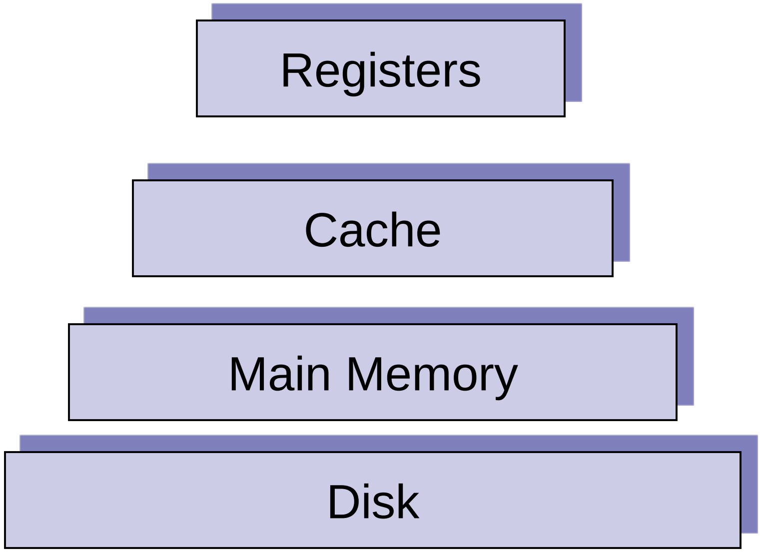
Lecture 6

Introduction to Memory Management

Operating Systems

Memory Management

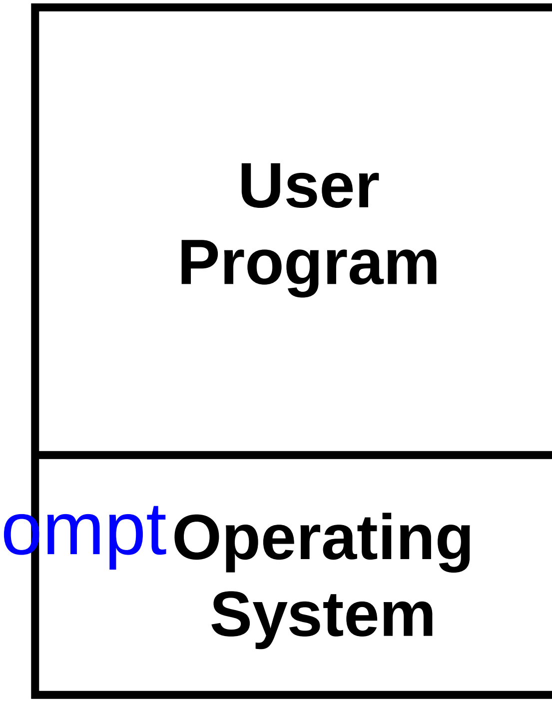
- A programmer will like to have memory
  - Infinitely large
  - Infinitely fast
  - Non volatile
- However, we have Memory Hierarchy:



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
  - Deallocate memory
  - Swapping between main memory and disks

