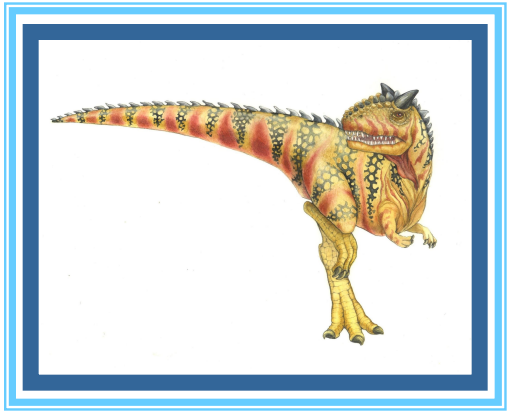
Chapter 8: Main Memory

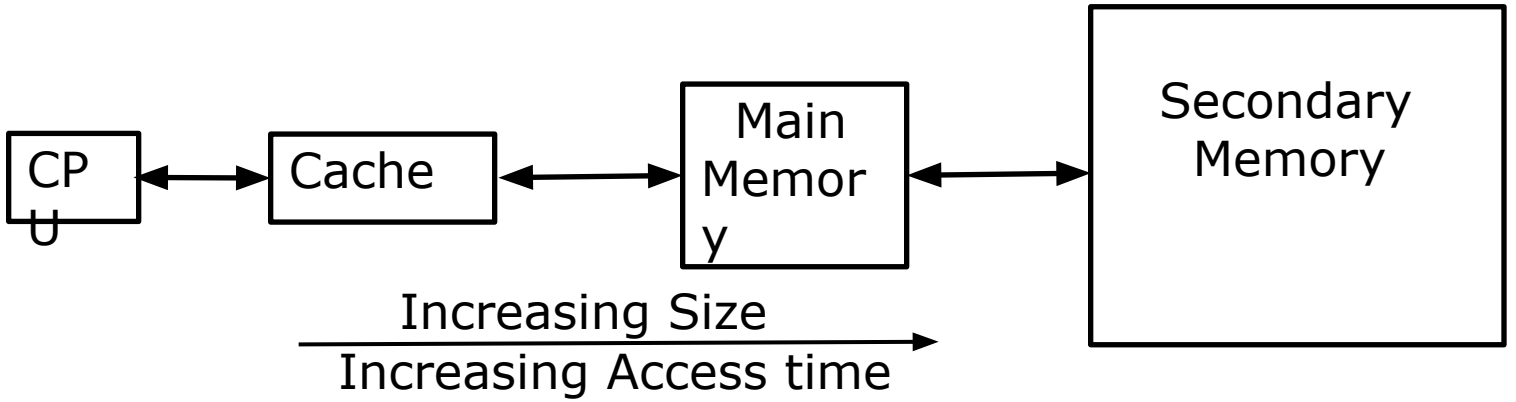


Objectives

- To provide a detailed description of various ways of organizing memory hardware
- Address Binding
  - Base and Limit registers
  - Hardware Address Protection
- Logical and Physical addresses
- Dynamic Relocation (MMU)
- Dynamic Linking
- Swapping

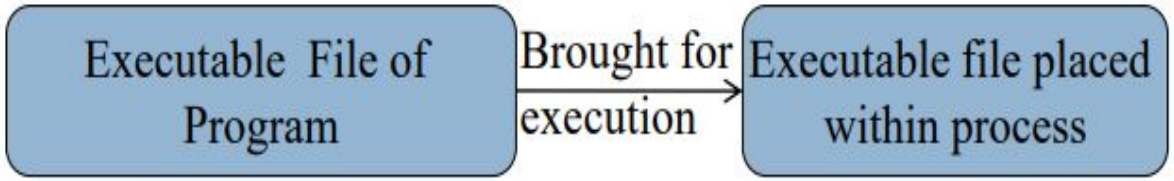
Background

- Program must be brought (from disk) into memory and placed within a process for it to be run
- 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



