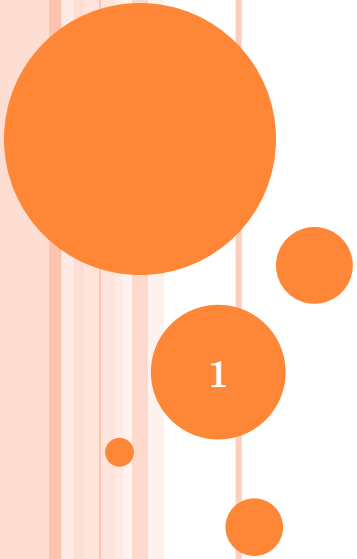


COMP 346 – FALL 2019

Tutorial 6

MEMORY MANAGEMENT

Memory Management

- Main memory (RAM) needs to be divided to accommodate multiple programs in execution.
- The more efficient the allocation of memory, the more programs will be able to run concurrently.

Requirements

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

Memory Management: Requiremtns

Relocation

- What it is:
  - The programmer does not know where the program will be placed in memory when it is executed.
  - While the program is executing, it may be swapped to disk and returned to main memory at a different location.
- What it implies:
  - Memory references must be **translated** in the code to actual physical memory address.

Memory Management: Requiremtns

Protection (closely related with Relocation)

- Why we need it:
  - To protect a process from interference by other processes.
- What it implies:
  - Processes require **permission** to access memory in another processes address space.
  - Because of relocation, it's impossible to check address in programs since the program could be relocated.
  - **Must be checked at run time!**

Memory Management: Requiremtns

Sharing (closely related with Relocation)

- What it is:
  - It allows several processes to access the same data.
  - It allows multiple programs to share the same program text.
- How it's done:
  - Each OS provides its own API.
  - Some libraries provide their own implementation (e.g., Qt, Boost for C++).

Memory Management: Requiremtns

Logical Organization

- What it is:
  - Programs are organized into **modules** (stack, text, uninitialized data, or logical modules such as libraries, objects, etc.).
- What it means:
  - Code modules may be compiled independently.
  - Different degrees of protection can be given to modules (read-only, execute-only).
  - Modules can be shared.

Memory Management: Requiremtns

Physical Organization

- What it is:
  - Memory is organized into two levels: **main** and **secondary memory**.
  - Main memory is relatively fast, expensive, and volatile.
  - Secondary memory is relatively slow, cheaper, has larger capacity, and non-volatile.
- Why we need it:
  - The memory available for a program plus its data may be insufficient.

Memory Partitioning Techniques

- Without using virtual memory:
  - Partitioning – Fixed or Dynamic
  - Simple paging
  - Simple segmentation

- Using virtual memory:
  - Paging
  - Segmentation
  - A combination of both

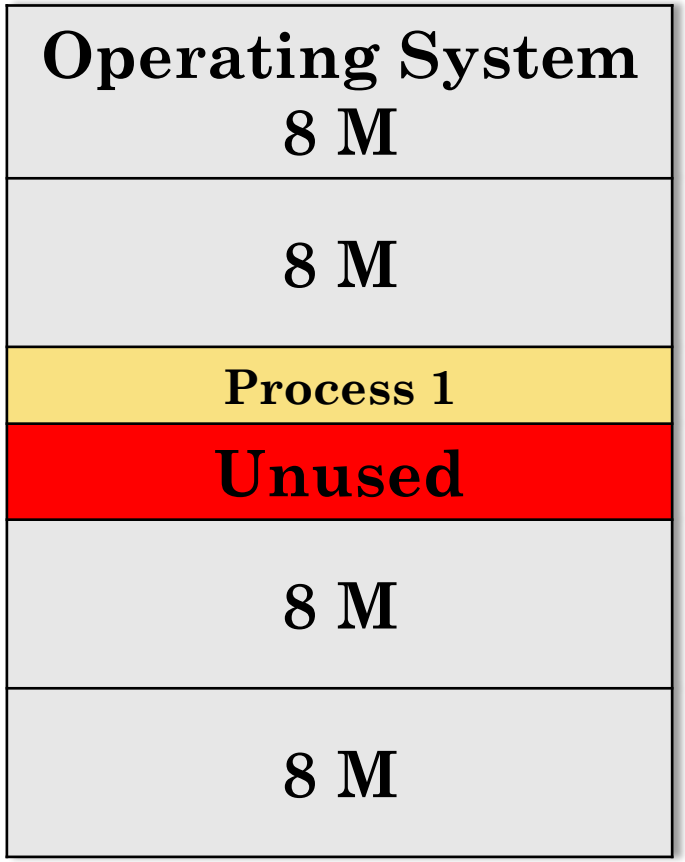
Fixed Partitioning

- Available memory is partitioned into regions with fixed boundaries.

- Using equal-size partitions:
  - **Process size ≤ partition size:** the process can be loaded into a given partition.
  - **Program size > partition size:** the process must be allocated several partitions.
  - If all partitions are full, the operating system can **swap out** a process from a partition.

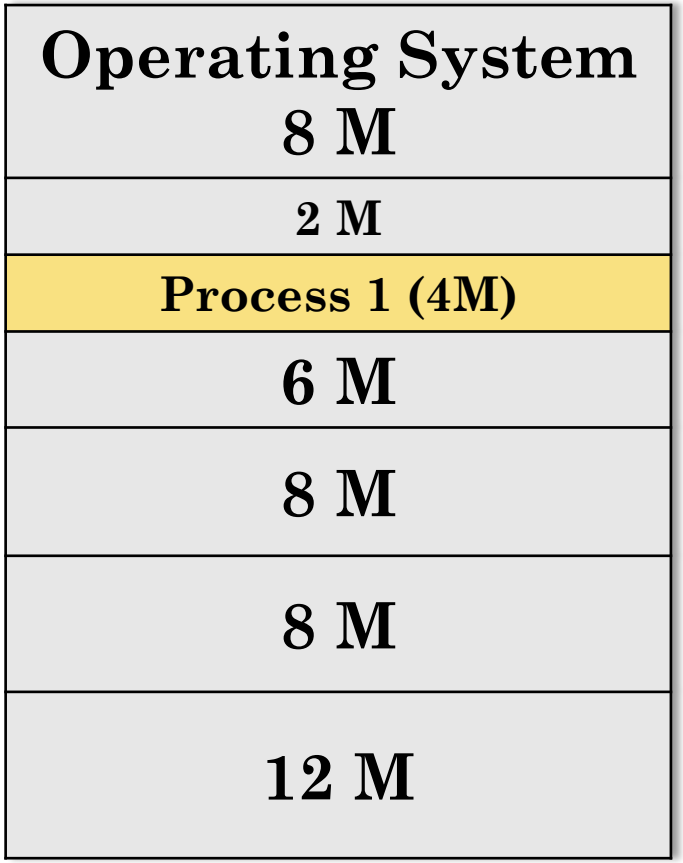
Fixed Partitioning – Equal size

- Main memory use is efficient:
  - **Internal fragmentation** – part of partition unused.