Uniprogramming

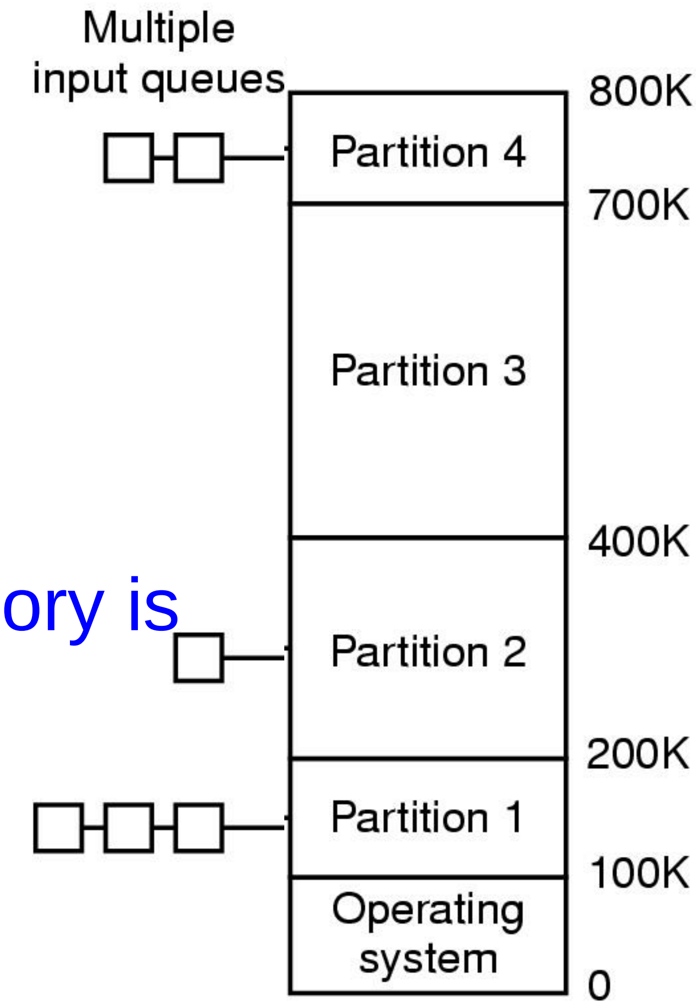


- Run just one program at a time
- Memory is shared between
  - A User program
  - Operating system

- 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

Multiprogramming with Fixed Partitions

- Multiprogramming:
  - When one process blocks due to I/O
  - Another one can use the CPU
- Divide memory into n (usually unequal fixed partitions)