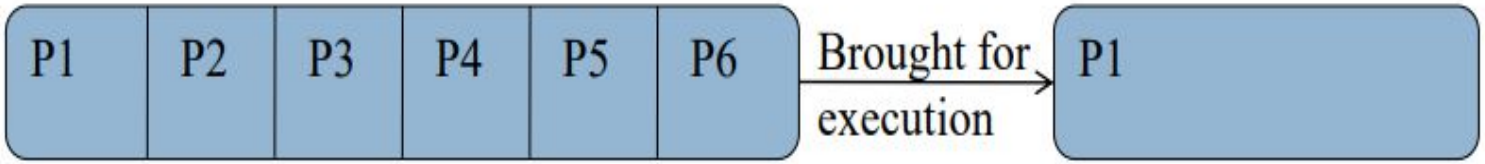
Address Binding

- Process moves between memory and disk during execution



Hard Disk

Main Memory



Address Binding

- Input queue – collection of processes on the disk that are waiting to be brought into memory to run the program
- One process selected—brought to memory
- Process executes—Access instruction and data
- Process terminate—memory acquired get released.
- Most systems allow a user process to reside in any part of the physical memory.
- Thus, address space of the computer starts at 00000, first address of the user process need not be 00000

Process Address Space

- The process address space is the set of logical addresses that a process references in its code. For example, when 32-bit addressing is in use, addresses can range from 0 to 0x7fffffff; that is, 2^31 possible numbers, for a total theoretical size of 2 gigabytes.
- The operating system takes care of mapping the logical addresses to physical addresses at the time of memory allocation to the program. There are three types of addresses used in a program before and after memory is allocated –

S.N.	Memory Addresses & Description
1	**Symbolic Addresses** The addresses used in a source code. The variable names, constants, and instruction labels are the basic elements of the symbolic address space.
2	**Relative Addresses/ Relocatable Addresses** At the time of compilation, a compiler converts symbolic addresses into relative addresses.
3	**Physical Addresses/ Absolute Addresses** The loader generates these addresses at the time when a program is loaded into main memory.

Binding of Instructions and Data to Memory

- Address binding of instructions and data to memory addresses can happen at three different stages
  - **Compile time**
  - **Load time**
  - **Execution time**

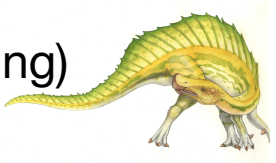
Binding of Instructions and Data to Memory

Compile Time

- If it is known in advance that where the process will be placed in memory then absolute code can be generated at the time of compilation.
- If we know in advance that a user process may store at starting from location R and that the generated code will start at that location and extend up from there.
- If at some later time starting location changes then it will be necessary to recompile this code.
- Ms-dos based dot com format programs are absolute code and bound at compile time.
- Traveling in a train, seat # 3, Bogi #2

Load Time:

- If it is not known at compile time that at which particular location the process will reside in memory when the compiler will generate relocatable code to find the address.
- In this case final binding is delayed until load time.
- In this case absolute address will be generated by the loader at the load time
- Example Traveling by air, Seat # is allocated during boarding (loading)



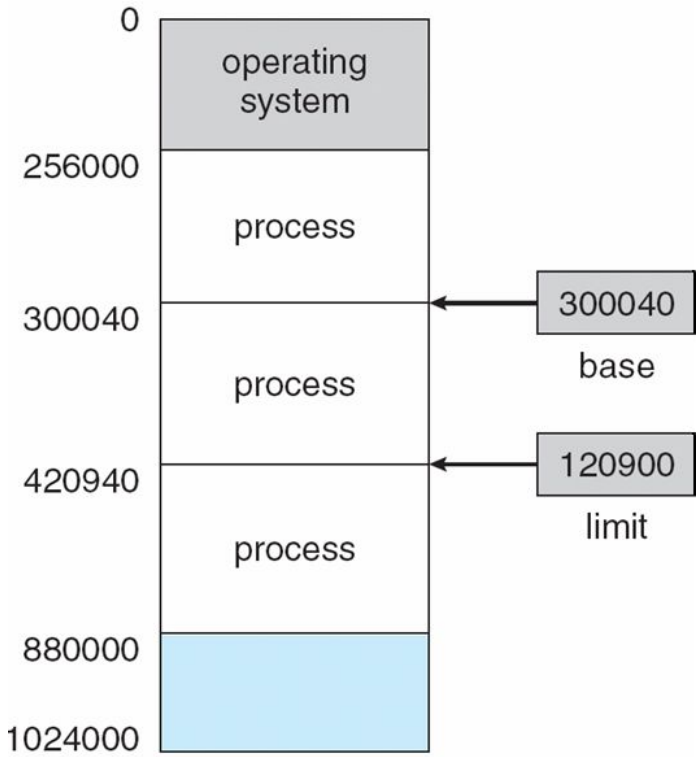
Binding of Instructions and Data to Memory

