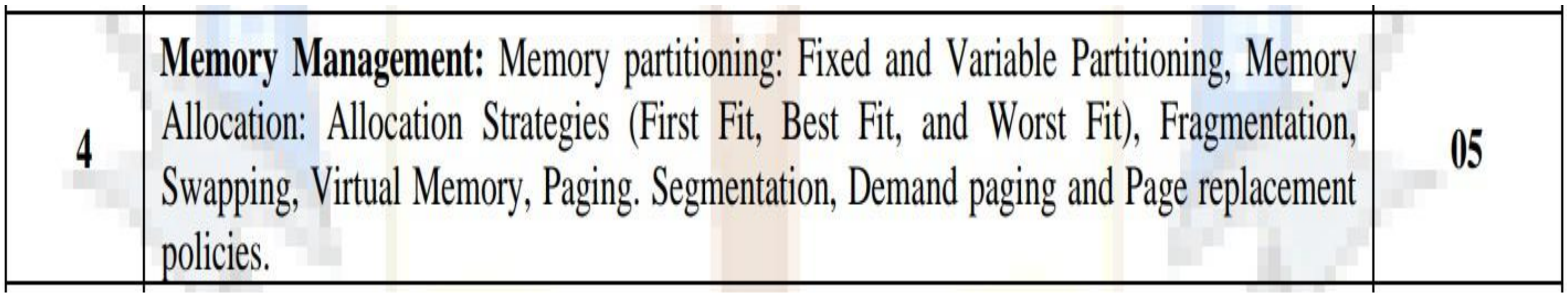
Chapter 4
Memory Management

Syllabus



4	**Memory Management:** Memory partitioning: Fixed and Variable Partitioning, Memory Allocation: Allocation Strategies (First Fit, Best Fit, and Worst Fit), Fragmentation, Swapping, Virtual Memory, Paging. Segmentation, Demand paging and Page replacement policies.	**05**

Memory Management

- Memory management is mainly concerned with the allocation of main memory to requesting process.

- No process can ever run before a certain amount of memory is allocated to it.

- The overall resource utilization and other performance criteria of a computer system are largely affected by the performance of the memory management module.

Memory Management Requirements

Memory Management Requirements

- Relocation
- Protection
- Sharing
- Logical Organization
- Physical Organization

Memory Management Requirements

- Relocation
  - programmer cannot know where the program will be placed in memory, when it is executed.
  - a process may be (often) **relocated** in main memory due to swapping.
  - swapping enables the OS to have a larger pool of ready-to-execute processes.
  - memory references in code (for both instructions and data) must be translated to actual physical memory address

Memory Management Requirements

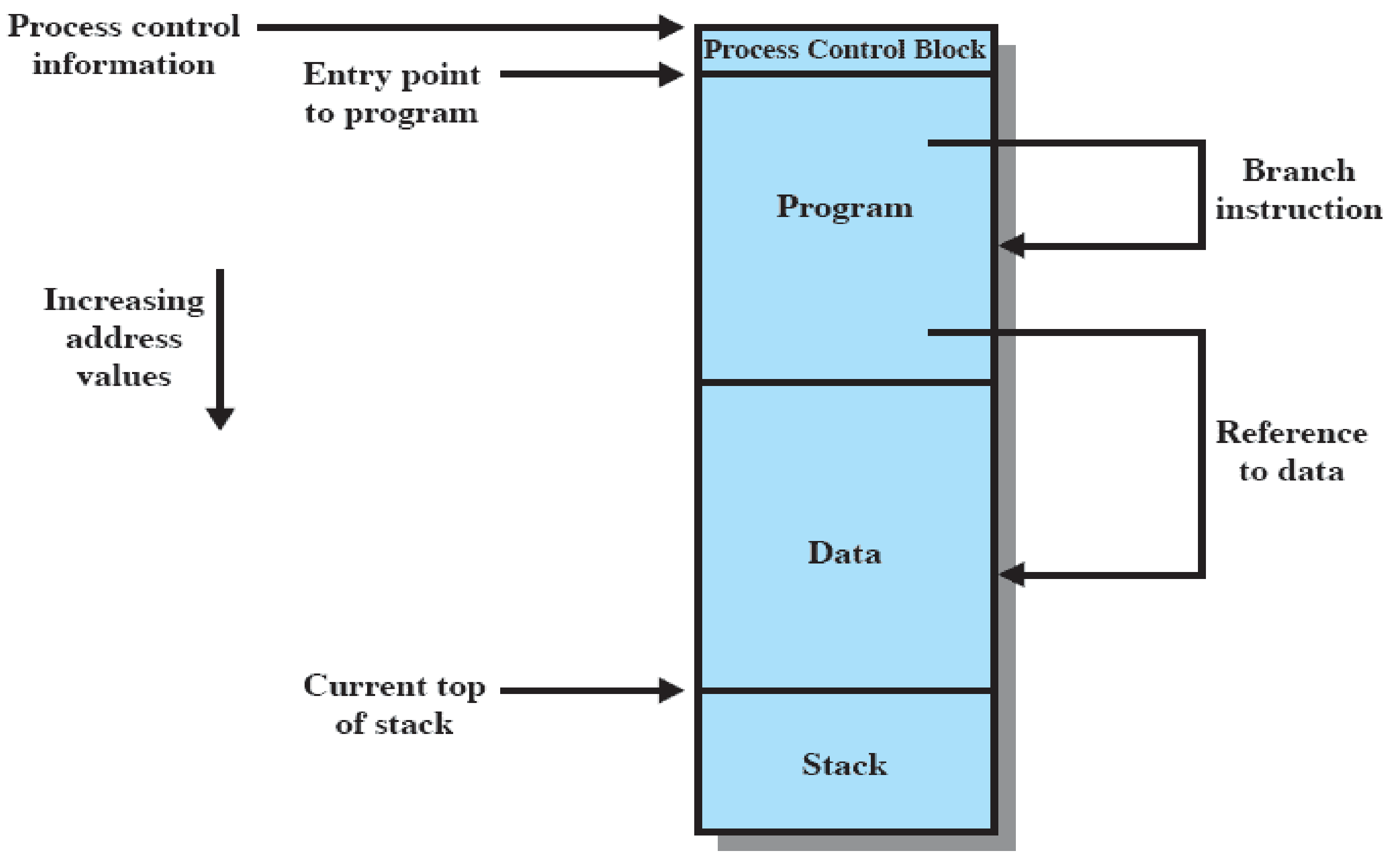


Figure 7.1 Addressing Requirements for a Process

Memory Management Requirements

- Process image occupies a contiguous region of main memory.
- OS needs to know the location of process control information and of execution task, as well as the entry point to begin execution of program for this process.
- **Branch instructions** contain an address to reference the instruction to be executed next.
- **Data reference** instructions contain the address of the byte or word of data referenced.
- Process hardware and OS software must be able to translate the memory references into actual physical memory addresses.

Memory Management Requirements

- Protection
  - processes should not be able to reference memory locations in another process without permission.
  - impossible to check addresses at compile time in programs since the program could be relocated.
  - address references must be checked at run time by hardware.

Memory Management Requirements

- Protection
  - Processes can refer only to the memory space allocated to those processes.
  - Program in one process cannot branch to an instruction in another process.
  - Program in one process cannot access data area of another process.
  - Memory protection requirement must be satisfied by processor hardware.

Memory Management Requirements

- Sharing
  - must allow several processes to access a common portion of main memory without compromising protection.
  - cooperating processes may need to share access to the same data structure.
  - better to allow each process to access the same copy of the program rather than have their own separate copy.

Memory Management Requirements

- Logical Organization
  - users write programs in modules with different characteristics :
    - instruction modules are execute-only
    - data modules are either read-only or read/write
    - some modules are private others are public
  - To effectively deal with user programs, the OS and hardware should support a basic form of module to provide the required protection and sharing.

Memory Management Requirements

- Logical Organization
  - Advantages:
    - Modules can be written and compiled independently.
    - Different degrees of protection(read only, execute only) can be given to different modules.
    - Possible to introduce mechanism by which modules can be shared among processes. User specifies sharing that is desired.

Memory Management Requirements

- Physical Organization
  - secondary memory is the long term store for programs and data while main memory holds program and data currently in use.
  - moving information between these two levels of memory is a major concern of memory management (OS)
    - it is highly inefficient to leave this responsibility to the application programmer.

Memory Management Requirements

- Physical Organization
  - Assigning responsibility to individual programmer is impractical and undesirable for two reasons:
    - The main memory available for a program plus its data may be insufficient.

      Solution: **Overlaying**=> Program and data are organized in such a way that various modules can be assigned same region of the memory., with a main program responsible for switching modules in and out as needed. But overlay programming wastes programmer time.
    - In multiprogramming environment, the programmer does not know at the time of coding how much space will be available or where that space will be.

Memory Partitioning

Introduction

- The main memory operation in memory management is to swap processes in and out of the main memory for processor to execute.
- In almost all of multiprog env. this involves the concept of virtual memory which is based on paging and segmentation.
- But before going into that we need to understand some simpler techniques.

Introduction

- But before going into that we need to understand some simpler techniques.
  - fixed partitioning
  - dynamic partitioning
  - paging
  - segmentation

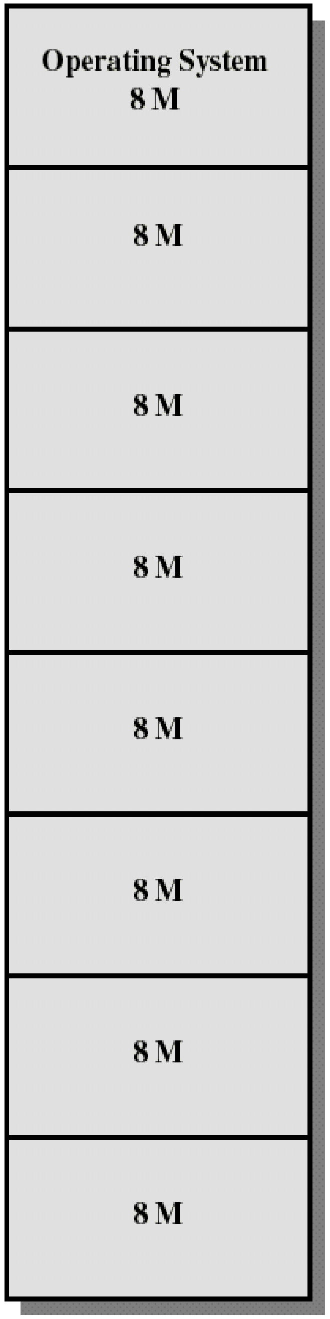
Memory Partitioning
Fixed Partitioning

Fixed Partitioning

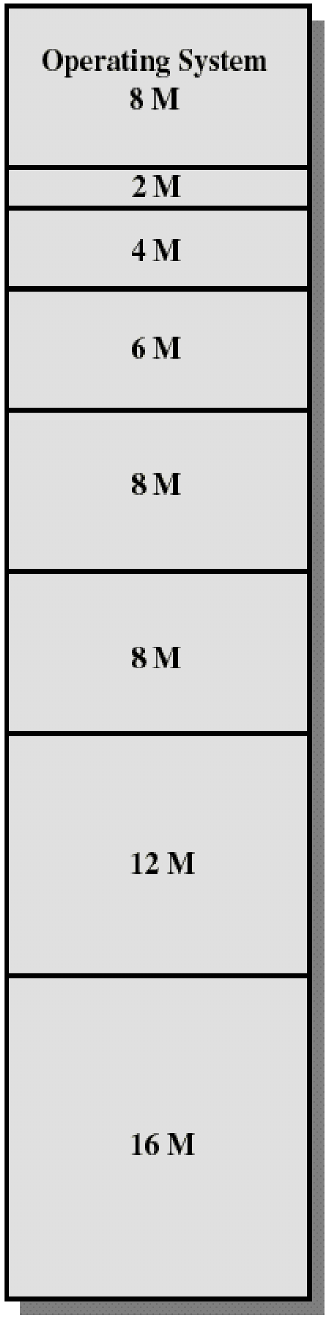
- In this one part of the main memory is dedicated to OS and other is available for remaining processes.
- Main memory is divided into fixed size memory blocks say 8Mbytes.
- So programs when loaded can be allocated to a particular block.
- When all blocks are full, the OS can swap one process with other as per its placement techniques.
- Examples of OS: An early IBM mainframe OS, OS/MFT

Fixed Partitioning

- Partition main memory into a set of non overlapping regions called partitions.
- Partitions can be of equal or unequal sizes



Equal-size partitions



Unequal-size partitions

Fixed Partitioning

- any process whose size is less than or equal to a partition size can be loaded into the partition
- if all partitions are occupied, the operating system can swap a process out of a partition

Fixed Partitioning

- Two difficulties with the use of **equal sized fixed partition :**
  - a program may be too large to fit in a partition. The programmer must then design the **program with overlays**.

    Overlaying: Keeping only some portion of the program in main memory and swapping as per requirement.
  - It creates **internal fragmentation.**

Fixed Partitioning

Internal Fragmentation

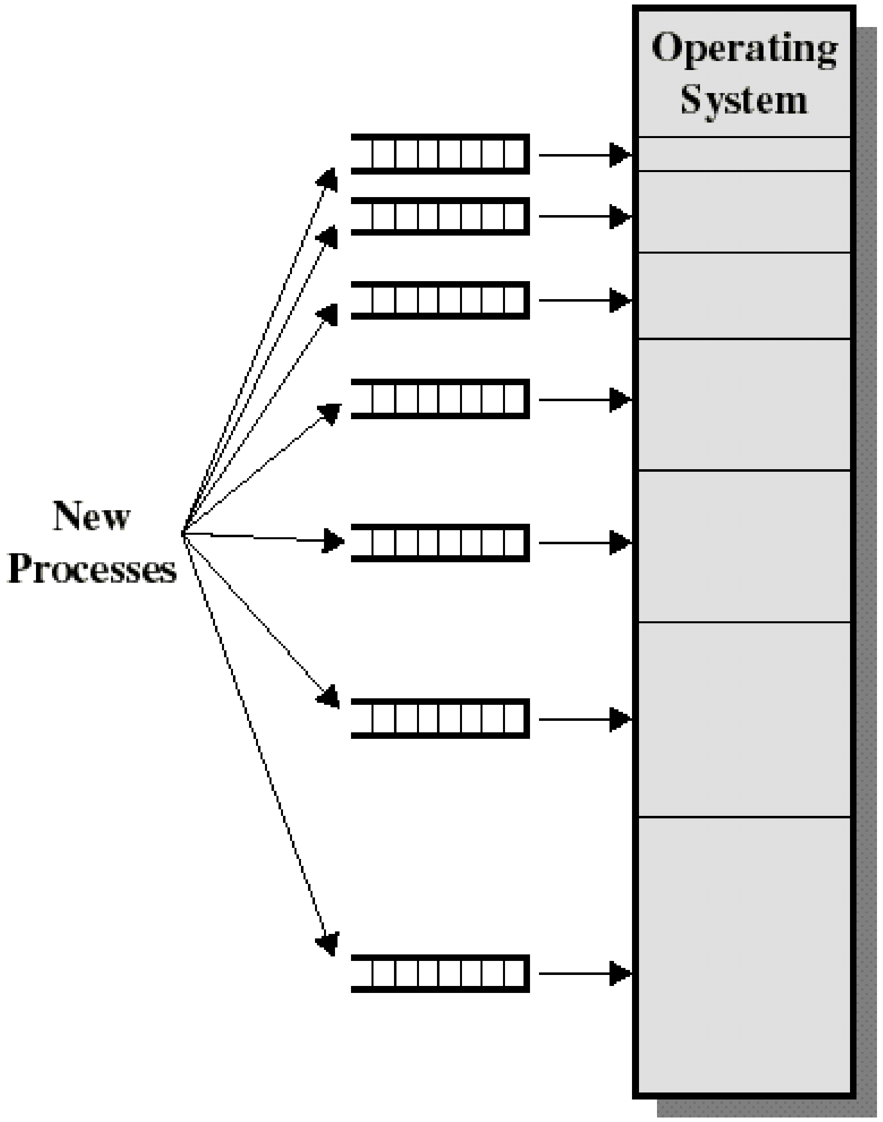
- The situation when there is wasted space inside a partition when the amount of data allocated in that partition is smaller than the partition size
- For e.g. The size of partition is 8Mbytes and the program is of size 2Mbytes but still it is given the whole 8Mbytes block in this technique.
- Unequal-size partitions lessens these problems but they still remain…