Fixed Partitioning – Unequal Sizes

- Lessens the problem with equal-size partitions.

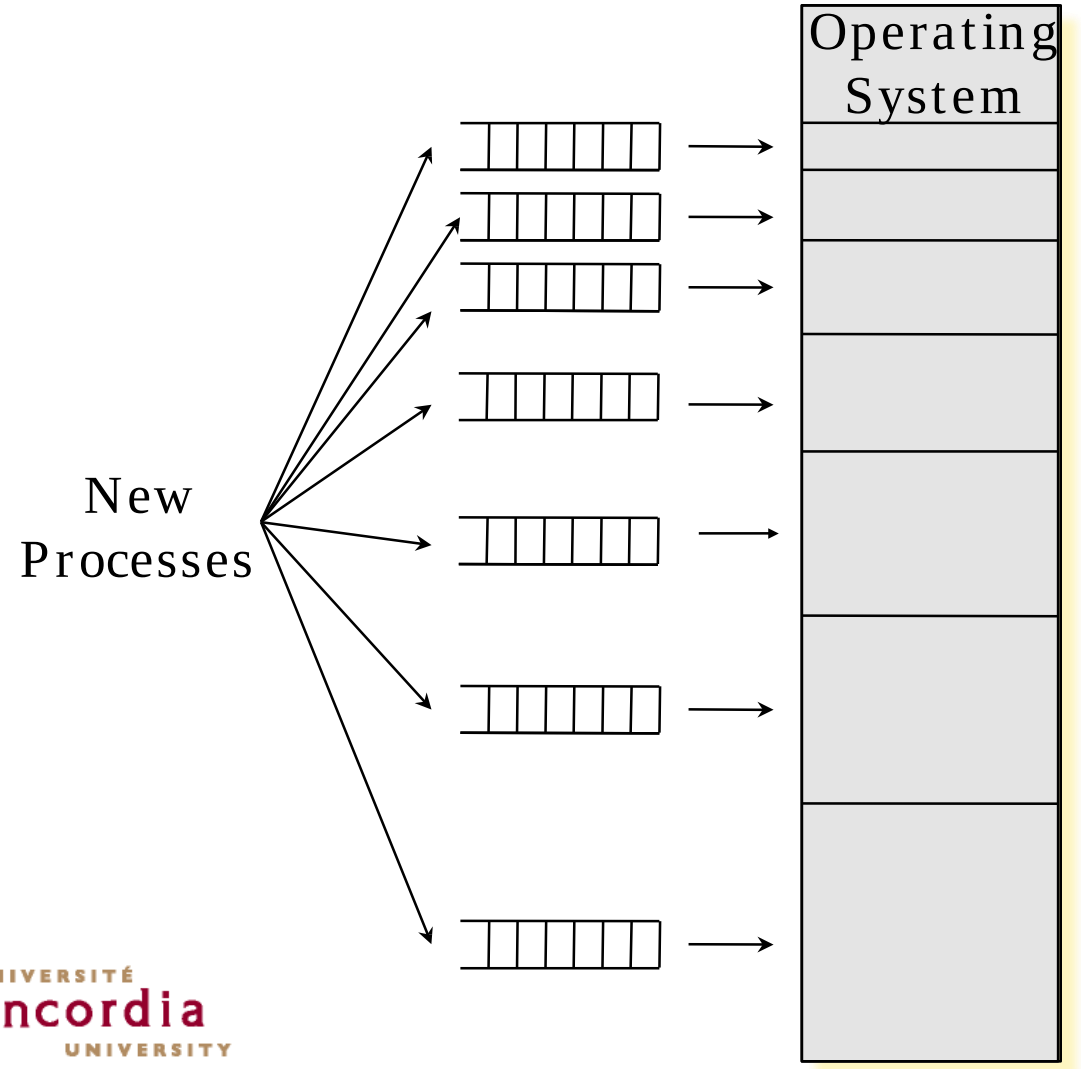


Placement Algorithms with Partitions

- **Equal-size partitions:**
  - Because all partitions are of equal size, it does not matter which partition is used.