Execution Time (Run Time):

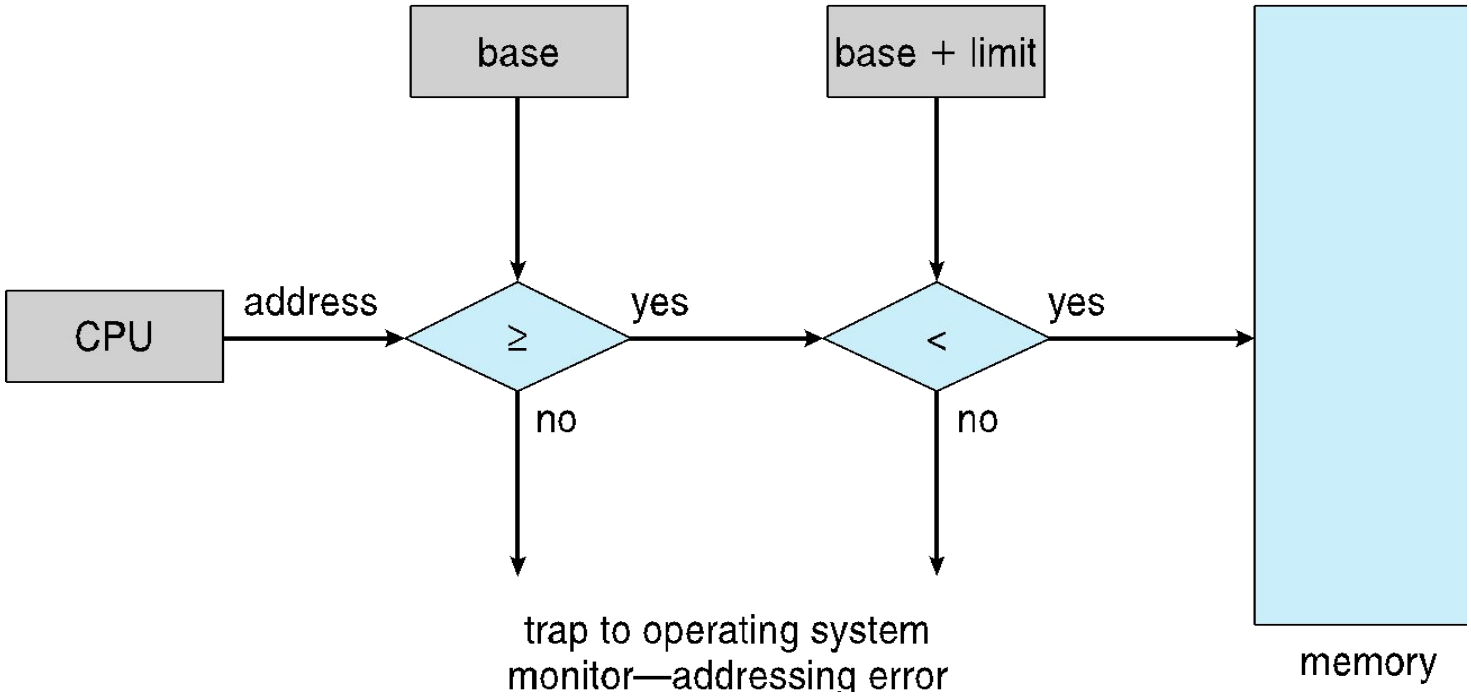
- If the process can be moved during its execution from one memory segment to another then address binding must be performed at run time.
- A special hardware-**MMU** (*Memory Management Unit*) is used to generate physical addresses.
- Example traveling in a bus, seat may change during travling