Placement Algorithm with Partitions

- Unequal-size partitions:
  - Two ways to assign processes to partitions.
    - Use of multiple queues
    - Use of single queue

Placement Algorithm with Partitions

- Unequal-size partitions: use of multiple queues
  - assign each process to the smallest partition within which it will fit
  - A queue for each partition size
  - tries to minimize internal fragmentation
  - Problem: some queues will be empty if no processes within a size range is present

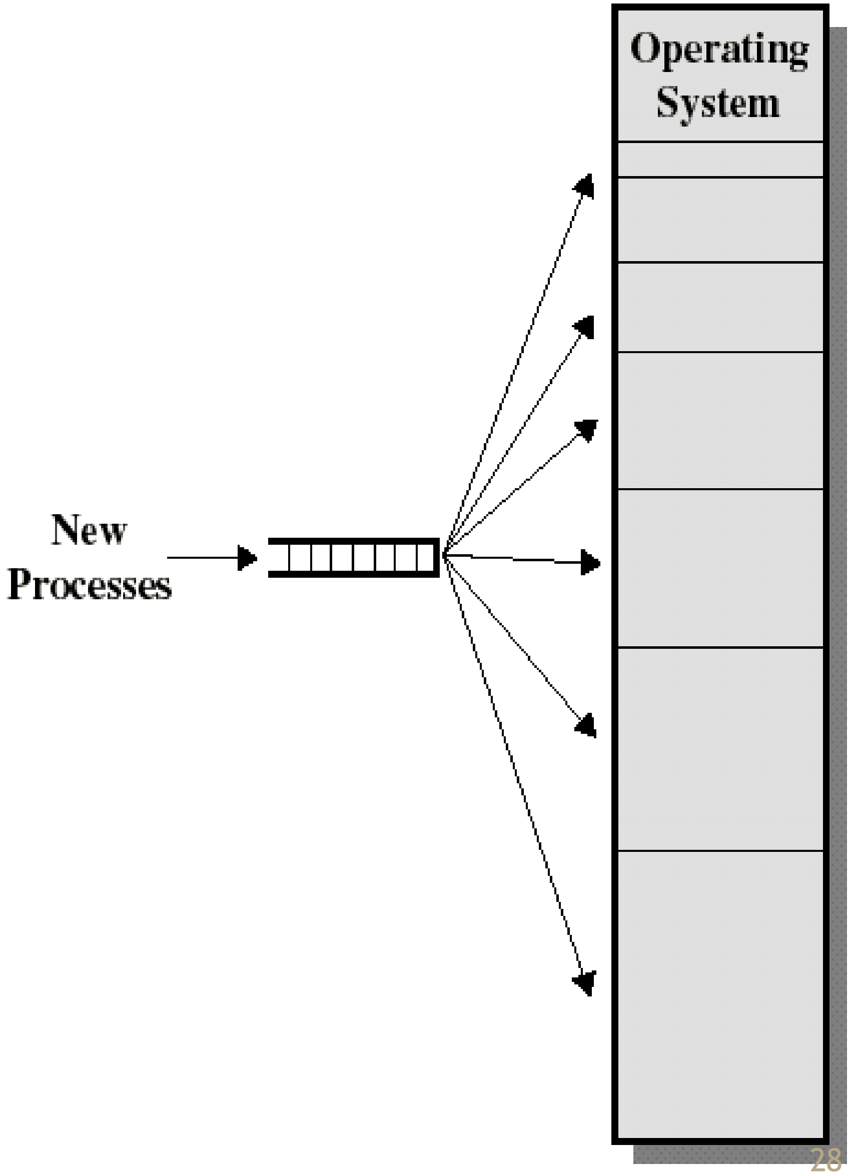


Placement Algorithm with Partitions

- Unequal-size partitions:

use of a single queue

- When its time to load a process into main memory the smallest available partition that will hold the process is selected.
- increases the level of multiprogramming at the expense of internal fragmentation.
- Swapping out of the smallest partition.



Placement Algorithm with Partitions

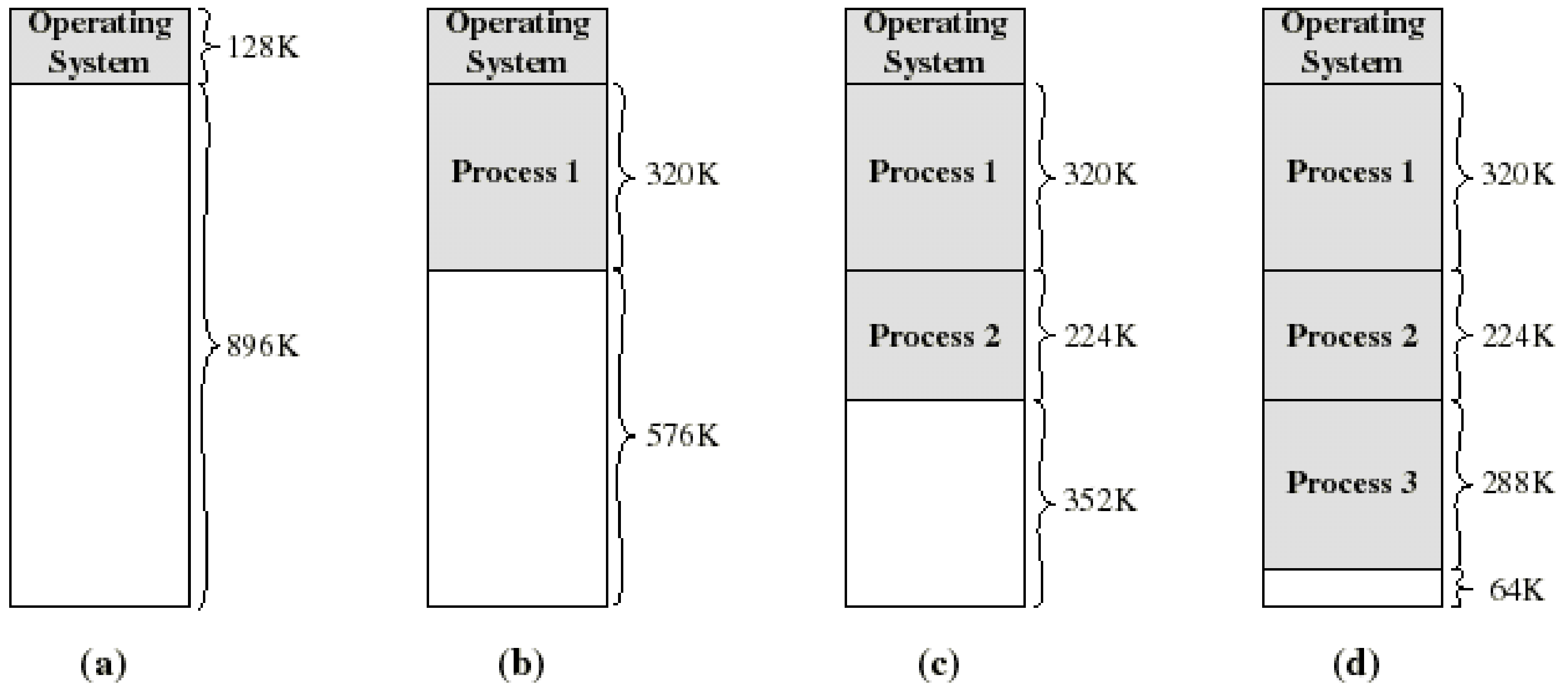
- Disadvantages of Unequal partition:
  - Number of partitions specified limits the number of processes
  - Partition sizes are preset at system generation time. So, small job will not utilize partition space efficiently.

Memory Partitioning
Dynamic Partitioning

Dynamic Partitioning

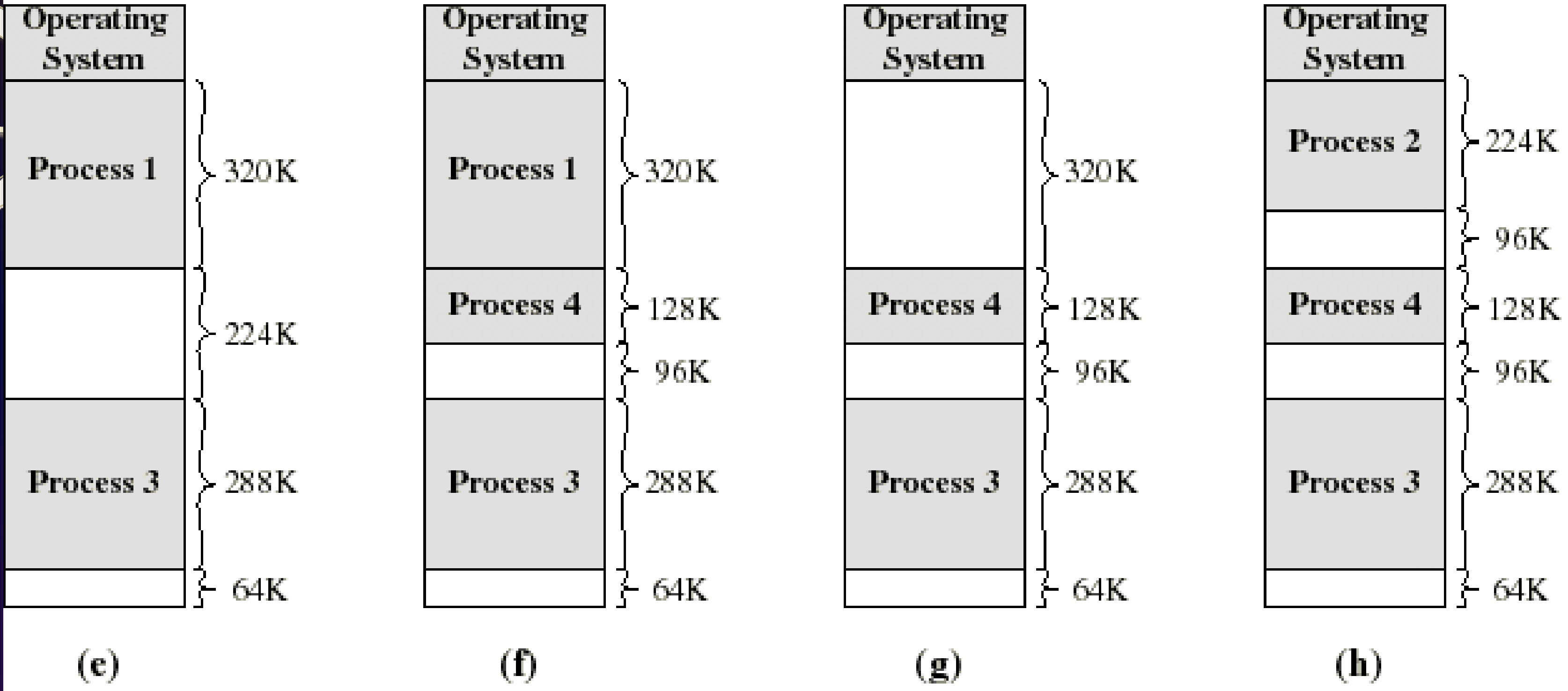
- Partitions are of variable length and number.
- Each process is allocated exactly as much memory as it requires.
- When a new process is there and memory available is not enough, swapping is done with another process.
- Used in IBM's OS/MVT (Multiprogramming with a Variable number of Tasks)

Dynamic Partitioning: an example



- A hole of 64K is left after loading 3 processes: not enough room for another process
- Eventually each process is blocked.The OS swaps out process 2 to bring in process 4

Dynamic Partitioning: an example



- another hole of 96K is created
- Eventually each process is blocked.The OS swaps out process 1 to bring in again process 2 and another hole of 96K is created...
- Compaction would produce a single hole of 256K

Dynamic Partitioning

- External Fragmentation:
  - The situation in which various small sized partitions are created is known as External Fragmentation.
  - External because the memory that is fragmented is not allocated to process, it is in between the various process allocated partitions opposite to that of internal fragmentation.

Dynamic Partitioning

Compaction

- Technique to resolve external fragmentation
- OS shifts processes so they are contiguous and all free memory is in one block.
- Problem is that compaction is time consuming and thus wastes processor's time.

Placement Algorithm

- Used to decide which free block to allocate to a process.
- Goal: to reduce usage of compaction (time consuming)

Possible algorithms:

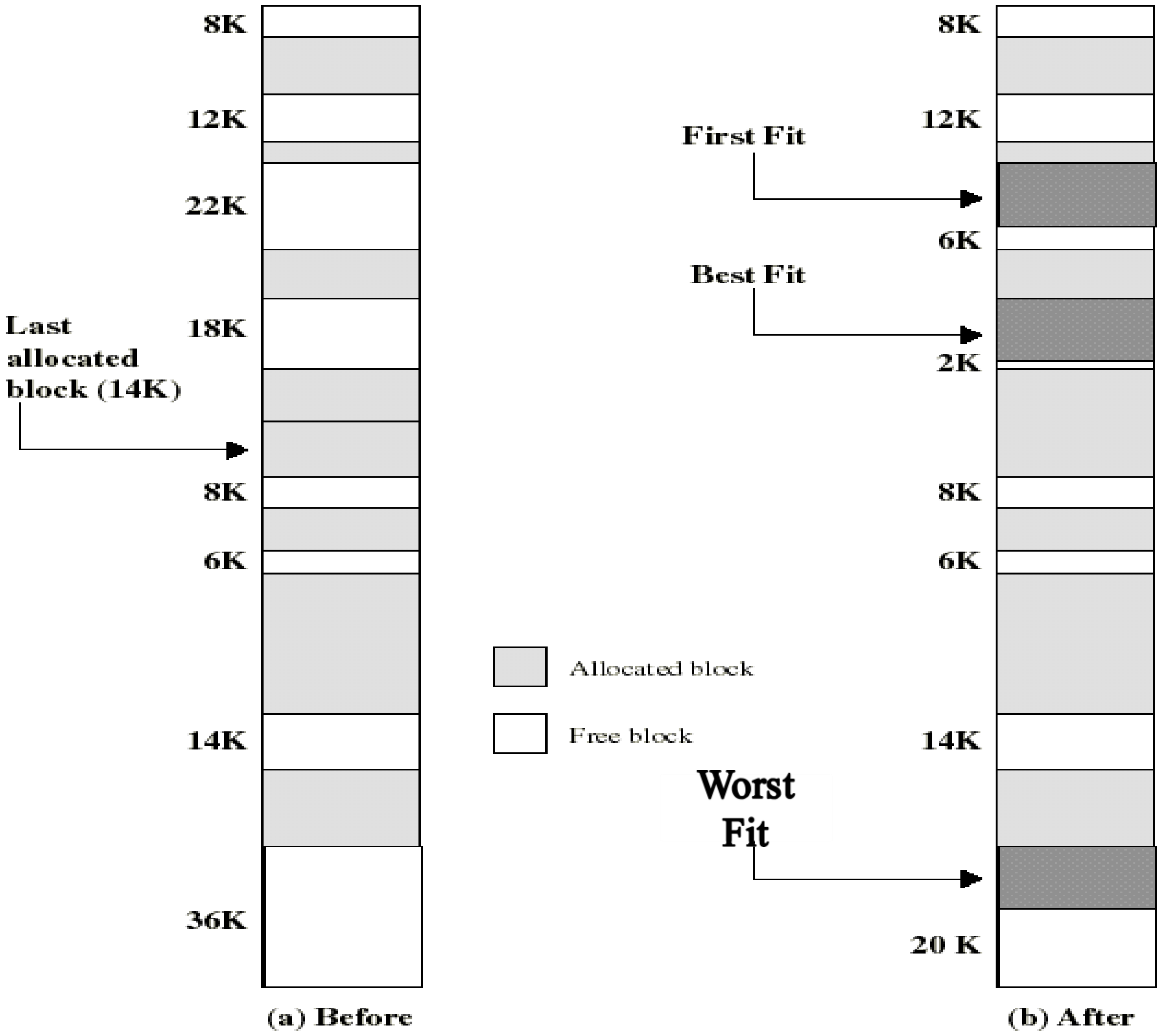
- Best-fit: Allocate the smallest hole that is big enough. We must search the entire list, unless the list is ordered by size. This strategy produces the smallest leftover hole.

