Subject Name: Operating Systems

Unit: 5 Unit Name: Memory Management

Faculty Name: Ms. Puja Padiya

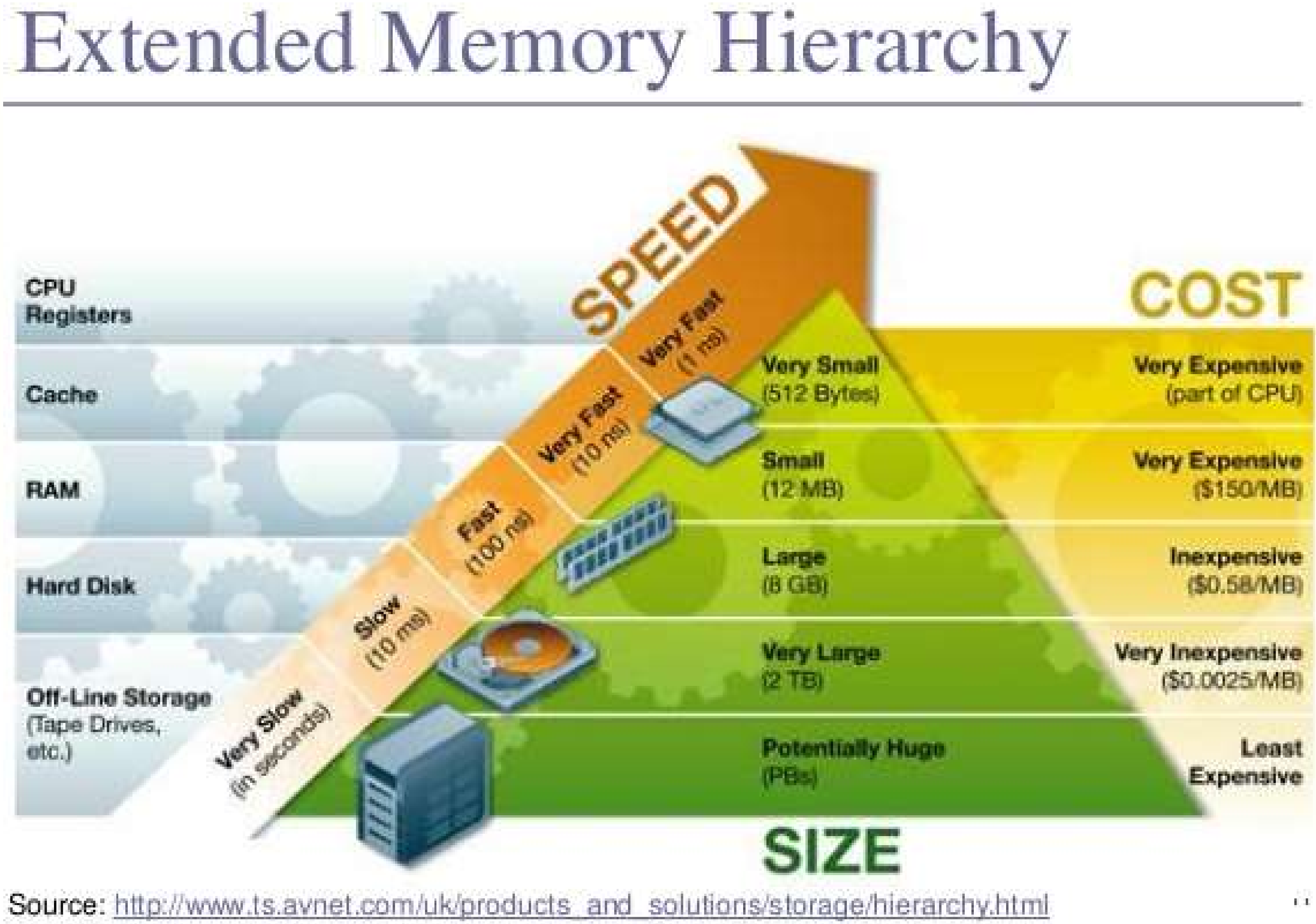
Index

Unit No: 5 Unit Name: Memory Management

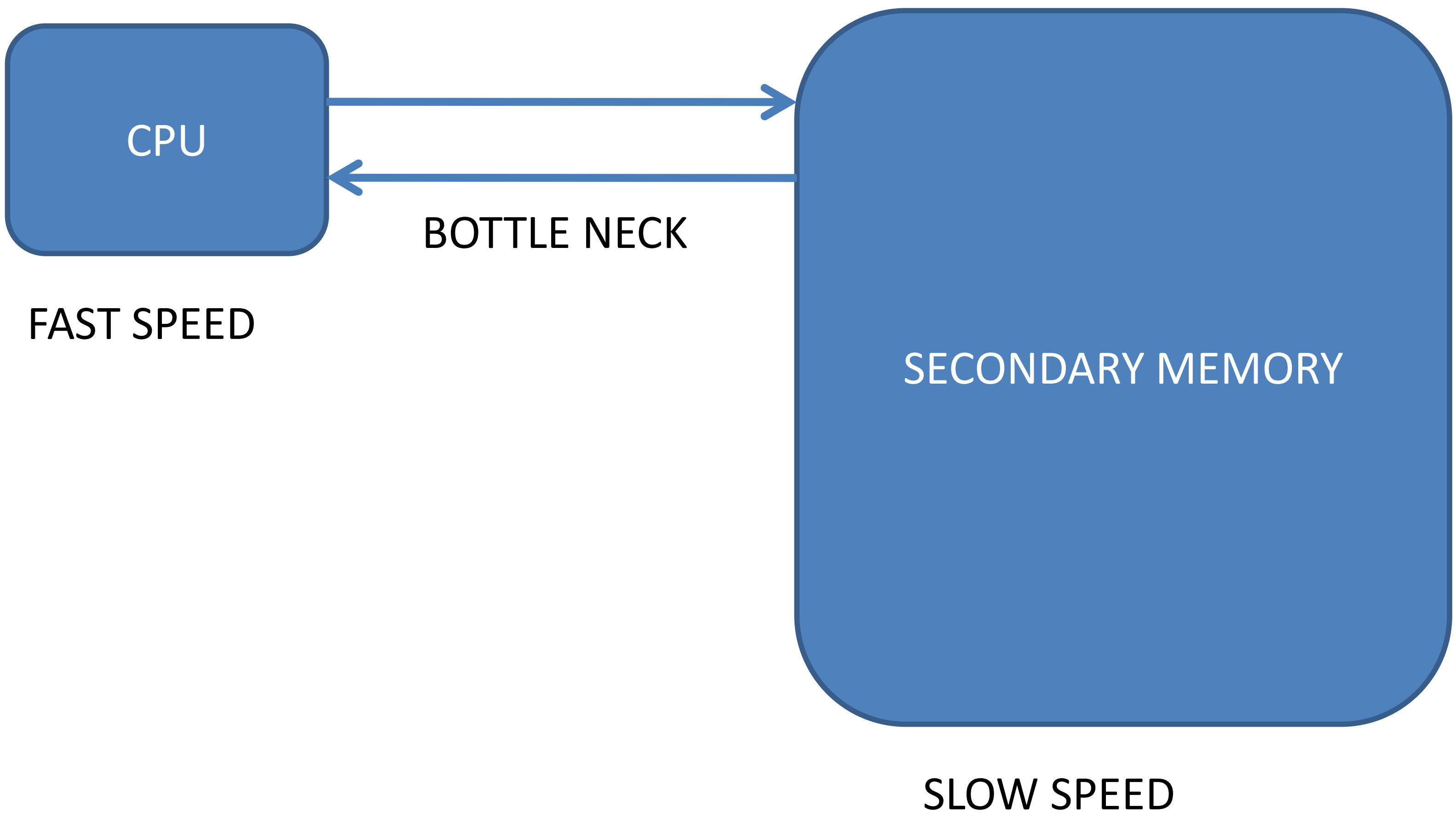
Lecture:

Memory Management: Memory Management Requirements, Memory Partitioning: Fixed Partitioning, Dynamic Partitioning

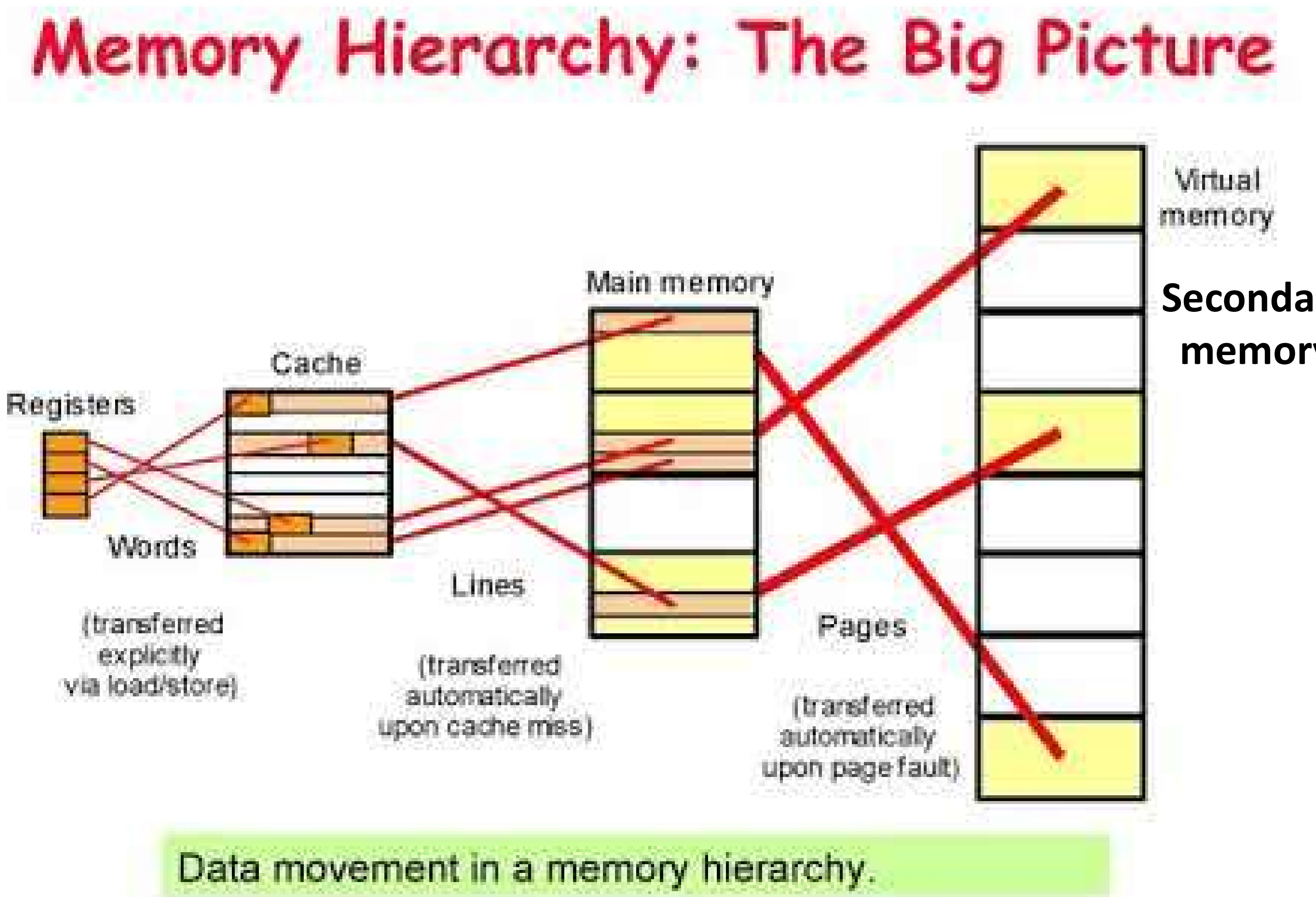
Memory Hierarchy



Memory Hierarchy



Data movement in memory hierarchy



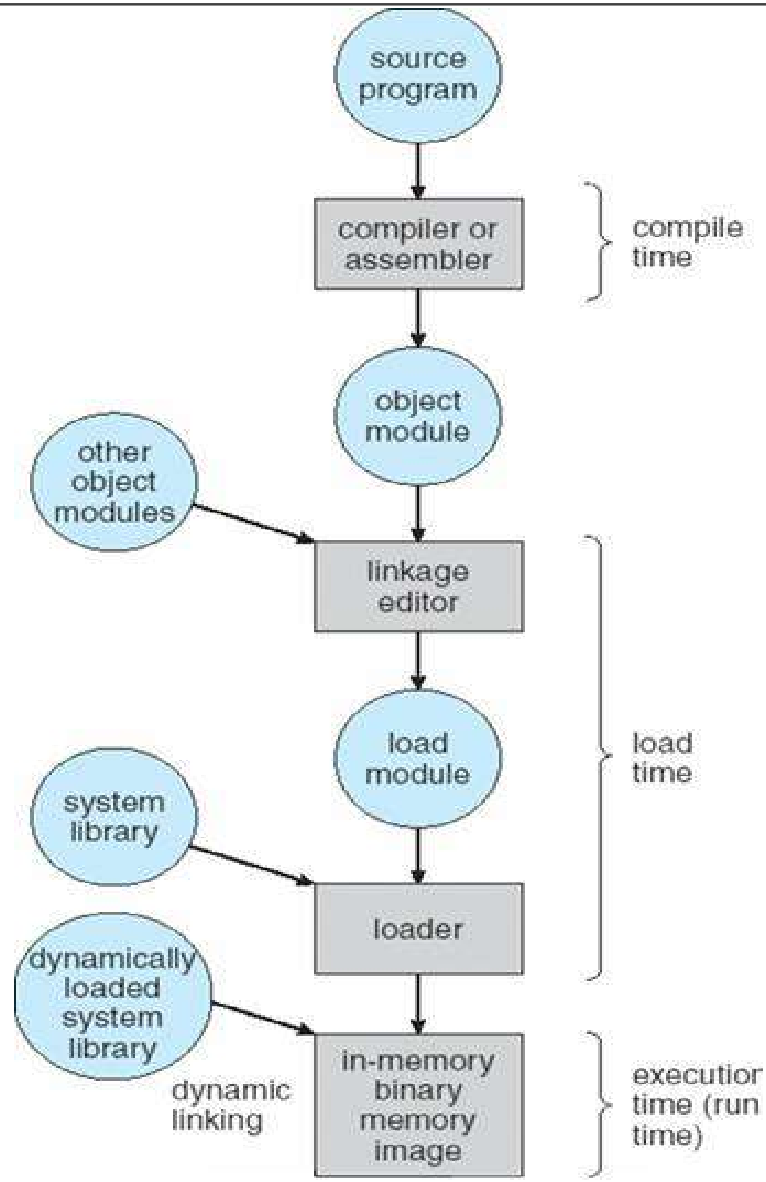
Secondary memory

Memory Management- Background

- Program must be brought into memory and placed within a process for it to be run
- Input queue or job queue – collection of processes on the disk that are waiting to be brought into memory to run the program
- User programs go through several steps before being run.

Memory Management- Background

Multistep Processing of a User Program



Memory Management- Background

- **Address Binding**
- 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; must recompile code if starting location changes
    - Minimum setup time
    - Logical address
  - **Load time**: Must generate **relocatable code** if memory location is **not known** at compile time.
    - Physical address
  - **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)

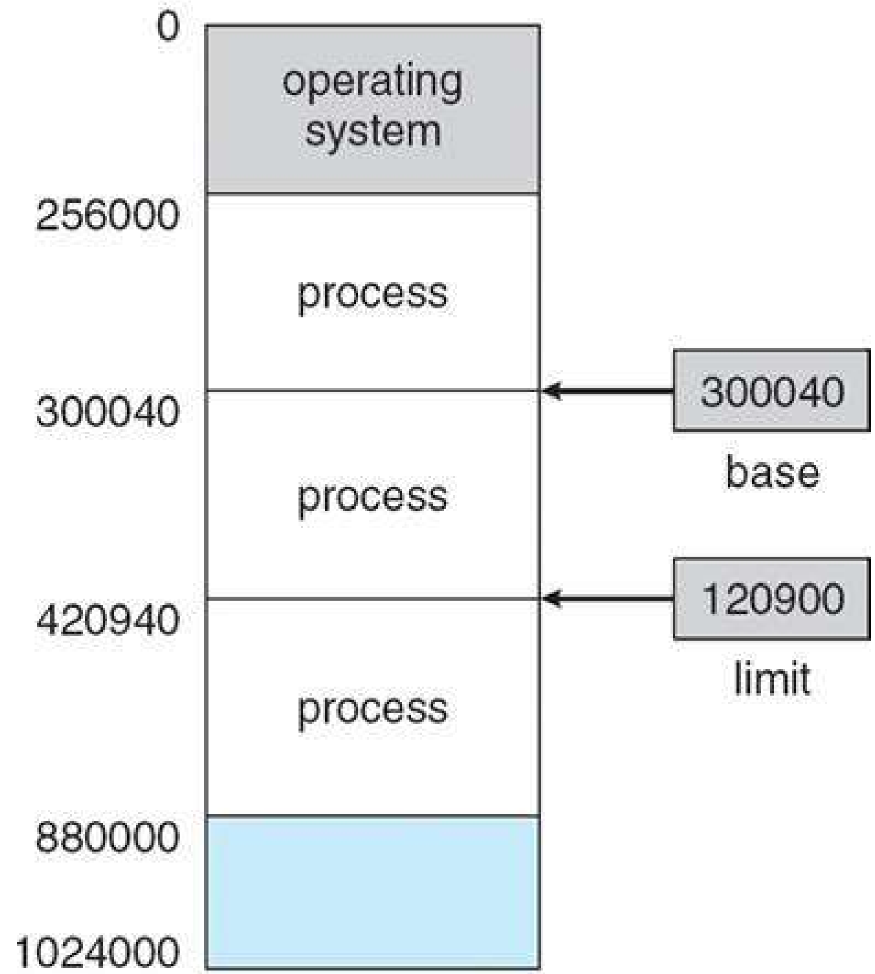


Figure: Base and Limit Registers

Memory Management- Background

- **Memory Protection**
  - **User processes** must be **restricted** so that they only access memory locations that "**belong**" to that particular process.
  - This is usually **implemented using a base register** and **a limit register** for each process.
  - ***Every*** memory access made by a user process is **checked against these two registers**

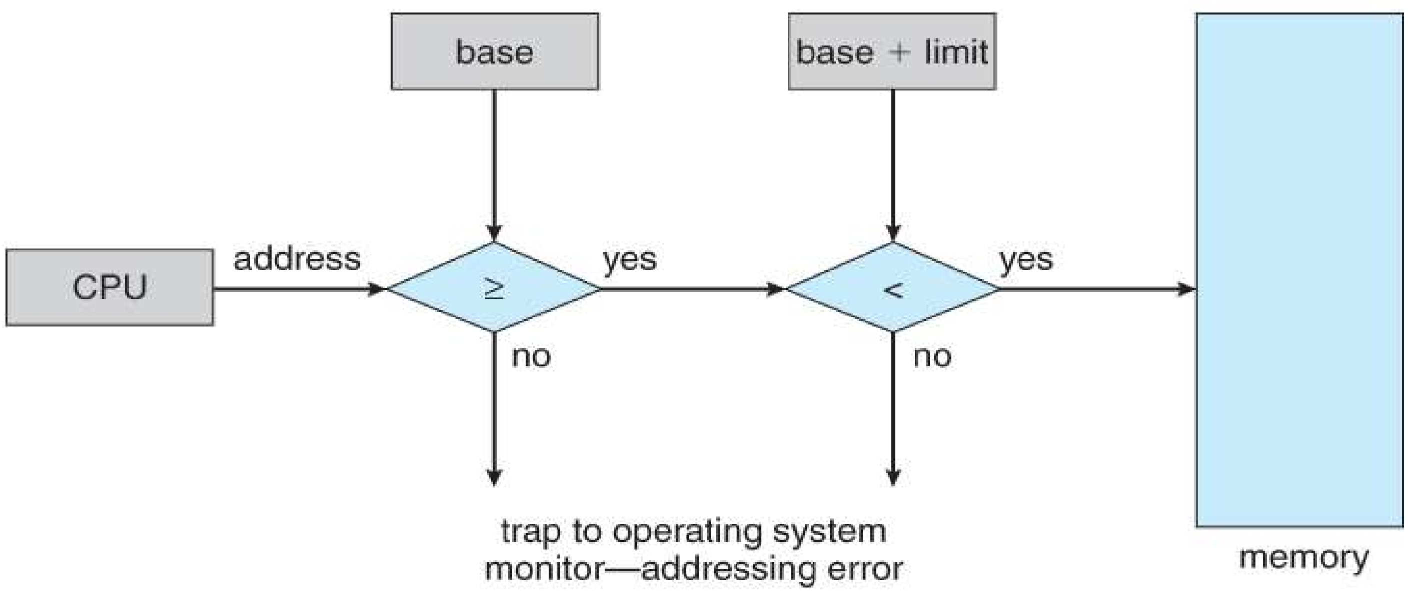


Figure: Hardware address protection with base and limit registers

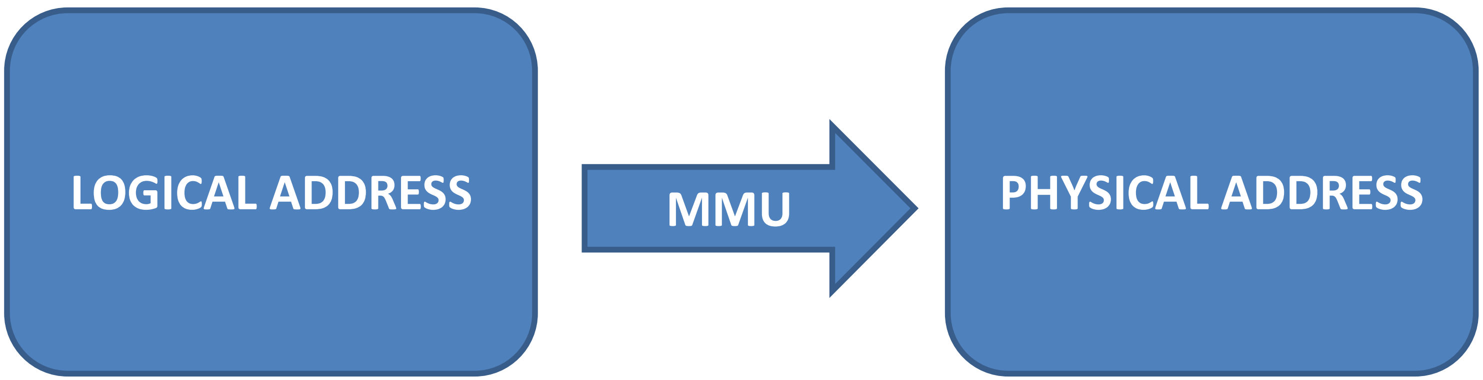
Memory Management- Background

- **Dynamic Loading**
- Routine is not loaded **until it is called**
  - Better memory-space utilization; **unused routine is never loaded**
  - Useful when large amounts of code are needed to handle **infrequently occurring cases**

- **Dynamic Linking**
- When Routine is loaded then it will link to library routines

Memory Management- Background

- **Logical vs. Physical Address Space**
  - **Logical address space** bound to a separate **physical address space** is central concept in memory management
  - **Logical address** – **generated by the CPU**; also referred to as **virtual address.**
  - **Physical address** – **address seen** by the **memory unit**.

