

Operating Systems

CS220

Lecture 15

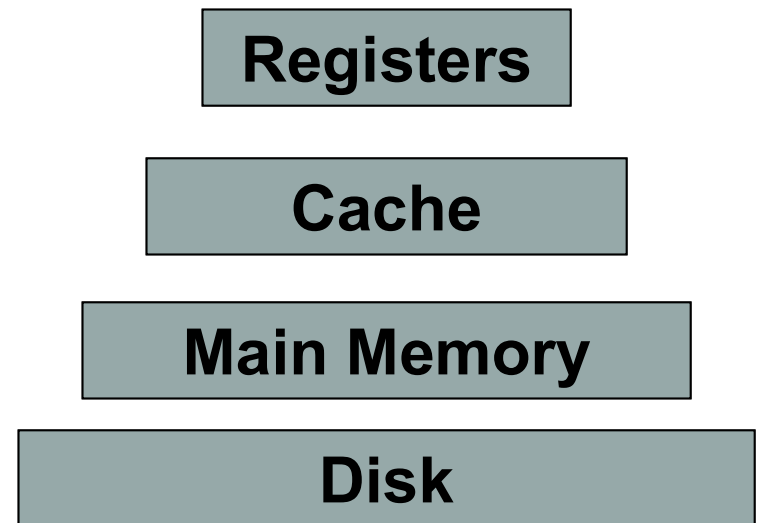
Memory Management

28th June 2021

By: Dr. Rana Asif Rehman

Background

- Program must be brought (from disk) into memory and placed within a process for it to be run
- A programmer will like to have memory
 - Infinitely large
 - Infinitely fast
 - Non volatile
- However, we have Memory Hierarchy:



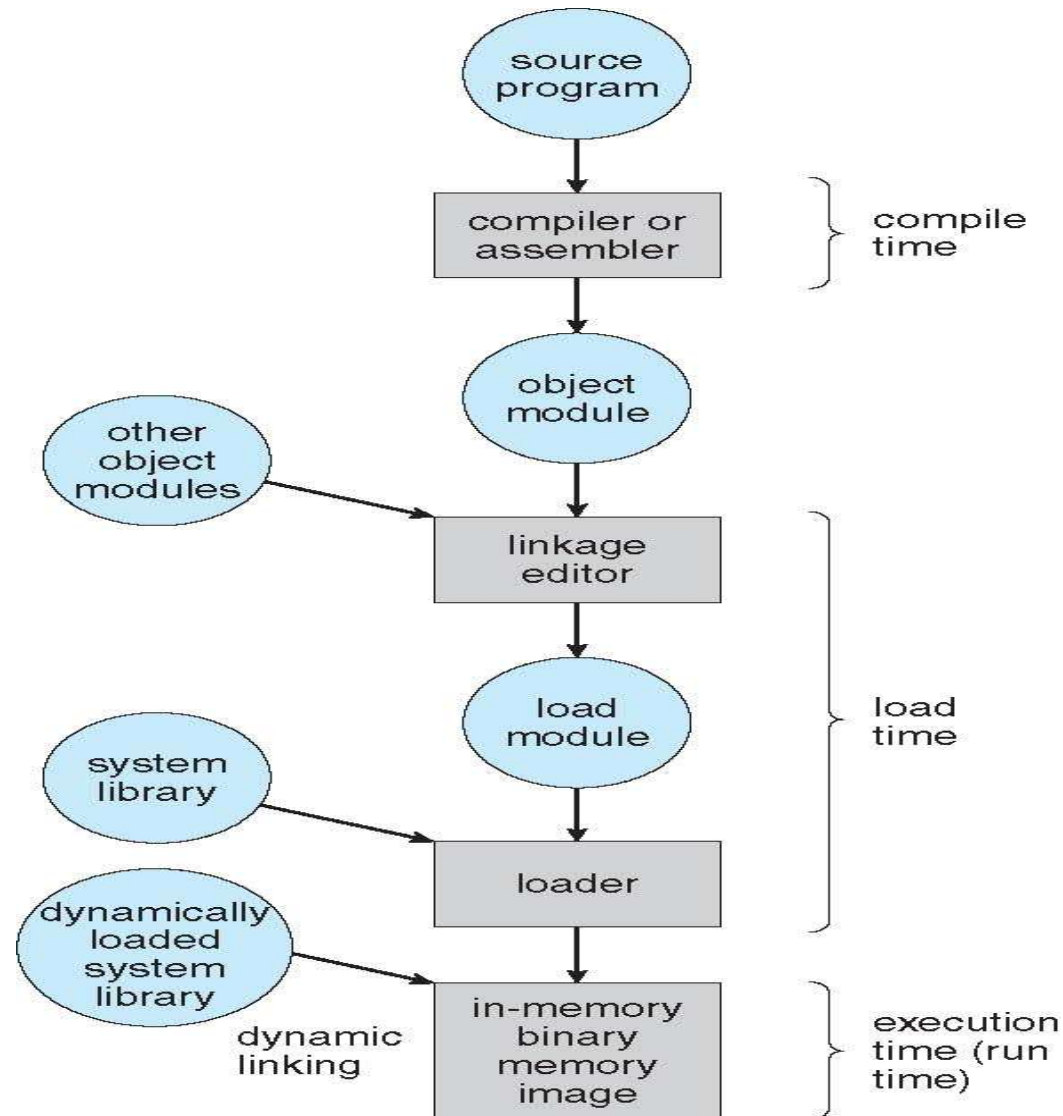
Background

- Main memory and registers are only storage CPU can access directly
- Memory unit only sees a stream of addresses + read requests, or address + data and write requests
- Register access in one CPU clock (or less)
- Main memory can take many cycles, causing a **stall**
- **Cache** sits between main memory and CPU registers
- Protection of memory required to ensure correct operation

Address Binding

- Programs on disk, ready to be brought into memory to execute form an **input queue**
 - Without support, must be loaded into address 0000
- Inconvenient to have first user process physical address always at 0000
 - How can it not be?
- Further, addresses represented in different ways at different stages of a program's life
 - Source code addresses usually symbolic
 - Compiled code addresses **bind** 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

Multistep Processing of a User Program



Memory Manager

- The part of the OS that manages the Memory Hierarchy
 - Which part of memory is in use and which is not in use
 - Allocate memory
 - De-allocate memory
 - Swapping between main memory and disks

Memory Management Requirements

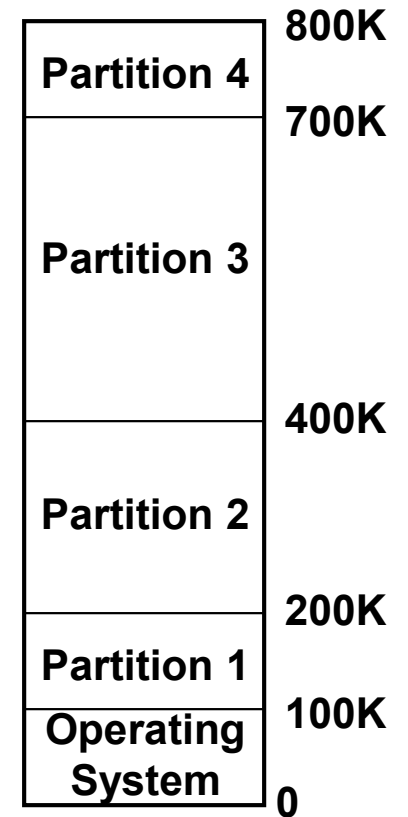
- Speed is not the only issue
 - Relocation
 - Protection
 - Sharing
 - Physical memory organization
 -

Requirements: Relocation

- The programmer does not know where the program will be placed in memory when it is executed,
 - it may be swapped to disk and return to main memory at a different location (relocated)
- Memory references must be translated to the actual physical memory address