Placement Algorithm

- 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.

- Worst-fit: Allocate the largest hole. Again, we must search the entire list, unless it is sorted by size. This strategy produces the largest leftover hole, which may be more useful than the smaller leftover hole from a best-fit approach.

Placement Algorithm

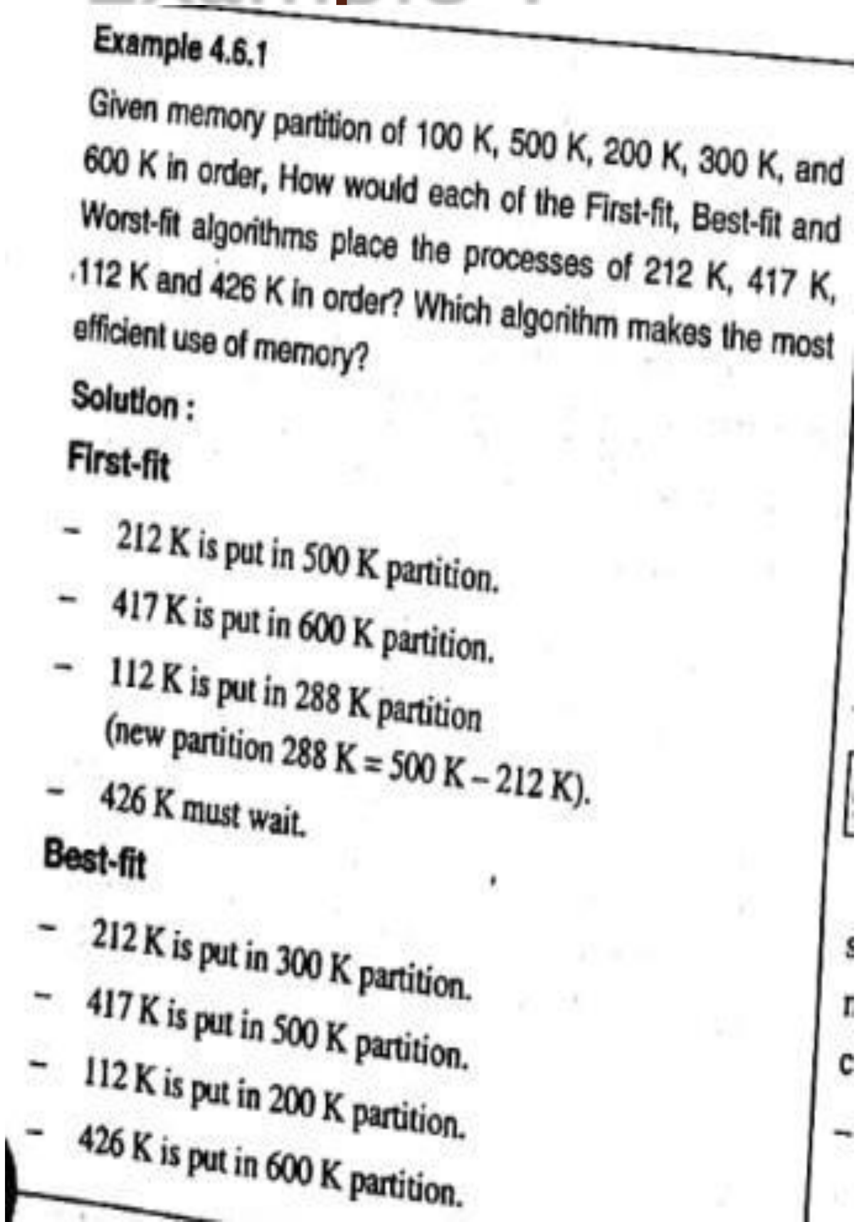


Example Memory Configuration Before and After Allocation of 16 Kbyte Block

Placement Algorithm: comments

- Worst-fit often leads to allocation of the largest block at the end of memory
- First-fit favors allocation near the beginning: tends to create less fragmentation than Next-fit
- Best-fit searches for smallest block: the fragment left behind is small as possible
  - main memory quickly forms holes too small to hold any process: compaction generally needs to be done more often

Example 1



Example 4.6.1

Given memory partition of 100 K, 500 K, 200 K, 300 K, and 600 K in order, How would each of the First-fit, Best-fit and Worst-fit algorithms place the processes of 212 K, 417 K, 112 K and 426 K in order? Which algorithm makes the most efficient use of memory?

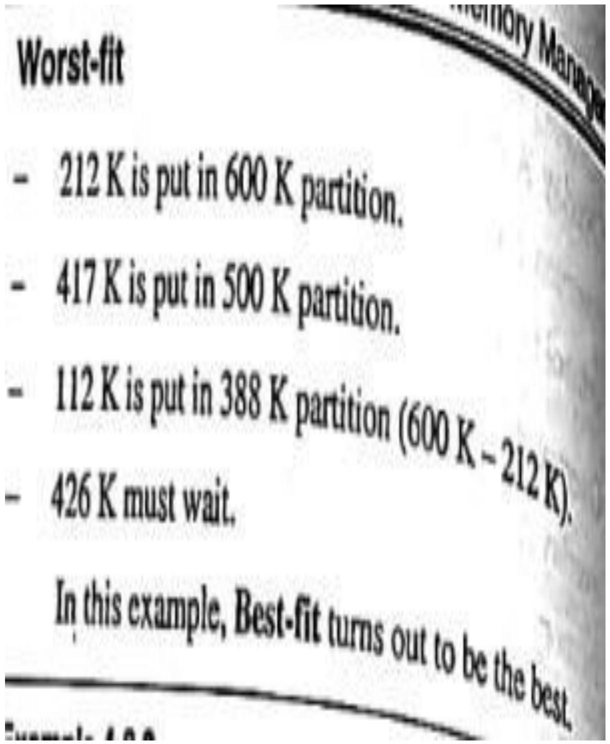
Solution :

First-fit

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

Best-fit

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

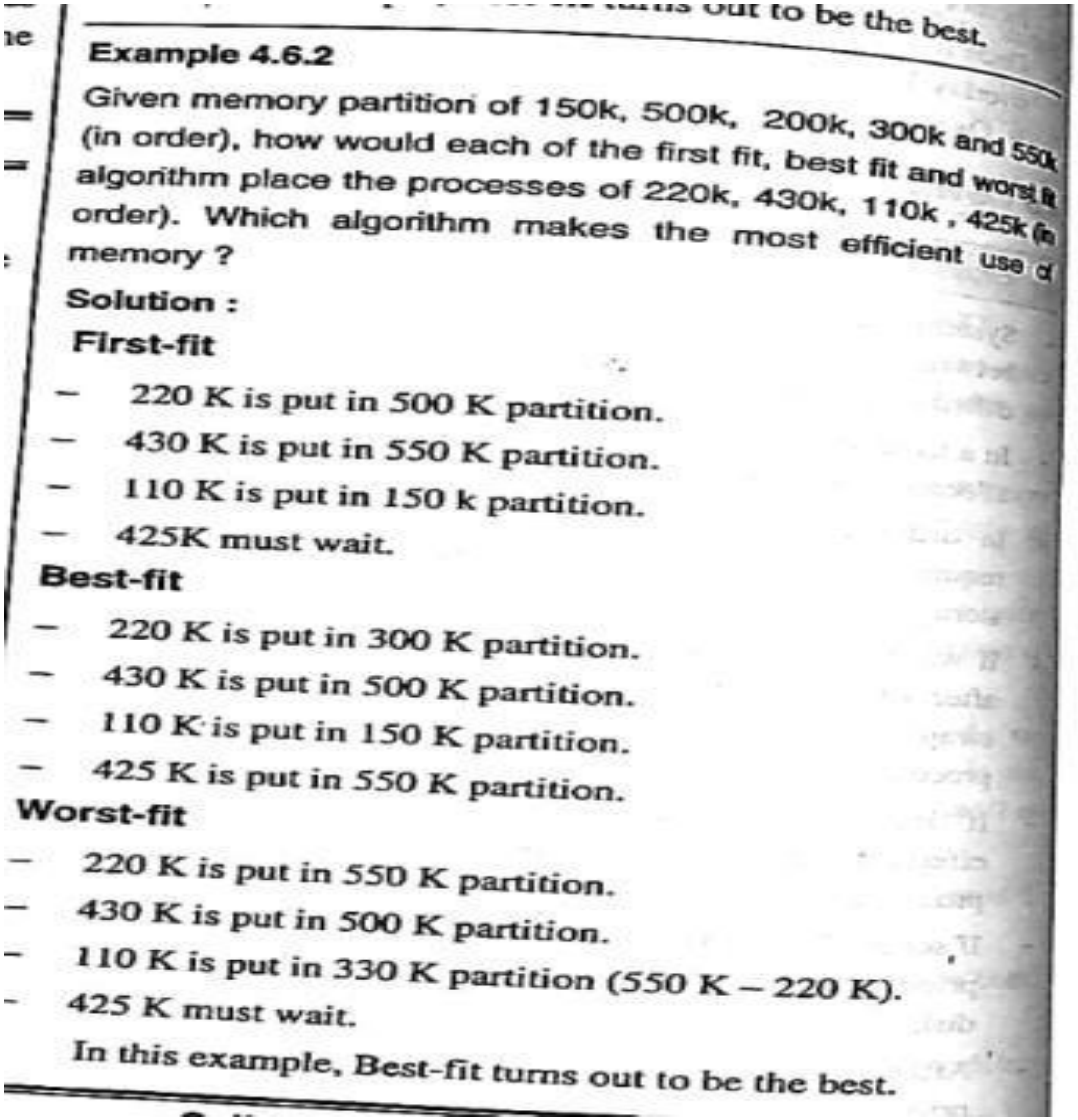


Worst-fit

- 212 K is put in 600 K partition.
- 417 K is put in 500 K partition.
- 112 K is put in 388 K partition (600 K – 212 K).
- 426 K must wait.

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

Example 2



Example 4.6.2

Given memory partition of 150k, 500k, 200k, 300k and 550k (in order), how would each of the first fit, best fit and worst fit algorithm place the processes of 220k, 430k, 110k , 425k (in order). Which algorithm makes the most efficient use of memory ?

Solution :

First-fit

- 220 K is put in 500 K partition.
- 430 K is put in 550 K partition.
- 110 K is put in 150 k partition.
- 425K must wait.

Best-fit

- 220 K is put in 300 K partition.
- 430 K is put in 500 K partition.
- 110 K is put in 150 K partition.
- 425 K is put in 550 K partition.

Worst-fit

- 220 K is put in 550 K partition.
- 430 K is put in 500 K partition.
- 110 K is put in 330 K partition (550 K – 220 K).
- 425 K must wait.

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

Address Types

- A physical address (absolute address) is a physical location in main memory. It is an address seen by Memory unit i.e. the one loaded into the memory address register.
- A logical address: Is an address generated by CPU. It is a reference to a memory location independent of the physical structure/organization of memory.
- Compilers produce code in which all memory references are logical addresses

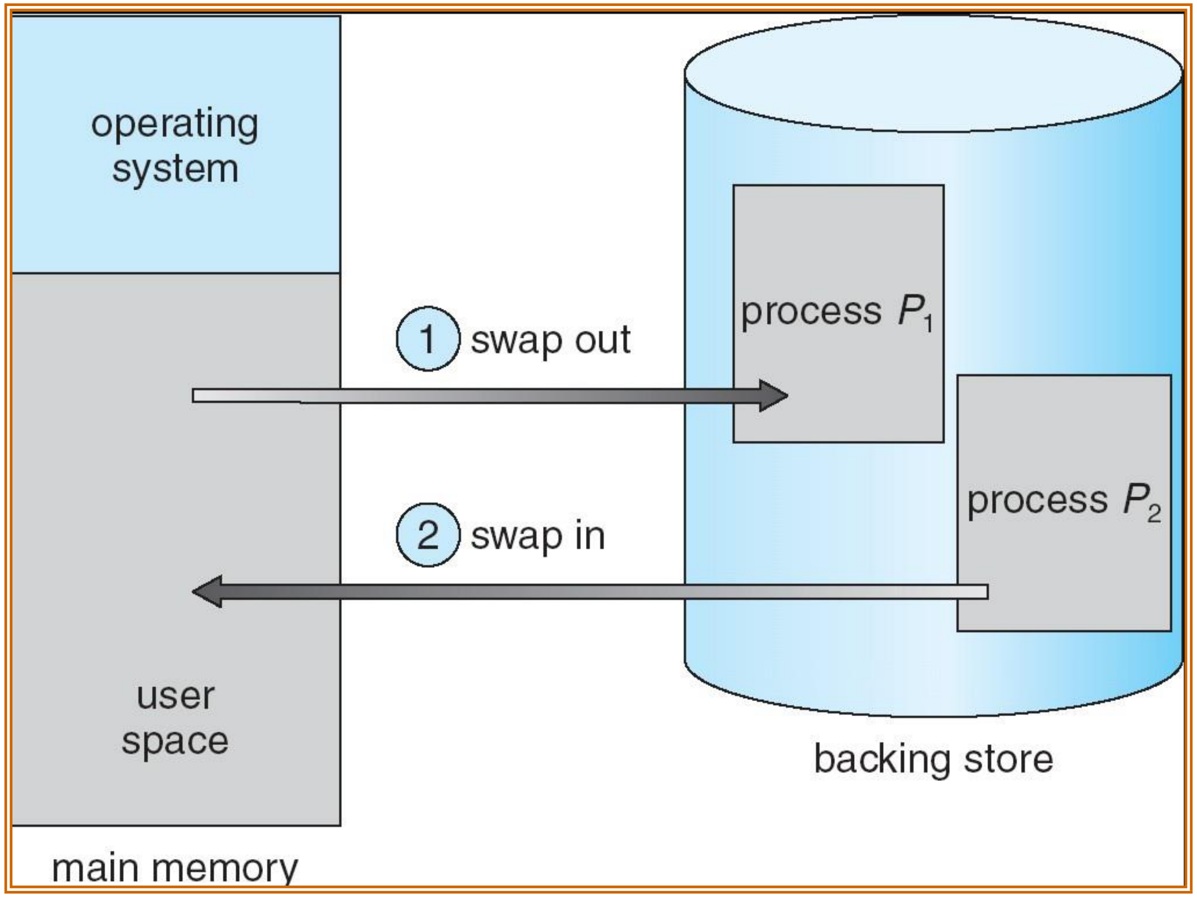
Memory-Management Unit (MMU)

- 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

Swapping

- A process can be swapped temporarily out of memory to a backing store, and then brought back into memory for continued execution
- **Backing store** – 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, Linux, and Windows)
- System maintains a **ready queue** of ready-to-run processes which have memory images on disk