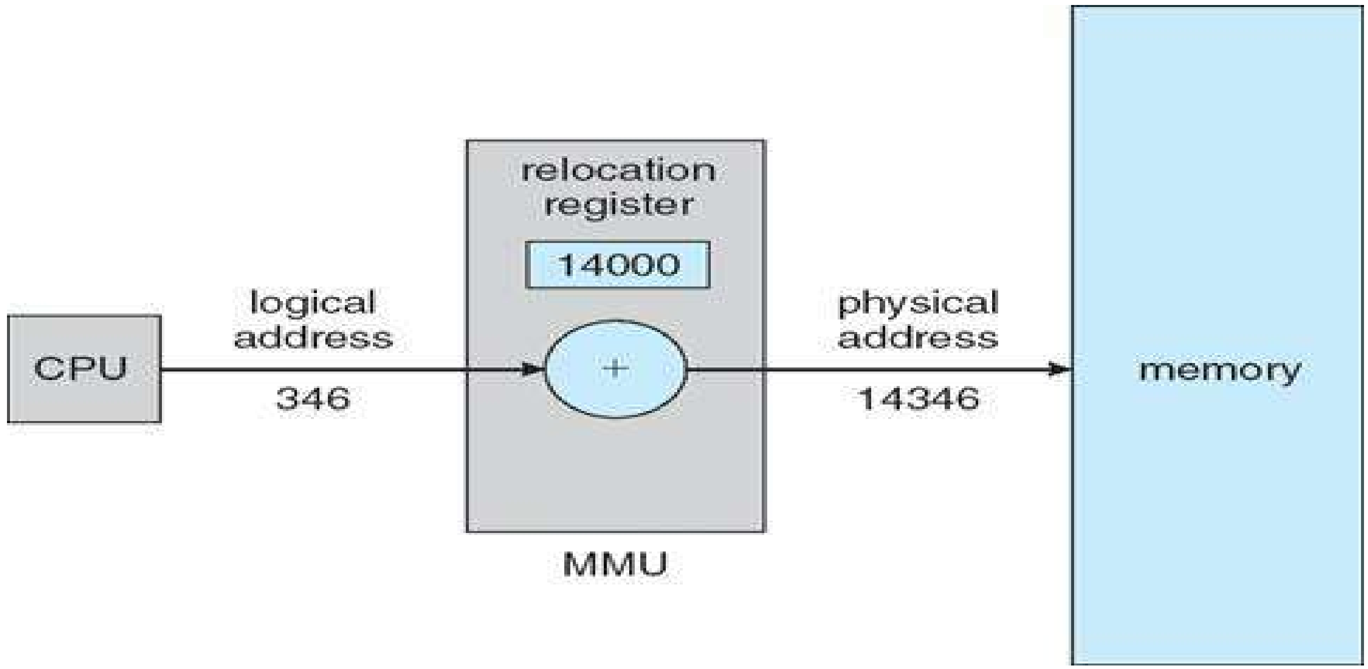


CPU

Memory Unit

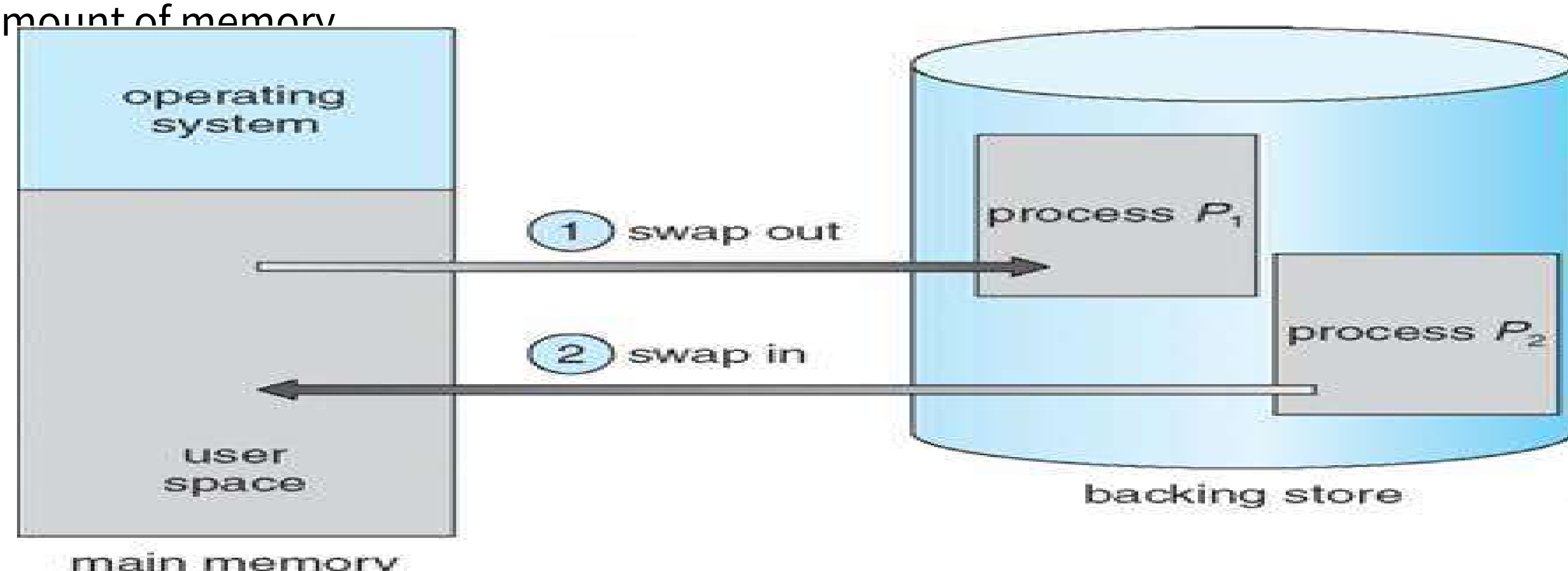
Memory Management- Background

- **Memory-Management Unit (MMU)**
  - Hardware device that maps **virtual** to **physical** address
  - The value in the **relocation register** is added to every address generated by a user process
  - The **user program deals with *logical* addresses**; it never sees **the real *physical* addresses**

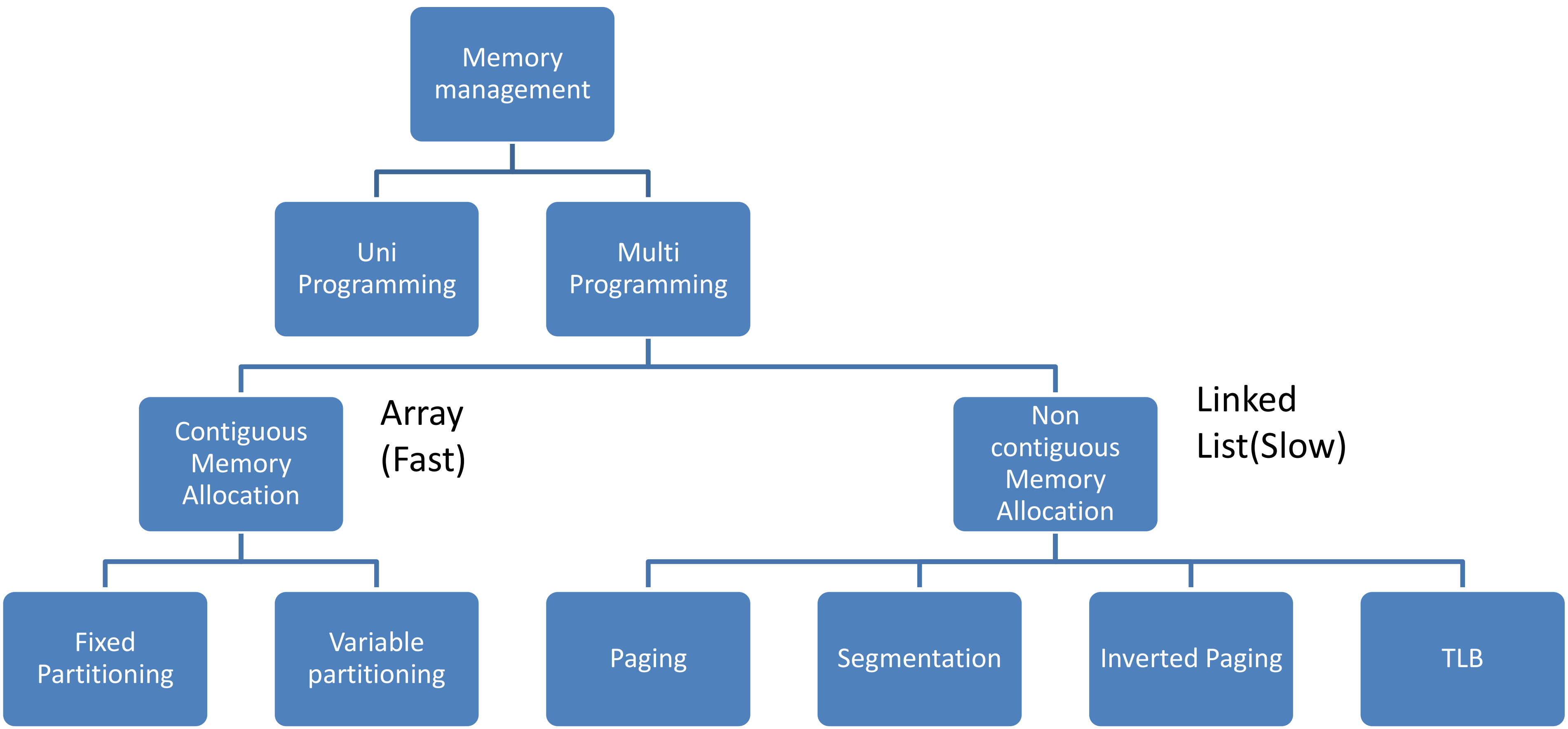


Swapping

- 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
- Major part of swap time is transfer time; total transfer time is directly proportional to the amount of memory



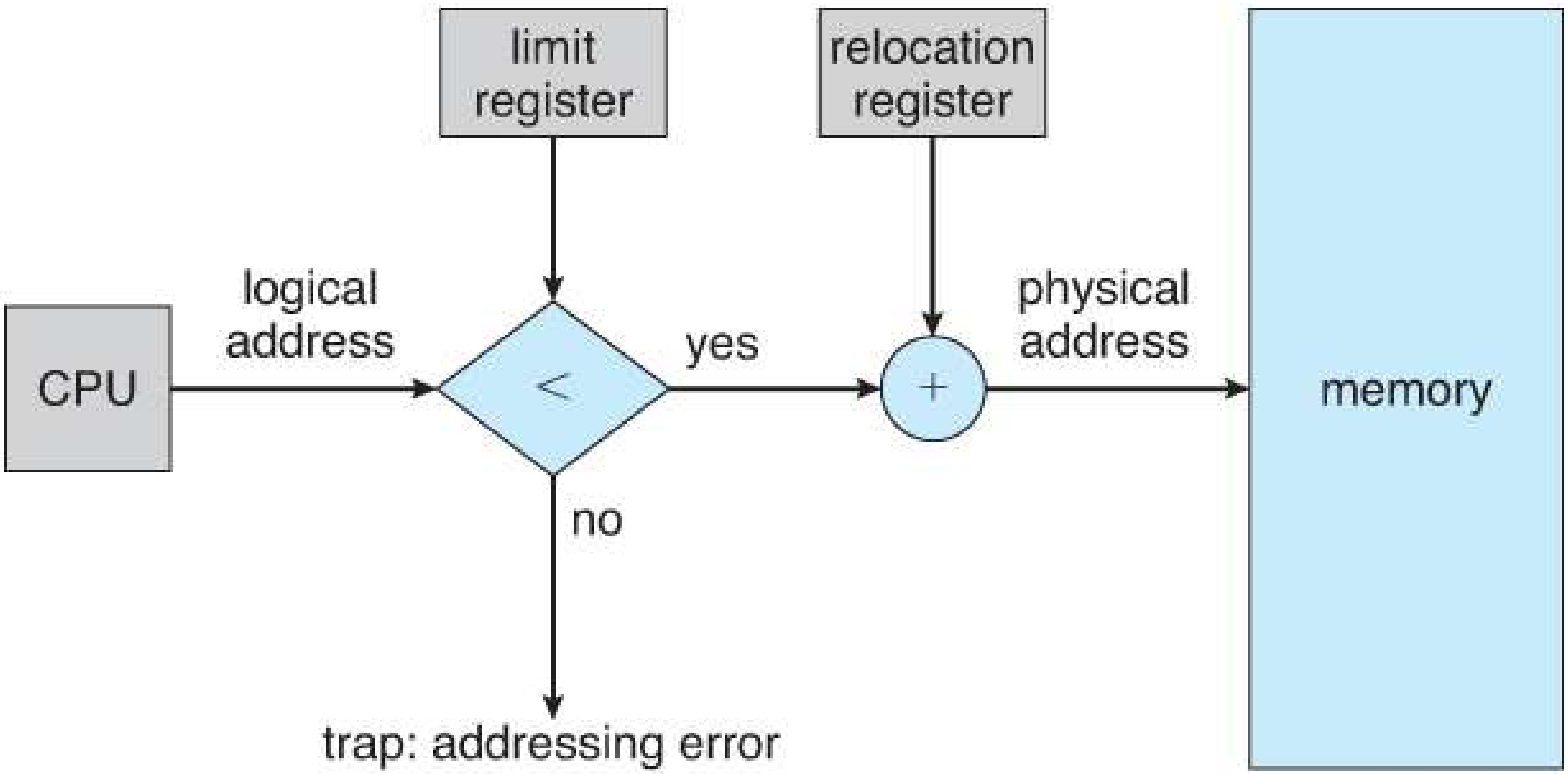
Memory management



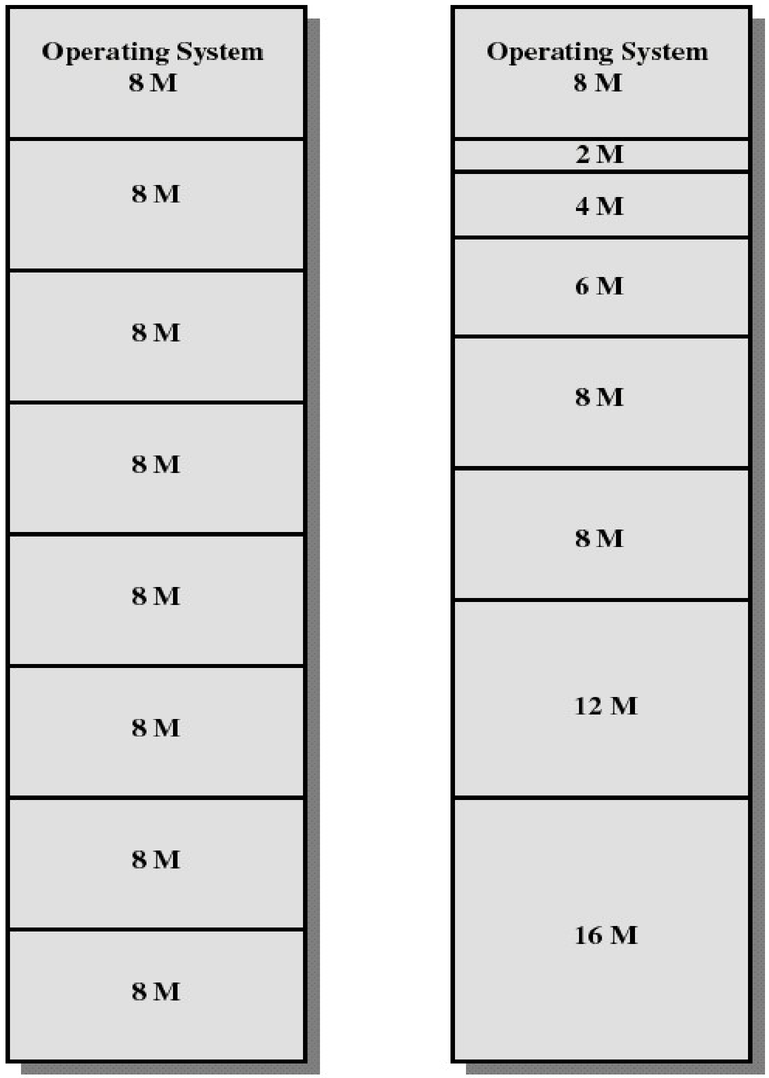
Memory Management- Contiguous Memory Allocation

- One approach to memory management is to load each process into a contiguous space.
- The operating system is allocated space first, usually at either low or high memory locations, and then the remaining available memory is allocated to processes as needed.
- **Memory Protection**
  - Protection against user programs accessing areas that they should not, allows programs to be relocated to different memory starting addresses as needed, and allows the memory space devoted to the OS to grow or shrink dynamically as needs change.

Memory Management- Contiguous Memory Allocation



Fixed Partitioning



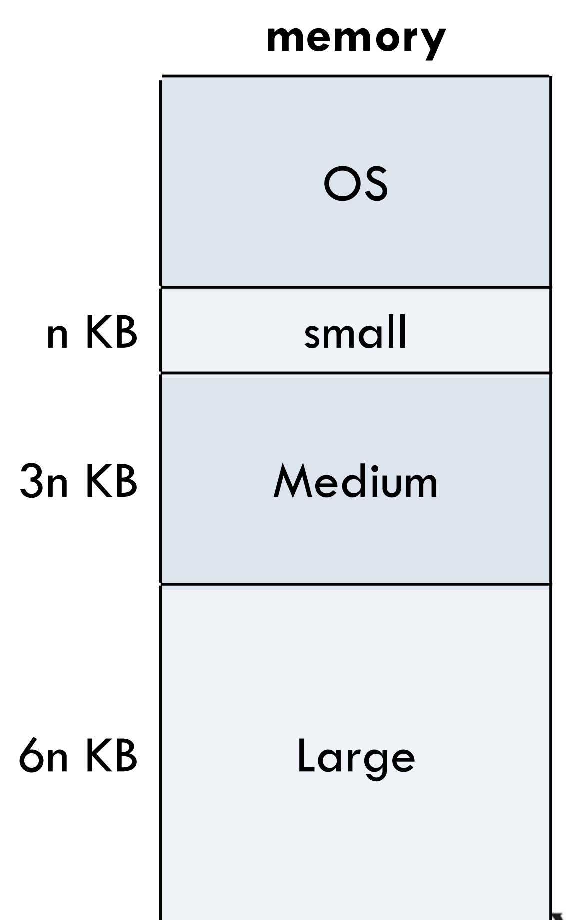
Equal-size partitions

Unequal-size partitions

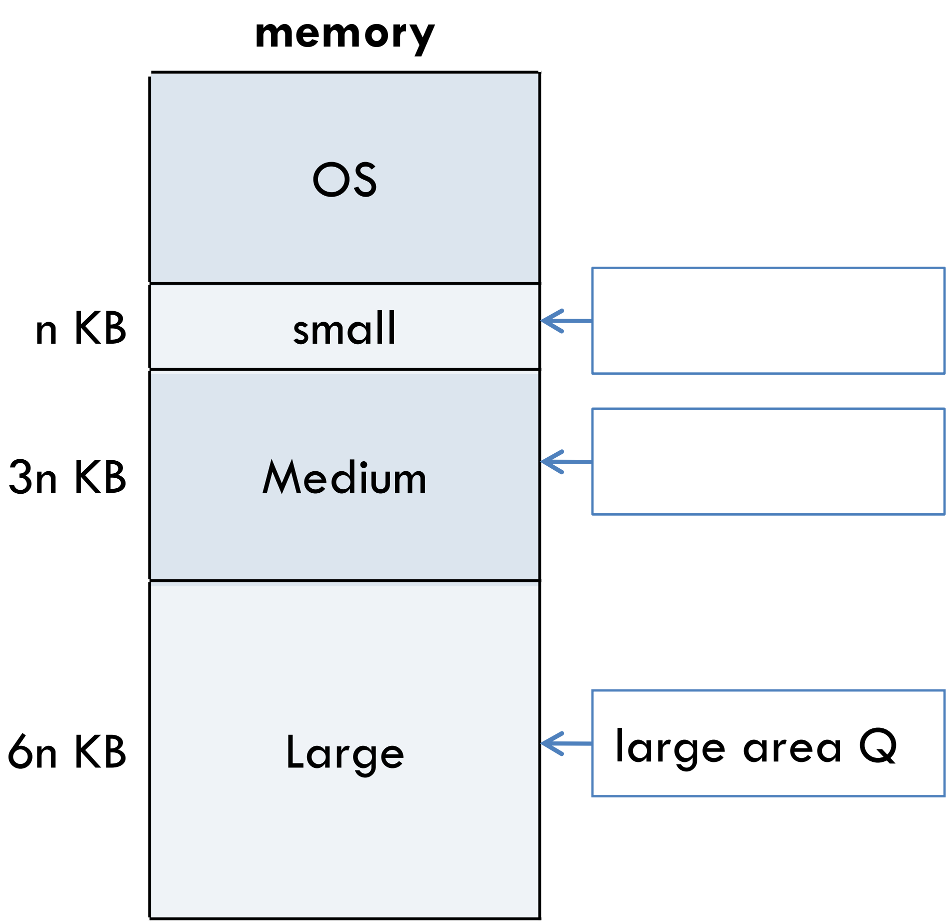
- Partition main memory into a set of non-overlapping memory regions called partitions.
- Fixed partitions can be of equal or unequal sizes.
- Leftover space in partition, after program assignment, is called internal fragmentation.