Base and Limit Registers

- A pair of **base** and **limit registers** define the logical address space
- CPU must check every memory access generated in user mode to be sure it is between base and limit for that user



Hardware Address Protection



Logical vs. Physical Address Space

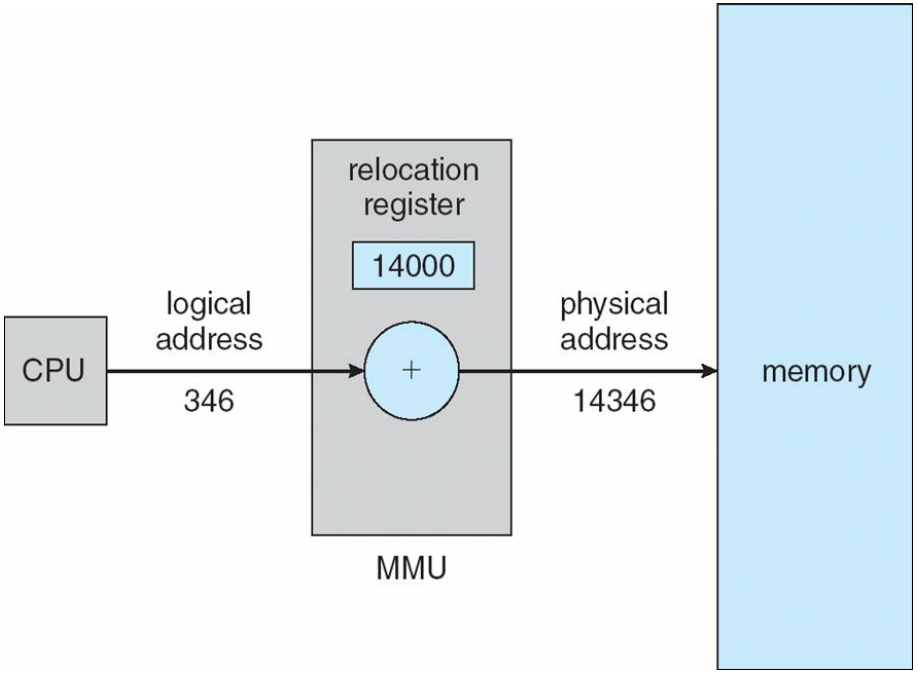
- The concept of a logical address space that is bound to a separate **physical address space** is central to proper memory management
  - **Logical address** – generated by the CPU; also referred to as **virtual address**
  - **Physical address** – address seen by the memory unit
- Logical and physical 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 i.e part of disk (virtual Memory)
- **Physical address space** is the set of all physical addresses generated by a program i.e Main Memory

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 relocation using a relocation register

- 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 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