Schematic View of Swapping



Paging

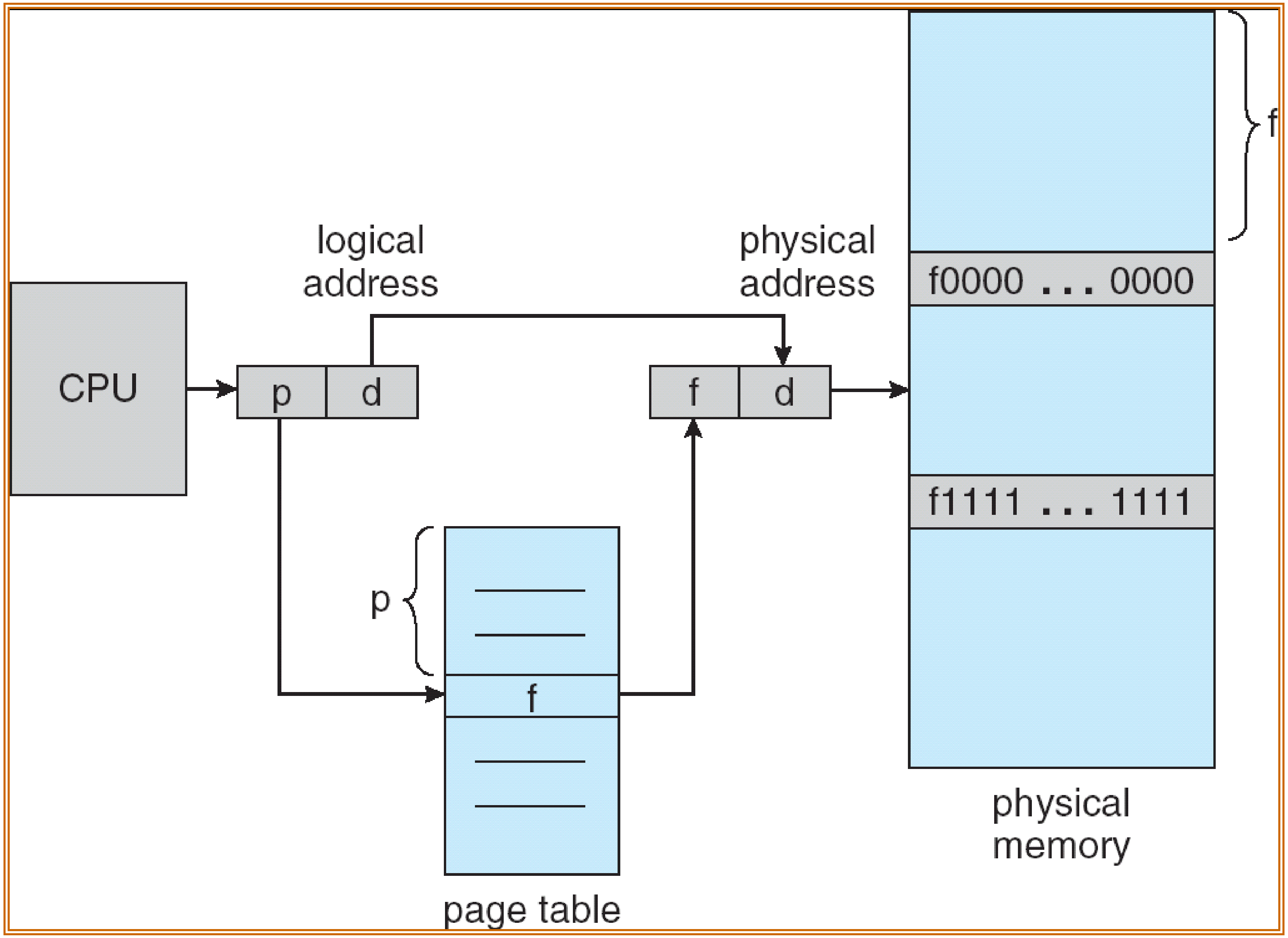
Paging

- Paging is a memory-management scheme that permits the physical address space a process to be noncontiguous.
- Paging avoids external fragmentation and the need for compaction.
- It also solves the considerable problem of fitting memory chunks of varying sizes onto the backing store.
- Paging is implemented by closely integrating the hardware and operating system, especially on 64-bit microprocessors.

Paging

- Divide physical memory into fixed-sized blocks called **frames** (size is power of 2, between 512 bytes and 8,192 bytes)
- Divide logical memory into blocks of 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 Hardware

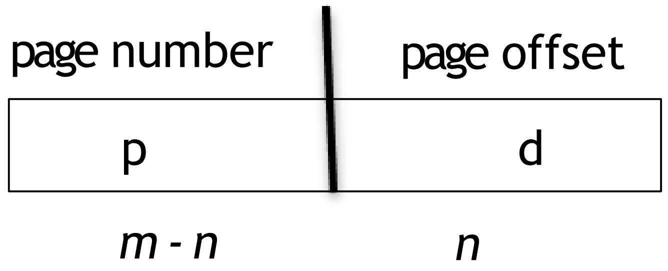


Paging

- Every address generated by the CPU is divided into two parts: a Page Number (p) and a Page Offset.
- 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.

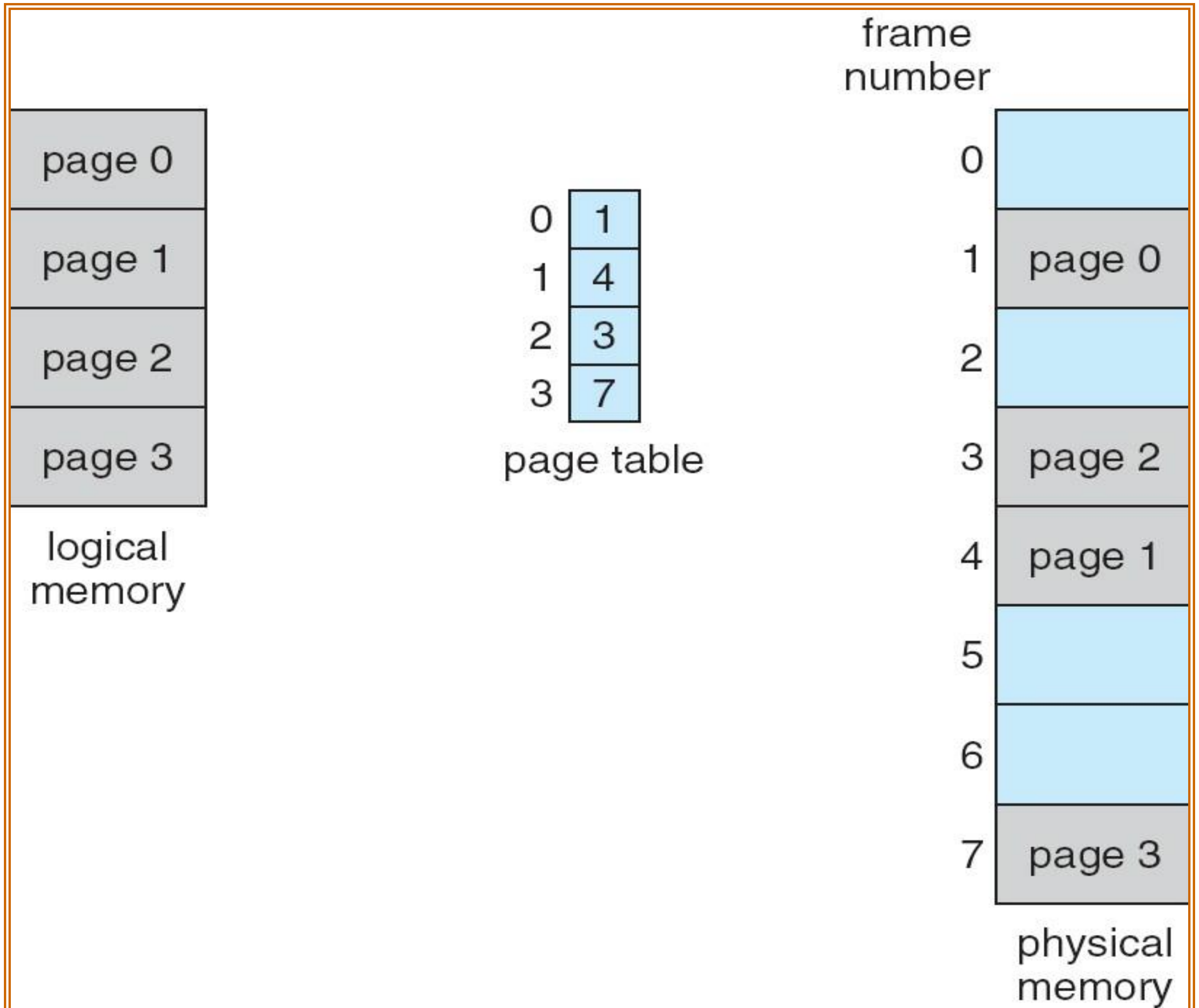
Address Translation Scheme

- 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.
  - The logical address is as follow. p is an index into the page table and d is the displacement within the page.



  - For given logical address space 2^m and page size 2^n.

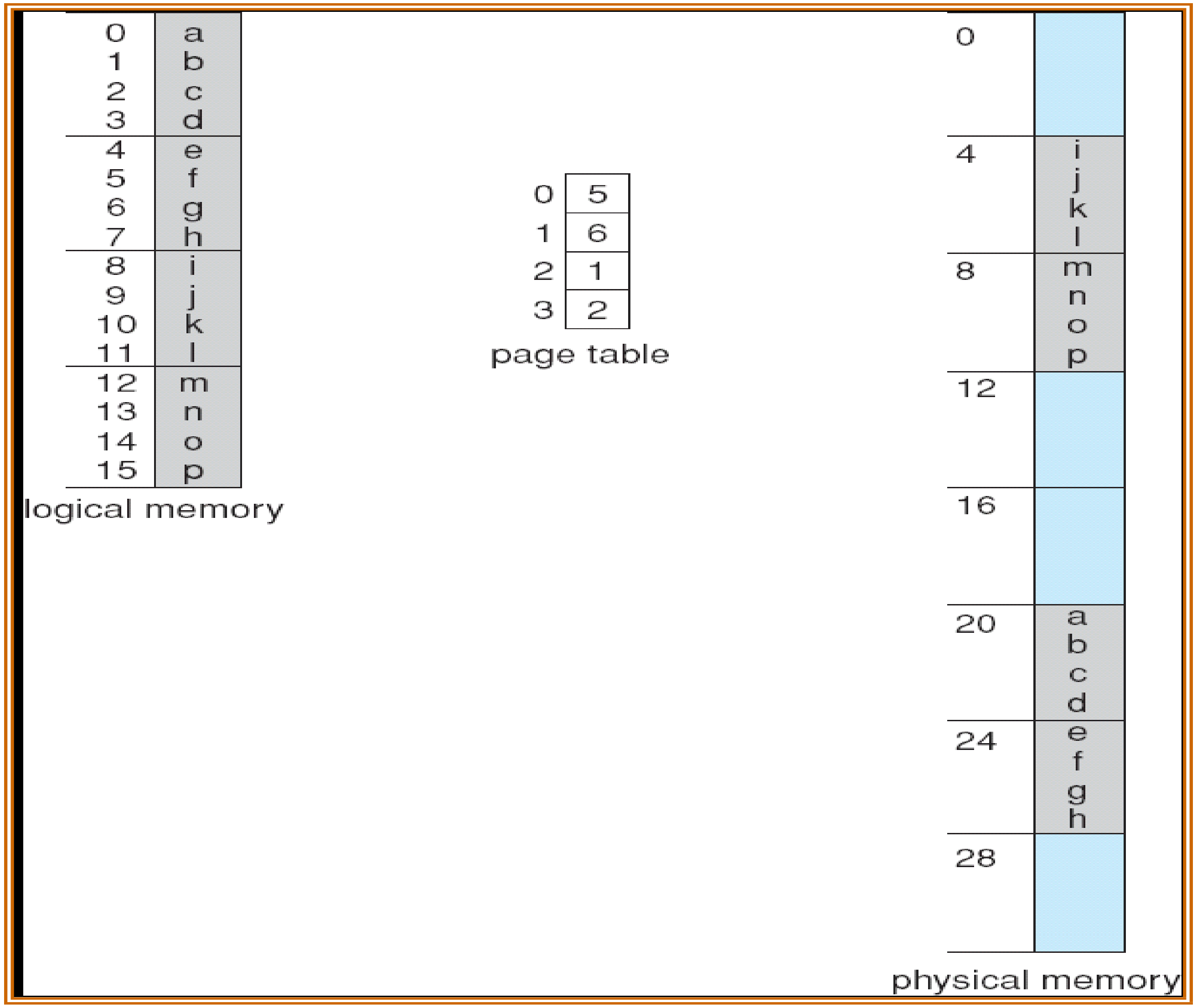
Paging Model of Logical and Physical Memory



Paging

- 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.
- 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) 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.

Paging Example



32-byte memory and 4-byte pages

Paging Example

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

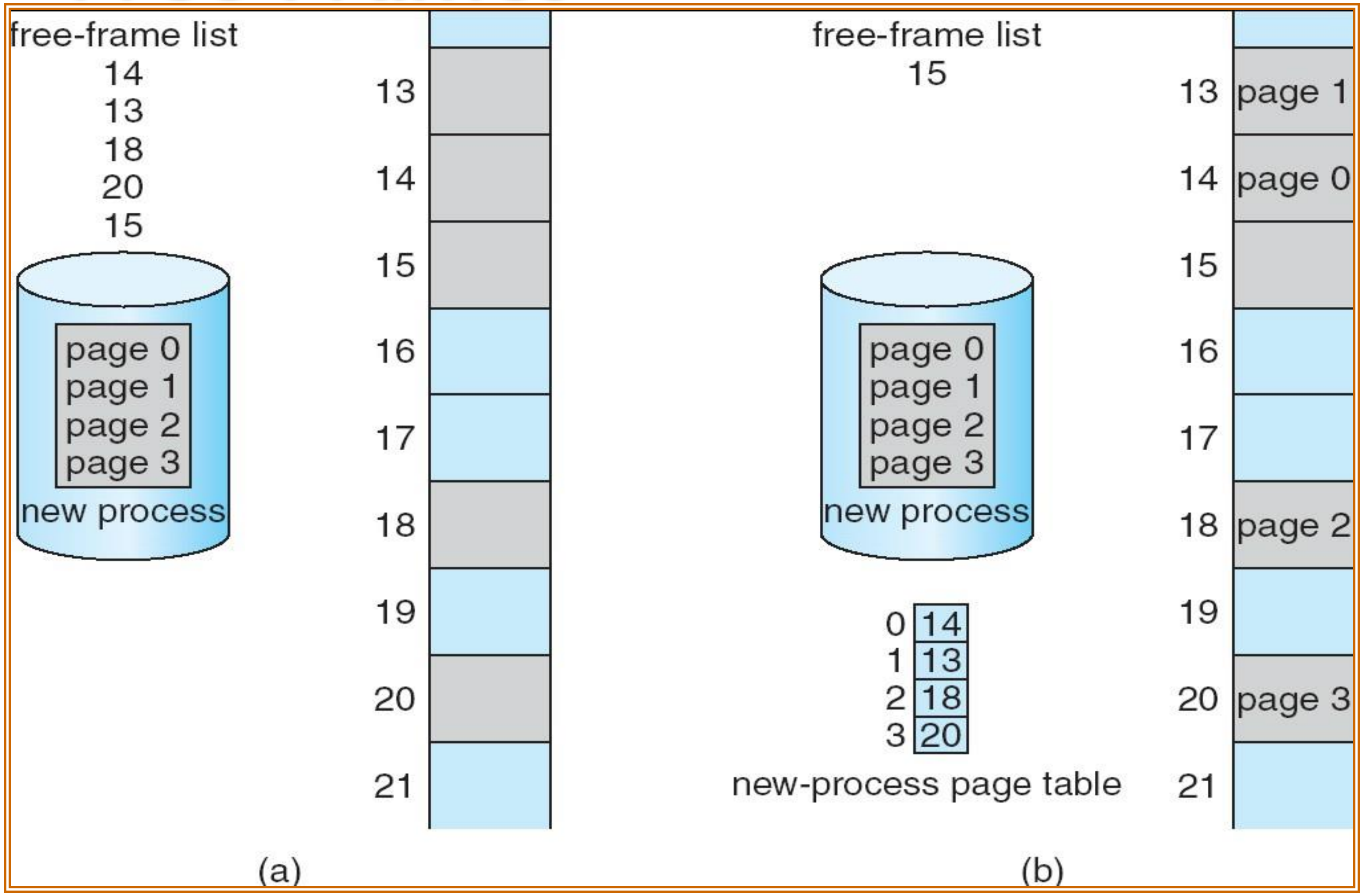
- Logical address 0 is page 0, offset 0.

- 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].

- Formula=>
- Physical address=(Frame No*Page size)+Offset

Free Frames



Before allocation

After allocation

ADVANTAGES

- No external Fragmentation
- Simple memory management algorithm
- Swapping is easy (Equal sized Pages and Page Frames)

DISADVANTAGES

- Internal fragmentation
- Page tables may consume more memory.
- Multi level paging leads to memory reference overhead.

