CSE2005 - OPERATING SYSTEMS

Module 5 –L1

Memory Management

Dr. Rishikeshan C A

VIT Chennai

Memory management

- In a multiprogramming system, in order to share the processor, a number of processes must be kept in memory.
- Memory management is achieved through memory management algorithms.
- Each memory management algorithm requires its own hardware support.
- In this Module, we shall see the partitioning, paging and segmentation methods.

Memory Management

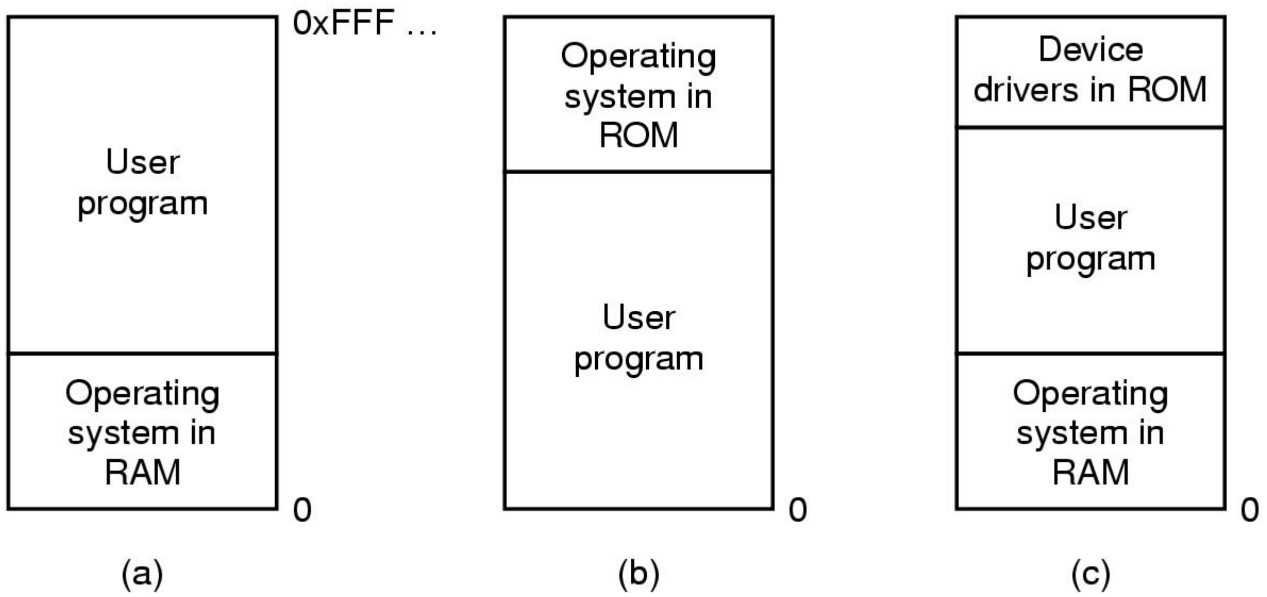
- Is the task carried out by the OS and hardware to accommodate multiple processes in main memory
- If only a few processes can be kept in main memory, then much of the time all processes will be waiting for I/O and the CPU will be idle
- Hence, memory needs to be allocated efficiently in order to pack as many processes into memory as possible
- In most schemes, the kernel occupies some fixed portion of main memory and the rest is shared by multiple processes

Memory Management Techniques

- Single Contiguous Memory Management
- Partitioned Memory Management
- Relocation Partitioned Memory Management
- Paged Memory Management
- Segmented Memory Management
- Demand–Paged Memory Management
  - Page Replacement Algorithms
- Overlay Memory Management

Basic Memory Management

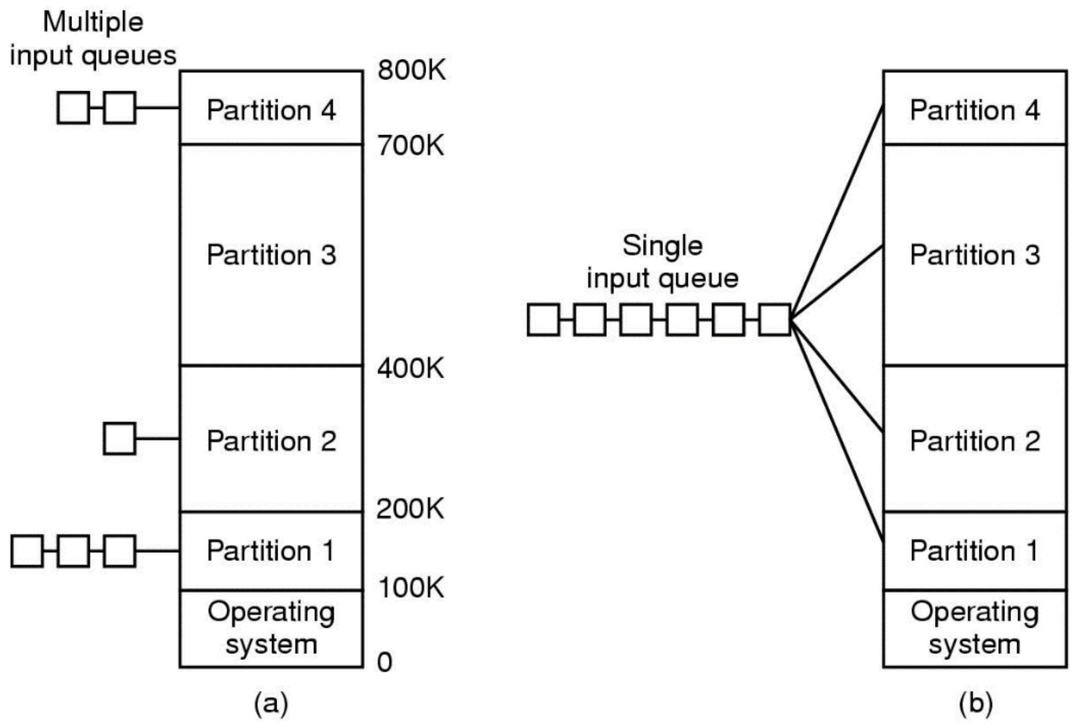
Monoprogramming without Swapping or Paging



Three simple ways of organizing memory

- an operating system with one user process

Multiprogramming with Fixed Partitions

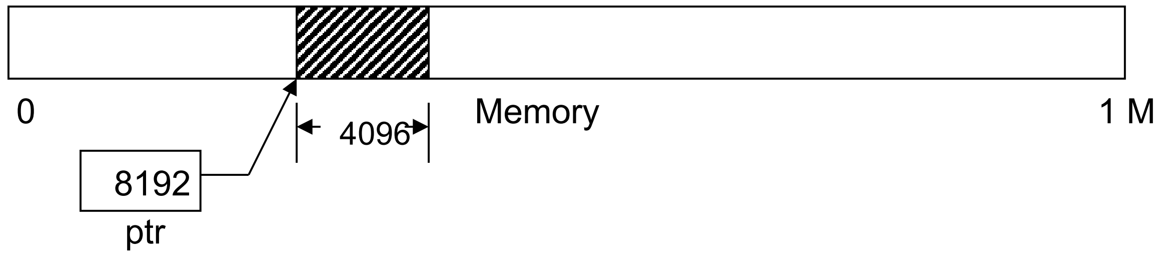


- Fixed memory partitions
  - separate input queues for each partition
  - single input queue

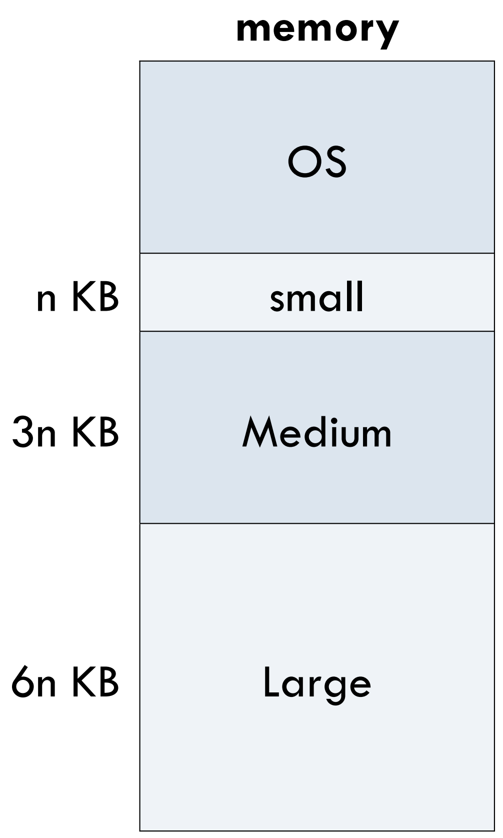
Allocating Memory

- Example of allocation

```
char* ptr = malloc(4096);          // char* is address of a single byte
```