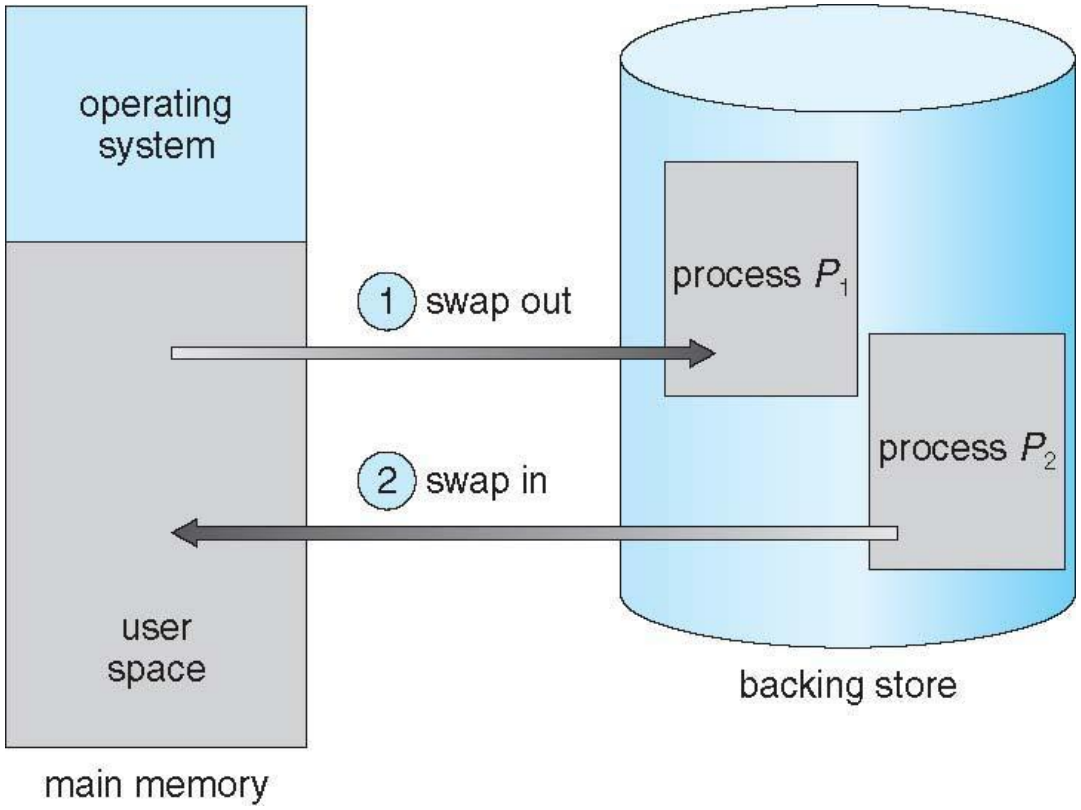
Swapping

- A process can be **swapped** temporarily out of memory to a backing store, and then brought back into memory for continued execution
  - Total physical memory space of processes can exceed physical memory
- **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 swapped
- System maintains a **ready queue** of ready-to-run processes which have memory images on disk

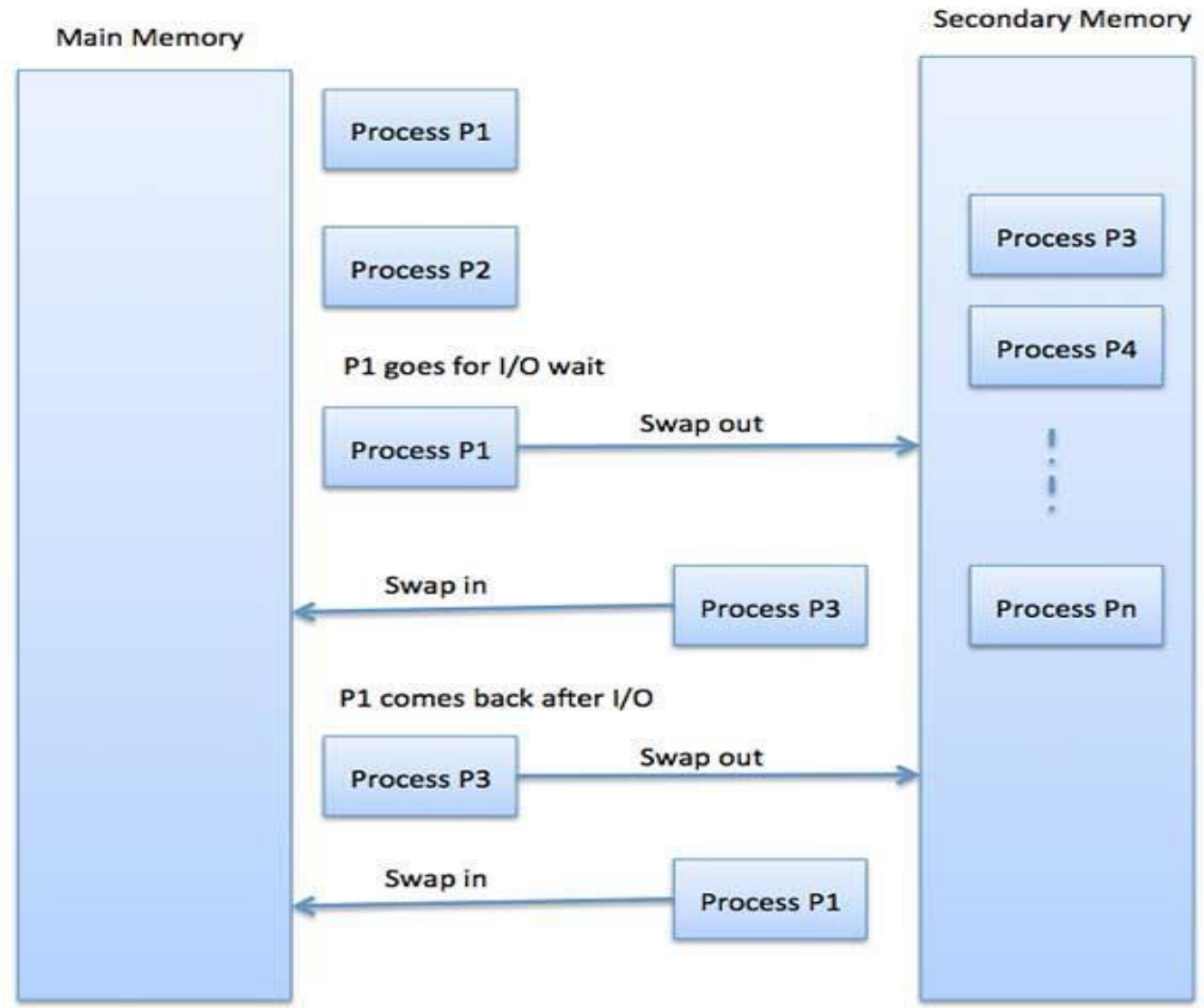
Swapping (Cont.)

