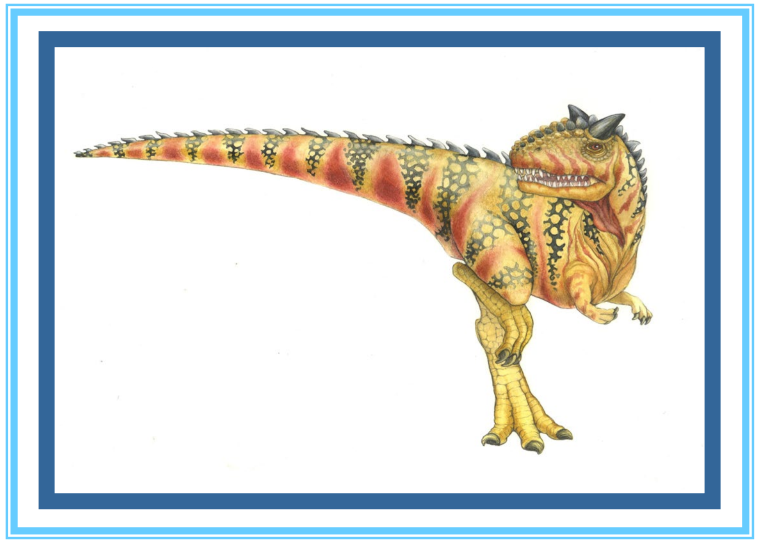
Chapter 8: Memory-Management Strategies



Chapter 8: Memory Management Strategies

OBJECTIVES

- To provide a detailed description of various ways of organizing memory hardware
- To discuss various memory-management techniques, including paging and segmentation

OUTLINE

1. BACKGROUND

Overview

- A program must be brought (from disk) into memory and placed within a process for it to be run.
- A program can be written in machine language, assembly language, or high-level language.
- Main memory and registers are the only storage entities that a CPU can access directly.
- The CPU fetches instructions from main memory according to the value of the program counter.
- Typical instruction execution cycle – fetch instruction from memory, decode the instruction, operand fetch, possible storage of result in memory.

- Memory unit only sees a stream of one of the following:
  - address + read requests (*e.g.*, load memory location `20010` into register number 8).
  - address + data and write requests (*e.g.*, store content of register 6 into memory location `1090`).

Memory unit does not know how these addresses were generated.

Register access can be done in one CPU clock (or less)

Address binding

- A program residing on the disk needs to be brought into memory in order to execute. Such a program is usually stored as a binary <u>executable file</u> and is kept in an **input queue**.

- In general, we do not know in advance where the program is going to reside in memory.

  Therefore, it is convenient to assume that the first physical address of a program always starts at location `0000`.

- Without some hardware or software support, program must be loaded into address `0000`.

> - It is impractical to have first physical address of user process to always start at location `0000`.
> - Most (all) computer systems provide hardware and/or software support for memory management.

- In general, addresses are represented in different ways at different stages of a program's life:
  - Addresses in the source program are generally symbolic.
    - *i.e.*, variable "`count`"
  - A compiler typically binds these symbolic addresses to relocatable addresses.
    - *i.e.*, "*14 bytes from beginning of this module*"
  - Linker or loader will bind relocatable addresses to absolute addresses.
    - *i.e.*, `74014`

Each **binding** maps one address space to another address space.

Logical vs physical address space

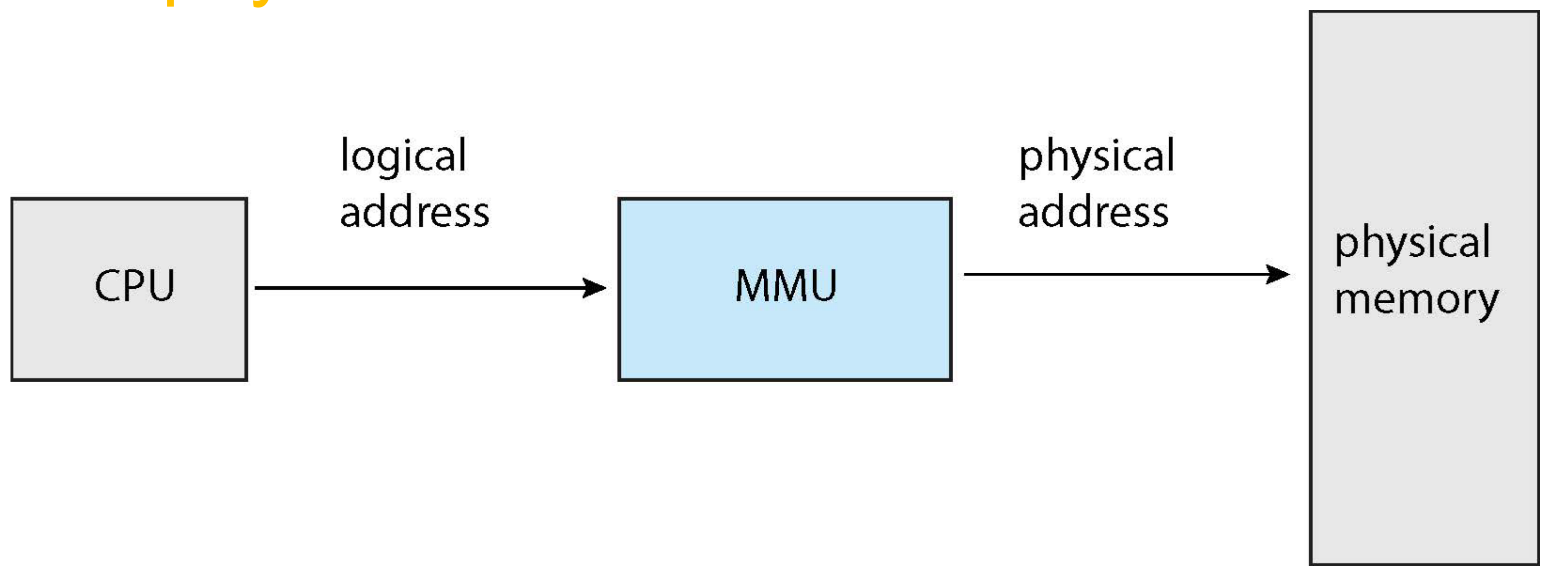
- An address generated by CPU is commonly referred to as a **logical address** (also referred to as virtual address)
- Once loaded into the memory-address register of the memory is commonly referred to as a physical address.
- The compile-time and load-time address-binding methods generate identical **logical addresses** and physical addresses.

Logical address *space* → the set of all **logical addresses** generated by a program.

Physical address *space* → the set of all physical addresses corresponding to the **logical addresses**.

Memory-Management Unit (MMU)

- Hardware device that at run time maps **logical** / **virtual addresses** to physical address.



- 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

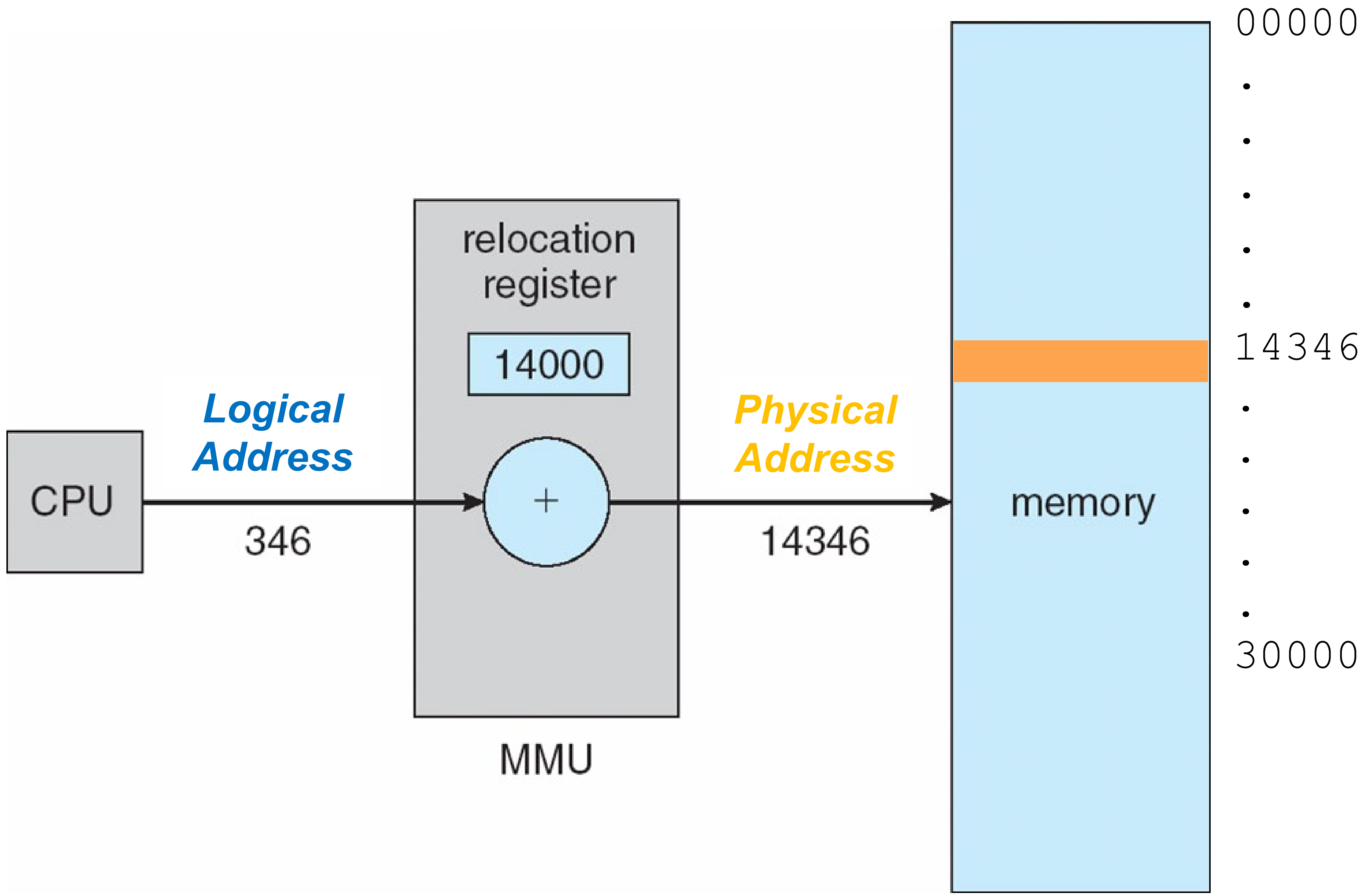


Figure 8.4 Dynamic relocation using a relocation register.

2. SWAPPING

- A process can be **swapped** out of memory to a backing store (temporarily) and then brought back into memory for continued execution
  - Total physical memory space of all processes can exceed the real physical memory of the system.
- **Backing store** – fast disk large enough to accommodate copies of all memory images for all processes; must provide direct access to these memory images.
- System maintains a **ready queue** of ready-to-run processes which are either in;
  - i) memory or
  - ii) have memory images on disk.

- Major part of swap time is transfer time; total transfer time is directly proportional to the amount of memory swapped.

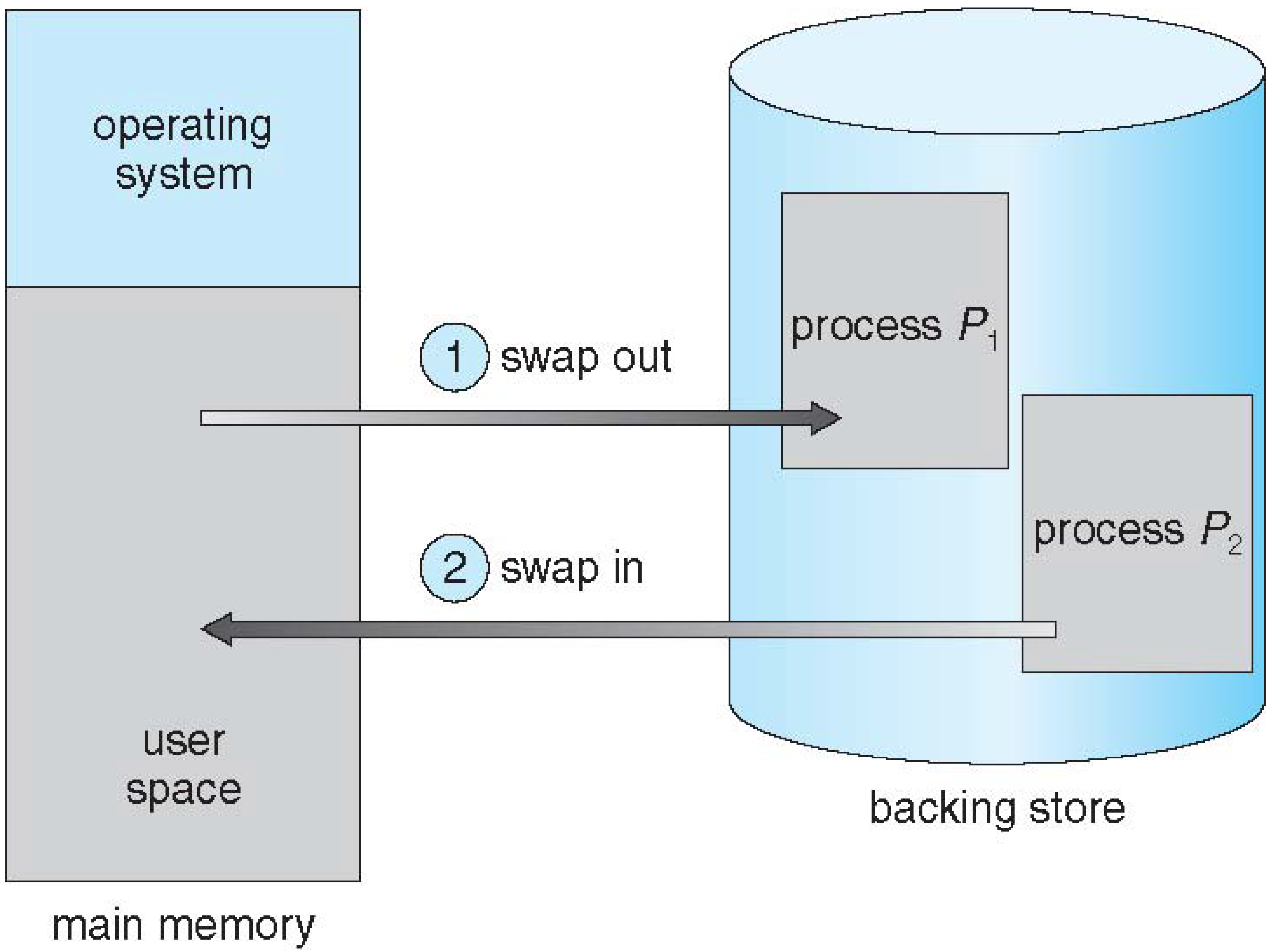


Figure 8.5 Swapping of two processes using a disk as a backing store.

3. MEMORY ALLOCATION

Each process is contained in a single **contiguous** section of memory.

Each process is contained in a **non-contiguous** / non-sequential location of memory.

- Memory Allocation
  - Contiguous
    - Fixed-Partition
    - Dynamic / Variable Partition
  - Non-Contiguous
    - Segmentation
    - Paging

4. CONTIGUOUS MEMORY ALLOCATION

- Main memory must support both OS and user processes
- Limited resource, must allocate efficiently
- Contiguous allocation is one of early methods

- Main memory divided usually into two **partitions**:
  - One for operating system, usually held in low memory
  - User processes then held in high memory
  - Each process is contained in single contiguous section of memory

(a) Fixed-sized Partition : Example

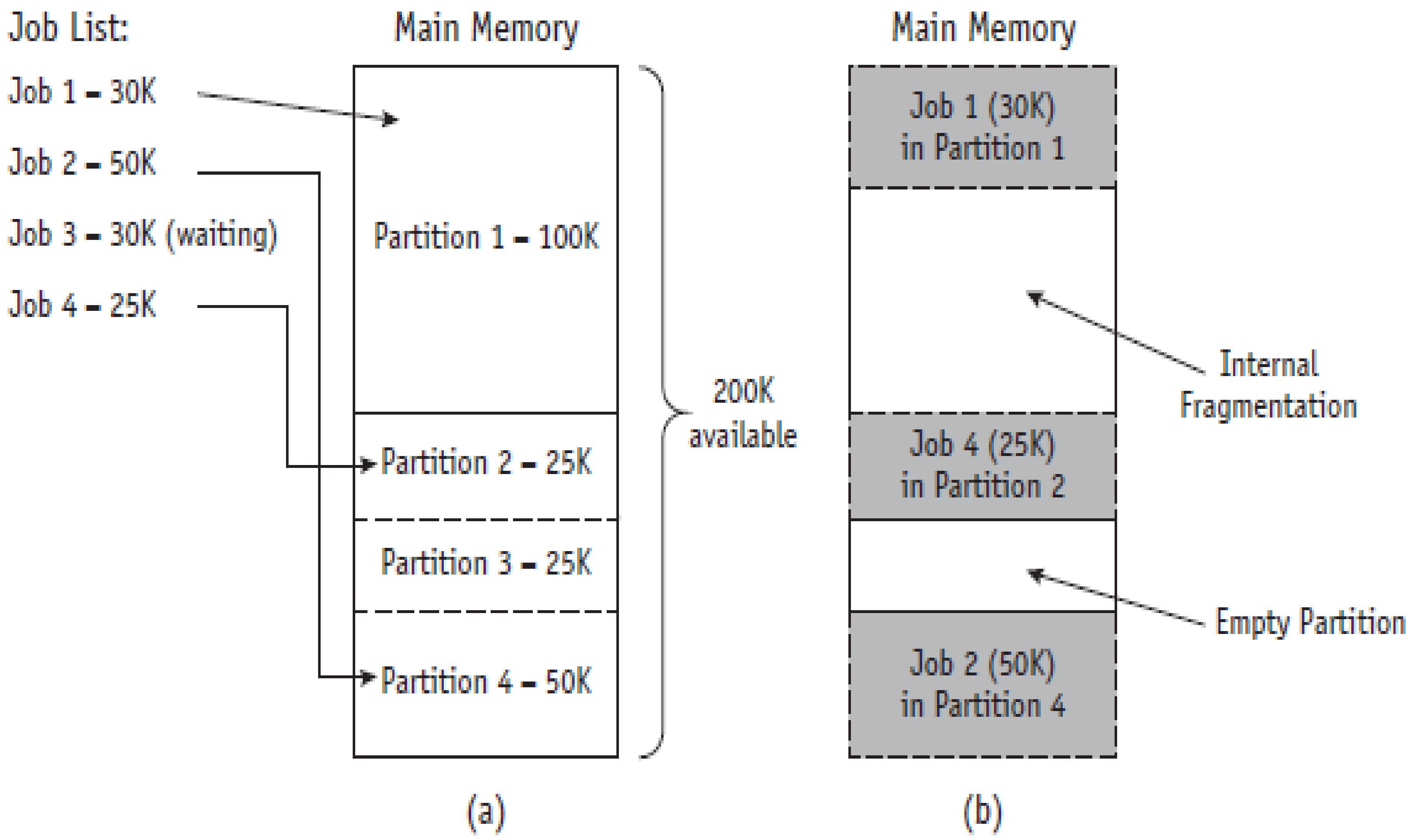
- Each partition may contain exactly one process.
- The degree of multiprogramming is limited by number of partitions
- But it requires
  - Protection of the job's memory space
  - Matching job size with partition size

A simplified fixed-partition memory table with the free partition shaded.

Partition Size	Memory Address	Access	Partition Status
100K	200K	Job 1	Busy
25K	300K	Job 4	Busy
25K	325K		Free
50K	350K	Job 2	Busy

Example 1

Partition Size	Memory Address	Access	Partition Status
100K	200K	Job 1	Busy
25K	300K	Job 4	Busy
25K	325K		Free
50K	350K	Job 2	Busy



(figure 2.3)

Main memory use during fixed partition allocation of Table 2.1. Job 3 must wait even though 70K of free space is available in Partition 1, where Job 1 only occupies 30K of the 100K available. The jobs are allocated space on the basis of "first available partition of required size."

- **Disadvantages Fixed Partitions**
  - Requires contiguous loading of entire program.
  - Job allocation method:

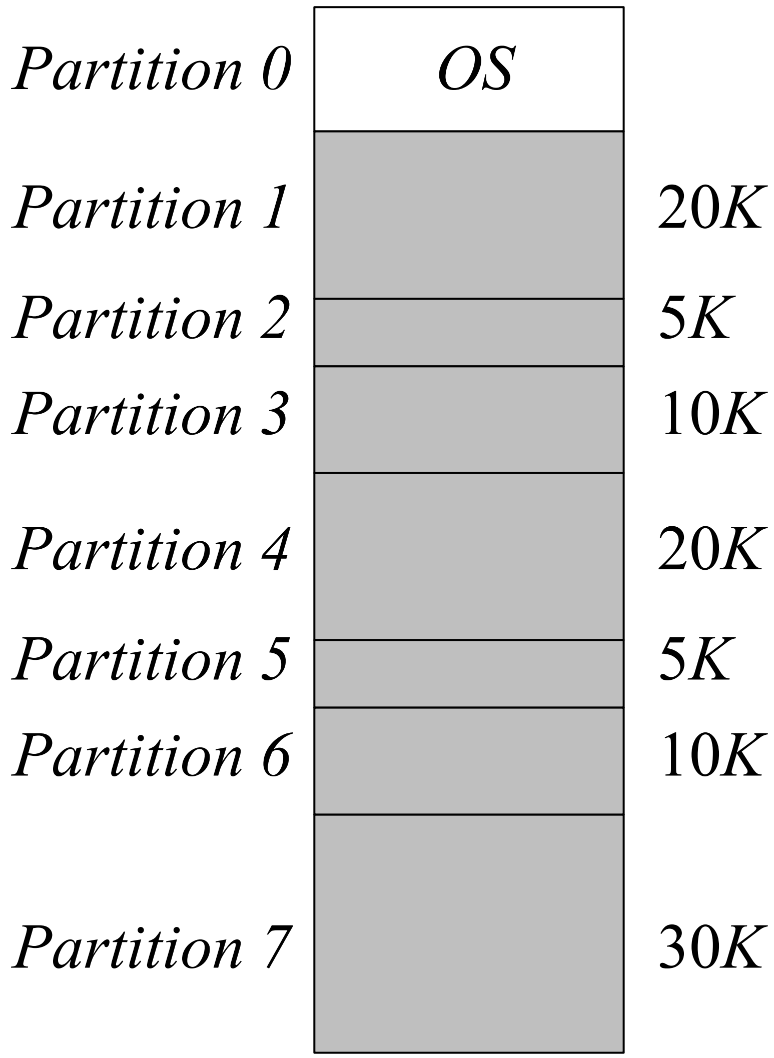
    → *First available partition with required size.*
  - Arbitrary partition size leads to undesired results:
    - Partition too small

      → Large jobs have longer turnaround time.
    - Partition too large

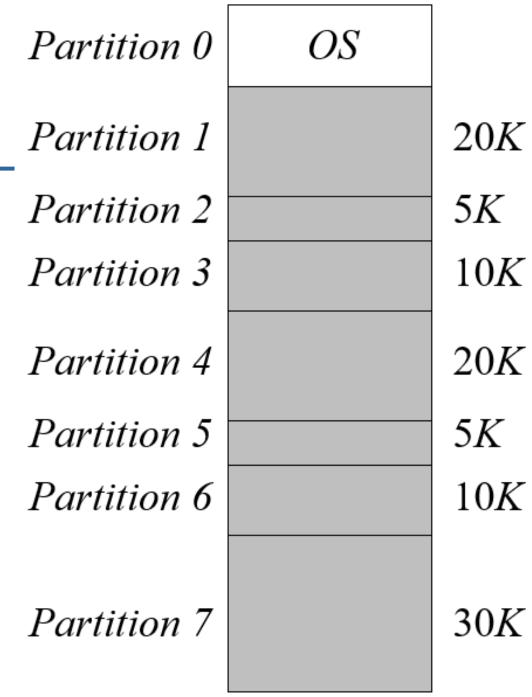
      → Memory waste: internal fragmentation.