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

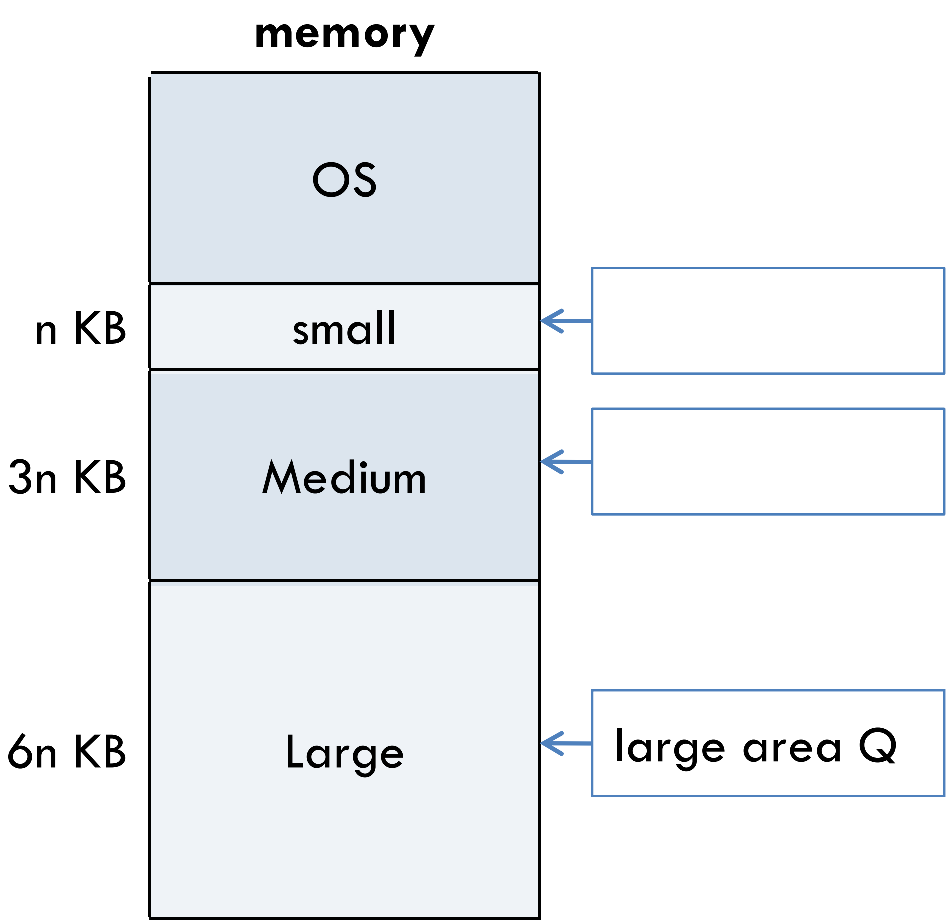


3.1 Fixed Partitioning



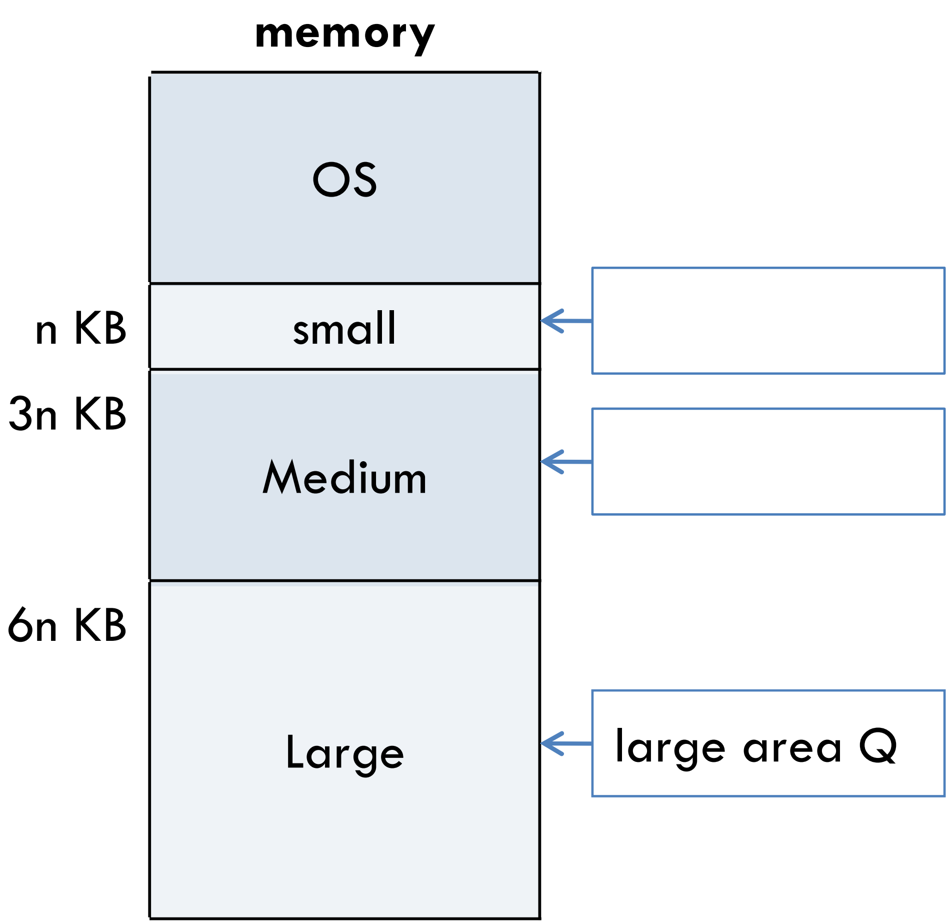
- Processes are classified on entry to the system according to their memory they requirements.
- We need one *Process Queue (PQ)* for each class of process.

3.1 Fixed Partitioning



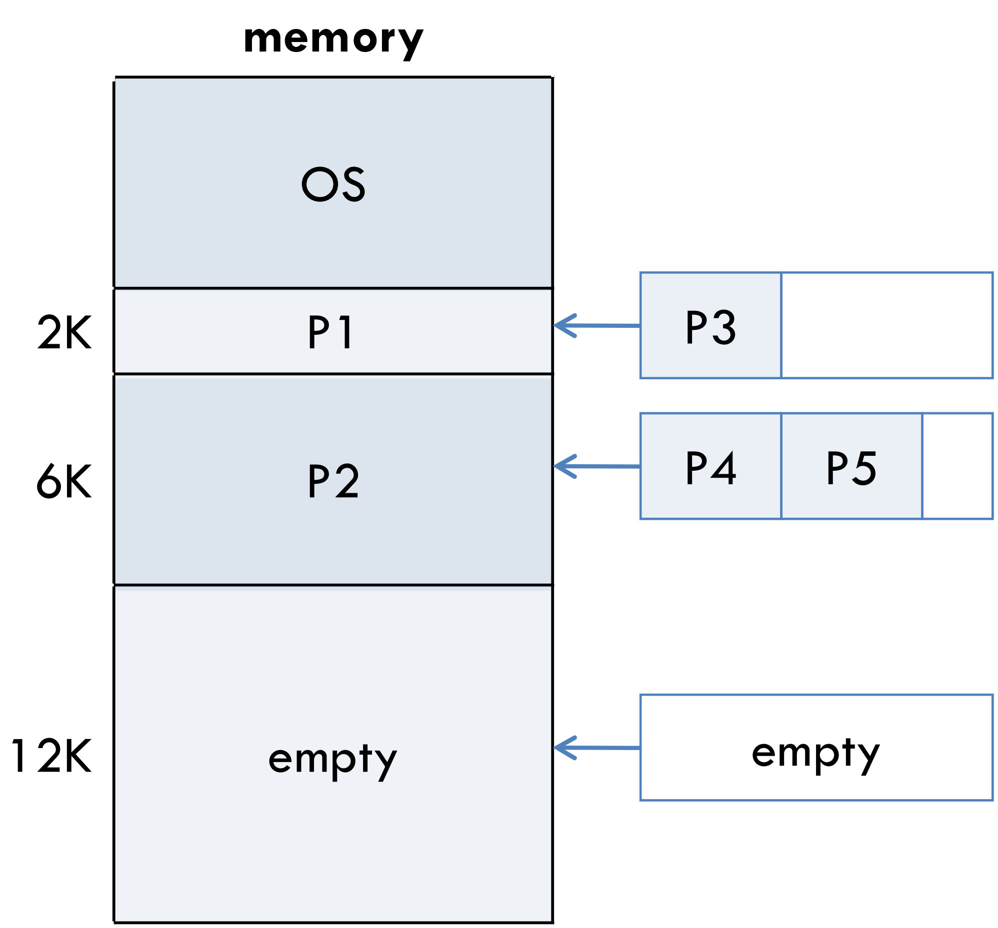
- If a process is selected to allocate memory, then it goes into memory and competes for the processor.
- The number of fixed partition gives the degree of multiprogramming.
- Since each queue has its own memory region, there is no competition between queues for the memory.

3.1 Fixed Partitioning



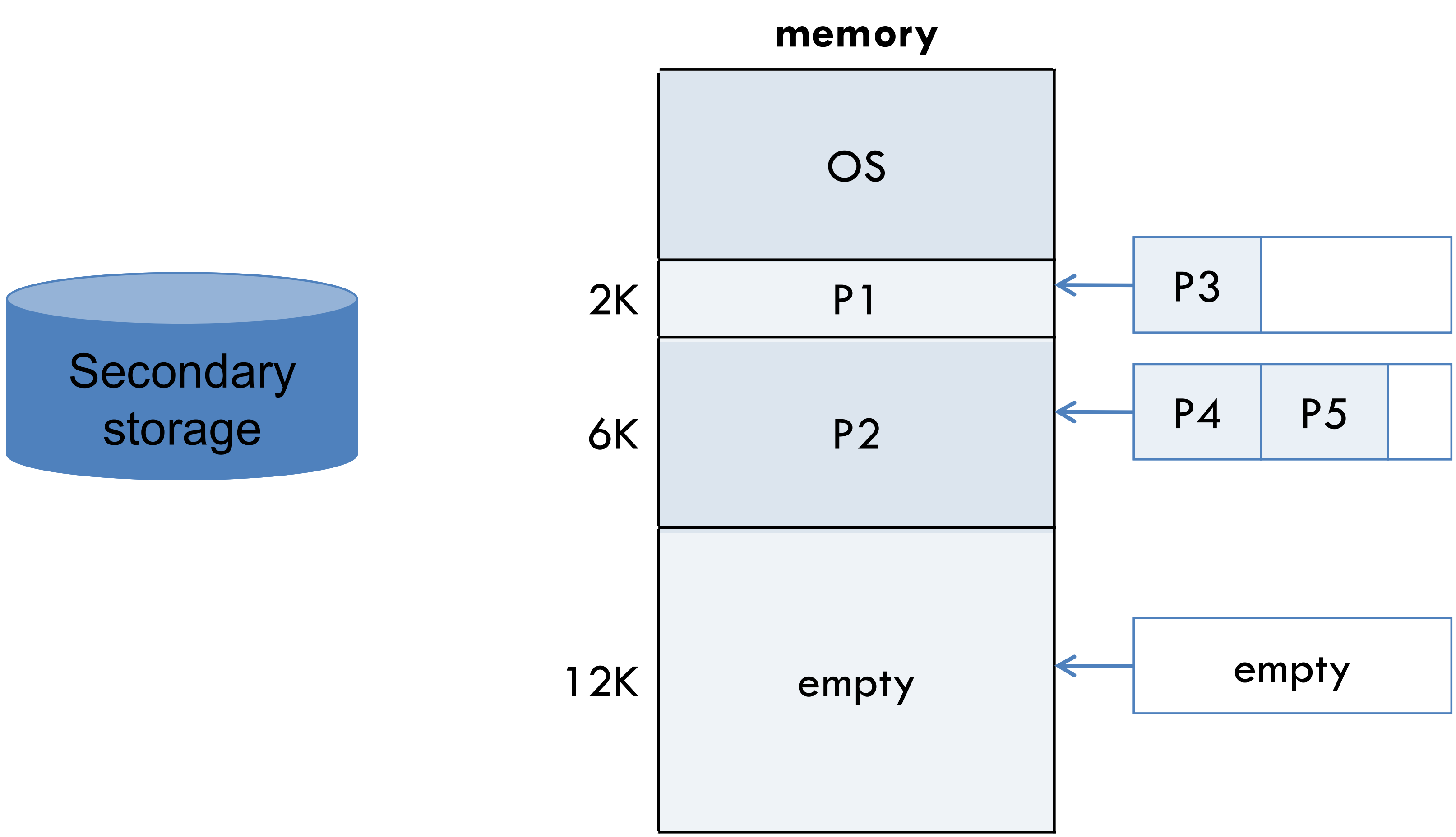
- The main problem with the fixed partitioning method is how to determine the number of partitions, and how to determine their sizes.

Fixed Partitioning with Swapping



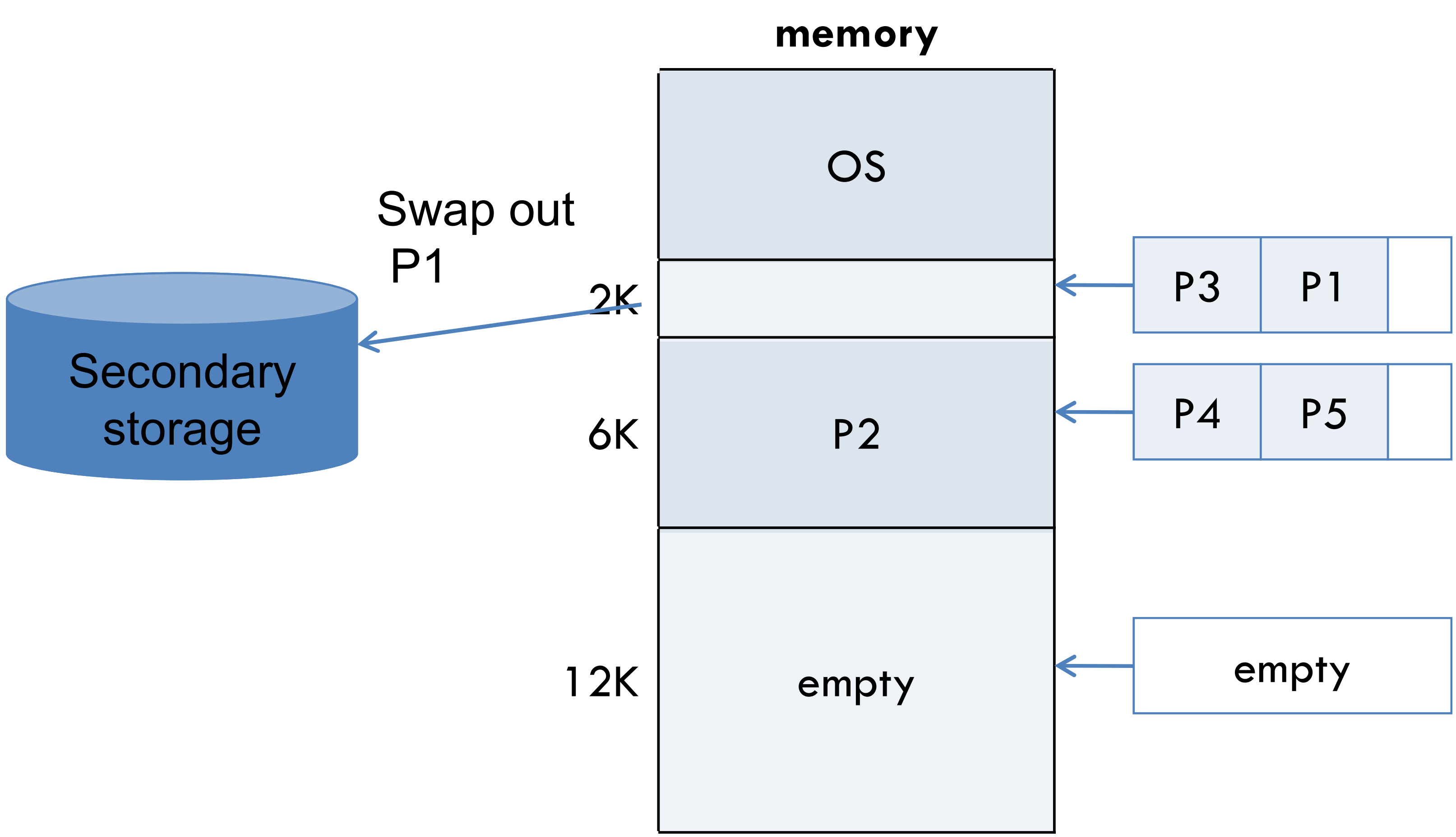
- This is a version of fixed partitioning that uses RRS with some time quantum.
- When time quantum for a process expires, it is swapped out of memory to disk and the next process in the corresponding process queue is swapped into the memory.

