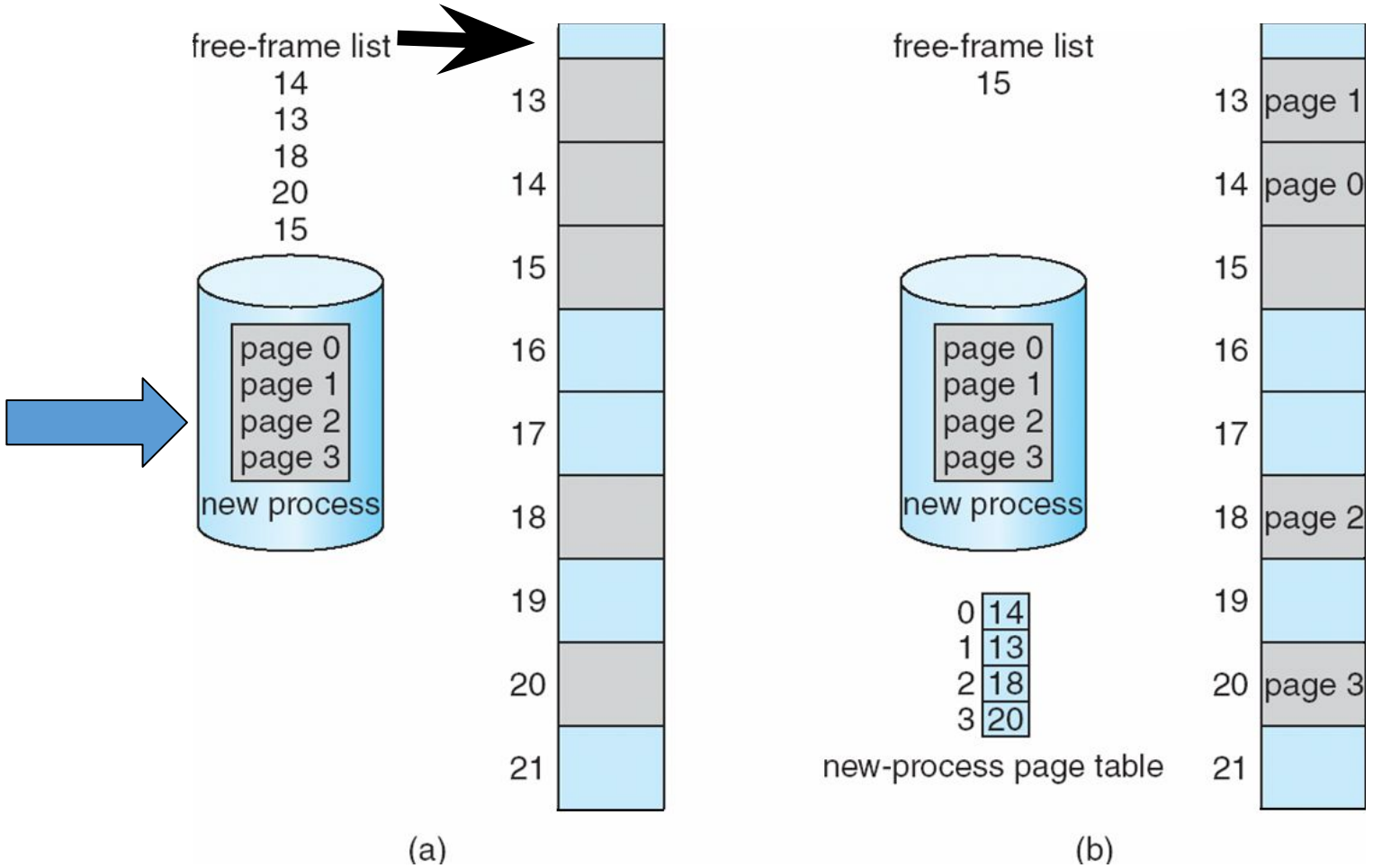
- Here, in the logical address, *n*= 2 and *m* = 4.
- Using a page size of 4 bytes and a physical memory of 32 bytes (8 pages), we show how the user's view of memory can be mapped into physical memory.
- Logical address 0 is page 0, offset 0. Indexing into the page table, we find that page 0 is in frame 5. Thus, logical address 0 maps to physical address 20 [= (5 x 4) + 0].
- Logical address 3 (page 0, offset 3) maps to physical address 23 [= (5 x 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 x 4) + O].
- Logical address 13 maps to physical address 9.
- You may have noticed that paging itself is a form of dynamic relocation.
- Every logical address is bound by the paging hardware to some physical address. Using paging is similar to using a table of base (or relocation) registers, one for each frame of memory.



- When we use a paging scheme, we have no external fragmentation: *any* free frame can be allocated to a process that needs it.
- However, we may have some internal fragmentation.
- Notice that frames are allocated as units. If the memory requirements of a process do not happen to coincide with page boundaries, the *last* frame allocated may not be completely full.
- For example, if page size is 2,048 bytes, a process of 72,766 bytes will need 35 pages plus 1,086 bytes.
- It will be allocated 36 frames, resulting in internal fragmentation of 2,048 - 1,086 = 962 bytes.
- In the worst case, a process would need *11* pages plus 1 byte. It would be allocated *11* + 1 frames, resulting in internal fragmentation of almost an entire frame.

# Free Frames

System maintains information of free frames
System identify frame requirement of each process




Before allocation

After allocation

## •Advantages of Paging

•Given below are some advantages of the Paging technique in the operating system:

•Paging mainly allows to storage of parts of a single process in a non-contiguous fashion.

•With the help of Paging, the problem of external fragmentation is solved.

•Paging is one of the simplest algorithms for memory management.

## •Disadvantages of Paging

•Disadvantages of the Paging technique are as follows:

•In Paging, sometimes the page table consumes more memory.

•Internal fragmentation is caused by this technique.

•There is an increase in time taken to fetch the instruction since now two memory accesses are required.

# Translation Look Aside Buffer

- h/w cache memory Translation look-aside buffer (TLB)
  - Fast lookup hardware cache memory

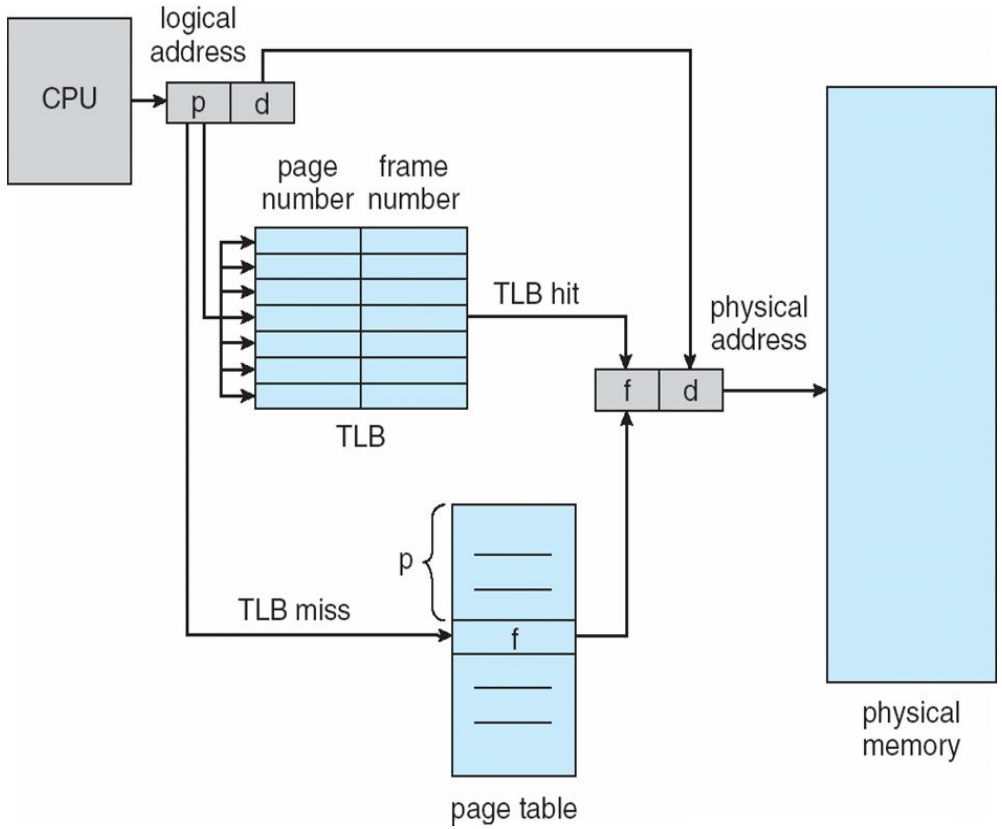
- TLB with Address Space Identifiers (ASID's)

# Translation Look Aside Buffer

- Each operating system has its own methods for storing page tables.
- 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. The problem with this approach is the time required to access a user memory location.
- If we want to access location i, we must first index into the page table, using the value in the PTBR offset by the page number for i. This task requires a memory access. It provides us with the frame number, which is combined with the page offset to produce the actual address.
- We can then access the desired place in memory. With this scheme, two memory accesses are needed to access a byte (one for the page-table entry, one for the byte). Thus, memory access is slowed by a factor of 2. This delay would be intolerable under most circumstances. We might as well resort to swapping.
- 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.

# Translation Look Aside Buffer

- Each entry in the TLB consists of two parts: a key (or tag) and a value.
- When the associative memory is presented with an item, the item is compared with all keys simultaneously.
- If the item is found, the corresponding value field is returned.
- The search is fast; the hardware, however, is expensive.
- Typically, the number of entries in a TLB is small, often numbering between 64 and 1,024.
- The TLB is used with page tables in the following way.
- The TLB contains only a few of the page-table entries.
- When a logical address is generated by the CPU, its page number is presented to the TLB.
- If the page number is found, its frame number is immediately available and is used to access memory.




Paging Hardware With TLB

# Translation Look Aside Buffer

- The whole task may take less than 10 percent longer than it would if an unmapped memory reference were used.
- If the page number is not in the TLB (known as a TLB miss), a memory reference to the page table must be made.
- When the frame number is obtained, we can use it to access memory (Figure 8.11).
- In addition, we add the page number and frame number to the TLB, so that they will be found quickly on the next reference. If the TLB is already full of entries, the operating system must select one for replacement. Replacement policies range from least recently used (LRU) to random.

- The percentage of times that a particular page number is found in the TLB is called the hit ratio.
- An 80-percent hit ratio 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 memory-access time, we weight each 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).

# Problem on EAT

- Consider a paging system with the page table stored in memory.
- 1. If a **memory reference takes 100 nanoseconds**, how long does a **paged memory reference** take?
- 2. If TLB **Hit Ratio is 80 percent,TLB search time is 20 nanoseconds,**
  - **a. find memory access time if page is in TLB.**
  - **b. find memory access time if page is not in TLB.**
  - **c. Find effective access time**
  - **d. Find EAT if TLB hit ratio 98%.**

HAt is the effective memory reference time. (Assume that finding a **page-table entry in the TLBs takes 2 nanoseconds**, if the entry is present.)

# EAT

| If page found in TLB | | | If page no. not found in TLB | |
|---|---|---|---|---|
| TLB Hit ratio | 80% = .80 | | TLB miss ratio | 20% = .20 |
| TLB search | 20 ns | | TLB search | 20 ns |
| Memory reference | 100 ns | | Page Table search | 100 ns |
| | | | Memory reference | 100 ns |
| Total Memory Access | 20+100 =120 ns | | Total Memory Access | 20+100+100 = 220 ns |
| EAT | (.80 * 120) + (1 - .80 )*220= 140 ns | | | |
| If 98% hit ratio EAT | .98 * 120 + .02 X 220 = 122 ns | | | |

# Implementation of Memory Protection in Paging

- Using protection bits with each frame
- Protection bits are kept in 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 nuncber.
  - At the same time that the physical address is being computed, the protection bits can be checked to verify that no writes are being made to a read-only page.
  - An attempt to write to a read-only page causes a hardware trap to the operating system (or memory-protection violation).