Exercise 1

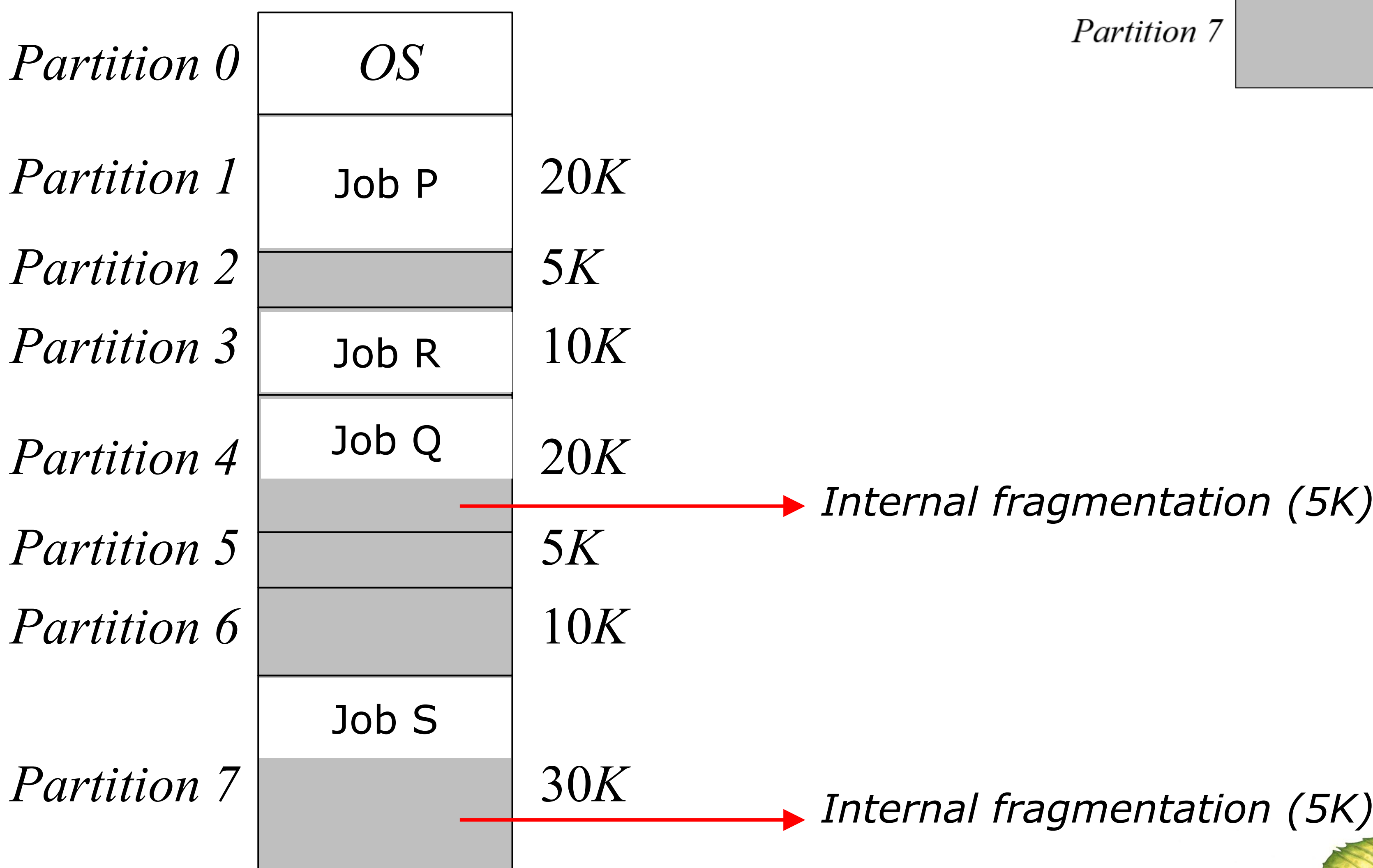
- The new jobs in the order of …
  - Job *P* (20K) – Job *Q* (15K) – Job *R* (10K) – Job *S* (25K) need to be allocated in a **fixed partition** memory as shown in the figure.
  - Label all the jobs in which partition each will be allocated.



Solution

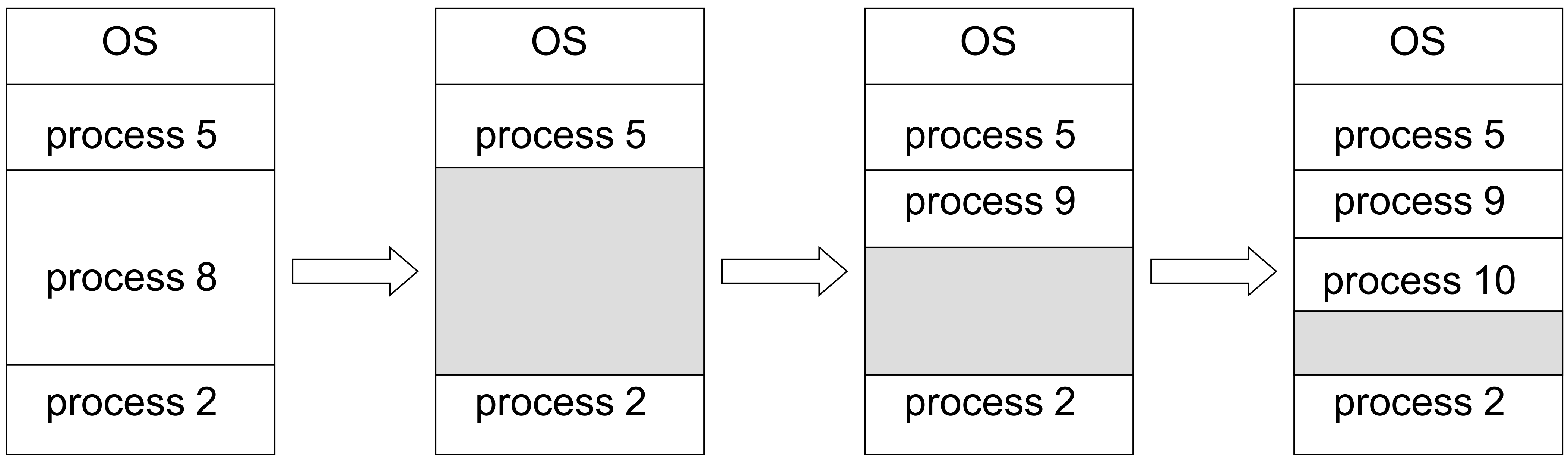


Fixed partition: Job P (20K)
Job Q (15K)
Job R (10K)
Job S (25K)



(b) Dynamic/Variable Partition

- **Dynamic/Variable-partition** sizes for efficiency (sized to a given process' needds)
  - **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
  - Process exiting frees its partition, adjacent free partitions will be combined
  - Operating system maintains information about:
    a) allocated partitions   b) free partitions (hole)



Holes / External fragmentation

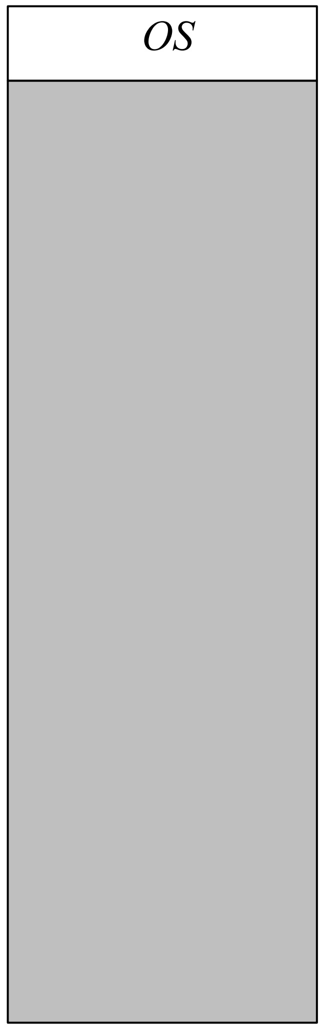
Example 2

Variable/Dynamic partition: Five snapshots (a – e) of main memory as 8 jobs submitted and allocated space on the basis “***First-Come, First-Serve***”.

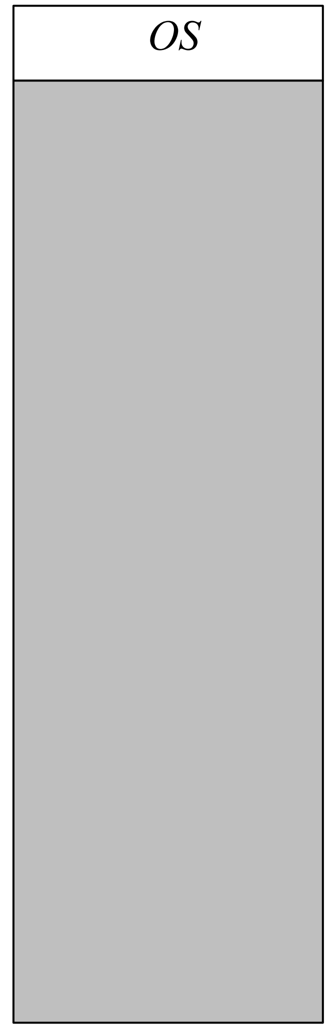
Init. jobs:
A (10*K*)
B (15*K*)
C (20*K*)
D (50*K*)

New jobs:
E (5*K*)
F (30*K*)

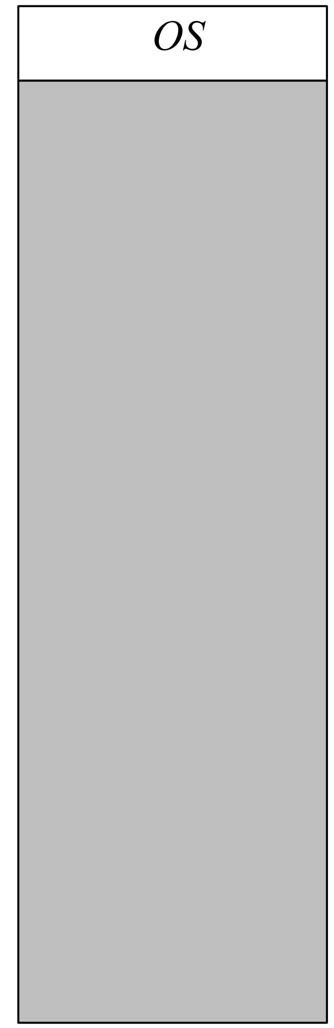
New jobs:
G (10*K*)
H (30*K*)



(a)
Initial job entry memory allocation.



(b)
After Job *A* and *D* have finished.



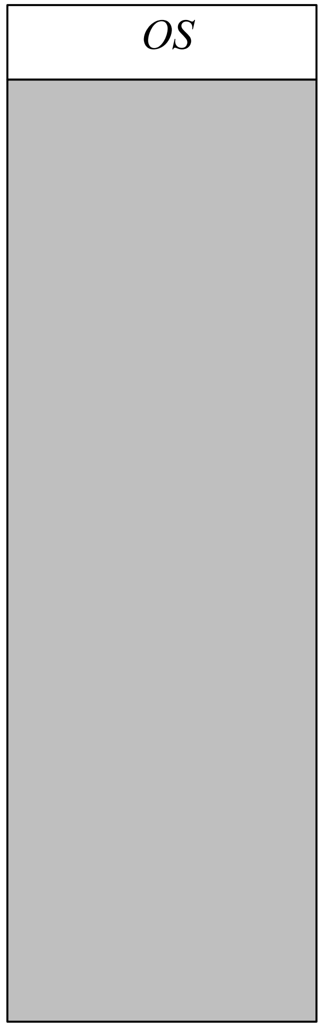
(c)
After Job *E* and *F* have entered.

Variable/Dynamic partition: Five snapshots (a – e) of main memory as 8 jobs submitted and allocated space on the basis "***First-Come, First-Serve***".

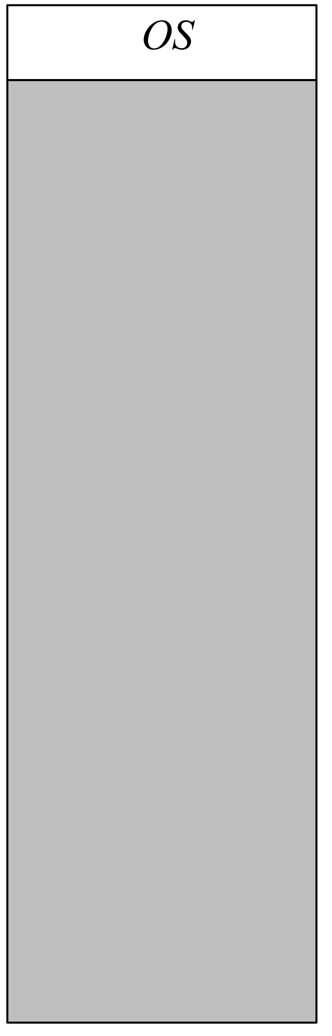
Init. jobs:
A (10*K*)
B (15*K*)
C (20*K*)
D (50*K*)

New jobs:
E (5*K*)
F (30*K*)

New jobs:
G (10*K*)
H (30*K*)



(d)
After Job *C* has finished.



(e)
After Job *G* has entered.

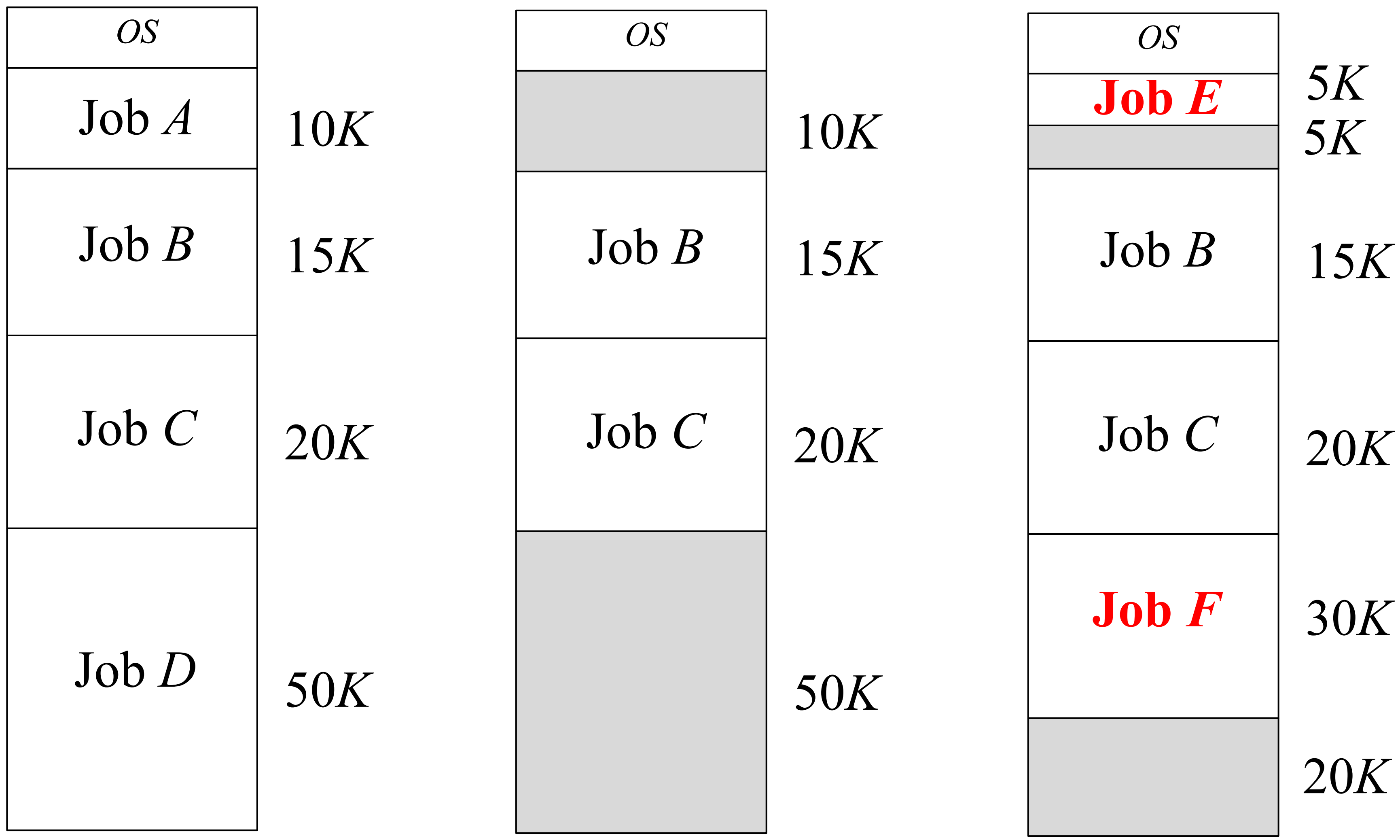
Solution

Variable/Dynamic partition: Five snapshots (a – e) of main memory as 8 jobs submitted and allocated space on the basis "*First-Come, First-Serve*".

<u>Init. jobs</u>:
A (10*K*)
B (15*K*)
C (20*K*)
D (50*K*)

<u>New jobs</u>:
E (5*K*)
F (30*K*)

<u>New jobs</u>:
G (10*K*)
H (30*K*)



(a) Initial job entry memory allocation.

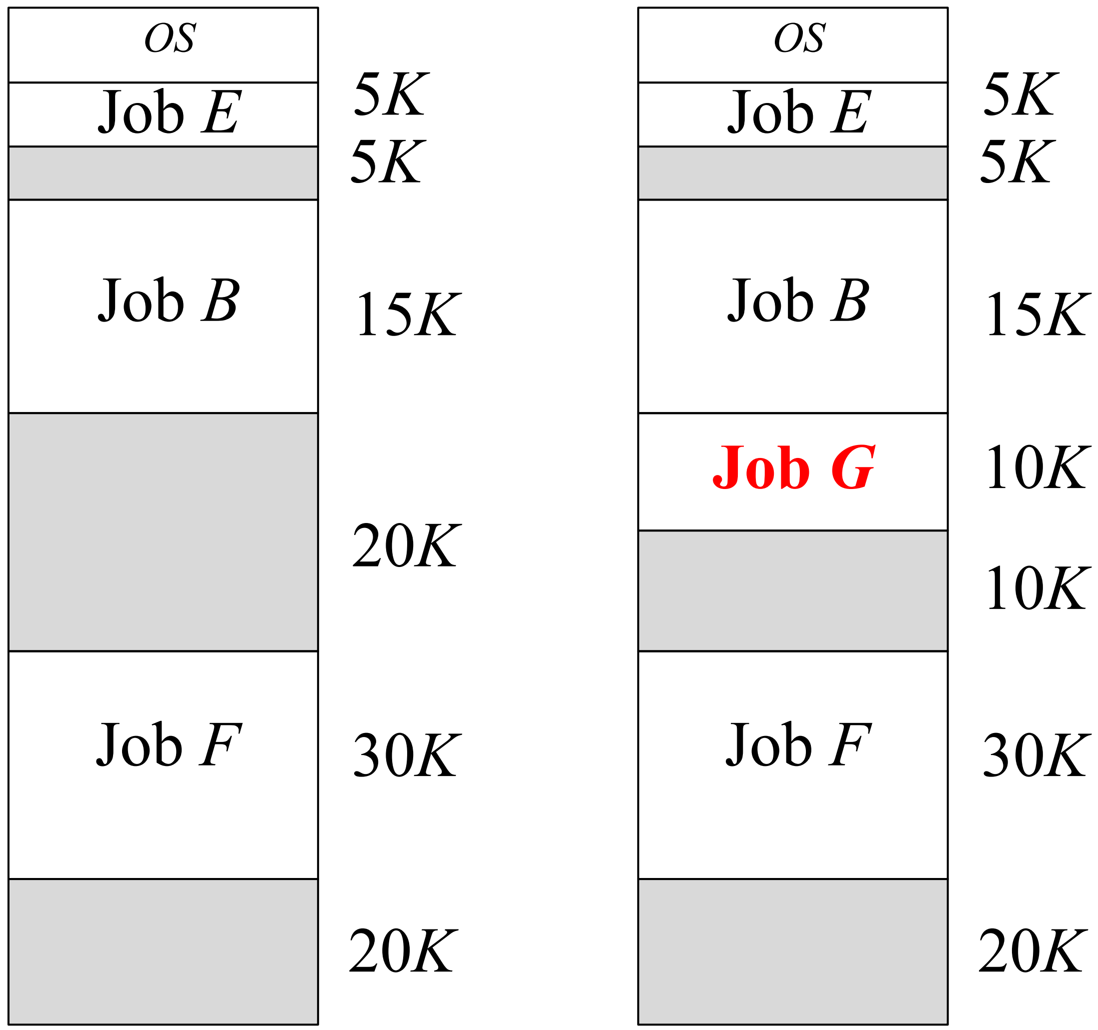
(b) After Job *A* and *D* have finished.

(c) After Job *E* and *F* have entered.

