Chapter – Three

Memory Management

Memory Management

- The CPU fetches **instructions** and **data** of a program from memory;
- Therefore, both the program and its data must reside in the main memory (RAM and ROM).
- Modern multiprogramming systems are capable of storing more than one program, together with the data they access, in the main memory.
- A fundamental task of the **memory management** component of an operating system is to ensure safe execution of programs by providing:
  - Sharing of memory
  - Memory protection

Memory Management…

Issues in sharing memory

Transparency

- Several processes may co-exist, unaware of each other, in the main memory and run regardless of the number and location of processes.

Safety (or protection)

- Processes must not corrupt each other (nor the OS!)

Efficiency

- CPU utilization must be preserved and memory must be fairly allocated.

Relocation

- Ability of a program to run in different memory locations.

Storage allocation

- Information stored in main memory can be classified in a variety of ways:
  - Program (code) and data (variables, constants)
  - Read-only (code, constants) and read-write (variables)
  - Address (e.g., pointers) or data (other variables);

- The OS, compiler, linker, loader and run-time libraries all cooperate to manage this information.

Memory Management: Definitions

- **Relocatable -** Means that the program image can reside anywhere in physical memory.
- **Binding -** Programs need real memory in which to reside. When is the location of that real memory determined?
  - This is called **mapping** logical to physical addresses.
  - This binding can be done at **compile** or **run time**.
- **Compiler** - If it's known where the program will reside, then absolute code is generated. Or fixed memory address for instructions and data.
- **Load** –load programs into memory for execution .
- **Execution** – running the programs instructions on the CPU.

Creating an executable code

- Before a program can be executed by the CPU, it must go through several steps:
  - Compiling (translating) - generates re-locatable object code.
  - Linking - combines the object code into a single self-sufficient *executable code*.
  - Loading - copies the executable code into memory.
  - Execution - dynamic memory allocation.

Functions of a linker

- **Linker** collects and puts together all the required pieces to form the executable code.

- Issues:
  - Relocation
    - ✓ Where to *put pieces.*
  - Cross-reference
    - ✓ where to *find pieces.*
  - Re-organization
    - ✓ *new memory layout.*

Functions of a loader

- A ***loader*** places the executable code in main memory starting at a pre-determined location (base or start address).

- This can be done in several ways, depending on hardware architecture:
  - ***Absolute loading****:* always loads programs into a **fixed** memory location.
  - ***Relocatable loading****:* allows loading programs in different memory locations.
  - ***Dynamic (run-time) loading****:* loads functions when first called (if ever).

Binding of Instructions and Data to Memory

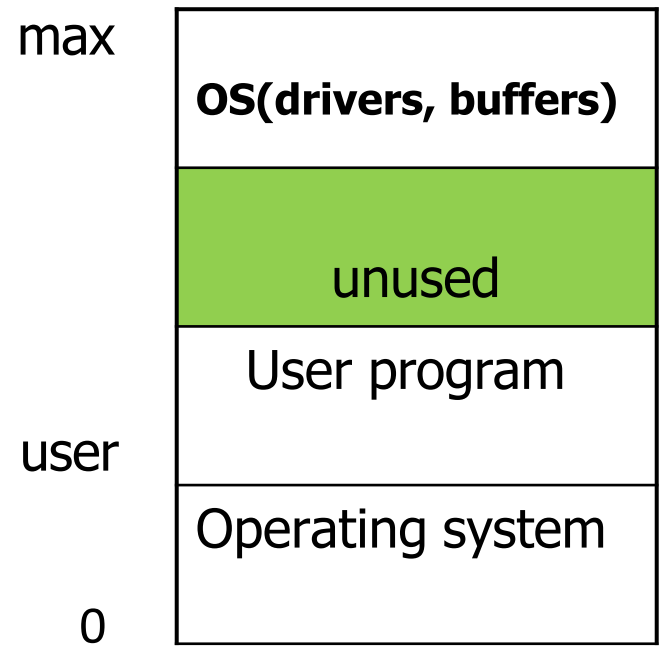
- The process of associating program instructions and data with specific memory address
- Address binding of instructions and data to memory addresses can happen at three different stages.
  - **Compile time**: If memory location known a priori, absolute code can be generated, other wise relocatable
  - **Load time**: Compiler must generate *relocatable* code if memory location is not known at compile time.
  - **Execution time**: Binding delayed until run time if the process can be moved during its execution from one memory segment to another.
  - Need hardware support for address maps (e.g., *base* and *limit registers*).

Simple management schemes

- An important task of a **memory management** system is to bring (load) programs into main memory for execution.
- The following ***contiguous memory allocation*** *techniques were commonly* employed by earlier operating systems :
    - Direct placement
    - Overlays
    - Partitioning

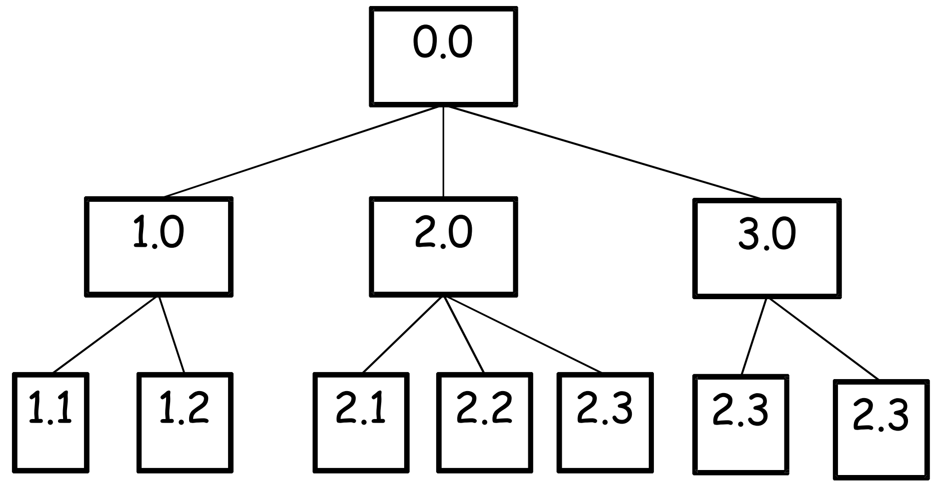
Direct placement

- because the user programs are always loaded (one at a time) into the same memory location (***absolute loading***).
- Or this approach assigning fixed addresses for programs.
- Examples: Early batch monitors, MS-DOS