- **Valid-invalid** bit attached to each entry in the page table:
  - "valid" indicates that the associated page is in the process' logical address space, and is thus a legal page
  - "invalid" indicates that the page is not in the process' logical address space (Illegal Page)

# Valid (v) or Invalid (i) Bit In A Page Table

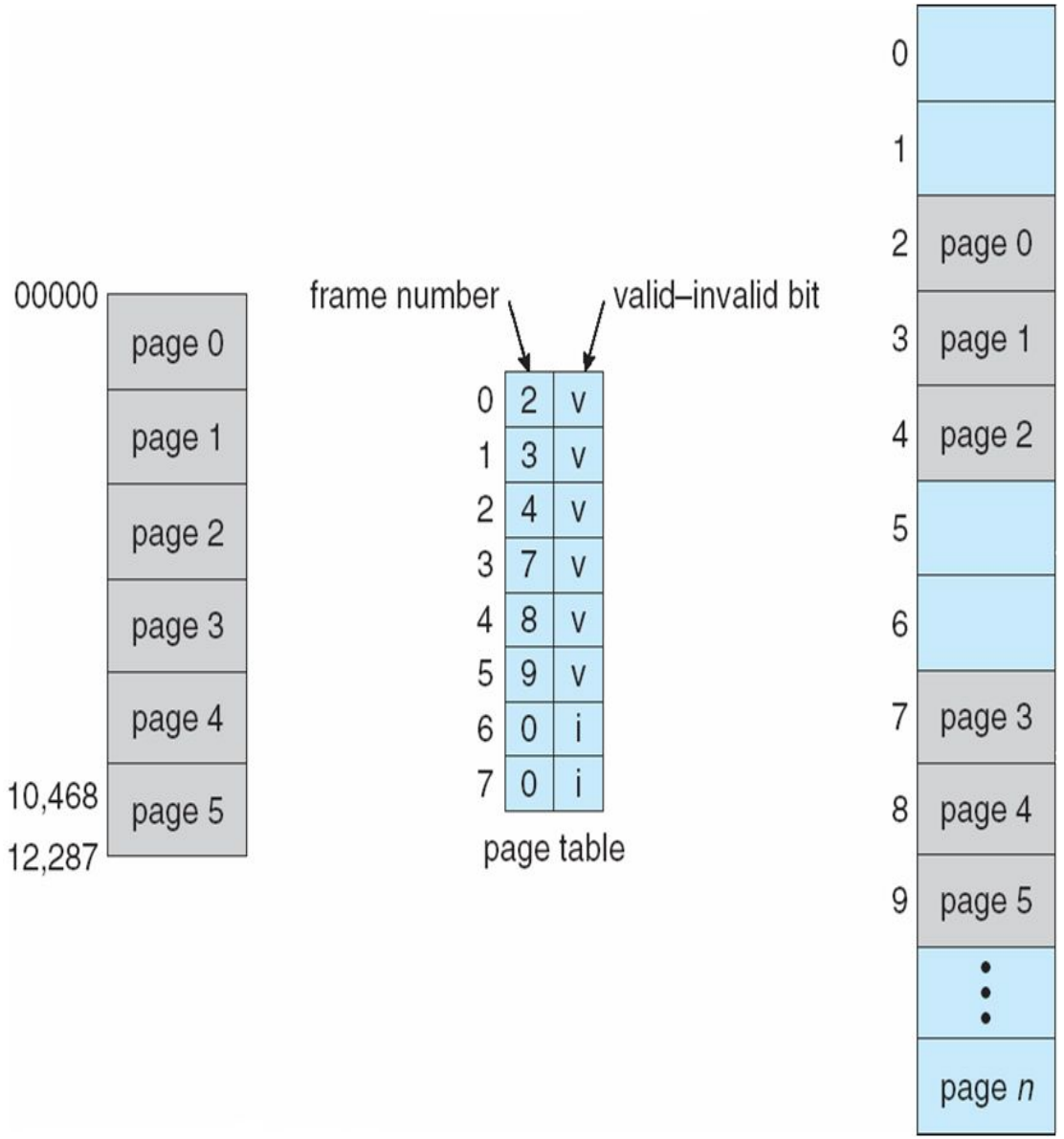
Suppose, for example, that in a system with a 14-bit address space (0 to 16383), we have a program that should use only addresses 0 to 10468.

Given a page size of 2 KB, we have the situation shown in Figure 8.12. Addresses in pages 0, 1, 2, 3, 4, and 5 are mapped normally through the page table. Any attempt to generate an address in pages 6 or 7, however, will find that the valid -invalid bit is set to invalid, and the computer will trap to the operating system (invalid page reference).

Notice that this scheme has created a problem.

Because the program extends only to address 10468, any reference beyond that address is illegal.

Howeve references to page 5 are classified as valid, so accesses to addresses up to 12287 are valid. Only the addresses from 12288 to 16383 are



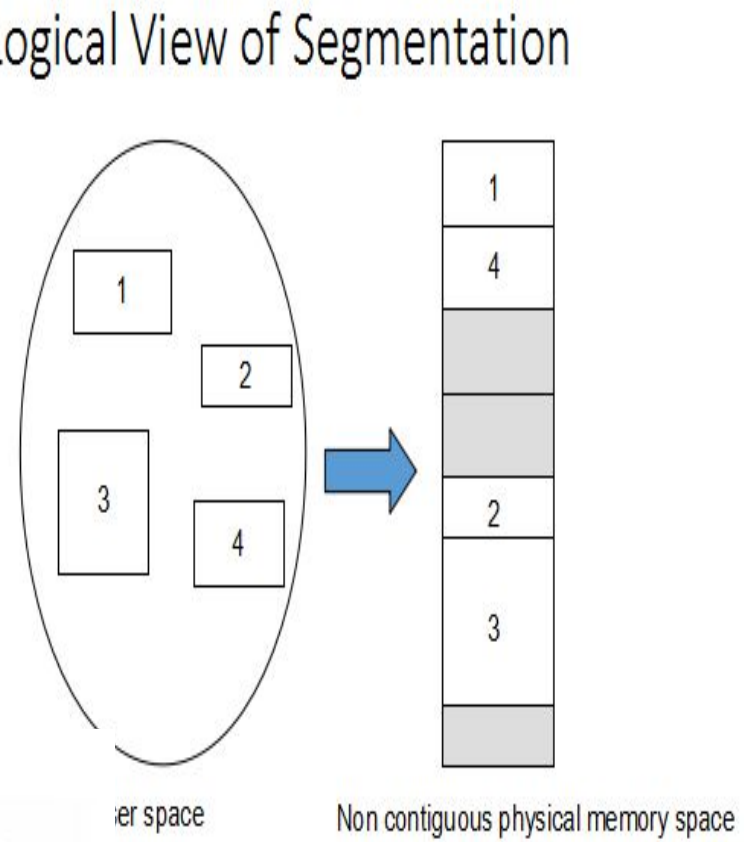
# Segmentation

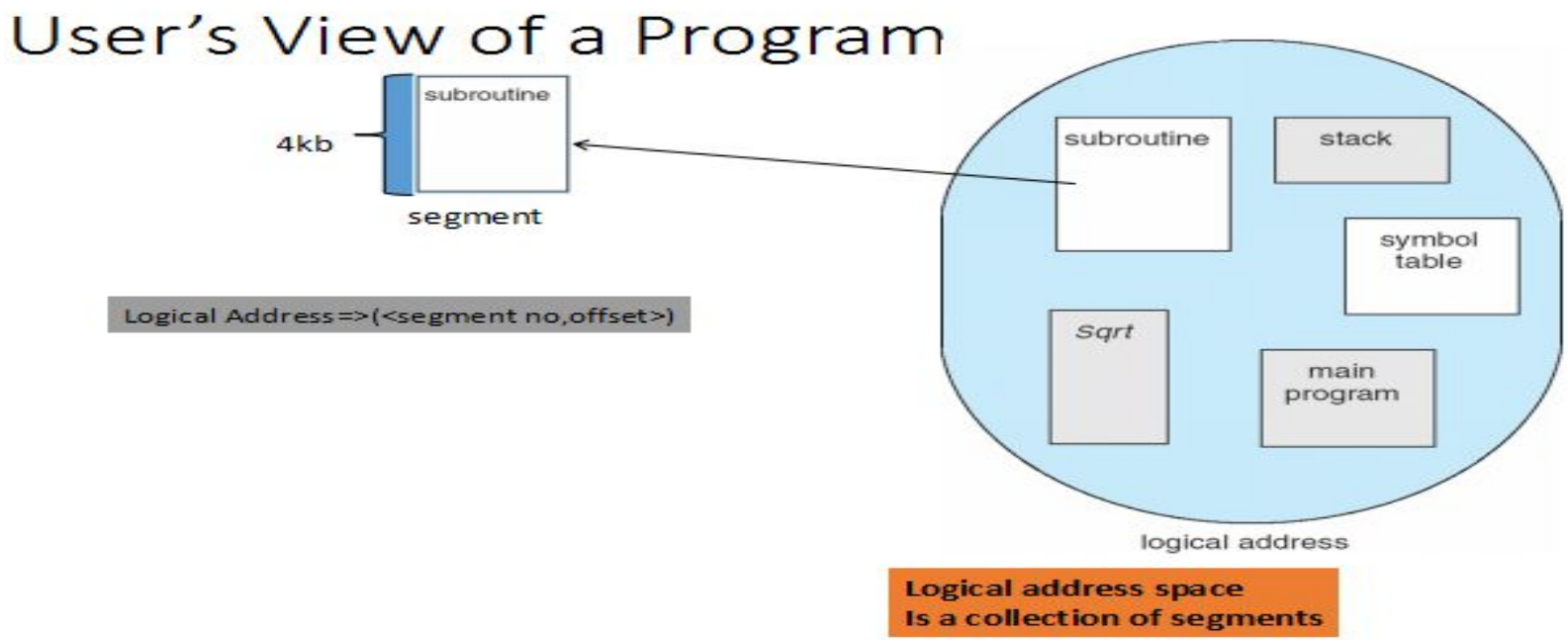
- An important aspect of memory management that became unavoidable with paging is the separation of the user's view of memory from the actual physical memory.
- The user's view of memory is not the same as the actual physical memory. The user's view is mapped onto physical memory.
- This mapping allows differentiation between logical memory and physical memory.
- Basic Method
- Do users think of memory as a linear array of bytes, some containing instructions and others containing data?
- Most people would say no.
- Rather, users prefer to view memory as a collection of variable-sized segments, with no necessary ordering among segments (Figure 8.18).
- Consider how you think of a program when you are writing it. You think of it as a main program with a set of methods, procedures, or functions. It may also include various data structures: objects, arrays, stacks, variables, and so on. Each of these modules or data elements is referred to by name. You talk about "the stack," "the math library," "the n1.ain program," without caring what addresses in memory these elements occupy. You are not concerned with whether the stack is stored before or after the Sqrt () function. Each of these segments is of variable length; the length is intrinsically defined by the purpose of the segment in the program.