•If the job size < Partition size, memory is wasted

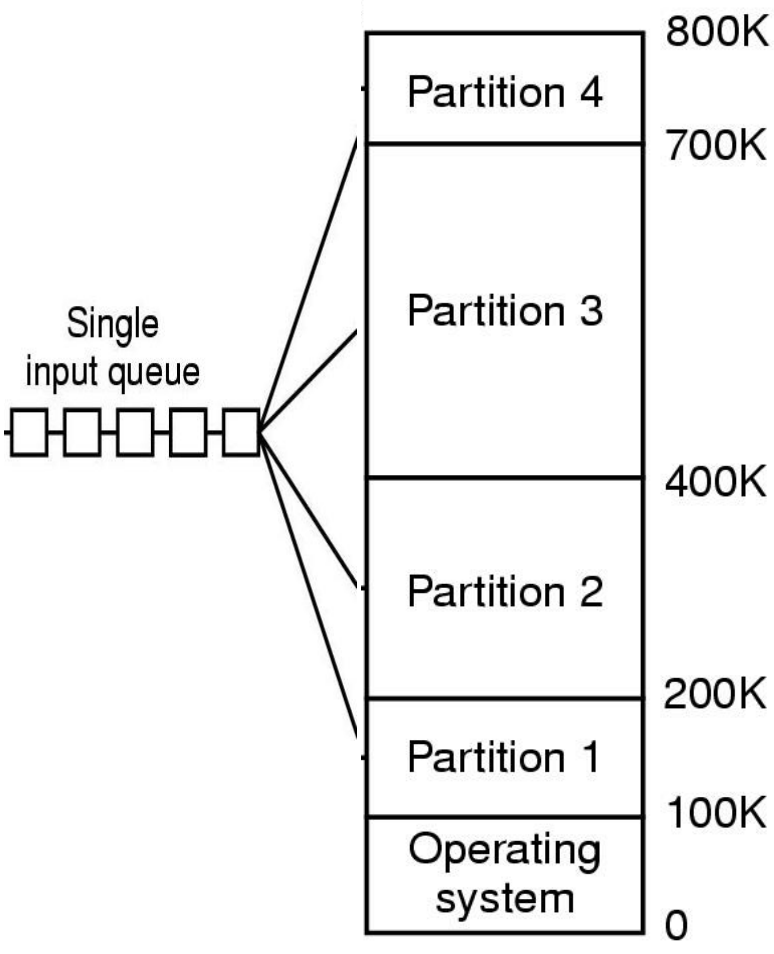
•When a job arrives

▪put it into the queue of

The *smallest partition* *large enough* to hold the job

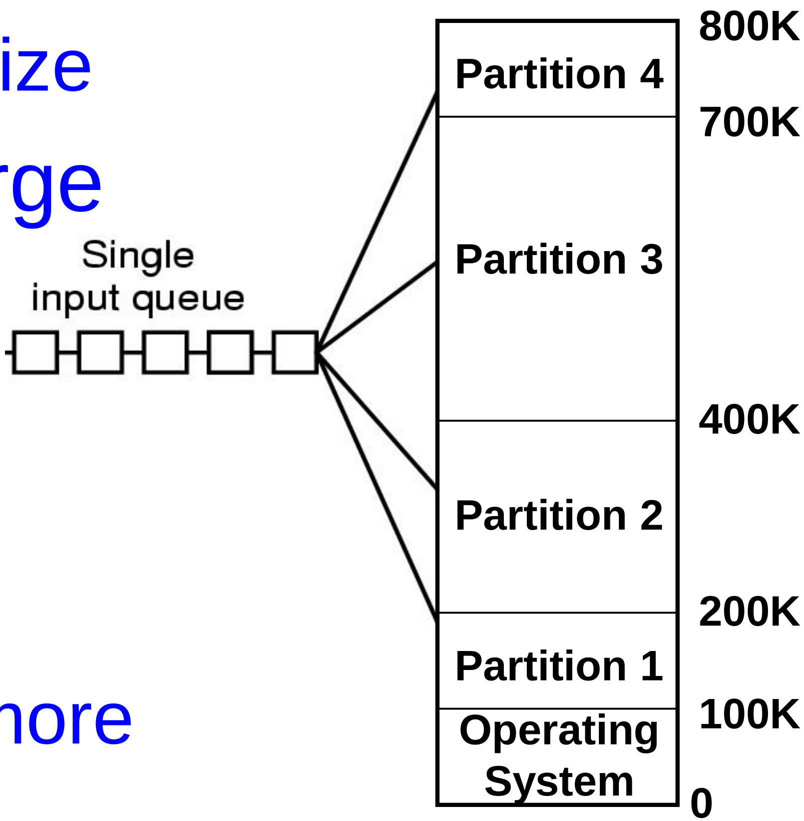
Multiprogramming with Fixed Partitions

- 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

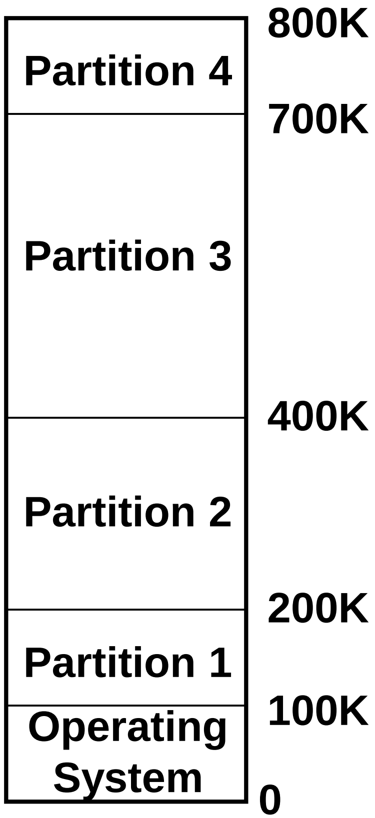
- Whenever a partition becomes free, a job is selected
  - Closest to the front of the queue
  - Smaller than the partition size
- Undesirable to waste a large partition for smaller job