Fixed Partitioning with Swapping

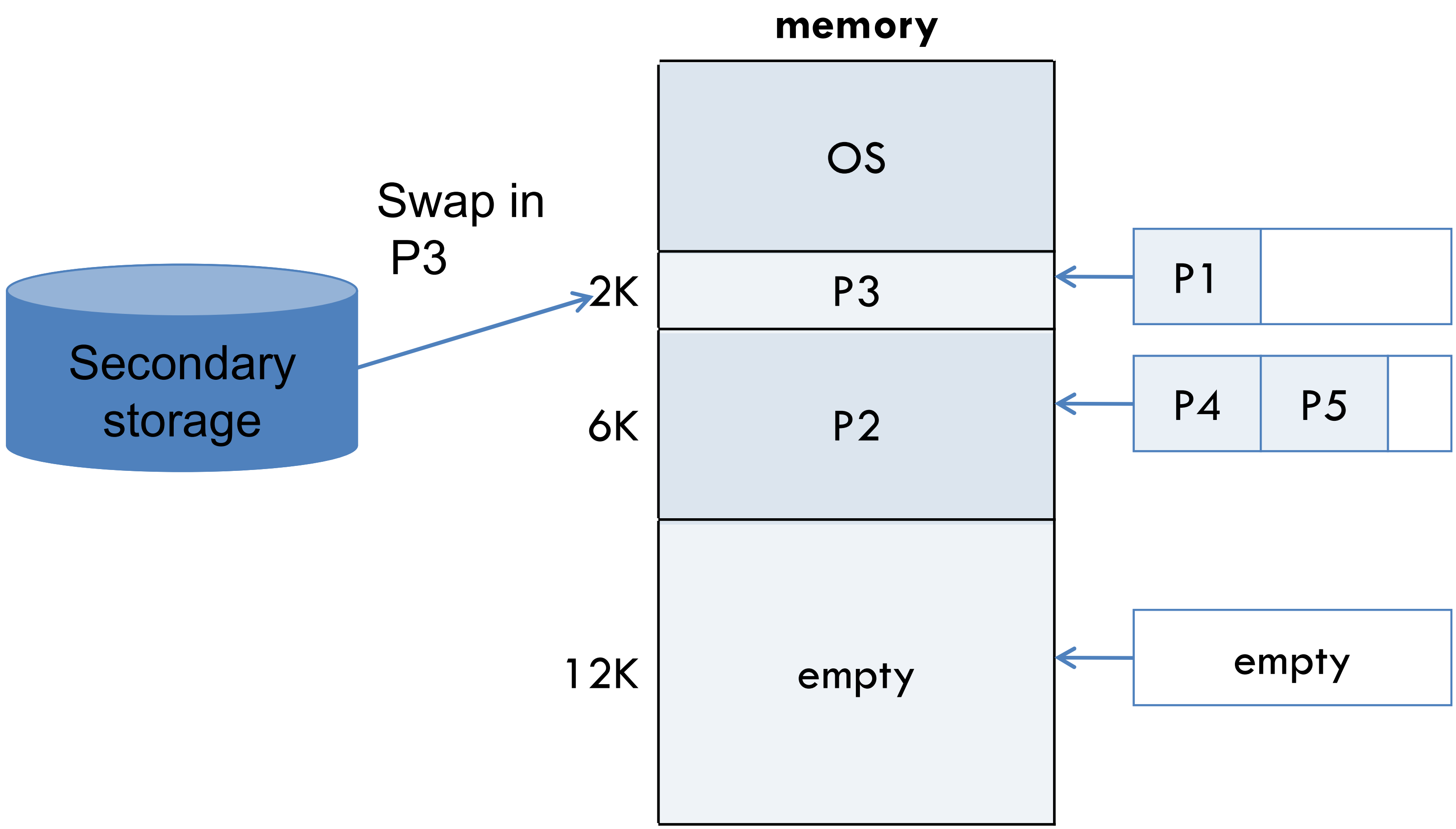


D Y PATIL DEEMED TO BE UNIVERSITY RAMRAO ADIK INSTITUTE OF TECHNOLOGY NAVI MUMBAI

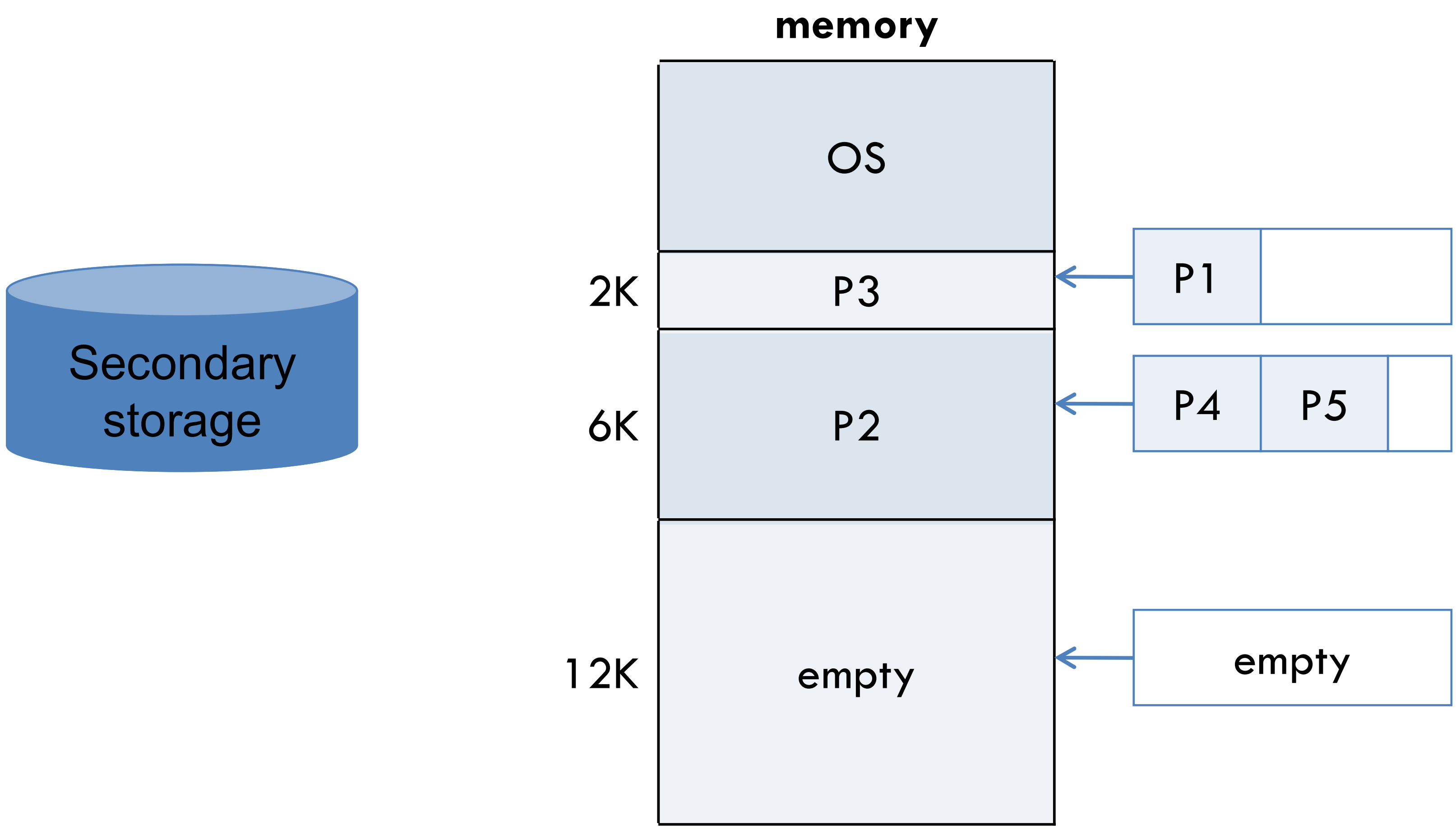
Fixed Partitioning with Swapping



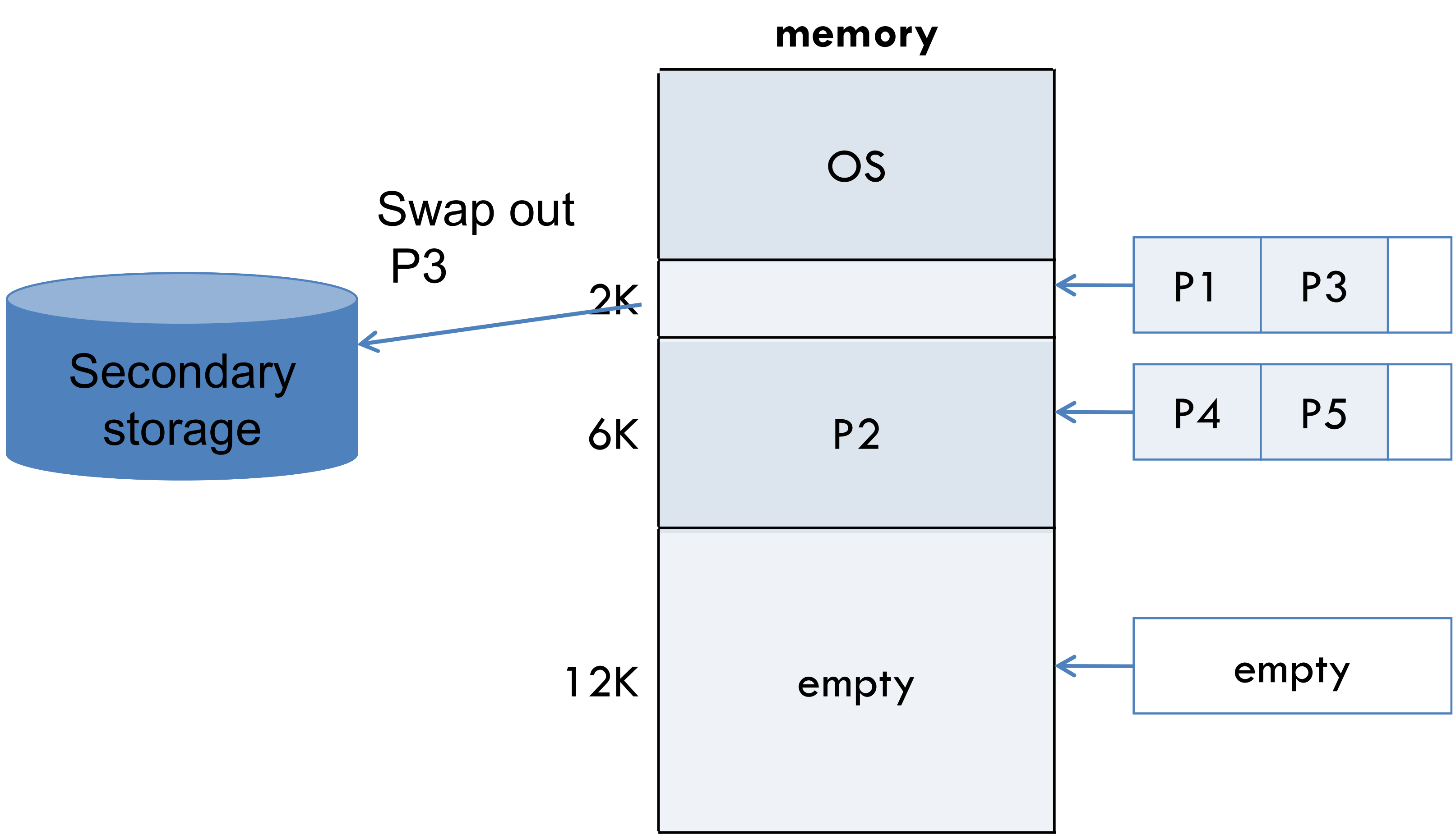
Fixed Partitioning with Swapping



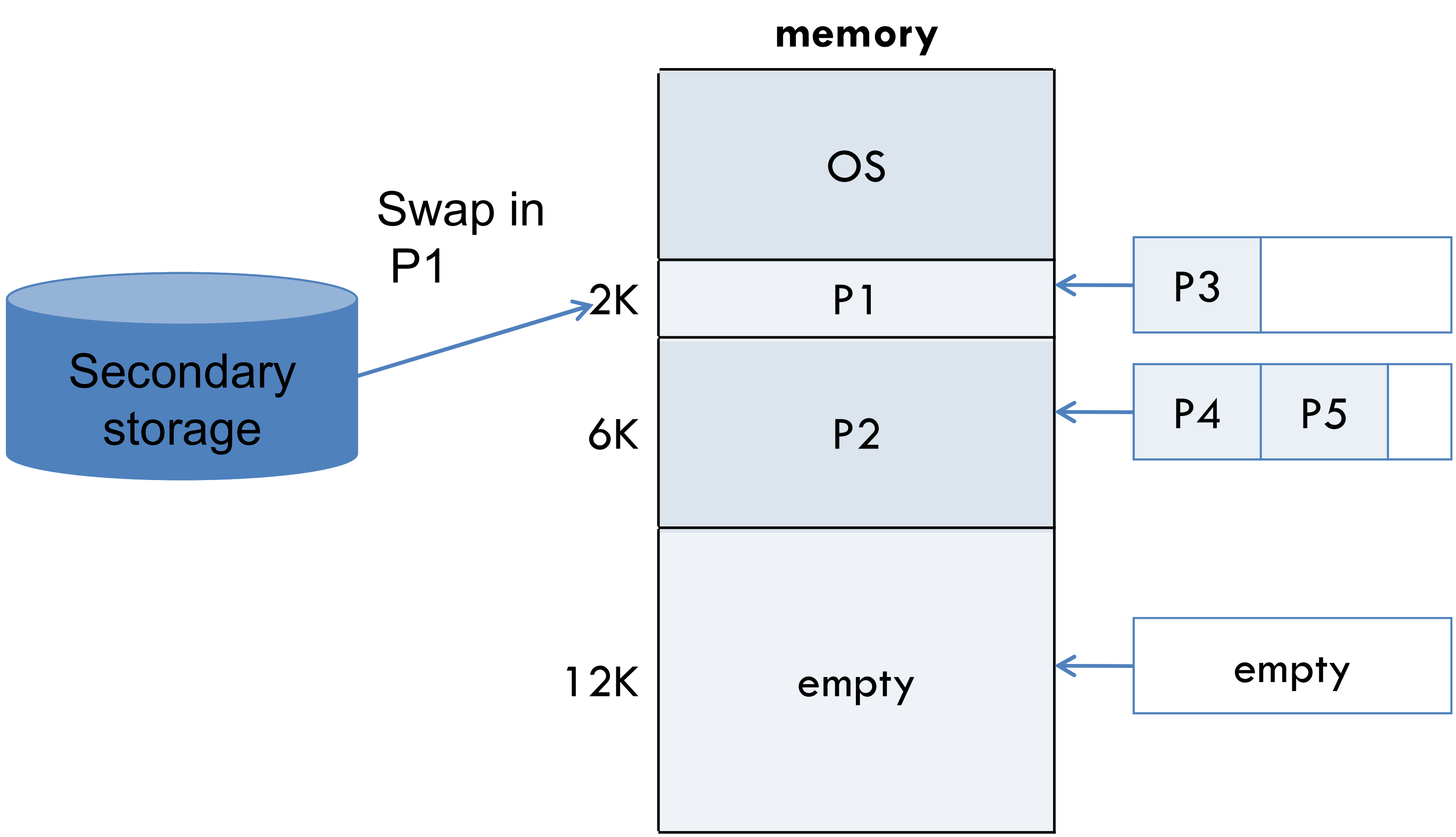
Fixed Partitioning with Swapping



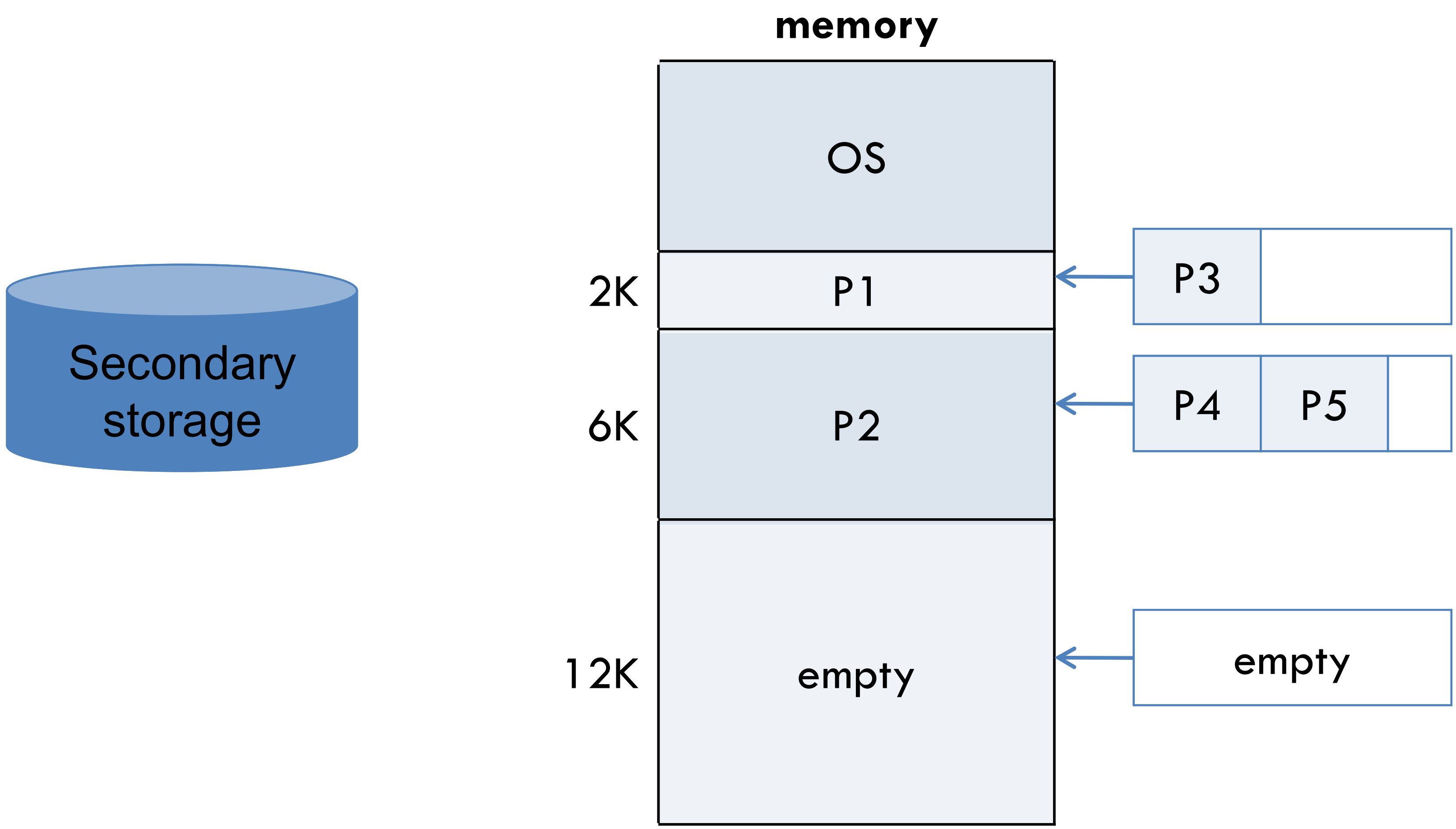
Fixed Partitioning with Swapping



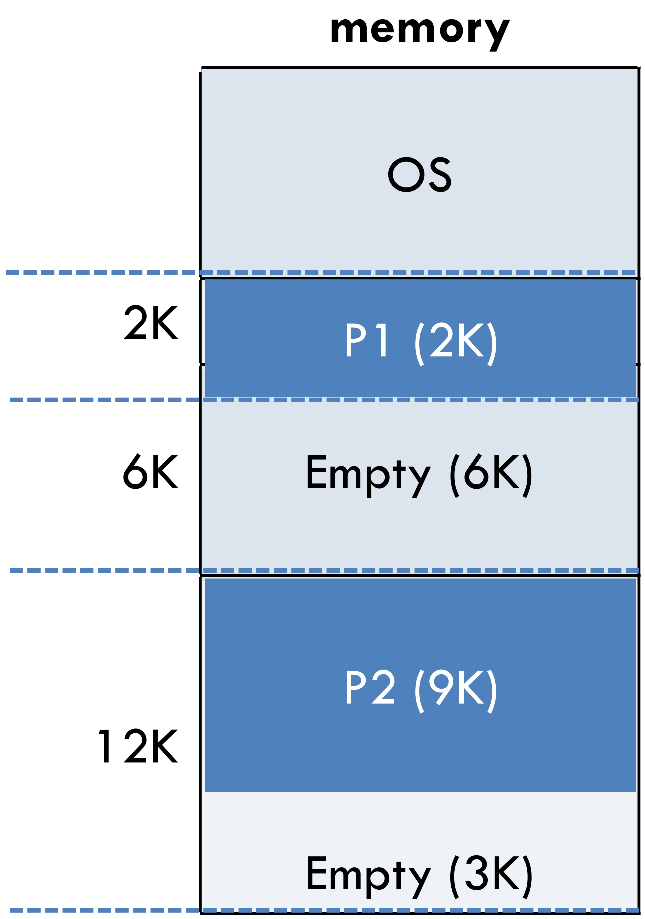
Fixed Partitioning with Swapping



Fixed Partitioning with Swapping



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 Partitioning

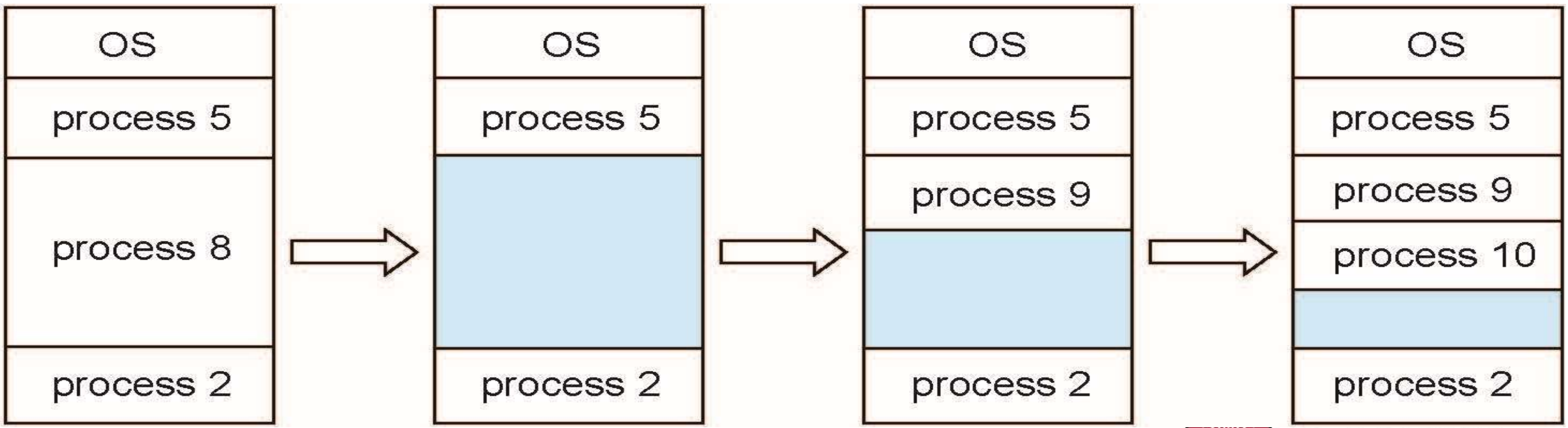
- With fixed partitions we have to deal with the problem of determining the number and sizes of partitions to minimize internal and external fragmentation.
- If we use variable partitioning instead, then partition sizes may vary dynamically.
- In the variable partitioning method, we keep a table (linked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

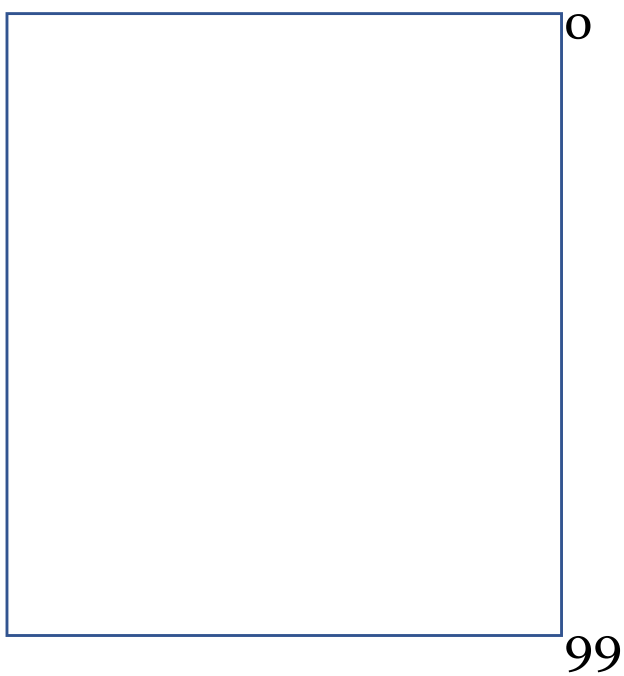
Variable Partitioning

- 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 partition working

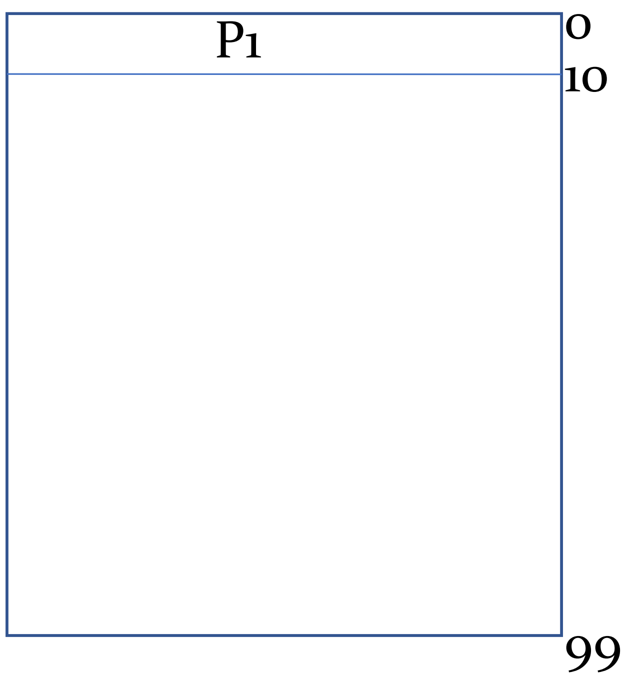
- Suppose total memory is 100Mb



Memory	Status
0-99	Free

Variable partition working

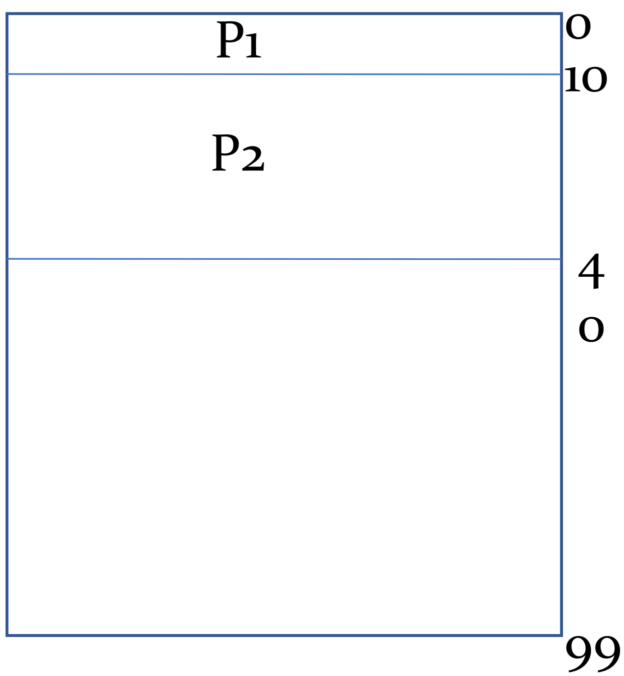
- Process P1 requires 10Mb space



Memory	Status
0-9	P1
10-99	free

Variable partition working

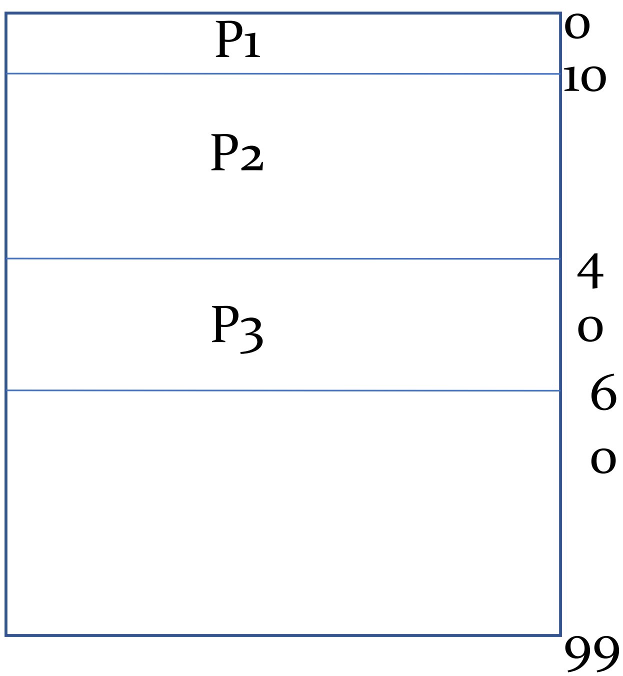
- Process P2 requires 30Mb space



Memory	Status
0-9	P1
10-99	free

Variable partition working

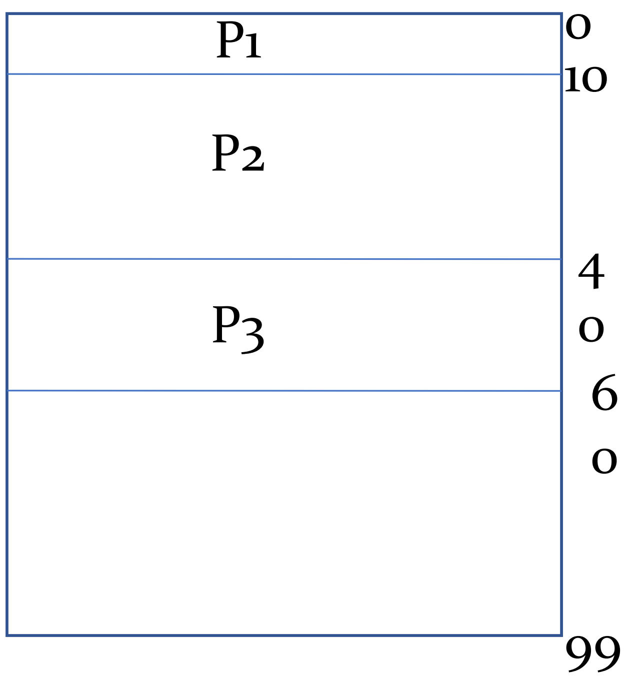
- Process P3 requires 20Mb space



Memory	Status
0-9	P1
10-39	P2
40-59	P3
60-99	Free

Variable partition working

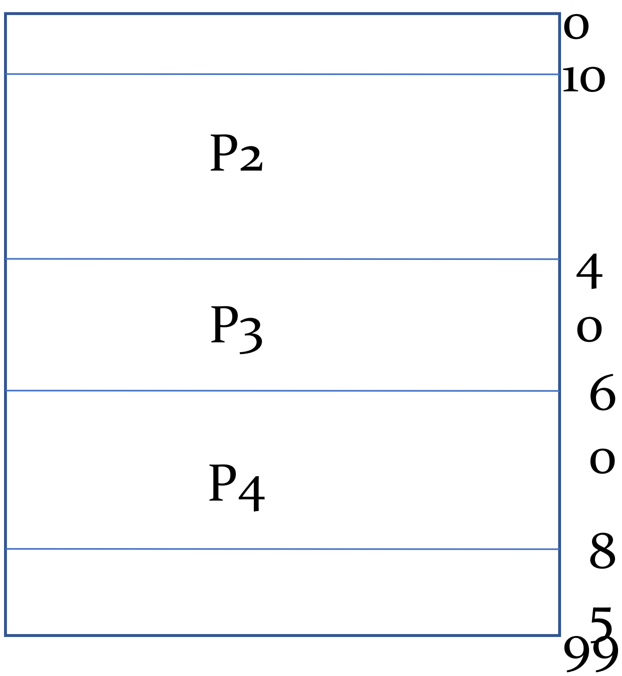
- Process P1 finishes its execution.
- So the space is freed