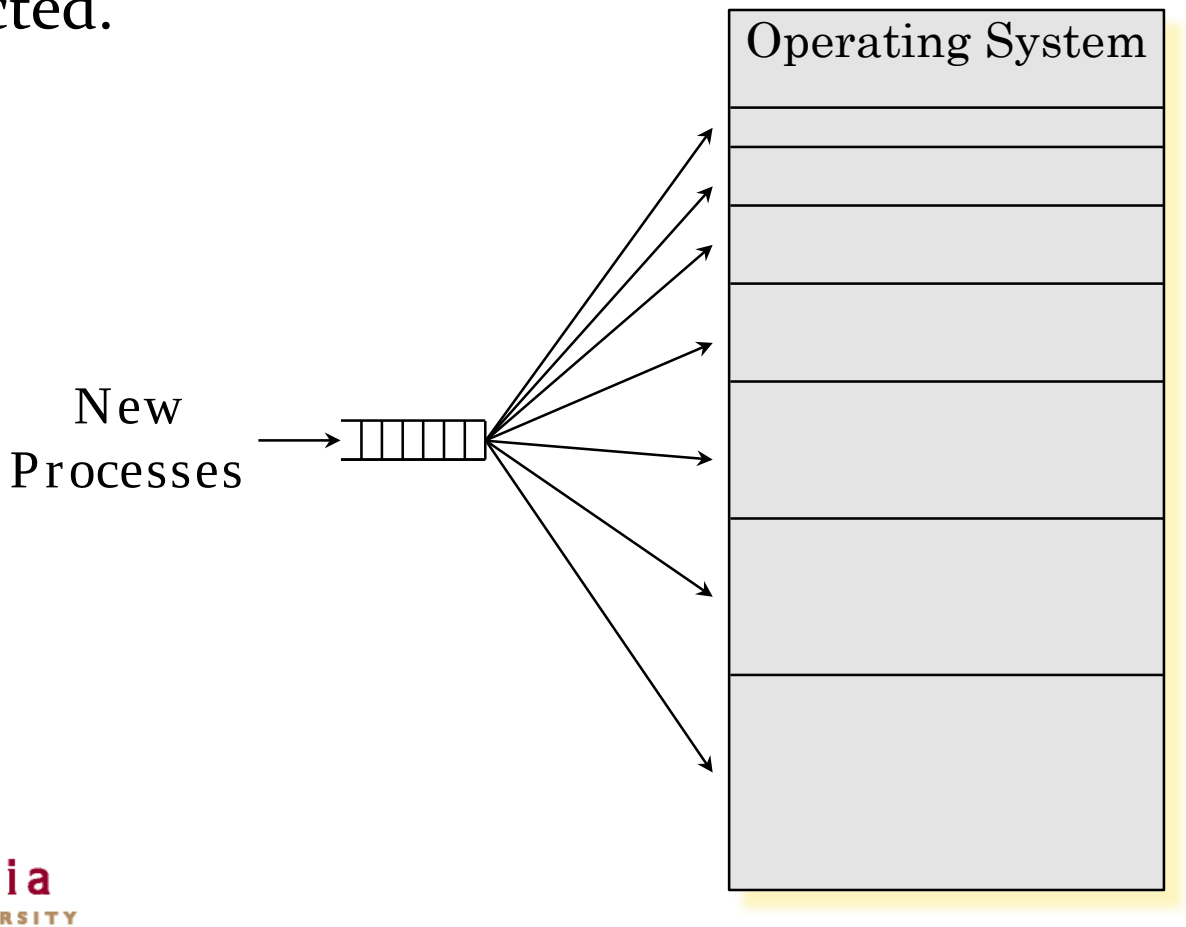
- **Unequal-size partitions:**
  - Processes can be allocated the first available partition ("**First Fit**" algorithm).
  - Processes can be allocated the next available partition ("**Next Fit**" algorithm).
  - Each process can be allocated the smallest partition within which it will fit ("**Best Fit**" algorithm).

One Process Queue per Partition



One Process Queue for All

- Smallest available partition that will hold the process is selected.



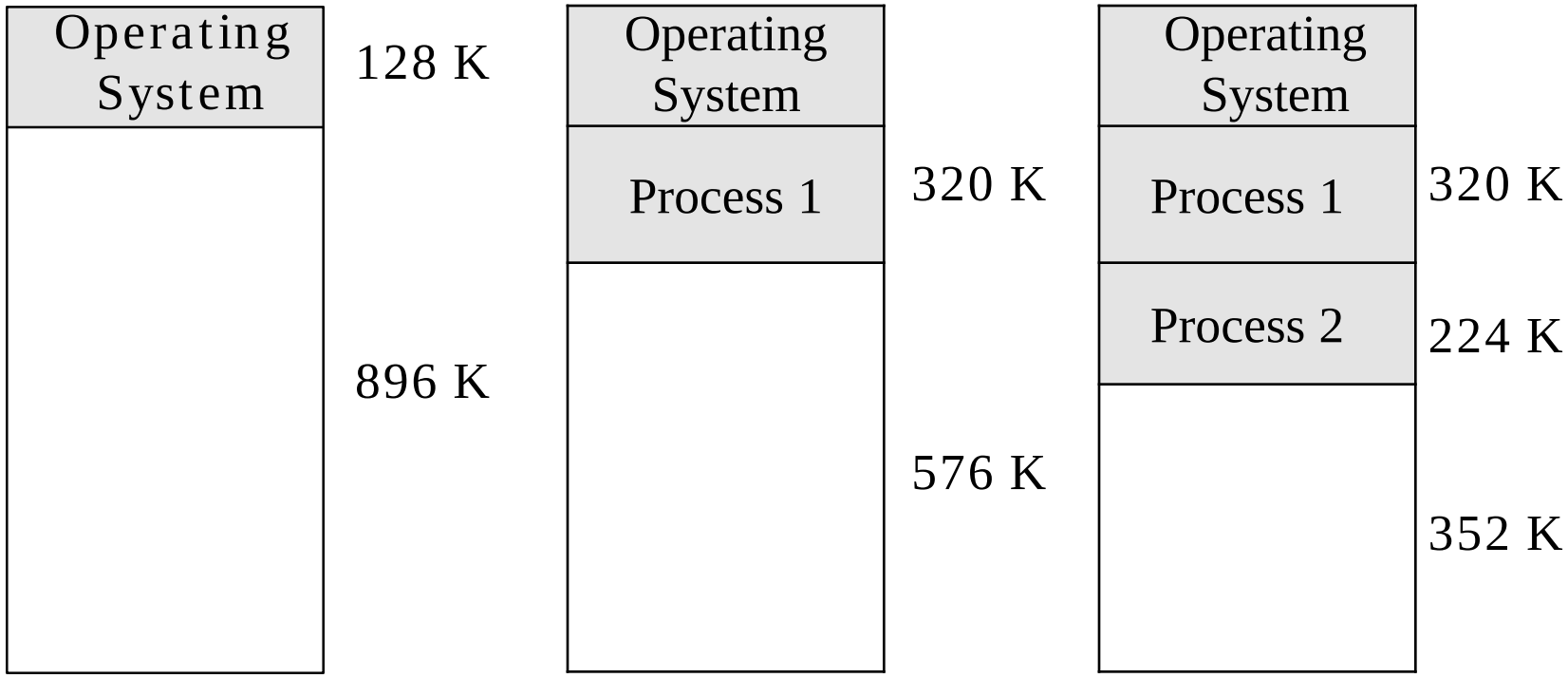
Dynamic Partitioning

- Partitions are of variable length and number.
- A process is allocated exactly the required amount of memory.