Init. jobs:
A (10*K*)
B (15*K*)
C (20*K*)
D (50*K*)

New jobs:
E (5*K*)
F (30*K*)

New jobs:
G (10*K*)
H (30*K*)



(d)
After Job *C* has finished.

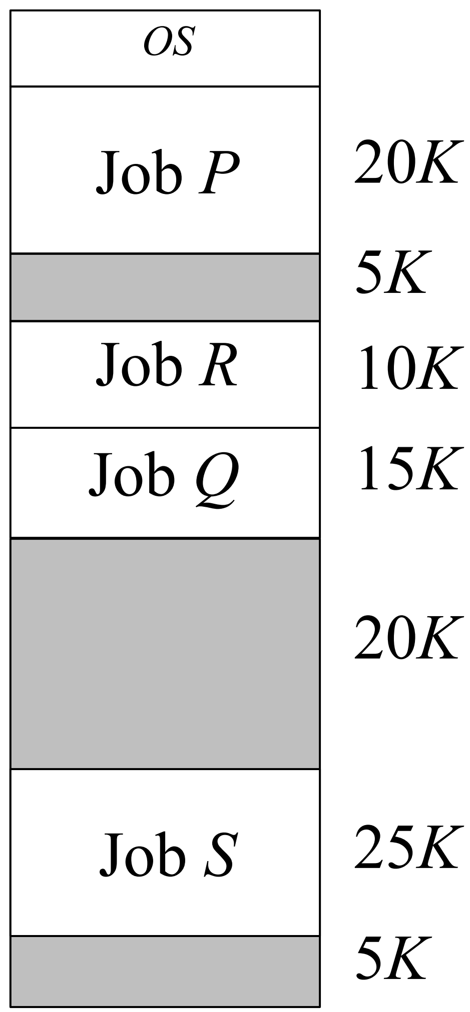
(e)
After Job *G* has entered.

Job *H* has to wait even though there's enough free memory in between partitions to accommodate it.

Exercise 2

Figure shows a few jobs have been allocated in a **dynamic partition** memory at time t_0. Suppose that job P and Q finished at t_1 and new jobs arrived at t_3. Allocate all the new jobs in the memory.

New jobs:
T (30K)
U (35K)
V (15K)



Solution

New jobs:
T (30K)
U (35K)
V (15K)

t_0: Initial jobs in memory.

t_1: After Job P and Q have finished.

t_3: After Job T and V have entered.

Job U need to wait.

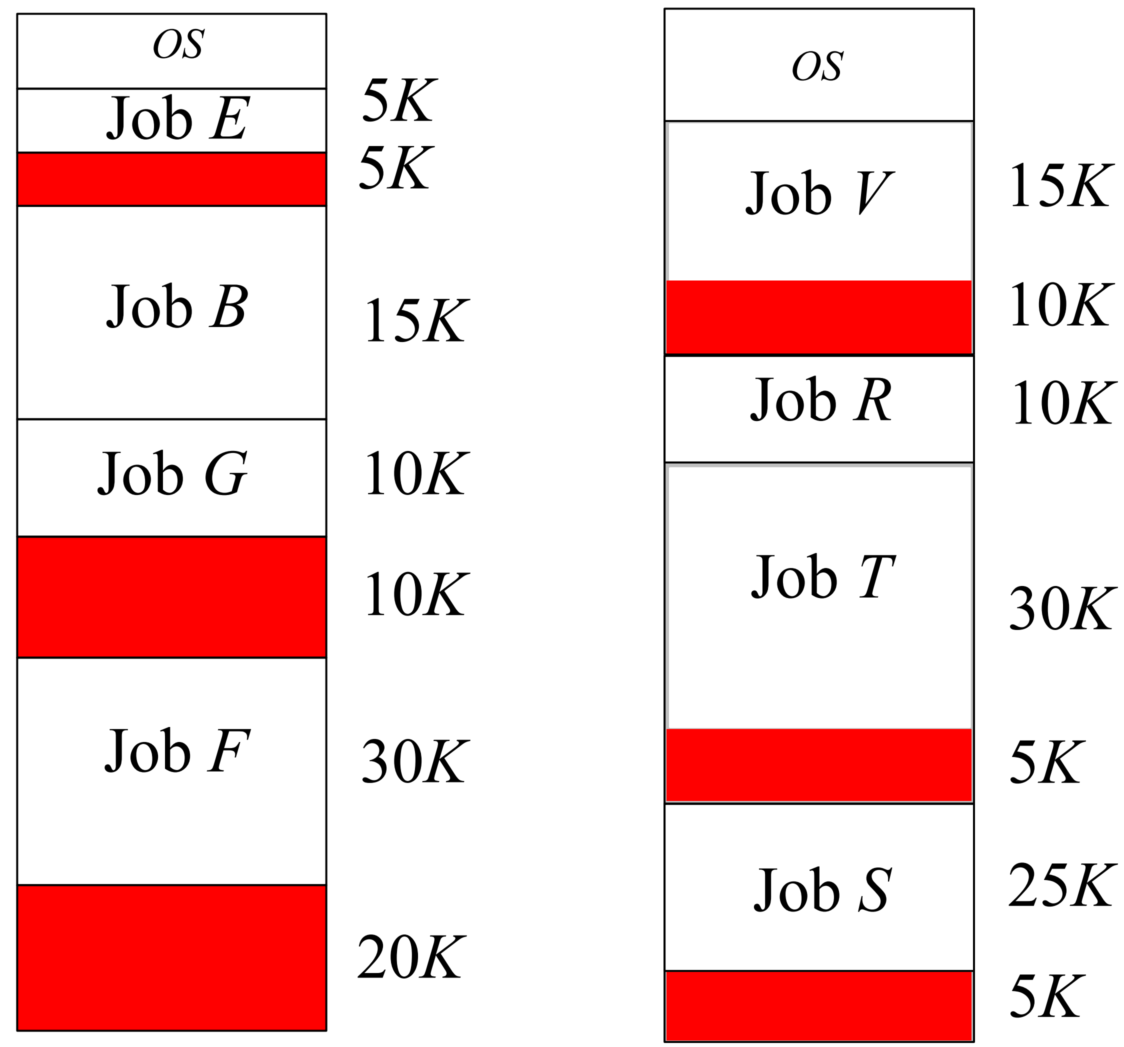
- Based on previous examples, job ***H*** and ***U*** respectively have to wait until any job ends with enough size.

- **<u>Disadvantages</u>**:

  Produce block of available memory (**holes**) with different sizes that scattered out the memory.

- **<u>Solution</u>**:

  Relocatable Dynamic Partition



(a) (b)

Holes or external fragmentation examples before

Fragmentation

- There are two types of fragmentation:

Internal fragmentation

- allocated memory may be slightly larger than requested memory;
- the difference of memory internal to a partition that not being used;
- Results from **Fixed partition allocation**.

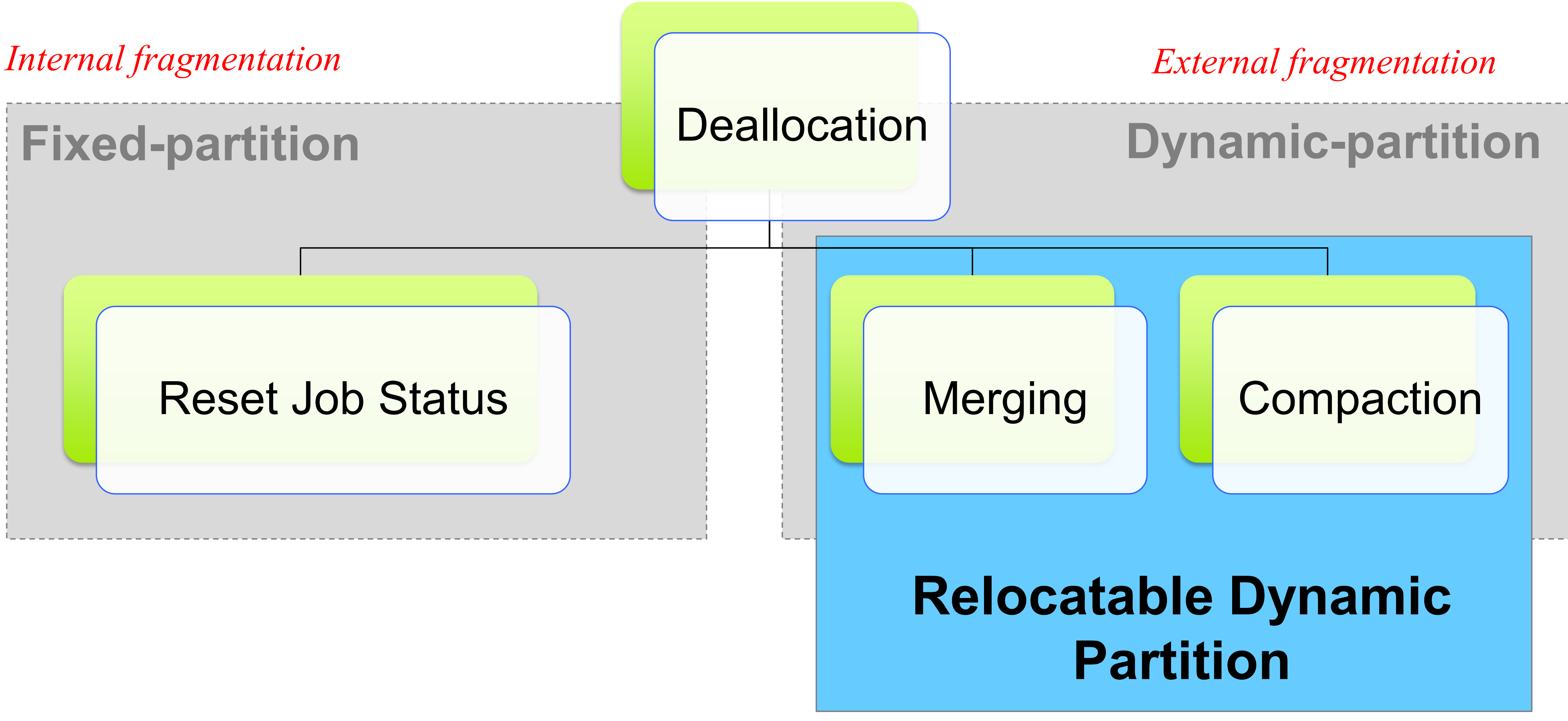
External fragmentation

- total memory space exists to satisfy a request, but it is not contiguous.
- Results from **Dynamic/Variable partition allocation.**

Deallocation

- Solution to reduce fragmentation

  → **Deallocation** : freeing allocated memory spaces.



- For fixed-partition system:
  - Straightforward process
  - Memory manager resets the status of job's memory block to free upon job completion
  - Example
    - Binary values with
      - **0** indicating FREE, and **1** indicating BUSY

A simplified fixed-partition memory table with the free partition shaded.

Partition Size	Memory Address	Access	Partition Status
100K	200K	Job 1	Busy
25K	300K	Job 4	Busy
25K	325K		Free
50K	350K	Job 2	Busy

- For dynamic-partition system:
  - Algorithm tries to merge free areas of memory
  - More complex

- Reduce external fragmentation by **merging and compaction**

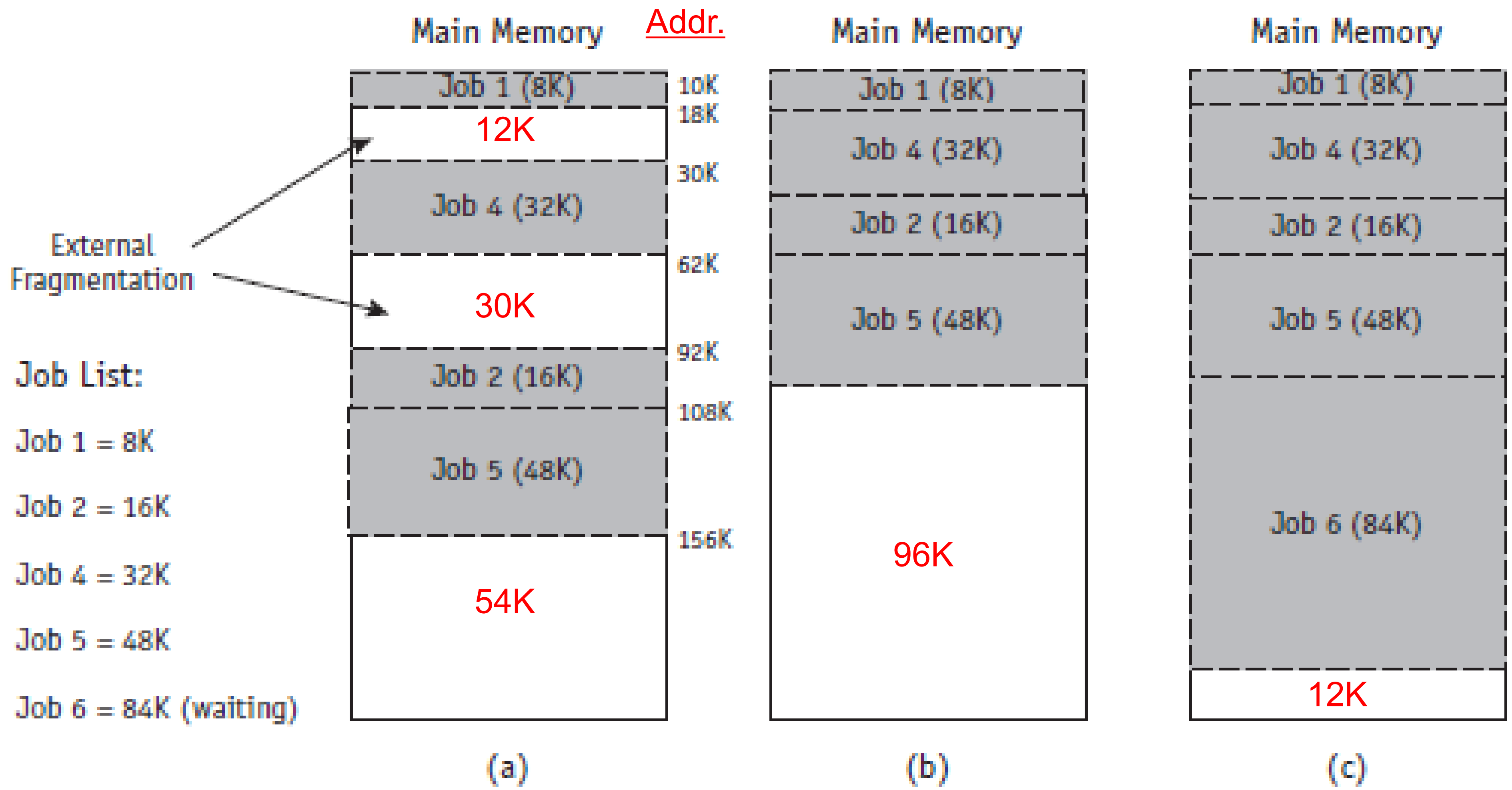
  - **Compaction**:
    - reclaiming fragmented sections of memory space.
    - shuffle memory contents to place all free memory together in one large block.
    - every program in memory must be relocated.

  - Compaction is possible *only* if relocation is dynamic, and is done at execution time
  - It is called **Relocatable Dynamic Partition**

Example 3

Solution to External Fragmentation – **Compaction**

Merge or combine all scattered holes at one location.



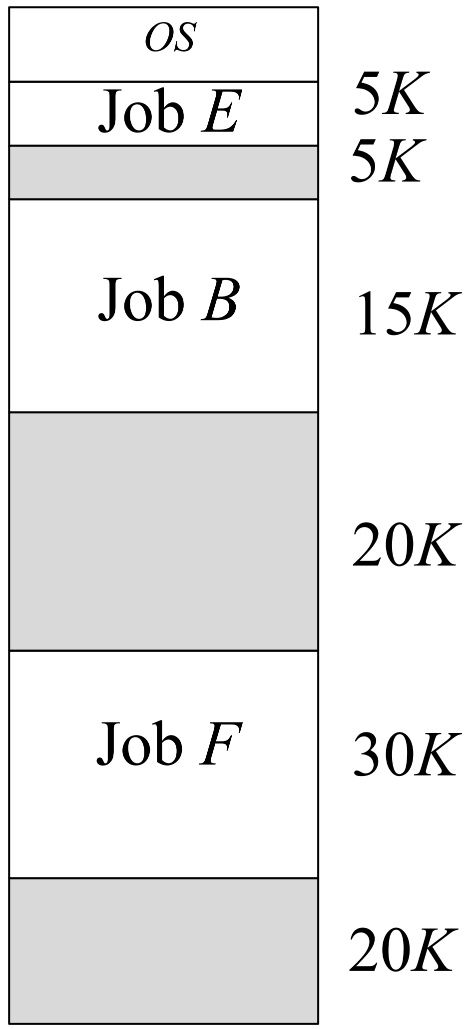
(figure 2.9)

Three snapshots of memory before and after compaction with the operating system occupying the first 10K of memory. When Job 6 arrives requiring 84K, the initial memory layout in (a) shows external fragmentation totaling 96K of space. Immediately after compaction (b), external fragmentation has been eliminated, making room for Job 6 which, after loading, is shown in (c).

Example 4

Relocatable Dynamic partition: Three snapshots (a – c) of main memory (dynamic partition system) before and after **compaction**.

Jobs:
B (15*K*)
E (5*K*)
F (30*K*)
H (30*K*) - waiting



(a)

External fragmentation total 45*K*.

Solution

__Jobs__:
B (15*K*)
E (5*K*)
F (30*K*)
H (30*K*) - waiting

(a)
External fragmentation total 45*K*.

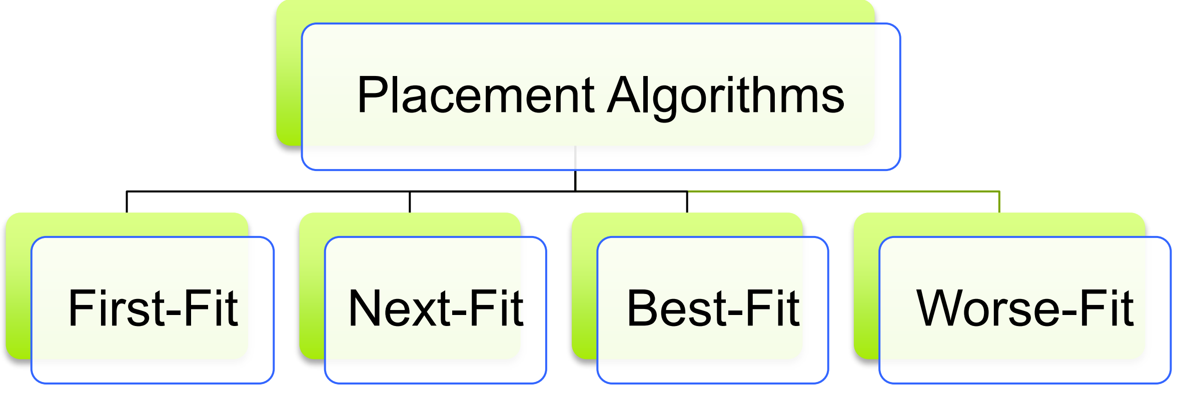
(b)
Compaction and merge to make a large empty space

(c)
Job *H* entered the memory.

Dynamic Storage-Allocation Problem

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

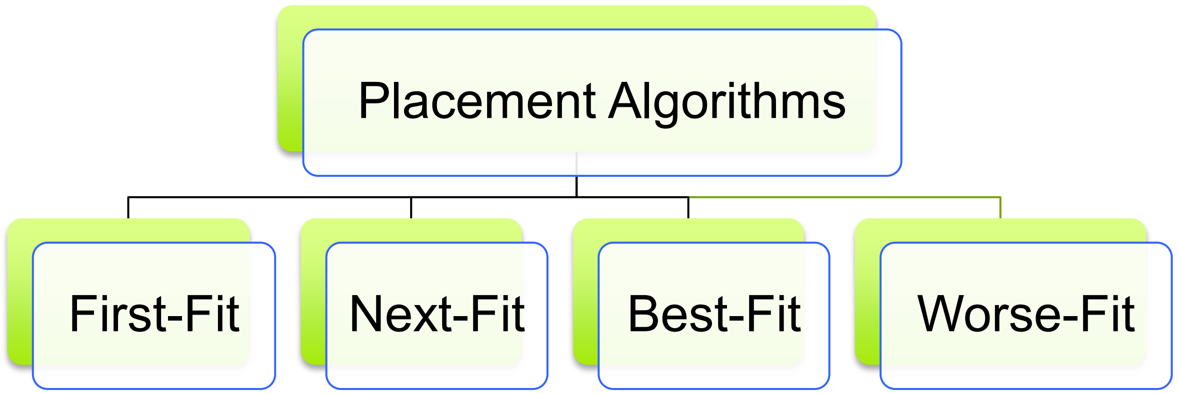
- Use the **Placement Algorithms** for selecting among free blocks of main memory.



Dynamic Storage-Allocation Problem

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

- Use the **Placement Algorithms** for selecting among free blocks of main memory.



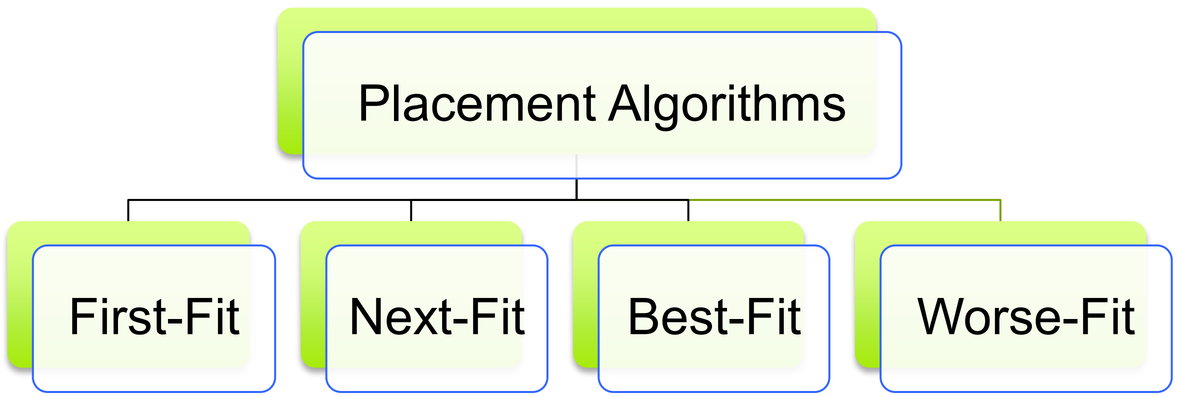
- **First-fit**:
  - Allocate the ***first hole*** that is big enough.
  - Search start at the beginning of memory.