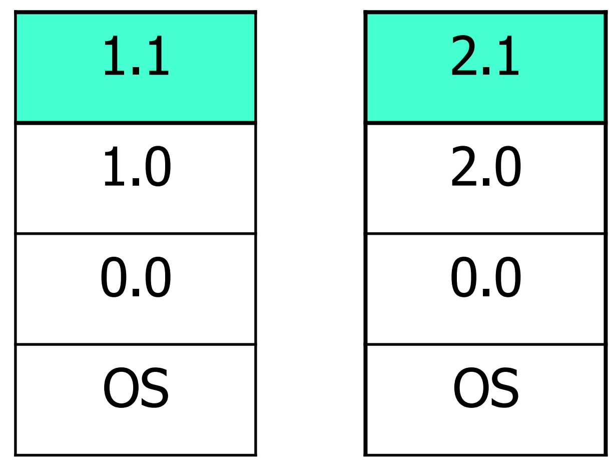


Overlays

- It allows a program larger than available memory by assigning for necessary parts.
- A program was organized (by the user) into a tree-like structure of object modules, called ***overlays***.
- The ***root overlay*** was always loaded into the memory,
- whereas the sub trees were (re-loaded as needed by simply overlaying existing code.)
- Dynamic loading



Overlay tree



Memory snapshot

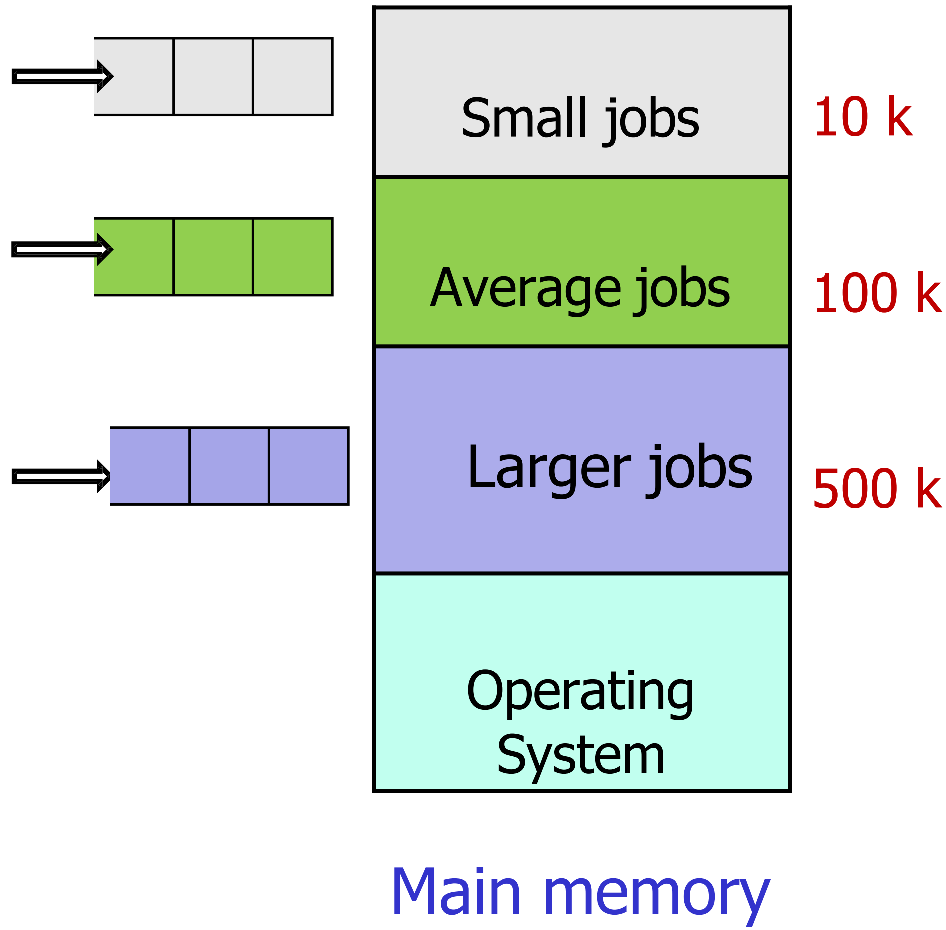
Partitioning

- A simple method to accommodate several programs in memory at the same time (to support multiprogramming) is **partitioning**.
- In this scheme, the memory is divided into a number of contiguous regions, called ***partitions.***
- Two forms of memory partitioning, depending on when and how partitions are created (and modified), are possible:
  - Static partitioning
  - Dynamic partitioning

Partitioning…

Static partitioning

- Main memory is divided into ***fixed*** *number of (fixed size)* partitions during system generation or start-up.
- Programs are queued to run in the smallest available partition.
- An executable prepared to run in one partition may not be able to run in another, without being re-linked.
- This technique uses ***absolute loading.***
  - It cause for internal fragmentation



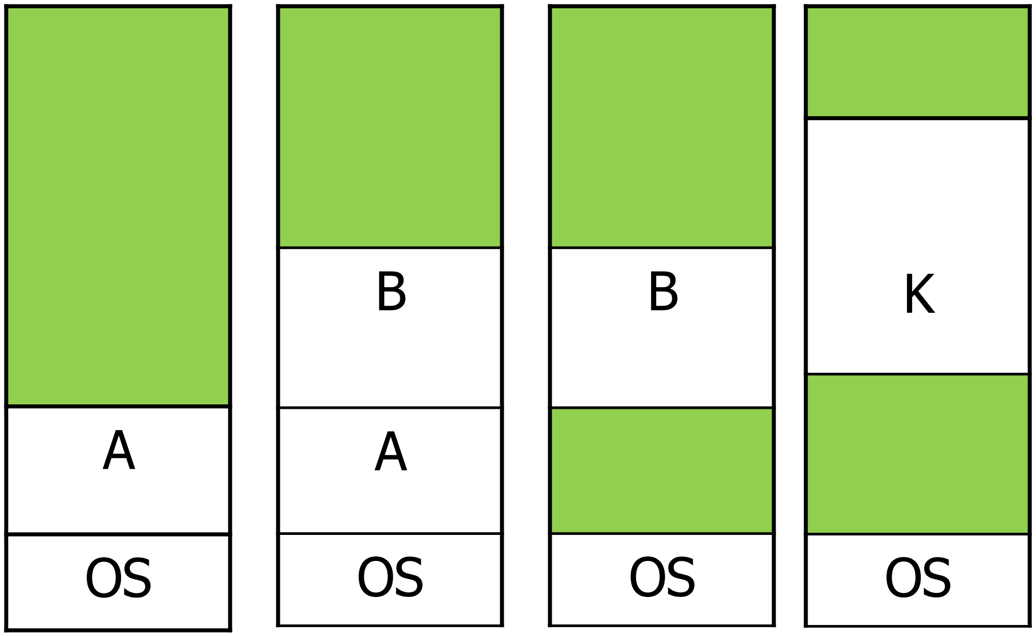
Partitioning…

Dynamic partitioning

- Any number of programs can be loaded to memory as long as there is room for each.
- When a program is loaded (***relocatable loading***), it is allocated memory in exact amount as it needs.