Segmentation

Segmentation

- Memory-management scheme that supports user view of memory
- A program is a collection of segments. A segment is a logical unit such as:

main program,

procedure,

function,

method,

object,

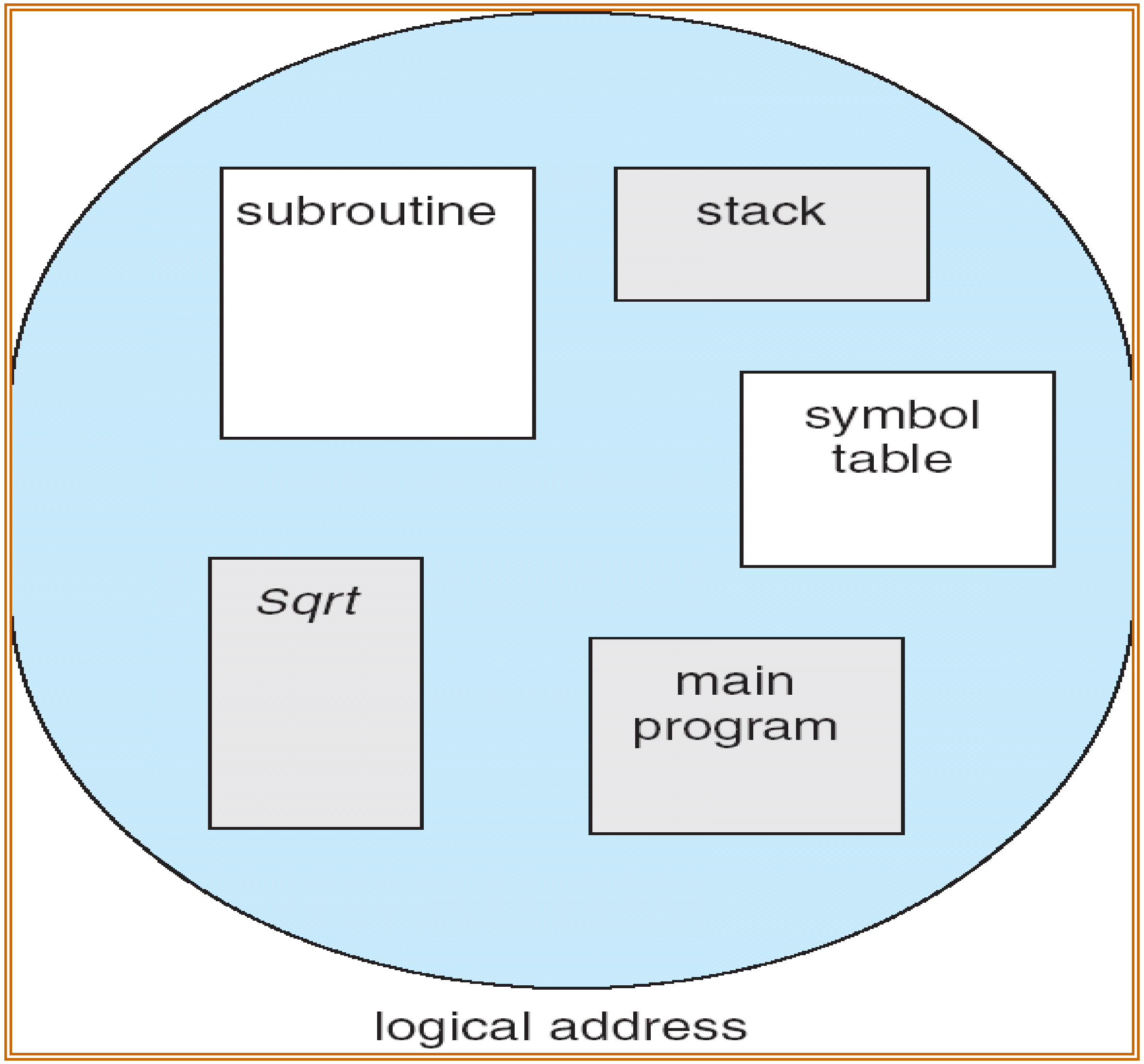
local variables, global variables,

common block,

stack,

symbol table, arrays

User's View of a Program



Segmentation

- 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.
- 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*:

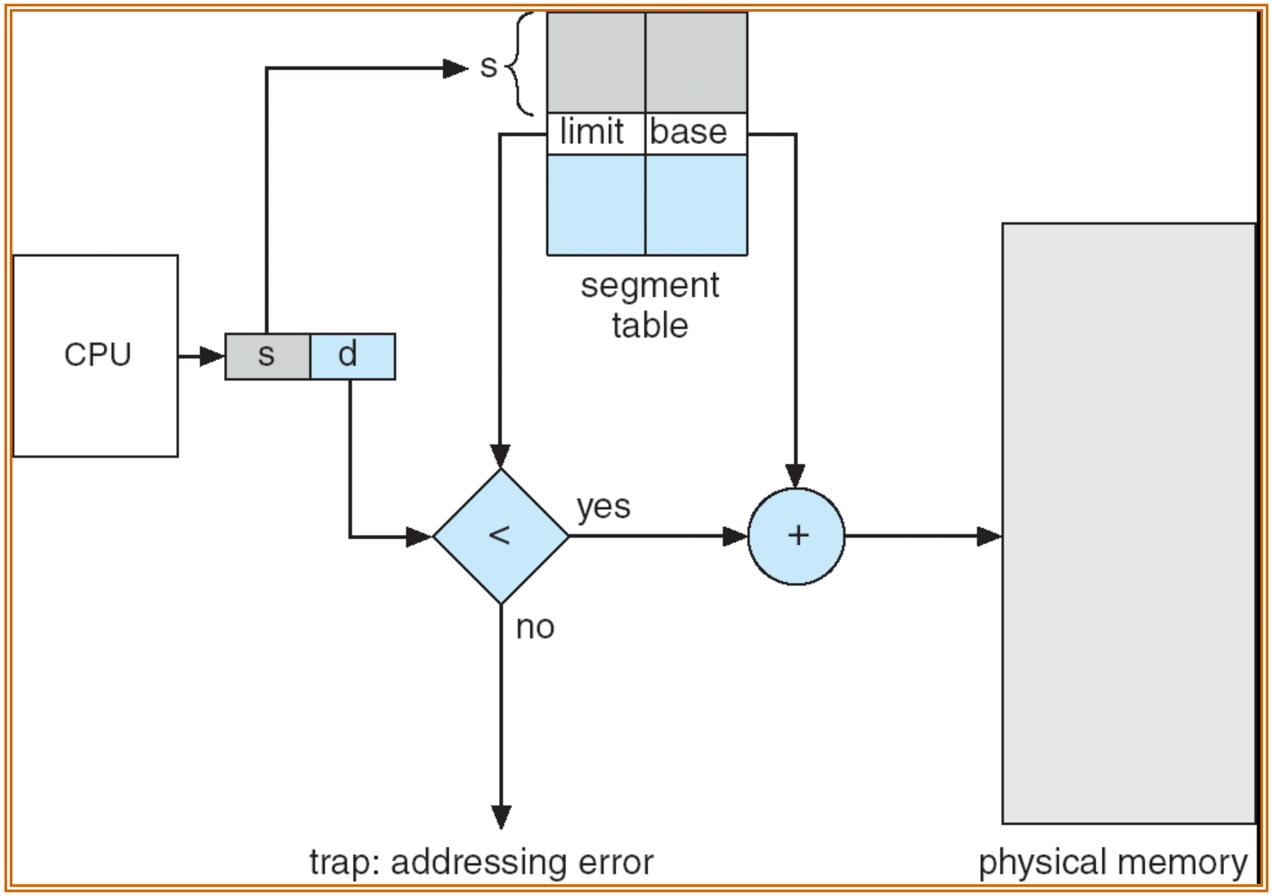
<segment-number, offset>.

- the user program is compiled, and the compiler automatically constructs segments reflecting the input program.

Segmentation Hardware

- map two dimensional user-defined addresses into one-dimensional physical addresses.
- This mapping is effected by a Segment table.
- 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, and the segment limit specifies the length of the segment.

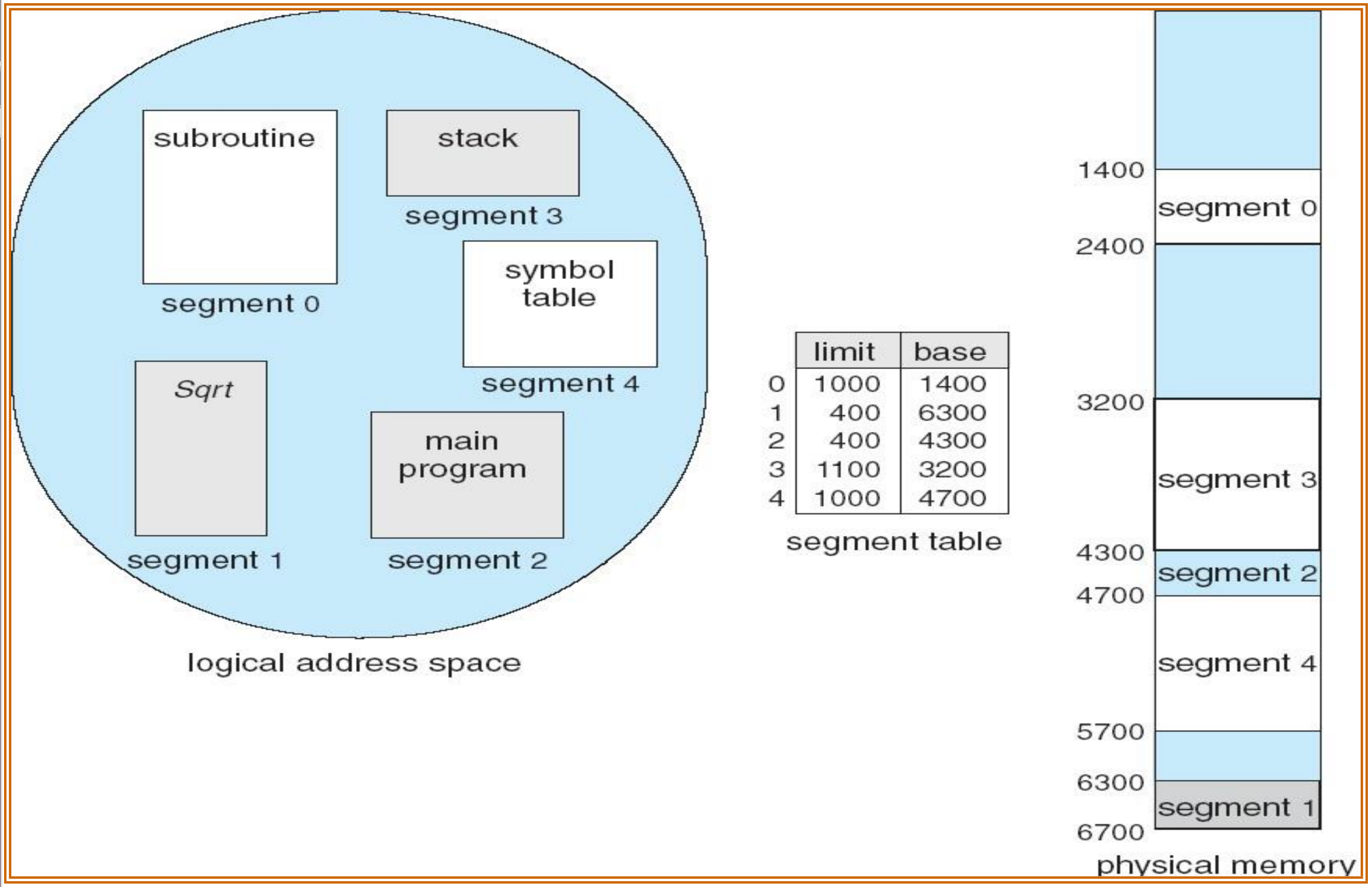
Segmentation Hardware



Segmentation Hardware

- logical address consists of two parts: a segment number, s, and an offset into that segment, *d.*
- The segment number is used as an index to the segment table.
- 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

Example of Segmentation



	limit	base
0	1000	1400
1	400	6300
2	400	4300
3	1100	3200
4	1000	4700

Example

Example:

segment 2=> 400 bytes long and begins at location 4300.

Offset = 53 =>mapped to =>4300+53=4353bytes

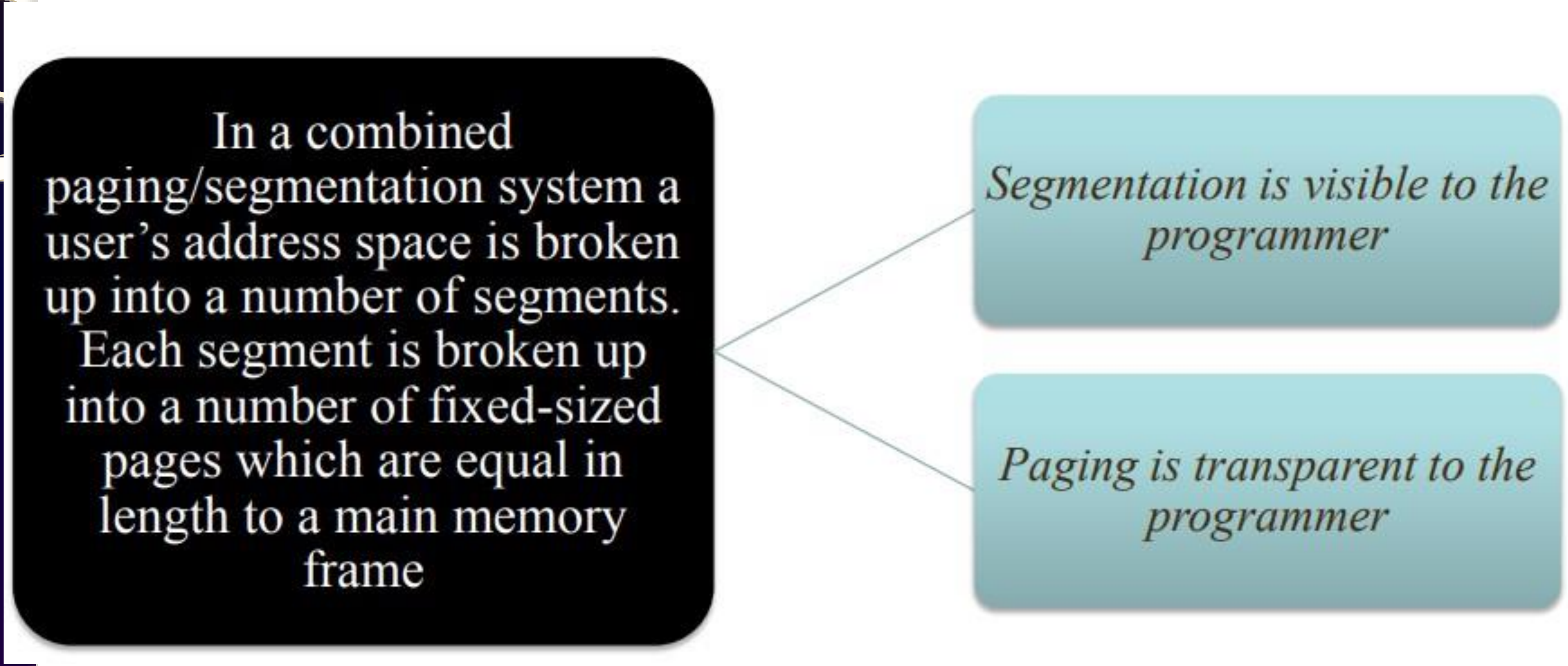
Example:

segment 0=> 1000 bytes long and begins at location 1400.

Offset = 1222 => results in trap as this segment is 1000bytes long only.

Segmented Paging

Segmented Paging



In a combined paging/segmentation system a user's address space is broken up into a number of segments. Each segment is broken up into a number of fixed-sized pages which are equal in length to a main memory frame

Segmentation is visible to the programmer

Paging is transparent to the programmer

Segmented Paging

- Pure segmentation is not very popular and not being used in many of the operating systems.
- However, Segmentation can be combined with Paging to get the best features out of both the techniques.

