Memory Management

Nirmala Shinde Baloorkar
Assistant Professor
Department of Computer Engineering

Outline

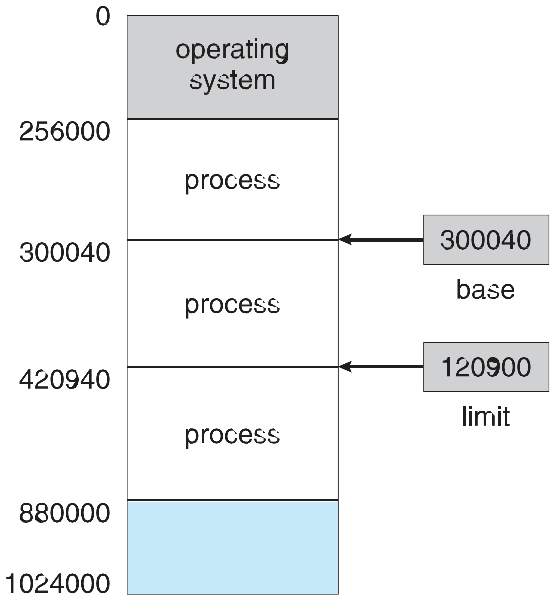
- Memory protection
- Memory allocation

Memory management

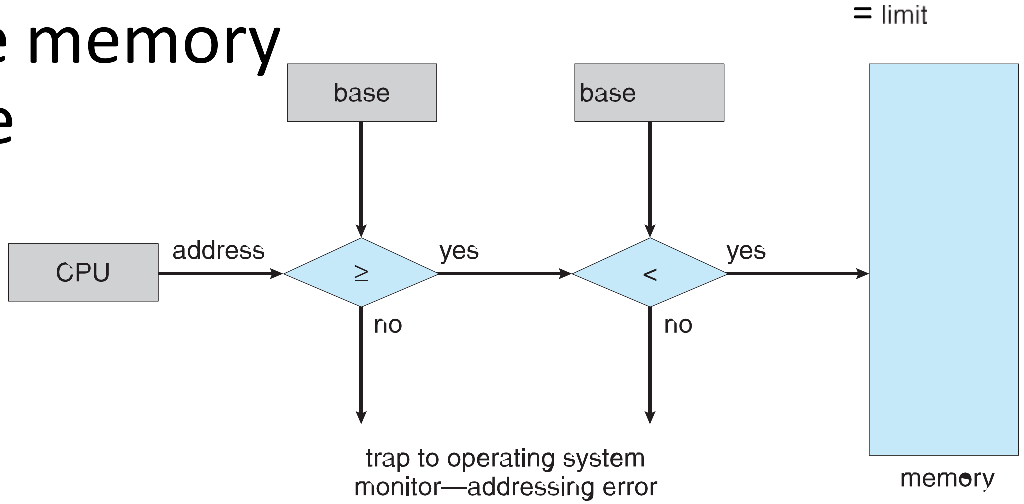
- Will have many programs in memory simultaneously
  - Program code loaded from storage
- The CPU can only access registers and main memory directly
  - Register access in a single cycle, but memory access takes many cycles
  - Multiple levels of cache attempt to hide main memory latency (L1, L2, L3)
- Memory unit sees only a stream of
  - Address plus read request
  - Address plus data plus write request
- Need to protect memory accesses to prevent malicious or just buggy user programs corrupting other programs, including the kernel

Hardware address protection

- **Base** and **limit** registers define the logical address space
  - Base is the smallest legal address, e.g., 300040
  - Limit is the size of the range, e.g., 120900
  - Thus program can access addresses in the range [300040, 420940)



- CPU must check every user-mode memory access to ensure it is in that range
  - Exception raised to OS if not