❒ The operating system keeps track of each partition (their size and locations in the memory.)

Main memory



Partition allocation at different times

➢ It cause for external fragmentation

Swapping…

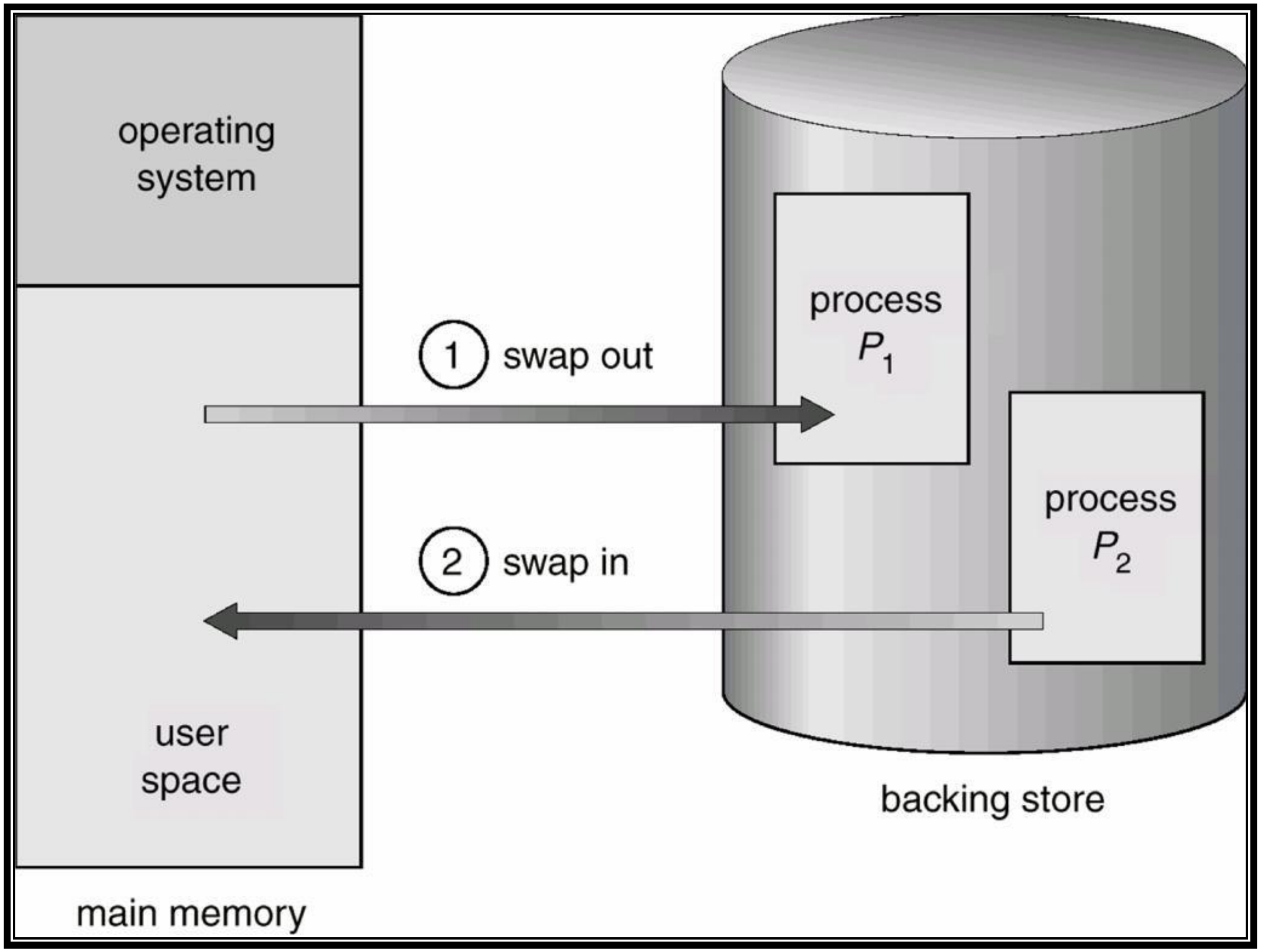
- A process can be *swapped* temporarily out of memory to a *backing store*, and then brought back into memory for continued execution.

- 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.

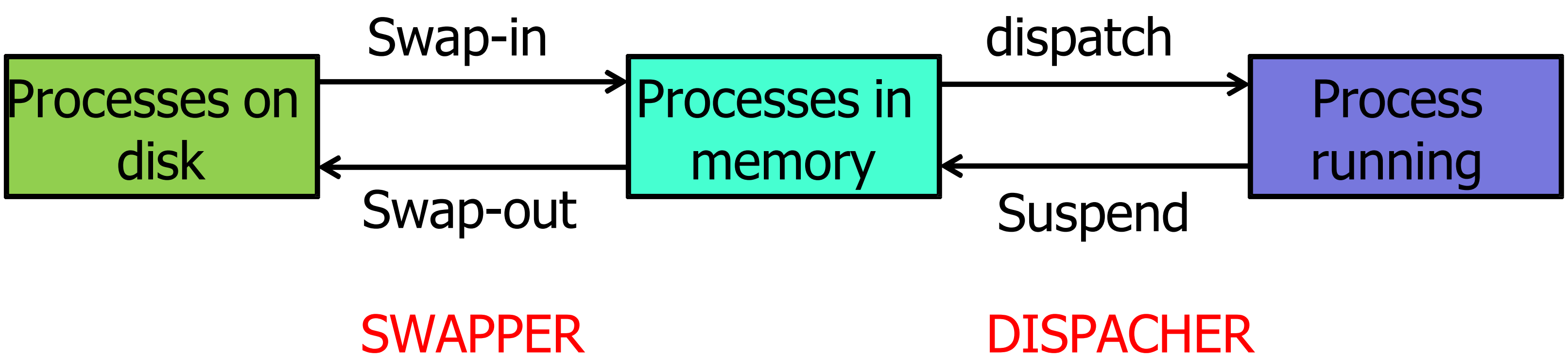
- UNIX, Linux, and Windows.