Relocation and Protection

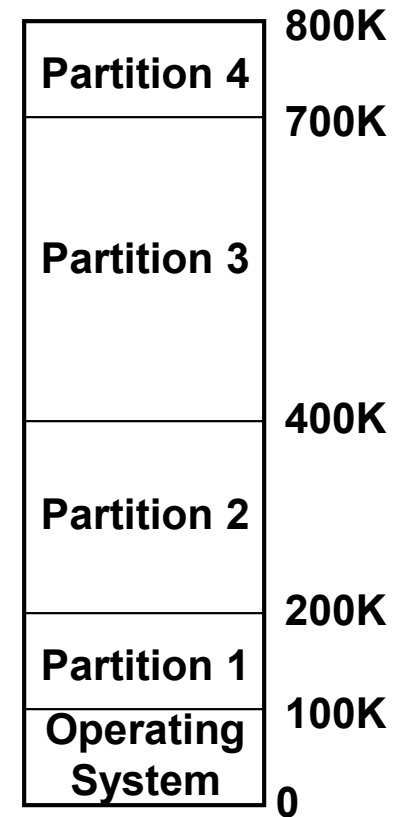
- Different jobs will run at different addresses
- Suppose, the first instruction is
 - call a procedure at absolute address 10 within the binary file



This call will jump inside the Operating system

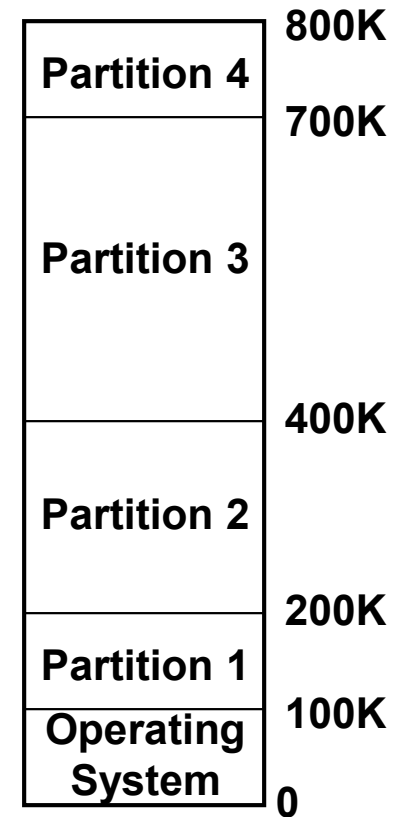
Relocation and Protection

- Solution:
- If the program is loaded in 1st partition, then
 - Call $100k + 10$
- If the program is loaded in 2nd partition, then
 - Call $200k + 10$
- So on...
- This is called **Relocation** problem



Relocation and Protection

- Solution:
 - Add an **offset** to each address in the program
 - The offset depends on the partition, e.g.,
 - if the Program is loaded in partition 1, add 100k to every address
 - if the Program is loaded in partition 2, add 200k to every address



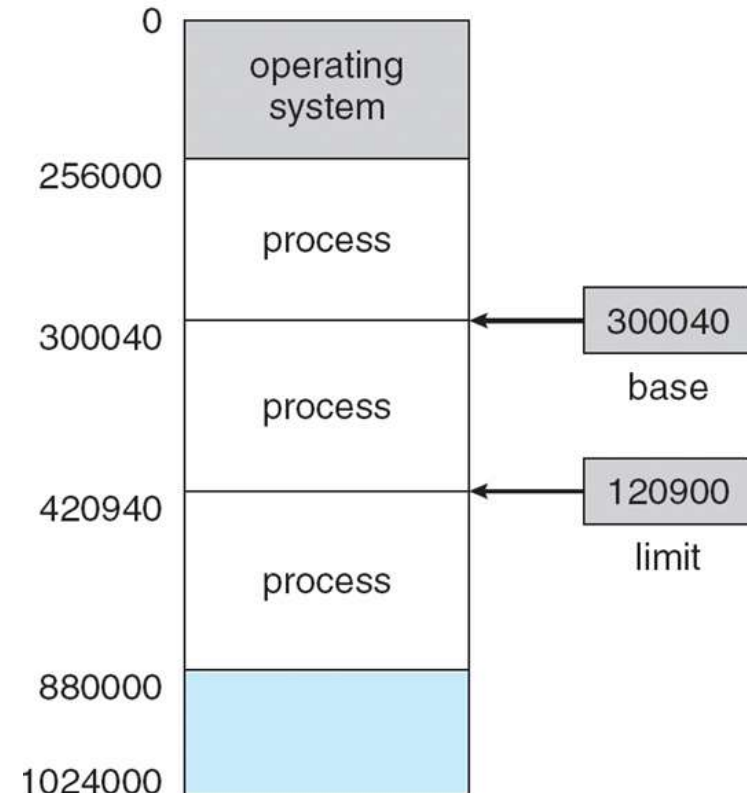
- A program can still generate an address that jumps in the OS or other user's code
- We need to **Protect** the code

Requirements: Protection

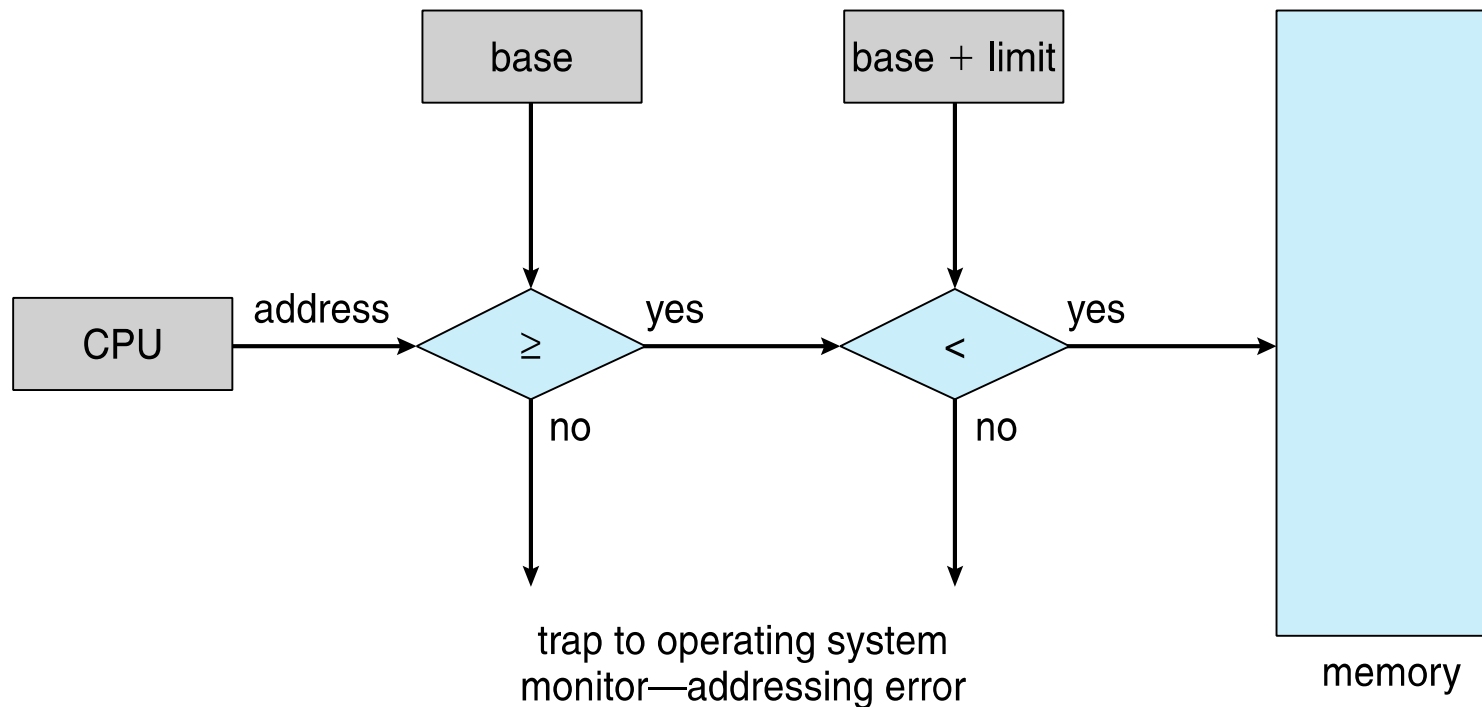
- Processes should not be able to reference memory locations in another process without permission
- Impossible to check absolute addresses at compile time
- Must be checked at run time

Relocation and Protection

- Solution:
 - Base and Limit Registers
- When a process is scheduled
 - Base Register is loaded with the starting address of the Partition
 - Limit Register is loaded with the length of the Partition
- Before referring to memory
 - Add the base register contents to generated memory address
 - Also check against the Limit register for protection