Memory	Status
0-9	Free
10-39	P2
40-59	P3
60-99	Free

Variable partition working

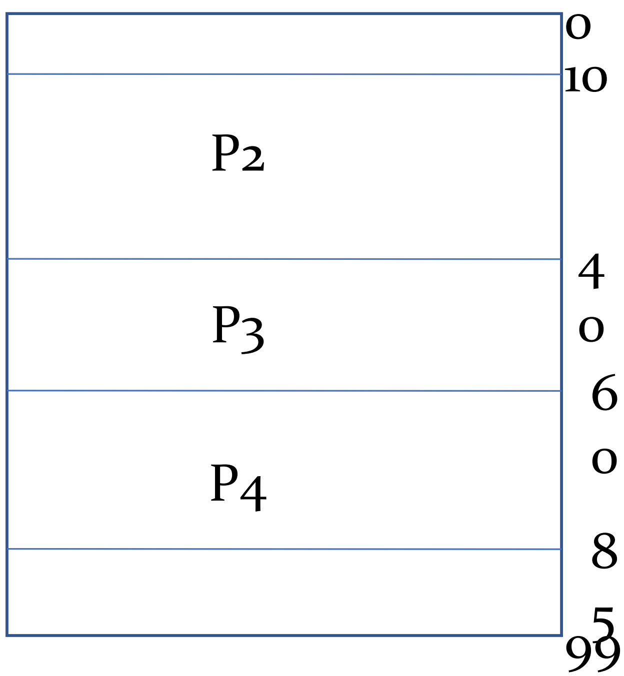
- Process P4 requires 25 Mb space



Memory	Status
0-9	Free
10-39	P2
40-59	P3
60-84	P4
85-99	Free

Variable partition working

- Process P3 finishes its execution.
- So the space is freed

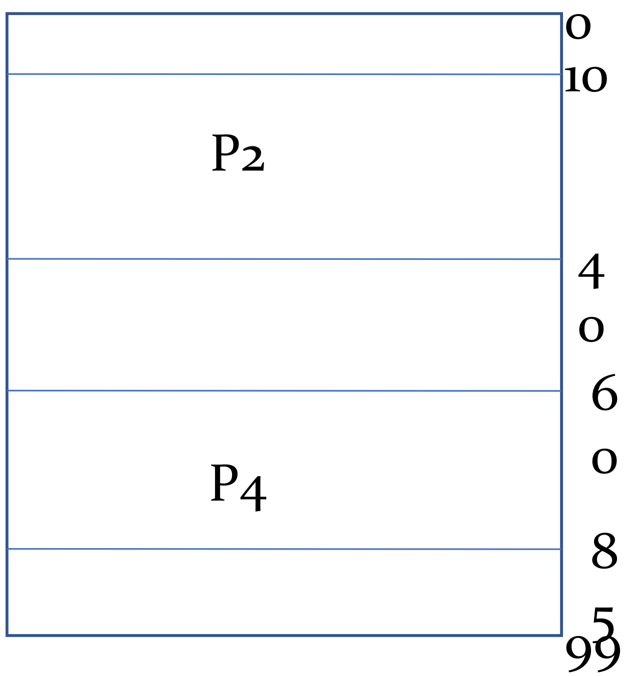


Memory	Status
0-9	Free
10-39	P2
40-59	Free
60-84	P4
85-99	Free

Variable partition working

- Process P5 requires 25 Mb space
- Can it be granted?

No, because 25Mb is not free in a single slot(hole)



Memory	Status
0-9	Free 10 Mb free
10-39	P2
40-59	Free 20 Mb free
60-84	P4
85-100	Free 15 Mb free

Total free space = 10+20+15=45Mb

External Fragmentation

- When there is enough total memory space to satisfy a request, but the available space is not contiguous.
- Solution
  - Compaction/Defragmentation – shuffle the memory contents so as to place all free memory together in one large block.

- **Fragmentation**
  - **External Fragmentation** – total memory space exists to satisfy a request, but it is not contiguous
  - 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 is done at execution time
    - I/O problem
      - **Latch job** in memory while it is involved in I/O
      - Do I/O only into **OS buffers**
  - ***Internal fragmentation*** also occurs, with all memory allocation strategies. This is caused by the fact **that memory is allocated in blocks of a fixed size**, whereas **the actual memory needed** will **rarely be that exact size**.

Thank You

Unit No: 5 Unit Name: Memory Management

Lecture:

Memory Allocation Strategies: Best-Fit, First Fit, Worst Fit, Next Fit, Buddy System, Relocation

Memory Management- Contiguous Memory Allocation

- **Memory Allocation**
  - One method of allocating contiguous memory is to **divide all available memory into equal sized partitions**, and to assign each process to their own partition. This restricts both the **number of simultaneous processes and the maximum size of each process**, and is no longer used.
  - An **alternate approach** is to keep a **list of unused (free) memory blocks (holes),** and to **find a hole of a suitable size** whenever a process needs to be loaded into memory. There are **many different strategies** for finding the "best" **allocation of memory to processes**, including the three most commonly discussed:
    - **First-fit**: → Allocate the **first hole** that is big enough
    - **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

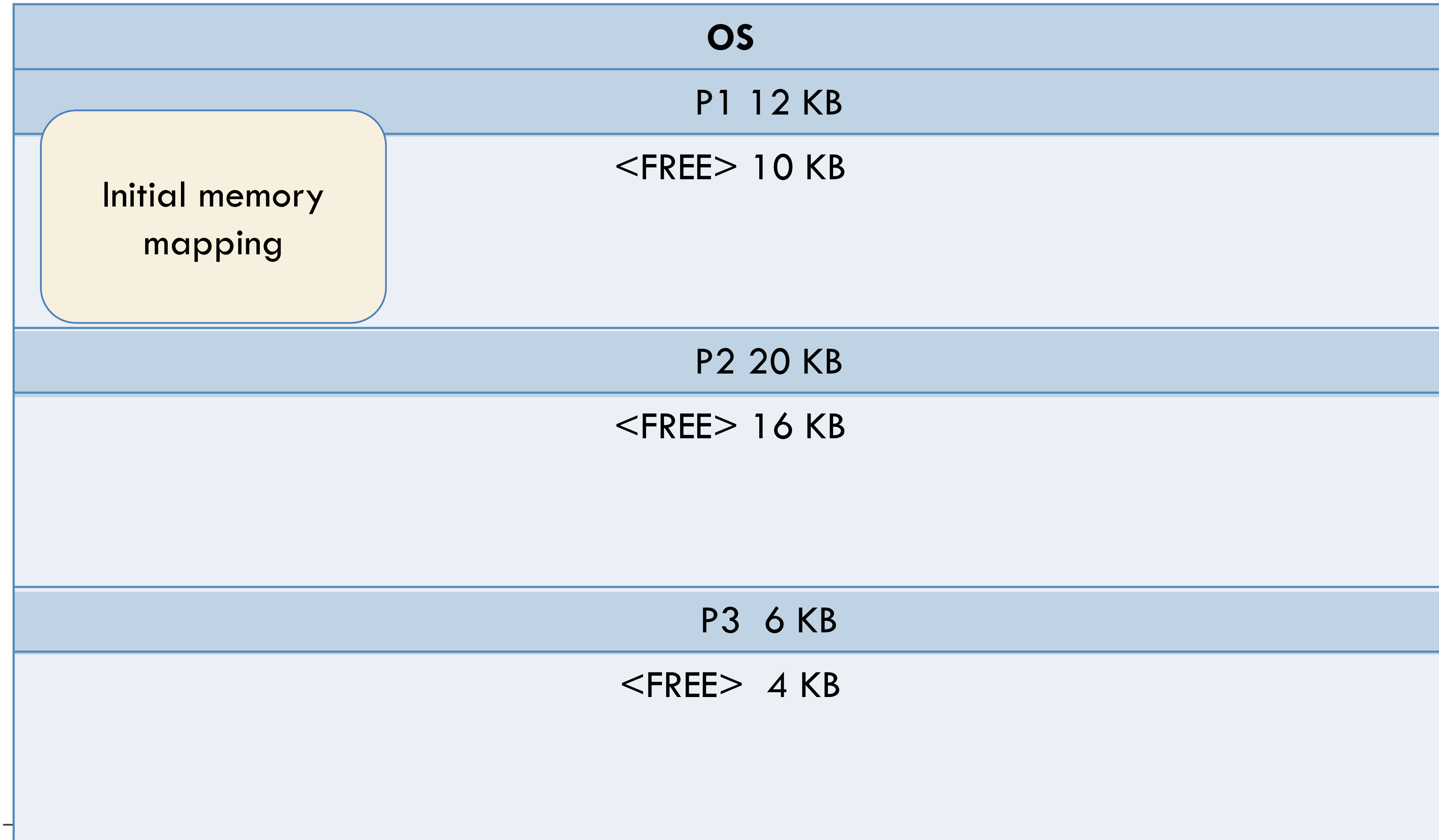
Allocation Strategies

- First Fit
  - Allocate the first spot in memory that is big enough to satisfy the requirements.
- Best Fit
  - Search through all the spots, allocate the spot in memory that most closely matches requirements.
- Next Fit
  - Scan memory from the location of the last placement and choose the next available block that is large enough.
- Worst Fit
  - The largest free block of memory is used for bringing in a process.

First Fit

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

First Fit

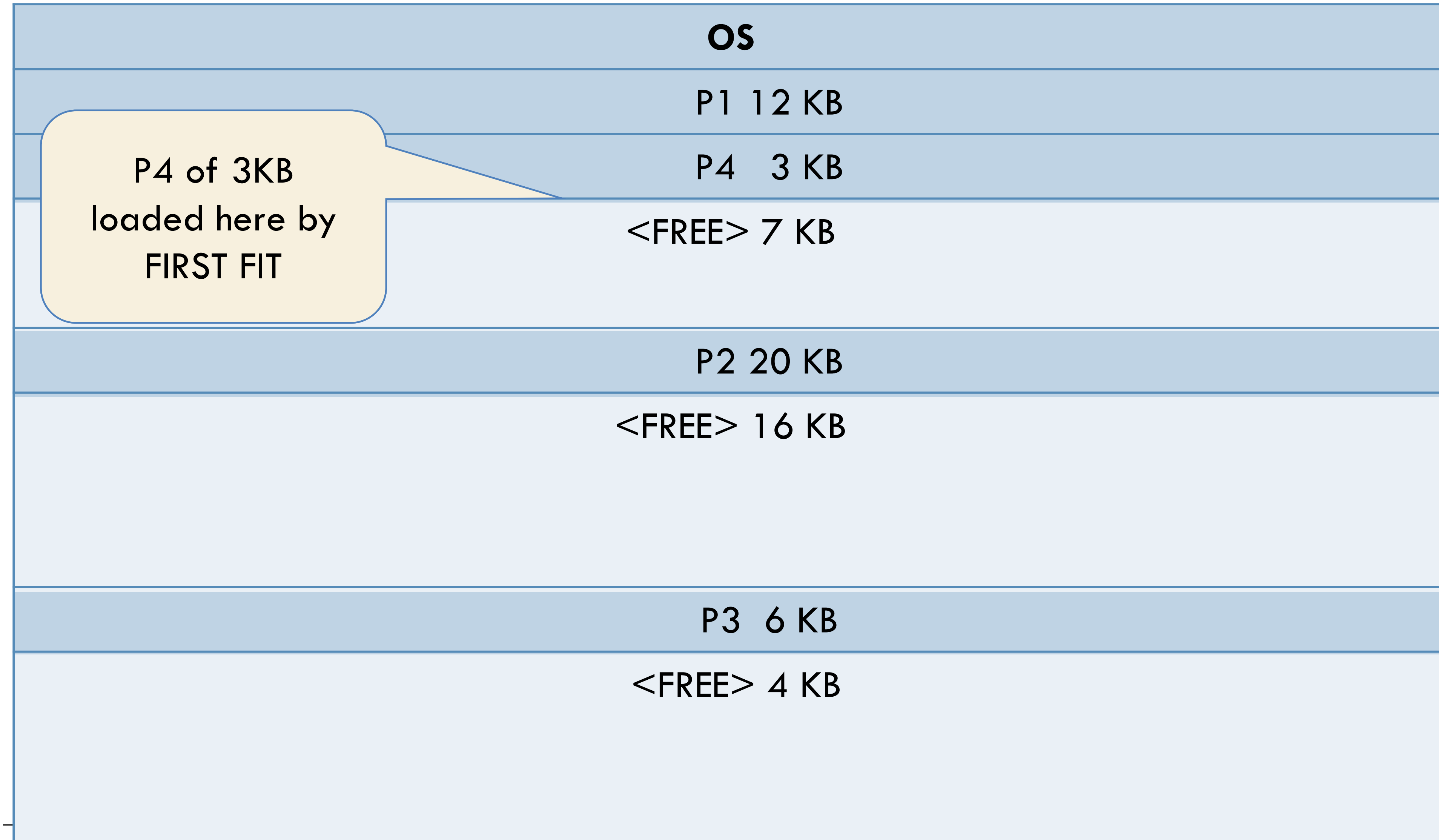

OS
P1 12 KB
Initial memory mapping
<FREE> 10 KB
P2 20 KB
<FREE> 16 KB
P3 6 KB
<FREE> 4 KB

First Fit

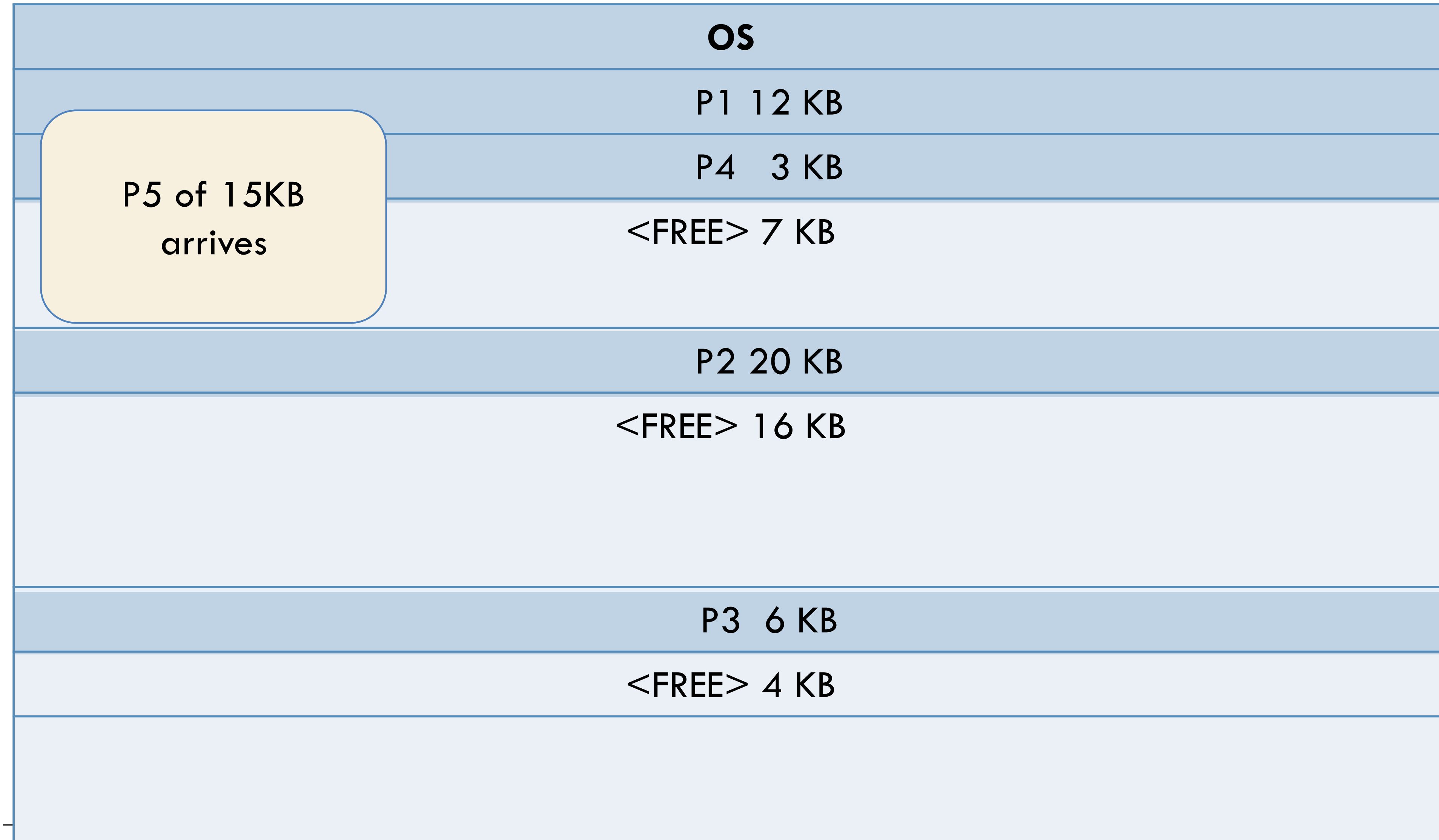
OS
P1 12 KB
<FREE> 10 KB
P2 20 KB
<FREE> 16 KB
P3 6 KB
<FREE> 4 KB

P4 of 3KB arrives

First Fit


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

First Fit


OS
P1 12 KB
P4 3 KB
<FREE> 7 KB
P5 of 15KB arrives
P2 20 KB
<FREE> 16 KB
P3 6 KB
<FREE> 4 KB