# Segmentation

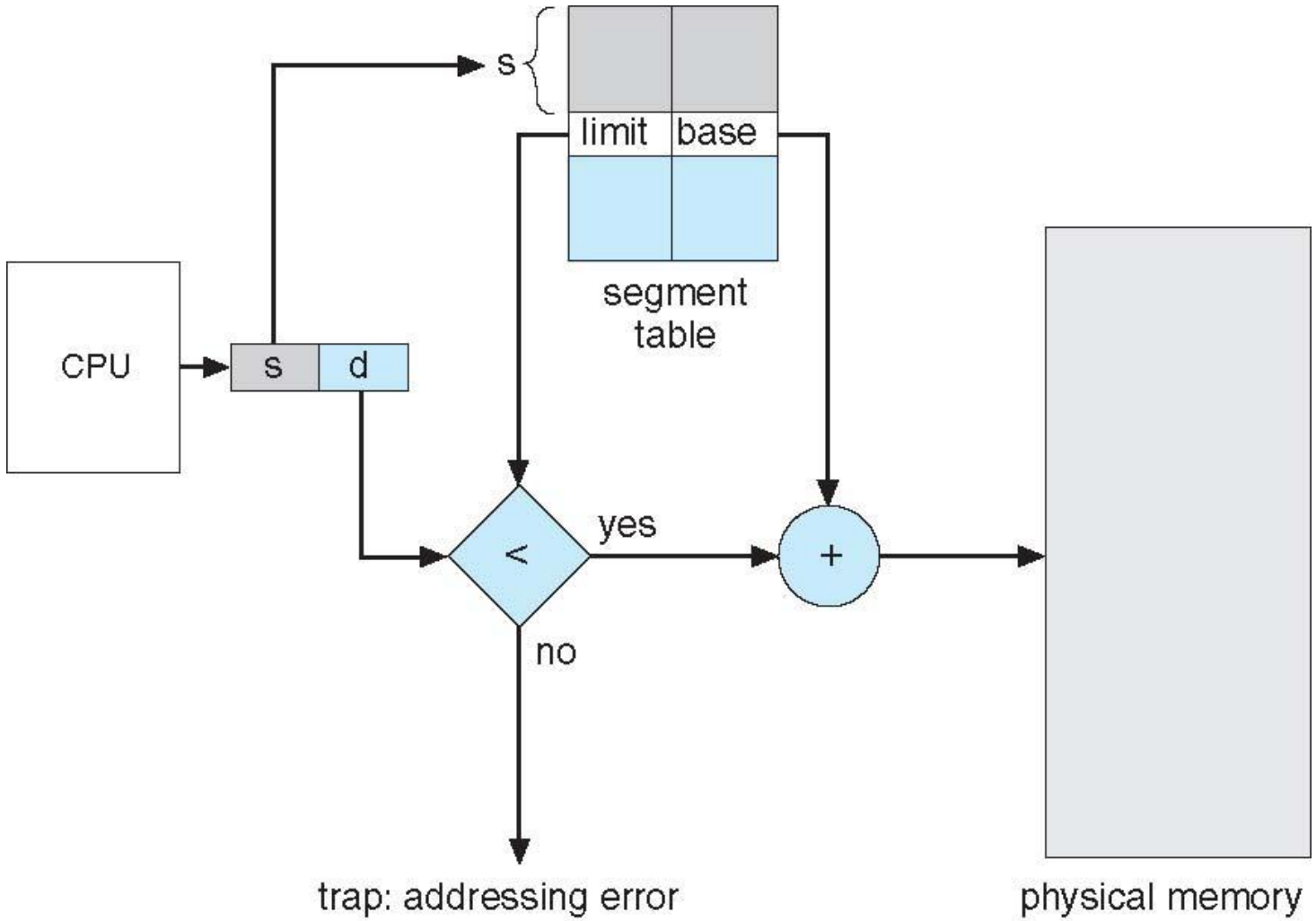
- Elements within a segment are identified by their offset from the begining of the segment: the first statement of the program, the seventh stack frame entry in the stack, the fifth instruction of the Sqrt (), and so on.
- Segmentation is a memory-management scheme that supports this user view of memory.
- A logical address space is a collection of segments.
- Each segment has a name and a length.
- The addresses specify both the segment name and the offset within the segment.
- The user therefore specifies each address by two quantities: a segment name and an offset. (Contrast this scheme with the paging scheme, in which the user specifies only a single address, which is partitioned by the hardware into a page number and an offset, all invisible to the programmer.)
- For simplicity of implementation, segments are numbered and are referred to by a segment number, rather than by a segment name. Thus, a logical address consists of a two tuple:
- <segment-number, offset>.

- Normally, the user program is compiled, and the compiler automatically constructs segments reflecting the input program.
- A C compiler might create separate segments for the following:
- The code
- Global variables
- The heap, from which memory is allocated
- The stacks used by each thread
- The standard C library





User's View of a Program
4kb
subroutine
segment
Logical Address=>(<segment no,offset>)
subroutine
stack
symbol table
Sqrt
main program
logical address
Logical address space
Is a collection of segments

# Segmentation Hardware

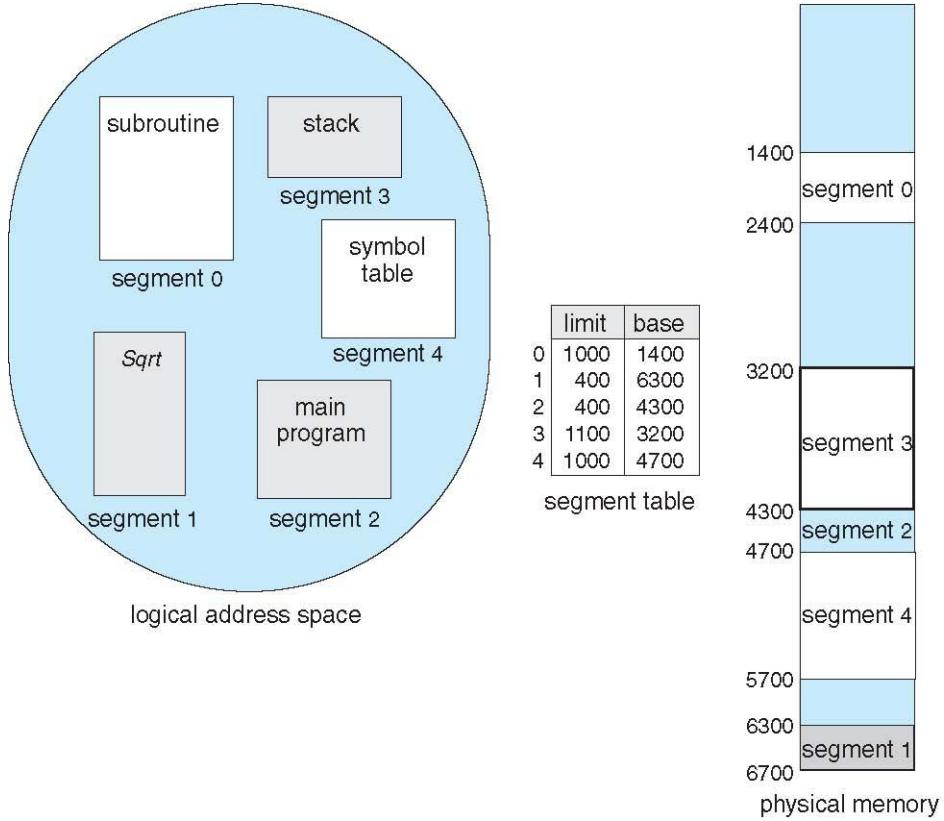


# Segmentation Architecture

- Logical address consists of a two tuple:

  <segment-number, offset>,

- **Segment table** – maps two-dimensional physical addresses; each table entry has:
  - **base** – contains the starting physical address where the segments reside in memory
  - **limit** – specifies the length of the segment
- **Segment-table base register (STBR)** points to the segment table's location in memory
- **Segment-table length register (STLR)** indicates number of segments used by a program;



# Problem 1

- Assuming a 1-KB page size, what are the page numbers and offsets for the following address references (provided as decimal numbers): a. 3085 b. 42095 c. 215201 d. 650000

- solution
- page size = 1 KB = 1024 B
- Page number Offset
- 3085/1024 = 3 3085 mod 1024 = 13
- 42095/1024 =41 42095 mod 1024 = 111
- 215201/1024 =210 215201 mod 1024 =161
- 650000/1024 = 634 650000 mod 1024 =784
- 2000001/1024= 1953 2000001 mod 1024 = 129

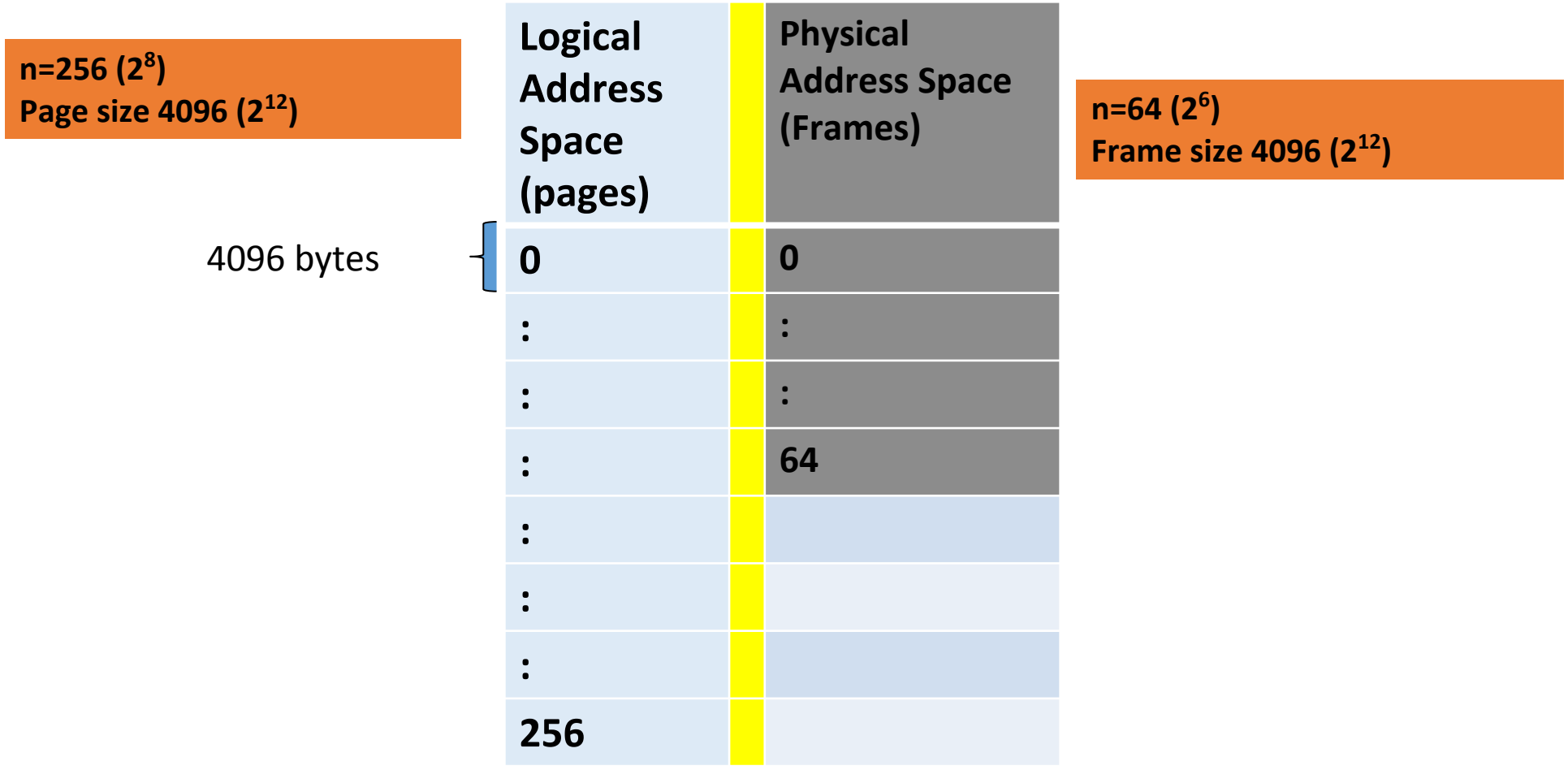
# Problem 2

- Assuming a 1-KB page size, what are the page numbers and offsets for the following address references (provided as decimal numbers):
- a. 2375
- b. 19366
- c. 30000
- d. 256
- E. 16385

| Logical address (decimal) | Page # (decimal) | Offset (decimal) |
|---|---|---|
| **2375** | **2** | **327** |
| **19366** | **18** | **934** |
| **30000** | **29** | **304** |
| **256** | **0** | **256** |
| **16385** | **16** | **1** |

# Problem 3

- Consider a logical address space of 256 pages with a 4-KB page size, mapped onto a physical memory of 64 frames.
- a. How many bits are required in the logical address?
- b. How many bits are required in the physical address?



# solution

- Size of logical address space

  $= 2^m$ = # of pages(n) x page size

  $= 256 \times 4096$

  $= 2^8 \times 2^{12} = 2^{20}$

**Number of required bits in logical adderss = 20**

- Size of physical address space

  = # of frames × frame size

  (frame size = page size )

  Size of physical address space

  $= 64 \times 4096$

  $= 2^6 \times 2^{12} = 2^{18}$

**Number of required bits in physical adderss = 18**

# Problem 4

- Consider a logical address space of 64 pages of 1024 words each, mapped onto a physical memory of 32 frames.
- a. How many bits are required in the logical address?
- b. How many bits are required in the physical address?

a) Logical memory = 64 pages= $2^6$ pages

size of each word = 1024 = $2^{10}$

Hence total logical memory = $2^6$ x $2^{10}$ = $2^{16}$

Hence the logical address will have 16 bits.

b) Physical memory = 32 frames= $2^5$ frames

size of each word = 1024 = $2^{10}$

Hence total physical size = $2^5$ x $2^{10}$ = $2^{15}$

Hence there will be 15 bits in the physical address

# Problem5

- ***Consider a logical address space of 32 pages of 1024 words each, mapped onto a physical memory of 16 frames.
- a. How many bits are required in the logical address?
- b. How many bits are required in the physical address?

a)Size of logical address space = 2m = # of pages × page size
= 32 × 1024 = 215 »» m=15 bit
b) Size of physical address space = # of frames × frame size
(frame size = page size )
Size of physical address space = 16 × 1024 = 214
»» number of required bits in the physical address =14 bit

# Problem 6

- Consider a paging system with the page table stored in memory.
- a. If a **memory reference takes 50 nanoseconds**, how long does a **paged memory reference** take?
- b. If we **add TLBs**, and **75 percent** of all page-table references are found **in the TLBs**, what is the effective memory reference time? (Assume that finding a **page-table entry in the TLBs takes 2 nanoseconds**, if the entry is present.)