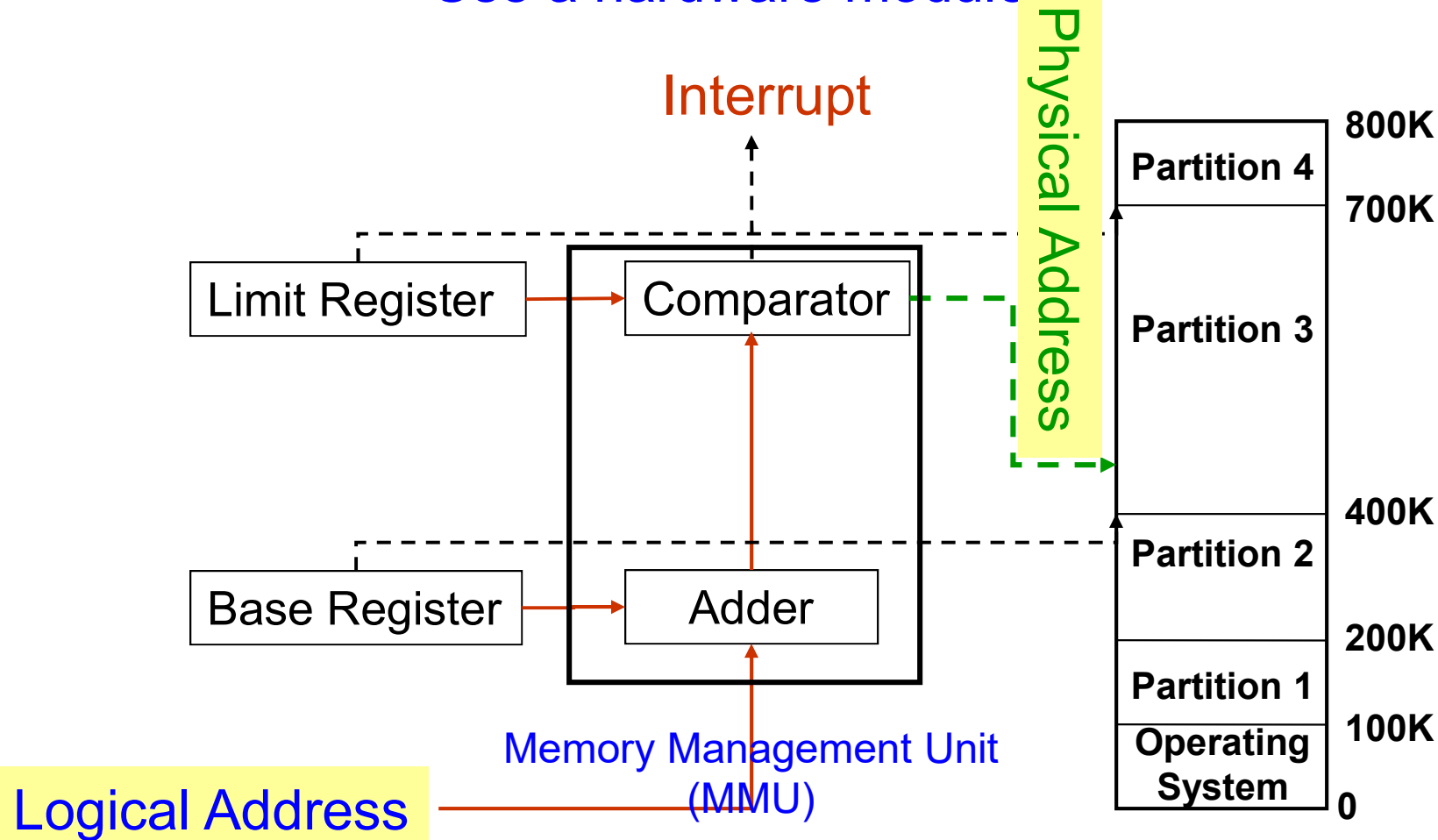


Hardware Address Protection



Relocation and Protection

Addition and comparison has to be performed on every address
Use a hardware module



Logical vs. Physical Address Space

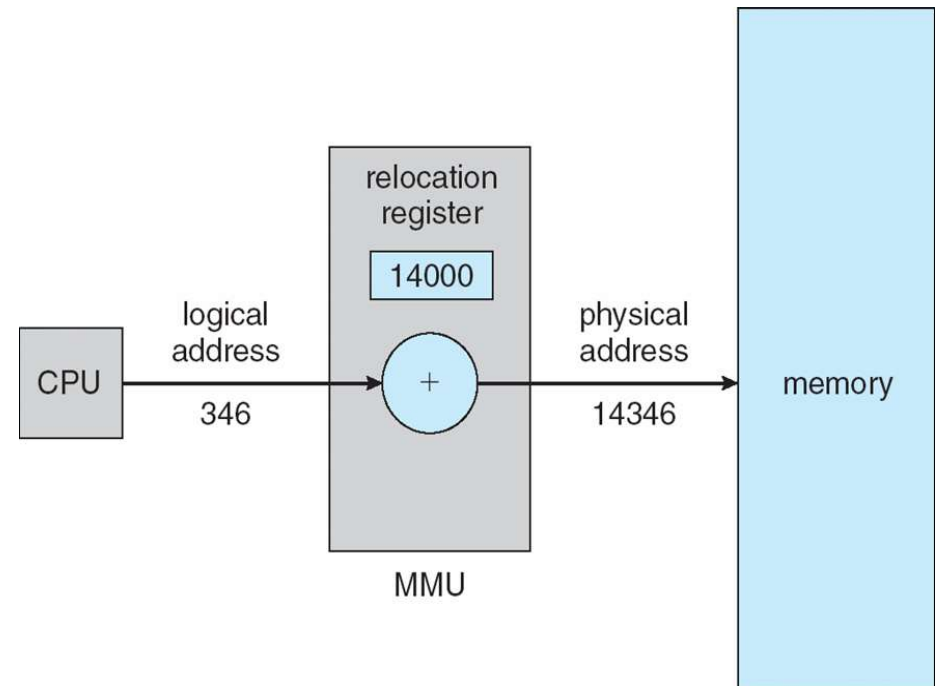
- The concept of a logical address space that is bound to a separate **physical address space** is central to proper memory management
 - **Logical address** – generated by the CPU; also referred to as **virtual address**
 - **Physical address** – address seen by the memory unit
- Logical and physical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

Memory-Management Unit (MMU)

- Hardware device that at run time maps virtual to physical address
- Many methods possible, covered in the rest of this chapter
- To start, consider simple scheme where the value in the relocation register is added to every address generated by a user process at the time it is sent to memory
 - Base register now called **relocation register**
 - MS-DOS on Intel 80x86 used 4 relocation registers
- The user program deals with *logical* addresses; it never sees the *real* physical addresses
 - Execution-time binding occurs when reference is made to location in memory
 - Logical address bound to physical addresses

Dynamic Loading

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



Dynamic relocation using a relocation register

Dynamic Linking

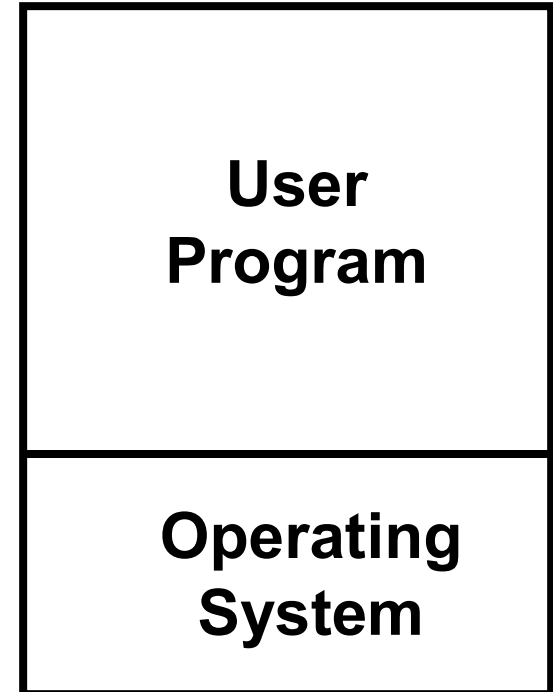
- **Static linking** – system libraries and program code combined by the loader into the binary program image
- Dynamic linking – linking postponed until execution time
- Small piece of code, **stub**, used to locate the appropriate memory-resident library routine
- Stub replaces itself with the address of the routine, and executes the routine
- Operating system checks if routine is in processes' memory address
 - If not in address space, add to address space
- Dynamic linking is particularly useful for libraries
- System also known as **shared libraries**
- Consider applicability to patching system libraries
 - Versioning may be needed