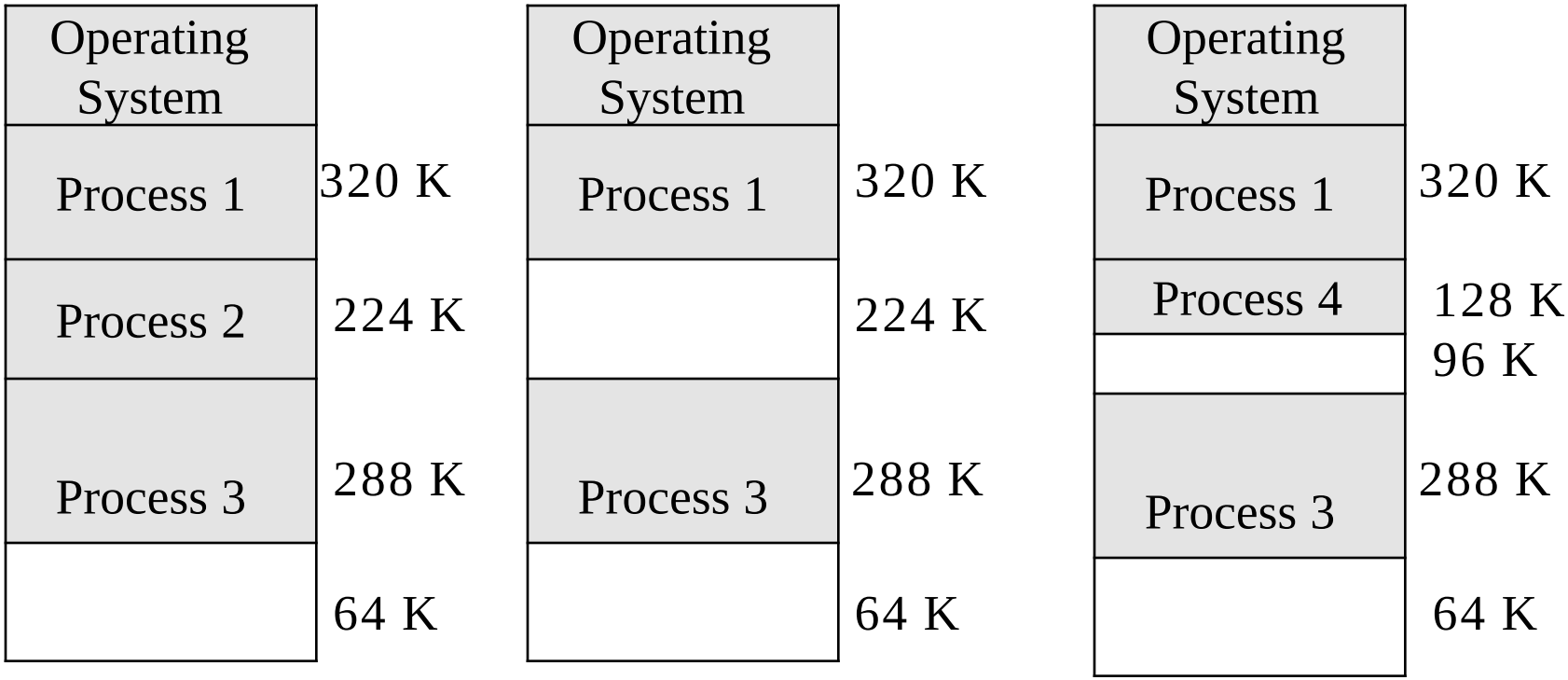
- **Compaction** – shifting processes so they are contiguous and all free memory is in one block.

- **External fragmentation** – small holes in memory between allocated partitions.

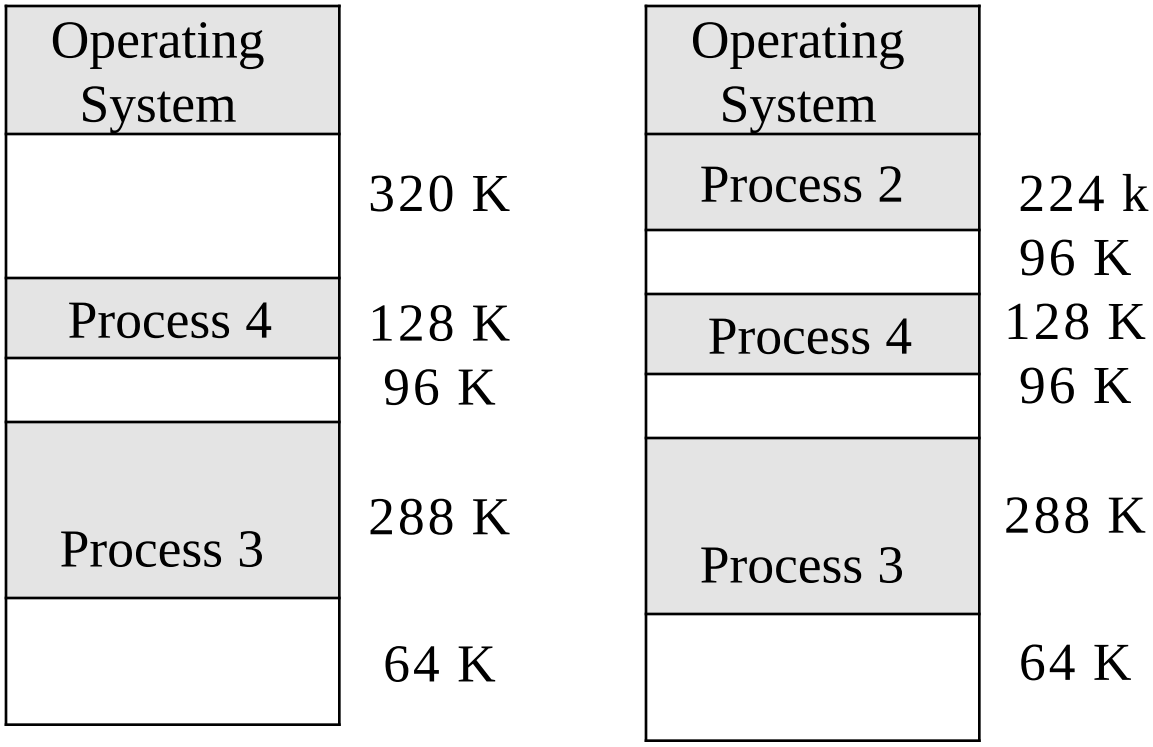
Example of Dynamic Partitioning



Example of Dynamic Partitioning



Example of Dynamic Partitioning



Dynamic Partitioning Placement Algorithm

- **"First-fit" algorithm**
  - Chooses the first block from the beginning that will fit the requirements of the process.