- **Next-fit**:
  - Similar to first-fit, except the search starts at the ***last hole*** allocated.

Dynamic Storage-Allocation Problem

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

- Use the **Placement Algorithms** for selecting among free blocks of main memory.

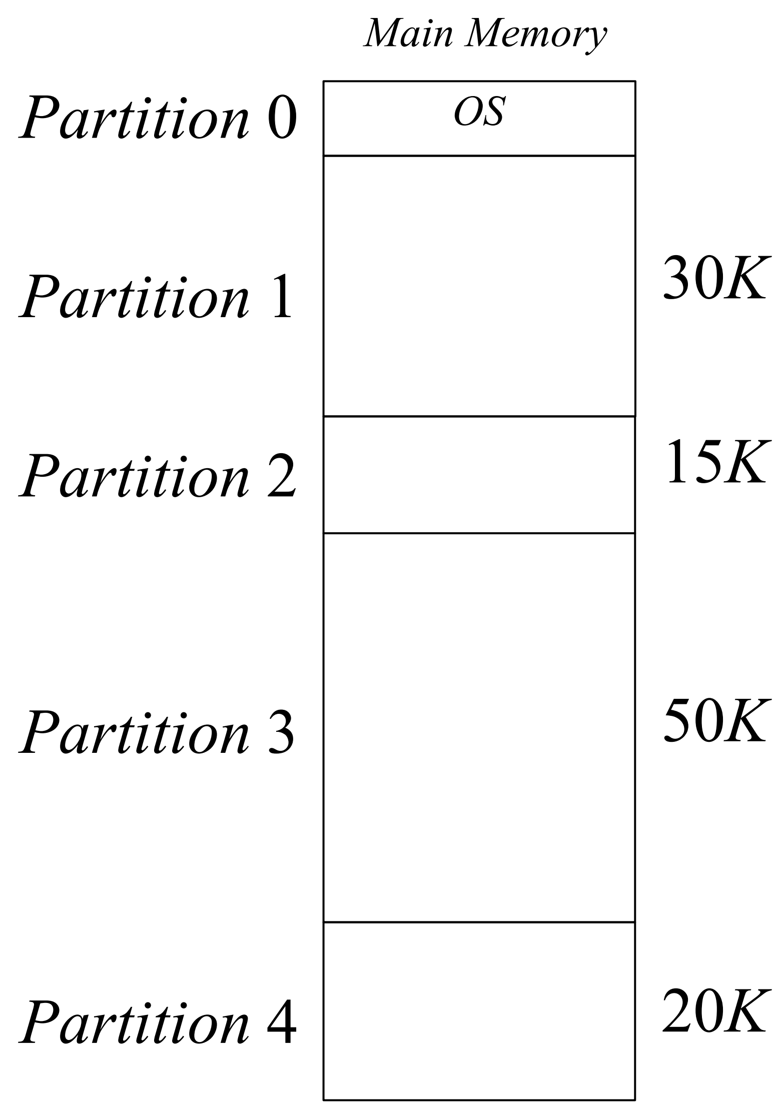


- **Best-fit**:
  - Allocate the ***smallest hole*** that is big enough;
  - must search entire list, unless ordered by size
  - Produces the smallest leftover hole

- **Worst-fit**:
  - Allocate the ***largest hole***; must also search entire list
  - Produces the largest leftover hole

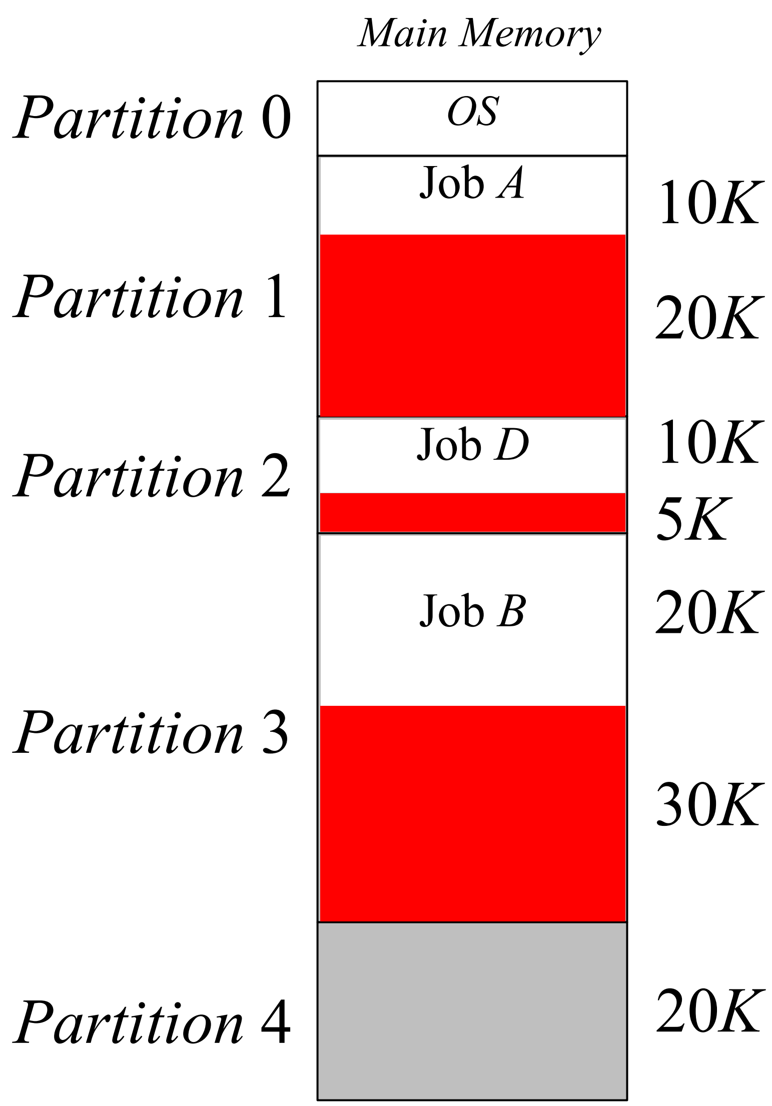
First-Fit Allocation

Example 5 : Assume fixed partition are used



Init. jobs:
A (10*K*)
B (20*K*)
C (30*K*)
D (10*K*)

Solution 5 : **First-fit allocation**:
(Assume fixed partition are used)



Init. jobs:
A (10*K*)
B (20*K*)
C (30*K*)
D (10*K*)

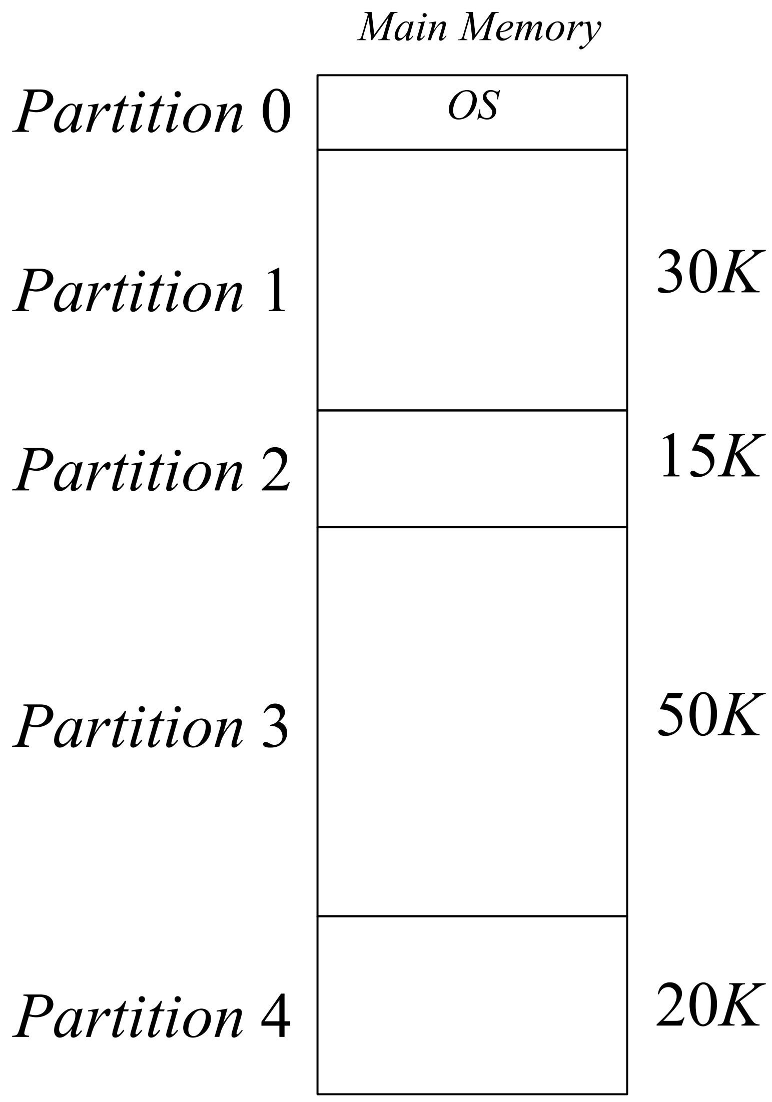
- Job *C* has to wait until a large block become available, even though there is 85*K* unused memory space.

Unused partition (20K)

Internal Fragmentation (55K)

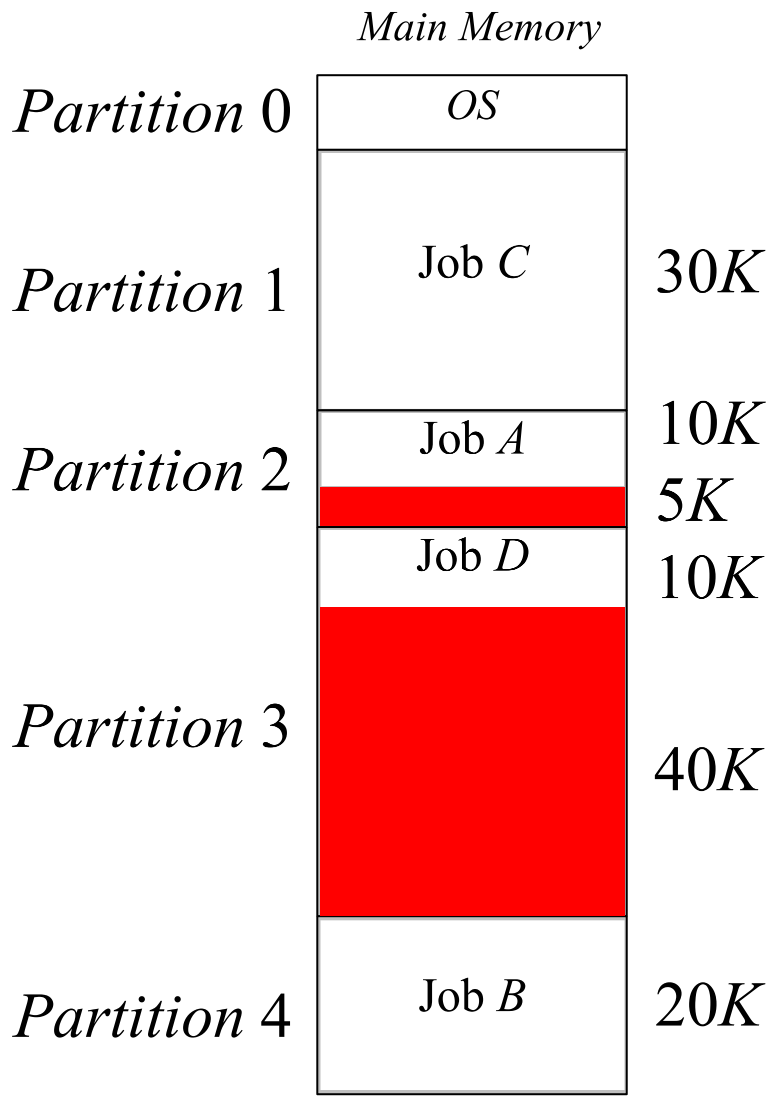
Example 6: Best-Fit Allocation

Example 6 : Assume fixed partition are used



<u>Init. jobs</u>:
A (10*K*)
B (20*K*)
C (30*K*)
D (10*K*)

Solution 6 : **Best-fit allocation**:
(Assume fixed partition are used)



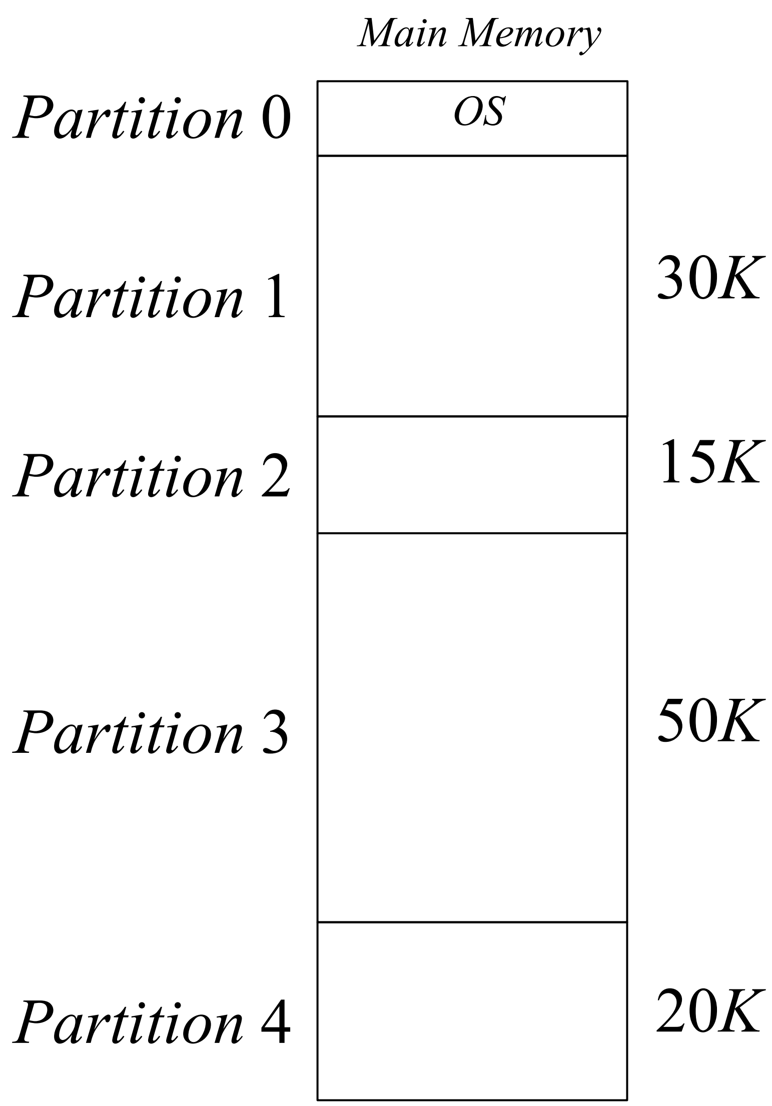
Init. jobs:
A (10K)
B (20K)
C (30K)
D (10K)

- ✓ All jobs can be allocated in the memory.
- ✓ Use the memory efficiently but slower implementation.

Internal Fragmentation (45K)

Example 7: Next-Fit Allocation

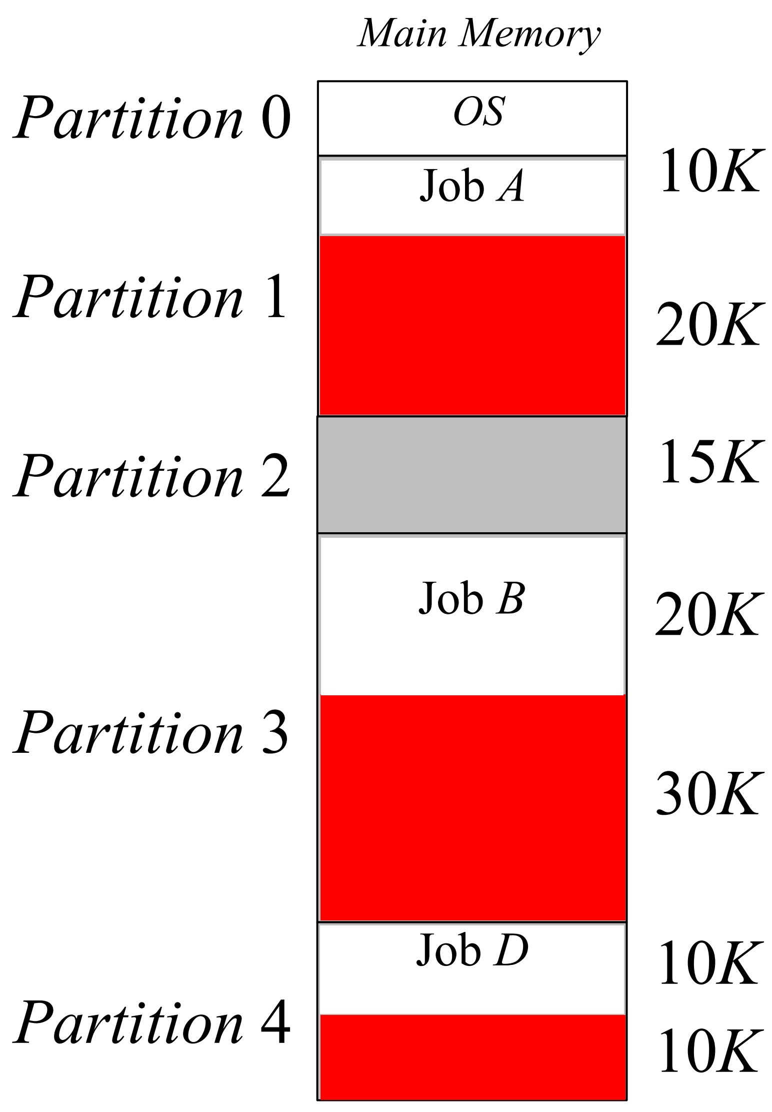
Example 7 : Assume fixed partition are used



Init. jobs:
A (10*K*)
B (20*K*)
C (30*K*)
D (10*K*)

Solution 7 : **Next-fit allocation**:

(Assume fixed partition are used)



Init. jobs:
A (10K)
B (20K)
C (30K)
D (10K)

- Job *C* has to wait until a any job exit and has enough 30*K* memory space.

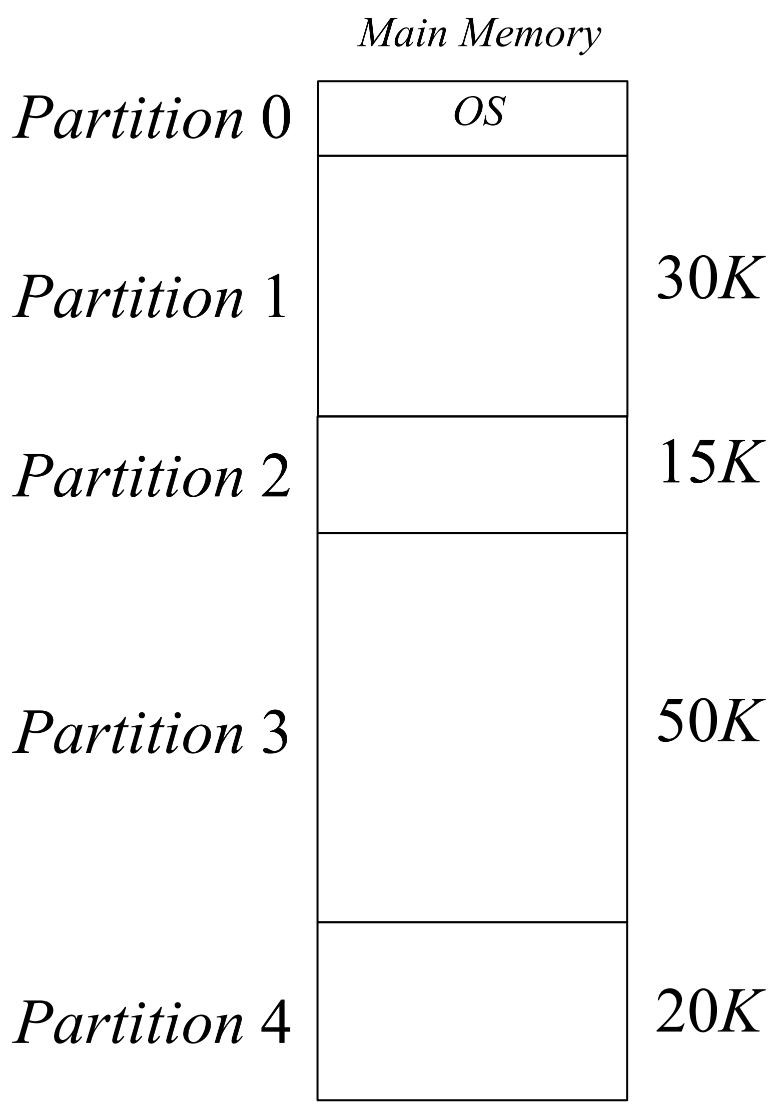
✓ This scheme also use the memory efficiently but slower implementation.