Schematic View of Swapping



Swapping…

- Swapping is a medium term scheduling discipline



- Swapping brings flexibility even to systems with fixed partitions, because:
  - *“ if needed, the operating system can always make room for high-priority jobs, no matter what!’“*

Swapping…

- **Swapping** is a medium-term scheduling method.
- The responsibilities of a swapper include:
  - Selection of processes to swap out
    - *criteria: suspended/blocked state, low priority, time spent in memory*
  - Selection of processes to swap in
    - *criteria: time spent on swapping device, priority*

Logical vs. Physical Address Space

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

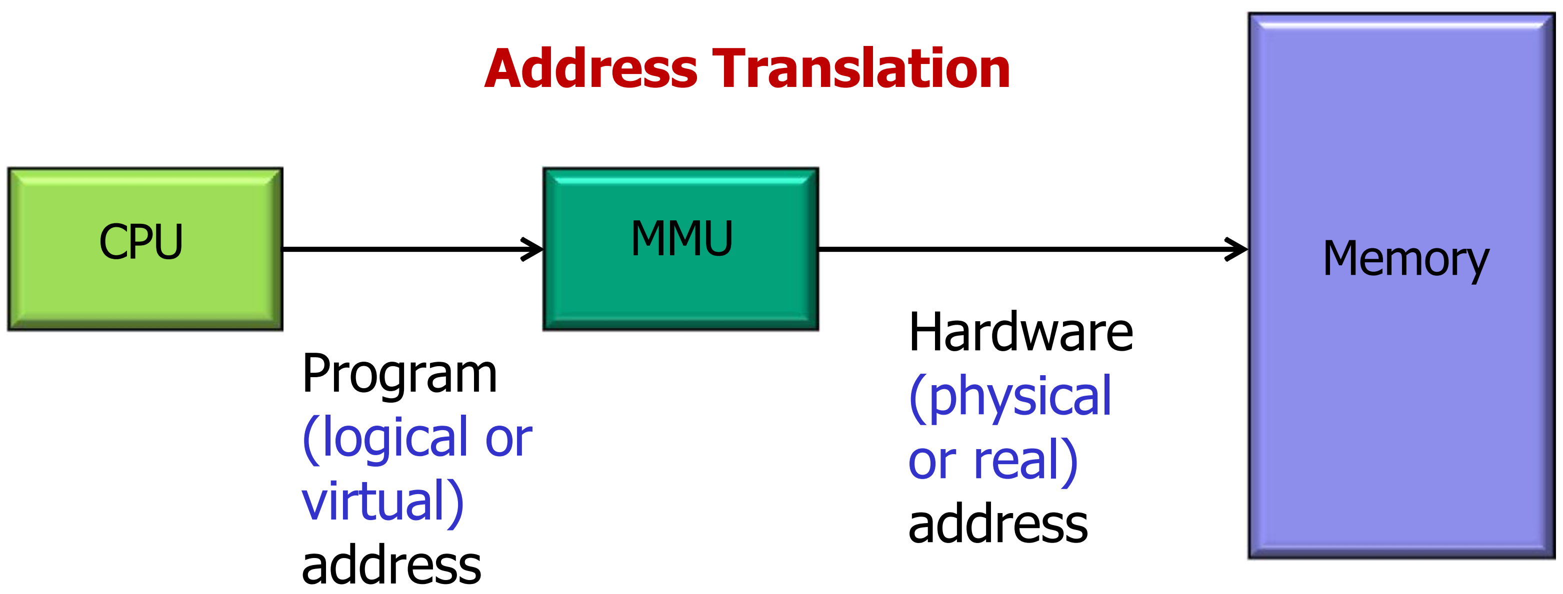
Memory Protection

- The other fundamental task of a memory management system is to *protect* programs *sharing* the memory from each other.
- This protection also covers the operating system itself.

- Memory protection can be provided at either of the two levels:
- Hardware:
  - ✓ *address translation*
- Software:
  - ✓ language dependent: *strong typing*
  - ✓ language independent: *software fault isolation*

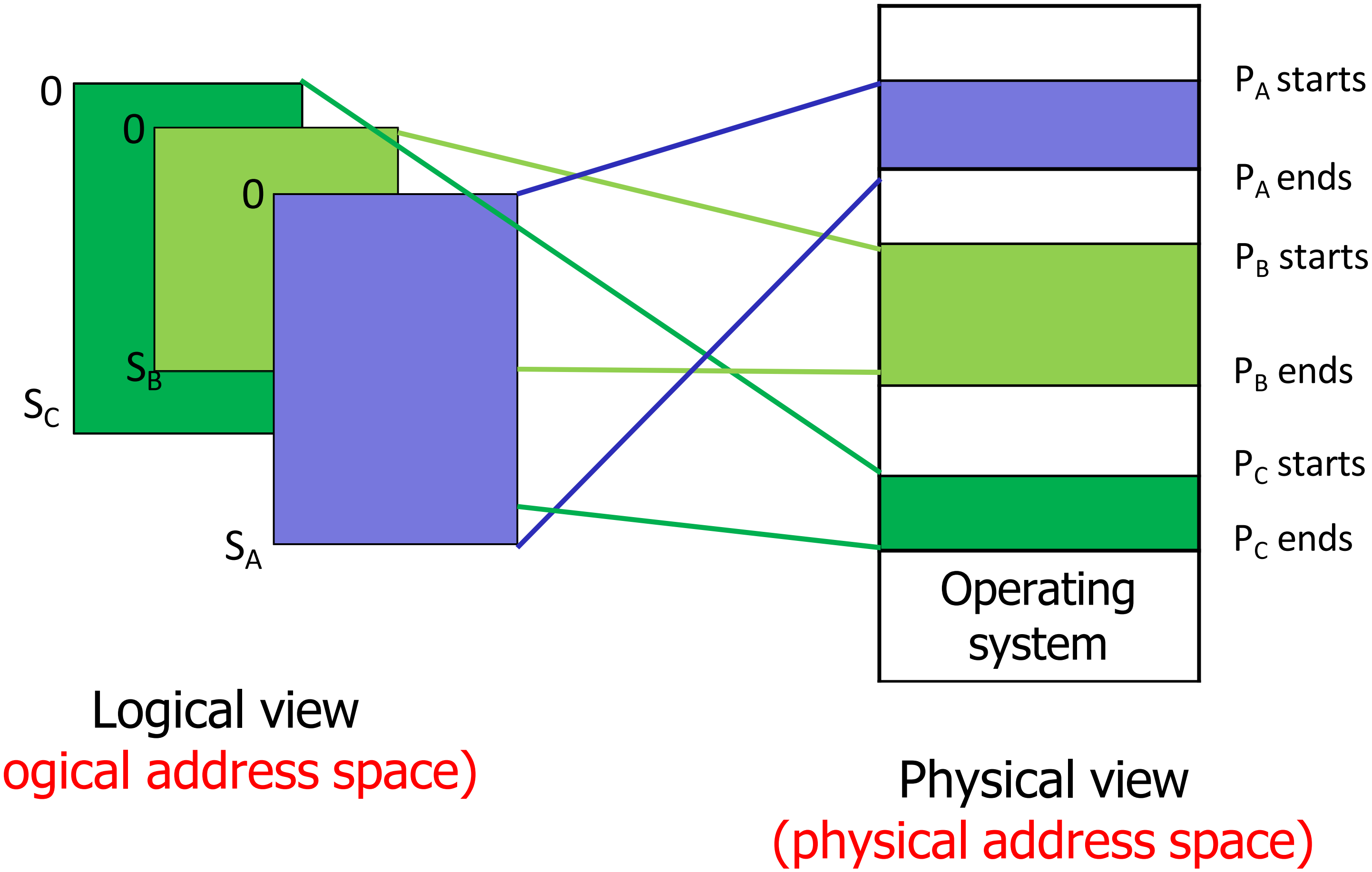
Dynamic relocation

- With *dynamic relocation*, each program-generated address (*logical address*) is translated to hardware address (*physical address*) at runtime for *every* reference, by a hardware device known as the **memory management unit (MMU)**.