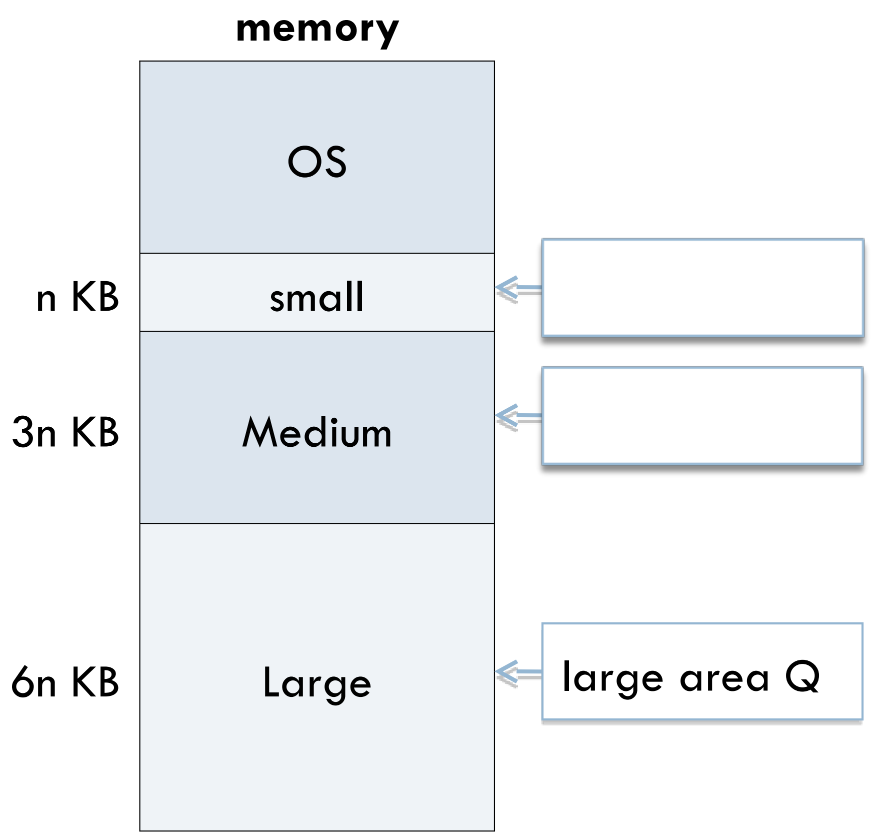


Fixed Partitioning



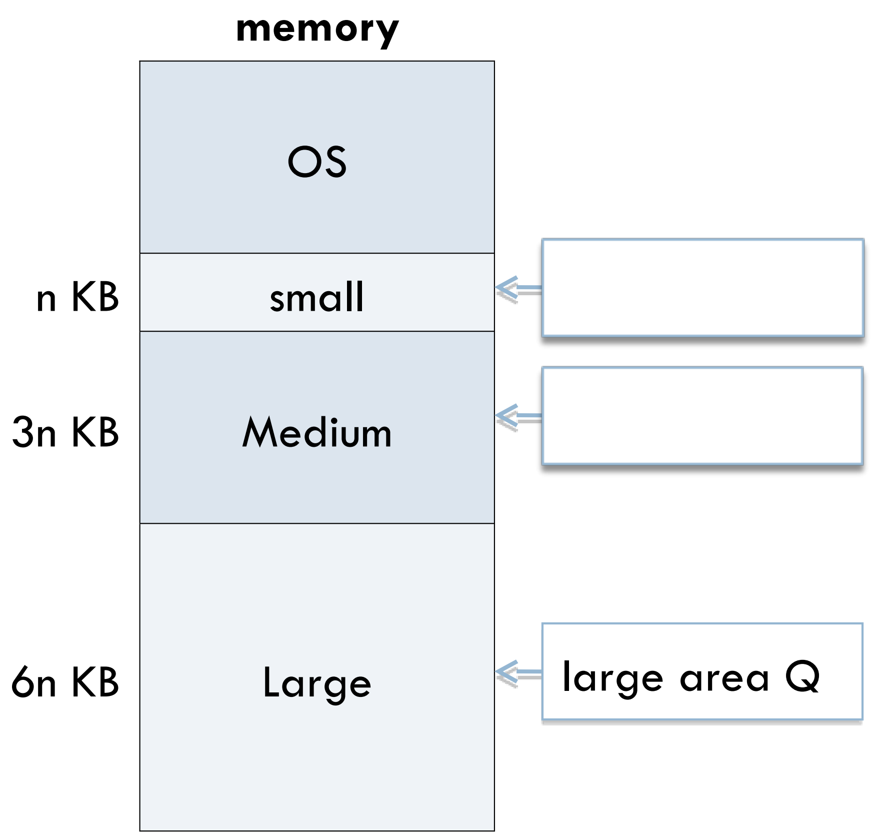
- In this method, memory is divided into partitions whose sizes are fixed.
- OS is placed into the lowest bytes of memory.
- Relocation of processes is not needed

Fixed Partitioning



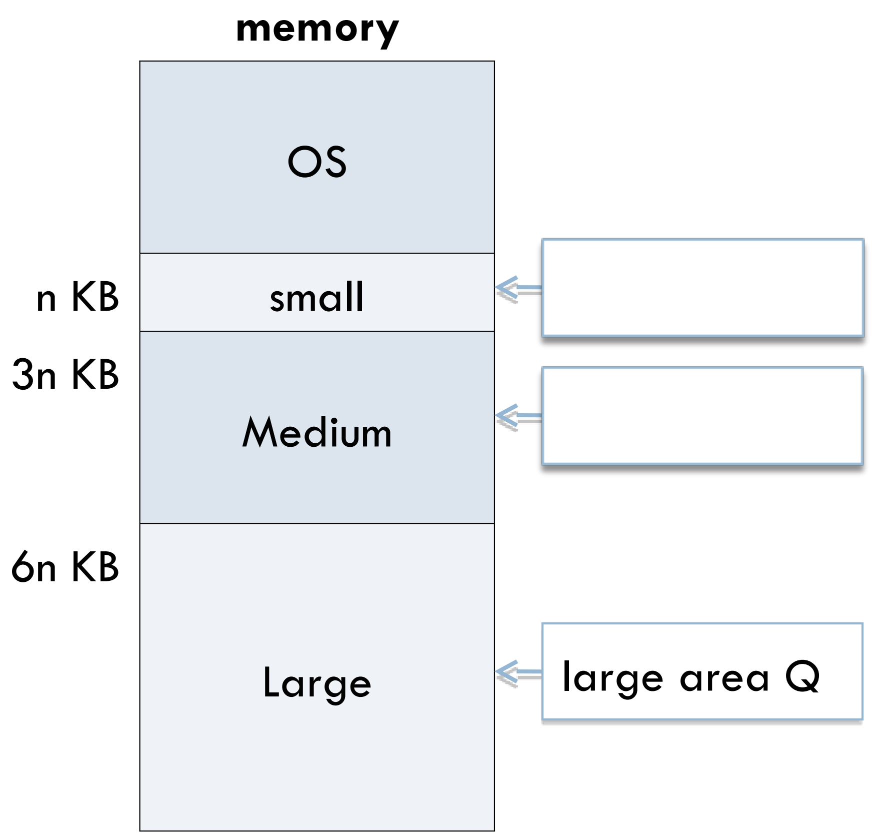
- Processes are classified on entry to the system according to their memory they requirements.
- We need one *Process Queue (PQ)* for each class of process.

Fixed Partitioning



- If a process is selected to allocate memory, then it goes into memory and competes for the processor.
- The number of fixed partition gives the degree of multiprogramming.
- Since each queue has its own memory region, there is no competition between queues for the memory.

Fixed Partitioning



- The main problem with the fixed partitioning method is how to determine the number of partitions, and how to determine their sizes.

3

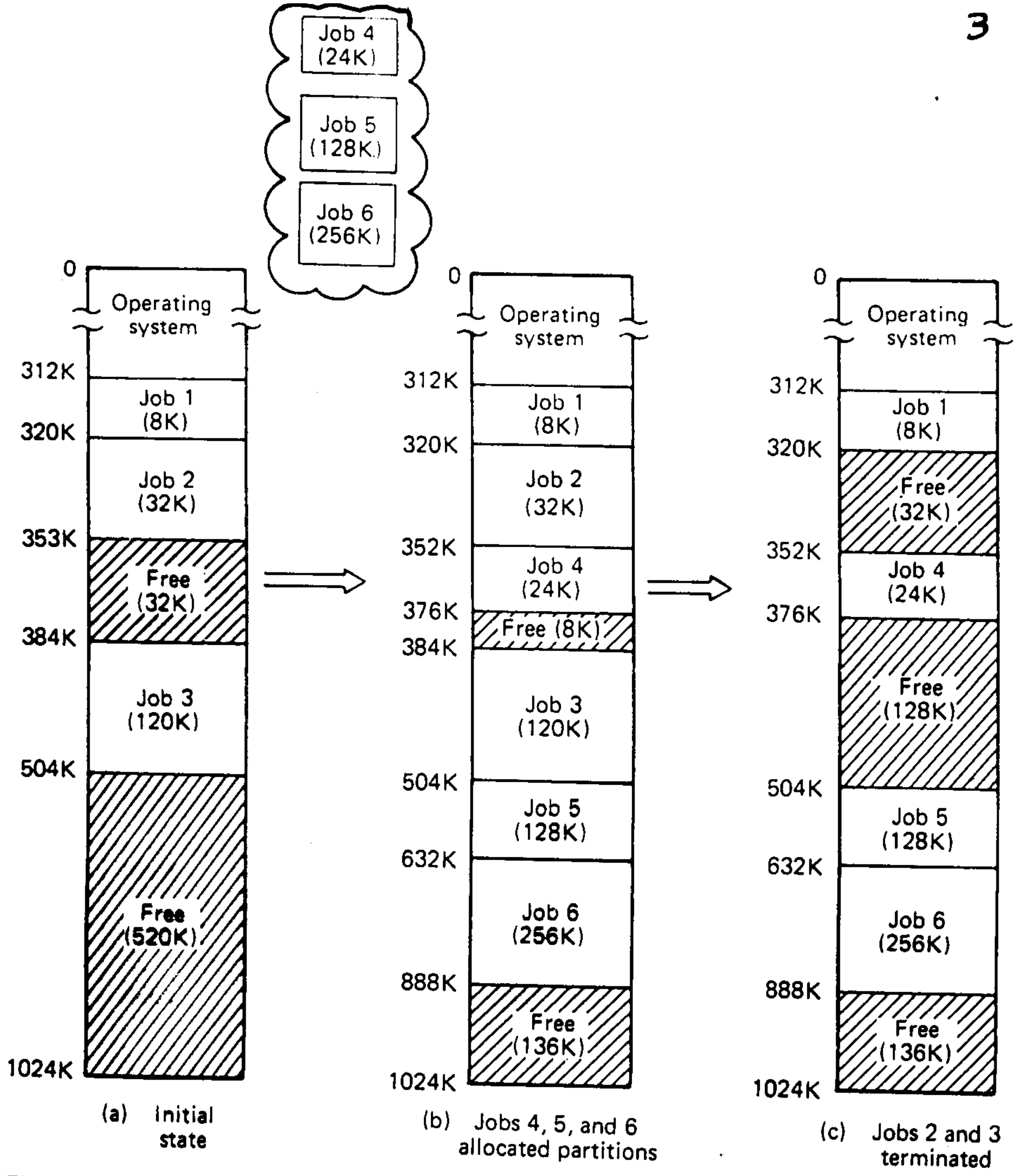
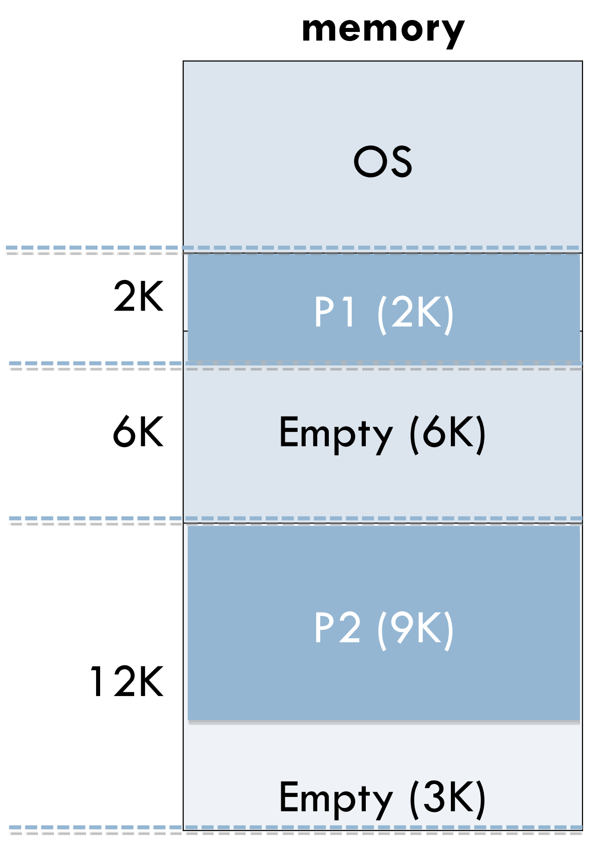