Memory Management/Organization

- In good old days, memory management was trivial as there was only one program
 - No OS, everything is responsibility of user program
- For the later uniprogramming systems, memory was shared between
 - A User program
 - Operating system

Uni-programming

- Run just one program at a time
- Memory is shared between
 - A User program
 - Operating system
- The user types a command on the prompt
- The OS
 - Copies the program from the disk to memory
 - Executes the program
 - After execution, prompt is available again
- The new program is copied into RAM, overwriting the previous one



Multi-programming systems

- Keep more than one process in memory
 - Benefits?
- How to organize memory to achieve this?
- One basic approach is to divide memory into n (usually unequal partitions)

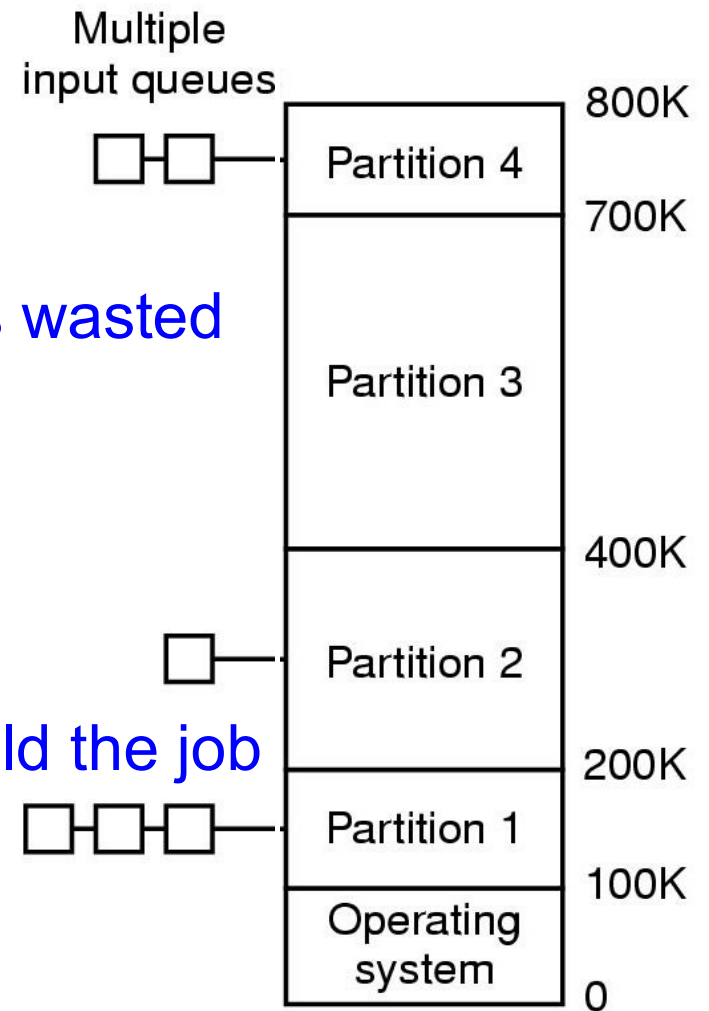
Memory Allocation Mechanisms

- Contiguous Memory Allocation
 - Fixed or Static Partitioning
 - Variable or Dynamic Partitioning
- Non-Contiguous Memory Allocation
 - Fixed or Static Partitioning (Paging)
 - Variable or Dynamic Partitioning (Segmentations)

Multiprogramming with Fixed Partitions

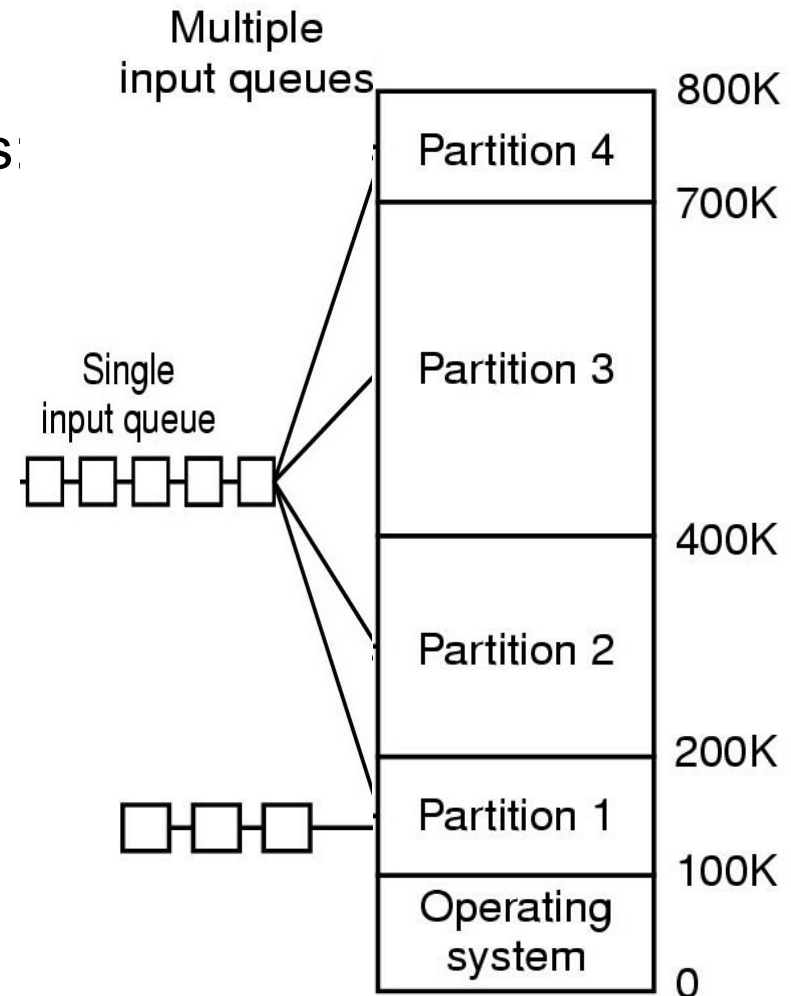
- If the job size $<$ Partition size, memory is wasted
- When a job arrives
 - its put into the input queue of

The *smallest partition large enough* to hold the job



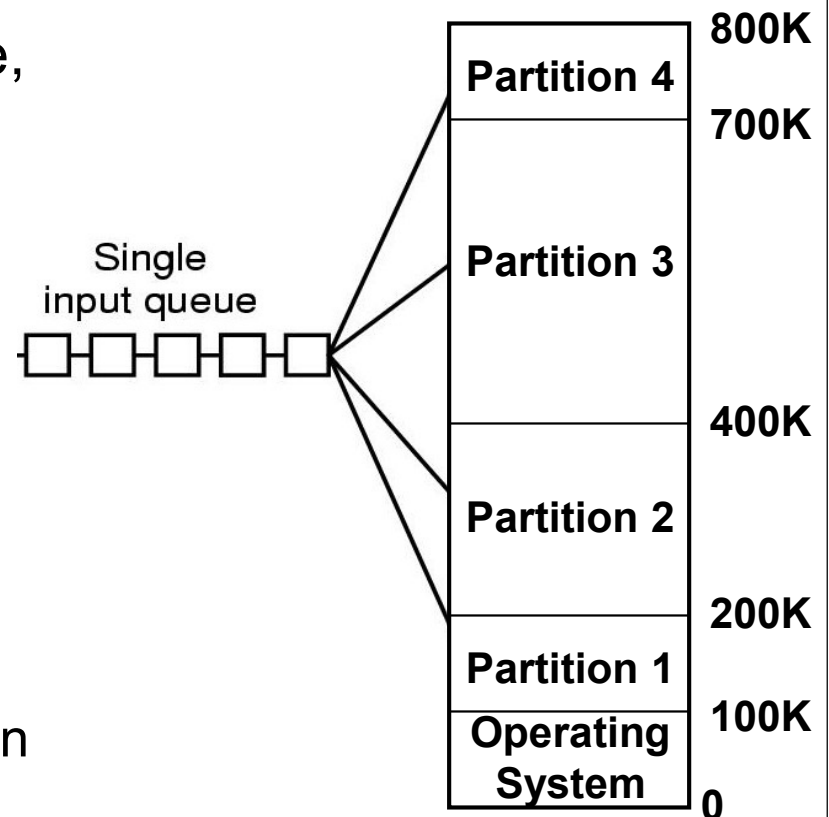
Multiprogramming with Fixed Partitions

- Disadvantage of Multiple Queues:
 - ❑ When a large partition is empty
 - ❑ And queues for small partition is full
 - ❑ Small jobs have to wait, even though plenty of memory is free
- Alternative: Maintain a single Queue