Two views of memory

- Dynamic relocation leads to two different views of main memory, called *address spaces*.



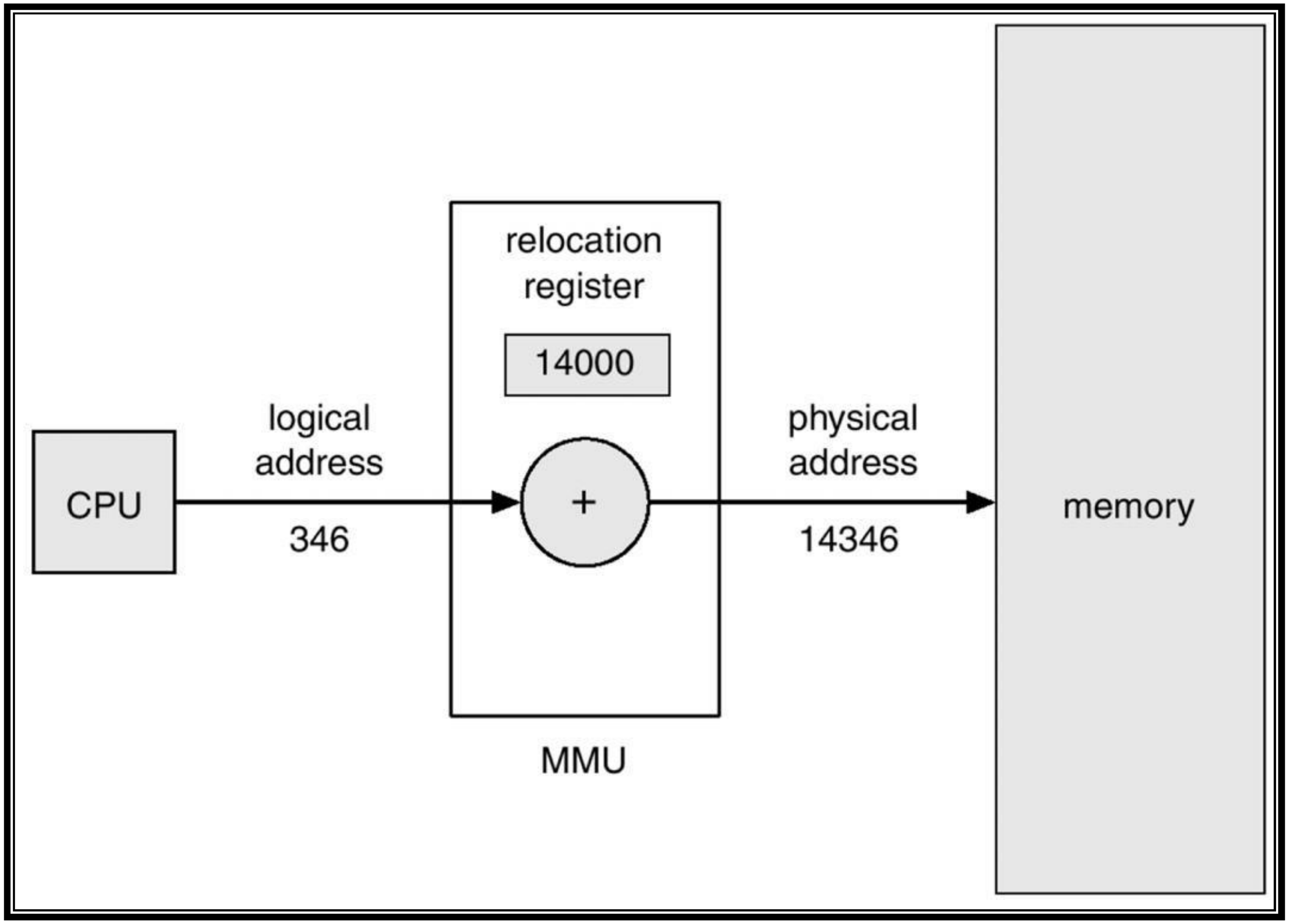
Memory-Management Unit (MMU)

- 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.

Dynamic relocation using a relocation/base register



Fragmentation

- ***Fragmentation*** refers to the unused memory that the management system cannot allocate.
- ***Internal fragmentation***
  - Waste of memory within a partition, caused by the difference between larger memory blocks and small requested memory.
  - Severe in static (fixed) partitioning schemes.
- ***External fragmentation***
  - Waste of memory between partitions, caused by scattered non-contiguous free space.
  - Severe in dynamic (variable size) partitioning schemes.
  - *Compaction is a technique that is used to overcome external fragmentation.*

Segmentation

- **Segmentation** is a *memory management scheme* that supports user's view of memory.
- and a technique in which memory divide into variable sized chunks which can be allocated to processes.
- Each chunk is called segment
- A segment is a region of contiguous memory.
- **Segmentation** generalizes the base-and-bounds technique by allowing each process to be split over several segments.
  - A **segment table** holds the base and bounds of each segment.
  - Although the segments may be scattered in memory, each segment is mapped to a contiguous region.
  - Additional fields (**Read/Write and Shared**) in the segment table adds protection and sharing capabilities to segments.