- **The fastest:**
  - May have to scan through a large number of existing processes.

Dynamic Partitioning Placement Algorithm

- **"Next-fit" algorithm**
  - Chooses the next block from the last allocation that will fit the requirements of the process.

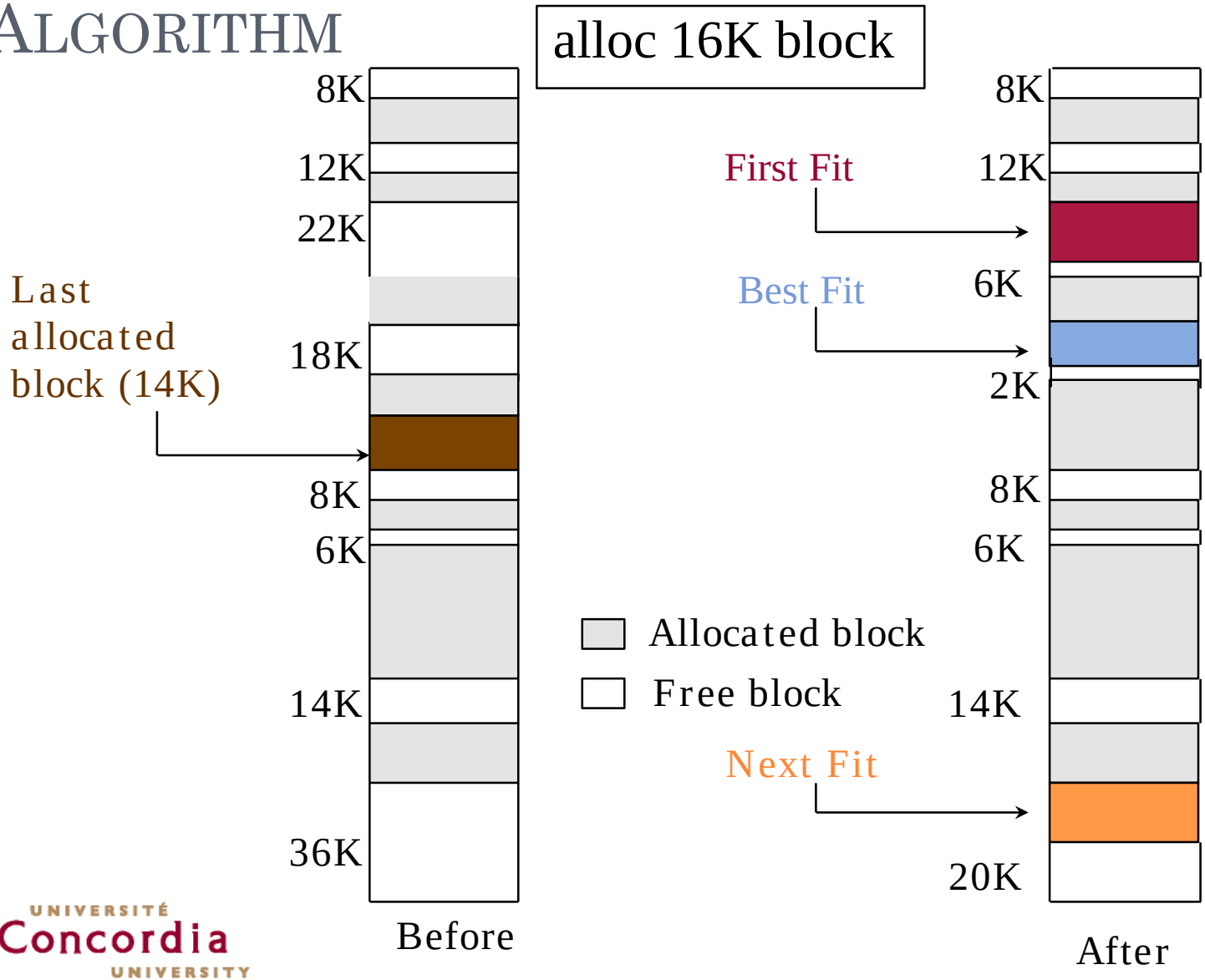
- **Average overall performance:**
  - Often involves allocating the last available block in memory.
  - Requires a significant number of compaction operations.

Dynamic Partitioning Placement Algorithm

- **"Best-fit" algorithm**
  - Chooses block that is closest in size to the request

- **Worst performance overall:**
  - Results in minimally sized fragments, requiring compaction.

Dynamic Partitioning Placement Algorithm



Q1

- Consider a memory manager that maintains a free list of blocks of sizes 600, 400, 1000, 250, 1600, and 1050 bytes. Assume that the block of 600 bytes is at the head of the list. Also, suppose that there are 5 processes (say P1, P2, P3, P4, P5) with respective sizes of 1352, 167, 753, 572, and 1050 bytes. Memory is allocated to the processes in order P1, P2, P3, P4, P5, and all the processes are loaded in memory prior to their execution.
- (i) Using the **best fit** placement algorithm, show which memory block is assigned to a given process.
- (ii) Using the **next fit** placement algorithm, show which memory block is assigned to a given process. Assume that the pointer currently points at the head of the list.
- (iii) Calculate the **total internal fragmentation** for the placement algorithms used in (i).
- (iv) Calculate the **total internal fragmentation** for the placement algorithms used in (ii).