# solution

- a. memory reference time= 50+50= 100 ns
- 50 ns to access the page table in RAM and 50 ns to access the word in memory
- B. if page entry found in TLB Memory access time = 2+ 50=52 ns

2 ns to access the page table in TLB and 50 ns to access the word in memory

- If page entry NOT found in TLB but found in page table in RAM Memory access time = 2 + 50 + 50 = 102 ns

2 ns to access the page table in TLB ,50 ns to access the page table in RAM and 50 ns to access the word in memory

**Effective access time = [0.75 × 52 ]+ [0.25 × 102]= 64.5 ns.**

# Problem 7

- 1. Consider the following segment table:

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

- Give the physical address for each of the above logical addresses

a) 0,430

b) 1,10

c) 2,500

d) 3,400

e) 4,112

# Solution

- a) <0,430>

  430 < 600 => 219 + 430 = 649

- b) <1,10>

  10 < 14 => 2300 + 10 = 2310

- c) <2,500>

  500 > 100 => segmentation fault

- d) <3,400>

  400 < 580 => 1327 + 400 = 1727

- e) <4,112>

  112 > 96 => segmentation fault

# Problem8

- 2. Suppose the Frame Size is 2048 and

  a process' page table is:

| Page Number | Frame Number | Valid |
|---|---|---|
| [0] | 4 | T |
| [1] | 2 | T |
| [2] | - | F |
| [3] | 3 | T |

Give the physical address for each of the following logical addresses or write "pagefault" if the reference would cause a page fault:

a)56 b)5000 c)6500

# Solution

- a) 56

  page# = 56 DIV 2048 = 0

  offset = 56 MOD 2048 = 56

  physical address = 4 * 2048 + 56 = 8248

- b) 5000

  page# = 5000 DIV 2048 = 2

  offset = 5000 MOD 2048 = 904

  page fault

- c) 6500

  page# = 6500 DIV 2048 = 3

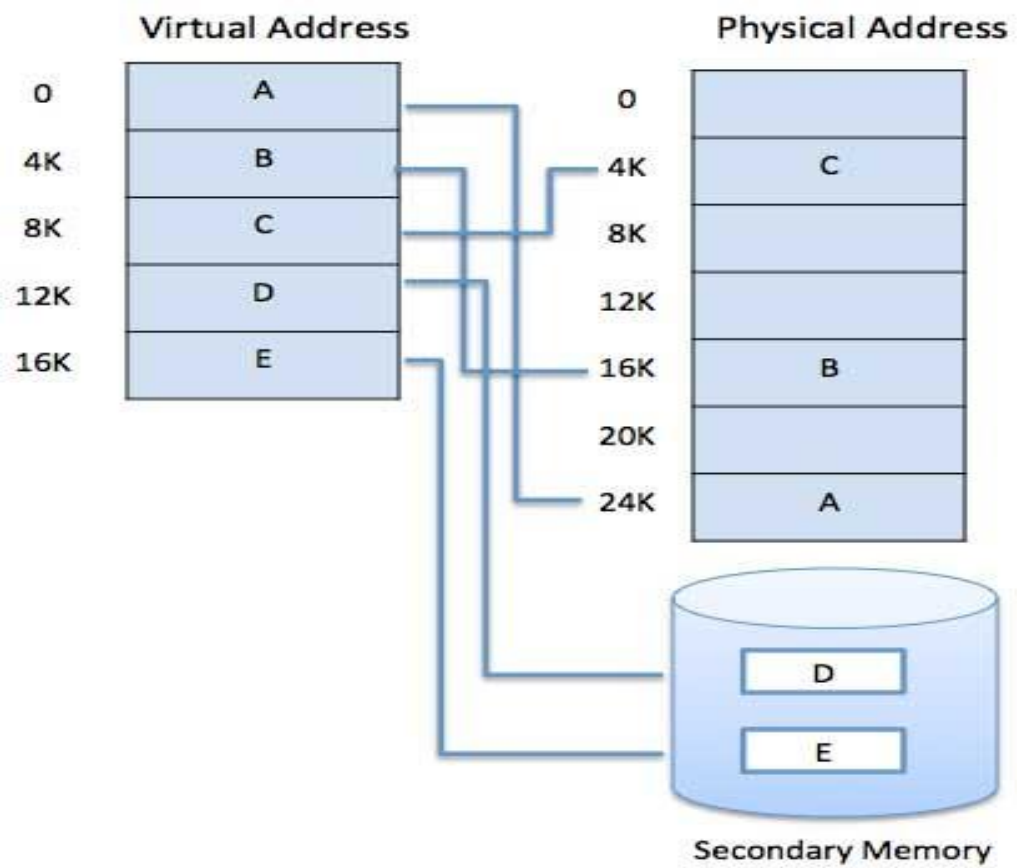
  offset = 6500 MOD 2048 = 356

  physical address = 3 * 2048 + 356 = 6500

# Virtual Memory Management :

- the goal of various memory-management strategies is to keep many processes in memory simultaneously to allow multiprogramming.
- However, they tend to require that an entire process be in memory before it can execute.
- Virtual memory is a technique that allows the execution of processes that are not completely in memory.
- One major advantage of this scheme is that programs can be larger than physical memory.
- Further, virtual memory abstracts main memory into an extremely large, uniform array of storage, separating logical memory as viewed by the user from physical memory.
- This technique frees programmers from the concerns of memory-storage limitations.
- Virtual memory also allows processes to share files easily and to implement shared memory. In addition, it provides an efficient mechanism for process creation.
- Virtual memory is not easy to implement, however, and may substantially decrease performance if it is used carelessly.

# Virtual Memory



# Background

- Virtual Memory involves the separation of logical memory as perceived by users from physical memory.
- This separation allows an extremely large virtual memory to be provided for programmers when only a smaller physical memory is available (Figure 9.1).
- Virtual memory makes the task of programming much easier, because the programmer no longer needs to worry about the amount of physical memory available;
- The Virtual address space of a process refers to the logical (or virtual) view of how a process is stored in memory. Typically, this view is that a process begins at a certain logical address-say, address 0-and exists in contiguous memory, as shown in Figure 9.2. , Physical memory may be organized in page frames and that the physical page frames assigned to a process may not be contiguous.
- It is up to the memory management unit (MMU) to map logical pages to physical page frames in memory.
- Note in Figure 9.2 that we allow for the heap to grow upward in memory as it is used for dynamic memory allocation. Similarly, we allow for the stack to grow downward in memory through successive function calls. The large blank space (or hole) between the heap and the stack is part of the virtual address space but will require actual physical pages only if the heap or stack grows. Virtual address spaces that include holes are known as sparse address spaces. Using a sparse address space is beneficial because the holes can be filled as the stack or heap segments grow or if we wish to dynamically link libraries (or possibly other shared objects) during program execution.

- Way of storing process in memory

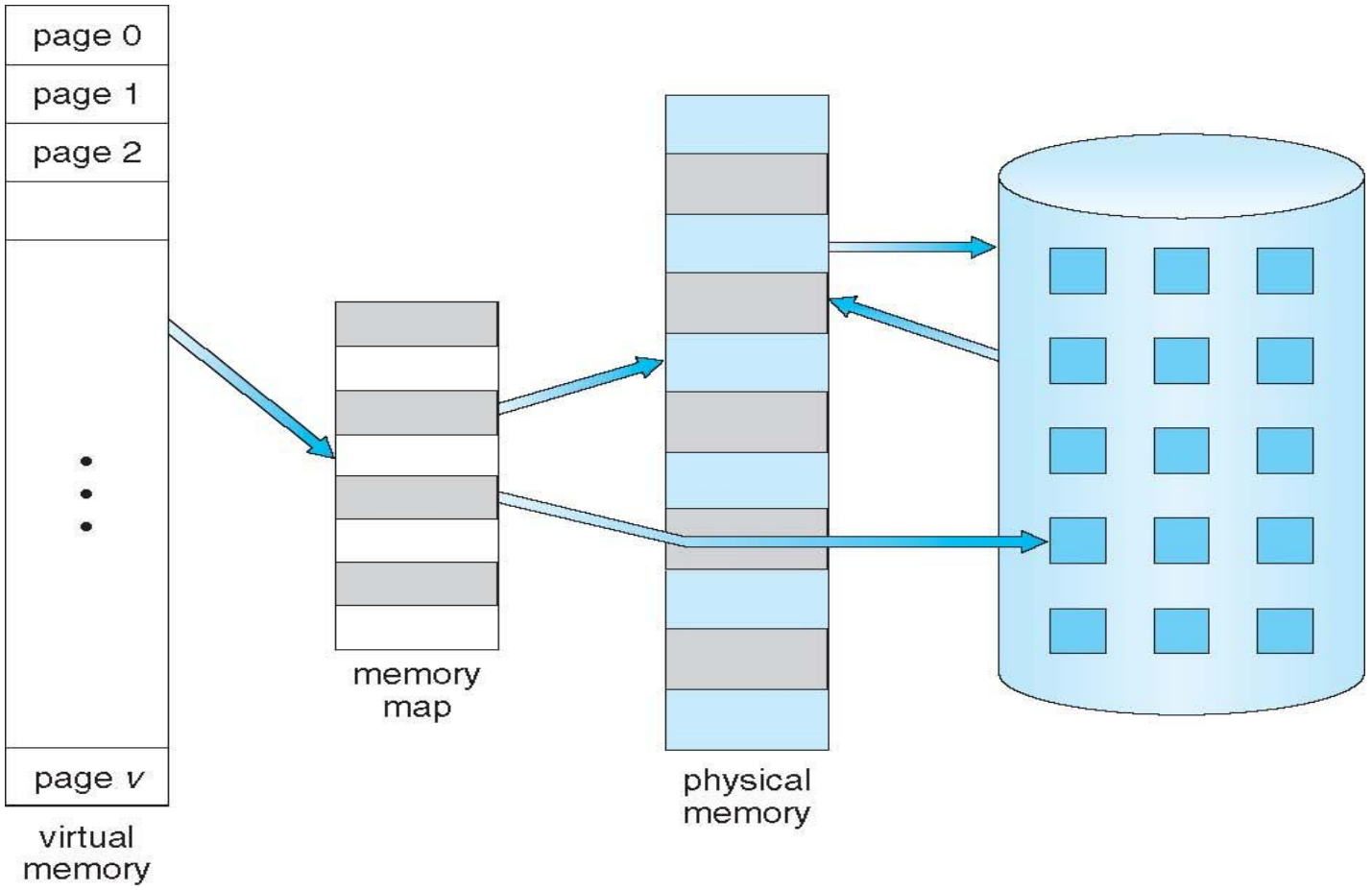
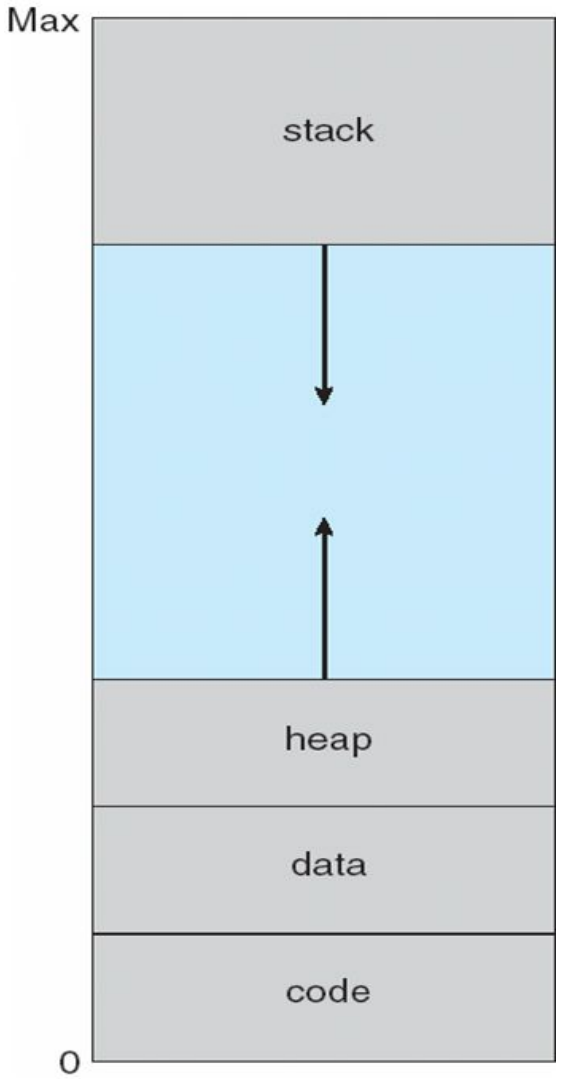



fig 9.1 virtual memory larger than physical memory

Fig 9.2 virtual address space



# Advatages of virtual memory

- System libraries can be shared by several processes through mapping of the shared object into a virtual address space.
- Although each process considers the shared libraries to be part of its virtual address space, the actual pages where the libraries reside in physical memory are shared by all the processes (Figure 9.3).
- Typically, a library is mapped read-only into the space of each process that is linked with it.
- Similarly, virtual memory enables processes to share memory. Two or more processes can communicate through the use of shared memory. Virtual memory allows one process to create a region of memory that it can share with another process.
- Processes sharing this region consider it part of their virtual address space, yet the actual physical pages of memory are shared, much as is illustrated in Figure 9.3.
- Virtual memory can allow pages to be shared during process creation with the fork() system call thus speeding up process creation.