Segmented Paging

In Segmented Paging, the main memory is divided into variable size segments which are further divided into fixed size pages.

- Pages are smaller than segments.
- Each Segment has a page table which means every program has multiple page tables.
- The logical address is represented as Segment Number (base address), Page number and page offset.

Virtual Memory Management

Virtual memory

- Virtual memory allows the execution of processes that are not completely in memory.
- Only part of the program needs to be in memory for execution.
- Advantage: programs can be larger than physical memory.
- Virtual memory allows processes to share files easily and to implement shared memory.
- Provides an efficientmechanismfor process creation.

Virtual memory

In many cases, the entire program is not needed. For instance, consider the following:

- Programs often have code to handle unusual error conditions. Since these errors seldom, if ever, occur in practice, this code is almost never executed.
- Arrays, lists, and tables are often allocated more memory than they actually need. An array may be declared 100 by 100 elements, even though it is seldom larger than 10 by 10 elements.
- Certain options and features of a program may be used rarely. For instance, the routines on U.S. government computers that balance the budget have not been used in many years.

Virtual memory

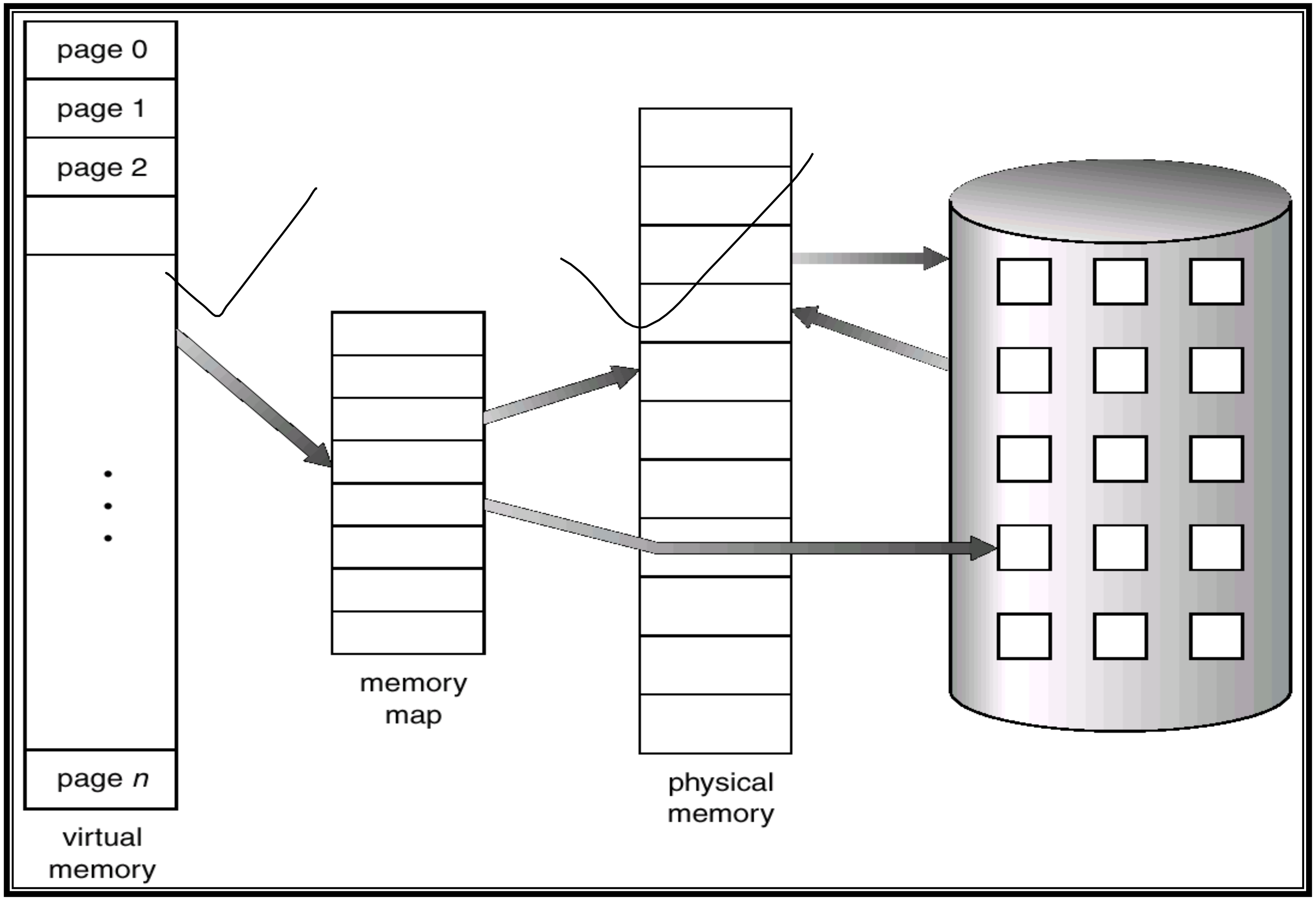
The ability to execute a program that is only partially in memory would confer many benefits:

- Users would be able to write programs for an extremely large *virtual address space*.
- Because each user program could take less physical memory, more programs could be run at the same time.
- Less I/O would be needed to load or swap user programs into memory, so
- each user program would run faster.

Virtual memory

- 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.

Virtual Memory That is Larger Than Physical Memory

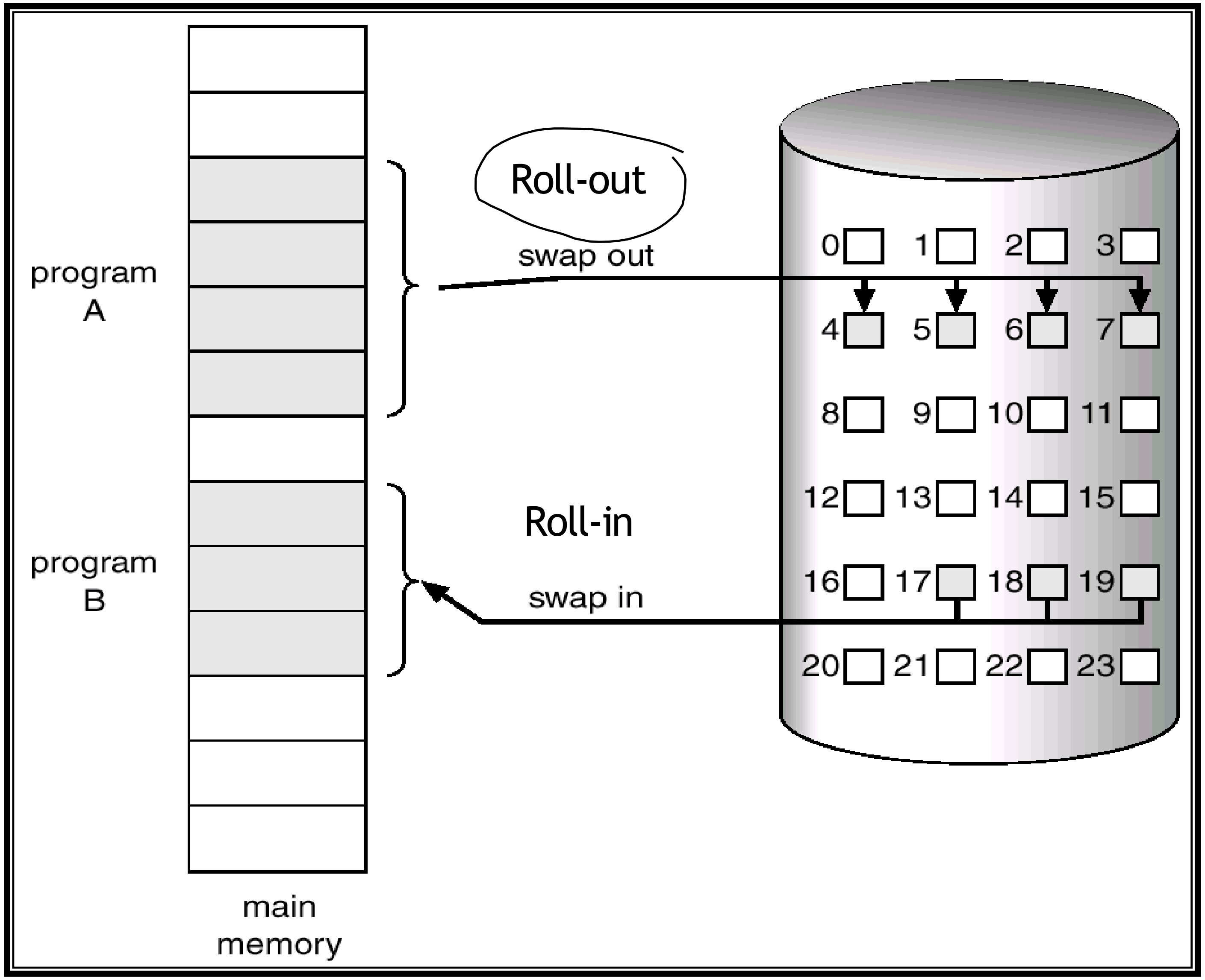


Implementation of virtual memory using Demand Paging

Demand Paging

- Bring a page into memory only when it is needed.
  - Less I/O needed
  - Less memory needed
  - Faster response
  - More users
- **Lazy swapper -** never swaps a page into memory unless page will be needed.
- Swapper that deals with pages is a **pager.**
- Page is needed ⇒ reference to it
  - invalid reference ⇒ abort
  - not-in-memory ⇒ bring to memory

Transfer of a Paged Memory to Contiguous Disk Space



Valid-Invalid Bit

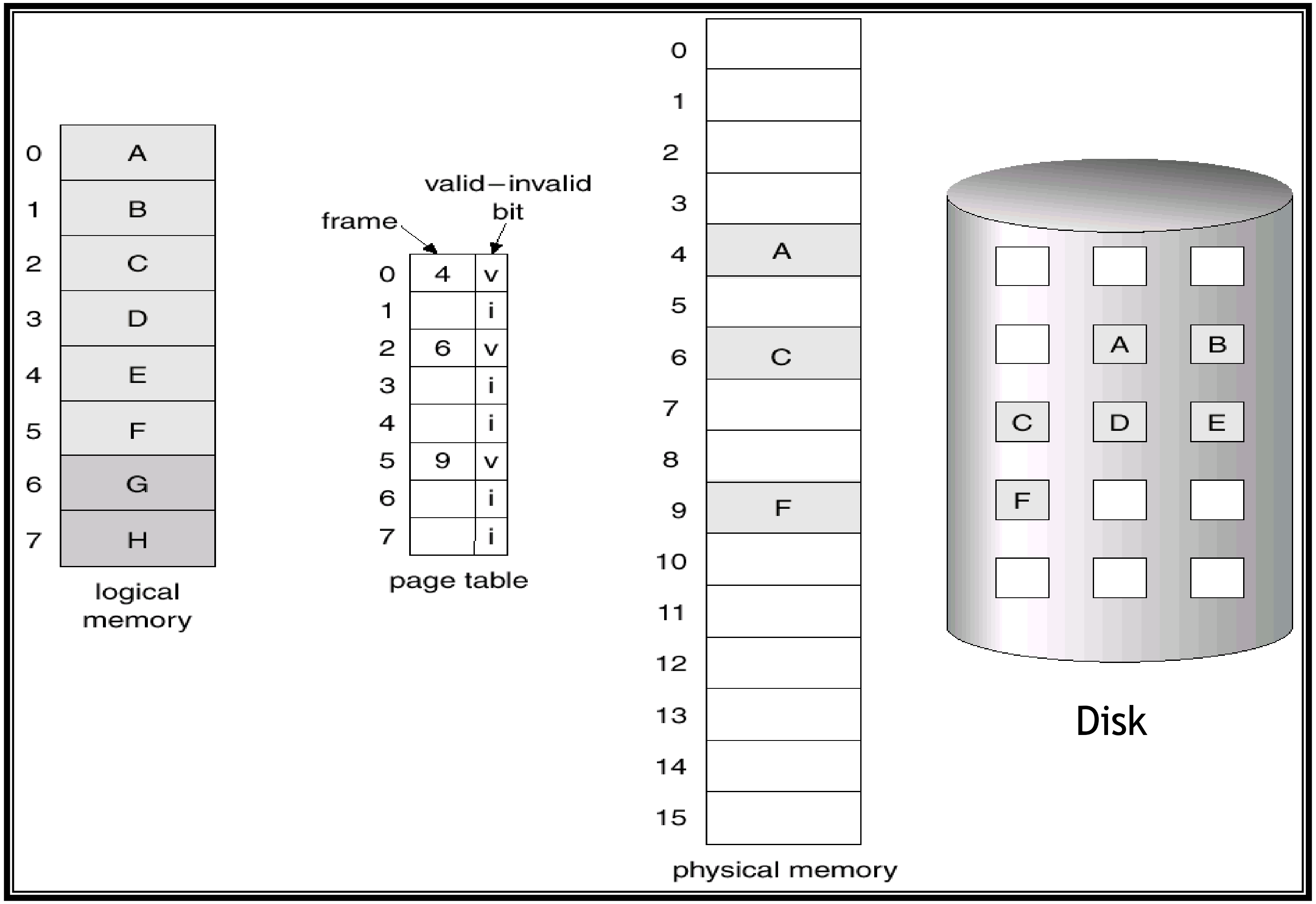
- With each page table entry a valid–invalid bit is associated (1 / v$\Rightarrow$ in-memory, 0/i $\Rightarrow$ not-in-memory)
- Initially valid–invalid but is set to 0 on all entries.
- Example of a page table snapshot.

Frame #	valid-invalid bit
	1
	1
	1
	1
	0
⋮	
	0
	0

page table

- During address translation, if valid–invalid bit in page table entry is 0 $\Rightarrow$ page fault.

Page Table When Some Pages Are Not in Main Memory



Page Fault

What if the process refers to (i.e., tries to access) a page not in-memory ?

- The first reference (i.e., address) to that invalid page will trap to operating system and causes a page fault

Page Fault

Procedure for handling a page fault :

1. OS checks an internal table to see if reference is valid or invalid memory access.

2. If Invalid reference :abort the process.

⇒address is not in logical address space of process.

Just not in memory :page in the referred page from the disk.

=>logical address is valid but page is simply not in Memory

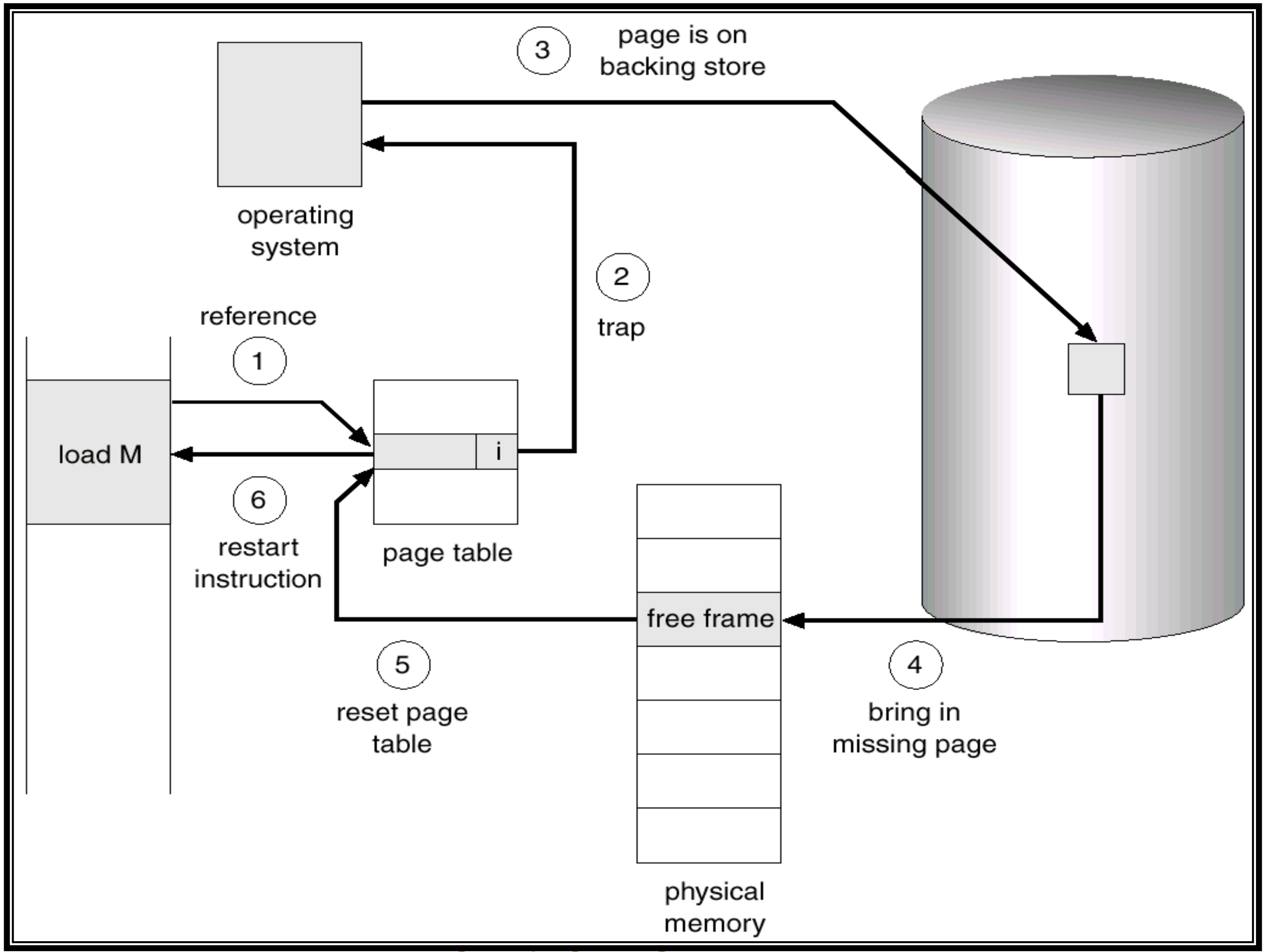
Page Fault

Procedure for handling a page fault :

Cont….