  - Search the queue, find the largest job that fits in
- Unfair for smaller jobs
  - A job may not be skipped more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



Relocation and Protection

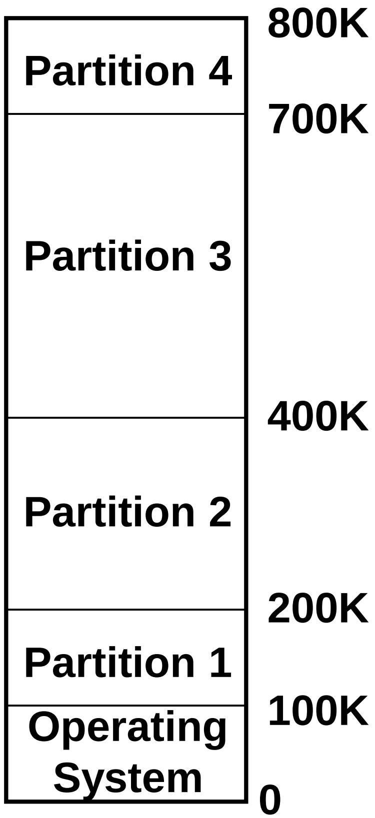
- Different jobs will run at different addresses
- Linker:
  - Combines the user procedures and the library procedures
  - Produces a single binary file
- Suppose, the first instruction is
  - call a procedure at absolute address 10 within the binary file
- If the process is loaded in 1st partition