- Does the swapped-out process need to swap back into same physical addresses?
- Depends on address binding method
  - Plus consider pending I/O to / from process memory space
- Modified versions of swapping are found on many systems (i.e., UNIX, Linux, and Windows)
  - Swapping normally disabled
  - Started if more than threshold amount of memory allocated
  - Disabled again once memory demand reduced below threshold

Schematic View of Swapping



Schematic View of Swapping



Context Switch Time including Swapping

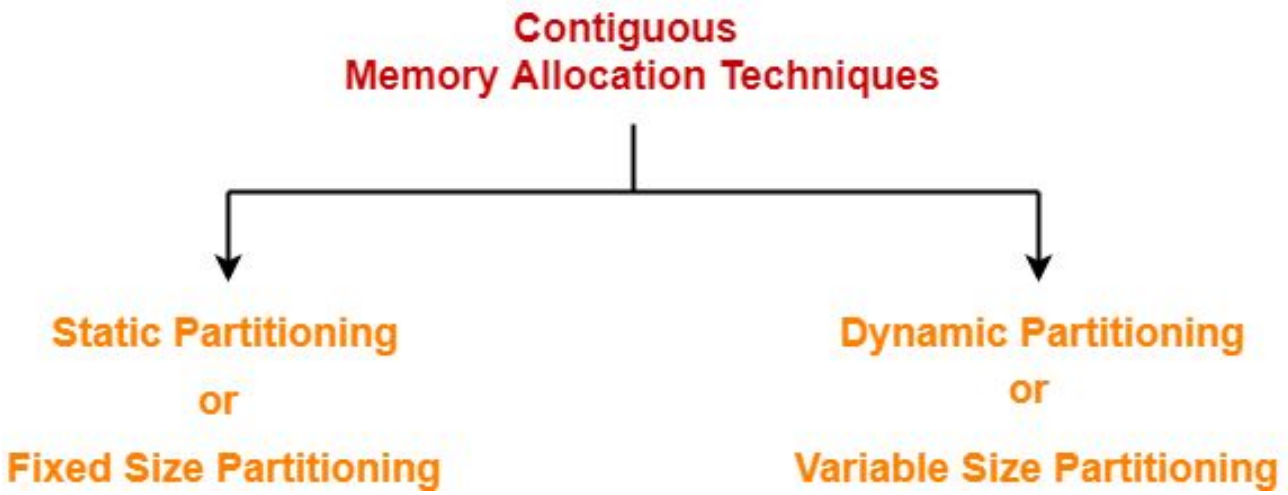
- If next processes to be put on CPU is not in memory, need to swap out a process and swap in target process
- Context switch time can then be very high
- 100MB process swapping to hard disk with transfer rate of 50MB/sec
  - Swap out time of 2000 ms
  - Plus swap in of same sized process
  - Total context switch swapping component time of 4000ms (4 seconds)
- Can reduce if reduce size of memory swapped – by knowing how much memory really being used
  - System calls to inform OS of memory use via `request_memory()` and `release_memory()`