Unused partition (15*K*)

Internal Fragmentation (60*K*)

Example 8: Worst-Fit Allocation

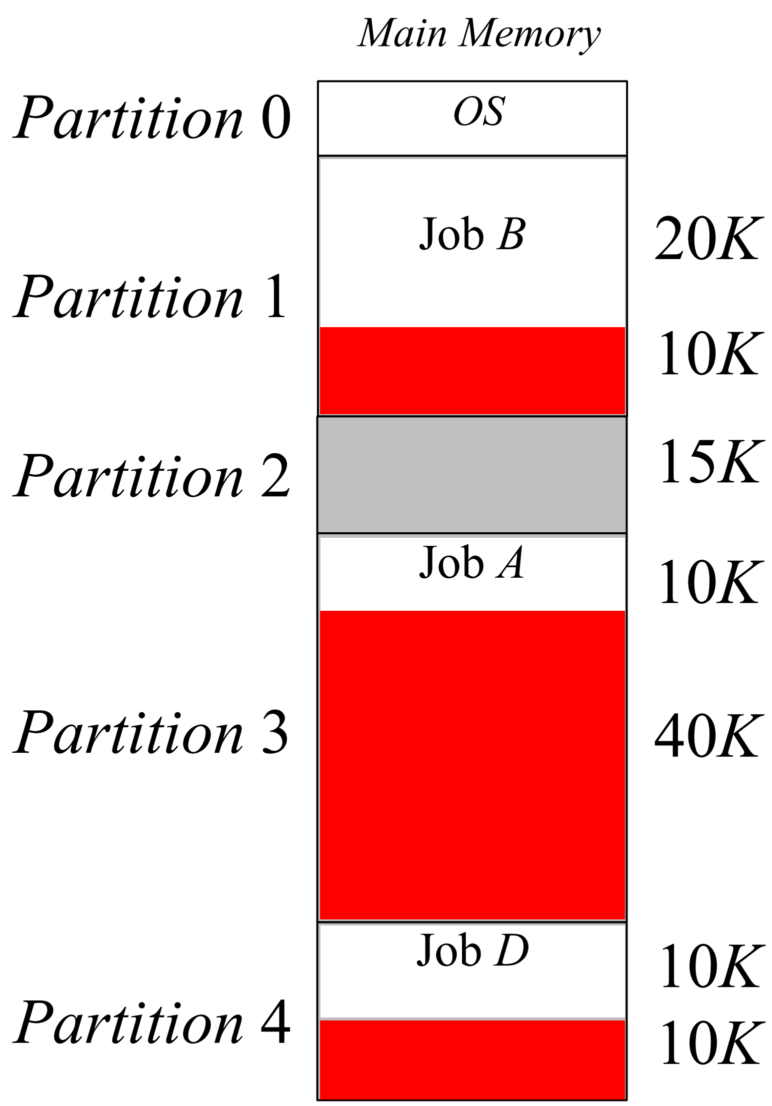
Example 8 : Assume fixed partition are used



Init. jobs:
A (10*K*)
B (20*K*)
C (30*K*)
D (10*K*)

Solution 8 : **Worst-fit allocation**:

(Assume fixed partition are used)



Init. jobs:
A (10K)
B (20K)
C (30K)
D (10K)

- Job C has to wait until a large block become available

Unused partition (15K)

Internal Fragmentation (60K)

Best-Fit vs First-Fit Allocation

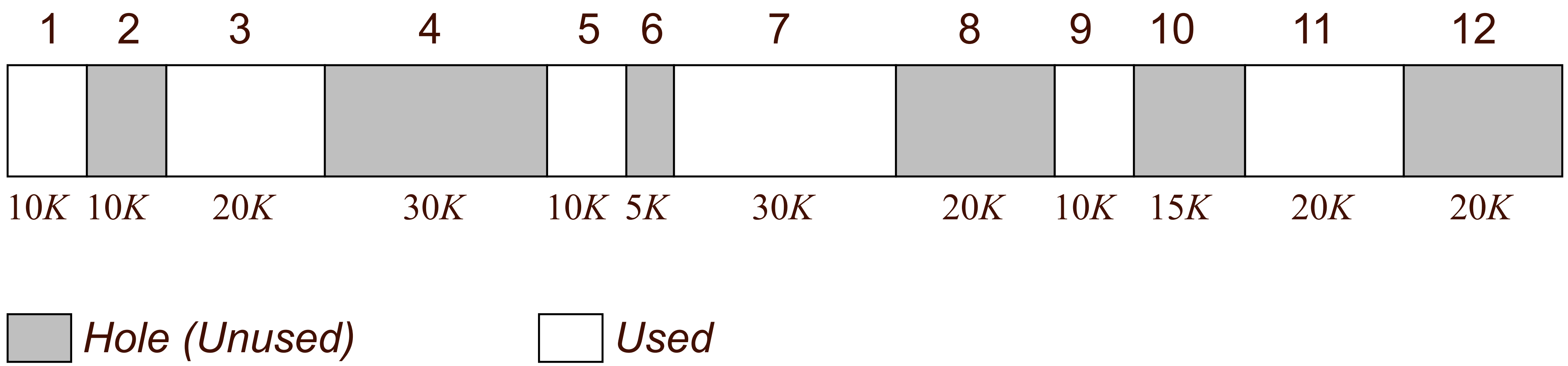
- **First-fit memory allocation**
  - Advantage: faster in making allocation
  - Disadvantage: leads to memory waste

- **Best-fit memory allocation**
  - Advantage: makes the best use of memory space
  - Disadvantage: slower in making allocation

First-fit and Best-fit better than Worst-fit in terms of speed and storage utilization.

Example 9

Given a memory allocation as specified below.



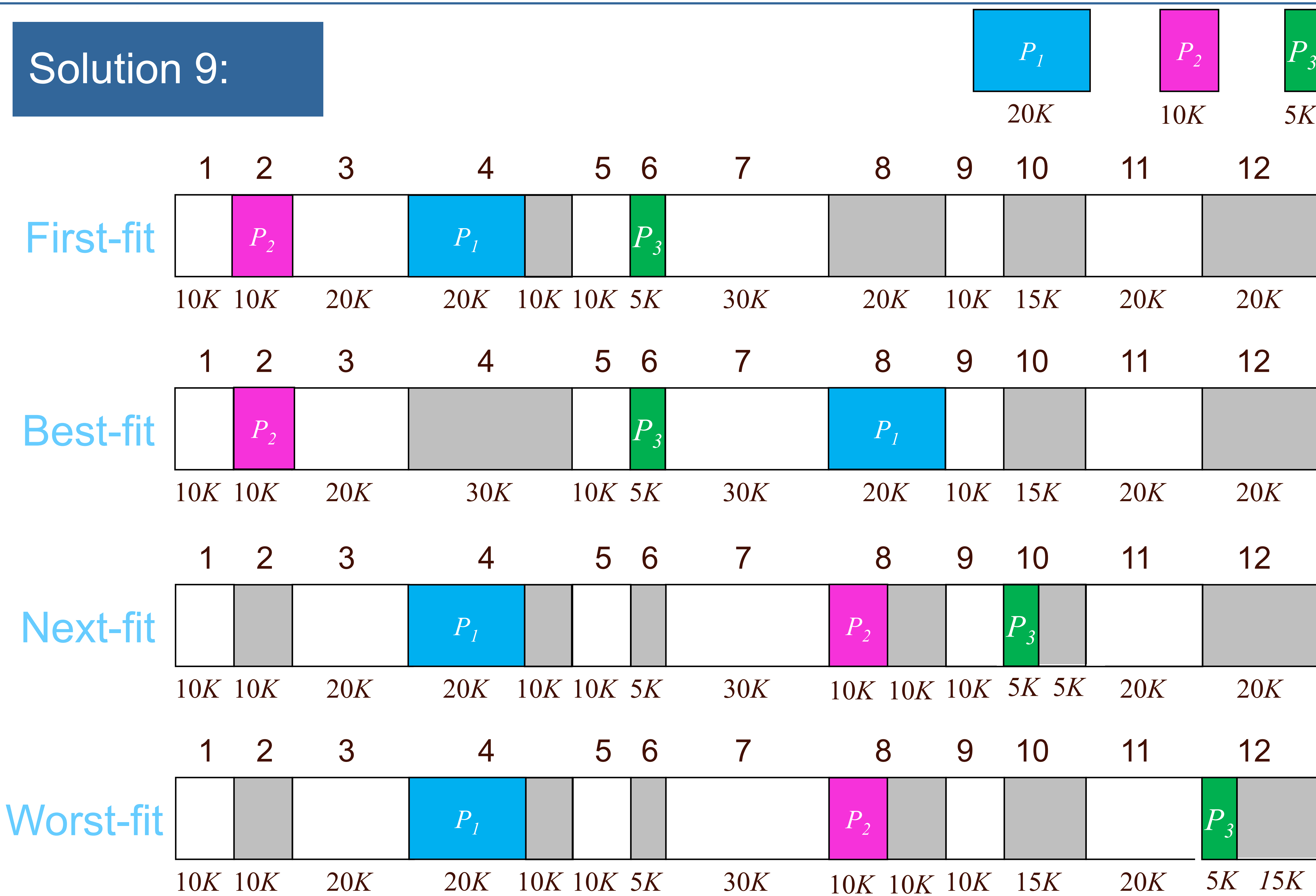
Assume partition requests are in the following order:

$$P_1 - 20K,\ P_2 - 10K,\ P_3 - 5K$$

Determine which partition each request will be allocated if the following strategy is used:

(i) First-fit (ii) Next-fit (iii) Best-fit (iv) Worst-fit

Solution 9:



P_1 20K
P_2 10K
P_3 5K
1 2 3 4 5 6 7 8 9 10 11 12
First-fit
P_2 P_1 P_3
10K 10K 20K 20K 10K 10K 5K 30K 20K 10K 15K 20K 20K
1 2 3 4 5 6 7 8 9 10 11 12
Best-fit
P_2 P_3 P_1
10K 10K 20K 30K 10K 5K 30K 20K 10K 15K 20K 20K
1 2 3 4 5 6 7 8 9 10 11 12
Next-fit
P_1 P_2 P_3
10K 10K 20K 20K 10K 10K 5K 30K 10K 10K 10K 5K 5K 20K 20K
1 2 3 4 5 6 7 8 9 10 11 12
Worst-fit
P_1 P_2 P_3
10K 10K 20K 20K 10K 10K 5K 30K 10K 10K 10K 15K 20K 5K 15K

Summary

- Early memory management techniques
  - Fixed partitions, dynamic partitions, and relocatable dynamic partitions

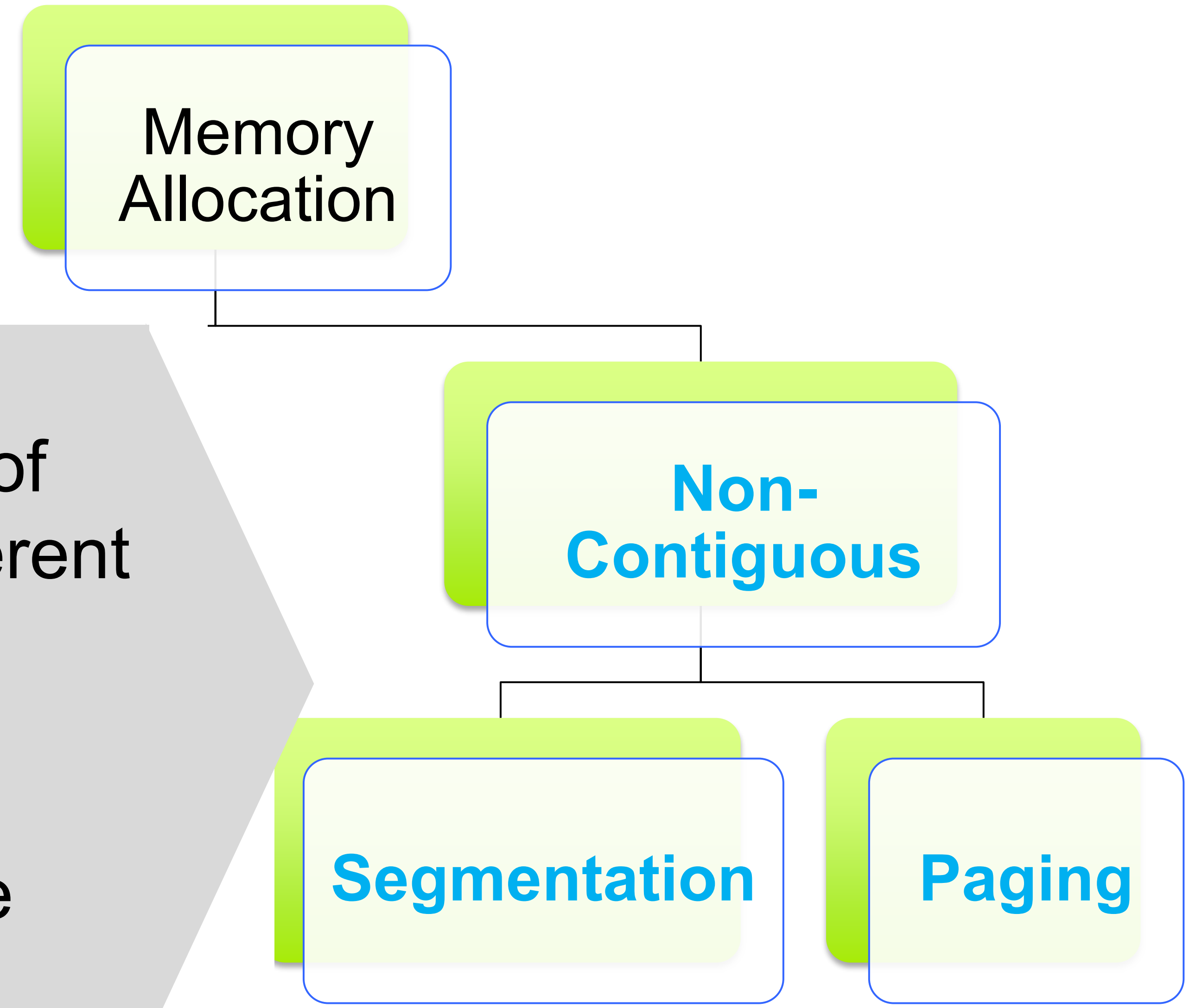
- Common requirements of four memory management techniques
  - Entire program loaded into memory
  - Contiguous storage
  - Memory residency until job completed

- Each places severe restrictions on job size

- Sufficient for first three generations of computers

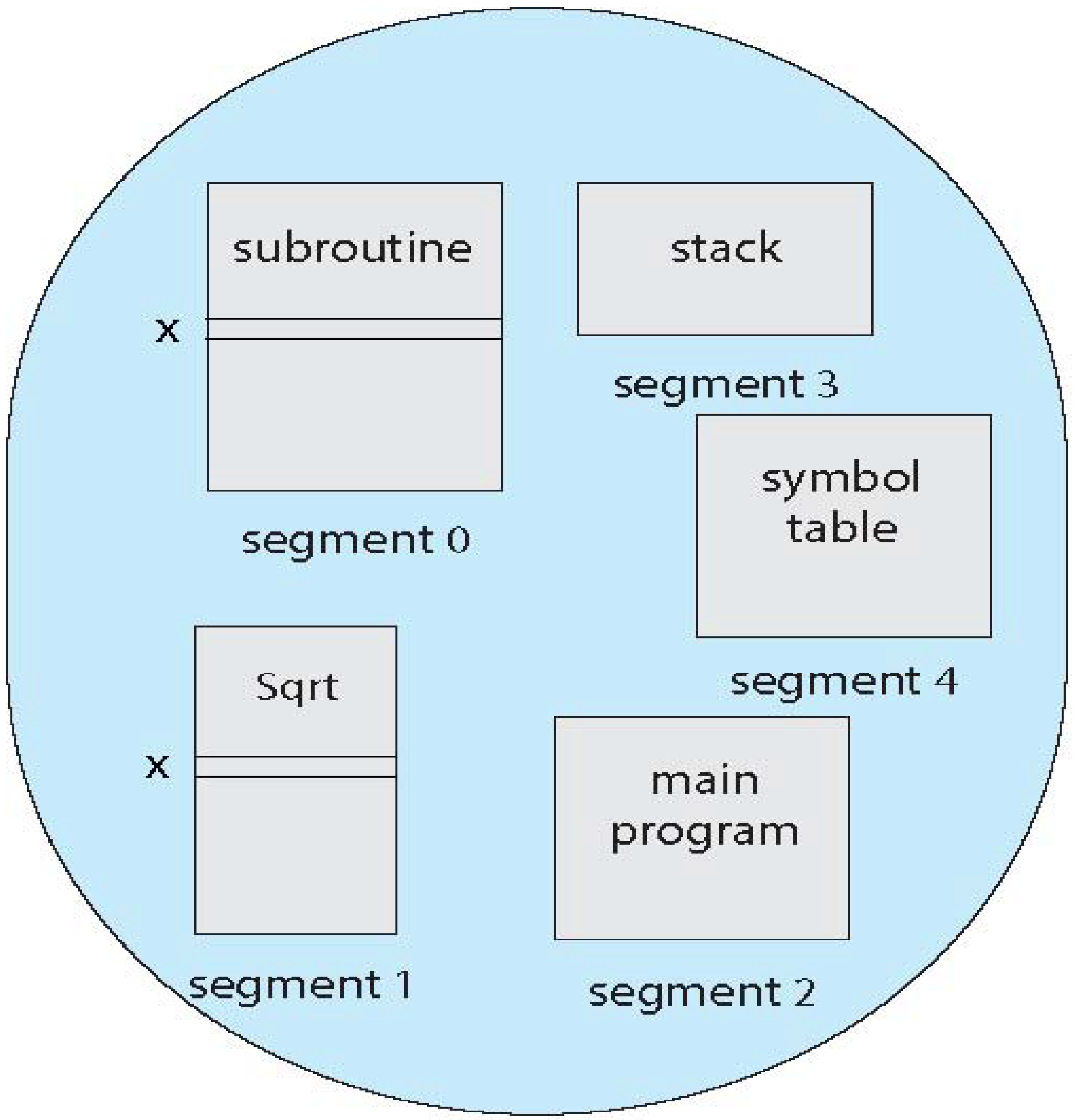
- New modern memory management trends
  - Common characteristics of memory schemes
    - Programs are not stored in contiguous memory
    - Not all segments reside in memory during job execution

5. NON CONTIGUOUS MEMORY ALLOCATION



- Partition a program into several small units, each of which can reside in a different parts of the memory.
- Need hardware support.
- Various methods to do the partitions:

(a) 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,	*global variables,*
procedure,	*common block,*
function,	*stack,*
method,	*local variables,*
object	*symbol table,*

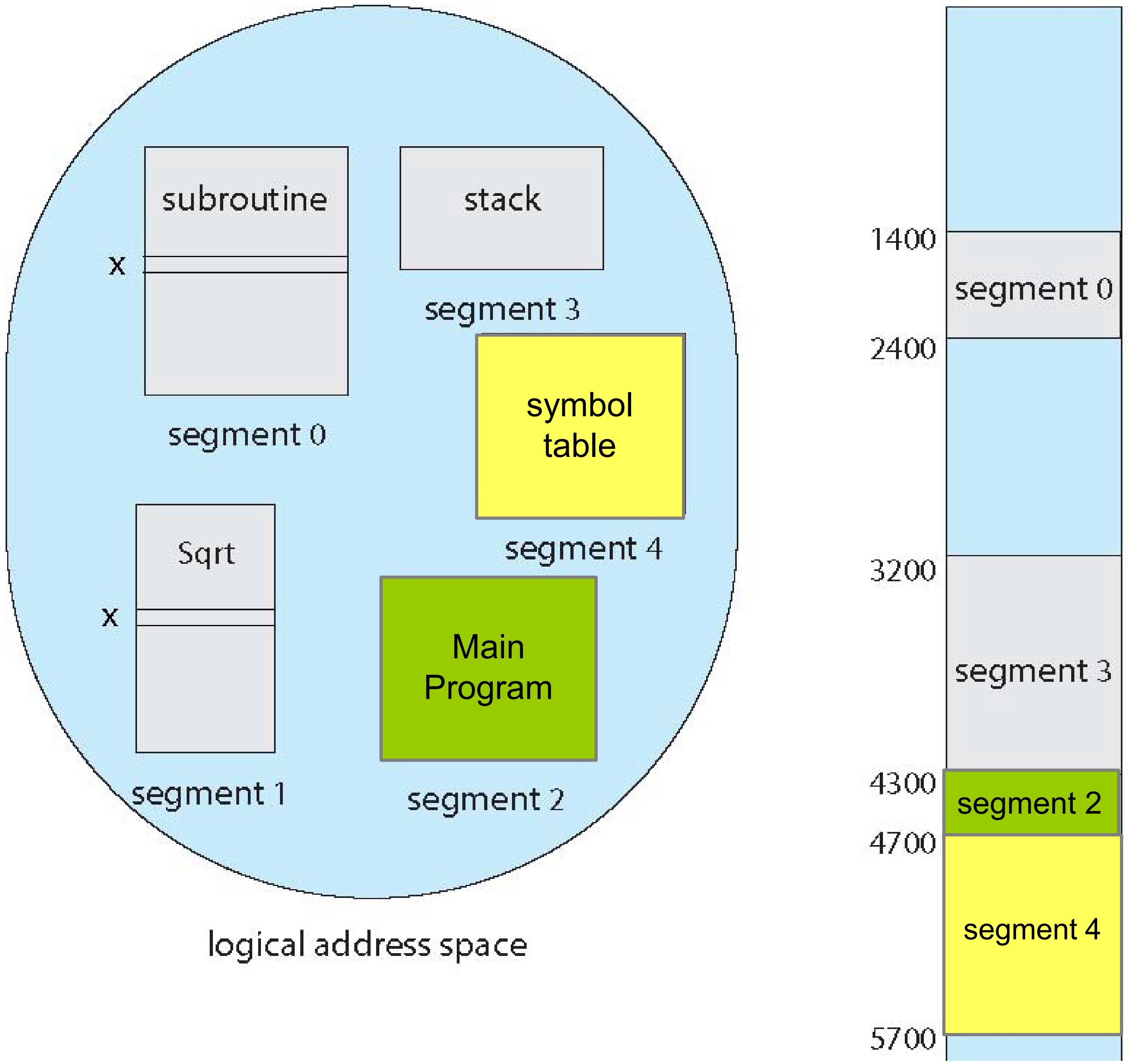
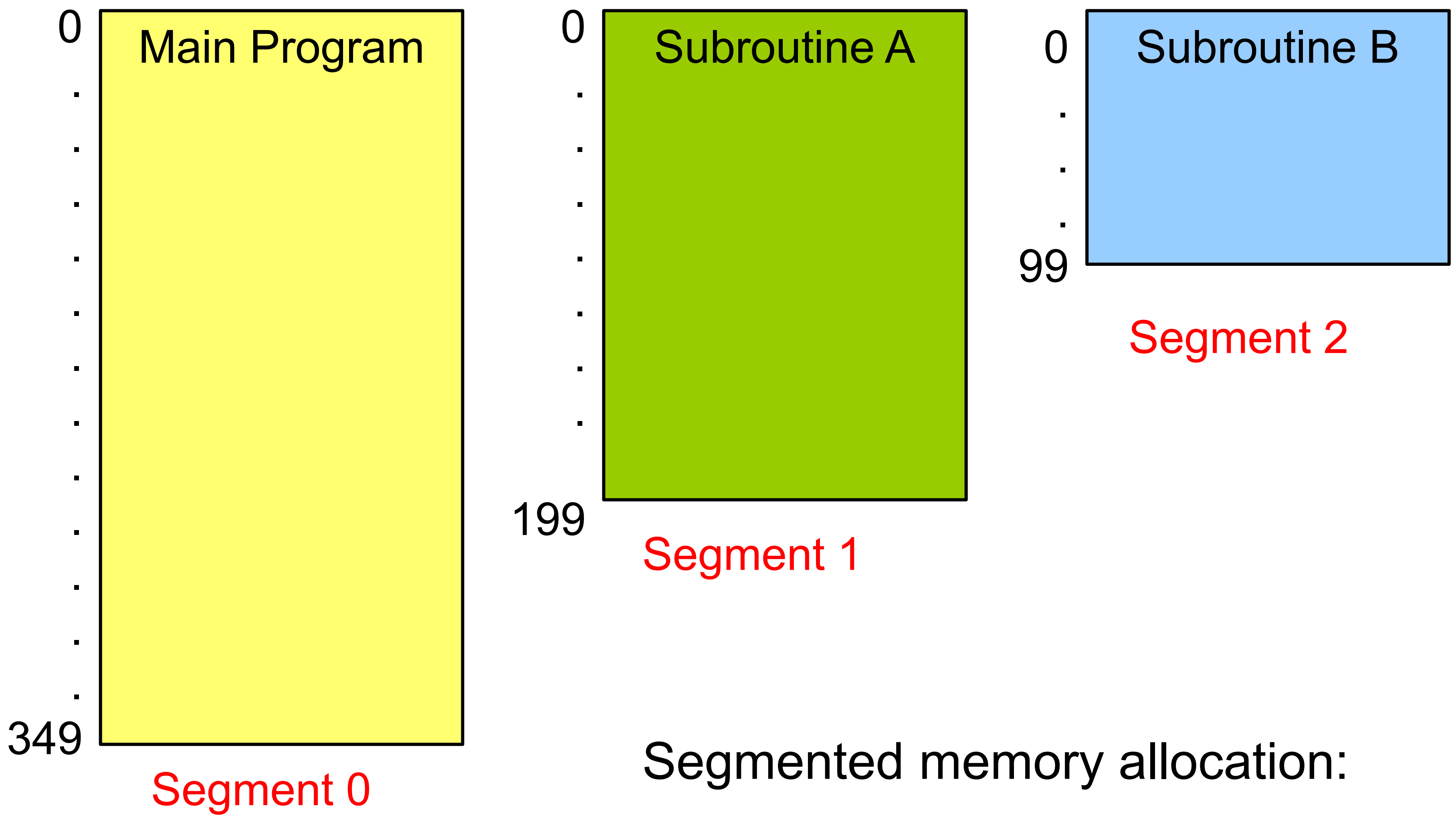