•This call will jump inside the Operating system



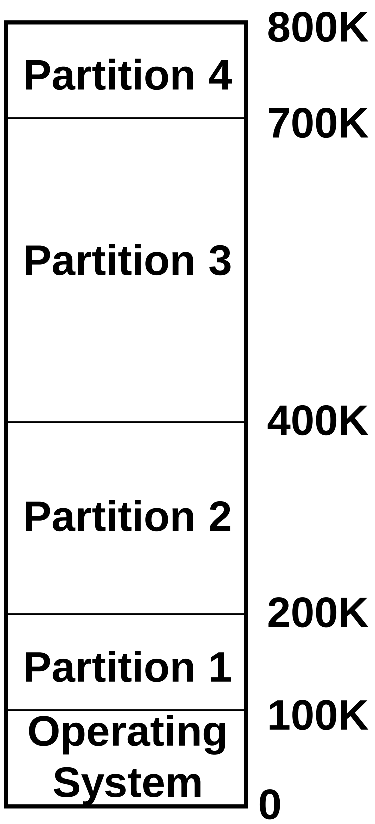
Relocation and Protection

- 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

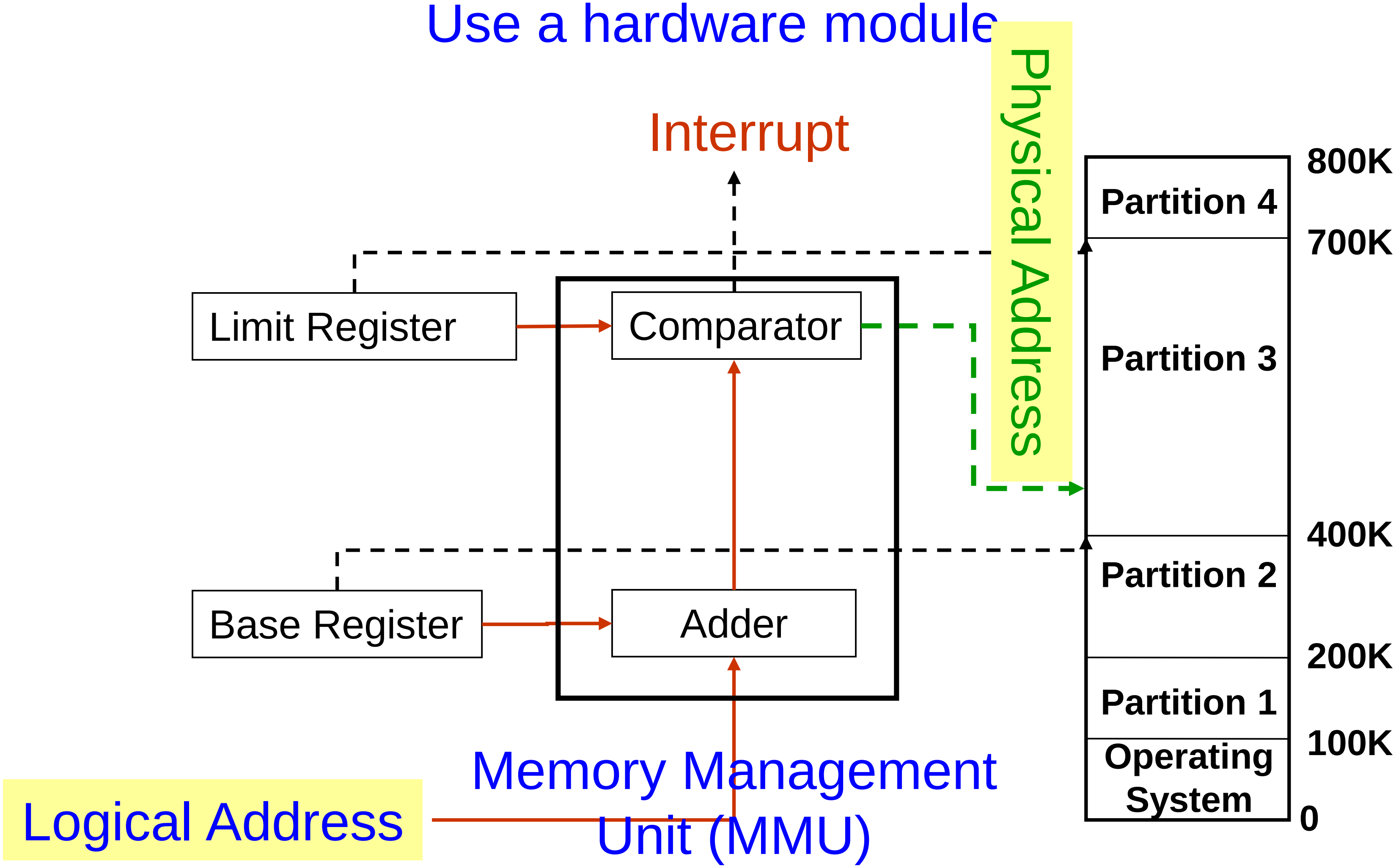
Relocation and Protection

- Solution:
  - Base and Limit Registers
- When a process is scheduled
- Base Register
  - Loaded with the starting address of the Partition
- Limit Register
  - 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