Figure Partition allocation and deallocation

Fragmentation

- Segments of memory can become unusable
  - FRAGMENTATION
  - result of allocation scheme
- Two types of fragmentation
  - external fragmentation
    - memory remains unallocated
    - variable allocation sizes
  - internal fragmentation
    - memory is allocated but unused
    - fixed allocation sizes

fragmentation

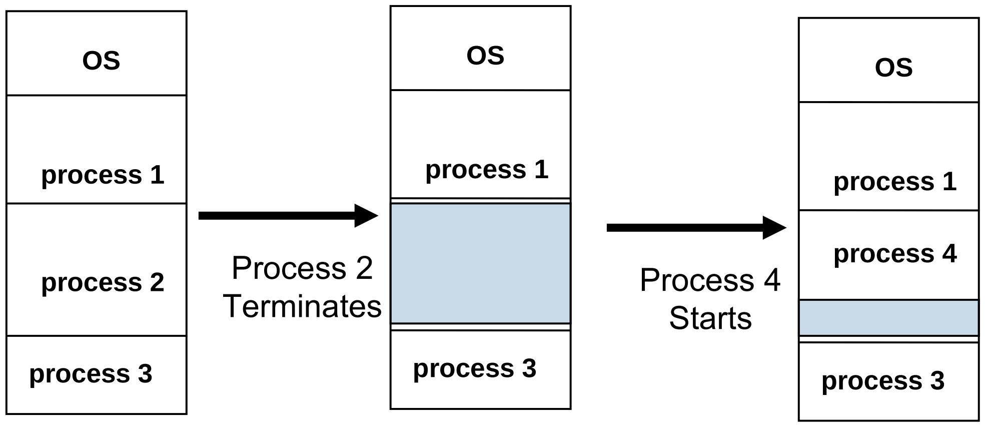


If a whole partition is currently not being used, then it is called *an* ***external fragmentation.***

If a partition is being used by a process requiring some memory smaller than the partition size, then it is called an ***internal fragmentation.***

Variable Partitions

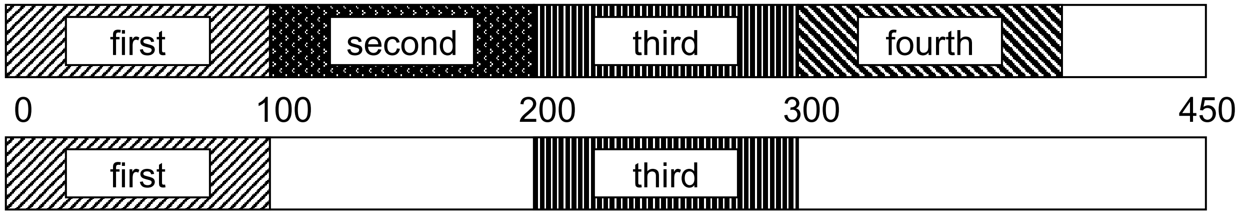
- More complex management problem
  - Must track free and used memory
  - Need data structures to do tracking
  - What holes are used for a process?
- *External fragmentation*
  - memory that is <u>outside any partition</u> and is too small to be usable by any process



External Fragmentation

- Imagine calling a series of *malloc* and *free*

```
char* first = malloc(100);
char* second = malloc(100);
char* third = malloc(100);
char* fourth = malloc(100);
free(second);
free(fourth);
char* problem = malloc(200);
```

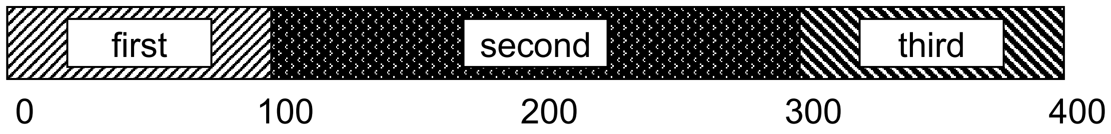


- 250 free bytes of memory, only 150 contiguous
  - unable to satisfy final malloc request

Internal Fragmentation

- Imagine calling a series of *malloc*
  - assume allocation unit is 100 bytes

```
char* first = malloc(90);
char* second = malloc(120);
char* third = malloc(10);
char* problem = malloc(50);
```



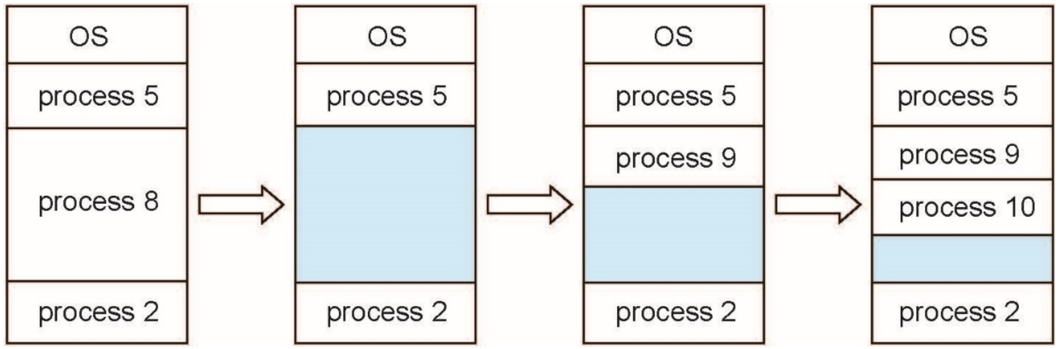
- All of memory has been allocated but only a fraction of it is used (220 bytes)
  - unable to handle final memory request

Internal vs. External Fragmentation

- Externally fragmented memory can be compacted
  - lots of issues with compaction
  - more on this later
- Fixed size allocation may lead to internal fragmentation, but less overhead
  - example
    - 8192 byte area of free memory
    - request for 8190 bytes
    - if exact size allocated - 2 bytes left to keep track of
    - if fixed size of 8192 used - 0 bytes to keep track of

Multiple-partition allocation

- Multiple-partition allocation
  - Degree of multiprogramming limited by number of partitions
  - **Variable-partition** sizes for efficiency (sized to a given process' needs)
  - **Hole** – block of available memory; holes 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
  - Operating system maintains information about:
    a) allocated partitions b) free partitions (hole)



Variable Partitioning

- With fixed partitions we have to deal with the problem of determining the number and sizes of partitions to minimize internal and external fragmentation.
- If we use variable partitioning instead, then partition sizes may vary dynamically.
- In the variable partitioning method, we keep a table (linked list) indicating used/free areas in memory.

Variable Partitioning

- Initially, the whole memory is free and it is considered as one large block.
- When a new process arrives, the OS searches for a block of free memory large enough for that process.
- We keep the rest available (free) for the future processes.
- If a block becomes free, then the OS tries to merge it with its neighbors if they are also free.

Variable Partitioning

- There are Three algorithms for searching the list of free blocks for a specific amount of memory.
  - First Fit
    - Next Fit
  - Best Fit
  - Worst Fit

Dynamic Storage-Allocation Problem

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

- **First-fit**: Allocate the ***first*** hole that is big enough
- **Next Fit:** Thus each time a request is made the pointer begins searching from the place it last finished. This helps in, to avoid the usage of memory always from the head (beginning) of the free block chain.
- **Best-fit**: Allocate the ***smallest*** hole that is big enough; must search entire list, unless ordered by size
  - Produces the smallest leftover hole
- **Worst-fit**: Allocate the ***largest*** hole; must also search entire list
  - Produces the largest leftover hole

First-fit and best-fit better than worst-fit in terms of speed and storage utilization

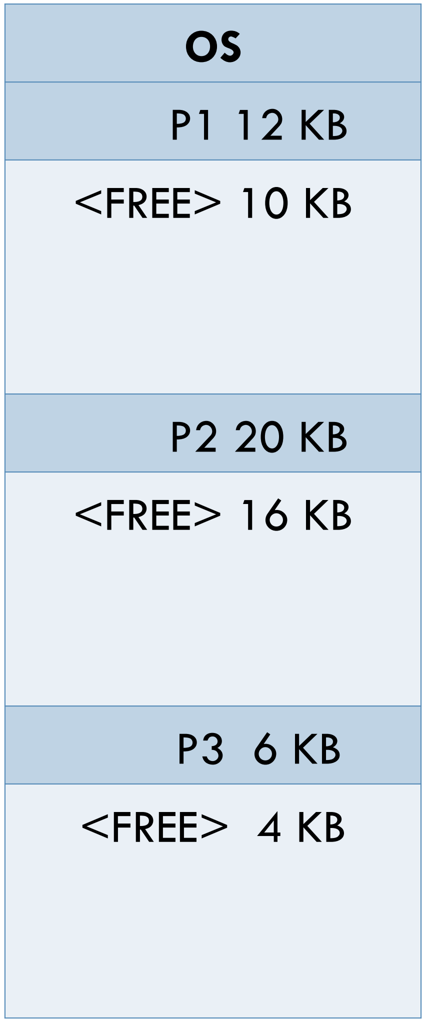
First fit

- **First Fit :** Allocate the first free block that is large enough for the new process.
- This is a fast algorithm.