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

OS
P1 12 KB
<FREE> 10 KB
P2 20 KB
<FREE> 16 KB
P3 6 KB
<FREE> 4 KB

Initial memory mapping

Best Fit

OS

P1 12 KB

<FREE> 10 KB

P4 of 3KB arrives

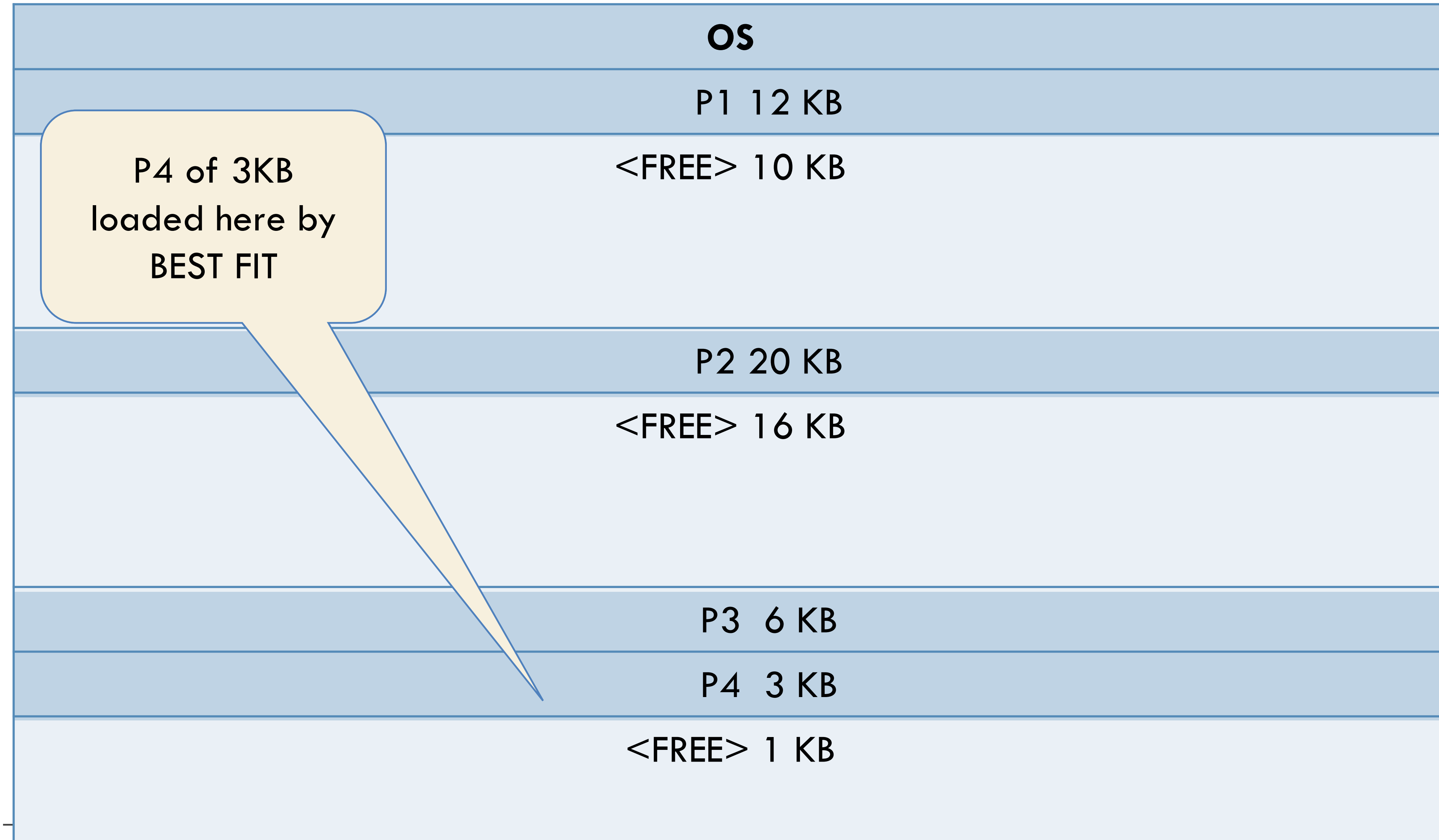
P2 20 KB

<FREE> 16 KB

P3 6 KB

<FREE> 4 KB

Best Fit


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

Best Fit

OS

P1 12 KB

<FREE> 10 KB

P5 of 15KB arrives

P2 20 KB

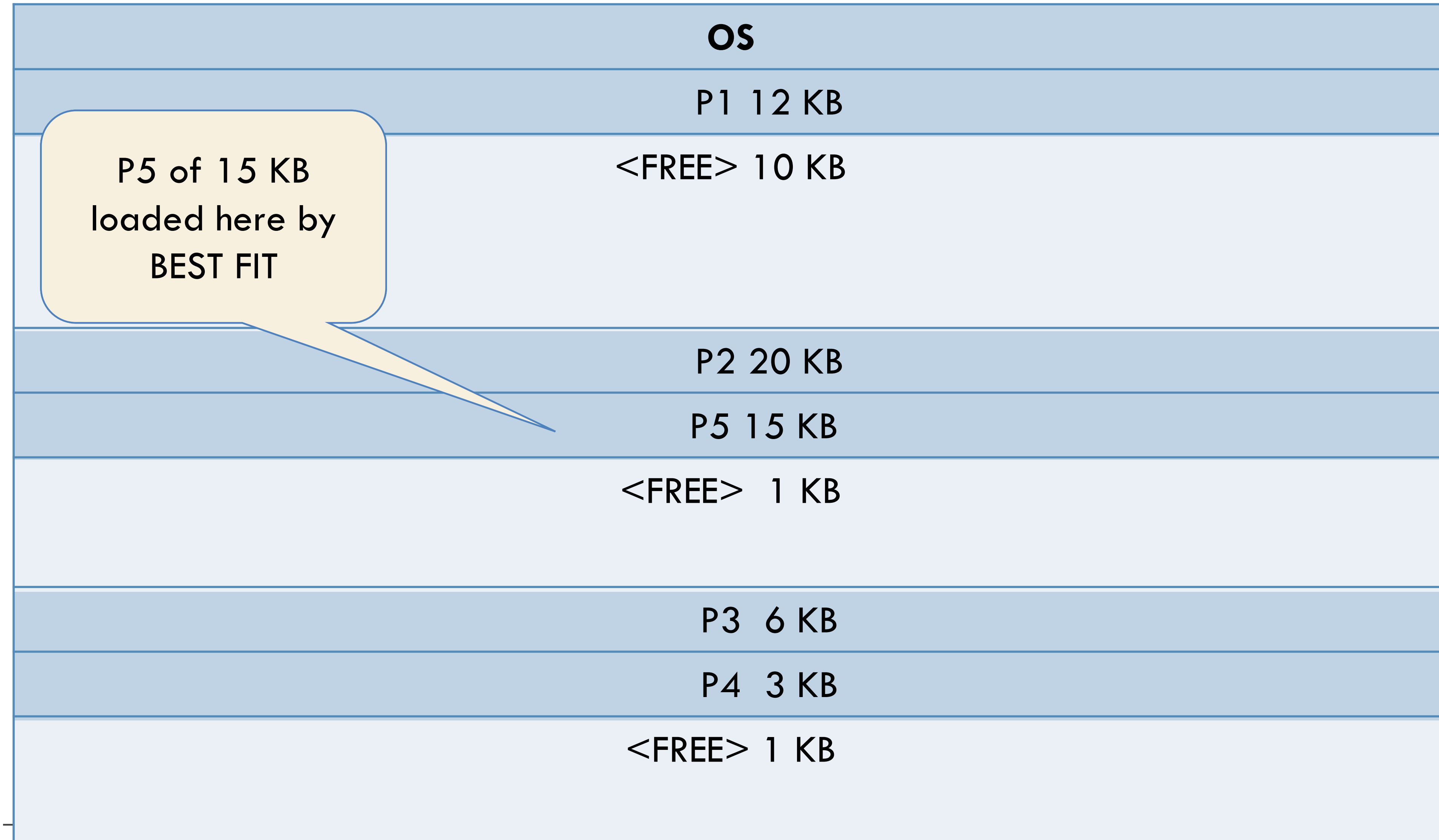
<FREE> 16 KB

P3 6 KB

P4 3 KB

<FREE> 1 KB

Best Fit



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

Worst Fit

- Worst Fit : 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

OS
P1 12 KB
<FREE> 10 KB
P2 20 KB
<FREE> 16 KB
P3 6 KB
<FREE> 4 KB

Initial memory mapping

Worst Fit

OS

P1 12 KB

<FREE> 10 KB

P4 of 3KB arrives

P2 20 KB

<FREE> 16 KB

P3 6 KB

<FREE> 4 KB

Worst Fit

OS
P1 12 KB
<FREE> 10 KB
P2 20 KB
P4 3 KB
<FREE> 13 KB
P3 6 KB
<FREE> 4 KB

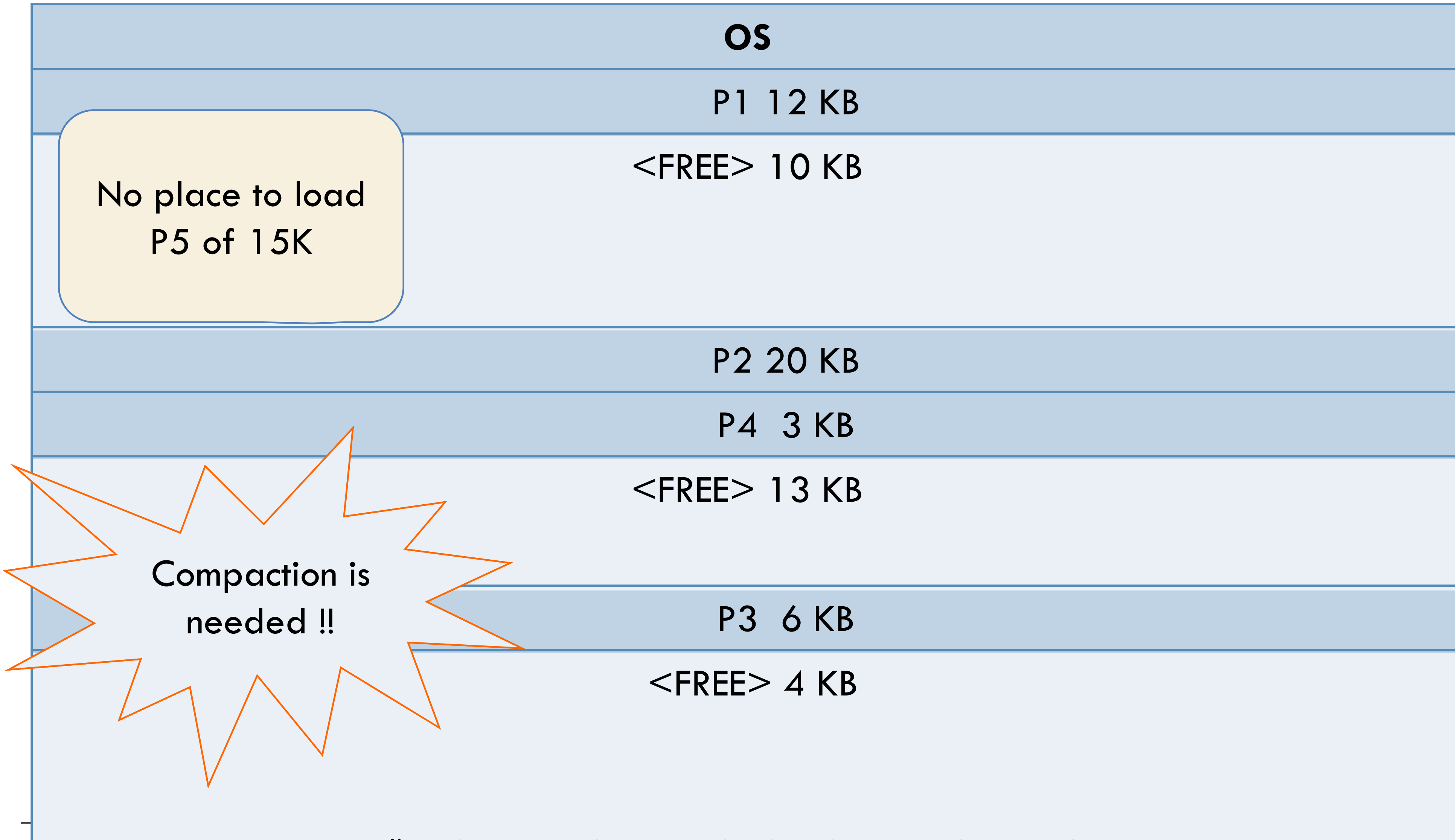
P4 of 3KB
Loaded here by
WORST FIT

Worst Fit

OS
P1 12 KB
<FREE> 10 KB
P2 20 KB
P4 3 KB
<FREE> 13 KB
P3 6 KB
<FREE> 4 KB

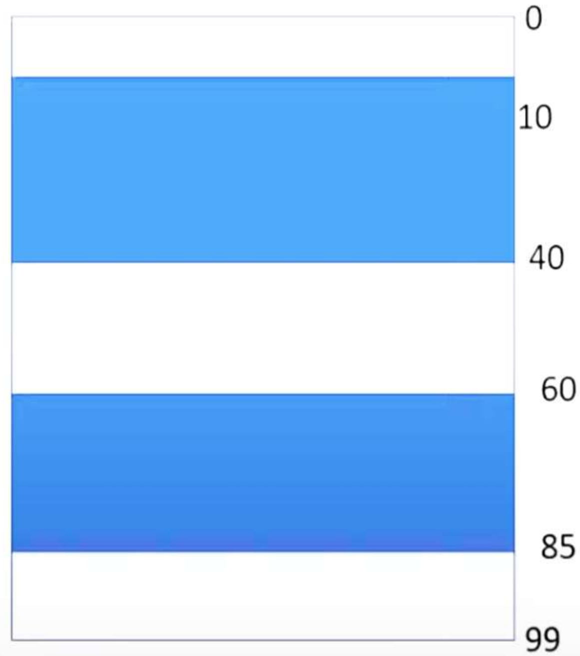
No place to load P5 of 15K

Worst Fit


OS
P1 12 KB
<FREE> 10 KB
No place to load P5 of 15K
P2 20 KB
P4 3 KB
<FREE> 13 KB
Compaction is needed !!
P3 6 KB
<FREE> 4 KB

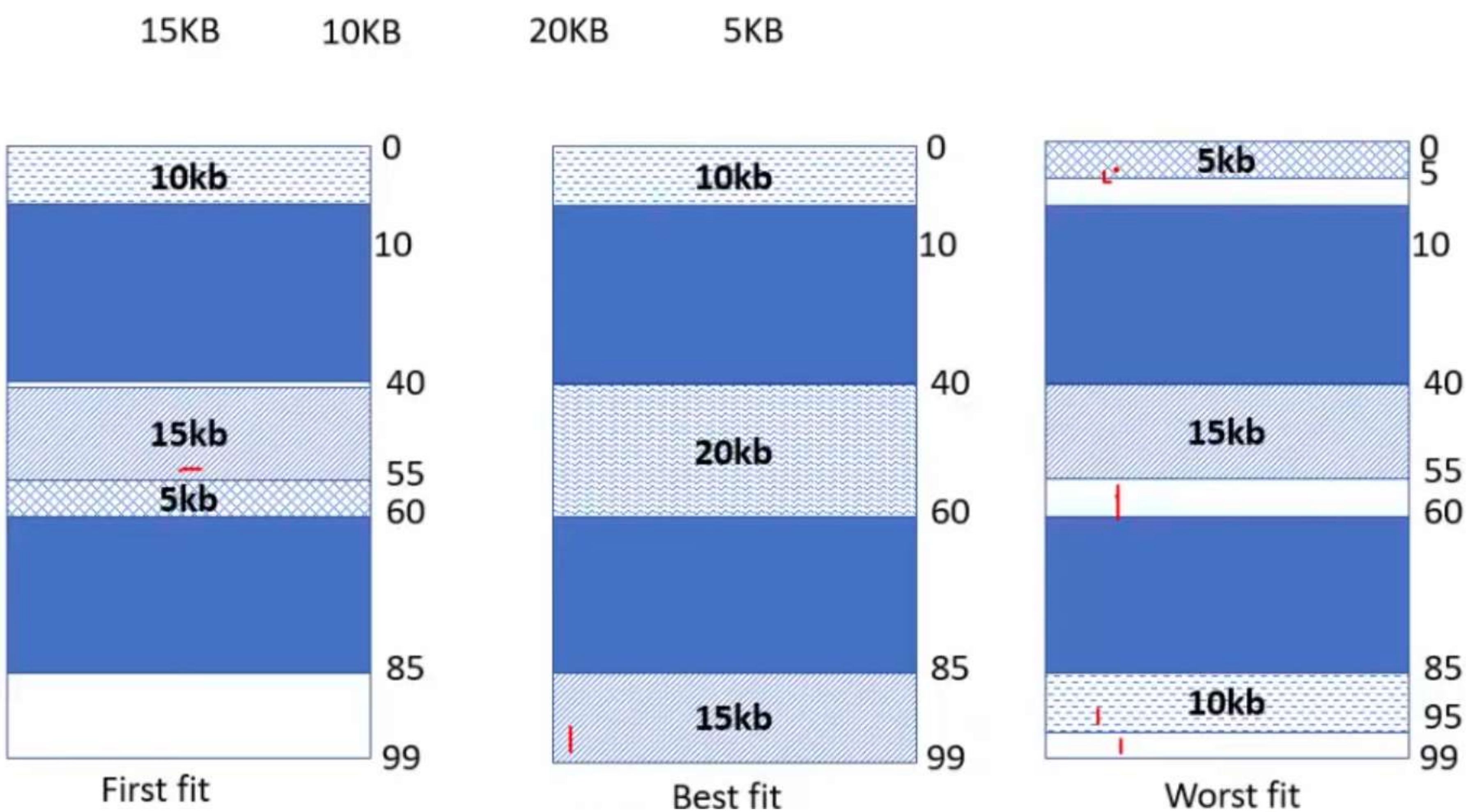
Block(hole) allocation- Example

- Given three free memory partitions of 10 KB, 20 KB and 15 KB(in order), how would each of the first-fit, best-fit, and worst-fit algorithms place processes of 15 KB, 10 KB, 20 KB and 5 KB (in order)?



Block(hole) allocation- Example

- 15 KB, 10 KB, 20 KB and 5 KB (in order)