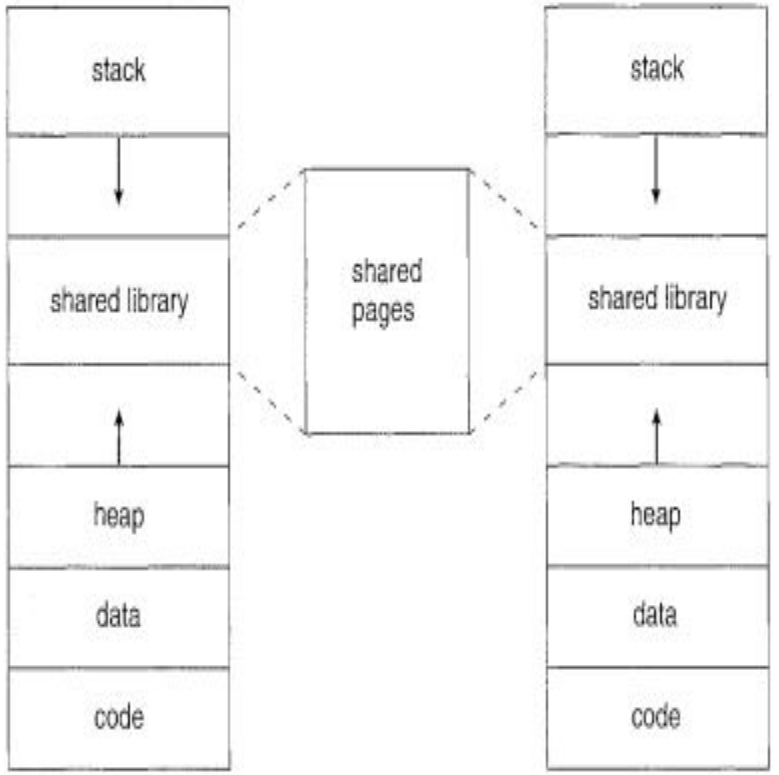



**Figure 9.3** Shared library using virtual memory.

- Virtual memory can be implemented via:

  - **Demand paging**

  - **Pages are loaded when demanded during execution**
  - **Pages that are never accessed never loaded**

  - **Advantages**
  - **Large virtual memory.**
  - **More efficient use of memory.**
  - **There is no limit on degree of multiprogramming.**
  - **Disadvantages**
  - **Number of tables and the amount of processor overhead for handling page interrupts are greater than in the case of the simple paged management techniques.**

# Page Fault

1. Occurs when page is

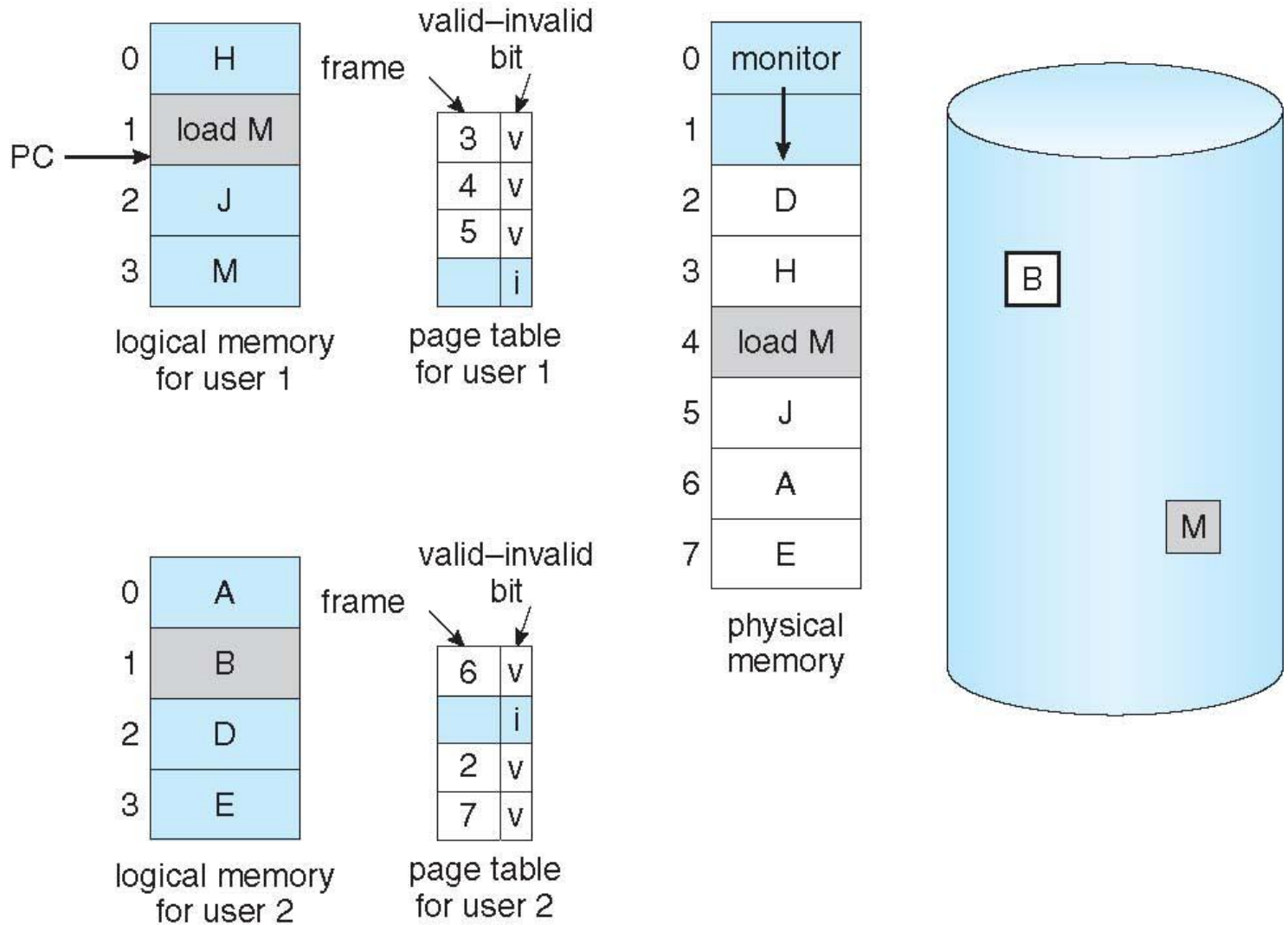
- Illegal Page
  - Abort process
- Invalid Page
  - Arrangement of free frame

- While executing a program, if the program references a page which is not available in the main memory because it was swapped out a little ago, the processor treats this invalid memory reference as a page fault and transfers control from the program to the operating system to demand the page back into the memory.

# Page Replacement

- If we increase our degree of multiprogramming, we are memory.
- If we run six processes, each of which is ten pages in size but uses only five pages, we have higher CPU utilization and throughput, ten frames to spare.
- It is possible, however, that each of these processes, for a particular data set, may suddenly try to use all ten of its pages, resulting in a need for sixty frames when only forty are available.
- The operating system could instead swap out a process, freeing all its frames and reducing the level of multiprogramming.
- Over-allocation of memory manifests itself as follows. While a user process is executing, a page fault occurs. The operating system determines where the desired page is residing on the disk but then finds that there are no free frames on the free-frame list; all memory is in use (Figure 9.9).
- The operating system has several options at this point. It could terminate the user process. However, demand paging is the operating system's attempt to improve the computer system's utilization and throughput. Users should not be aware that their processes are running on a paged system-paging should be logically transparent to the user. So this option is not the best choice.
- The operating system could instead swap out a process, freeing all its frames and reducing the level of multiprogramming. This option is a good one in certain circumstances, and we consider it further in Section 9.6. Here, we discuss the most common solution:Page replaceent

# 9.9 Need For Page Replacement



## Page Replacement

Page replacement takes the following approach.

If no frame is free, we find one that is not currently being used and free it. We can free a frame by writing its contents to swap space and changing the page table (and all other tables) to indicate that the page is no longer in memory (Figure 9.10). We can now use the freed frame to hold the page for which the process faulted.

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

1. Find the location of the desired page on the disk.
2. Find a free frame:
    a. If there is a free frame, use it.
    b. If there is no free frame, use a page-replacement algorithnc 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.

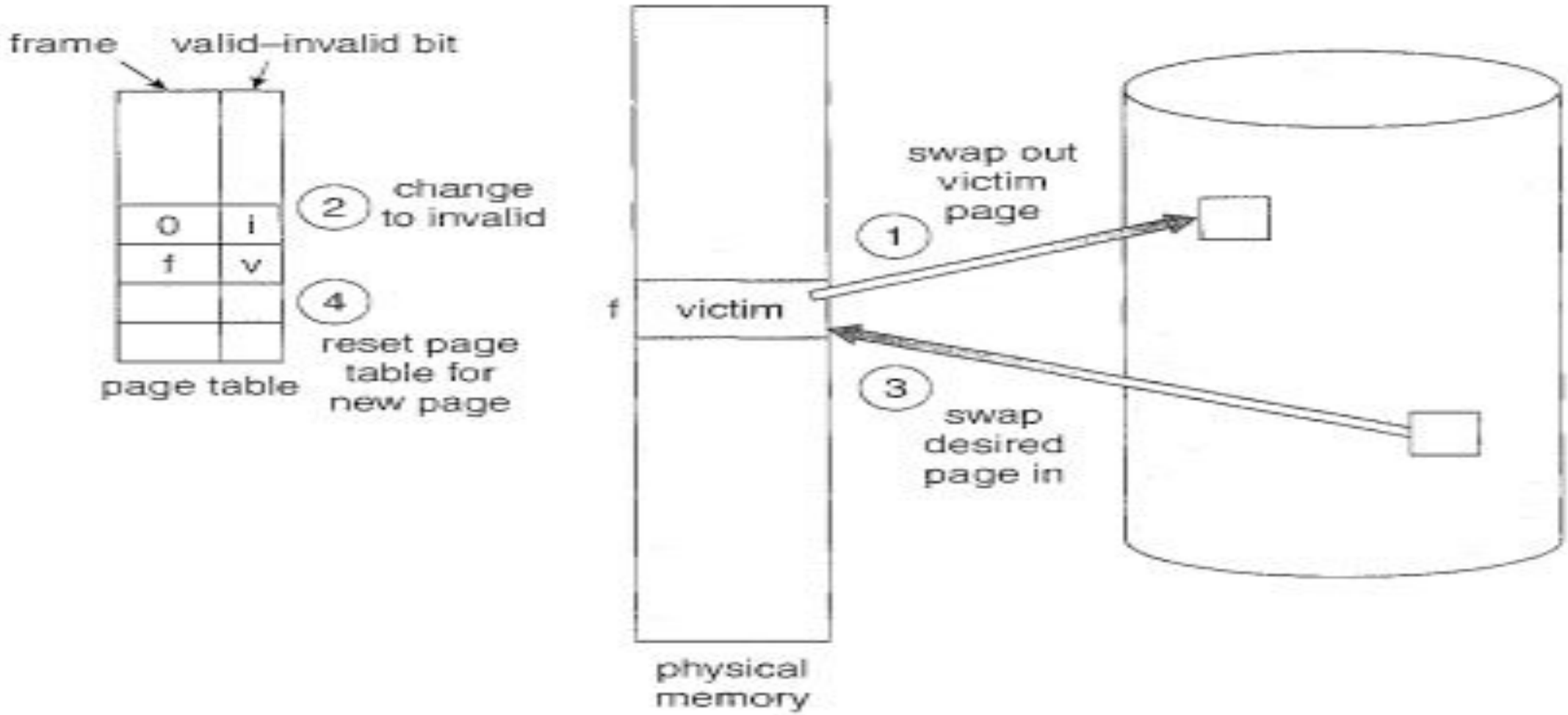



**Figure 9.10** Page replacement.

# Page Replacement

- Notice that, if no frames are free, two page transfers (one out and one in) are required.
- This situation effectively doubles the page-fault service time and increases the effective access time accordingly.
- We can reduce this overhead by using a modify bit or dirty bit (or 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 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 (for example, pages of binary code). Such pages cannot be modified; thus, they may be discarded when desired. This scheme can significantly reduce the time required to service a page fault, since it reduces I/O time by one-half if the page has not been modified.
- 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.