S1

- (i) Using the best fit placement algorithm, show which memory block is assigned to a given process.

P1 ← 1600, P2 ← 250, P3 ← 1000, P4 ← 600, P5 ← 1050

- (ii) Using the next fit placement algorithm, show which memory block is assigned to a given process. Assume that the pointer currently points at the head of the list.

P1 ← 1600, P2 ← 1050, P3 ← 1000, P4 ← 600, P5 ← none (external fragmentation)

- (iii) Calculate the total internal fragmentation for the placement algorithms used in (i).

```
fi = (1600-1352) + (250-67) + (1000-753) + (600-572) +
     (1050-1050) = 248 + 83 + 247 + 28 + 0 = 606 bytes
```

- (iv) Calculate the total internal fragmentation for the placement algorithms used in (ii).

```
fi = (1600-1352) + (1050-167) + (1000-753) + (600-572)
   = 248 + 883 + 247 + 28 = 1403 bytes
```

Relocation

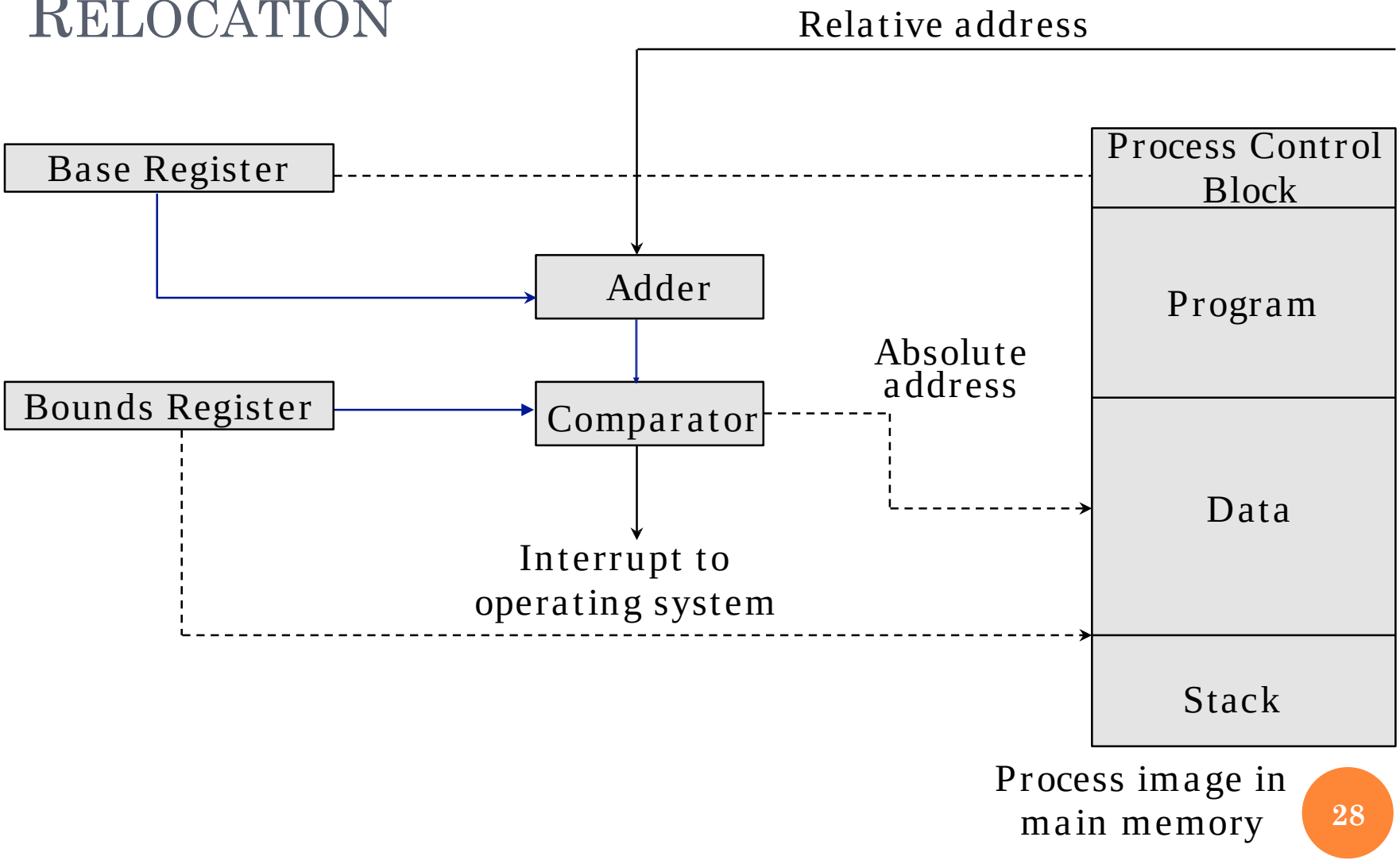
- As a process is created, **absolute memory locations** are assigned to it.
- These locations are different every time the program is executed.
- Compaction also causes changes to these locations.

Addresses

- **Logical**
  - A reference to a memory location independent of the current assignment of data to memory.
  - Requires a **mapping** with the physical address.

- **Physical**
  - The **absolute** address or actual location in memory.