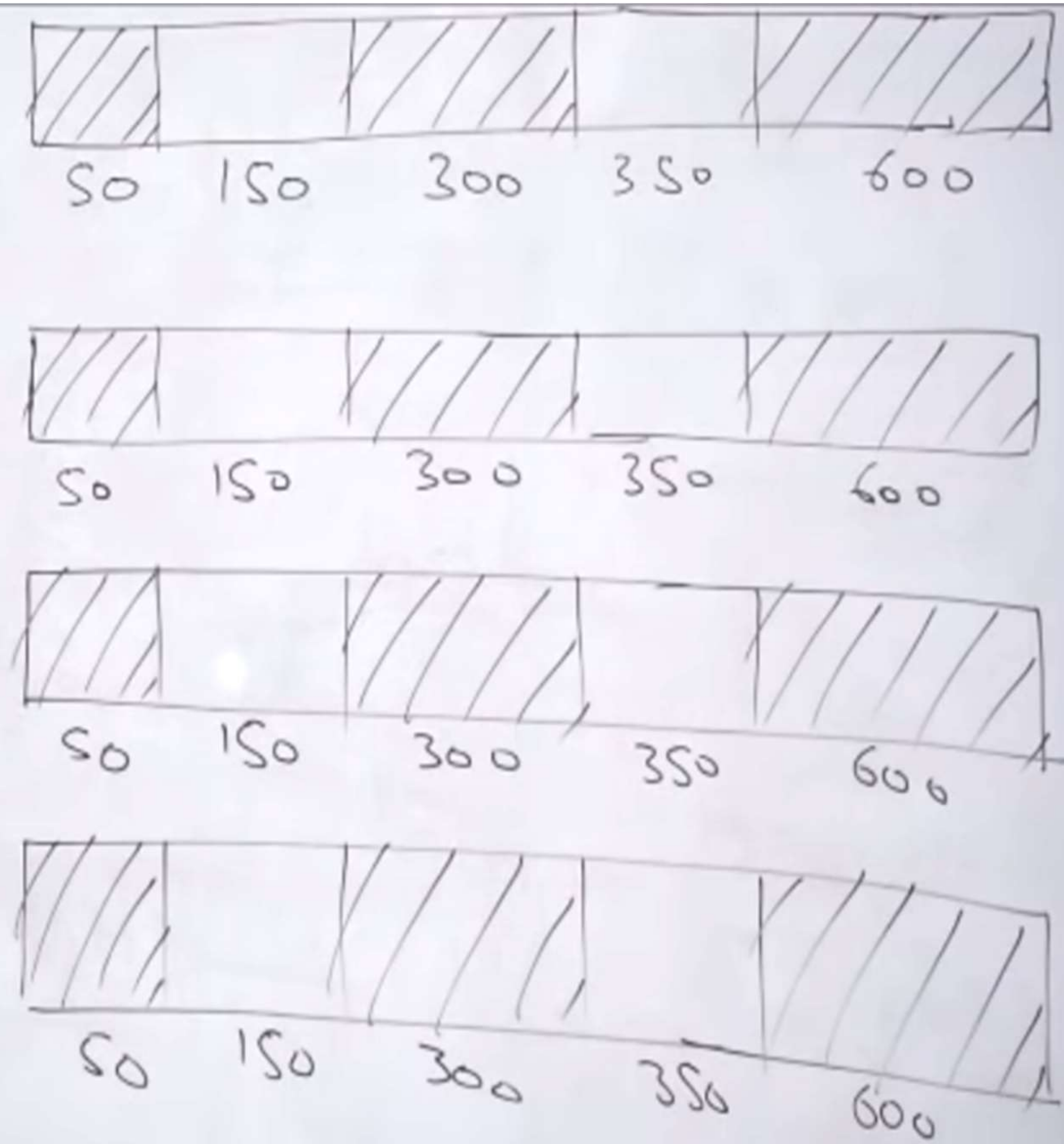


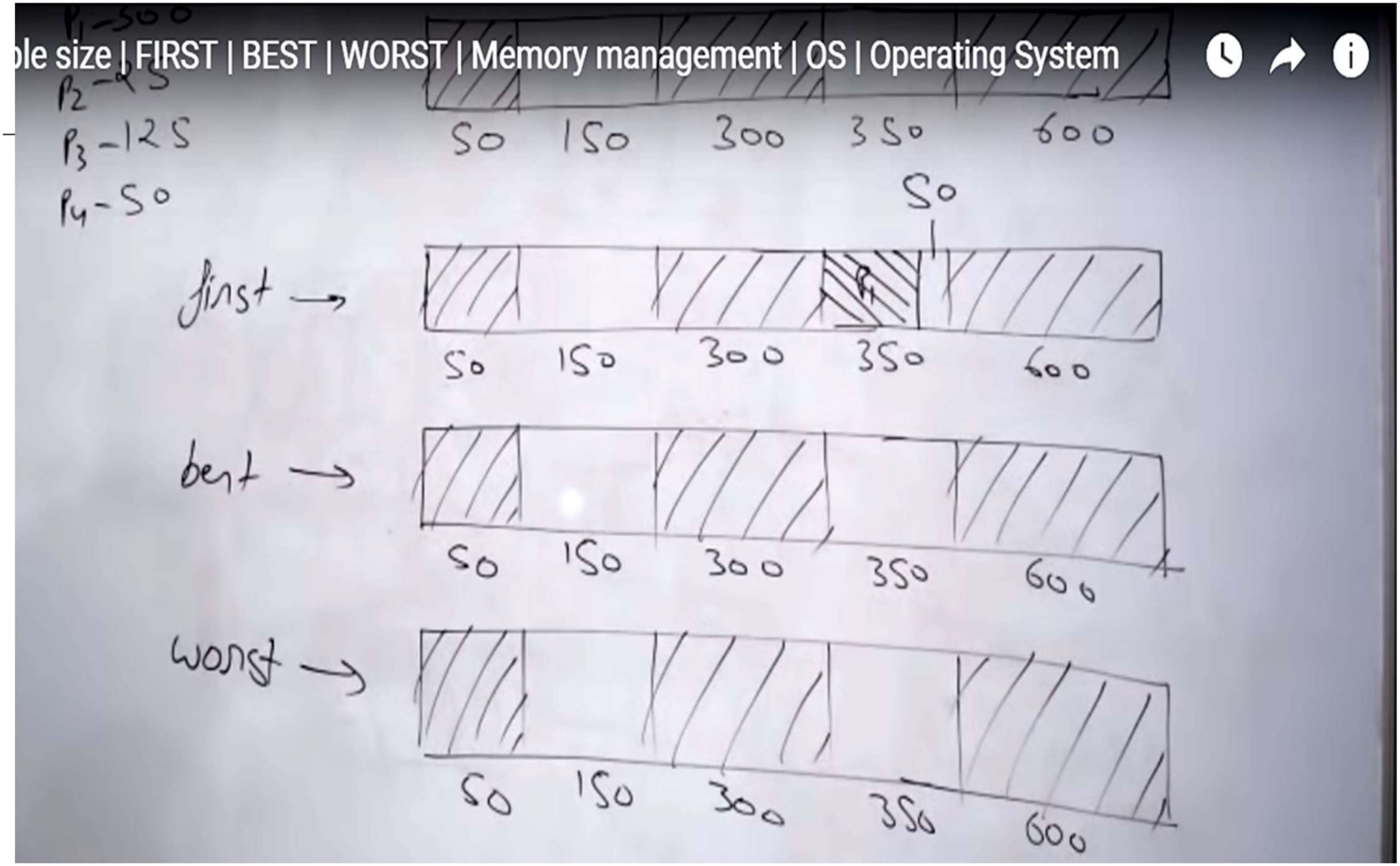
$P_1 - 300$

$P_2 - 25$

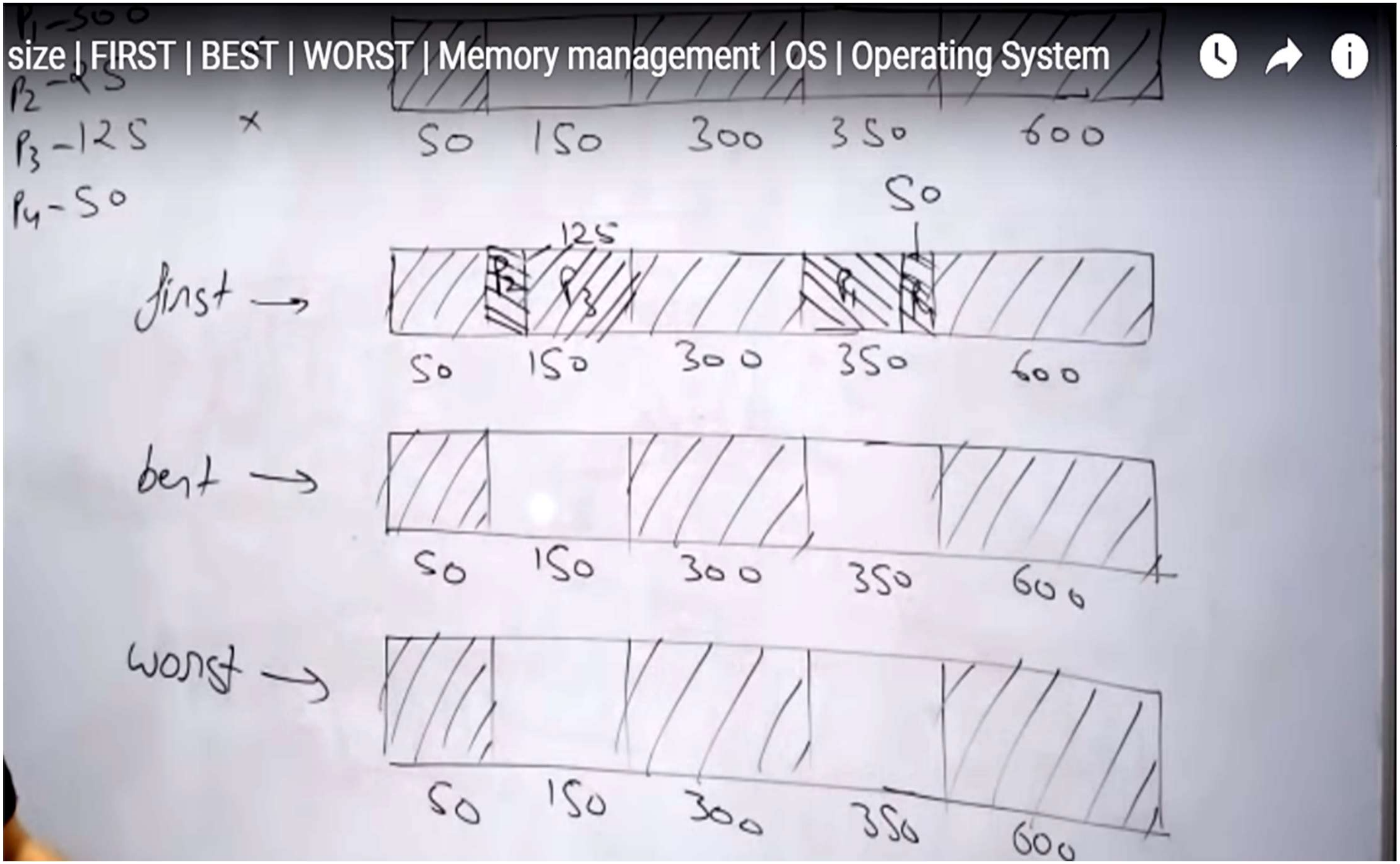
$P_3 - 125$

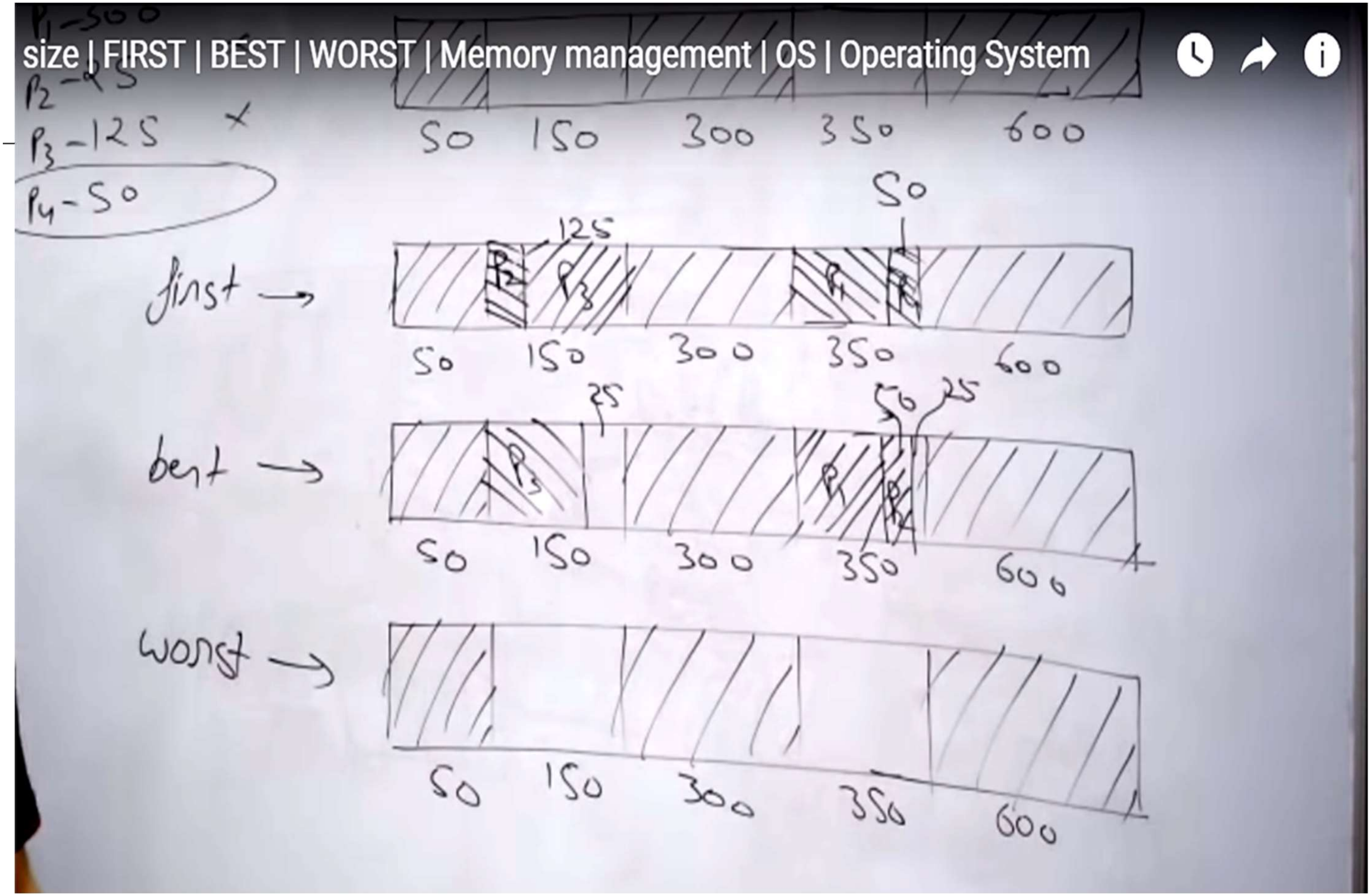
$P_4 - 50$



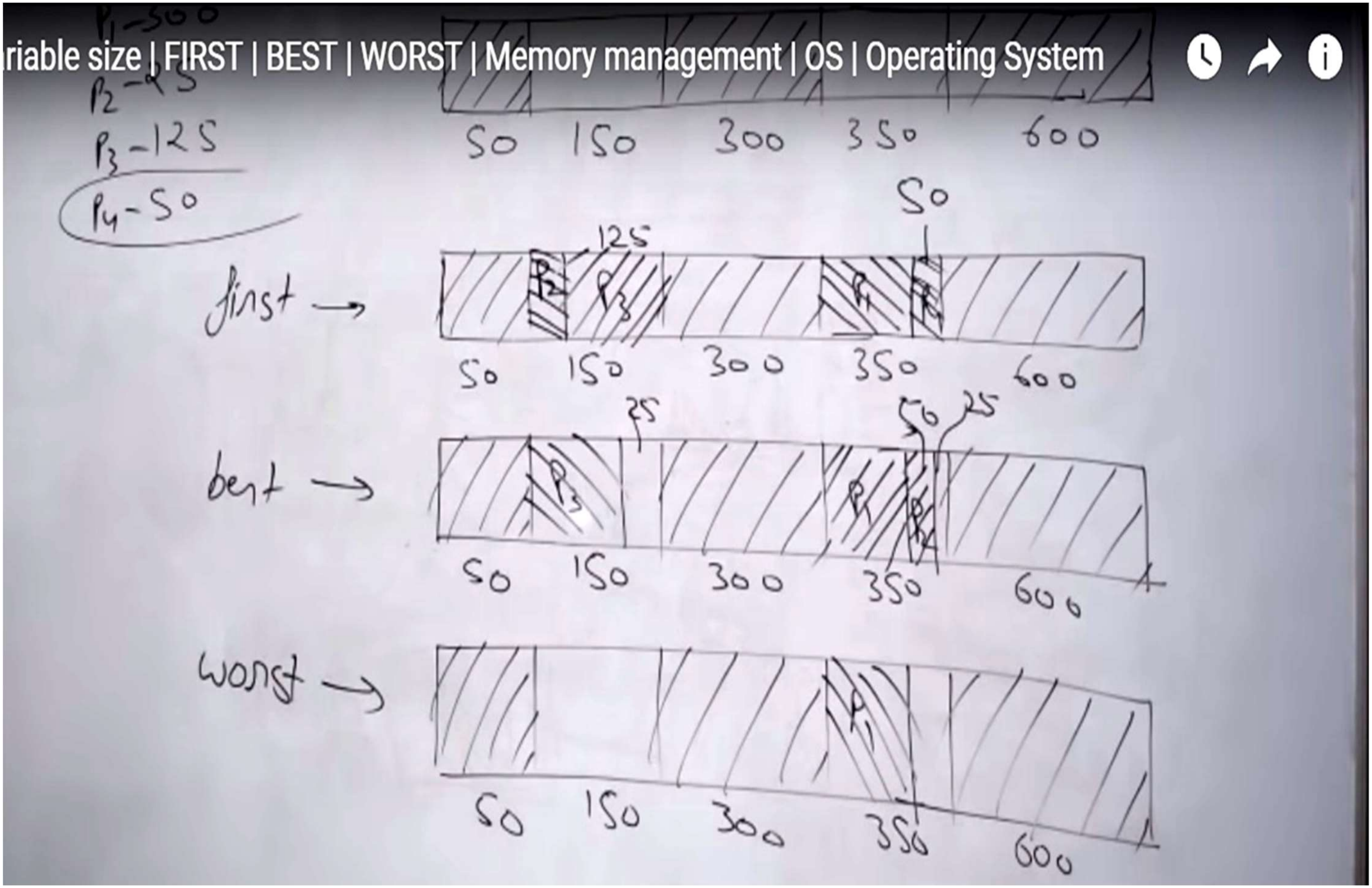


ble size | FIRST | BEST | WORST | Memory management | OS | Operating System
P_1-300
P_2-25
P_3-125
P_4-50
50
150
300
350
600
first →
50
P_1
best →
worst →


size | FIRST | BEST | WORST | Memory management | OS | Operating System
P1 - 500
P2 - 25
P3 - 125
P4 - 50
50 150 300 350 600
first →
125
50
P2 P3 P1 P4
50 150 300 350 600
best →
50 150 300 350 600
worst →
50 150 300 350 600



size | FIRST | BEST | WORST | Memory management | OS | Operating System
P3 - 125
P4 - 50
first →
50 150 300 350 600
best →
50 150 300 350 600
worst →
50 150 300 350 600

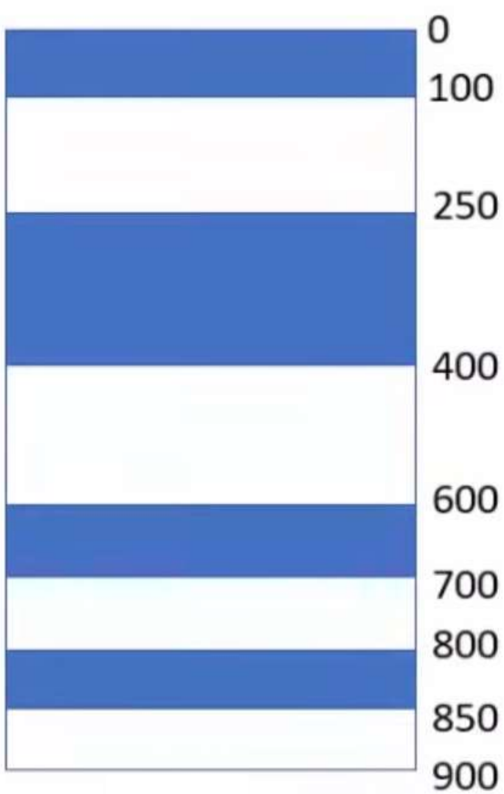

riable size | FIRST | BEST | WORST | Memory management | OS | Operating System
P1-300
P2-25
P3-125
P4-50
50 150 300 350 600
first →
best →
worst →
125
25
50
25
P1
P2
P3
P4

Practice Problem

- Given four free memory partitions of 150 KB, 200 KB , 100 KB and 50 KB(in order), how would each of the first-fit, best-fit, and worst-fit algorithms place processes of 150 KB, 100 KB, 200 KB and 50 KB (in order)? Which algorithm performs the best?

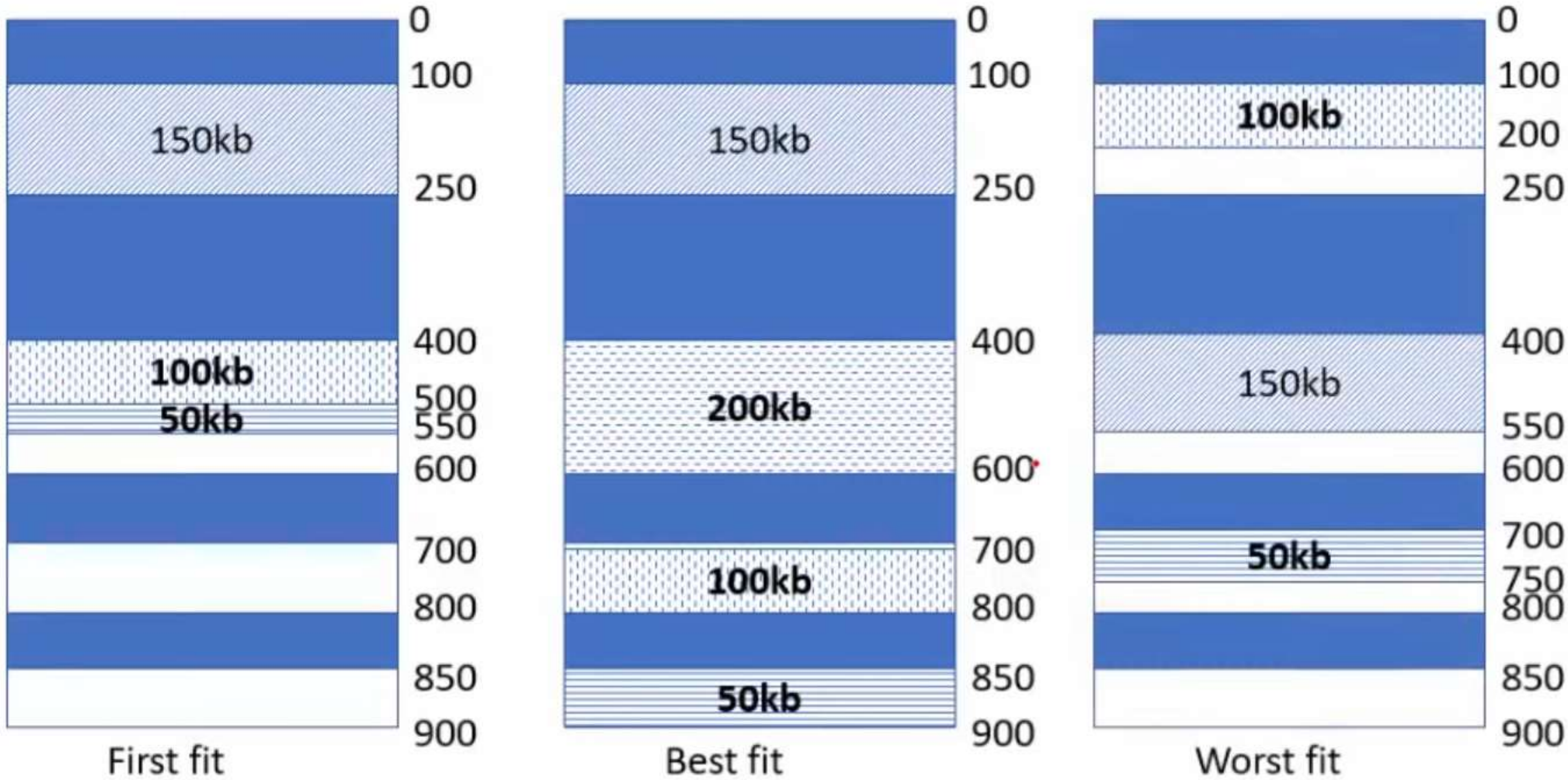
Practice Problem

- Given four free memory partitions of 150 KB, 200 KB , 100 KB and 50 KB(in order), how would each of the first-fit, best-fit, and worst-fit algorithms place processes of 150 KB, 100 KB, 200 KB and 50 KB (in order)? Which algorithm performs the best?



Practice Problem

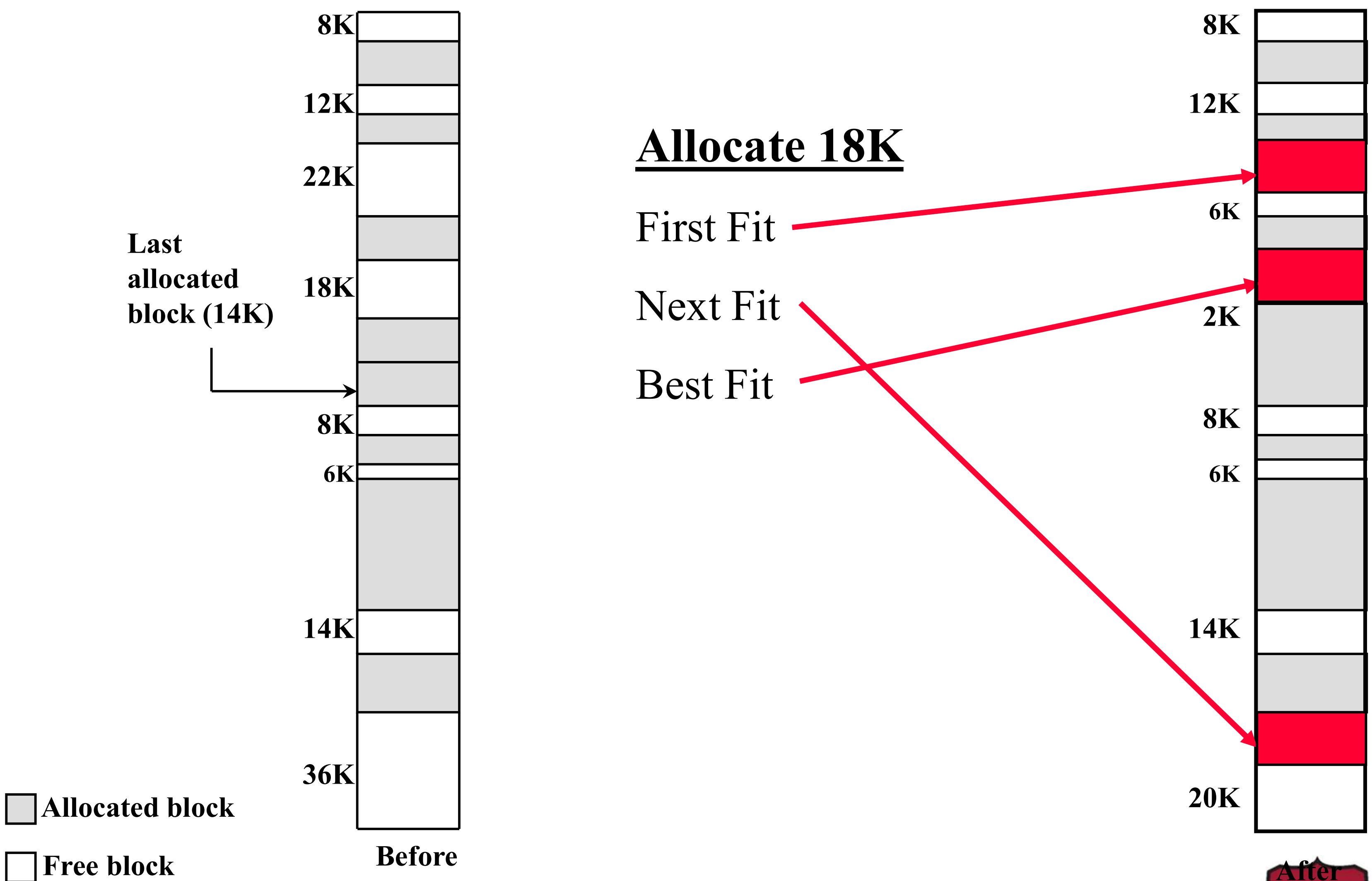
- Given four free memory partitions of 150 KB, 200 KB , 100 KB and 50 KB(in order), how would each of the first-fit, best-fit, and worst-fit algorithms place processes of 150 KB, 100 KB, 200 KB and 50 KB (in order)? Which algorithm performs the best?



Which Allocation Strategy?

- The first-fit algorithm is not only the simplest but usually the best and the fastest as well.
  - May litter the front end with small free partitions that must be searched over on subsequent first-fit passes.
- The next-fit algorithm will more frequently lead to an allocation from a free block at the end of memory.
  - Results in fragmenting the largest block of free memory.
  - Compaction may be required more frequently.
- Best-fit is usually the worst performer.
  - Guarantees the fragment left behind is as small as possible.
  - Main memory quickly littered by blocks too small to satisfy memory allocation requests.

Dynamic Partitioning Placement Algorithm



Memory Fragmentation

- As memory is allocated and deallocated fragmentation occurs
- External -
  - Enough space exists to launch a program, but it is not contiguous
- Internal -
  - Allocate more memory than asked for to avoid having very small holes

Memory Fragmentation

- Statistical analysis shows that given *N* allocated blocks, another 0.5 *N* blocks will be lost due to fragmentation.
  - On average, 1/3 of memory is unusable
    - (50-percent rule)
- Solution – Compaction.
  - Move allocated memory blocks so they are contiguous
  - Run compaction algorithm periodically
    - How often?
    - When to schedule?