Multiprogramming with Fixed Partitions

- Whenever a partition becomes free, a job is selected
 - ❑ Closest to the front of the queue
 - ❑ Smaller than the partition size
- Undesirable to waste a large partition for smaller job
 - ❑ Search the queue, find the largest job that fits is
- Unfair for smaller jobs
 - ❑ A job may not be skipped more than k times

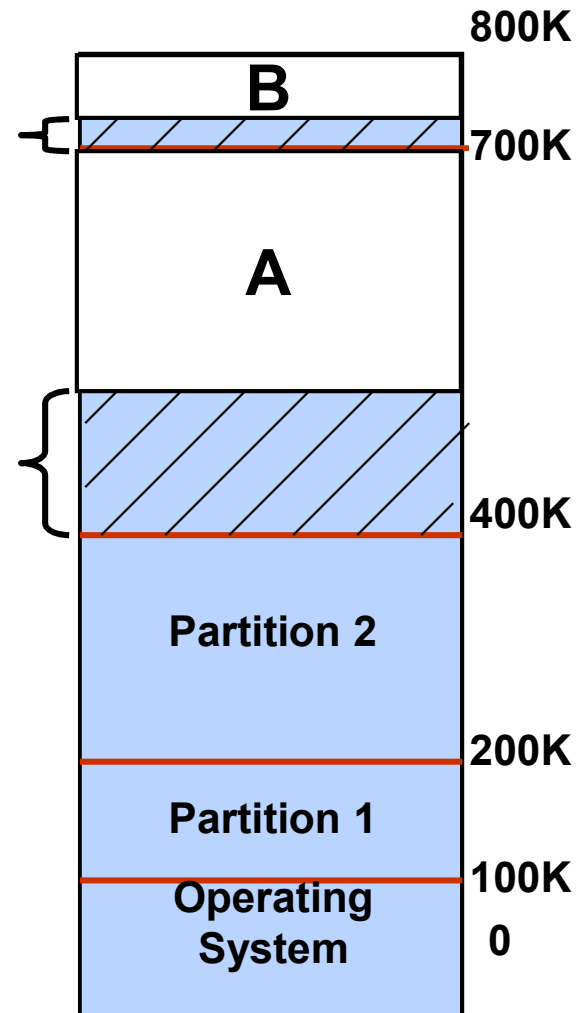


Internal Fragmentation

Memory that is internal to a partition but not being used

Internal Fragmentation

Internal Fragmentation



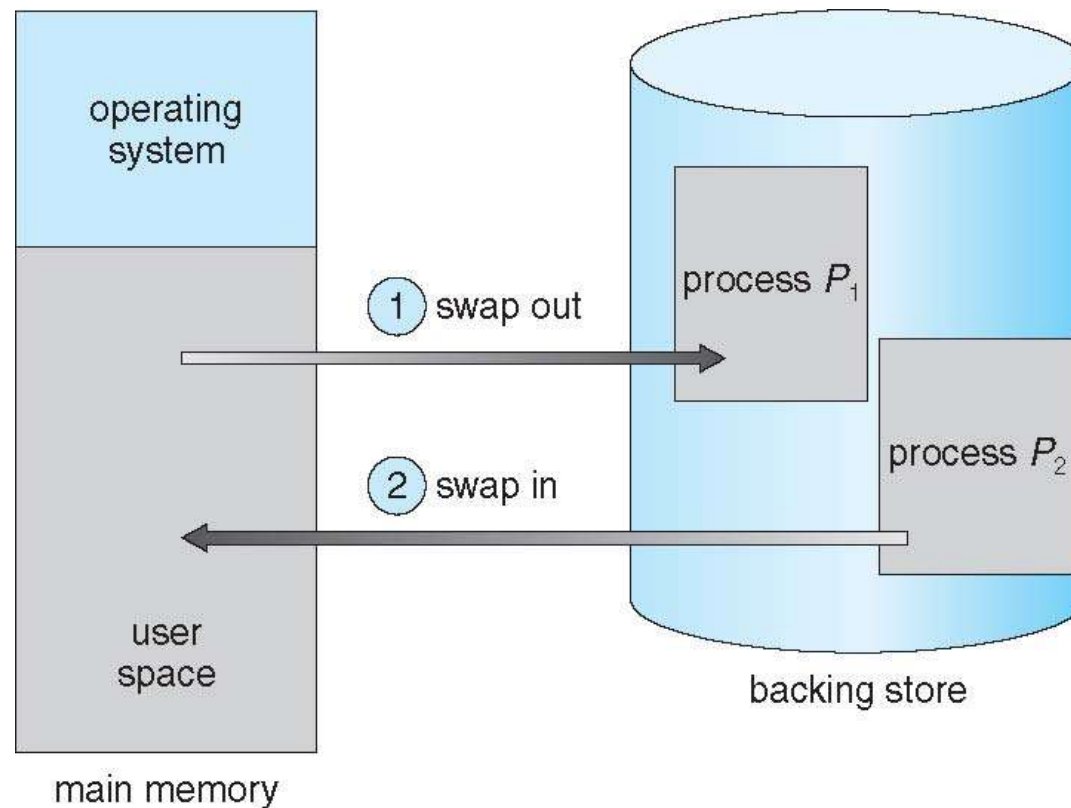
Swapping

- If *not enough space* in memory for all the currently active processes
- Excess processes must be kept on disk
 - Fully → Swapping
 - Partially → Virtual Memory
- Example of Swapping
 - Round Robin Scheduling
- When the quantum expires
 - Swap out the currently running process
 - Swap in another process to freed memory space

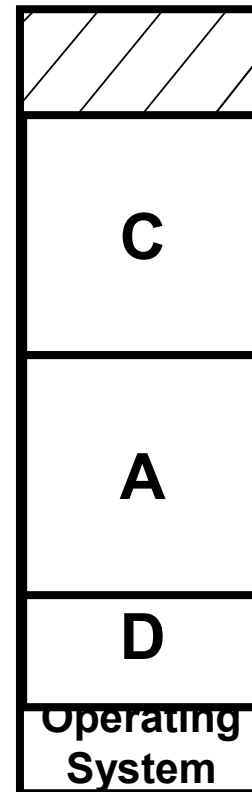
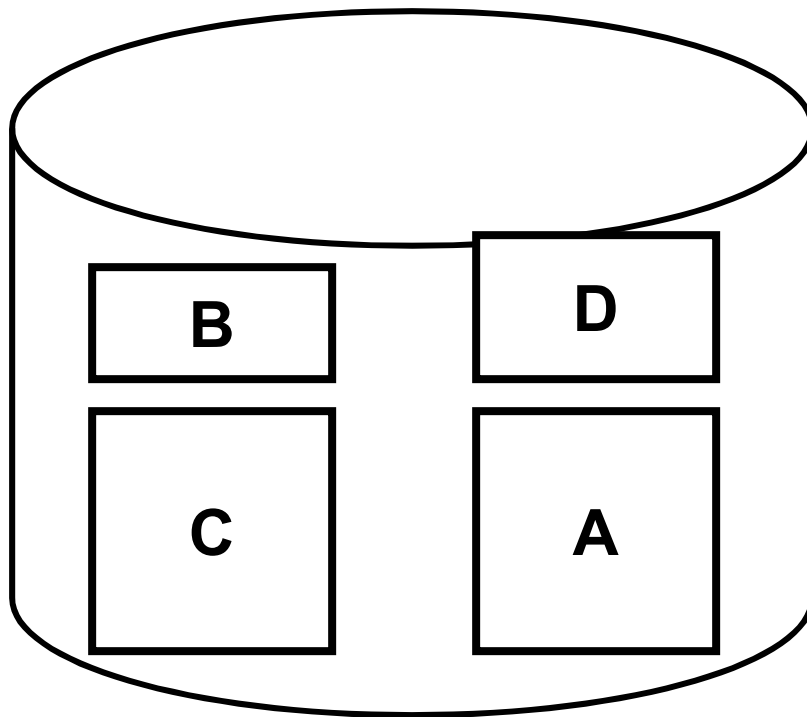
Swapping