# Page Replacement Algorithms

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

# FIFO Page Replacement

- The simplest page-replacement algorithm
- It 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.
- The FIFO page-replacement algorithm is easy to understand and program.
- However, its performance is not always good.
- 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.

# First In First Out (FIFO):

Advantages –

- It is simple and easy to understand & implement.

Disadvantages –

- The process effectiveness is low.
- When we increase the number of frames while using FIFO, we are giving more memory to processes. So, page fault should decrease, but here the page faults are increasing. This problem is called as Belady's Anomaly.
- Every frame needs to be taken account off.

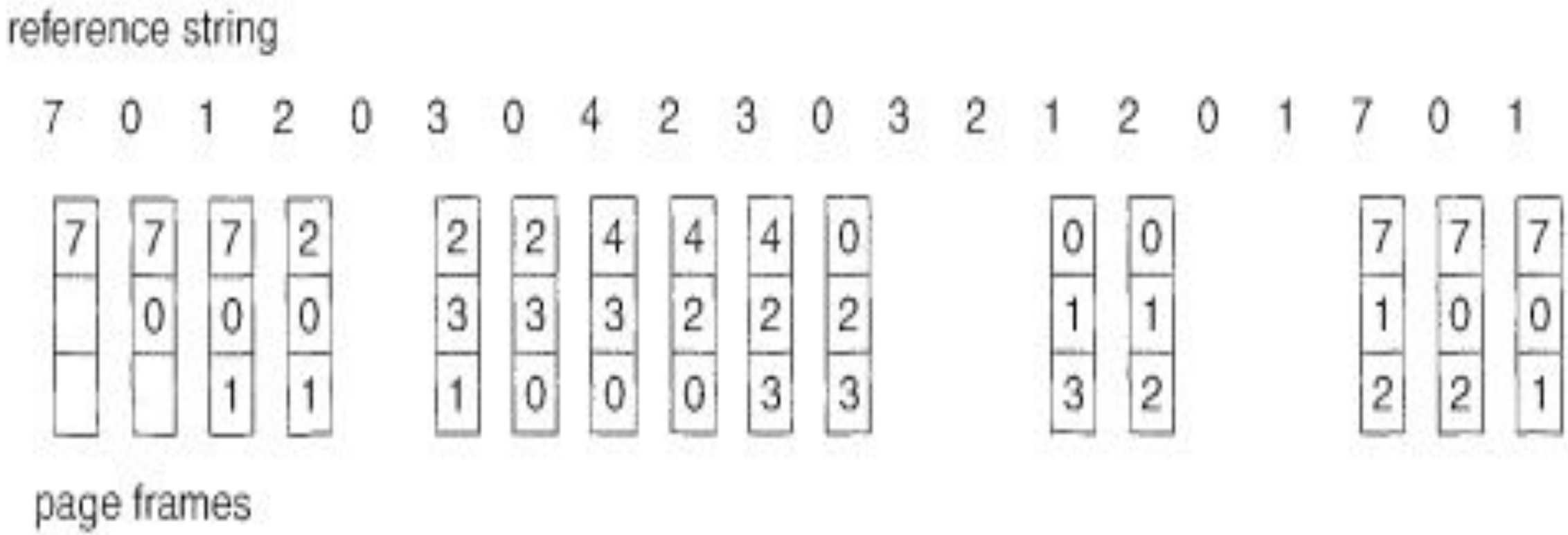


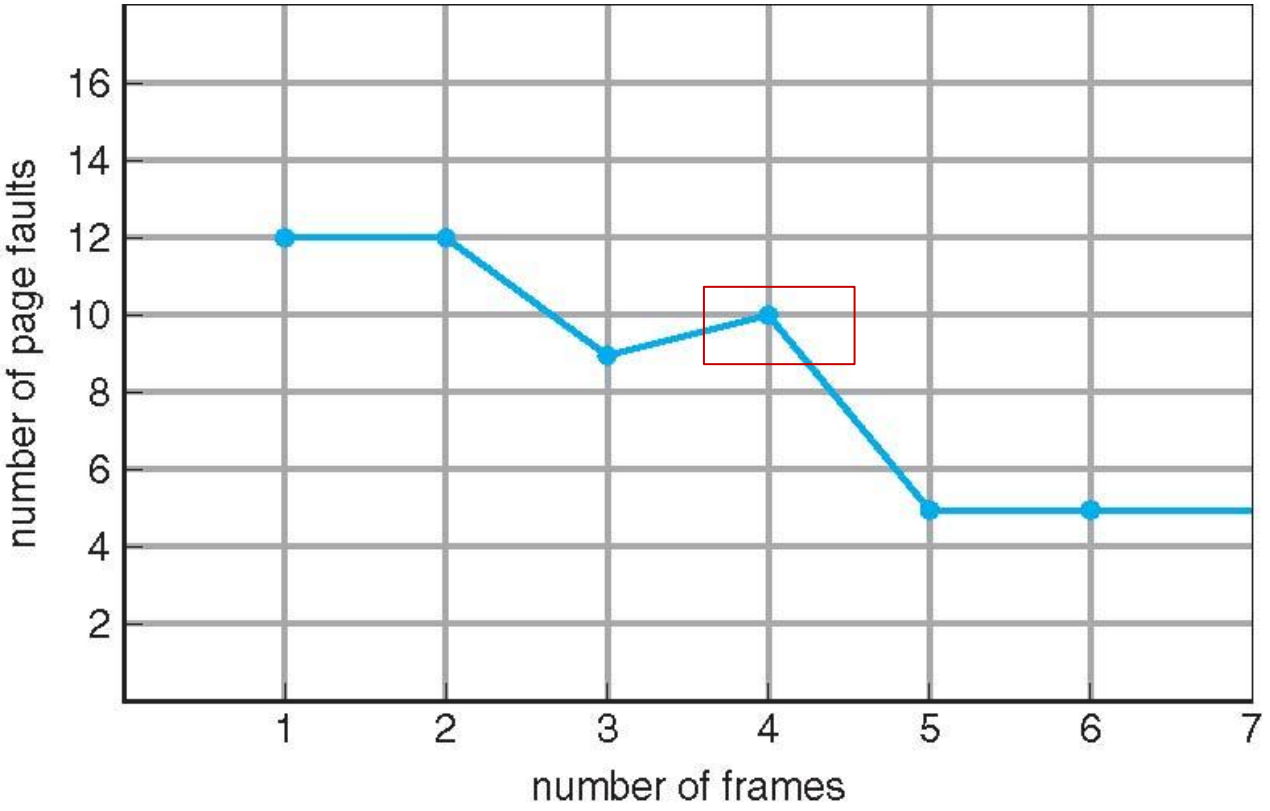

**Figure 9.12** FIFO page-replacement algorithm.

# FIFO Page Replacement

| | 7 | 0 | 1 | 2 | 0 | 3 | 0 | 4 | 2 | 3 | 0 | 3 | 2 | 1 | 2 | 0 | 1 | 7 | 0 | 1 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | ↓ | ↓ | ↓ | ↓ | | ↓ | ↓ | ↓ | ↓ | ↓ | ↓ | | | ↓ | ↓ | | | ↓ | ↓ | ↓ |
| 1 | 7 | 7 | 7 | 2 | | 2 | 2 | 4 | 4 | 4 | 0 | | | 0 | 0 | | | 7 | 7 | 7 |
| 2 | - | 0 | 0 | 0 | | 3 | 3 | 3 | 2 | 2 | 2 | | | 1 | 1 | | | 1 | 0 | 0 |
| 3 | - | - | 1 | 1 | | 1 | 0 | 0 | 0 | 3 | 3 | | | 3 | 2 | | | 2 | 2 | 1 |
| out | | | | 7 | | 0 | 1 | 2 | 3 | 0 | 4 | | | 2 | 3 | | | 0 | 1 | 2 |
| | x | x | x | x | √ | x | x | x | x | x | x | √ | √ | x | x | √ | √ | x | x | x |

15 Page Faults

# FIFO Illustrating Belady's Anomaly




Belady's Anamoly: Increase in number of frames increases page faults.

It can be only seen in FIFO.

# Optimal Algorithm

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

Advantages –

- Complexity is less and easy to implement.
- Assistance needed is low i.e Data Structure used are easy and light.

Disadvantages –

- OPR is perfect, but not possible in practice as the operating system cannot know future requests.
- Error handling is tough.

# Optimal Page Replacement

e.g. After 7,0,1 page fault will be for page 2. Reference to page 2 replaces Page 7 as among 7,0,1 page 7 will be needed last.

| | | 7 | 0 | 1 | 2 | 0 | 3 | 0 | 4 | 2 | 3 | 0 | 3 | 2 | 1 | 2 | 0 | 1 | 7 | 0 | 1 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | ↓ | ↓ | ↓ | ↓ | | ↓ | | ↓ | | | ↓ | | | ↓ | | | | ↓ | | |
| 1 | - | 7 | 7 | 7 | 2 | | 2 | | 2 | | | 2 | | | 2 | | | | 7 | | |
| 2 | - | - | 0 | 0 | 0 | | 0 | | 4 | | | 0 | | | 0 | | | | 0 | | |
| 3 | - | - | - | 1 | 1 | | 3 | | 3 | | | 3 | | | 1 | | | | 1 | | |
| out | | | | | 7 | | 1 | | 0 | | | 4 | | | 3 | | | | 2 | | |
| | | x | x | x | x | √ | x | √ | x | √ | √ | x | √ | √ | x | √ | √ | √ | X | √ | √ |

9 Page

# Least Recently Used (LRU)

- The LRU stands for the Least Recently Used.
- Least Recently Used page replacement algorithm keeps track of page usage over a short period of time.
- It works on the concept that pages that have been highly used in the past are likely to be significantly used again in the future.
- It removes the page that has not been utilized in the memory for the longest time.
- LRU is the most widely used algorithm because it provides fewer page faults than the other methods.
- Use past knowledge rather than future
- Replace page that has not been used in the most amount of time
- e.g 12 faults – better than FIFO but worse than OPT
- Generally good algorithm and frequently used

# Least Recently Used (LRU)

| 1 | 2 | 3 | 4 | 5 | 1 | 3 | 1 | 6 | 3 | 2 | 3 |
|---|---|---|---|---|---|---|---|---|---|---|---|

| 1 | 1 | 1 | 1 | 5 | 5 | 5 | 5 | 5 | 5 | 2 | 2 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | 2 | 2 | 2 | 2 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| | | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 |
| | | | 4 | 4 | 4 | 4 | 4 | 6 | 6 | 6 | 6 |
| M | M | M | M | M | M | H | H | M | H | M | H |

M = Miss
H = Hit

# Least Recently Used Page Replacement

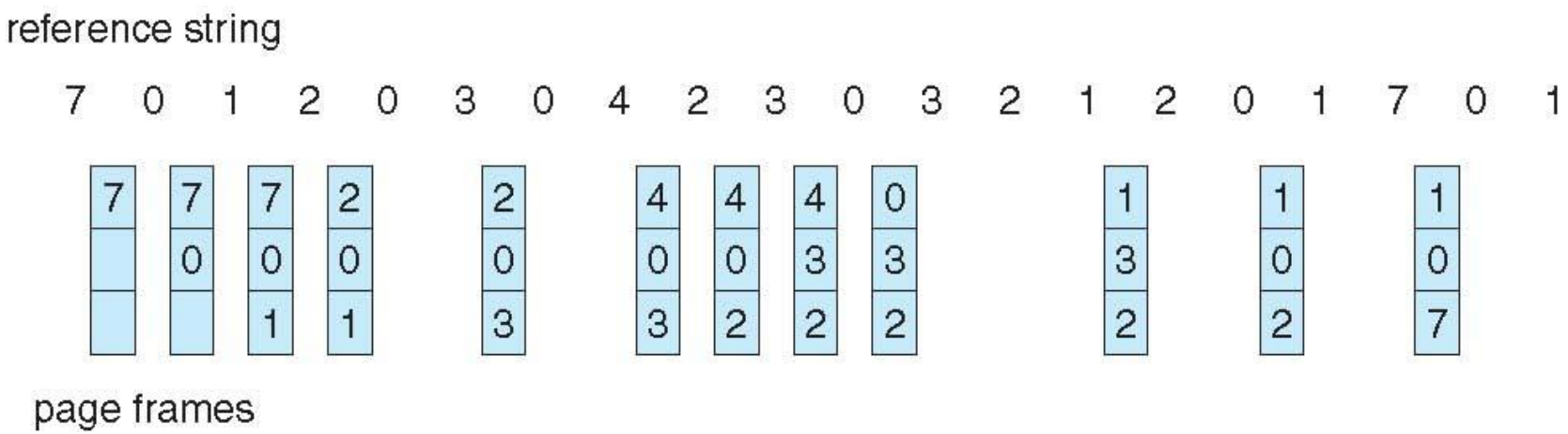
LRU chooses the page that has not been used for the longest period of time.