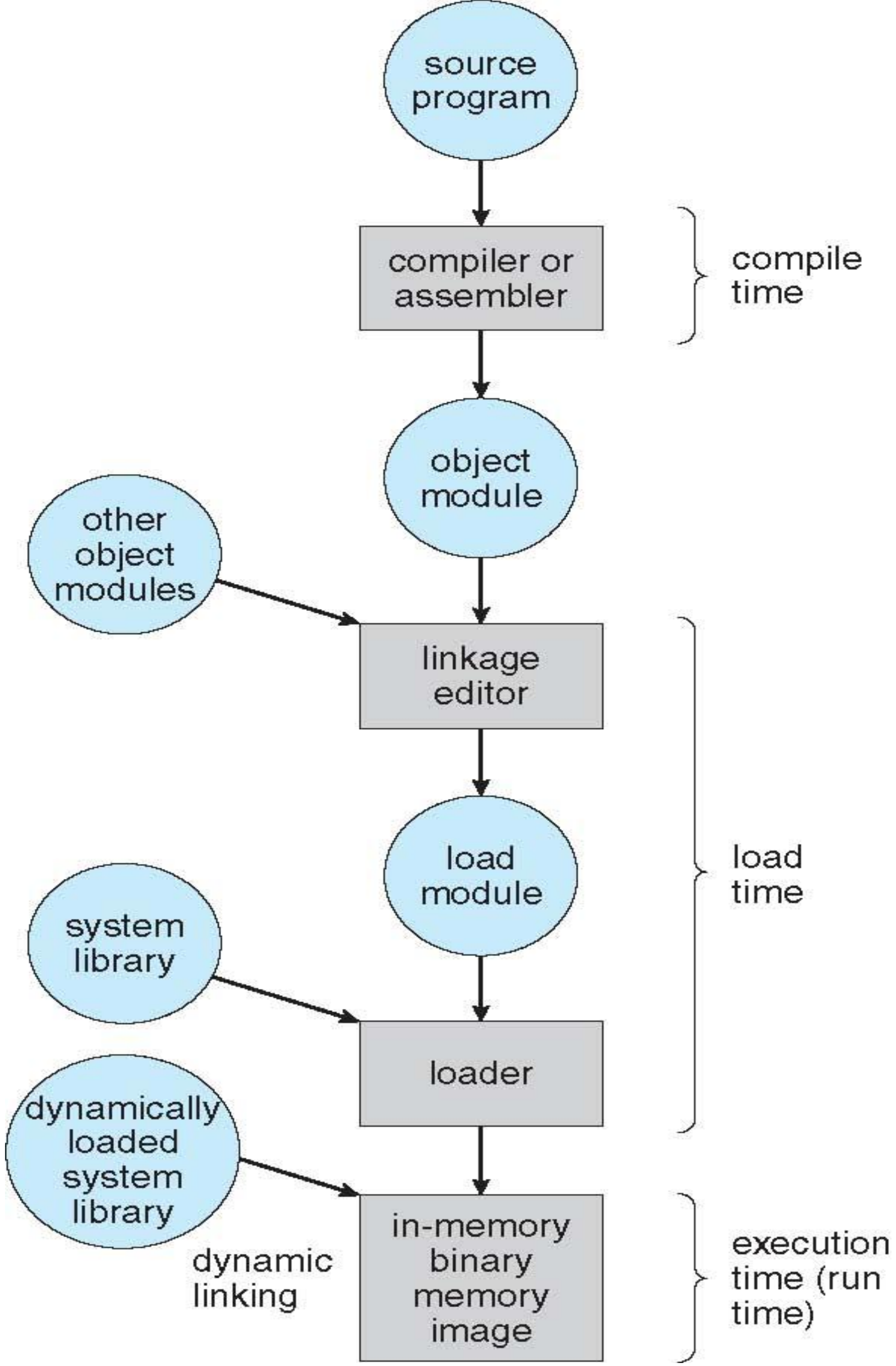
Address binding

```
int x, y;
x = 5;
y = x + 3;
```

```
str #5, [Rx]      ; store 5 into x
ldr R1, [Rx]      ; load value of x from memory
add R2, R1, #3    ; and add 3 to it
str R2, [Ry]      ; and store result in y
```

- Programs on disk are brought into memory to create running processes – but where in memory to put them given program code will refer to memory locations?
  - Consider a simple program and the assembly code it might generate
  - [Rx] means the contents of memory at address Rx
- Address binding happens at three different points
  - **Compile time**: If memory location known *a priori*, absolute code can be generated; requires recompilation if base location changes
  - **Load time**: Need to generate re-locatable 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
- Bindings map one address space to another – requires hardware support