- Another Example:
 - Priority based scheduling
- If a higher priority process arrives
 - Swap out a lower priority process
 - Swap in the higher priority process
- When the higher priority process exits
 - Swap in the lower priority process

Schematic View of Swapping



Swapping

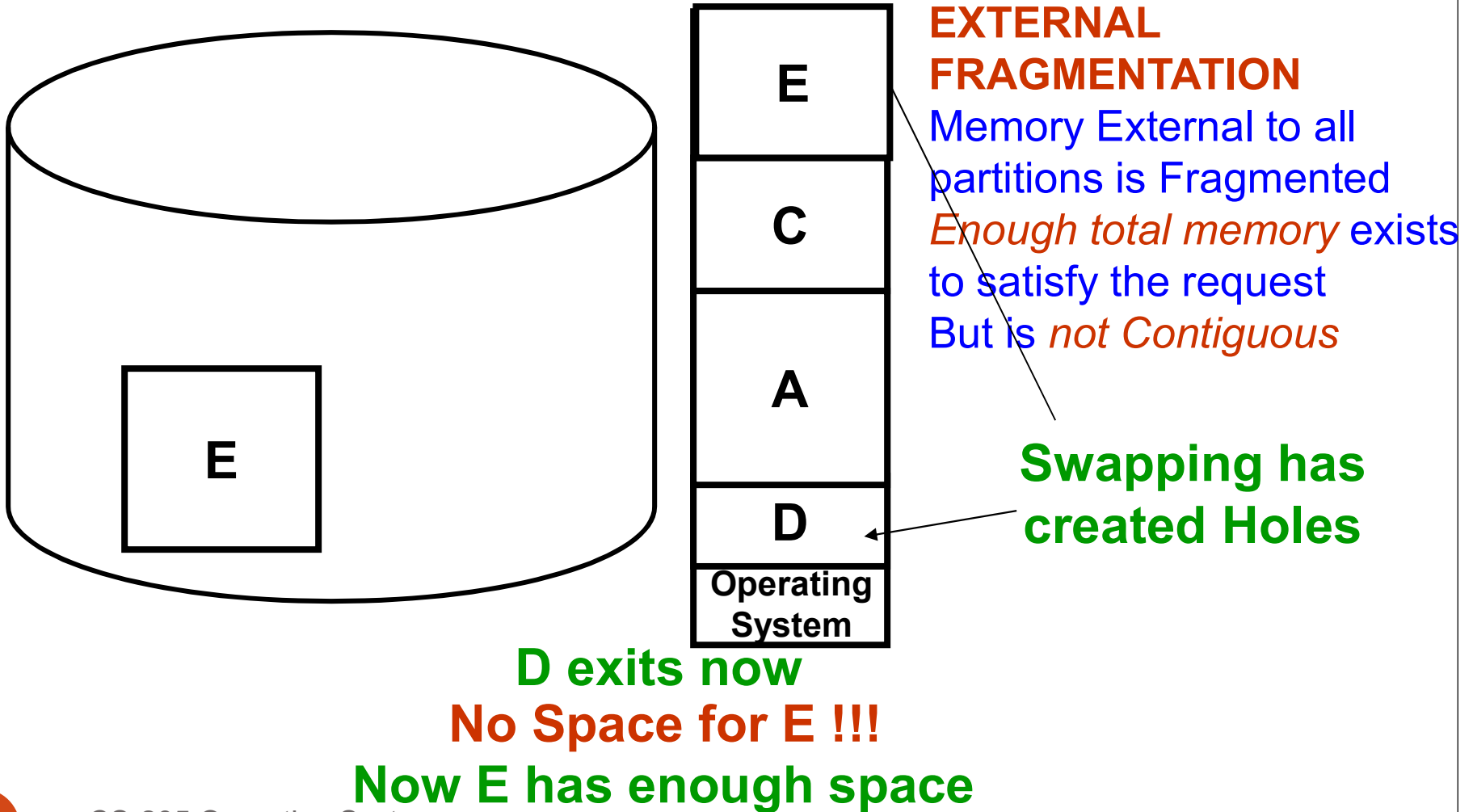


The number, size and location of the partition is decided dynamically.

D is of Higher Priority, A is of Lower Priority!!!
B exits now, Swap in A

Swapping

Its possible to combine all the holes into one big hole
This is called COMPACTION



Compaction

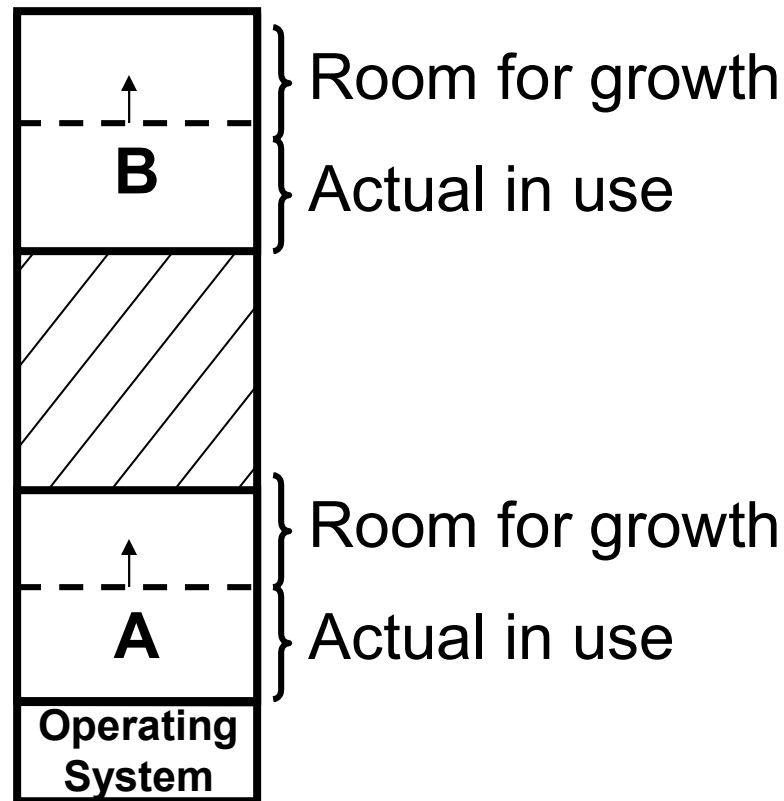
- Compaction involves CPU overhead
- If addresses are not generated relative to the partition location:
 - Then Compaction is not possible
- How much memory should be allocated for a process?
 - If all the memory is allocated statically in a program
 - Then, OS knows exactly the amount of memory to be allocated: **executable code + variables**
- However, if memory is allocated dynamically (say using **new**)

Memory allocation for Processes

- Problem may occur whenever a process tries to grow
- If a **hole** is adjacent to the growing process then it can be allowed to grow
- If **another process** is adjacent to the growing process, then
 - The growing process should be **moved to a larger hole**
 - If a larger hole is **not** available, then, one or more processes should be **swapped out**
 - If a process **cannot** be swapped out (say, there is not enough space on disk
 - The growing process has to **wait**
 - Or should be **killed!!!**

Memory allocation for Processes

- If most of the processes grow as they run,
- It is better to allocate a little extra memory for a process