- Next-fit: variation: first hole from *last placement* that fits

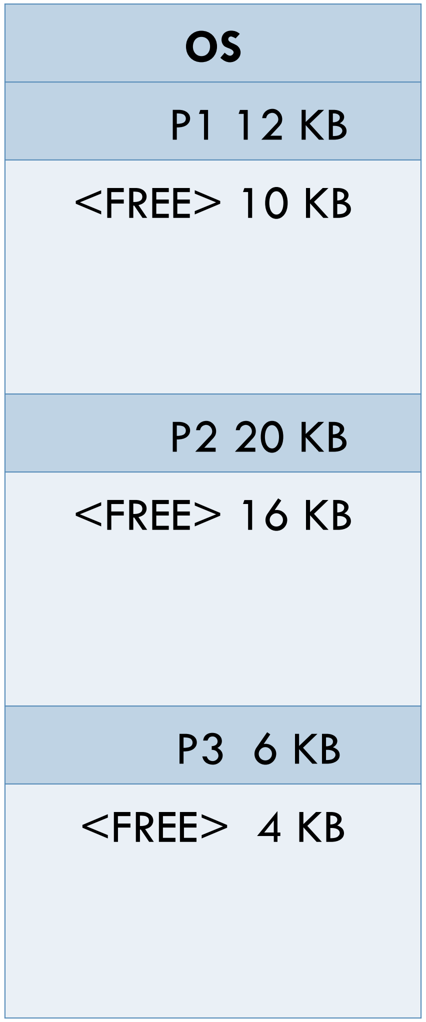
First fit

Initial memory mapping

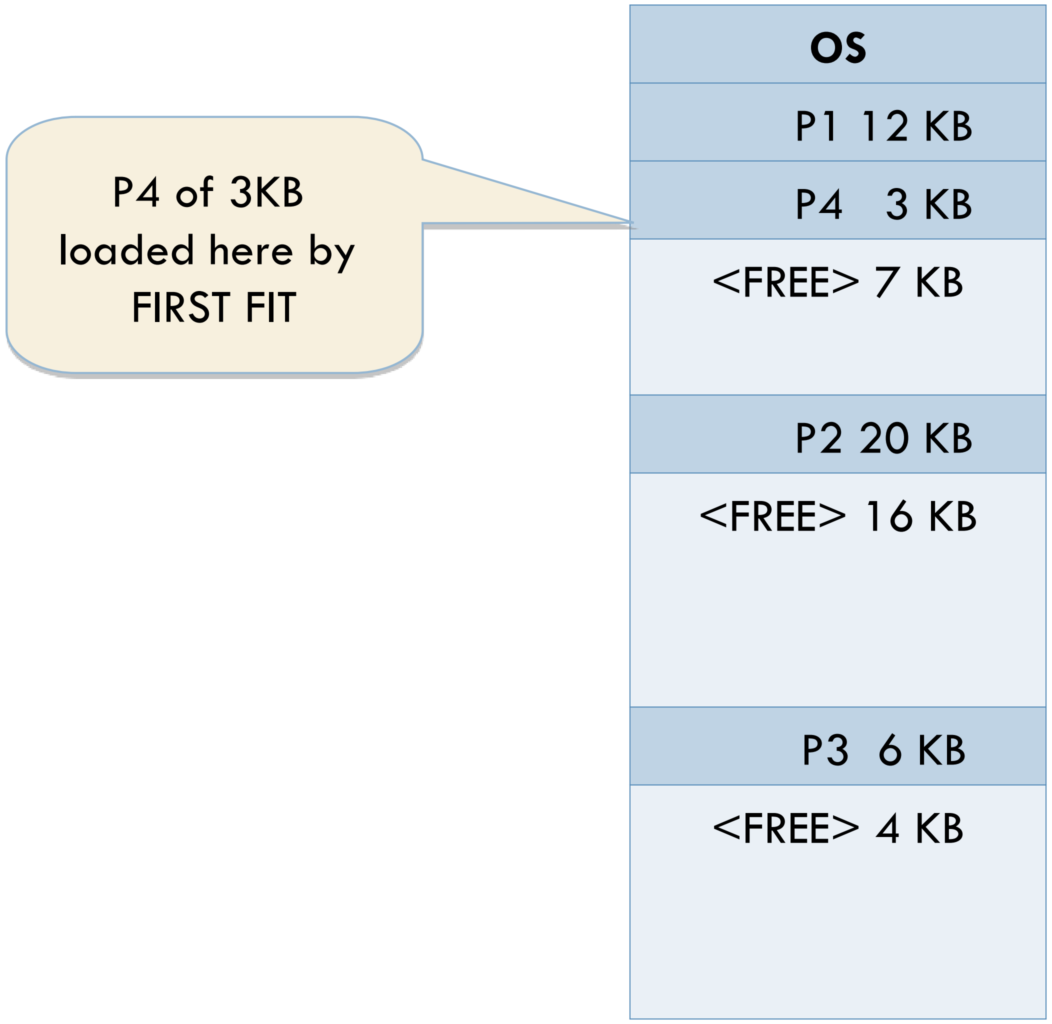


First fit

P4 of 3KB arrives

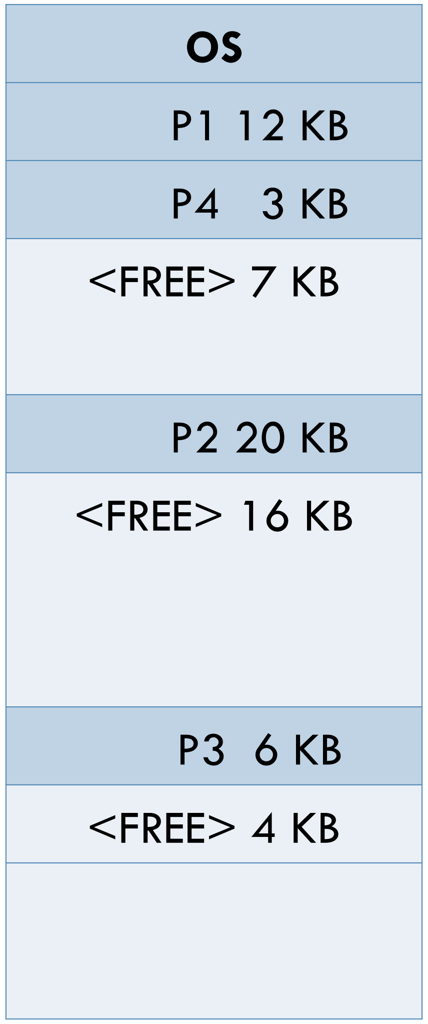


First fit

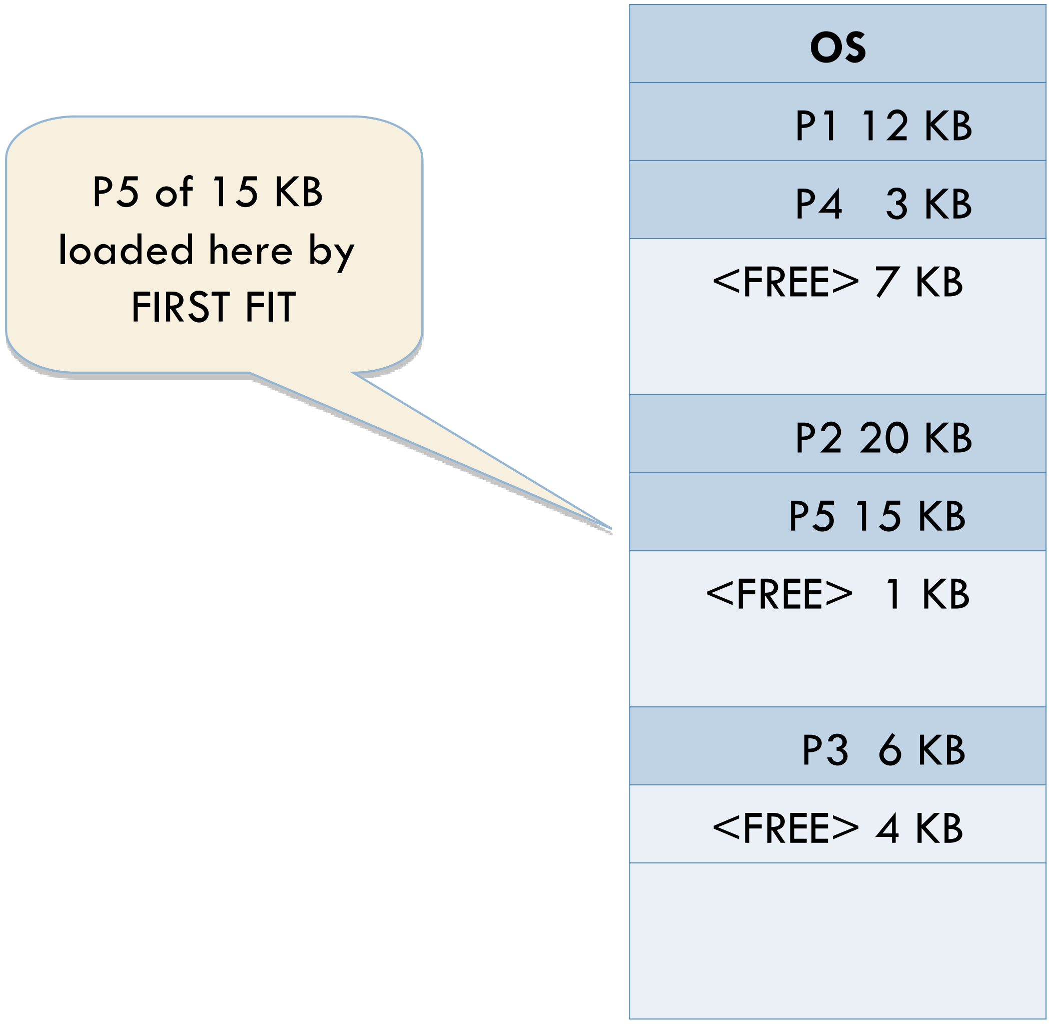

OS
P1 12 KB
P4 3 KB
<FREE> 7 KB
P2 20 KB
<FREE> 16 KB
P3 6 KB
<FREE> 4 KB
P4 of 3KB loaded here by FIRST FIT