Multistep Processing of a User Program

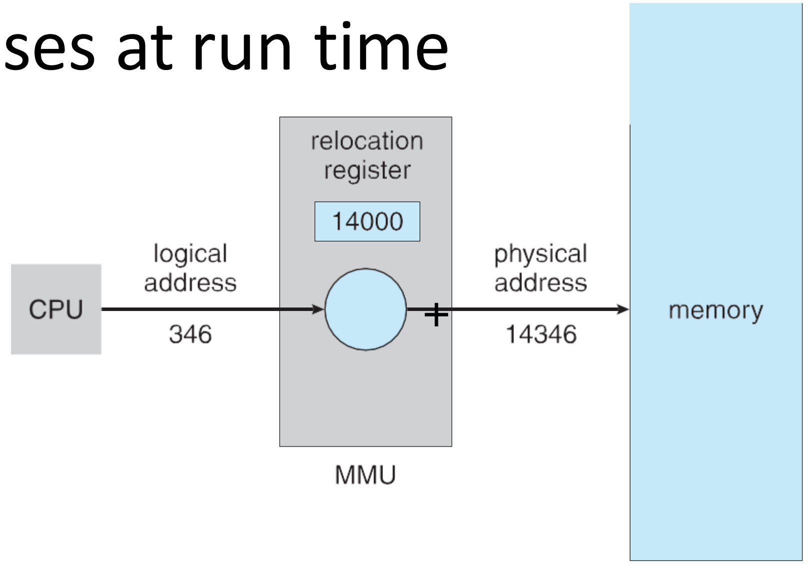


Logical vs physical addresses

- The concept of a logical address space that is bound to a separate physical address space is central to proper memory management
  - **Logical** (**virtual**) **address** – as generated by the CPU
  - **Physical address** – address seen by the memory unit
  - Identical in compile-time and load-time address-binding schemes
  - Differ in execution-time address-binding schemes
- The logical/physical address space is the set of all logical/physical addresses generated by a program
- Need hardware support to perform the mapping from logical to physical addresses at run time

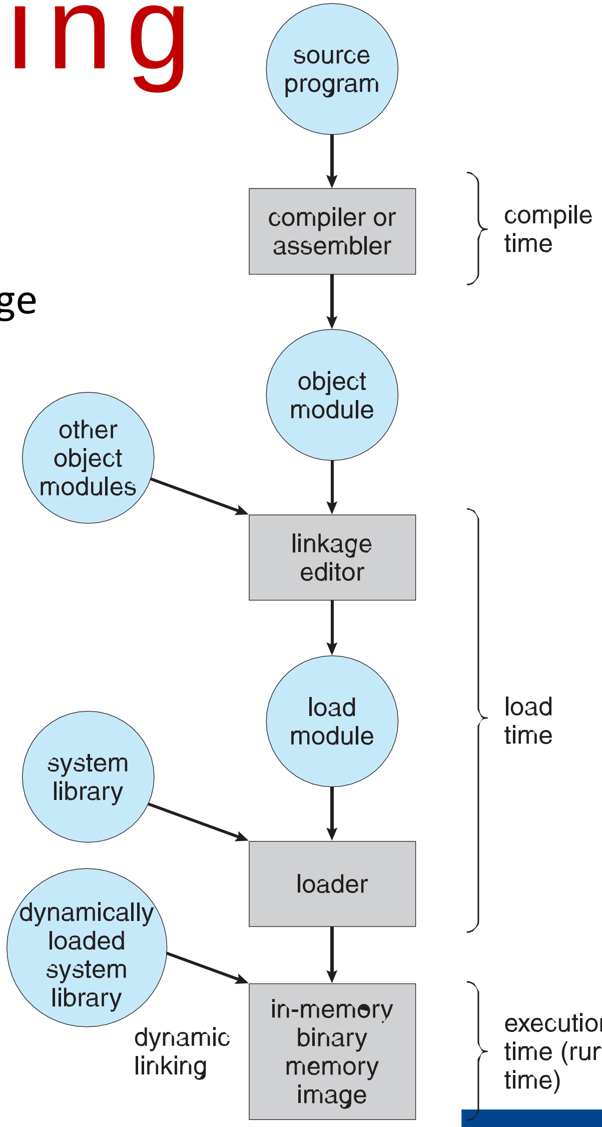
Memory Management Unit(MMU)

- Hardware that maps logical to physical addresses at run time
- Conceptually simple scheme: replace base register with **relocation register**
- Add the value in the relocation register to every address generated by a user process at the time it is sent to memory
  - User programs deal with logical addresses, never seeing physical addresses
- Execution-time binding occurs when reference is made to location in memory
  - Logical address is bound to physical address by the MMU