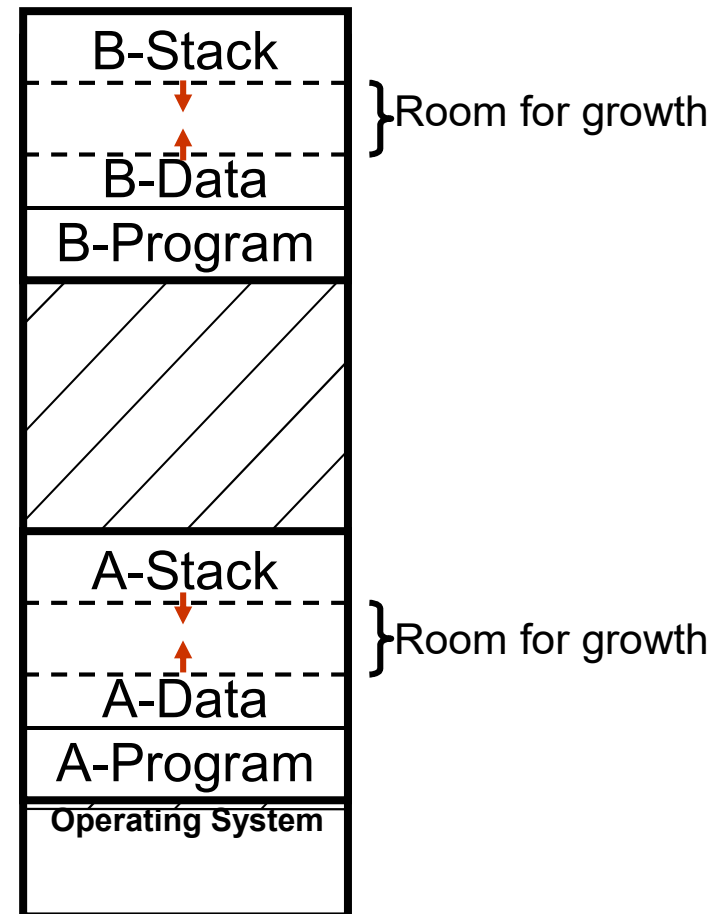


Memory allocation for Processes

- If processes have two growing segments
 - Data
 - (as heap for dynamically allocated variables)
 - Stack

• The memory can be used by either of the two segments

- If it runs out the process either
 - Has to be moved to a larger hole
 - Or swapped out
 - Or Killed

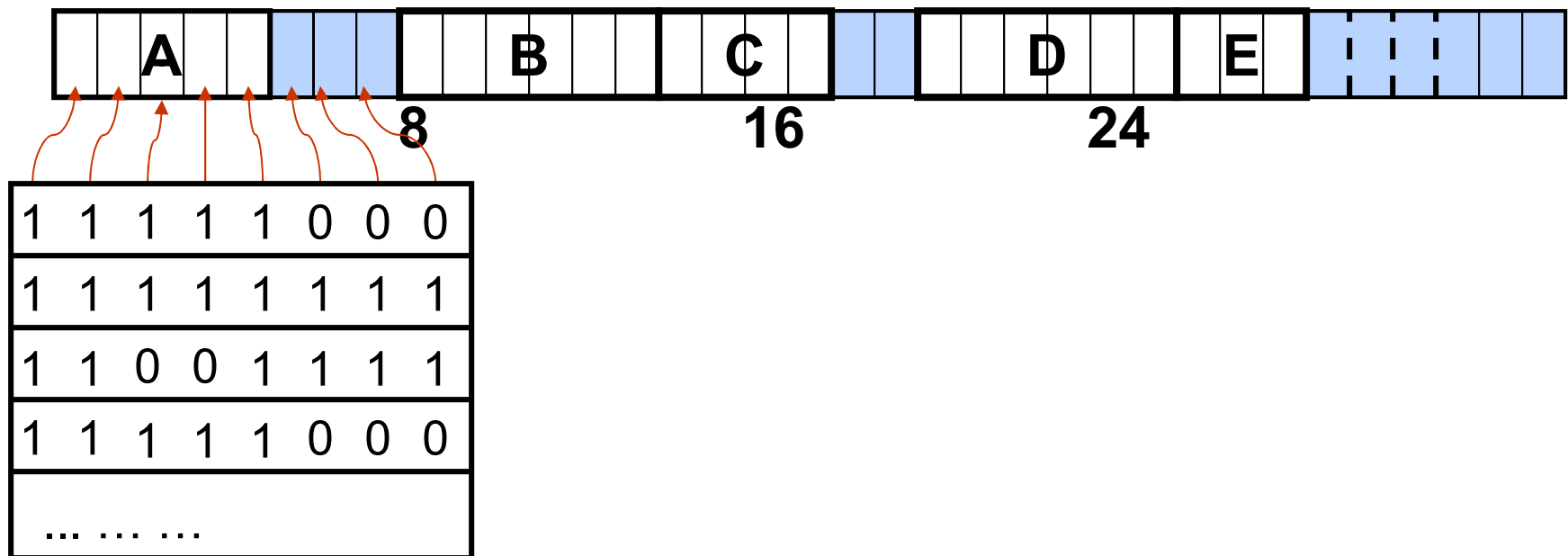


How to keep track of Memory

- We need to keep track of
 - Allocated memory
 - Free memory
- Memory management with bitmaps
- Memory management with linked lists

Memory management with bitmaps

- Memory is divided up into allocation units
 - Few words
 - Or several kilobytes



Memory management with bitmaps

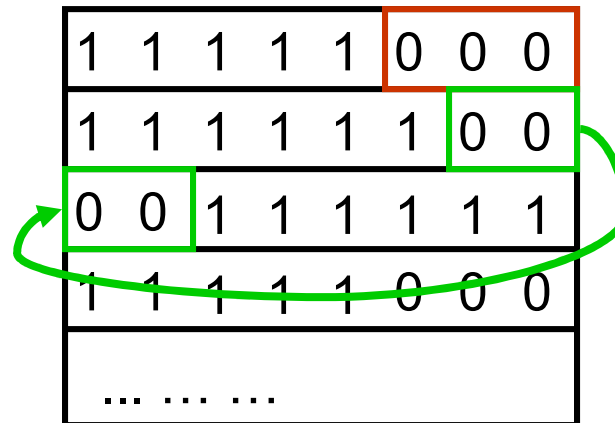
- Size of allocation unit is important
- Smaller allocation unit
 - Larger bitmap required
- Larger allocation unit
 - Smaller bitmap required
 - More memory will be wasted
 - If the process size is not an exact multiple of the allocation unit

Memory management with bitmaps

- To bring a k unit process in memory
- Search for a k run consecutive 0 bits in the map
- Search can be slow
- Since, k run may cross word boundaries