Hardware Support For Relocation



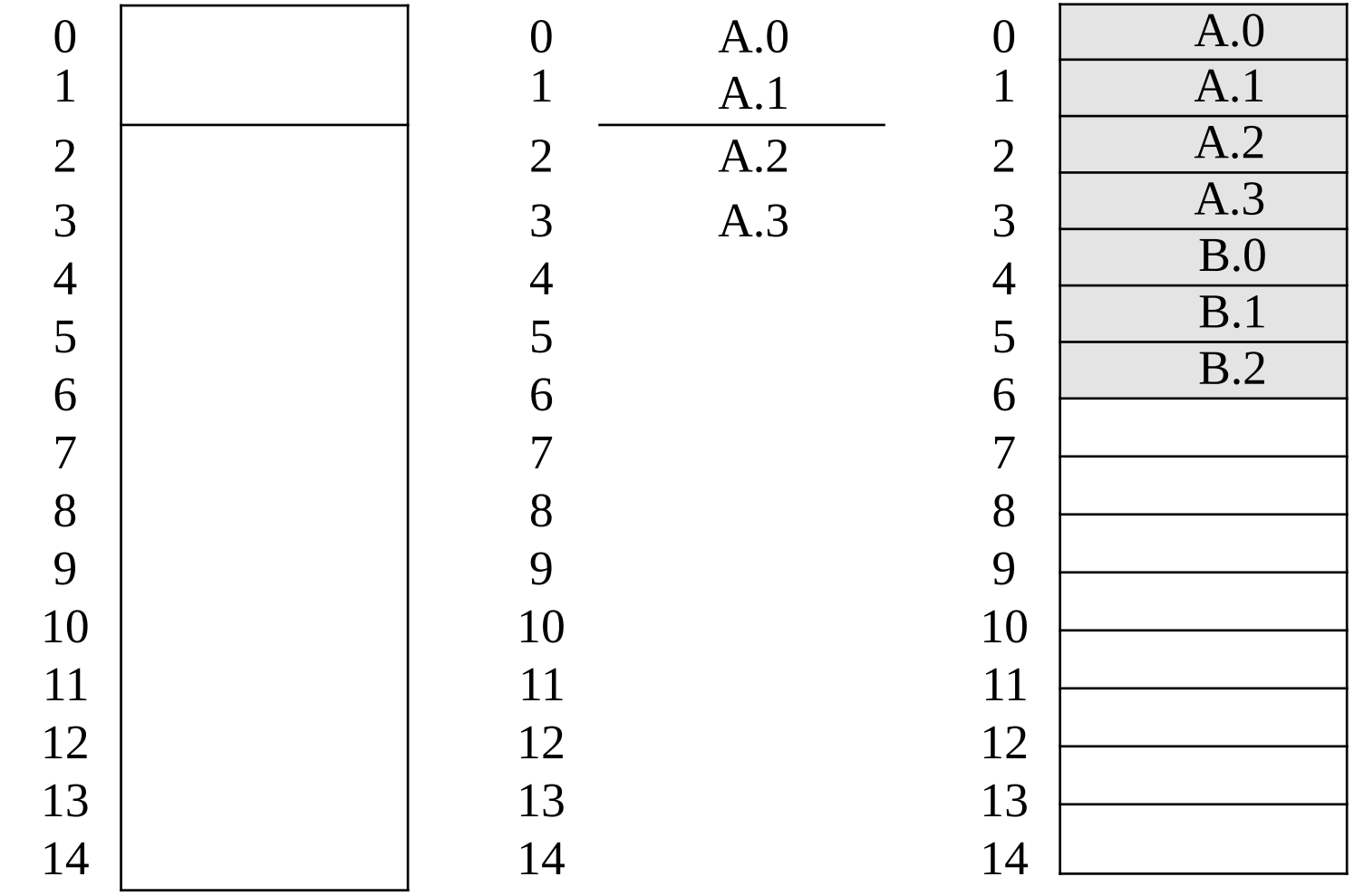
Paging

- Partitions memory into equal-sized chunks.
  - These are called **frames**.
- Divide each process into same-sized chunks.
  - These are called **pages**.

- The operating system maintains a page table for each process.
  - Contains a **mapping** between a page and a frame.

```
absolute = page_number * chunk_size + offset
```

Paging



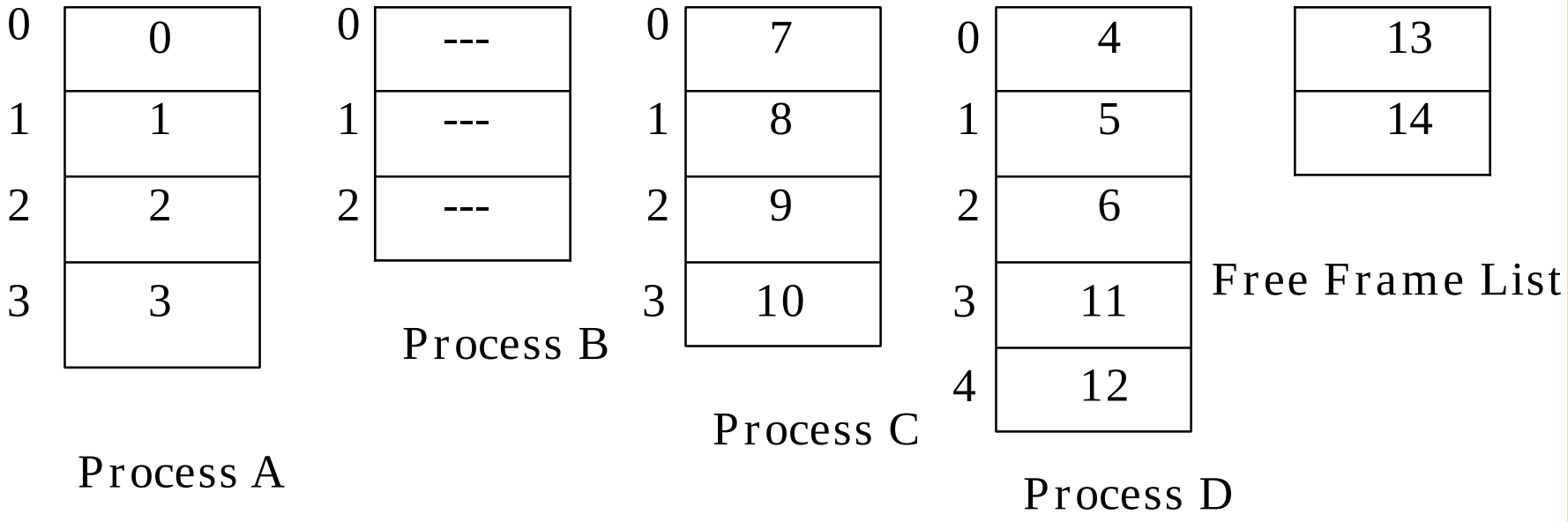
Paging

0	A.0
1	A.1
2	A.2
3	A.3
4	B.0
5	B.1
6	B.2
7	C.0
8	C.1
9	C.2
10	C.3
11	
12	
13	
14	

0	A.0
1	A.1
2	A.2
3	A.3
4	
5	
6	
7	C.0
8	C.1
9	C.2
10	C.3
11	
12	
13	
14	

0	A.0
1	A.1
2	A.2
3	A.3
4	D.0
5	D.1
6	D.2
7	C.0
8	C.1
9	C.2
10	C.3
11	D.3
12	D.4
13	
14	

Page Tables for Example



Segmentation

- Segments do not have to be the same length.
- However, there is a maximum segment length.