Dynamic linking and loading

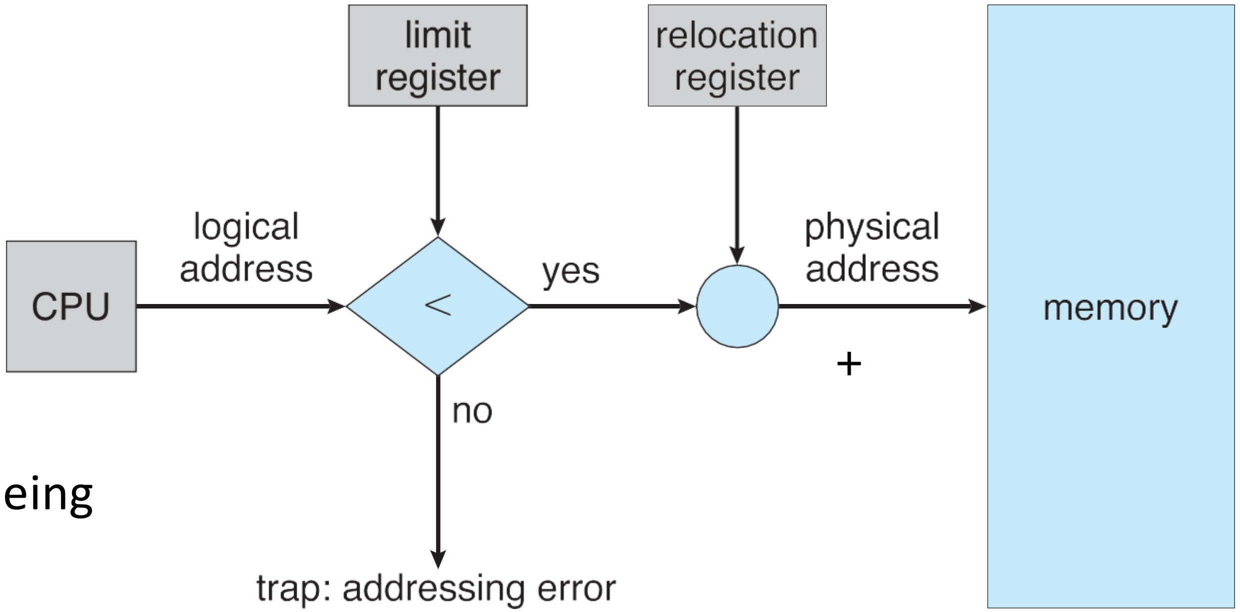
- Linking combines different object code modules to create a program
  - **Static linking** – all libraries and program code combined into the binary program image
  - **Dynamic linking** – postpone linking to execution time
- Dynamic linking is particularly useful for **system** or **shared** libraries
  - May need to track versions
- Calls replaced with a **stub**
  - A small piece of code to locate the appropriate in-memory routine
- Stub replaces itself with the address of the routine, and executes the routine
  - Operating system checks if routine is in processes’ memory address, adding it if not
- Dynamic loading avoids loading routines until they’re called
  - Better memory usage as unused routines are never loaded
  - Requires they be compiled with relocatable addresses
  - Useful when large amounts of code are needed infrequently
- OS can help by providing libraries to implement dynamic loading