Segmentation…

- 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

Segmentation Architecture

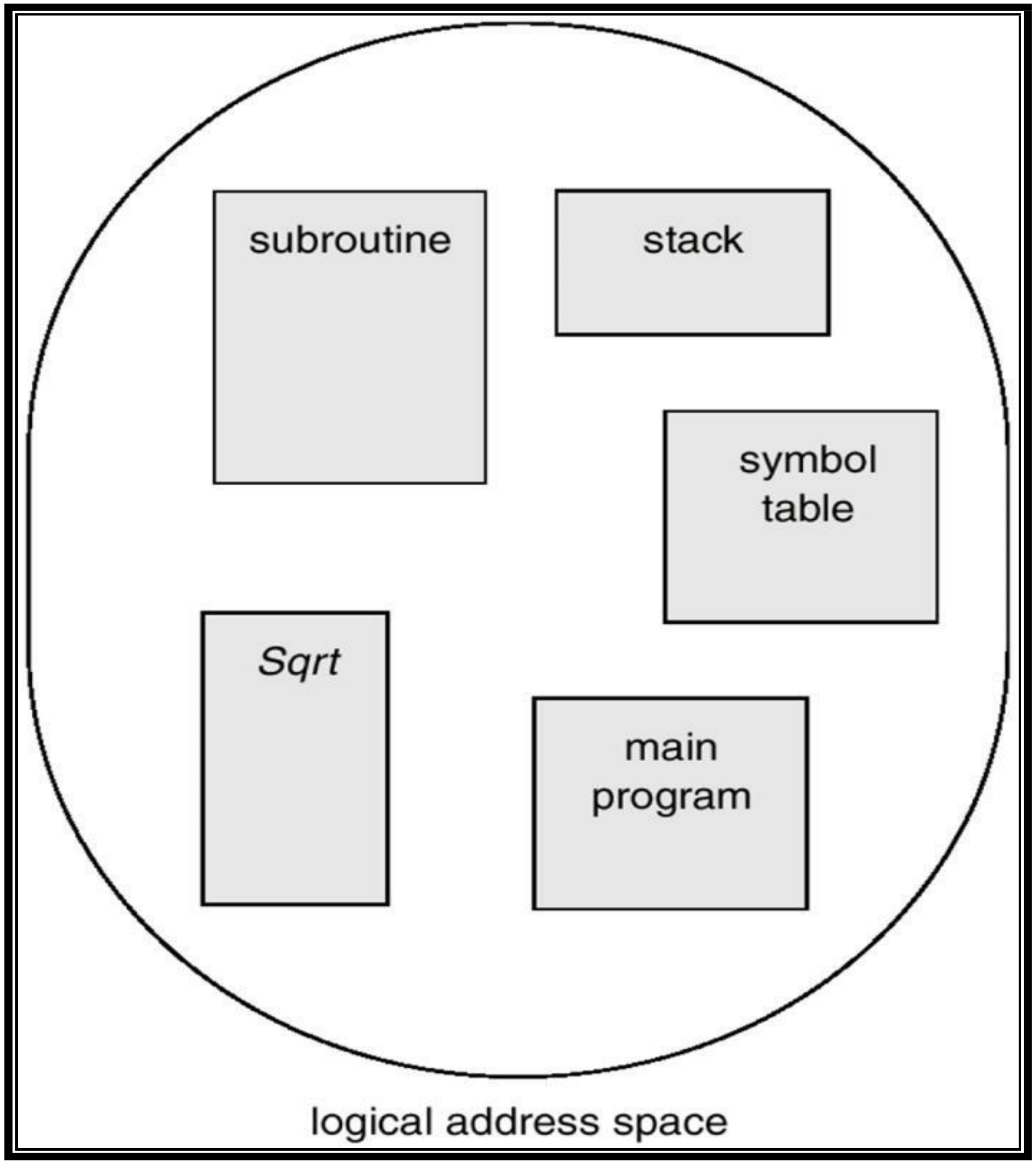
- Logical address consists of a two tuple:

  <segment-number, offset>,

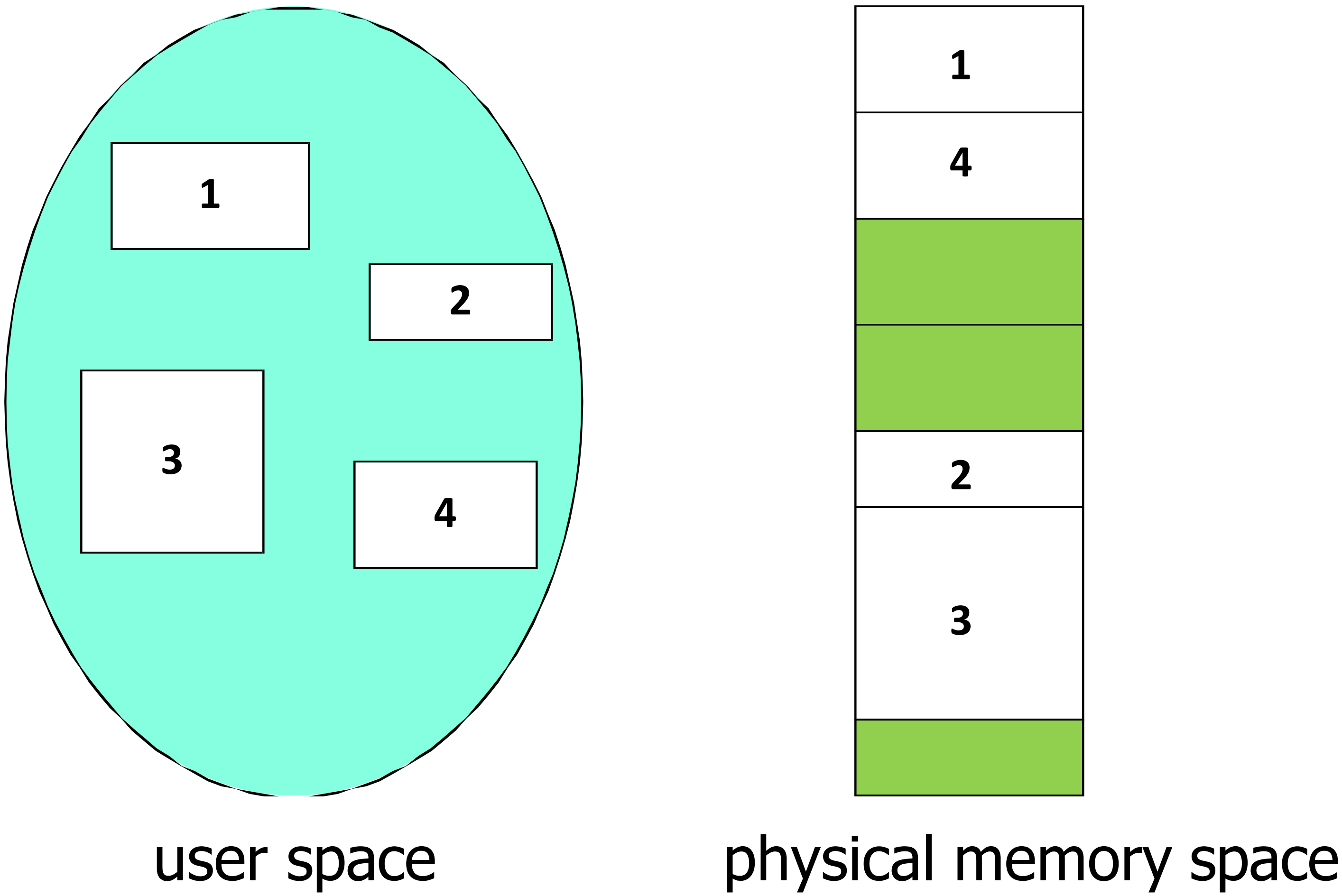
- *Segment table* – maps two-dimensional physical addresses; each table entry has:
  - *base* – contains the starting physical address where the segments reside in memory.
  - *limit* – specifies the length of the segment.
- *Segment-table base register (STBR)* points to the segment table's location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

  segment number s is legal if s < STLR.