Figure 8.2 Logical and Physical Memory

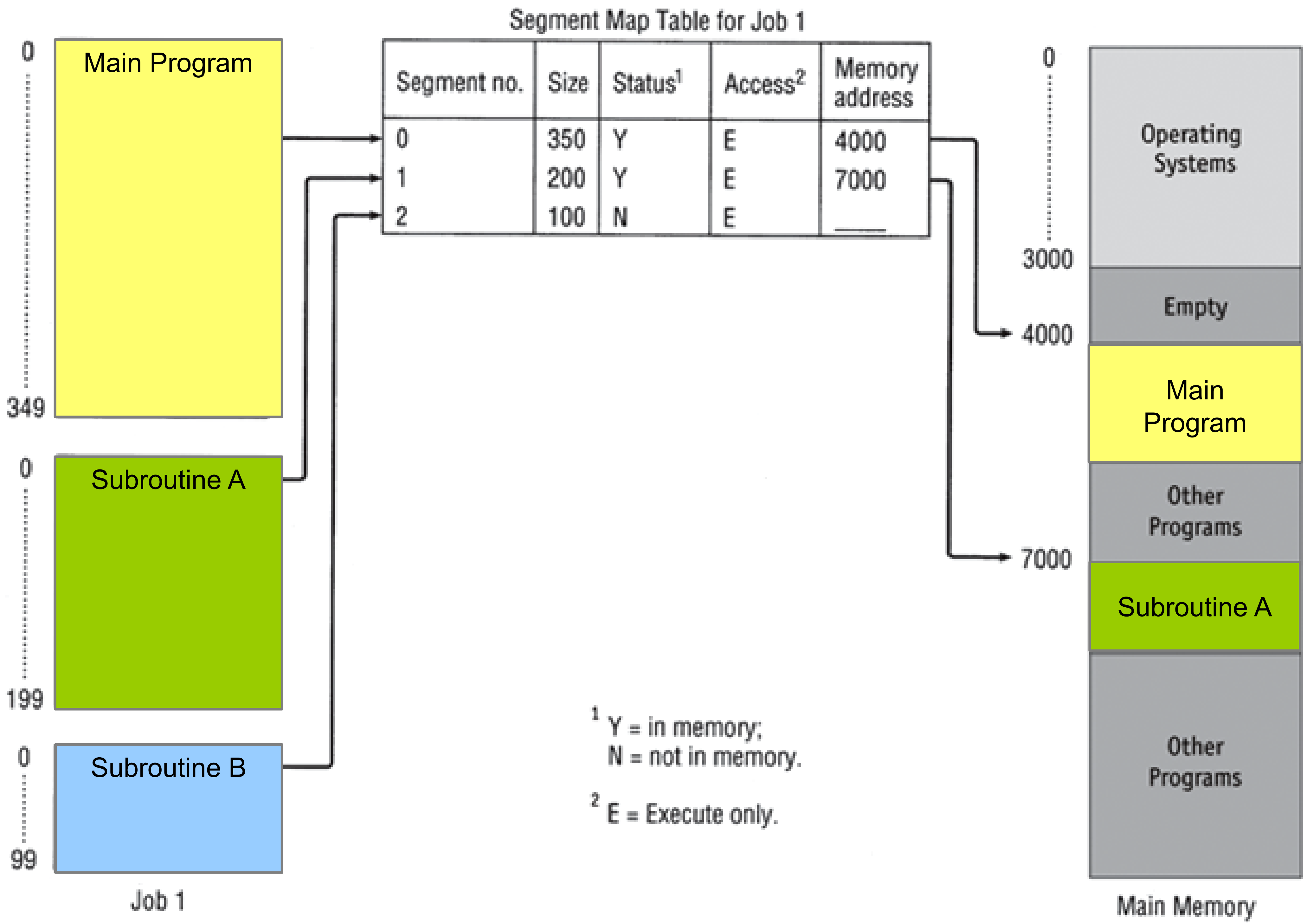
Segmented Memory Allocation

- Each job is divided into several segments.
  - Segments are different sizes.
  - One for each module containing related functions.
- Main memory is allocated dynamically.
- Program's structural modules determine segments.
  - Each segment is numbered when compiled/assembled.
  - Need **Segment Table**.



Segmented memory allocation:

- The job includes a Main Program, Subroutine *A*, and Subroutine *B*.
- It is one job divided into three segments.



Segment no.	Size	Status[1]	Access[2]	Memory address
0	350	Y	E	4000
1	200	Y	E	7000
2	100	N	E	____

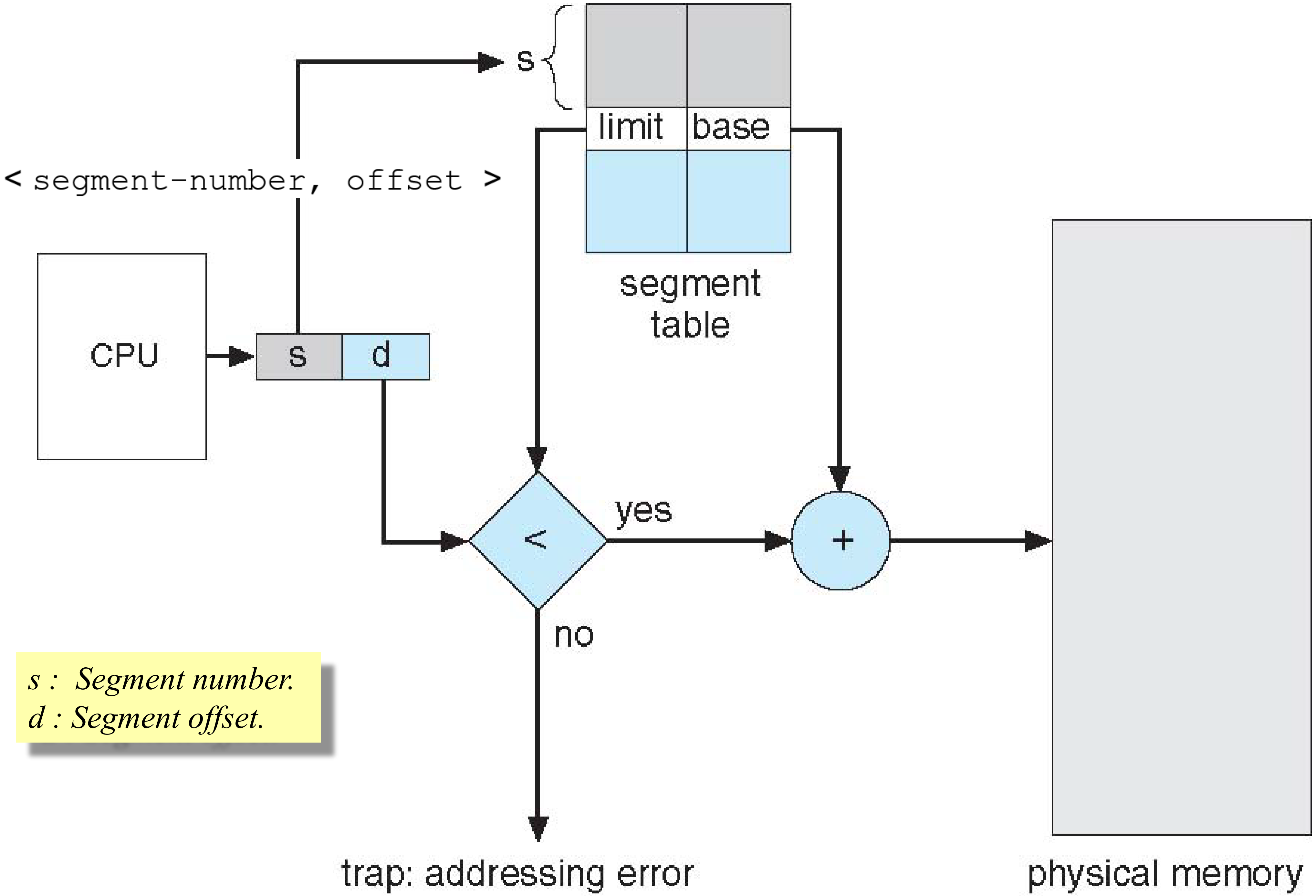
(figure 3.14) *The Segment Map Table tracks each segment for Job 1.*

Segmentation Hardware

- Addressing scheme require segment number and displacement / offset
- Logical address consists of a two tuple:

  `< segment-number, offset >`

- Need to map a two-dimensional logical addresses to a one-dimensional physical address.
- Done via **segment table**:
  - **base** – contains the starting physical address where the segments reside in memory
  - **limit** – specifies the length of the segment.



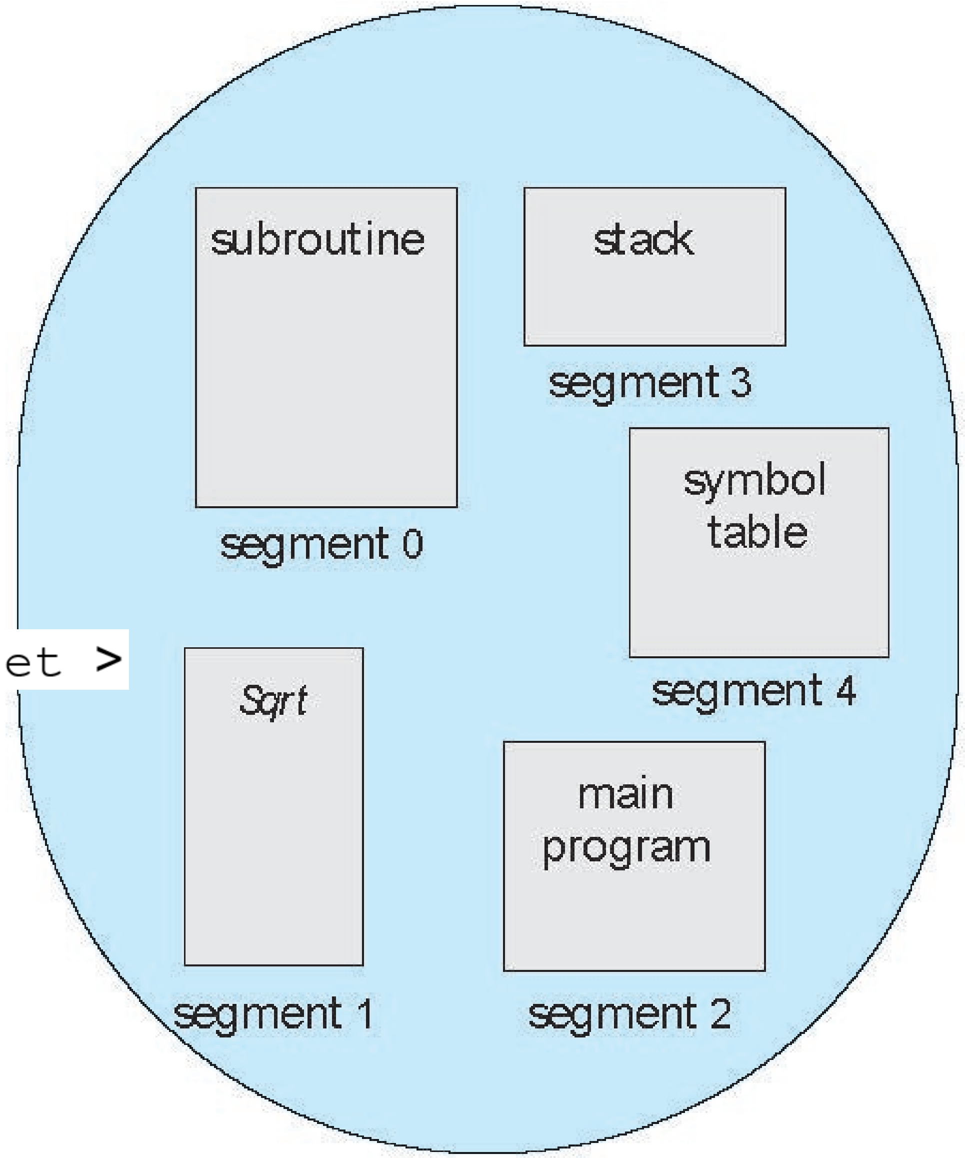
s : *Segment number.*
d : *Segment offset.*

Segmentation hardware.

Example 9

What is the physical memory address for the logical address 2 : 53?

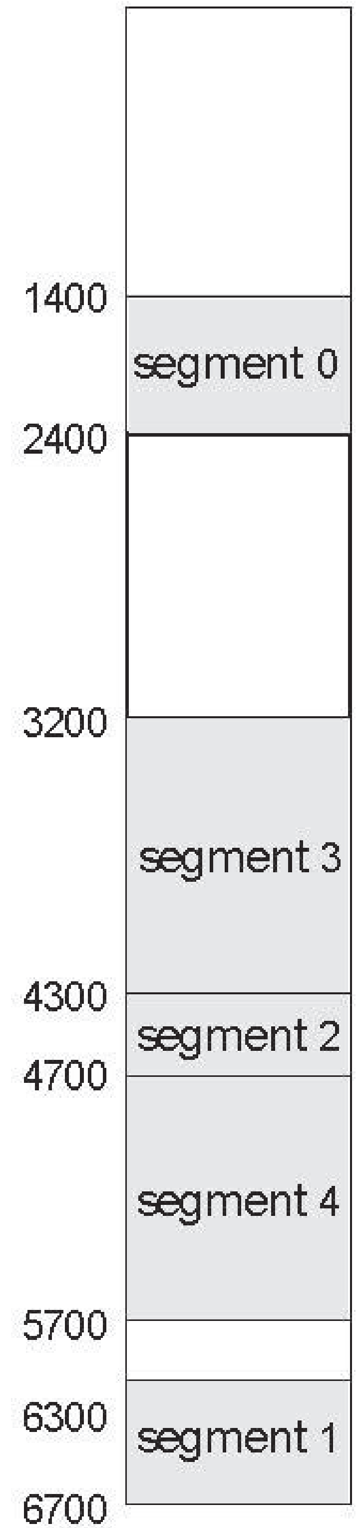
< segment-number, offset >



logical address space

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

segment table



physical memory

Example of segmentation.

Solution

What is the physical memory address for the logical address 2 : 53 ?

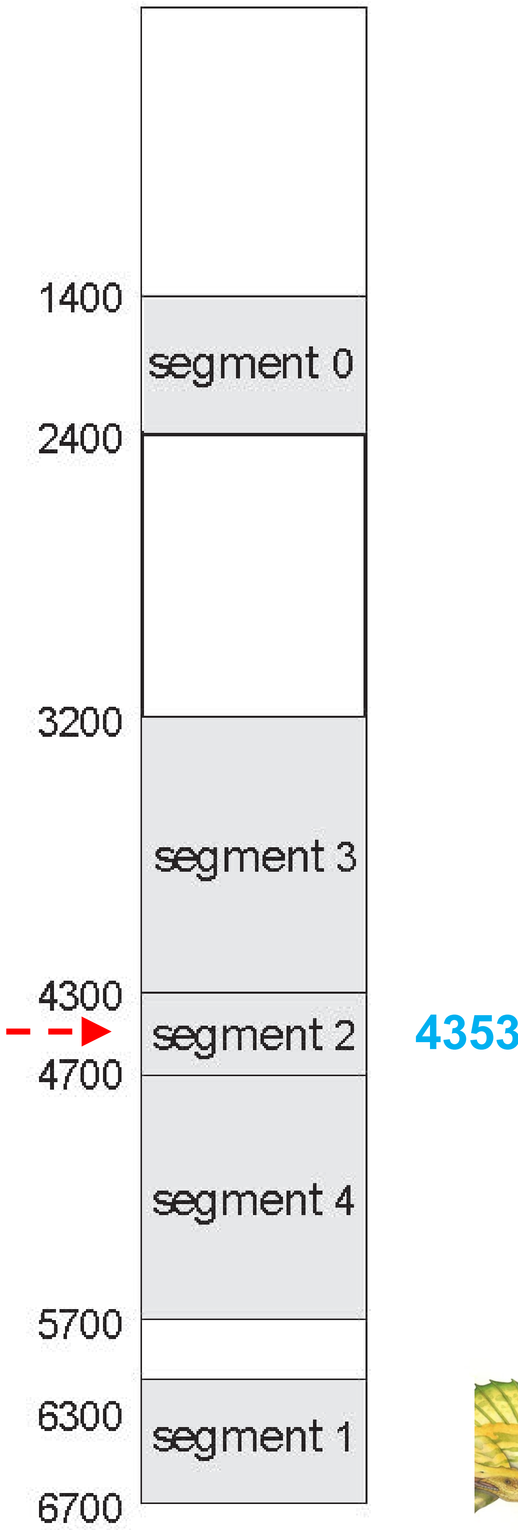
Segment 2 is 400 bytes long and begins at location 4300.

→ Physical address

= 4300 + **53** = **4353**

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

segment table

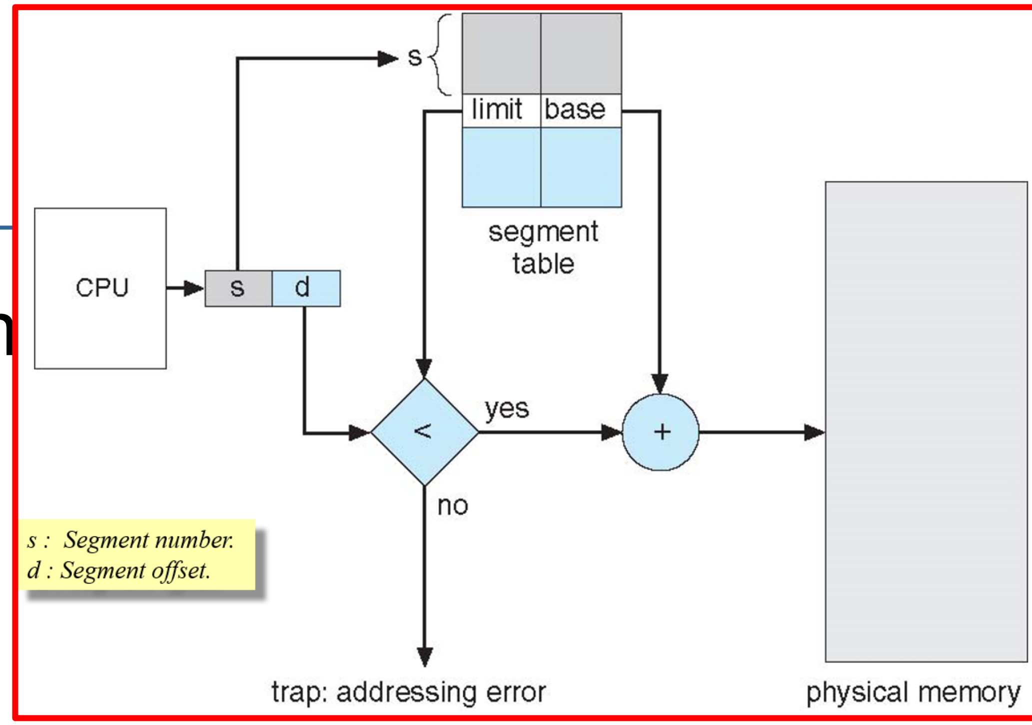


physical memory

What is the physical m
address?

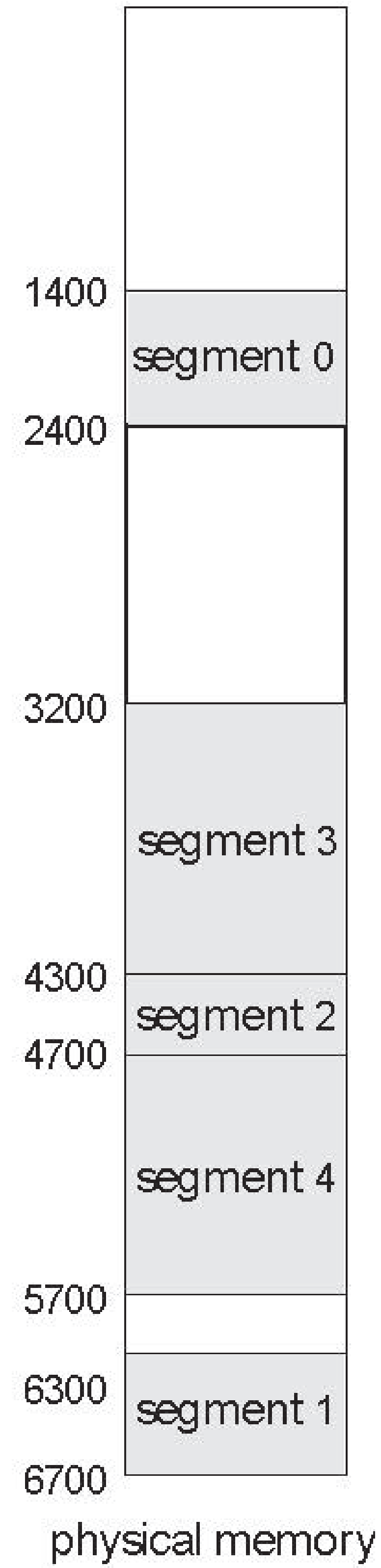
a) 3 : 852

b) 0 : 1222



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

segment table



Answer:

a) A reference to segment 3, byte 852, is mapped to Physical memory address:

- **= 3200 (the base of segment 3) + 852**
- **= 4052.**

b) A reference to byte 1222 of segment 0 would result in a trap to the OS, as this segment is only 1000 bytes long.