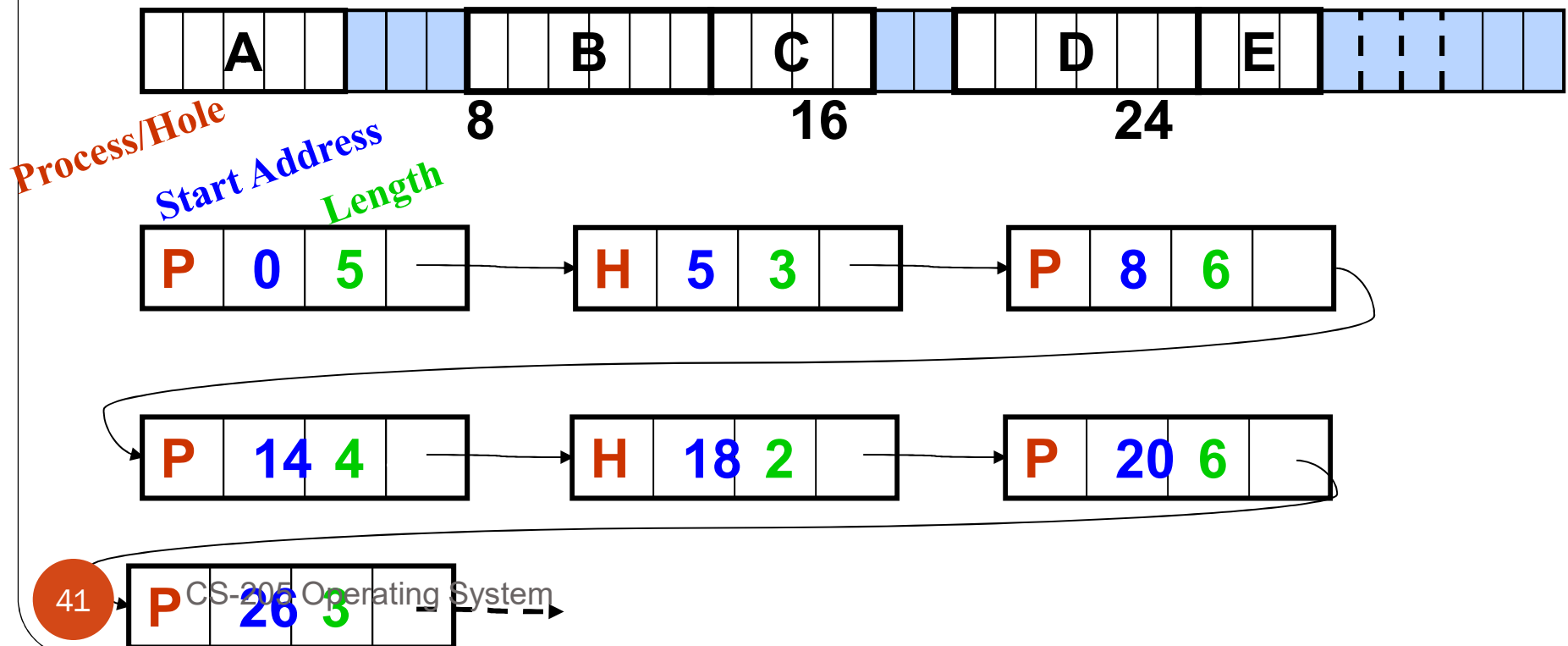
Find run of length = 3

Find run of length = 4



Memory management with Linked Lists

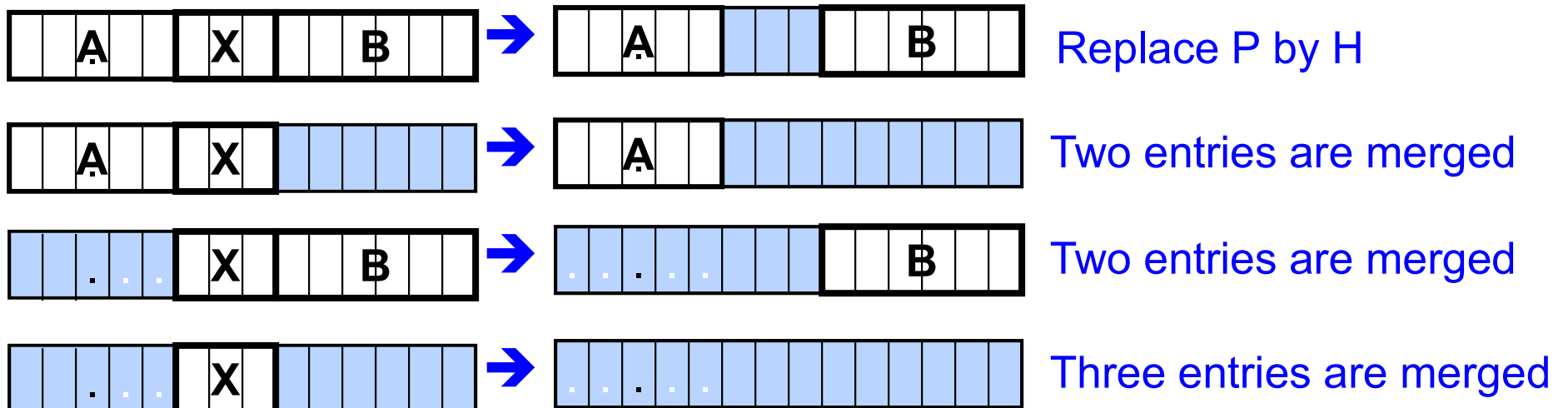
- Linked list of memory segments
 - Free segments ➔ Holes
 - Allocated segments ➔ Processes



Memory management with Linked Lists

- Segment list is sorted by addresses
- Sorting helps in updating the list, when a process is swapped out or exits
- A process usually has two neighbors

Updating the list requires



Memory Allocation with Linked Lists

- Several algorithms for allocating memory with linked list
- **FIRST FIT:**
 - Scan the segment list, until
 - A hole large enough is found
 - Split the hole into two pieces (if not exact match)
 - One for the new process
 - One for the unused memory
- The algorithm is fast since it searches as little as possible

Memory Allocation with Linked Lists

- **NEXT FIT**
 - Works the same as First Fit, except,
 - Does not always start searching from the beginning
 - Rather starts searching the list, from where it left last time
- Simulations show, that gives no better performance than First Fit

Memory Allocation with Linked Lists

- **BEST FIT**
 - Search the entire list
 - Take the smallest hole that is adequate
- Best Fit tries to find the hole closest to the size of Process
- Slower than First Fit
 - Every time has to search the entire list
- But still results in more memory wastage
 - Tends to fill up memory, with tiny useless holes
 - First Fit generates larger holes on the average

Memory Allocation with Linked Lists

- **WORST FIT**

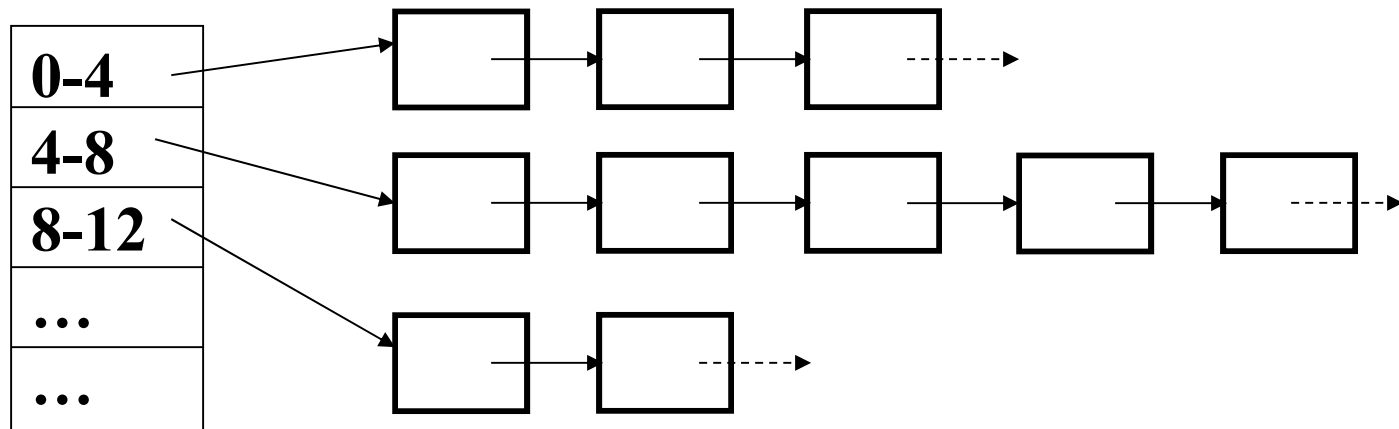
- Take the largest hole available
- So that, the hole broken off will be large enough to be useful
- What if the larger holes left by worst fit are not useful
- ➔ more memory wasted than Best Fit

Memory Allocation with Linked Lists

- Search time of all the four algorithms can be improved by
 - Keeping separate lists for Process and Holes
 - While allocating memory only have to search the lists of holes
- DRAWBACK:
 - Problem while de-allocating memory
 - A node from the process list has to be inserted in the hole list

Memory Allocation with Linked Lists

- The Hole list can be kept sorted by size
- As soon as a hole that fits is found, no more searching is required
- Another improvement can be,
 - Maintain different lists for different sizes of holes
 - QUICK FIT



Memory Allocation with Linked Lists

- **QUICK FIT**
- Finding a hole of required size is extremely fast
- **DRAWBACK:**
 - When a process terminates/swapped out
 - Finding the neighbor to see whether the merge possible is expensive
 - If merge is not done, then sooner memory will be fragmented into large holes

Terminologies

- Sleep
- Standby
- Hibernate
- Hybrid
- Shutdown
- Restart
- Lock
- Switch User
- Log out / Sign out

References

- Chapter 3, Modern Operating System
- http://cseweb.ucsd.edu/classes/sp00/cse120_A/mem.html
- Operating System Concepts (Silberschatz, 9th edition)
Chapter 8