Relocation and Protection

Addition and comparison has to be performed on every address

Use a hardware module

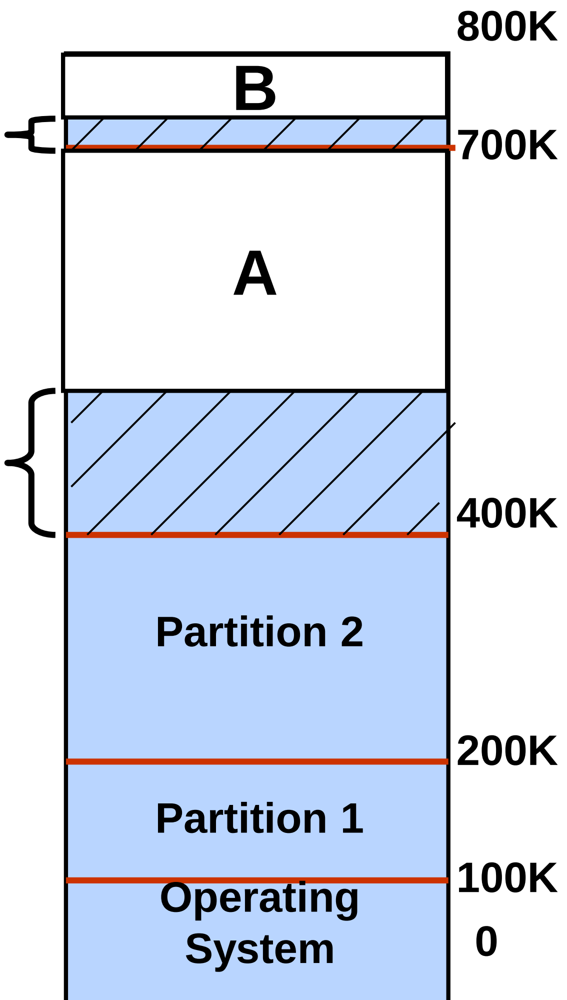


Internal Fragmentation

Memory that is internal to a partition but not being used

Internal Fragmentation

Internal Fragmentation



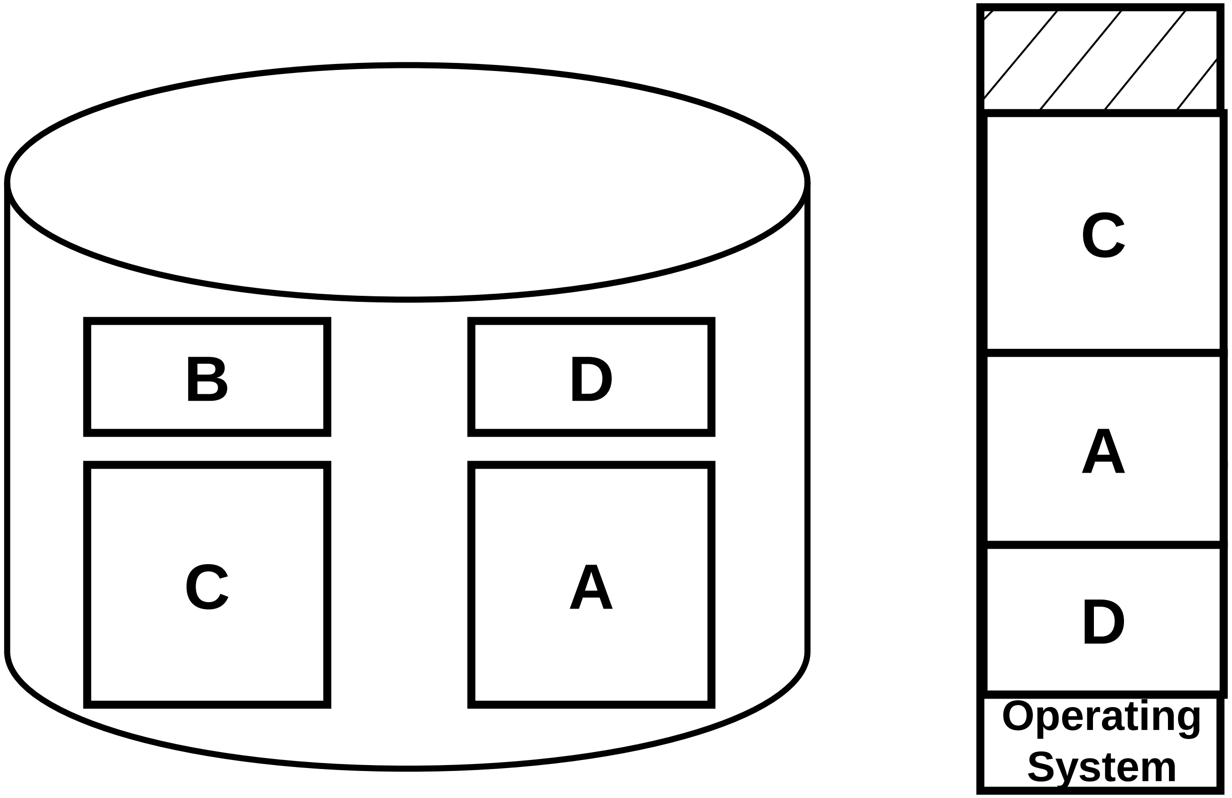
Swapping

- If ***not enough space*** in memory for all the currently active processes
- Excess processes must be kept on disk
- 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