First fit

P5 of 15KB arrives



first fit

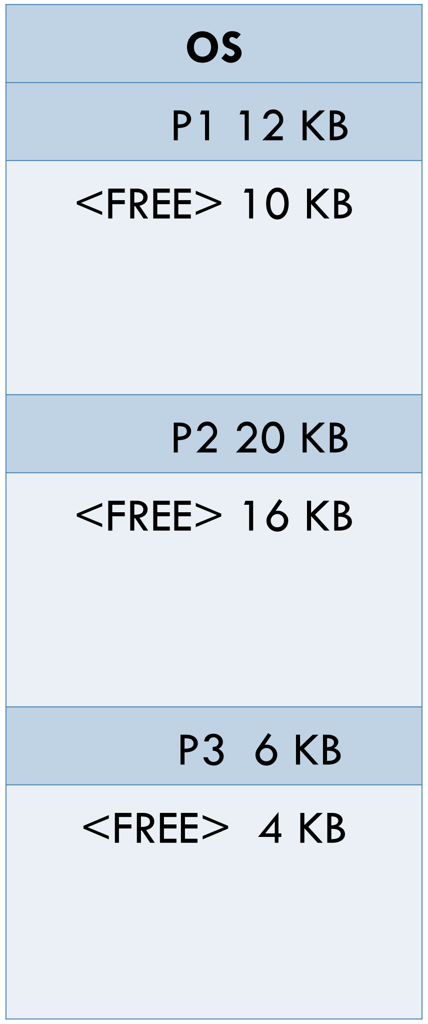

P5 of 15 KB loaded here by FIRST FIT
OS
P1 12 KB
P4 3 KB
<FREE> 7 KB
P2 20 KB
P5 15 KB
<FREE> 1 KB
P3 6 KB
<FREE> 4 KB

Best fit

- Best Fit : Allocate the smallest block among those that are large enough for the new process.
- In this method, the OS has to search the entire list, or it can keep it sorted and stop when it hits an entry which has a size larger than the size of new process.
- This algorithm produces the smallest left over block.
- However, it requires more time for searching all the list or sorting it
- If sorting is used, merging the area released when a process terminates to neighboring free blocks, becomes complicated.

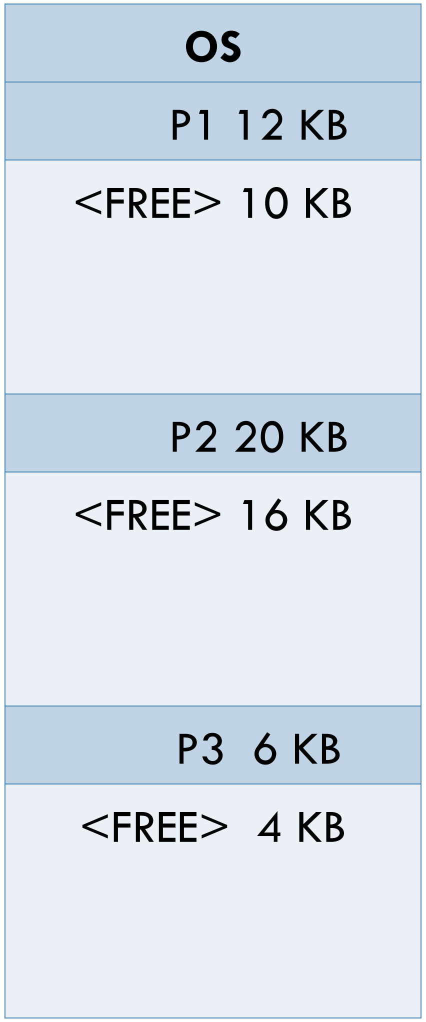
Best fit

Initial memory mapping

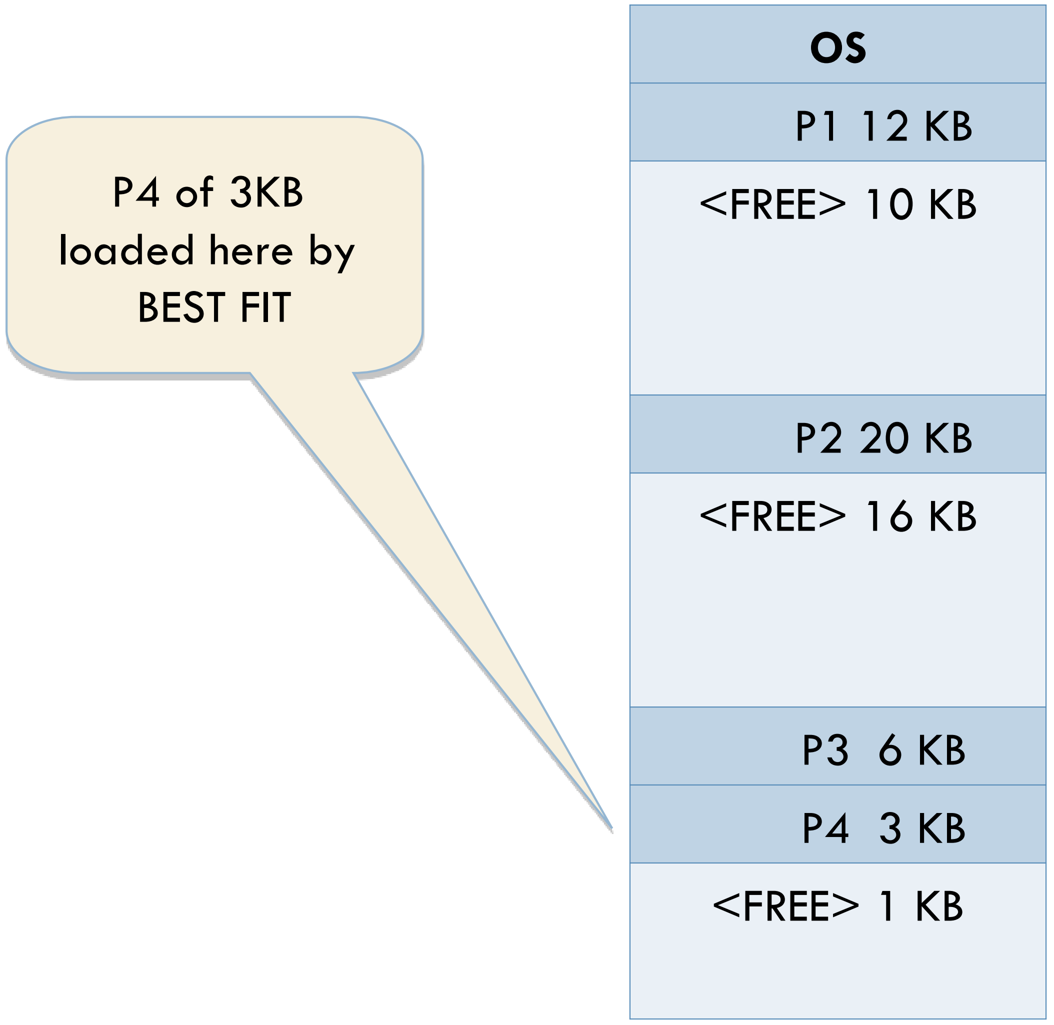


Best fit

P4 of 3KB arrives

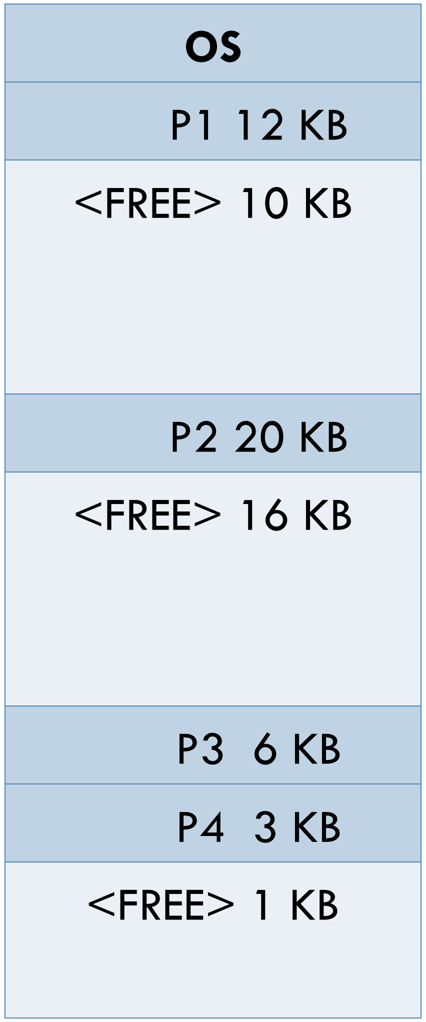


Best fit

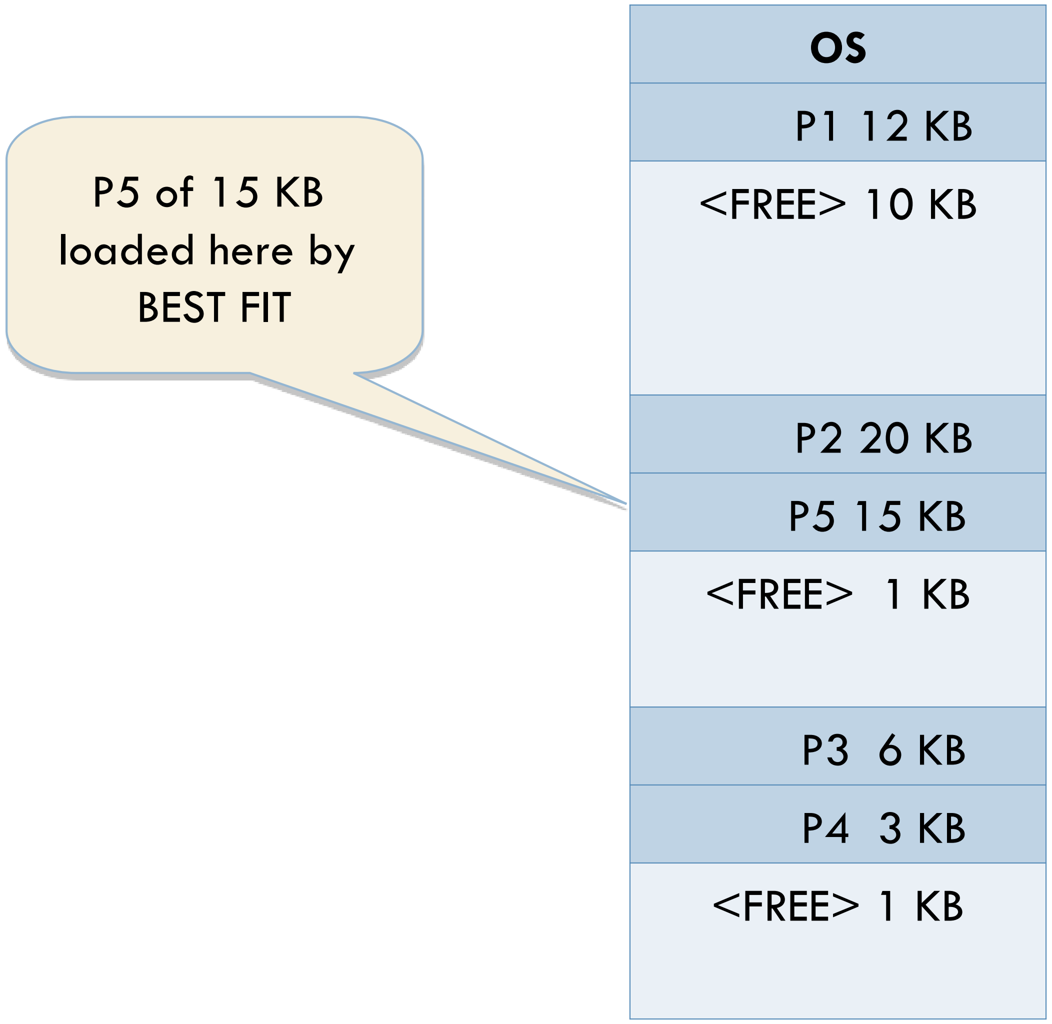

P4 of 3KB loaded here by BEST FIT
OS
P1 12 KB
<FREE> 10 KB
P2 20 KB
<FREE> 16 KB
P3 6 KB
P4 3 KB
<FREE> 1 KB