Compaction

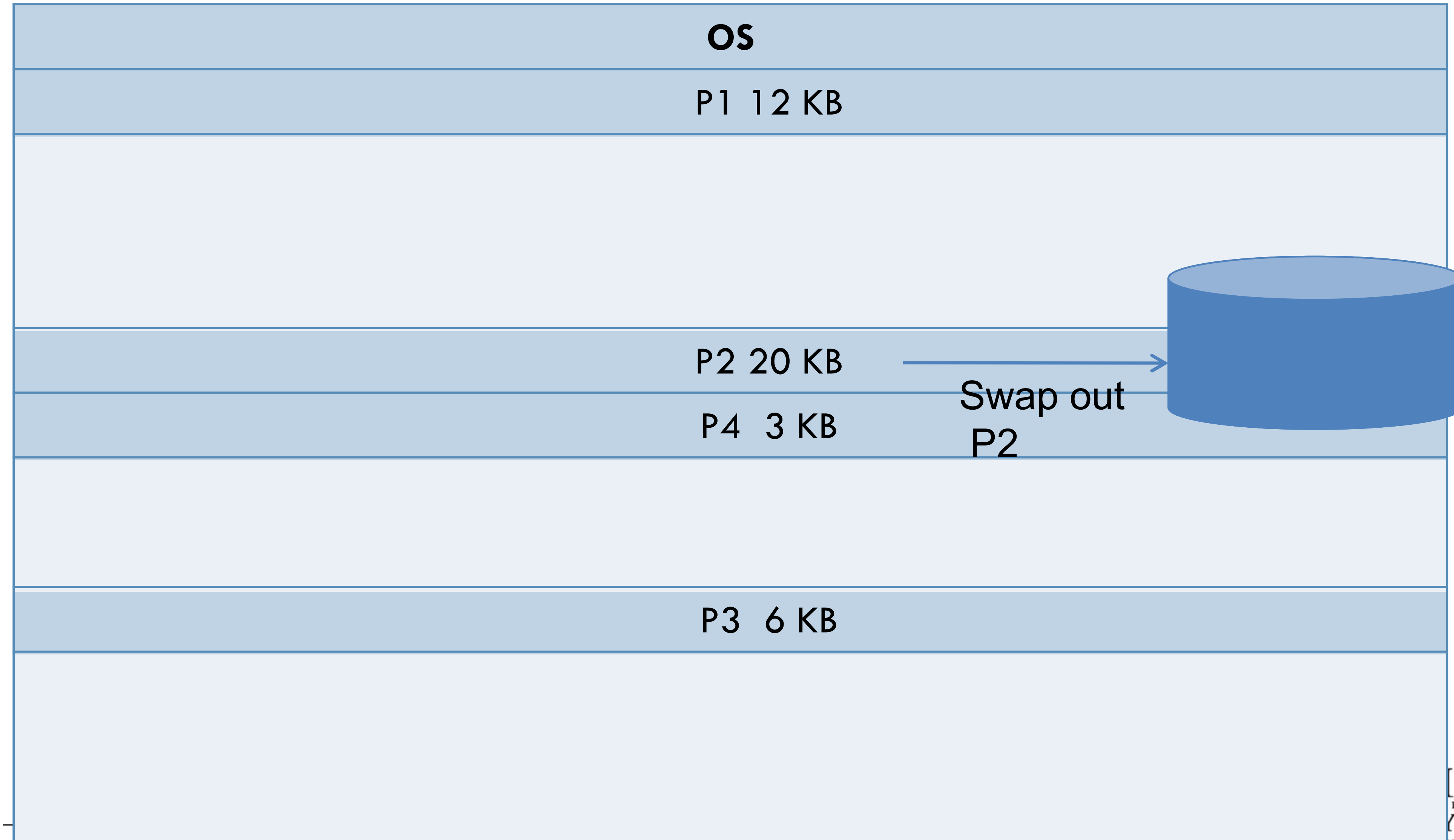
- Compaction is a method to overcome the external fragmentation problem.
- All free blocks are brought together as one large block of free space.
- Compaction requires dynamic relocation.
- Certainly, compaction has a cost and selection of an optimal compaction strategy is difficult.
- One method for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

Compaction

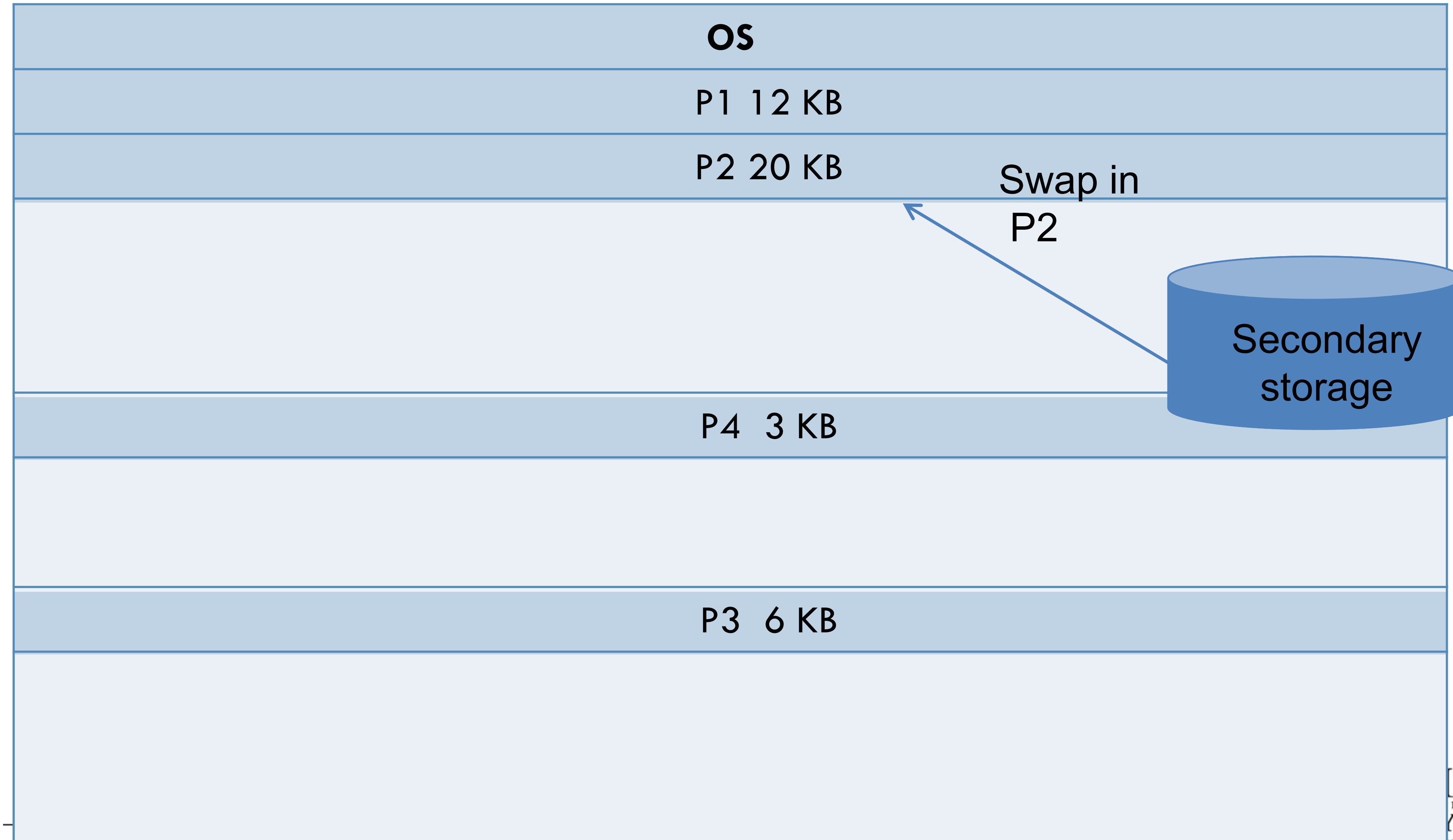
OS
P1 12 KB
<FREE> 10 KB
P2 20 KB
P4 3 KB
<FREE> 13 KB
P3 6 KB
<FREE> 4 KB

Memory mapping before compaction

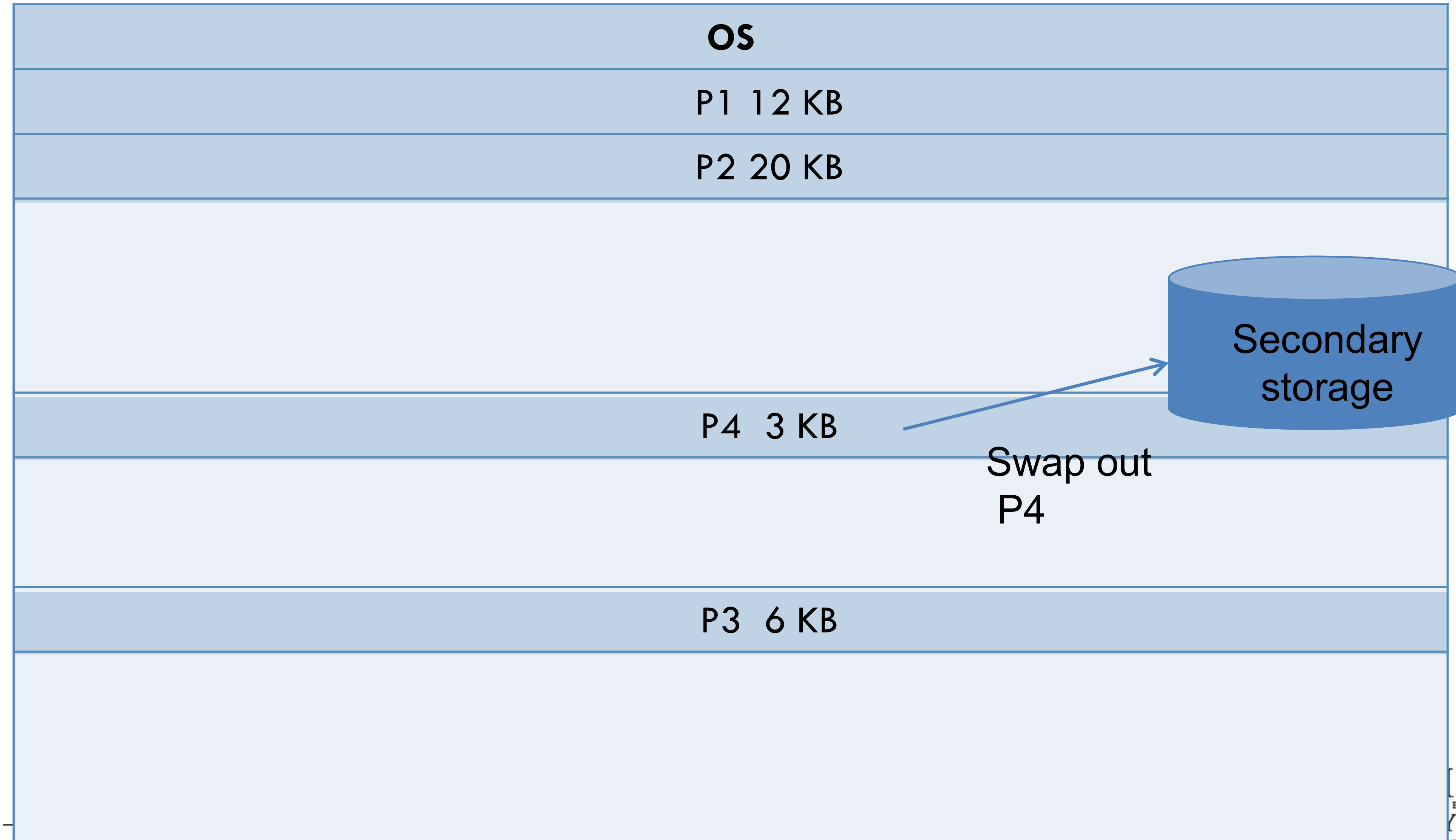
Compaction



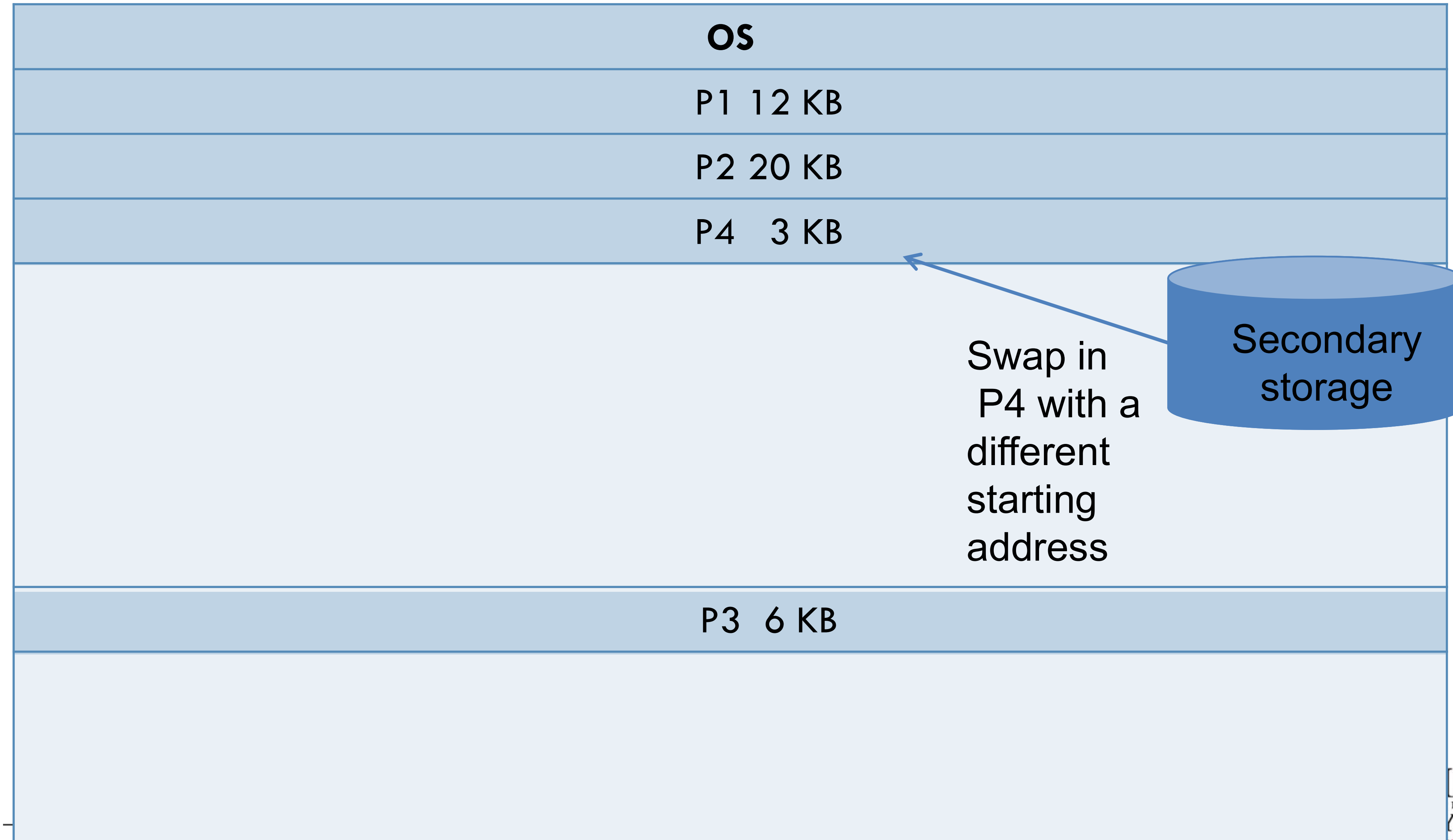
Compaction



Compaction

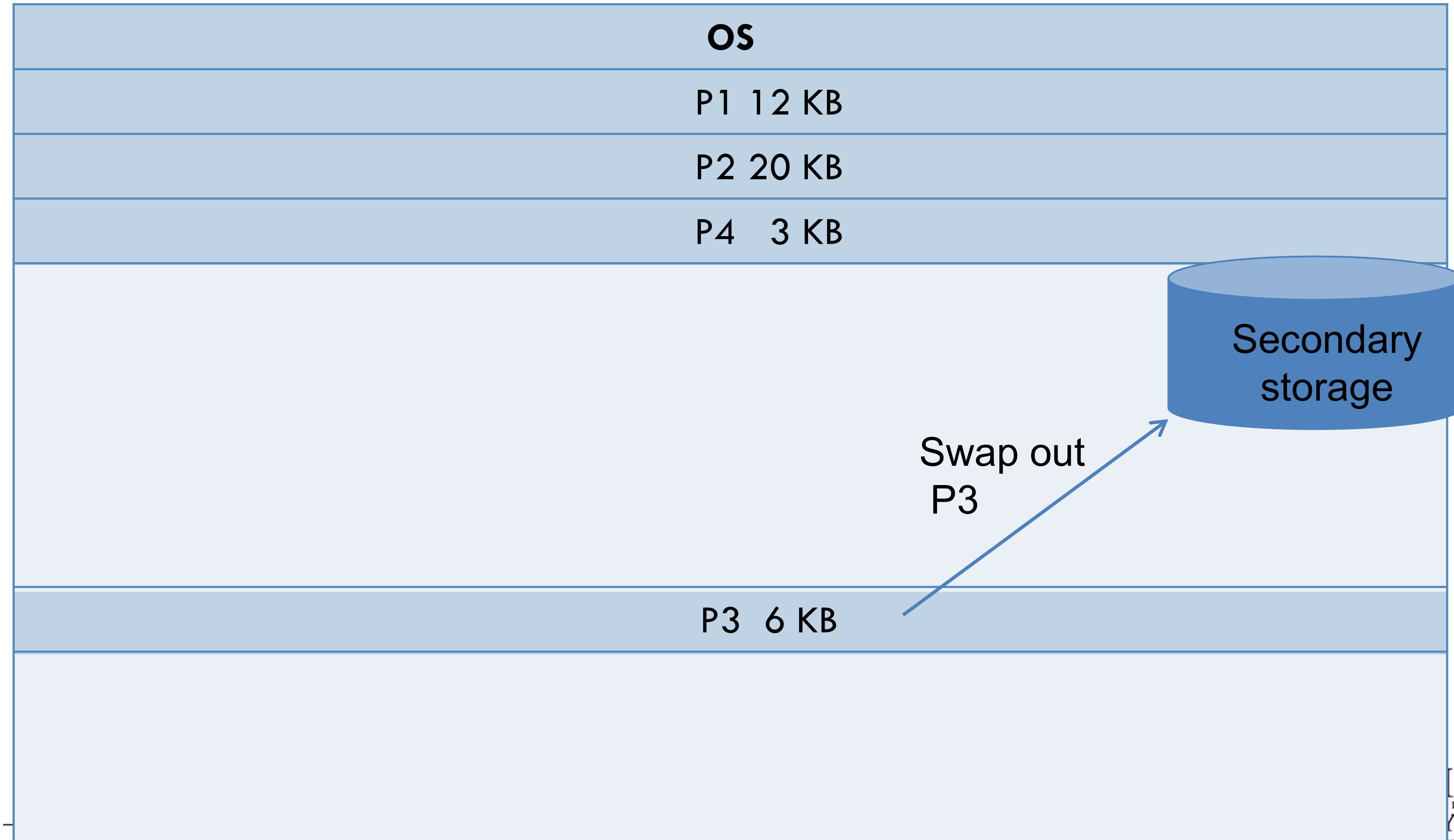


Compaction

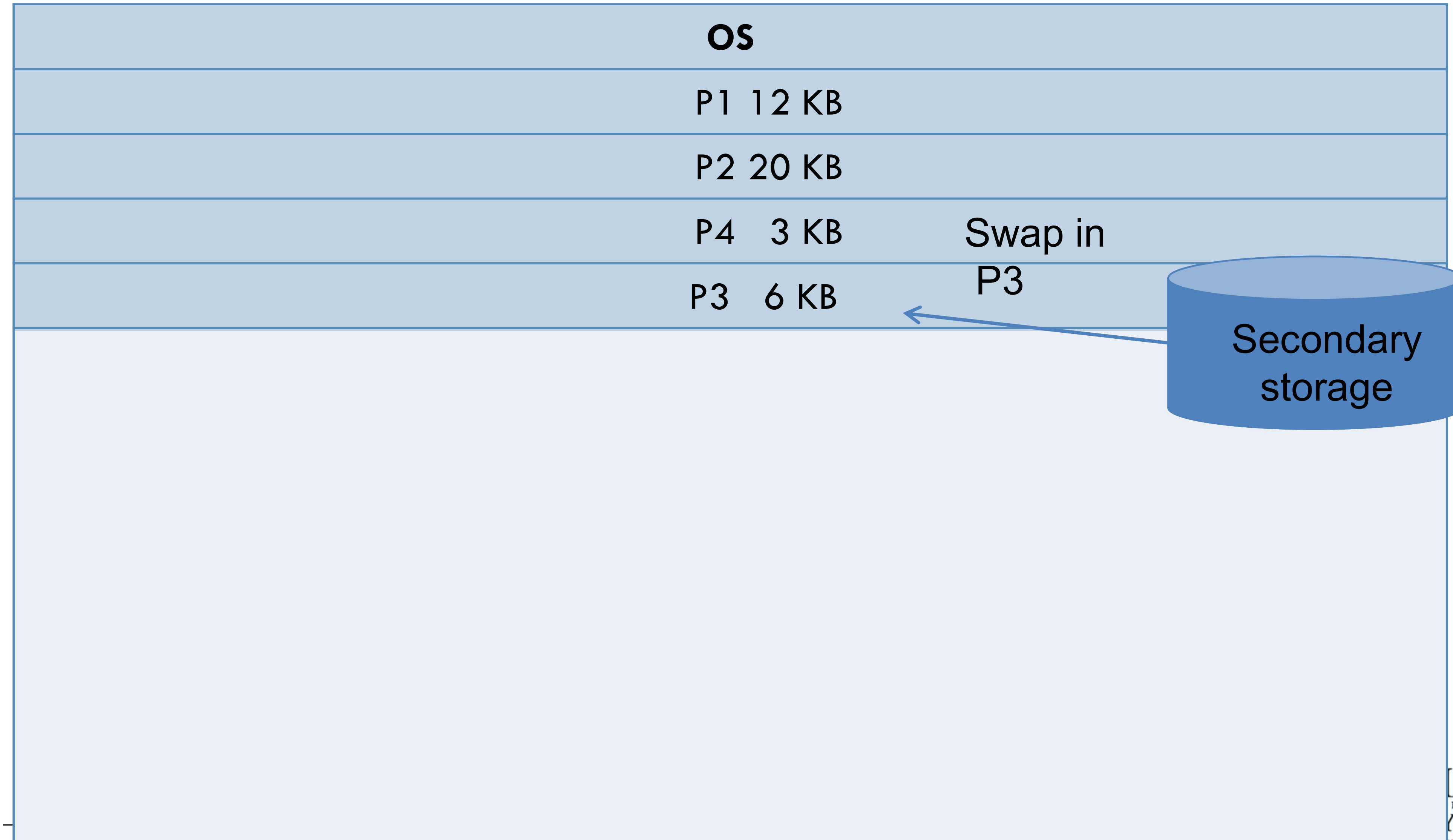


OS
P1 12 KB
P2 20 KB
P4 3 KB
Secondary storage
Swap in P4 with a different starting address
P3 6 KB

Compaction



Compaction



Compaction

OS
P1 12 KB
P2 20 KB
P4 3 KB
P3 6 KB
<FREE> 27 KB

Memory mapping after compaction

Now P5 of 15KB can be loaded here

Compaction

OS
P1 12 KB
P2 20 KB
P4 3 KB
P3 6 KB
P5 12 KB
<FREE> 12 KB

P5 of 15KB is loaded

Relocation

- **Static relocation:** A process may be loaded into memory, each time possibly having a different starting address
    - Necessary for variable partitioning
- **Dynamic relocation:** In addition to static relocation, the starting address of the process may change while it is already loaded in memory
    - Necessary for compaction

Buddy System

- Tries to allow a variety of block sizes while avoiding excess fragmentation
- Blocks generally are of size 2^k, for a suitable range of k
- Initially, all memory is one block
- All sizes are rounded up to 2^s
- If a block of size 2^s is available, allocate it
- Else find a block of size 2^{s+1} and split it in half to create two buddies
- If two buddies are both free, combine them into a larger block
- Largely replaced by paging
    - Seen in parallel systems and Unix kernel memory allocation

Thank You