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

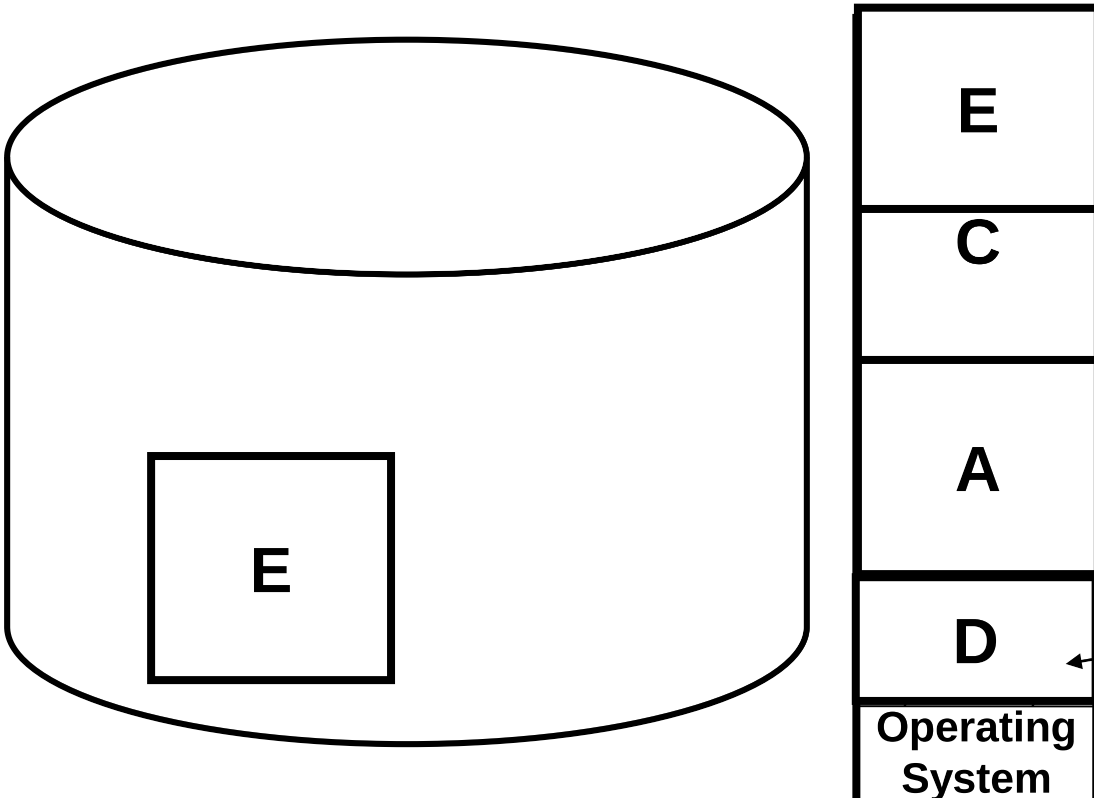
This is called COMPACTION

EXTERNAL FRAGMENTATION

Memory External to all partitions is Fragmented

Enough total memory exists to satisfy the request

But is *not Contiguous*



Swapping has created Holes

D exits now

No Space for E !!!

Now E has enough space

Swapping

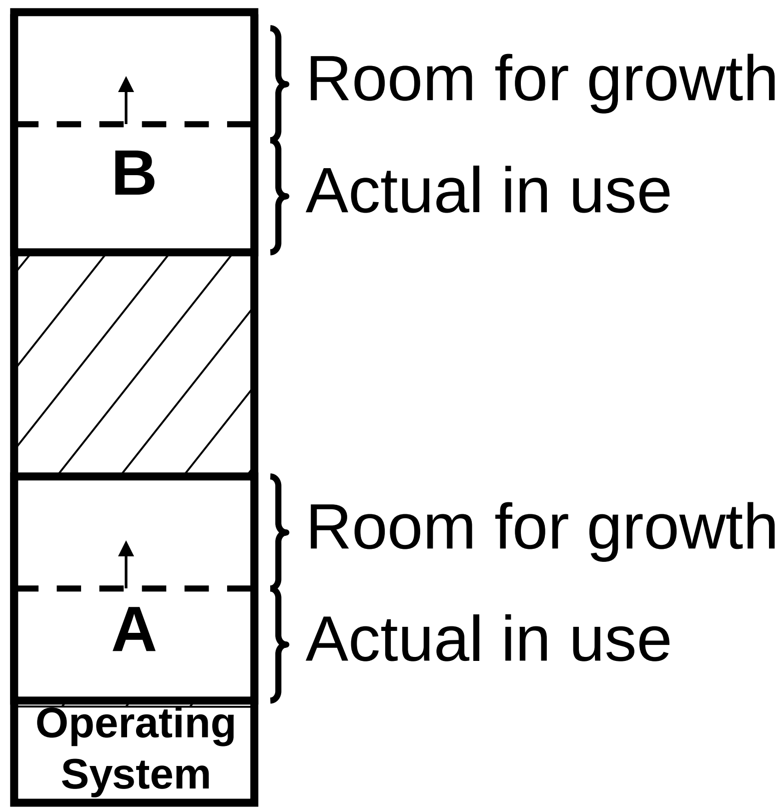
- Compaction involves CPU overhead
- If addresses are not generated relative to the partition location:
  - Then Compaction is not possible
- How much memory should be allocated for a process?
  - If all the memory is allocated statically in a program
  - Then, OS knows exactly the amount of memory to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

Swapping

- 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!!!

Swapping

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



Swapping

- 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