Best fit

P5 of 15KB arrives



Best fit

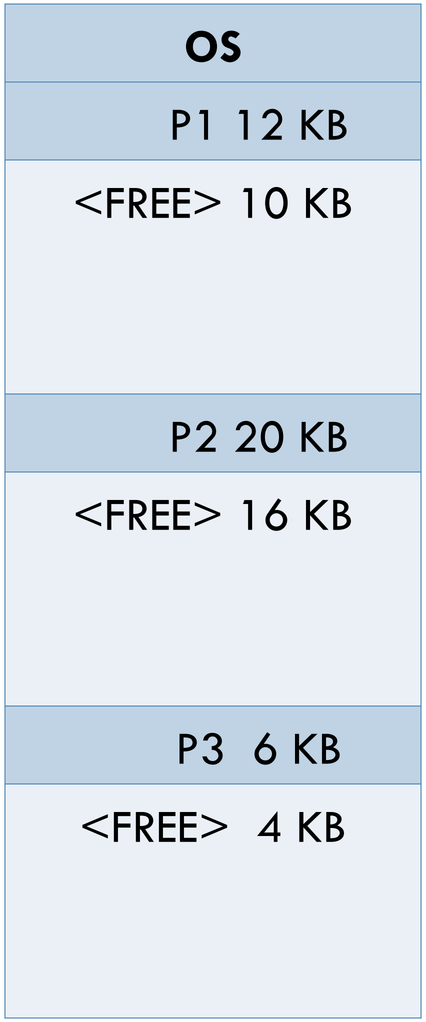

P5 of 15 KB loaded here by BEST FIT
OS
P1 12 KB
<FREE> 10 KB
P2 20 KB
P5 15 KB
<FREE> 1 KB
P3 6 KB
P4 3 KB
<FREE> 1 KB

Worst fit

- <u>Worst Fit :</u> Allocate the largest block among those that are large enough for the new process.
- Again a search of the entire list or sorting it is needed.
- This algorithm produces the largest over block.

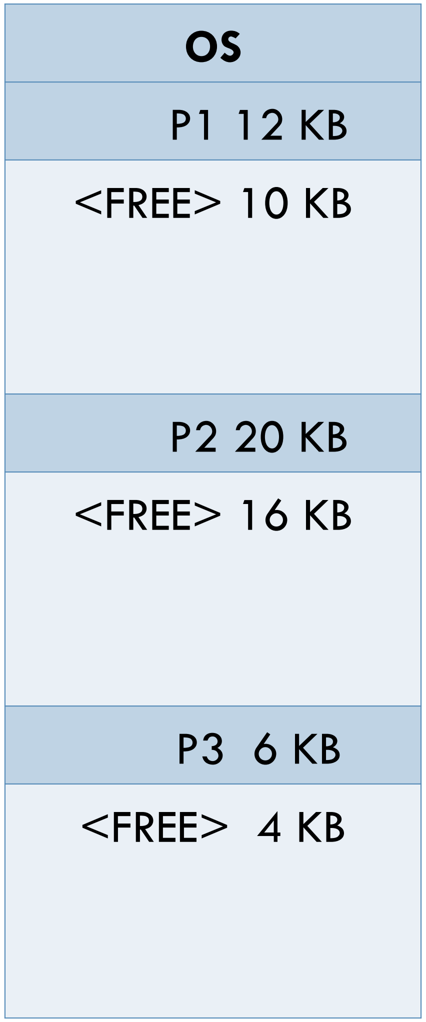
worst fit

Initial memory mapping

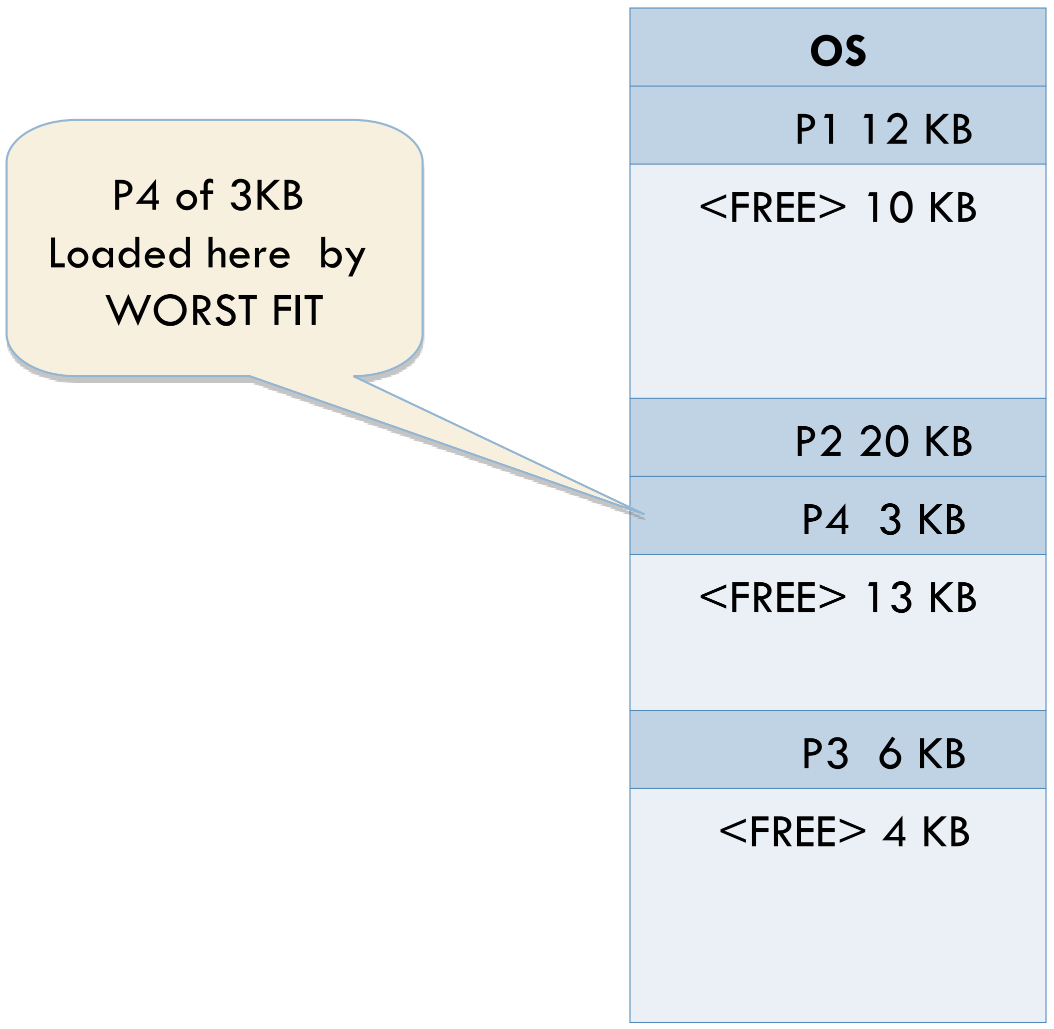


Worst fit

P4 of 3KB arrives

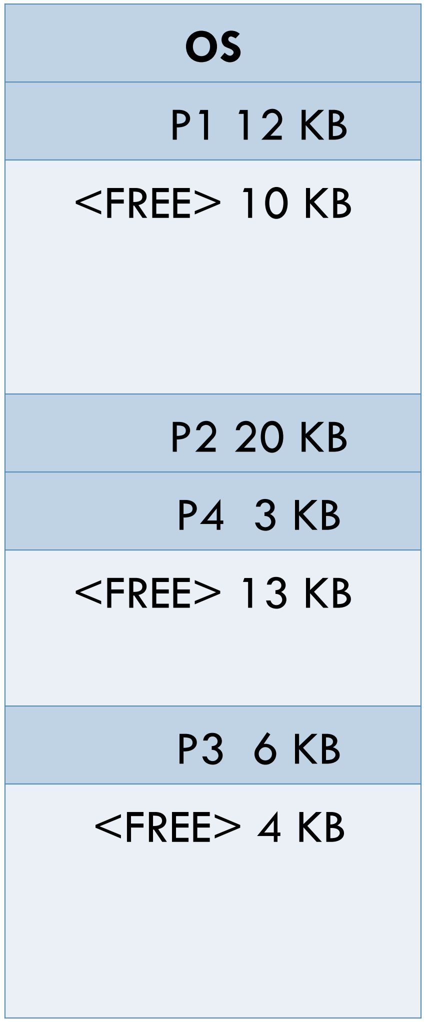


Worst fit


OS
P1 12 KB
<FREE> 10 KB
P2 20 KB
P4 3 KB
<FREE> 13 KB
P3 6 KB
<FREE> 4 KB
P4 of 3KB Loaded here by WORST FIT