E,g, Reference to page 3 replaces page 1.

| | | 7 | 0 | 1 | 2 | 0 | 3 | 0 | 4 | 2 | 3 | 0 | 3 | 2 | 1 | 2 | 0 | 1 | 7 | 0 | 1 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | ↓ | ↓ | ↓ | ↓ | | ↓ | | ↓ | ↓ | ↓ | ↓ | | | ↓ | | ↓ | | ↓ | | |
| 1 | - | 7 | 7 | 7 | 2 | | 2 | | 4 | 4 | 4 | 0 | | | 1 | | 1 | | 1 | | |
| 2 | - | - | 0 | 0 | 0 | | 0 | | 0 | 0 | 3 | 3 | | | 3 | | 0 | | 0 | | |
| 3 | - | - | - | 1 | 1 | | 3 | | 3 | 2 | 2 | 2 | | | 2 | | 2 | | 7 | | |
| out | | | | | 7 | | 1 | | 2 | 3 | 0 | 4 | | | 0 | | 3 | | 2 | | |
| | | x | x | x | x | √ | x | √ | X | X | x | x | √ | √ | X | √ | X | √ | X | √ | √ |

12 Page

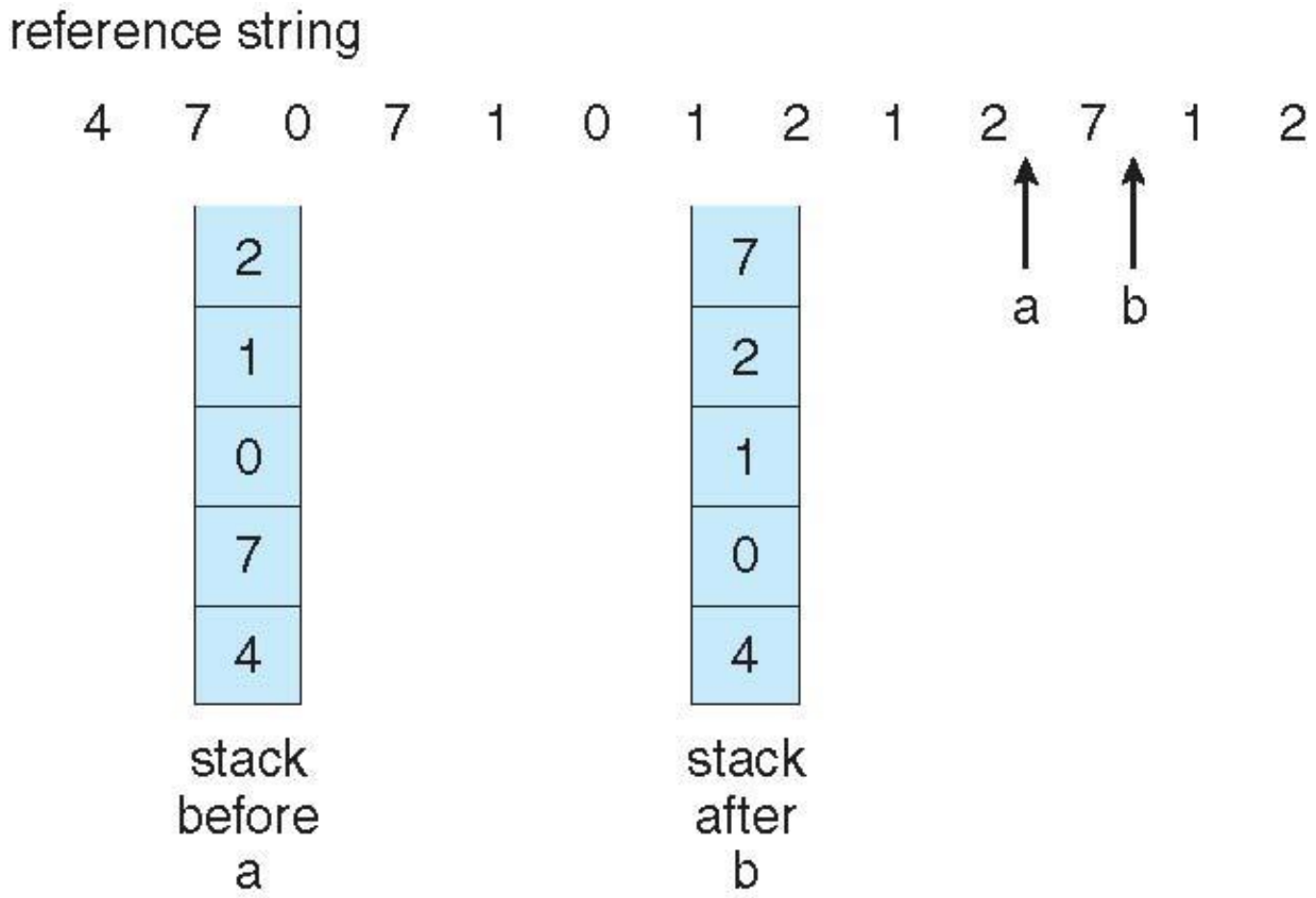
# Least Recently Used (LRU) Algorithm



# LRU Algorithm (Cont.)

- Counter implementation
  - Every page entry has a counter; every time page is referenced through this entry, copy the clock into the counter
  - When a page needs to be changed, look at the counters to find smallest value
    - Search through table needed
- Stack implementation
- LRU and OPT are cases of **stack algorithms** that don't have Belady's Anomaly

# Use Of A Stack to Record The Most Recent Page References



# Thrashing

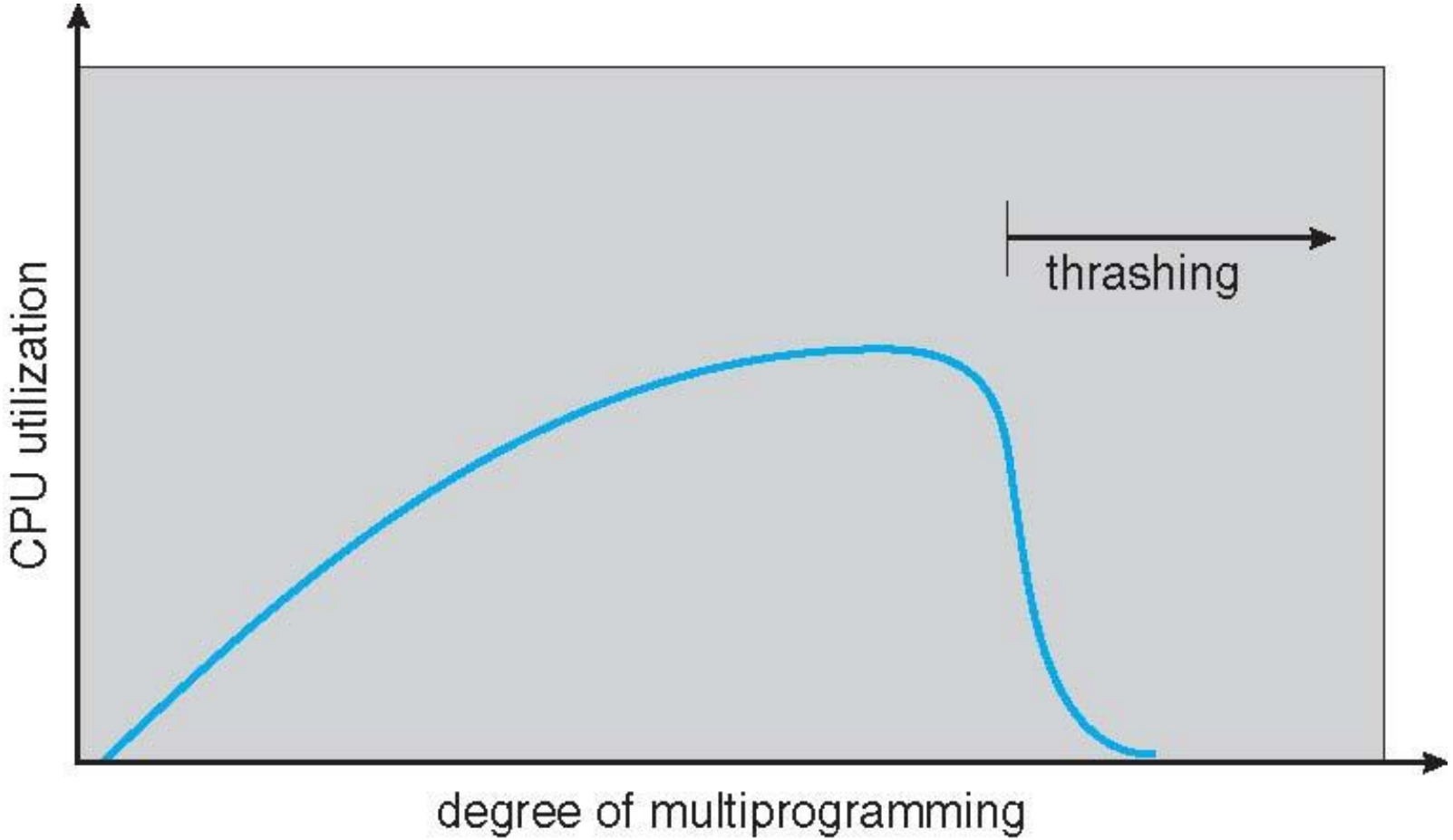
- Need of minimum number of frames is architecture dependent.
- Need of sufficient number of frames.
  - If not available,
  - Page fault rate very high.
  - System replaces active pages
  - But quickly need replaced frame back
  - This continue paging activity is called Thrashing

# Cause of Thrashing

- In Thrashing ,
  - System spends most of the time in performing I/O activity, and leads to
  - **Low CPU utilization**
  - Operating system thinking that it needs to increase the degree of multiprogramming
  - Another process added to the system
- If **Global Page Replacement** is used, system replaces active pages from other process
- If Local Page Replacement is used, local pages are replaced.

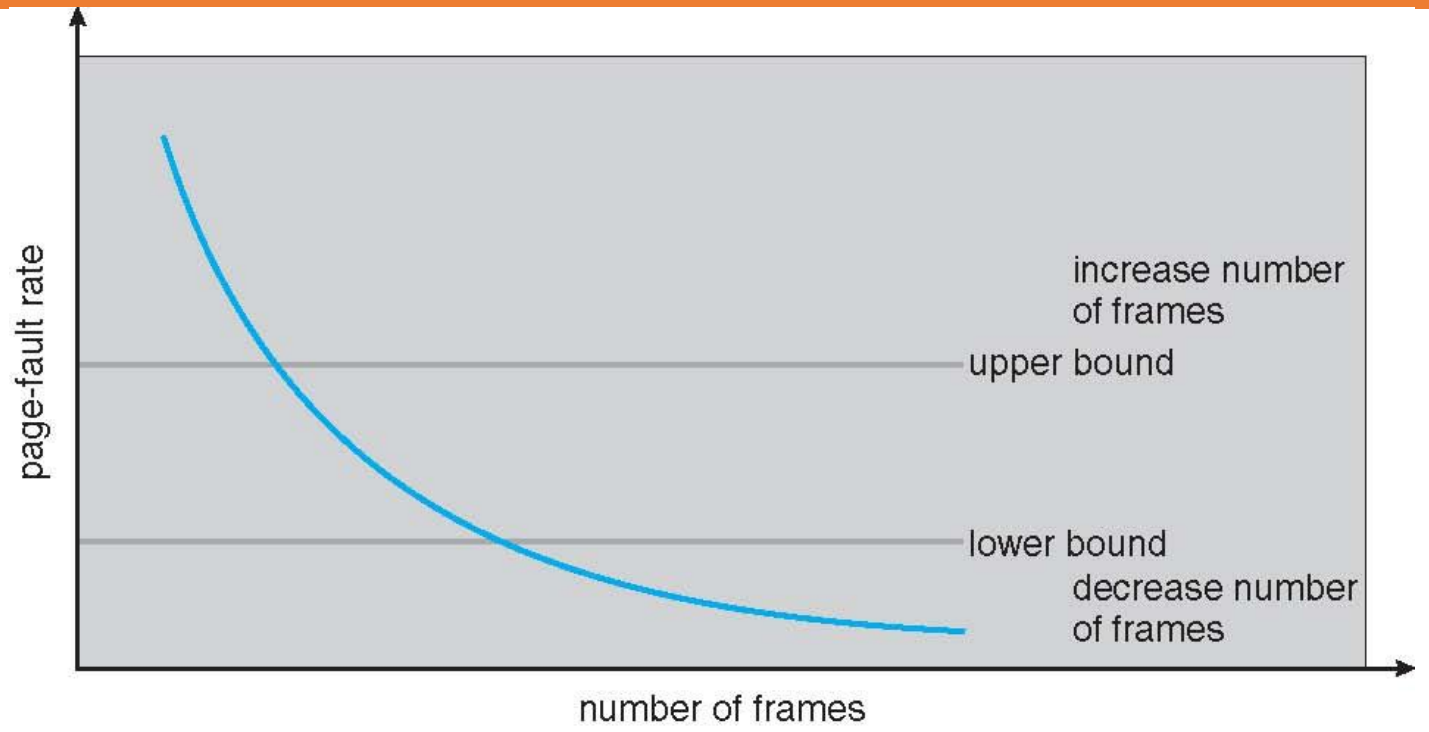
- **Thrashing** ≡ a process is busy swapping pages in and out