3. Find a free frame.
4. Read the referred page into this allocated frame via scheduled disk operation.
5. Update both internal table and page-table by setting validation bit = **v.**
6. Restart the instruction that caused the page fault and resume process execution.

Steps in Handling a Page Fault



What happens if there is no free frame?

- Page replacement– find some page in memory, but not really in use, swap it out.
  - **algorithm**
  - **performance** – want an algorithm which will result in minimum number of page faults.

- Same page may be brought into memory several times.

Hardware Support

Hardware support needed for demand paging; same as hardware for paging and swapping

- **Page table**: with valid / invalid bit, or special protection bits
- **Secondary memory**: swap device with swap space; for not in-memory pages

Page Replacement Algorithms

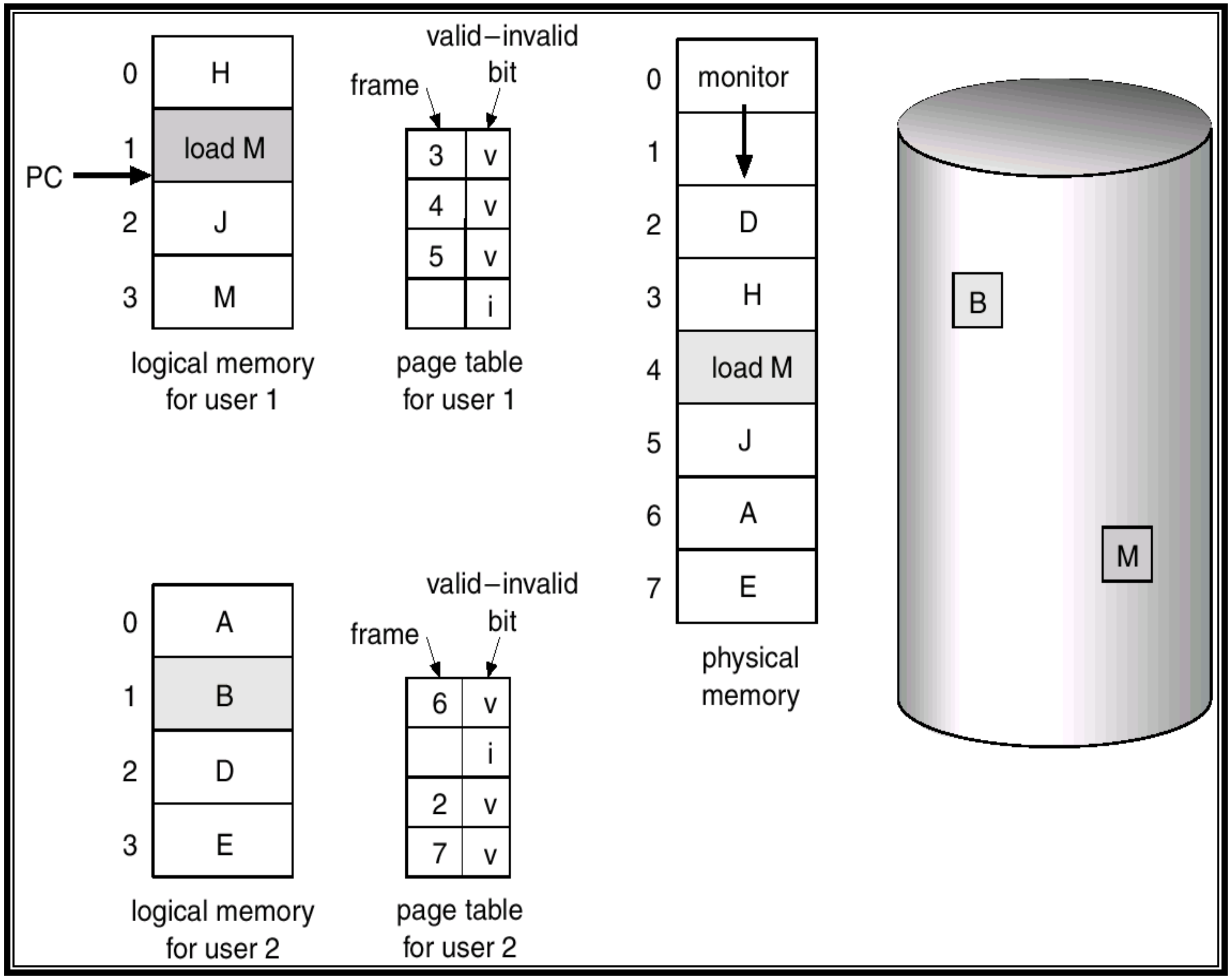
Page Replacement

- Prevent over-allocation of memory by modifying page-fault service routine to include page replacement.

- Use *modify* (*dirty*) *bit* to reduce overhead of page transfers – only modified pages are written to disk.

- Page replacement completes separation between logical memory and physical memory – large virtual memory can be provided on a smaller physical memory.

Need For Page Replacement



Basic Page Replacement

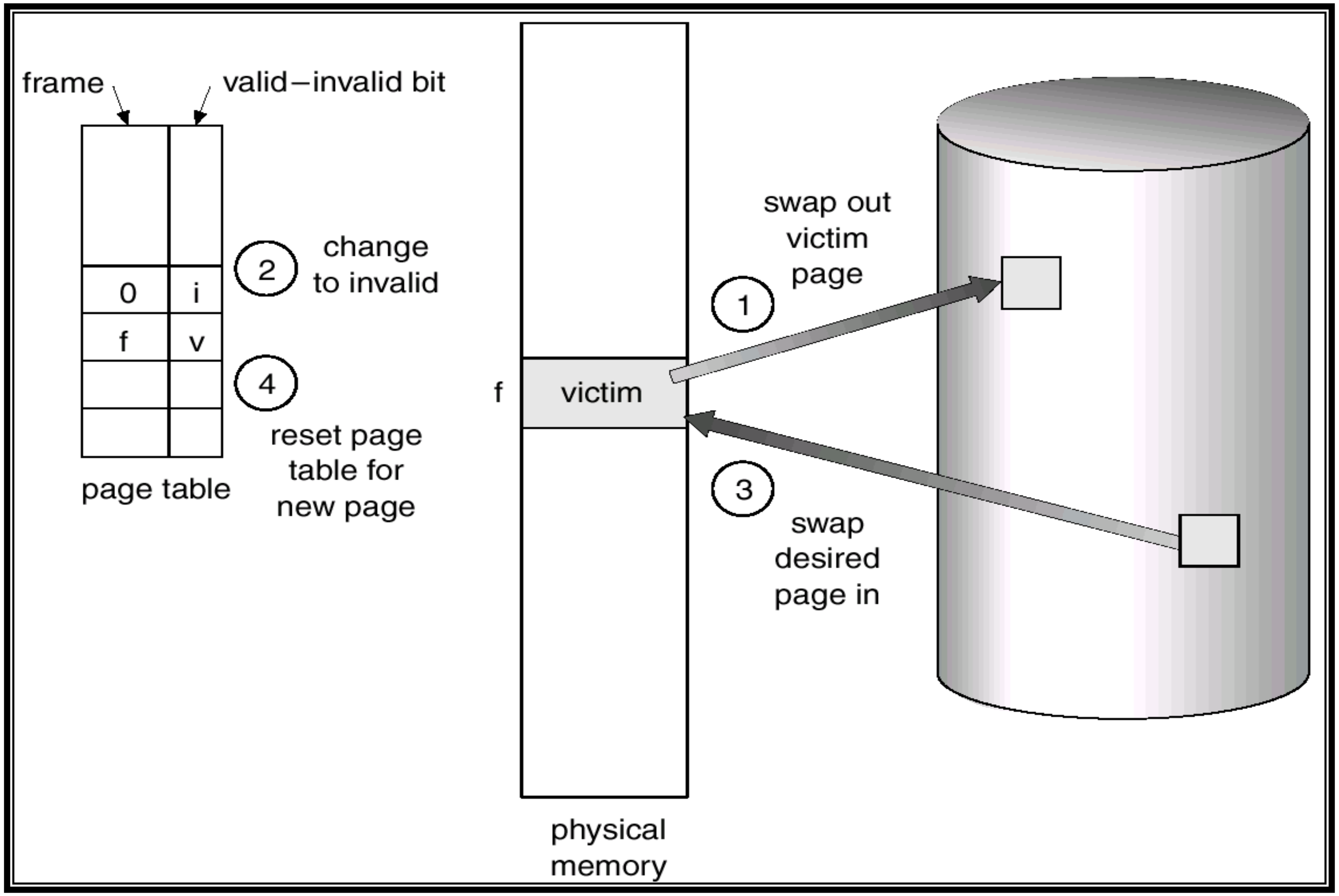
1. Find the location of the desired page on disk.

2. Find a free frame:
   - If there is a free frame, use it.
   - If there is no free frame, use a page replacement algorithm to select a *victim* frame.

3. Read the desired page into the (newly) free frame. Update the page and frame tables.

4. Restart the process.

Page Replacement



Page Replacement Algorithms

- Want lowest page-fault rate.
- Evaluate algorithm by running it on a particular string of memory references (reference string) and computing the number of page faults on that string.

First-In-First-Out

First-In-First-Out (FIFO) Algorithm

- A FIFO replacement algorithm associates with each page the time when that page was brought into memory.
- Each page brought into memory is inserted into a first-in first-out queue.
- Page to be replaced is the oldest page; the one at the head of the queue.
- When a page is brought into memory, we insert it at the tail of the queue.

Example:

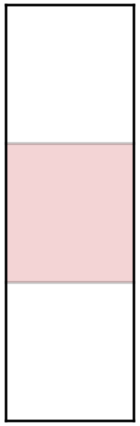
Reference string: 7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1

Frame size = 3

First-In-First-Out (FIFO) Algorithm

Reference string: 7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1

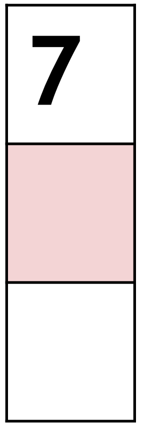
Frame size = 3



First-In-First-Out (FIFO) Algorithm

Reference string: 7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1

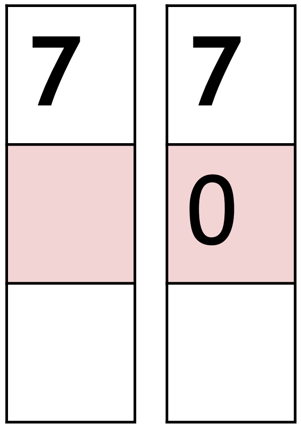
Frame size = 3



First-In-First-Out (FIFO) Algorithm

Reference string: 7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1

Frame size = 3



First-In-First-Out (FIFO) Algorithm

Reference string: 7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1

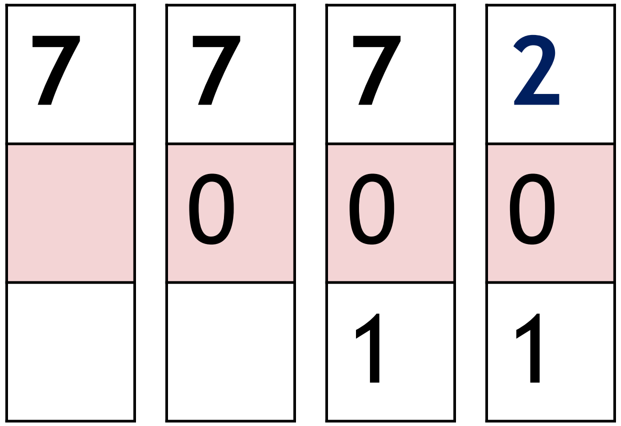
Frame size = 3

7	7	7
	0	0
		1

First-In-First-Out (FIFO) Algorithm

Reference string: 7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1

Frame size = 3



First-In-First-Out (FIFO) Algorithm

Reference string: 7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1

Frame size = 3

7	7	7	2	2
	0	0	0	0
		1	1	1

First-In-First-Out (FIFO) Algorithm

Reference string: 7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1

Frame size = 3

7	7	7	2	2	2
	0	0	0	0	3
		1	1	1	1

First-In-First-Out (FIFO) Algorithm

Reference string: 7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1

Frame size = 3

7	7	7	2	2	2	2
	0	0	0	0	3	3
		1	1	1	1	0

First-In-First-Out (FIFO) Algorithm

Reference string: 7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1

Frame size = 3

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

First-In-First-Out (FIFO) Algorithm

Reference string: 7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1

Frame size = 3

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

First-In-First-Out (FIFO) Algorithm

Reference string: 7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1

Frame size = 3

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

First-In-First-Out (FIFO) Algorithm

Reference string: 7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1

Frame size = 3

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

First-In-First-Out (FIFO) Algorithm

Reference string: 7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1

Frame size = 3

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

0
2
3

First-In-First-Out (FIFO) Algorithm

Reference string: 7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1

Frame size = 3

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

0	0
2	2
3	3

First-In-First-Out (FIFO) Algorithm

Reference string: 7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1

Frame size = 3

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

0	0	0
2	2	1
3	3	3

First-In-First-Out (FIFO) Algorithm

Reference string: 7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1

Frame size = 3

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

0	0	0	0
2	2	1	1
3	3	3	2

First-In-First-Out (FIFO) Algorithm

Reference string: 7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1

Frame size = 3

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

0	0	0	0	0
2	2	1	1	1
3	3	3	2	2

First-In-First-Out (FIFO) Algorithm

Reference string: 7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1

Frame size = 3

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

0	0	0	0	0	0
2	2	1	1	1	1
3	3	3	2	2	2

First-In-First-Out (FIFO) Algorithm

Reference string: 7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1

Frame size = 3

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

0	0	0	0	0	0	7
2	2	1	1	1	1	1
3	3	3	2	2	2	2

First-In-First-Out (FIFO) Algorithm

Reference string: 7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1

Frame size = 3

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

0	0	0	0	0	0	7	7
2	2	1	1	1	1	1	0
3	3	3	2	2	2	PF 2	2

First-In-First-Out (FIFO) Algorithm

Reference string: 7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1

Frame size = 3

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

0	0	0	0	0	0	7	7	7
2	2	1	1	1	1	1	0	0
3	3	3	2	2	2	2	2	1

First-In-First-Out (FIFO) Algorithm

Reference string: 7,0,1,2,0,3,0,4,2,3,0,3,2,1,2,0,1,7,0,1

Frame size = 3

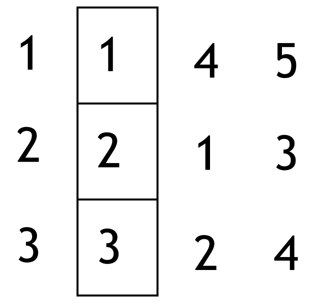
7	7	7	2	2	2	2	4	4	4	0
	0	0	0	0	3	3	3	2	2	2
		1	1	1	1	0	0	0	3	3
PF	PF	PF	PF		PF	PF	PF	PF	PF	PF