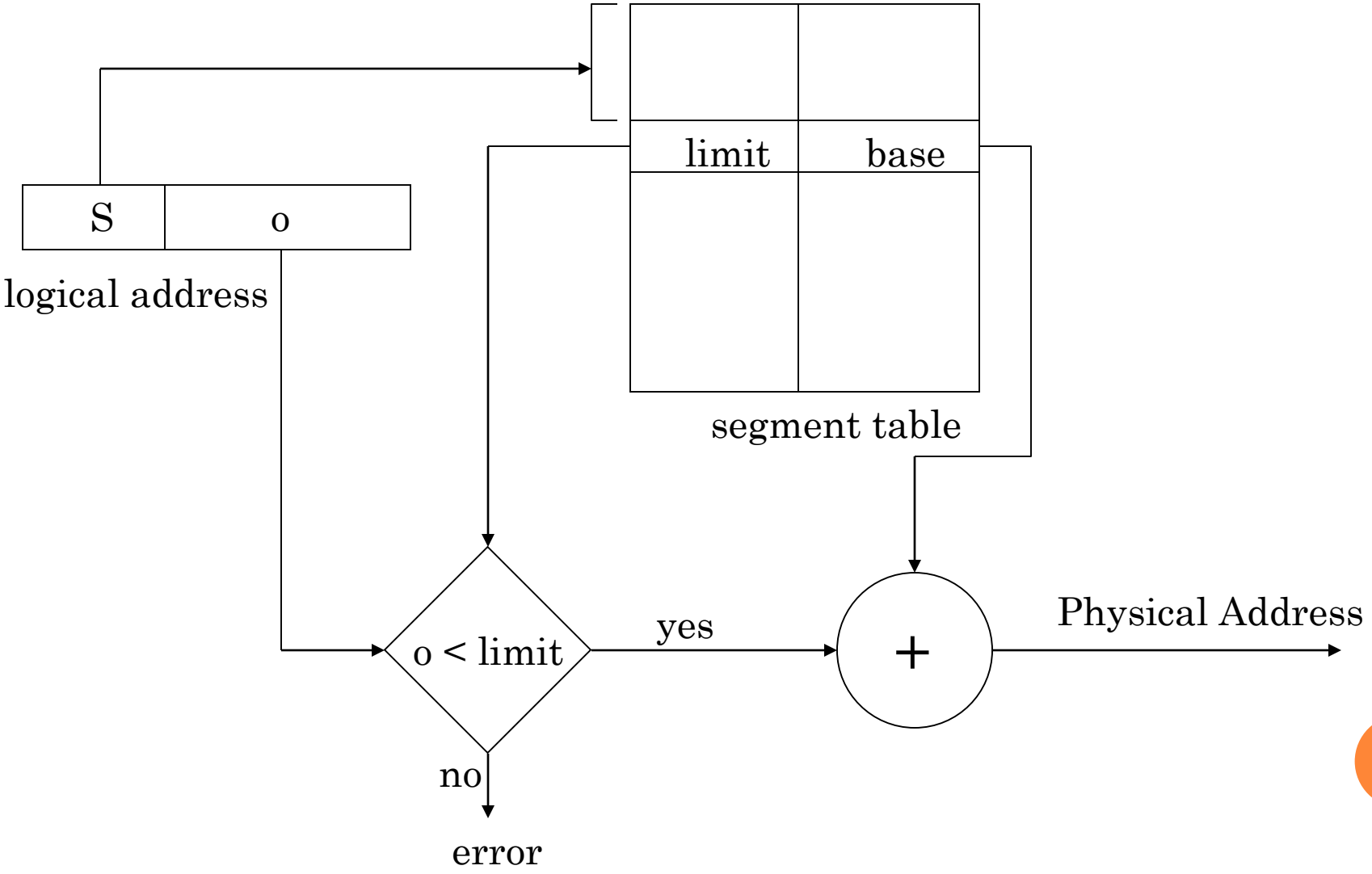
- Addressing consists of two parts - a **segment number** and an **offset**.
- Since segments are not equal, segmentation is similar to dynamic partitioning.

Addressing Segments

- The segment number is used as an index in a mapping table (similar to paging).
- This table contains the physical address of the start of the segment (called the **base address**).
- The **offset** is added to the base address to generate the physical address.
  - The offset is checked against a limit (the size of the segment).

```
absolute = base_address + offset
```

Addressing Segments



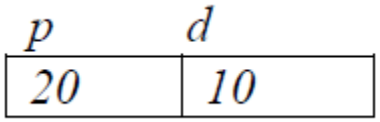
Q.2

- Consider a system with a logical address space of 1 GB, a physical memory of 64 MB, a block size of 1 KB, and a process address space of 143 KB.

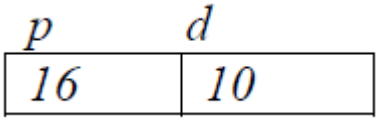
- (i) Calculate the format of a **logical address** .
- (ii) Calculate the format of a **physical address** .
- (iii) For the logical address 20514 calculate **the physical address** . Assume that the page that contains the logical address 20514 is loaded in frame number 4100.

S.2

- (i) Calculate the format of a logical address.



- (ii) Calculate the format of a physical address.



- (iii) For the logical address 20514 calculate the physical address. Assume that the page that contains the logical address 20514 is loaded in frame number 4100.

a) ***Page number of address 20514 is:*** `floor(20514/1024) = 20`

b) ***Displacement within page 20 is:*** `20514 % 1024 = 34`

c) ***Base address of frame number 4100 in physical memory is:*** `4100 * 1024 = 4198400`

d) ***Physical address:*** `4198400 + 34 = 4198434`