# Thrashing (Cont.)



# Page-Fault Frequency

- More direct approach than WS model
- Establish “acceptable” upper & lower bound for page fault rate.
- Accordingly increase or decrease frames



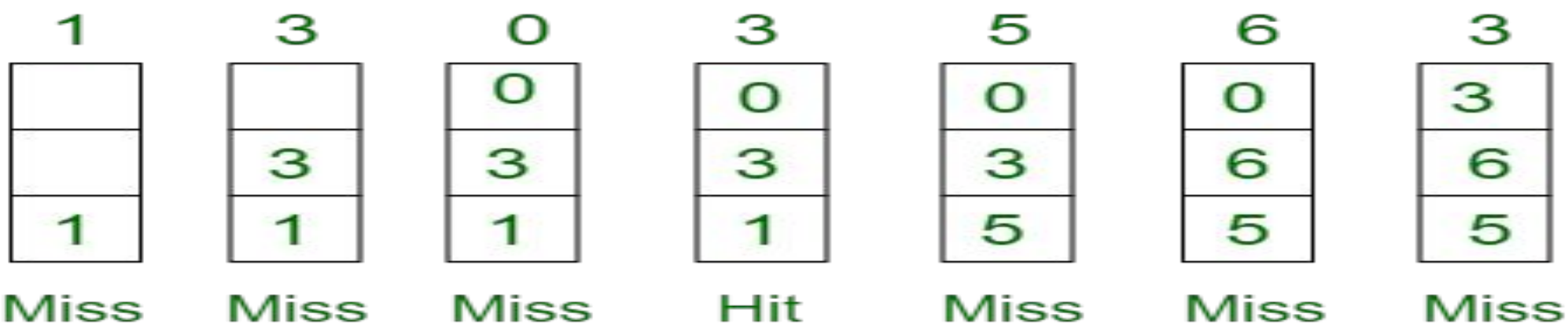
# Least Recently Used (LRU):

- Advantages –
- It is open for full analysis.
- In this, we replace the page which is least recently used, thus free from Belady's Anomaly.
- Easy to choose page which has faulted and hasn't been used for a long time.
- Disadvantages –
- It requires additional Data Structure to be implemented.
- Hardware assistance is high.

- Consider the following page reference string.
- 1, 3, 0, 3, 5, 6,3
- How many page faults would occur for the following replacement algorithm, Assuming three frames respectively?
- Which will be the efficient algorithm? Why?
- i) LRU page replacement.
- ii) FIFO page replacement.
- iii) Optimal page replacement

Page reference

1, 3, 0, 3, 5, 6, 3




Total Page Fault = 6

- Consider the following page reference string:
- 4 ,7, 6, 1, 7, 6, 1, 2, 7, 2
- How many page faults would occur for the optimal page replacement algorithm, assuming three frames and all frames are initially empty.

- Calculating Hit ratio-
- = Total number of references – Total number of page misses or page faults
- Thus, Hit ratio
- = Total number of page hits / Total number of references

- Thus, Miss ratio
- Total number of page misses / Total number of references
- Alternatively,
- Miss ratio
- = 1 – Hit ratio

- Consider the following page reference string:
- 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5
- How many page faults would occur for the optimal page replacement algorithm, assuming three frames and all frames are initially empty.

- Q. Consider a reference string: 4, 7, 6, 1, 7, 6, 1, 2, 7, 2.
- The number of frames in the memory is 3. Find out the number of page faults respective to:

- Optimal Page Replacement Algorithm
- FIFO Page Replacement Algorithm
- LRU Page Replacement Algorithm

# FIFO Page Replacement Algorithm

| Request | 4 | 7 | 6 | 1 | 7 | 6 | 1 | 2 | 7 | 2 |
|---|---|---|---|---|---|---|---|---|---|---|
| Frame 3 | | | 6 | 6 | 6 | 6 | 6 | 6 | 7 | 7 |
| Frame 2 | | 7 | 7 | 7 | 7 | 7 | 7 | 2 | 2 | 2 |
| Frame 1 | 4 | 4 | 4 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| Miss/Hit | Miss | Miss | Miss | Miss | Hit | Hit | Hit | Miss | Miss | Hit |

Number of Page Faults in FIFO = 6

# Optimal Page Replacement Algorithm

| Request | 4 | 7 | 6 | 1 | 7 | 6 | 1 | 2 | 7 | 2 |
|---|---|---|---|---|---|---|---|---|---|---|
| Frame 3 | | | 6 | 6 | 6 | 6 | 6 | 2 | 2 | 2 |
| Frame 2 | | 7 | 7 | 7 | 7 | 7 | 7 | 7 | 7 | 7 |
| Frame 1 | 4 | 4 | 4 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| Miss/Hit | Miss | Miss | Miss | Miss | Hit | Hit | Hit | Miss | Hit | Hit |

Number of Page Faults in Optimal Page Replacement Algorithm = 5

# LRU Page Replacement Algorithm

| Request | 4 | 7 | 6 | 1 | 7 | 6 | 1 | 2 | 7 | 2 |
|---|---|---|---|---|---|---|---|---|---|---|
| Frame 3 | | | 6 | 6 | 6 | 6 | 6 | 6 | 7 | 7 |
| Frame 2 | | 7 | 7 | 7 | 7 | 7 | 7 | 2 | 2 | 2 |
| Frame 1 | 4 | 4 | 4 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| Miss/Hit | Miss | Miss | Miss | Miss | Hit | Hit | Hit | Miss | Miss | Hit |

Number of Page Faults in LRU = 6

## Example

Assume there are 3 frames, and consider the reference string given in example . Show the content of memeory after each memory reference if FIFO page replacement algorithm isused.

Reference string: 5, 7, 6, 0, 7, 1, 7, 2, 0, 1, 7, 1, 0

Find also the number of page faults

**Example**

Assume there are 3 frames, and consider the reference string given in example . Show the content of memeory after each memory reference if FIFO page replacement algorithm isused.

Find also the number of page faults

| | 5 | 7 | 6 | 0 | 7 | 1 | 7 | 2 | 0 | 1 | 7 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| f1 | 5 | 5 | 5 | 0 | 0 | 0 | 0 | 2 | 2 | 2 | 7 | 7 | 7 |
| f2 | | 7 | 7 | 7 | 7 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 |
| f3 | | | 6 | 6 | 6 | 6 | 7 | 7 | 7 | 1 | 1 | 1 | 1 |
| pf | 1 | 2 | 3 | 4 | same | 5 | 6 | 7 | 8 | 9 | 10 | same | same |

10 page faults are caused by FIFO

**Example**

Assume we have 3 frames and consider the reference string below.

Reference string: 5, 7, 6, 0, 7, 1, 7, 2, 0, 1, 7, 1, 0

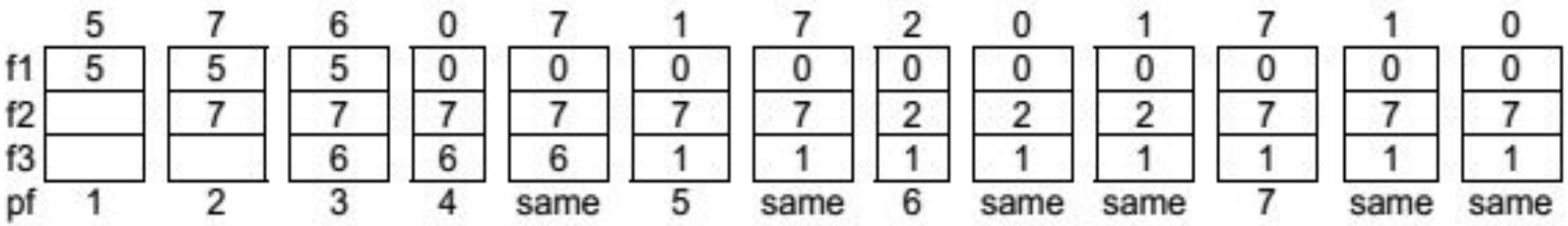
Show the content of memory after each memory reference if OPT page replacement algorithm is used. Find also the number of page faults

**Example**

Assume we have 3 frames and consider the reference string below.

Reference string: 5, 7, 6, 0, 7, 1, 7, 2, 0, 1, 7, 1, 0

Show the content of memory after each memory reference if OPT page replacement algorithm is used. Find also the number of page faults



| | 5 | 7 | 6 | 0 | 7 | 1 | 7 | 2 | 0 | 1 | 7 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| f1 | 5 | 5 | 5 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| f2 | | 7 | 7 | 7 | 7 | 7 | 7 | 2 | 2 | 2 | 7 | 7 | 7 |
| f3 | | | 6 | 6 | 6 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| pf | 1 | 2 | 3 | 4 | same | 5 | same | 6 | same | same | 7 | same | same |

According to the given information, this algorithm generates a page replacement scheme with 7 page faults.

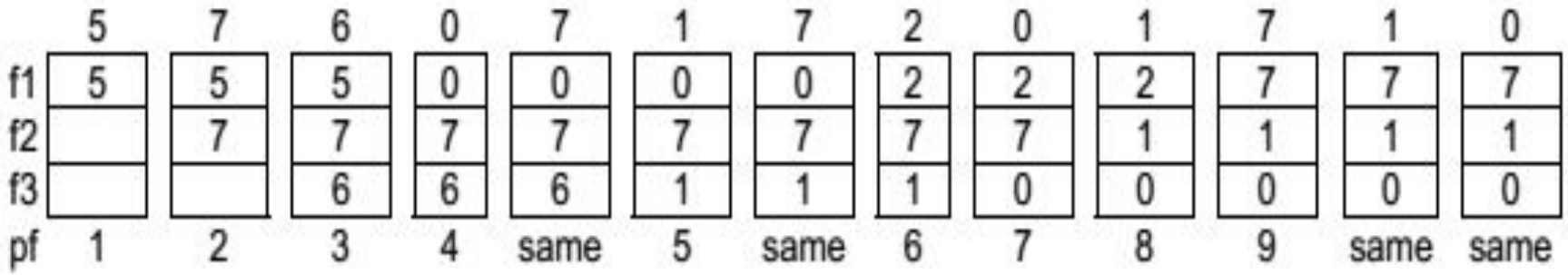
## Example

Assume there are 3 frames, and consider the reference string given in example . Show the content of memory after each memory reference if LRU page replacement algorithm is used. Find also the number of page faults

Reference string: 5, 7, 6, 0, 7, 1, 7, 2, 0, 1, 7, 1, 0

**Example**

Assume there are 3 frames, and consider the reference string given in example . Show the content of memory after each memory reference if LRU page replacement algorithm is used. Find also the number of page faults



| | 5 | 7 | 6 | 0 | 7 | 1 | 7 | 2 | 0 | 1 | 7 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| f1 | 5 | 5 | 5 | 0 | 0 | 0 | 0 | 2 | 2 | 2 | 7 | 7 | 7 |
| f2 | | 7 | 7 | 7 | 7 | 7 | 7 | 7 | 7 | 1 | 1 | 1 | 1 |
| f3 | | | 6 | 6 | 6 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 |
| pf | 1 | 2 | 3 | 4 | same | 5 | same | 6 | 7 | 8 | 9 | same | same |

This algorithm resulted in 9 page faults.