Memory Allocation

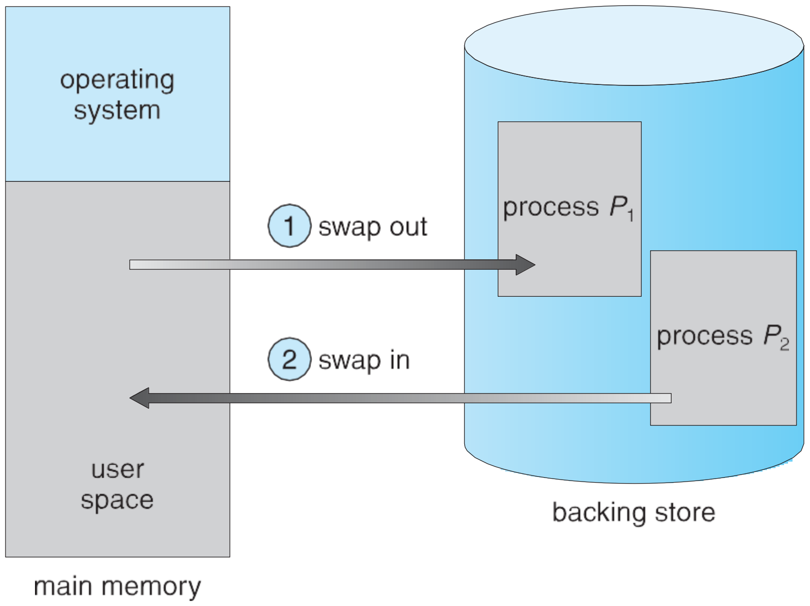
Memory allocation

- Main memory must support both kernel and user processes
  - Limited resource, must allocate efficiently
  - **Contiguous allocation** is early method putting each process in one chunk of memory
- How to determine chunks?
  - Multiple fixed-sized partitions limits the degree of multiprogramming; prefer **variable partitioning**
- Main memory usually partitioned into two
  - Resident kernel, usually held in low memory alongside interrupt vectors
  - User processes then held in high memory, each in a single contiguous section
- Relocation registers used to protect
  - User processes from each other, and
  - OS code and data from being modified
- Can then allow actions such as kernel code being transient and kernel changing size



Swapping

- When physical memory requested exceeds physical memory in machine, temporarily swap processes out
  - Move processes from main memory to storage
- Significant performance impact
  - Time to transfer process to/from storage directly proportional to the amount of memory swapped
  - Context switches can thus become very expensive
  - E.g., 100MB process with storage transfer rate of 50MB/s
- Swapping default disabled
  - Enabled only while allocated memory exceeds threshold
  - Plus consider pending I/O to or from process memory space
  - System maintains a ready queue of ready-to-run processes with memory images on disk
- Must swapped out processes be swapped into the same physical addresses?
  - Depends on address binding method



Multiple variable-partition allocation

- **Holes**, blocks of available memory 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 combined
- OS maintains information about:
  - allocated partitions and
  - free partitions (holes)

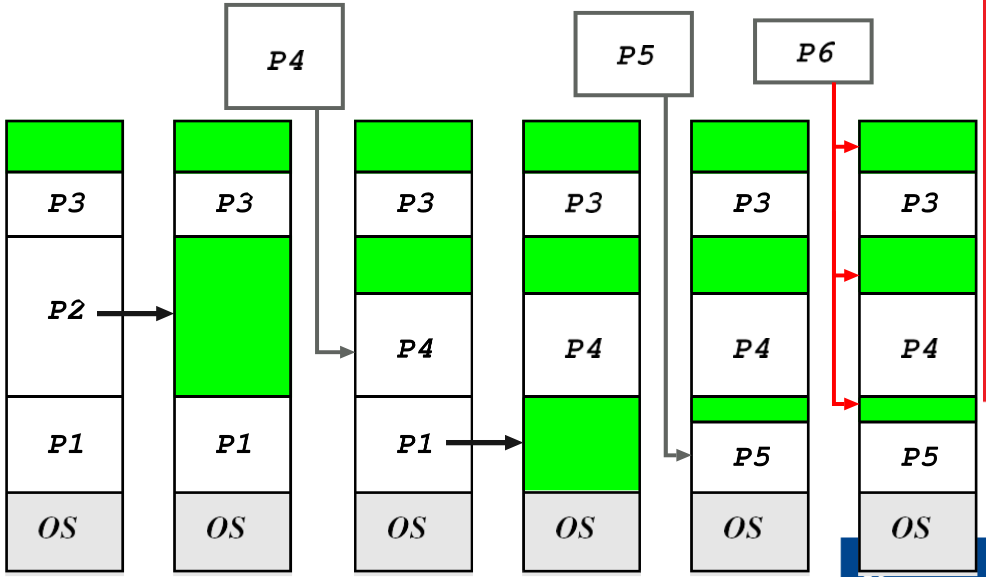
Dynamic allocation problem

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

- **First-fit**, allocate the first hole that is big enough
- **Best-fit**, allocate the smallest hole that is big enough
  - Requires searching entire list, unless maintained ordered by size
  - Produces the smallest leftover hole
- **Worst-fit**, allocate the largest hole
  - Also requires searching entire list, producing the largest leftover hole
- First-fit and best-fit better than worst-fit in terms of speed and storage utilization

Fragmentation

- Fragmentation results in memory being unused and unusable
- **External Fragmentation**
  - Occurs when free memory exists to satisfy a request but it is not contiguous
  - Can eventually result in blocking as insufficient contiguous memory to swap any process in
- **Internal Fragmentation**
  - Occurs when allocated memory is slightly larger than requested memory
  - Memory internal to a partition, but unused
- Analysis of first-fit indicates that for *N* blocks allocated, 0.5 *N* blocks lost to fragmentation



Example 1

- Given memory partitions of 200K, 150K, 600K, 350K, 300K and 250K (in order), how would each of the First-fit, Best-fit, and Worst-fit algorithms place processes of p1=300K, p2=25K, p3=125K and p4=50K (in order)? What block will be assigned to each process if the first fit, best fit and worst fit?