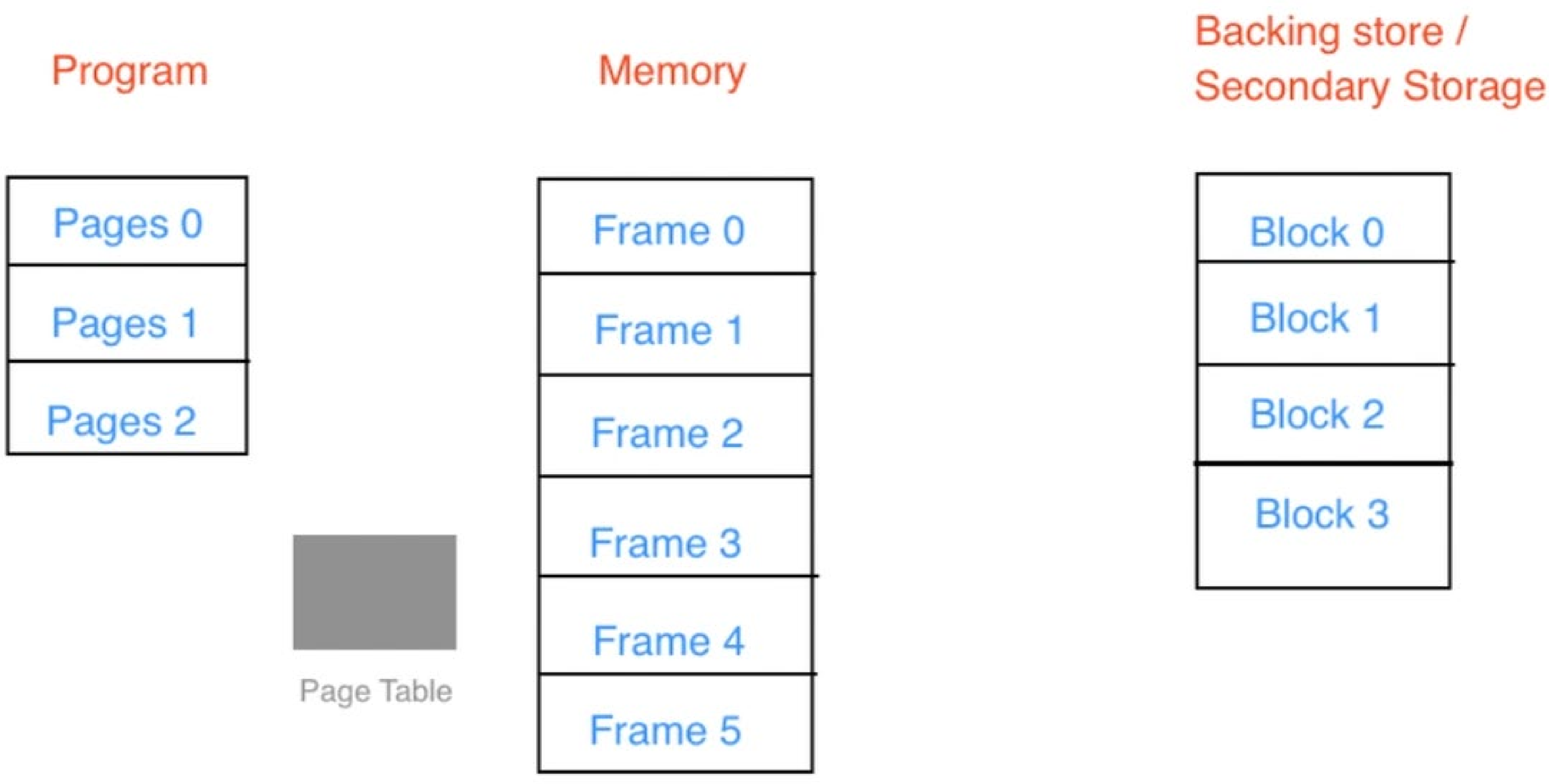
(b) Paging

- Physical address space of a process can be in **non-contiguous**.

- Process is divided into **fixed-size** blocks, each of which may reside in a different part of physical memory.

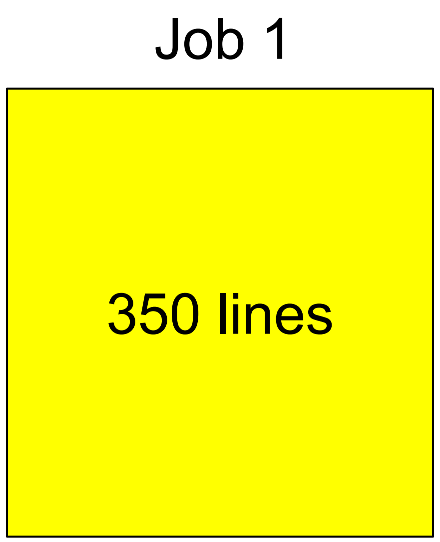
1. Divide physical memory into fixed-size blocks, called **frames**:
   - Size of a frame is power of 2 between 512 Bytes and 8192 bytes (2^9 - 2^{13}).
2. Divide logical memory into blocks of same size as frames, called **pages**.

3. Backing store, where the program is permanently residing, is also split into storage units called **blocks**, which are the same size as the frames and pages.



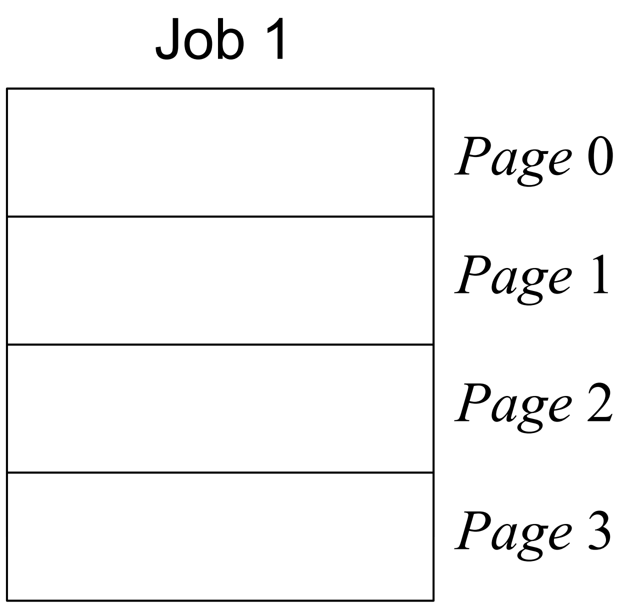
Paging

Example 9a : Consider a program with 350 lines long. Each page frame in a system can hold 100 lines. What happen in paged memory allocation?



Paging

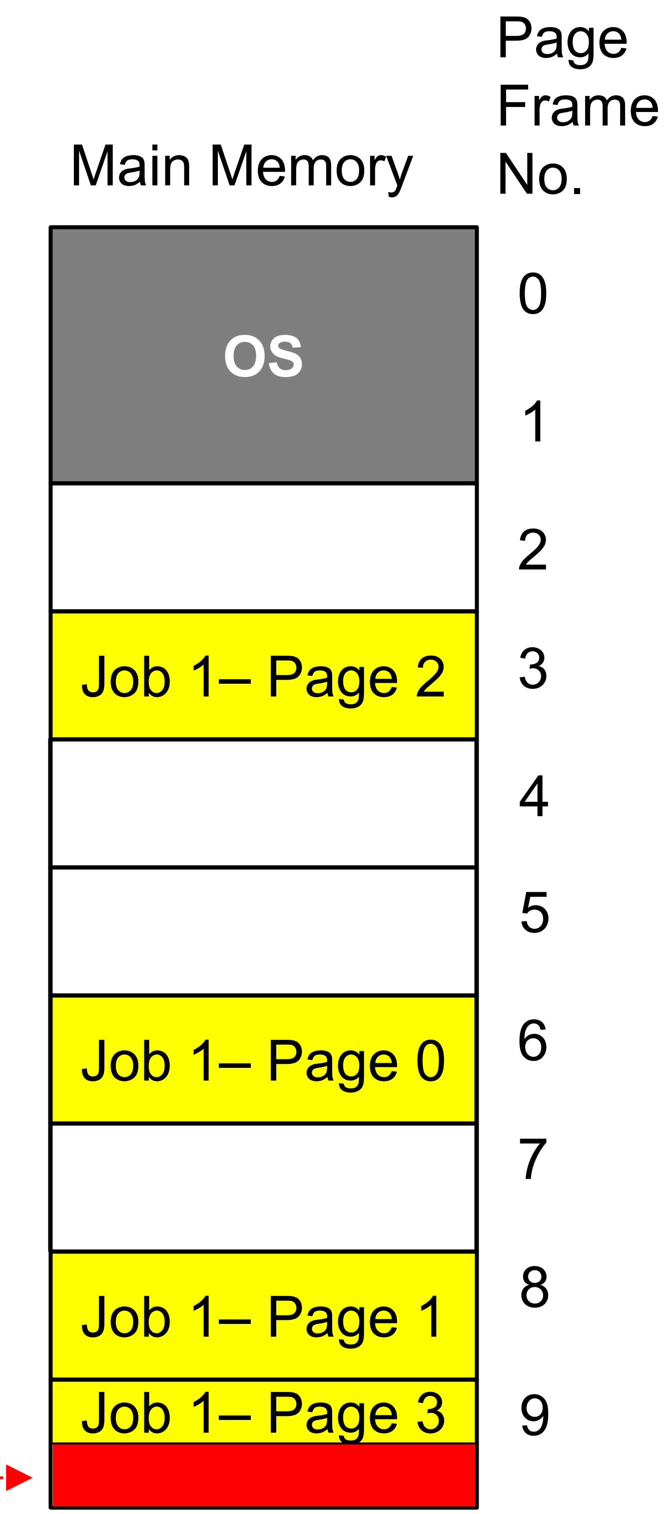
Solution 9a :



Page size = Frame size

Page Table	
0	6
1	8
2	3
3	9

Page# → Frame#

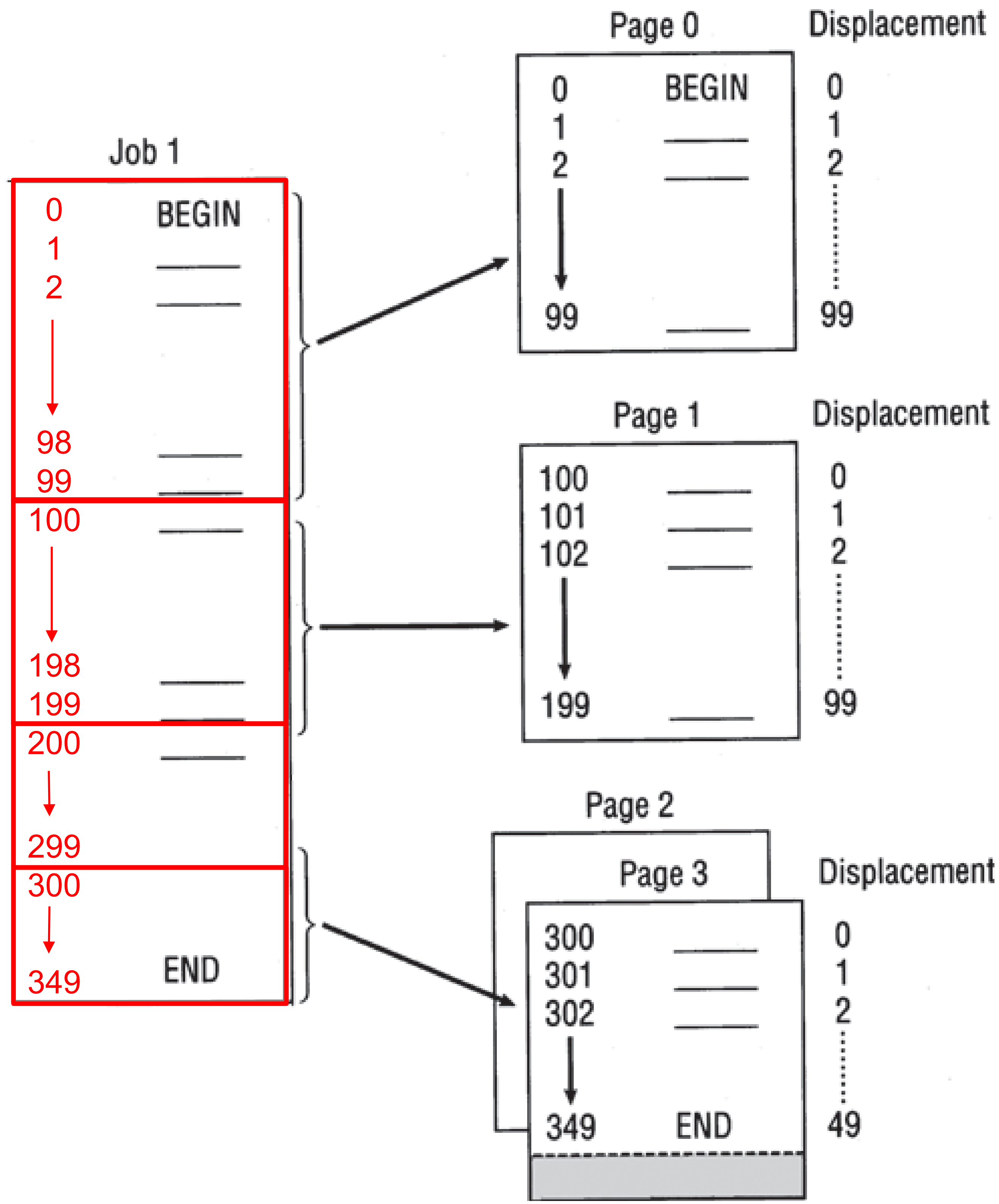


Internal Fragmentation

Paging

Example 9b :

- Job 1 is 350 lines long and is divided into 4 pages of 100 lines each.



Paging

- **Advantages**:
  - Allows job allocation in non-contiguous memory.
  - Efficient memory use.
- **Disadvantages**:
  - Increased overhead from address resolution.
  - Internal fragmentation in last page.
- Page size selection is crucial:
  - **Too small**: generates very long page table.
  - **Too large**: excessive internal fragmentation.

Paging

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.

Assume:
✓ logical address space = 2^m
✓ page size = 2^n

Page number	Page offset
p	d
$m - n$	n

Paging

Example 10 : Considered size of logical address is 16 MB with it page size 1 KB. Find the bits for the page offset of the logical memory address.

Solution 10 :

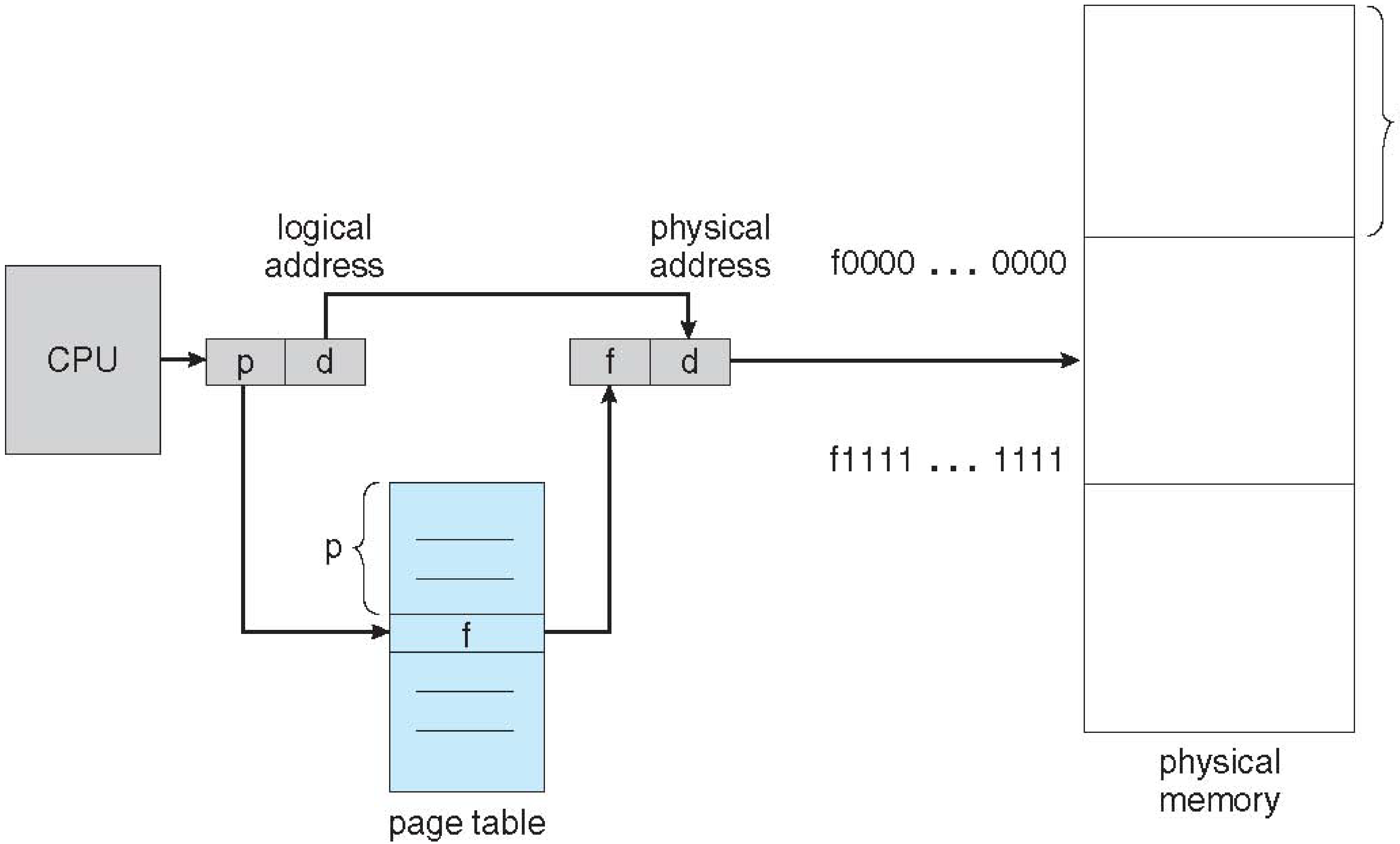
– size of logical address = 16 MB = 16 x 2^{20} Bytes
= 2^4 x 2^{20} = 2^{24}

– page size = 1 KB = 1024 Byte = 2^{10}

Page number	Page offset
p	d
$= m - n$ $= 24 - 10$ $= 14$ *bits*	$= n$ $= 10$ *bits*

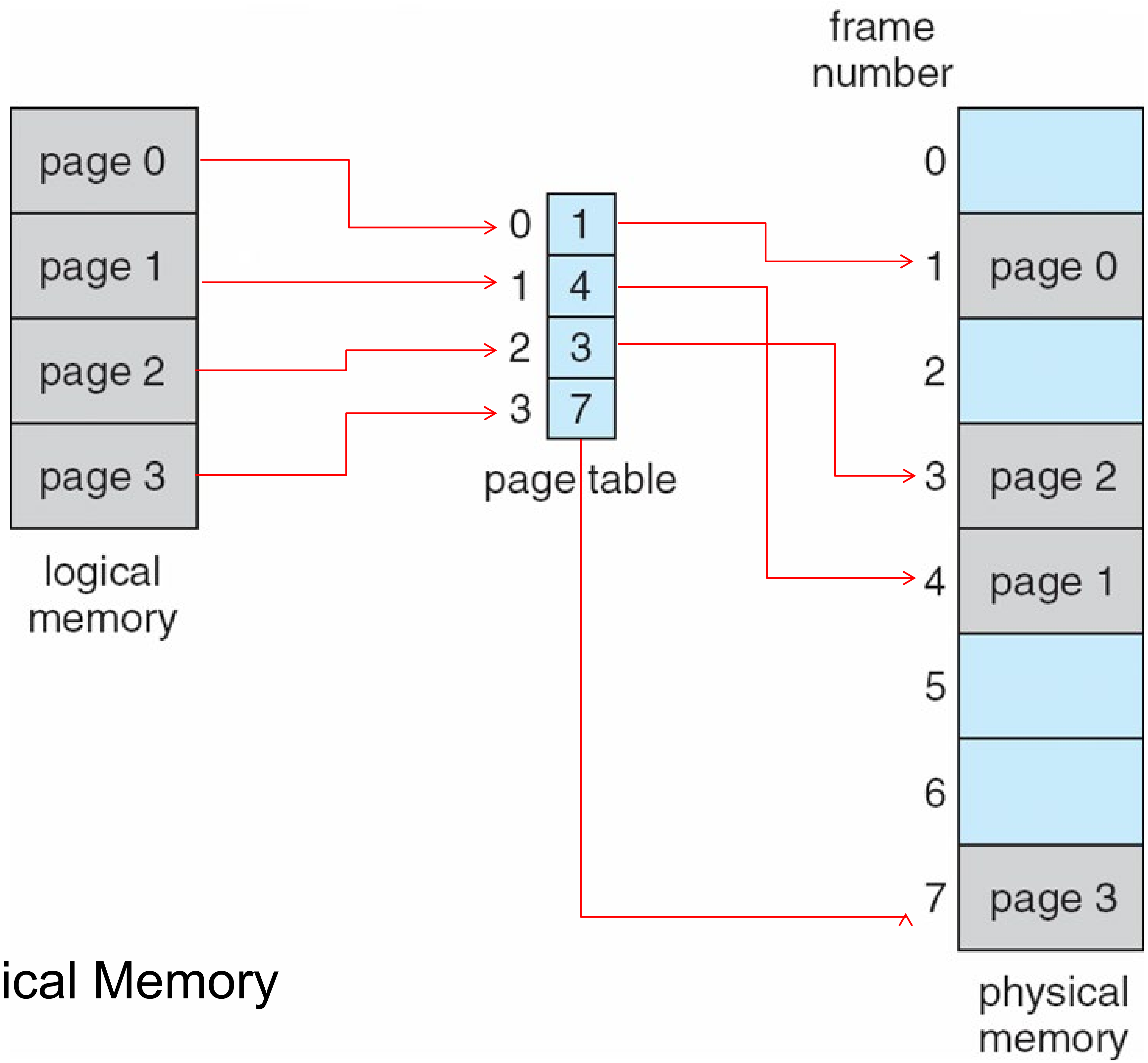
Paging

Paging hardware



Paging

Example 11 :