0	0	0	0	0	0	7	7	7
2	2	1	1	1	1	1	0	0
3	3	3	2	2	2	2	2	1
		PF	PF			PF	PF	PF

Page Fault = 15

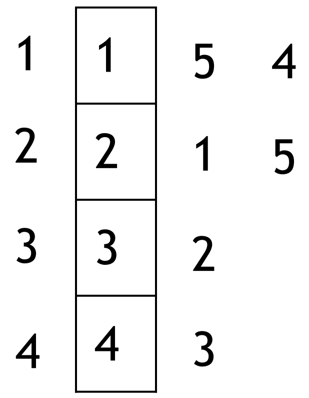
First-In-First-Out (FIFO) Algorithm

- Reference string: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5
- 3 frames (3 pages can be in memory at a time per process)



9 page faults

- 4 frames



10 page faults

- FIFO Replacement – Adding more frames can cause more page faults! => **Belady's Anomaly**

Optimal Page Replacement

Optimal Page Replacement Algorithm

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

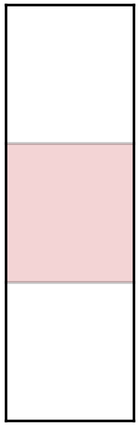
Reference string: 7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1

Frame size = 3

Optimal Page Replacement Algorithm

Reference string: 7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1

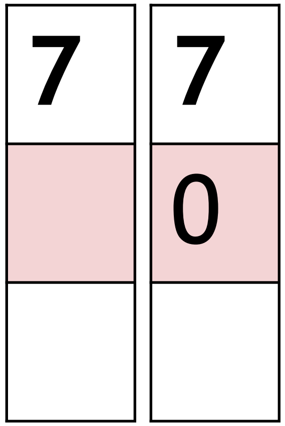
Frame size = 3



Optimal Page Replacement Algorithm

Reference string: 7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1

Frame size = 3



Optimal Page Replacement Algorithm

Reference string: 7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1

Frame size = 3

7	7	7
	0	0
		1

Optimal Page Replacement Algorithm

Reference string: 7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1

Frame size = 3

7	7	7	2
	0	0	0
		1	1

Optimal Page Replacement Algorithm

Reference string: 7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1

Frame size = 3

7	7	7	2	2
	0	0	0	0
		1	1	1

Optimal Page Replacement Algorithm

Reference string: 7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1

Frame size = 3

7	7	7	2	2	2
	0	0	0	0	0
		1	1	1	3

Optimal Page Replacement Algorithm

Reference string: 7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1

Frame size = 3

7	7	7	2	2	2	2
	0	0	0	0	0	0
		1	1	1	3	3

Optimal Page Replacement Algorithm

Reference string: 7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1

Frame size = 3

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

Optimal Page Replacement Algorithm

Reference string: 7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1

Frame size = 3

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

Optimal Page Replacement Algorithm

Reference string: 7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1

Frame size = 3

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

Optimal Page Replacement Algorithm

Reference string: 7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1

Frame size = 3

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

Optimal Page Replacement Algorithm

Reference string: 7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1

Frame size = 3

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

2
0
3

Optimal Page Replacement Algorithm

Reference string: 7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1

Frame size = 3

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

2	2
0	0
3	3

Optimal Page Replacement Algorithm

Reference string: 7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1

Frame size = 3

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

2	2	2
0	0	0
3	3	1

Optimal Page Replacement Algorithm

Reference string: 7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1

Frame size = 3

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

2	2	2	2
0	0	0	0
3	3	1	1

Optimal Page Replacement Algorithm

Reference string: 7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1

Frame size = 3

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

2	2	2	2	2
0	0	0	0	0
3	3	1	1	1

Optimal Page Replacement Algorithm

Reference string: 7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1

Frame size = 3

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

2	2	2	2	2	2
0	0	0	0	0	0
3	3	1	1	1	1

Optimal Page Replacement Algorithm

Reference string: 7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1

Frame size = 3

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

2	2	2	2	2	2	7
0	0	0	0	0	0	0
3	3	1	1	1	1	1

Optimal Page Replacement Algorithm

Reference string: 7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1

Frame size = 3

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

2	2	2	2	2	2	7	7
0	0	0	0	0	0	0	0
3	3	1	1	1	1	1	1

Optimal Page Replacement Algorithm

Reference string: 7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1

Frame size = 3

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

2	2	2	2	2	2	7	7	7
0	0	0	0	0	0	0	0	0
3	3	1	1	1	1	1	1	1

Optimal Page Replacement Algorithm

Reference string: 7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1

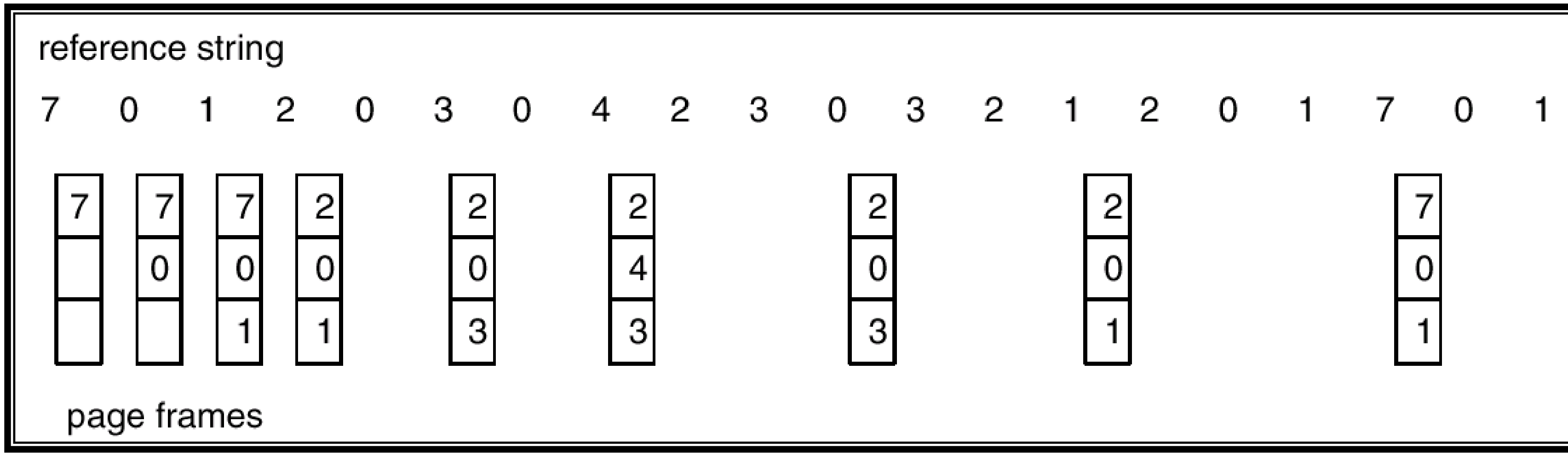
Frame size = 3

7	7	7	2	2	2	2	2	2	2	2
	0	0	0	0	0	0	4	4	4	0
		1	1	1	3	3	3	3	3	3
PF	PF	PF	PF		PF		PF			PF

2	2	2	2	2	2	7	7	7
0	0	0	0	0	0	0	0	0
3	3	1	1	1	1	1	1	1
		PF				PF		

Page Fault = 09

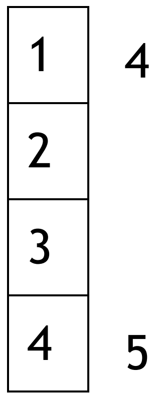
Optimal Page Replacement



Optimal Page Replacement Algorithm

- 4 frames example

  1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5



6 page faults

Optimal Page Replacement Algorithm

- It has the lowest page-fault rate of all algorithms and will never suffer from Belady's anomaly. Such an algorithm does exist and has been called OPT or MIN.
- It is difficult to implement, because it requires future knowledge of the reference string.

Least Recently Used

Least Recently Used

- 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.

Least Recently Used

Reference string: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

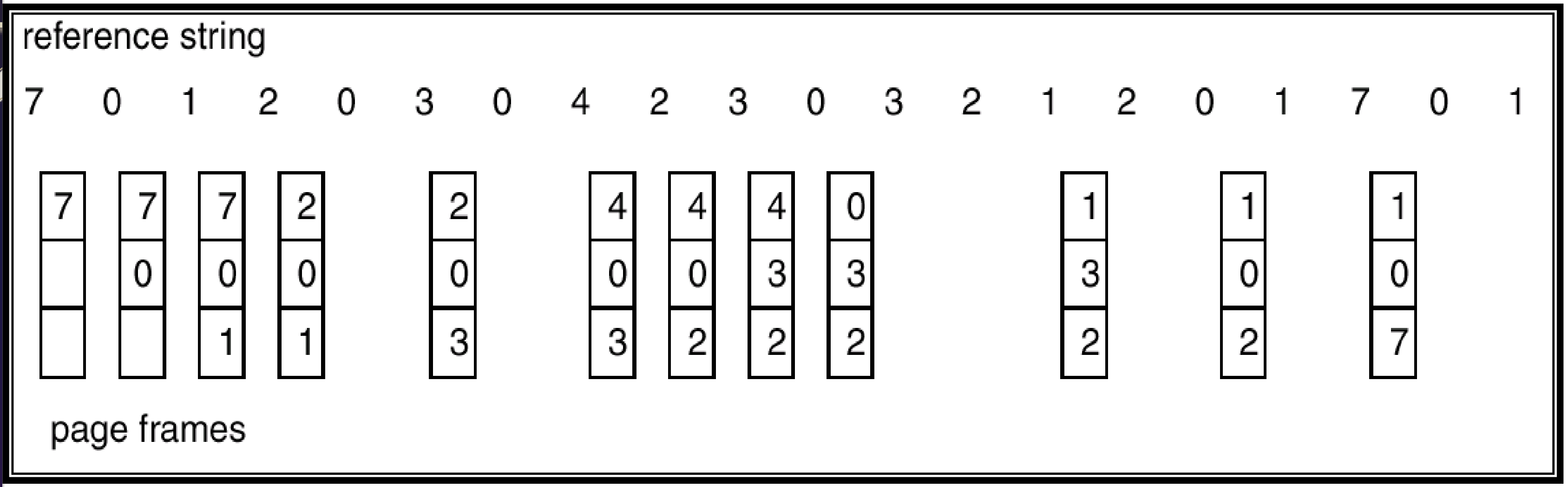
Page frame size=4

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

1	5
2	2
4	4
3	3

Page Fault = 8

LRU Page Replacement



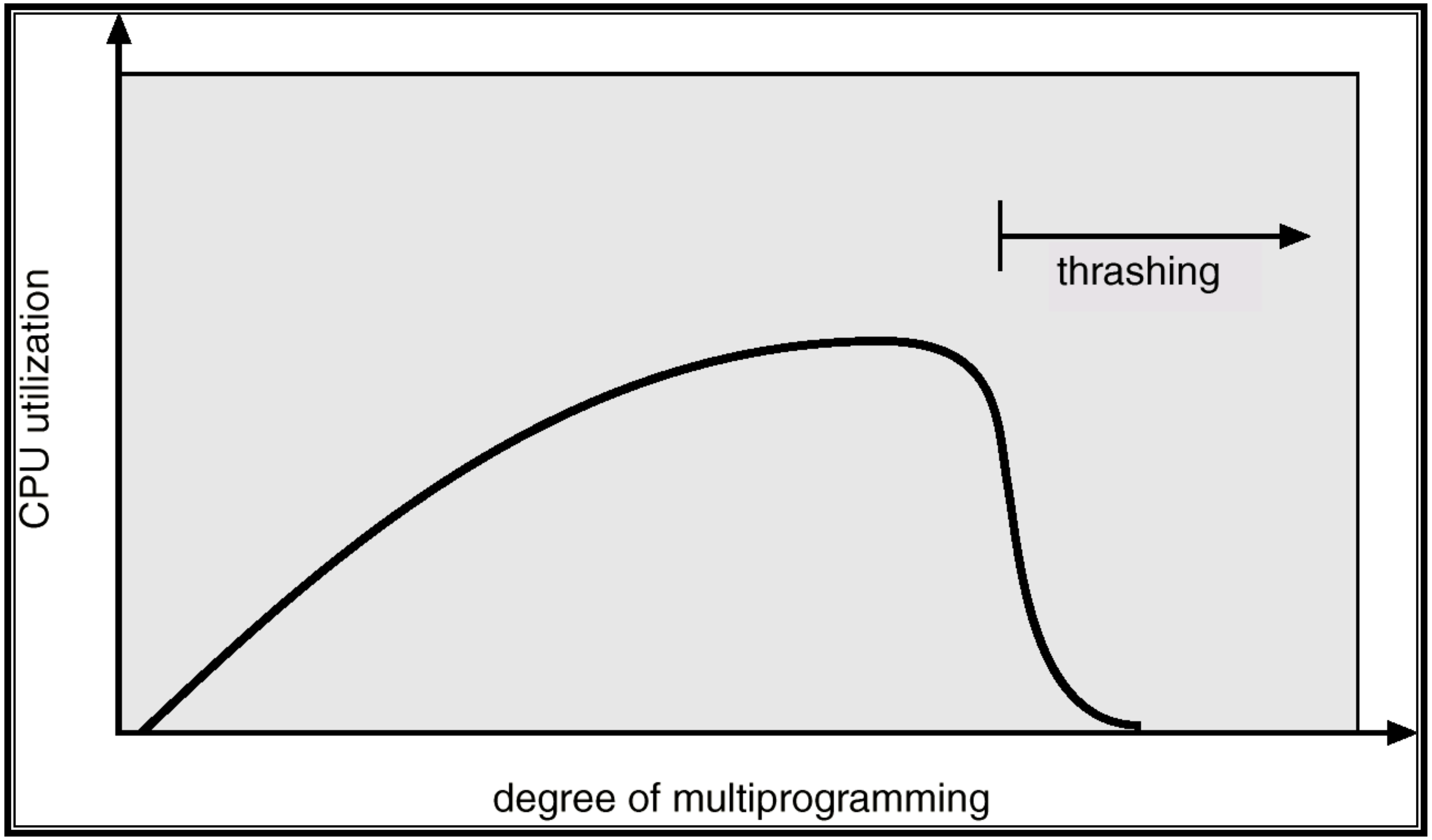
Thrashing

Thrashing

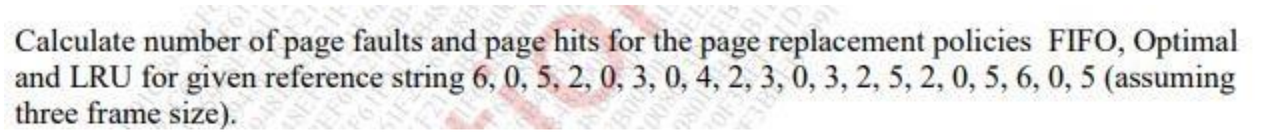
- If a process does not have "enough" pages, the page-fault rate is very high. This leads to:
  - low CPU utilization.
  - operating system thinks that it needs to increase the degree of multiprogramming.
  - another process added to the system.

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

Thrashing



Consider the following page reference string 1, 2,3,4,5,2,6,7,3,2,4,1,7,1,4,3,2,3,4,7,1. Compare the number of page faults with frame sizes 3 and 4 with LRU replacement algorithm.



Calculate number of page faults and page hits for the page replacement policies FIFO, Optimal and LRU for given reference string 6, 0, 5, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 5, 2, 0, 5, 6, 0, 5 (assuming three frame size).