Worst fit

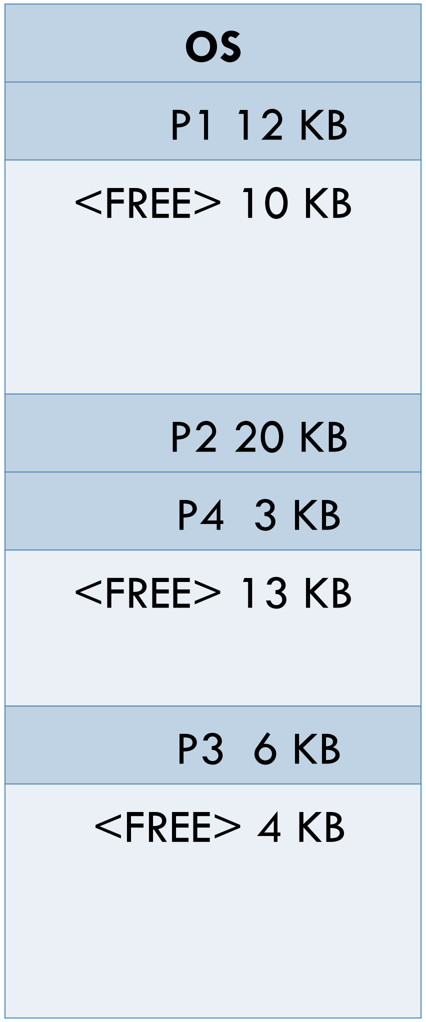
No place to load P5 of 15K



Worst fit

No place to load P5 of 15K

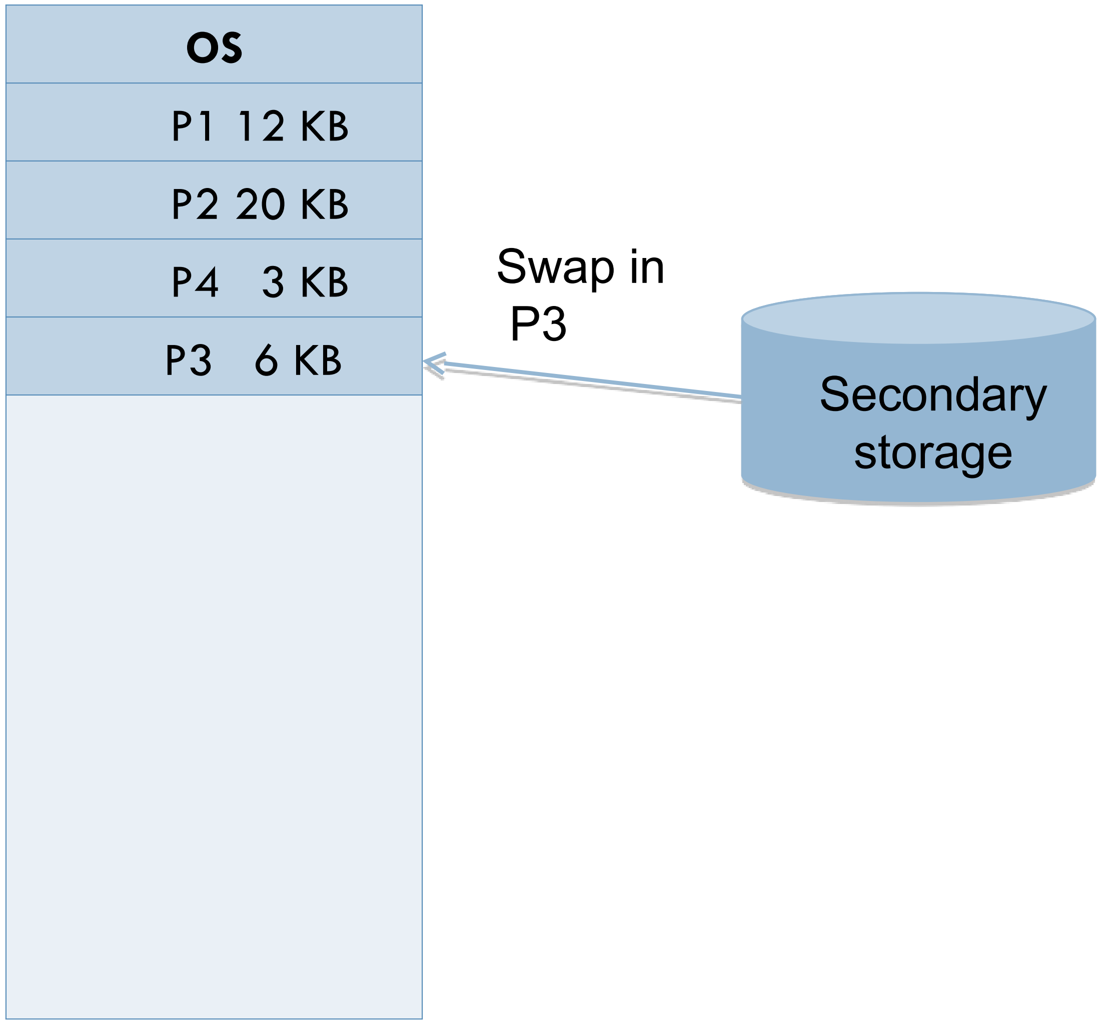
Compaction is needed !!



Compaction

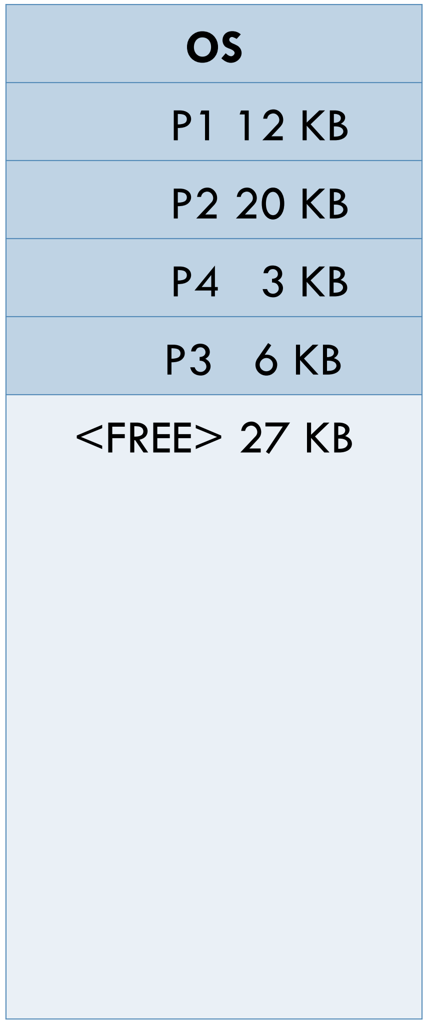
- Compaction is a method to overcome the external fragmentation problem.
- All free blocks are brought together as one large block of free space.
- Compaction requires dynamic relocation.
- Certainly, compaction has a cost and selection of an optimal compaction strategy is difficult.
- One method for compaction is swapping out those processes that are to be moved within the memory, and swapping them into different memory locations