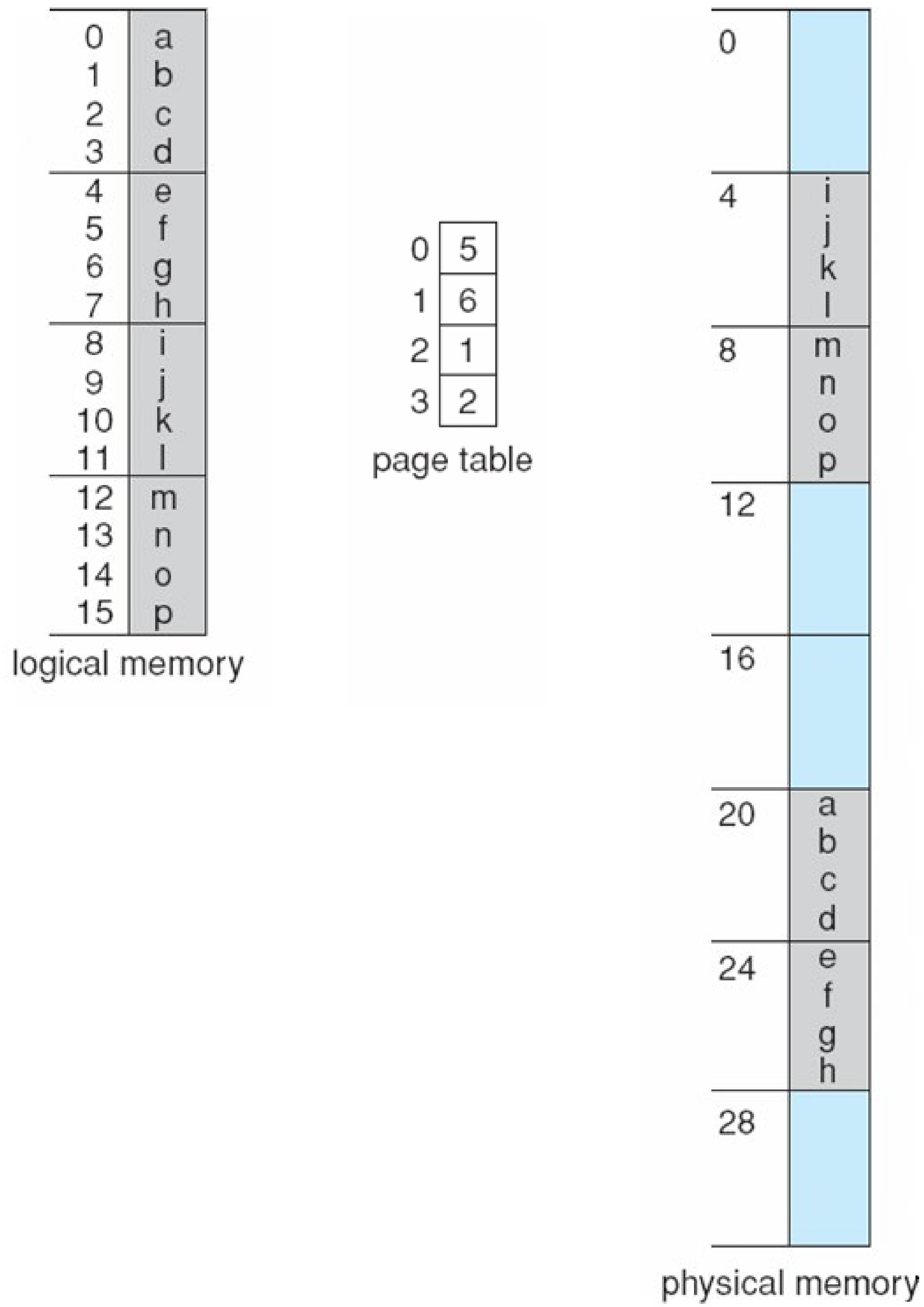


Paging Model:
Logical and Physical Memory

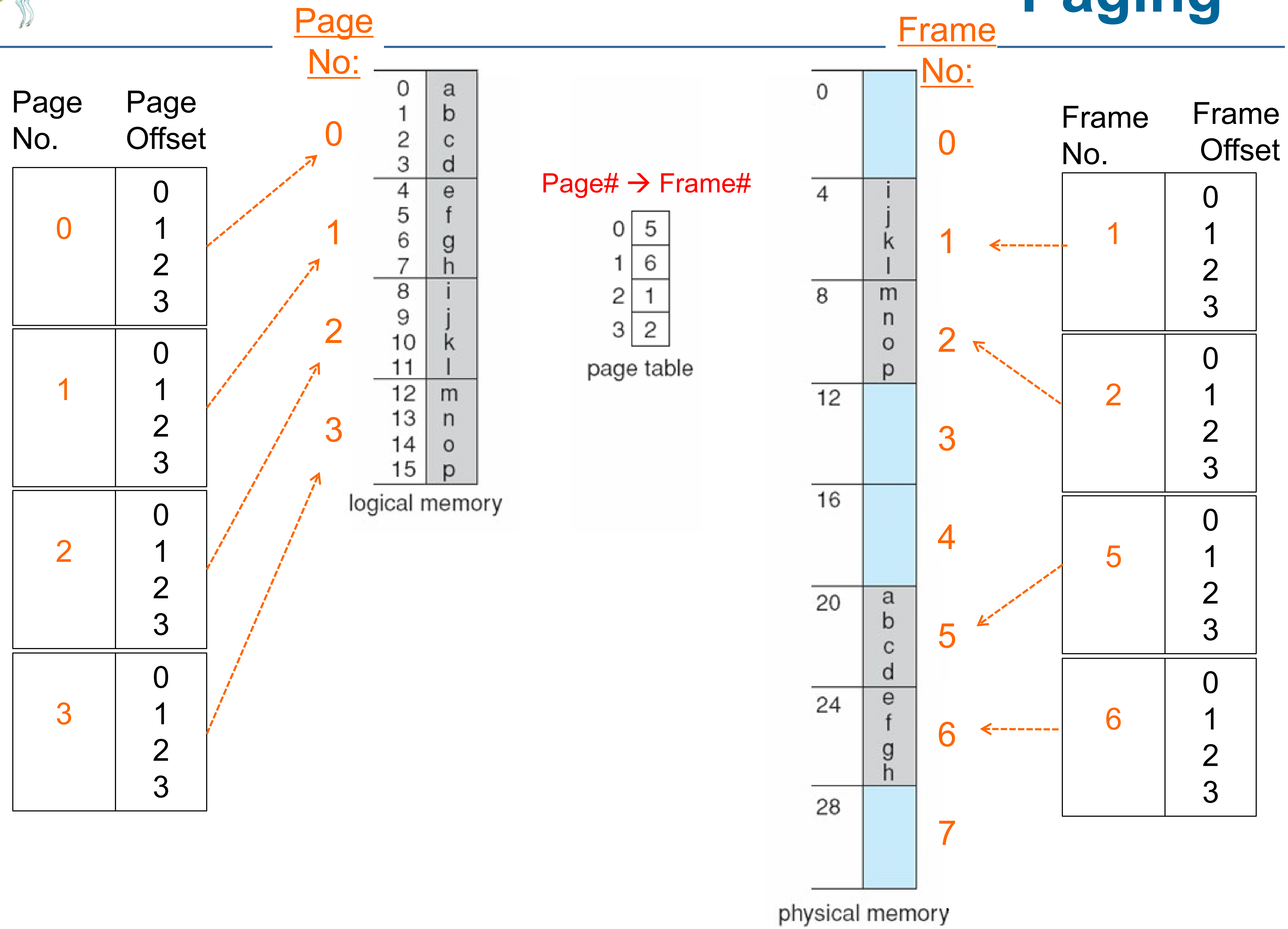
Paging

Example 12a :

Assume $m = 4$ and $n = 2$ and 32-byte memory and 4-byte pages.



Paging



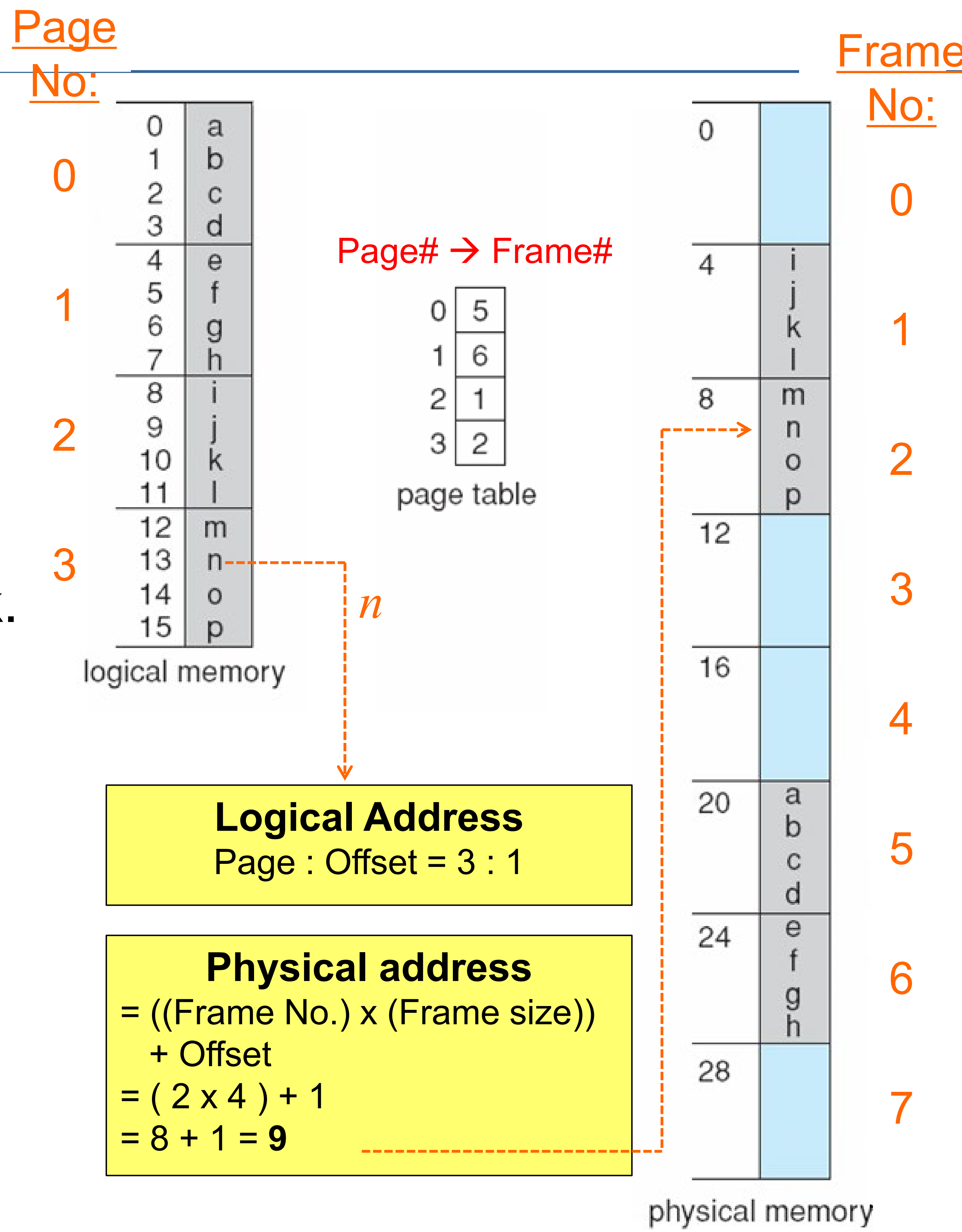
Page No.
Page Offset
0
1
2
3
0
1
2
3
Page No:
0
1
2
3
0 a
1 b
2 c
3 d
4 e
5 f
6 g
7 h
8 i
9 j
10 k
11 l
12 m
13 n
14 o
15 p
logical memory
Page# → Frame#
0 5
1 6
2 1
3 2
page table
0
4 i j k l
8 m n o p
12
16
20 a b c d
24 e f g h
28
physical memory
Frame No:
0
1
2
3
4
5
6
7
Frame No.
Frame Offset
1
2
5
6
0
1
2
3

Paging

Example 12b :

Based on **Example 12a**, get the physical address for n.

Show your work.



Logical Address
Page : Offset = 3 : 1

Physical address
= ((Frame No.) x (Frame size)) + Offset
= (2 x 4) + 1
= 8 + 1 = **9**

Paging

Exercise 8.6 :

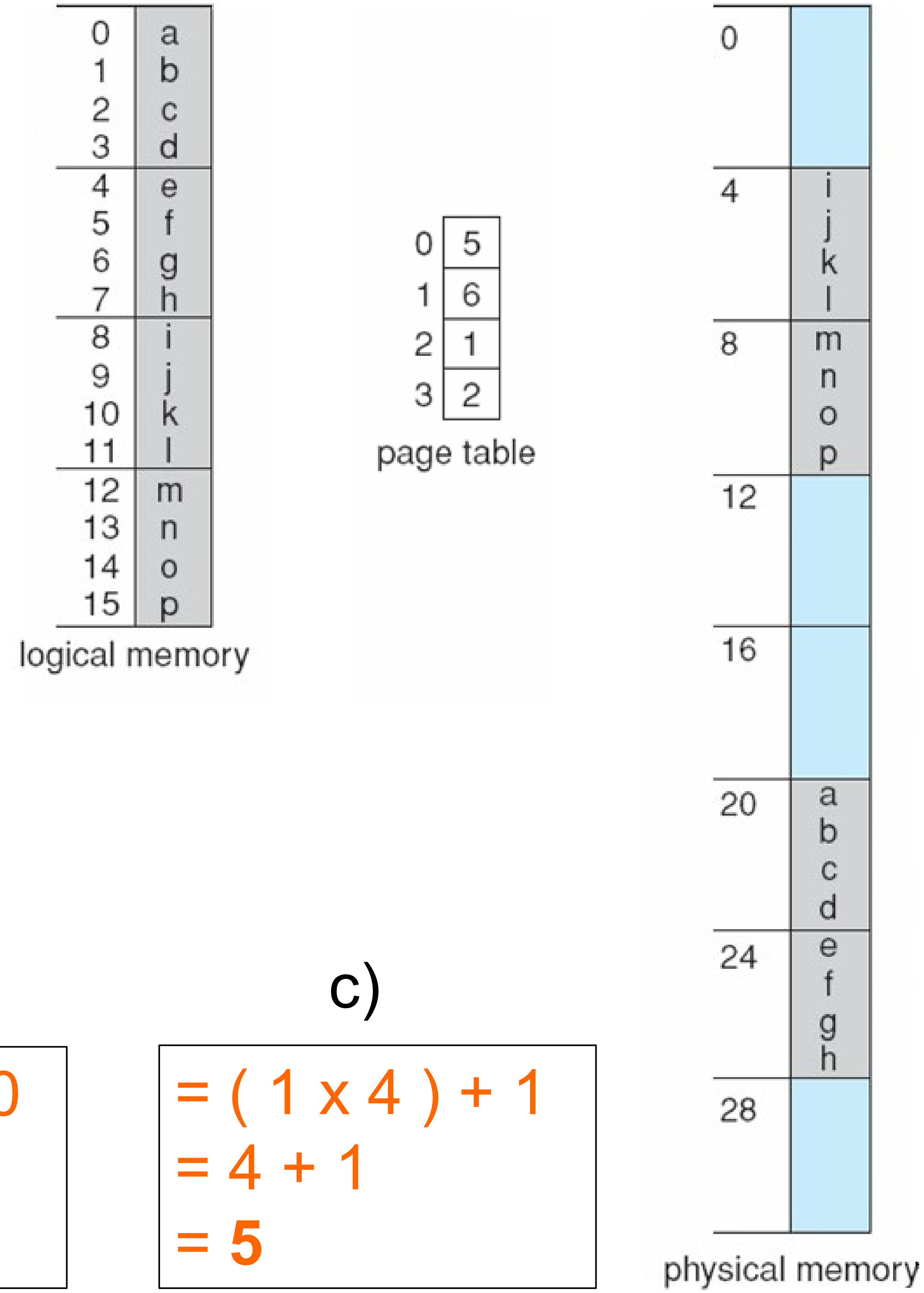
What is the physical address for the following logical addresses with given the number of the page and the offset?

a) 0 : 3

b) 1 : 0

c) 2 : 1

Physical address
= ((Frame No.) x (Frame size)) + Offset



Solution 8.6 :

a)

= (5 x 4) + 3
= 20 + 3
= **23**

b)

= (6 x 4) + 0
= 24 + 0
= **24**

c)

= (1 x 4) + 1
= 4 + 1
= **5**

Home work

Exercise 8.7 : Consider a paging system with 4 pages of logical address space, a page size of 4 bytes and a physical memory of 64 bytes as depicted in the following diagram. [8 *marks*]

Logical Memory

Logical address	Content
0	**A**
1	**B**
2	**C**
3	**D**
. .	. .
8	**G**
9	**H**
10	**I**
11	**J**
12	**K**
13	**L**
14	**M**
15	**N**

a) What is the frame size of the physical memory?

b) How many frames of the physical memory?

c) List the logical addresses for the following page.

Page No.	Logical Address
1	
3	

Paging

d) Calculate the page offset for logical memory location that stores the data *H*.

d) Suppose a page table for the above paging system is as follows:

0	1
1	3
2	8
3	10

Calculate the physical address in binary for data *H*. [3 *marks*]

Memory protection

- Memory protection implemented by associating **protection bits** with each frame to indicate if "*read-only*" or "*read-write*" access is allowed.
  - Can also add more bits to indicate "*execute-only*" and so on.
- **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 is a legal page.
  - "*invalid*" indicates that the page is not in the process' logical address space.
  - Or use **Page-Table Length Register (PTLR)**
- Any violations result in a trap to the kernel.

Paging

Example 13 :

(Page: 370)

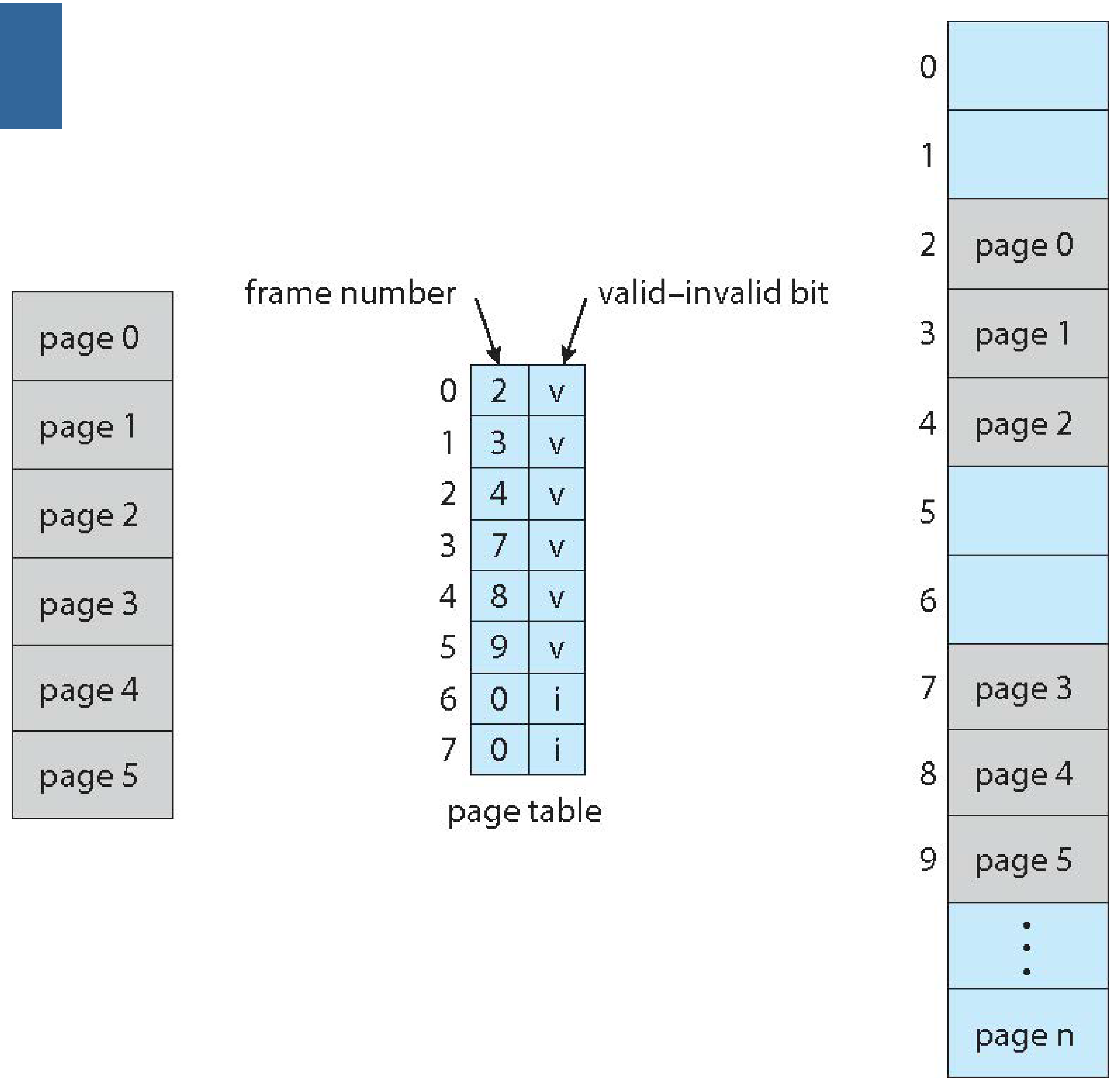


Figure 8.15 Valid (v) or Invalid (i) Bit In A Page Table.

Summary

Summary

- Memory-management algorithms for multiprogrammed operating systems range from the simple single-user system approach to segmentation and paging.

- The most important determinant of the method used is the **hardware** provided.
  - ✓ Every memory address generated by the CPU must be **checked** for legality and possibly mapped to a physical address.
  - ✓ Cannot be implemented in software.

End of Chapter 8