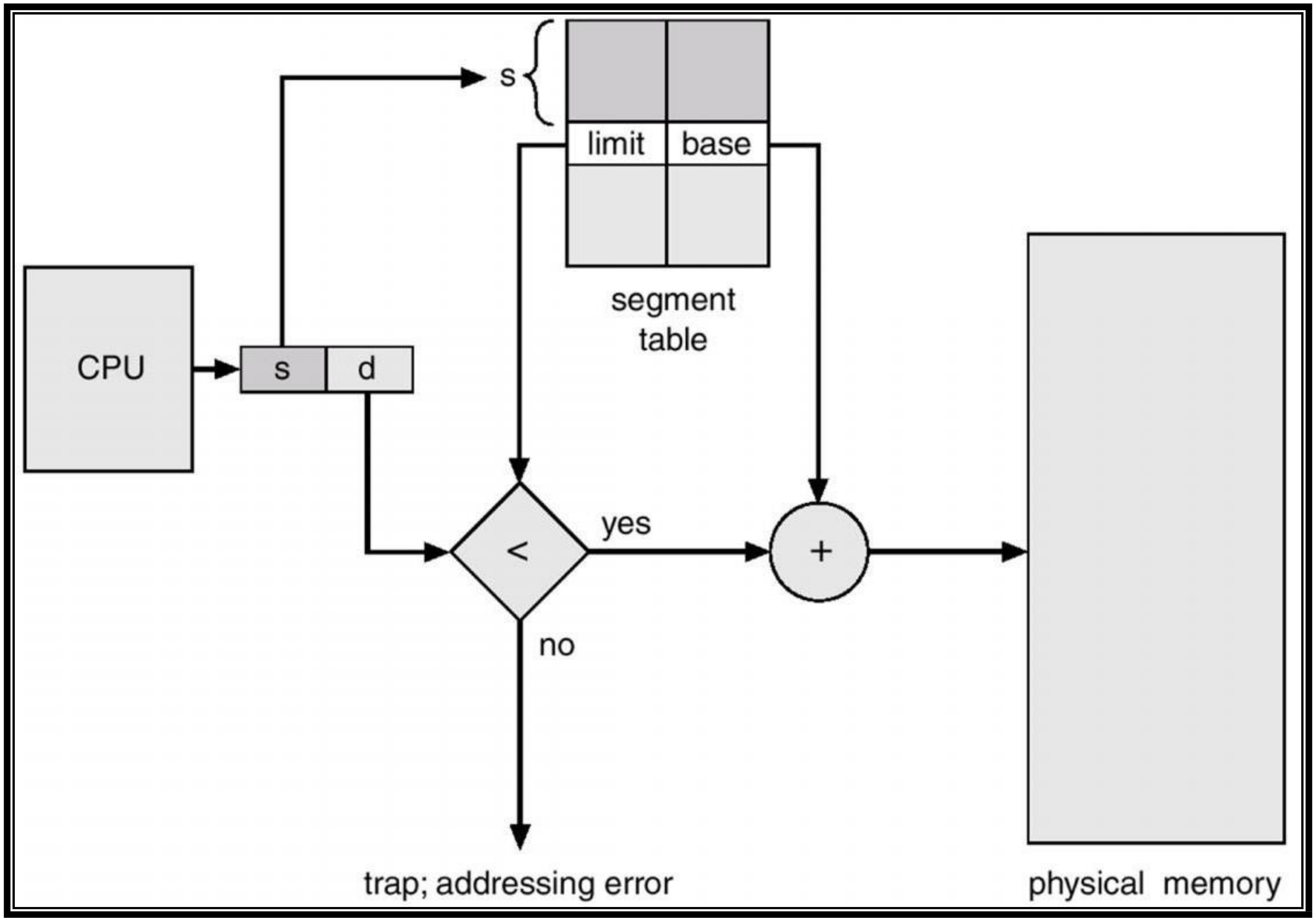
User’s View of a Program



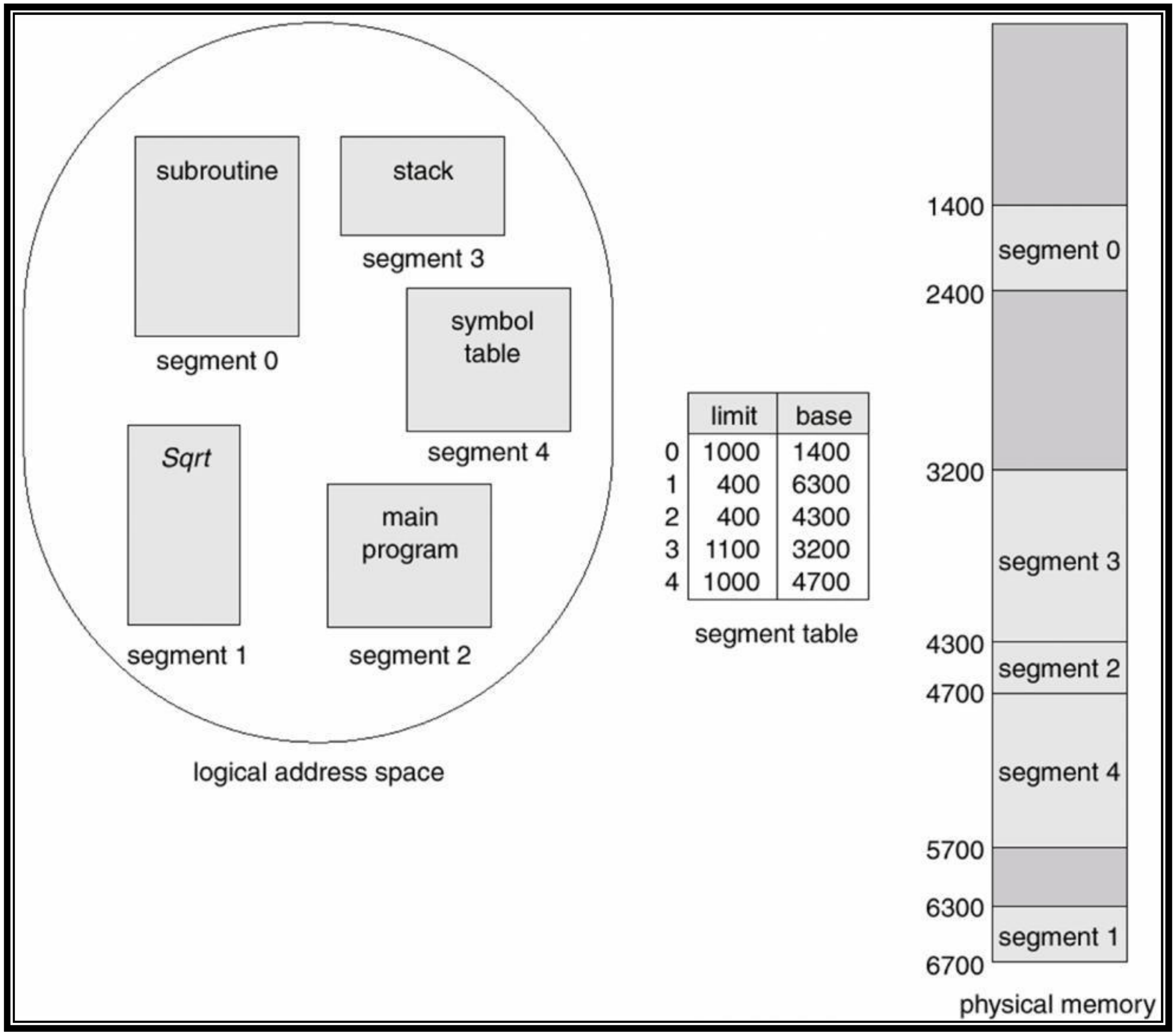
Logical View of Segmentation



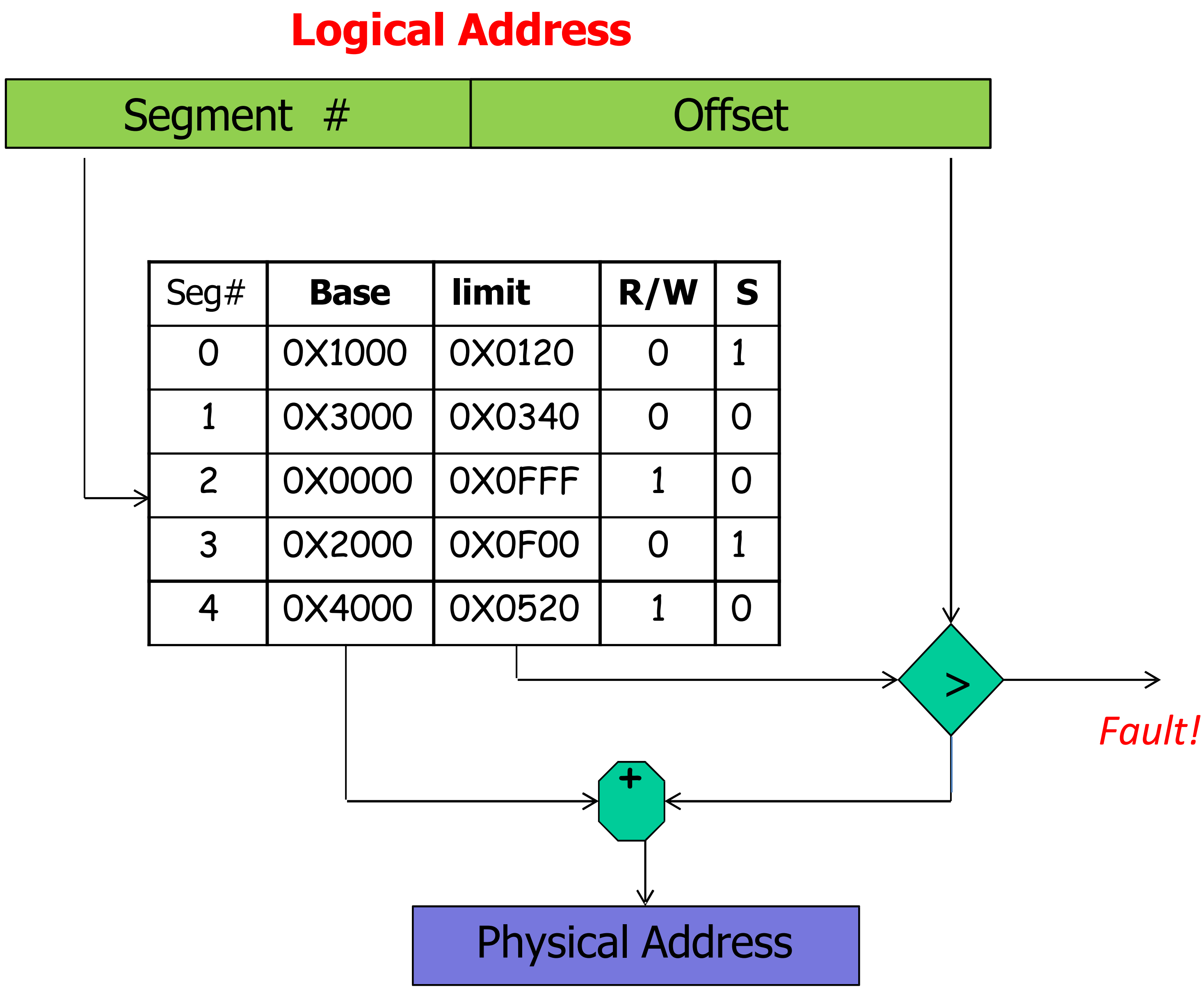
Segmentation Hardware



Example of Segmentation



Relocation with segmentation



Seg#	**Base**	**limit**	**R/W**	**S**
0	0X1000	0X0120	0	1
1	0X3000	0X0340	0	0
2	0X0000	0X0FFF	1	0
3	0X2000	0X0F00	0	1
4	0X4000	0X0520	1	0

Relocation with segmentation

Example Logical Address to physical address

Q1, On the system using segmentation, compute physical address for each of the logical address is given in the following segment table

a. 0,99
b. 2,78
c. 1,256
d. 3,22

Segment	Base	limit
0	330	124
1	876	211
2	111	99
3	498	302

Segmentation…

- Segmentation, as well as the base and bounds approach, causes **external fragmentation** and requires memory compaction.
- An advantage of the approach is that only a segment, instead of a whole process, may be swapped to make room for the (new) process.