Solution

First-Fit Algorithm

Memory Block	Process No	Process Size	Status	Internal Fragmentation
200K	p2	25K	Allocated	175K
150K	p3	125K	Allocated	25K
600K	p1	300K	Allocated	300K
350K	p4	50K	Allocated	300K
300K	-	-	Not Allocated	0K
250K	-	-	Not Allocated	0K

Best-Fit Algorithm

Memory Block	Process No	Process Size	Status	Internal Fragmentation
200K	p4	50K	Allocated	150K
150K	p2	25K	Allocated	125K
600K	p3	125K	Allocated	475K
350K	p1	300K	Allocated	50K
300K	-	-	Not Allocated	0K
250K	-	-	Not Allocated	0K

Worst-Fit Algorithm

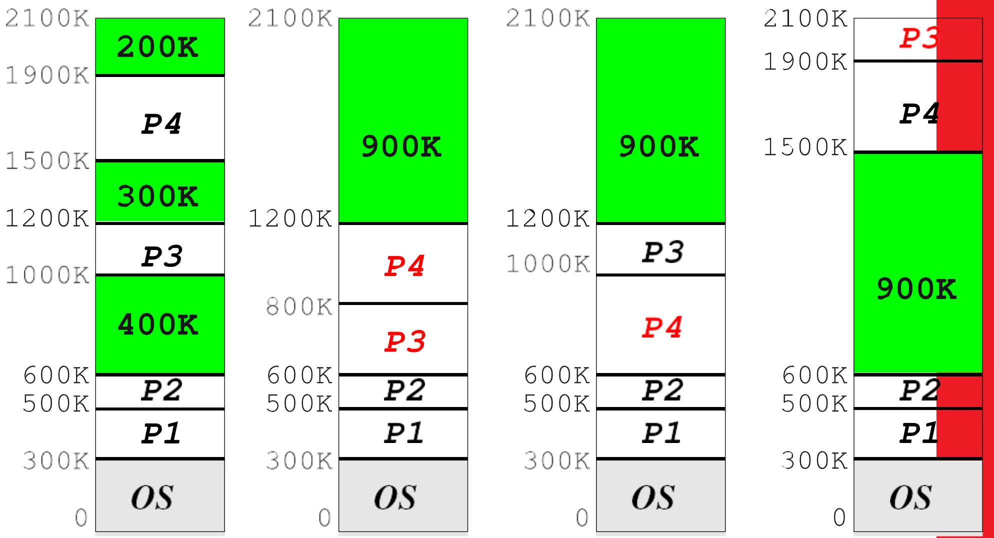
Memory Block	Process No	Process Size	Status	Internal Fragmentation
200K	p4	50K	Allocated	150K
150K	p3	125K	Allocated	25K
600K	p1	300K	Allocated	300K
350K	p2	25K	Allocated	325K
300K	-	-	Not Allocated	0K
250K	-	-	Not Allocated	0K

Solution - Fragmentation

- **First Fit - Internal Fragmentation**: (300K from 3rd partition) + (175K from 1st partition) + (25K from 2nd partition) + (300K from 4th partition) = 800K
- **Best Fit - Internal Fragmentation**: (150K from 1st partition) = 150K.
- **Worst Fit - Internal Fragmentation**: (275K from 3rd partition) + (75K from 1st partition) + (300K from 4th partition) = 650K.
- **External Fragmentation**: None, as all processes have been allocated memory in ALL the algorithm.

Compaction

- Reduce external fragmentation by **compaction**
  - Shuffle memory contents to place all free memory together in one large block
- Compaction is possible only if
  - relocation is dynamic, and
  - done at execution time
- I/O problem
  - Pin job in memory while involved in I/O
  - Do I/O only into OS buffers
- Now consider that backing store has same fragmentation problems



Summary

- Memory protection
  - Address binding
  - Logical and physical addresses
  - Memory Management Unit (MMU)
  - Linking and loading
- Memory allocation
  - Swapping
  - Dynamic allocation
  - Fragmentation
  - Compaction

Question ?