Compaction



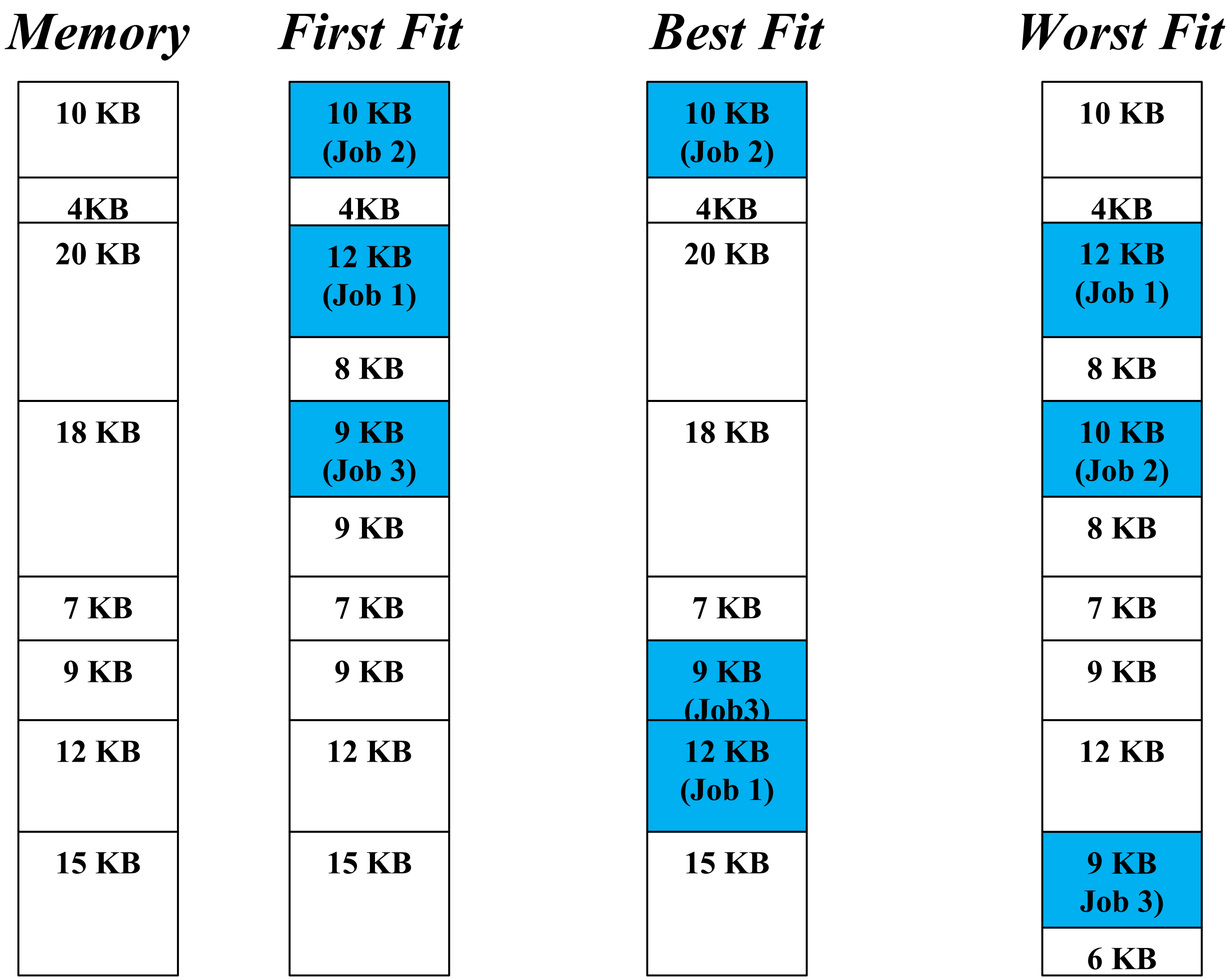
Compaction

Memory mapping after compaction

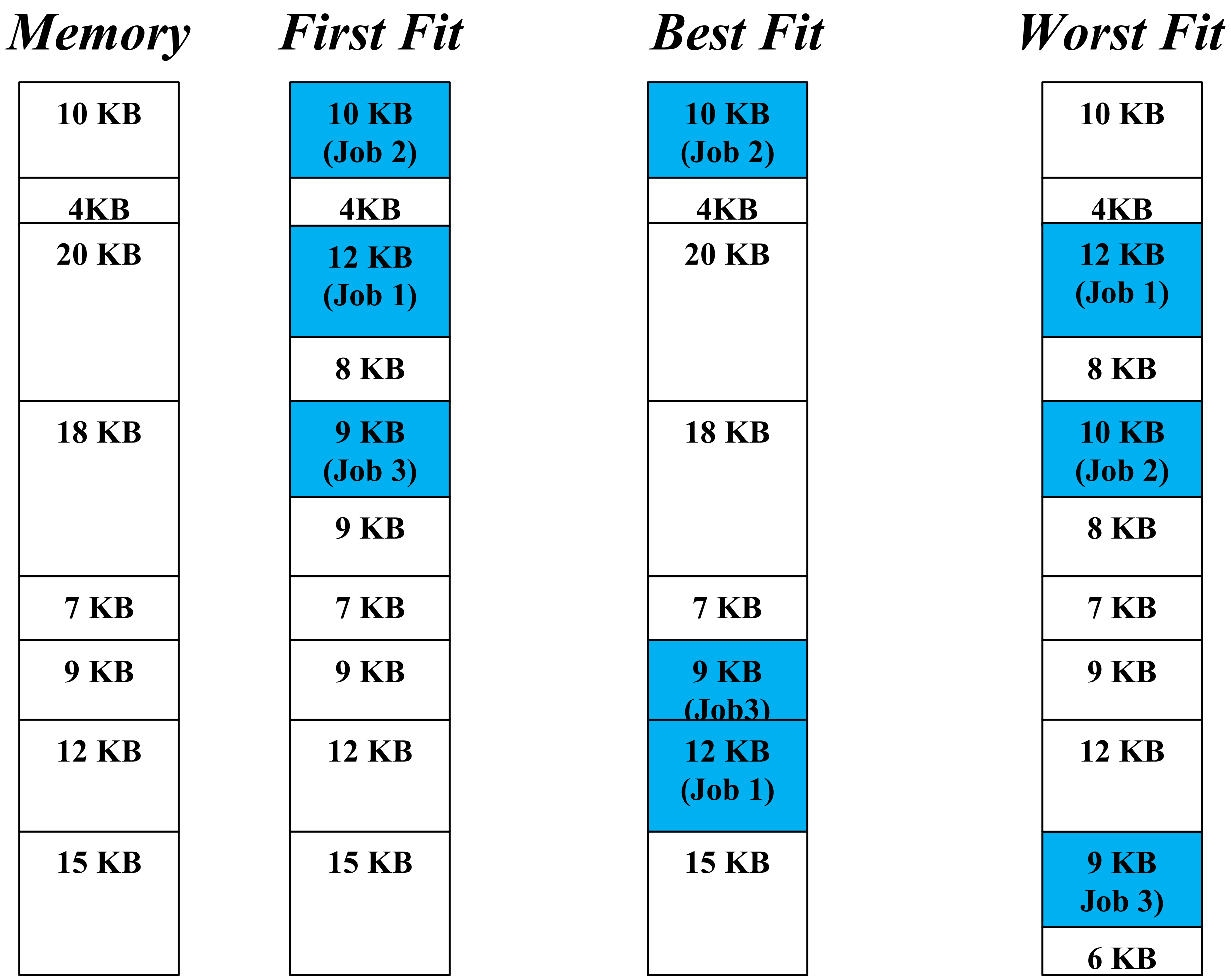


Now P5 of 15KB can be loaded here

1. Consider a swapping system in which memory consists of the following hole sizes in memory order: 10KB, 4KB, 20KB, 18KB, 7KB, 9KB, 12KB, and 15KB. Which hole is taken for successive segment requests of 12KB, 10KB, 9KB for *first fit*? Now repeat the question for *best fit, and worst fit*.



1. Consider a swapping system in which memory consists of the following hole sizes in memory order: 10KB, 4KB, 20KB, 18KB, 7KB, 9KB, 12KB, and 15KB. Which hole is taken for successive segment requests of 12KB, 10KB, 9KB for *first fit*? Now repeat the question for *best fit, and worst fit*.

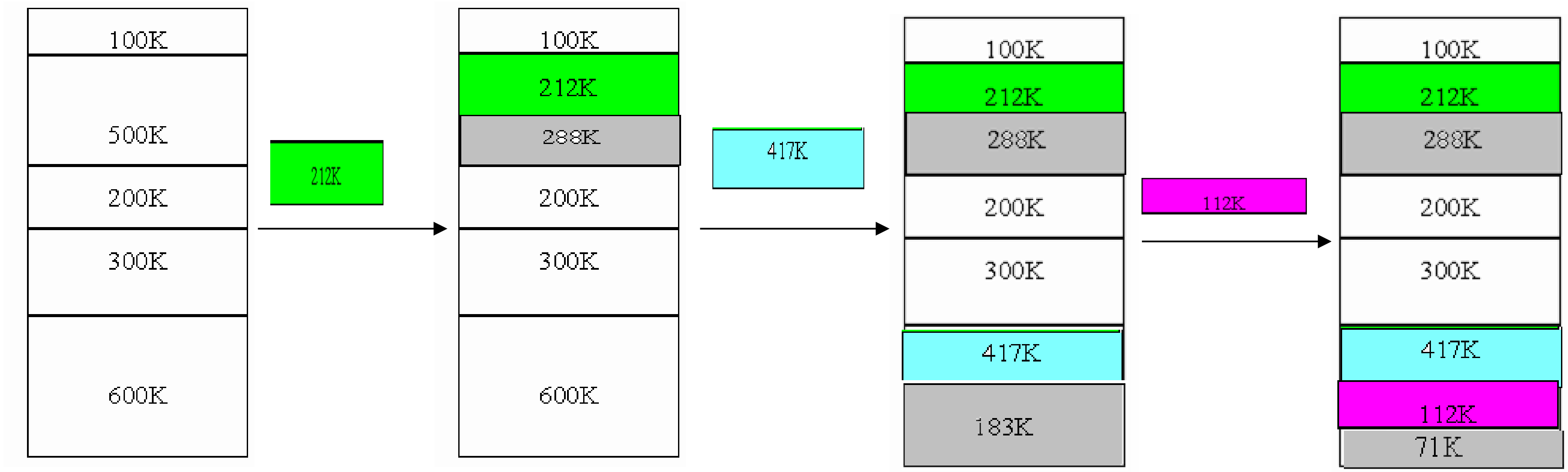


Memory Allocation Policies

Now take another theoretical example.

- Given the partition of 100K,500K,200K,300K,600K as shown, the different algorithms will place the processes 212K,417K,112K,426K respectively.
- The request for 426K will be rejected in case of next fit and worst fit algorithm because any single partition is less than 426K

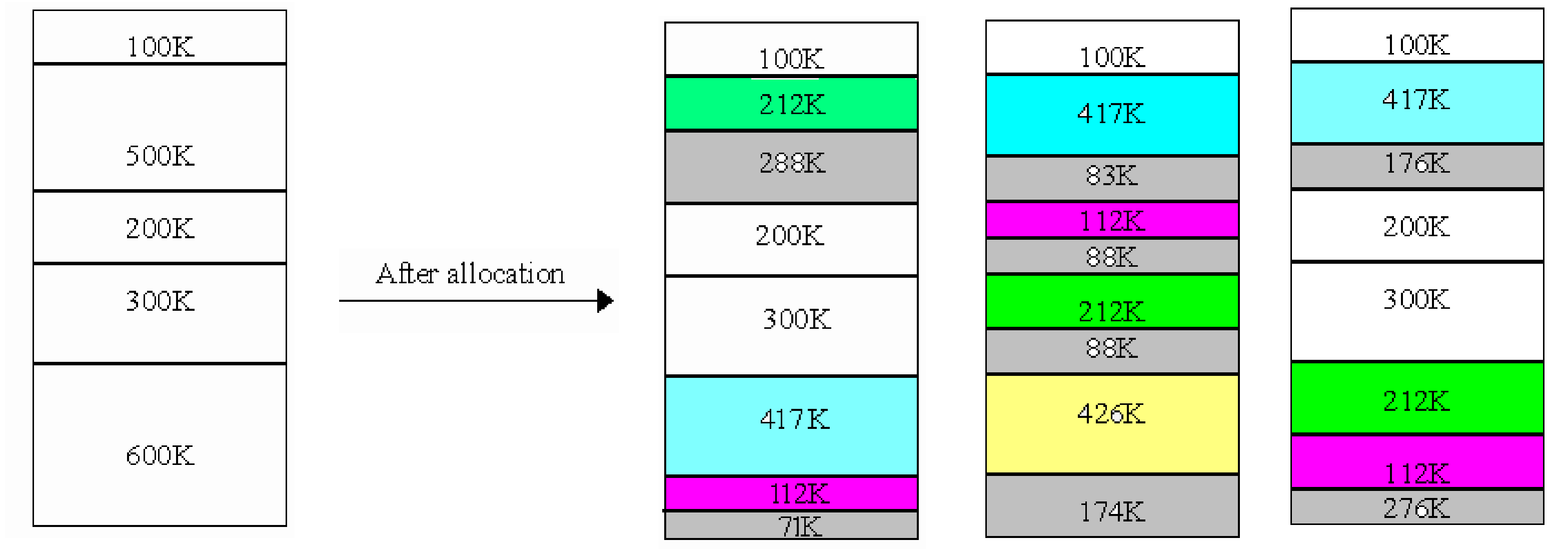
Next Fit (212k,417k,112k)



The request for 426K will be rejected

Similarly we cant implement for Other Two Policies

212K-Green
417K-Blue
112K-Pink
426K-Yellow
External Fragmentation-Gray
Unused Partitions-White



Next Fit

Best Fit

Worst Fit

2. Given memory partitions of 12KB, 7KB, 15KB, 20KB, 9KB, 4KB, 10KB, and 18KB (in order), how would each of the *first-fit* and *best-fit* algorithms place processes of 10KB, 12KB, 6KB, and 9KB (in order)?

Thank you