Context Switch Time and Swapping (Cont.)

- Other constraints as well on swapping
  - Pending I/O – can't swap out as I/O would occur to wrong process
  - Or always transfer I/O to kernel space, then to I/O device
    - Known as **double buffering**, adds overhead
- Standard swapping not used in modern operating systems
  - But modified version common
    - Swap only when free memory extremely low

Contiguous Allocation

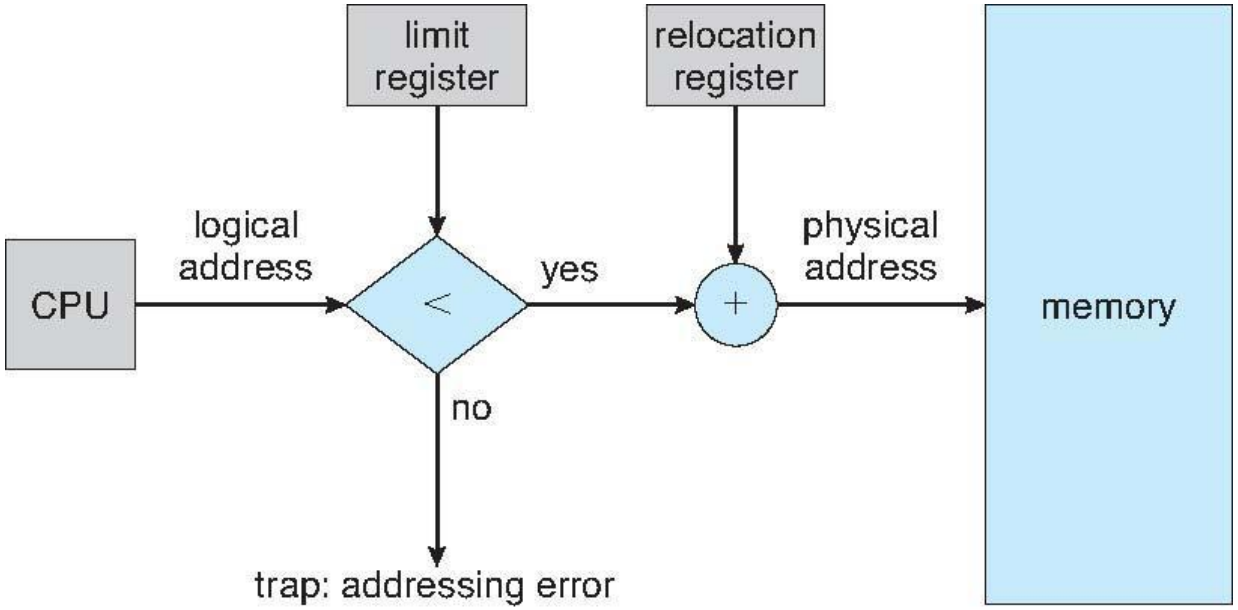
- Main memory must support both OS and user processes
- Limited resource, must allocate efficiently
- Contiguous allocation is one early method
- Main memory usually divided into two **partitions**:
  - Resident operating system, usually held in low memory with interrupt vector
  - User processes then held in high memory
  - Each process contained in single contiguous section of memory



Contiguous Allocation (Cont.)

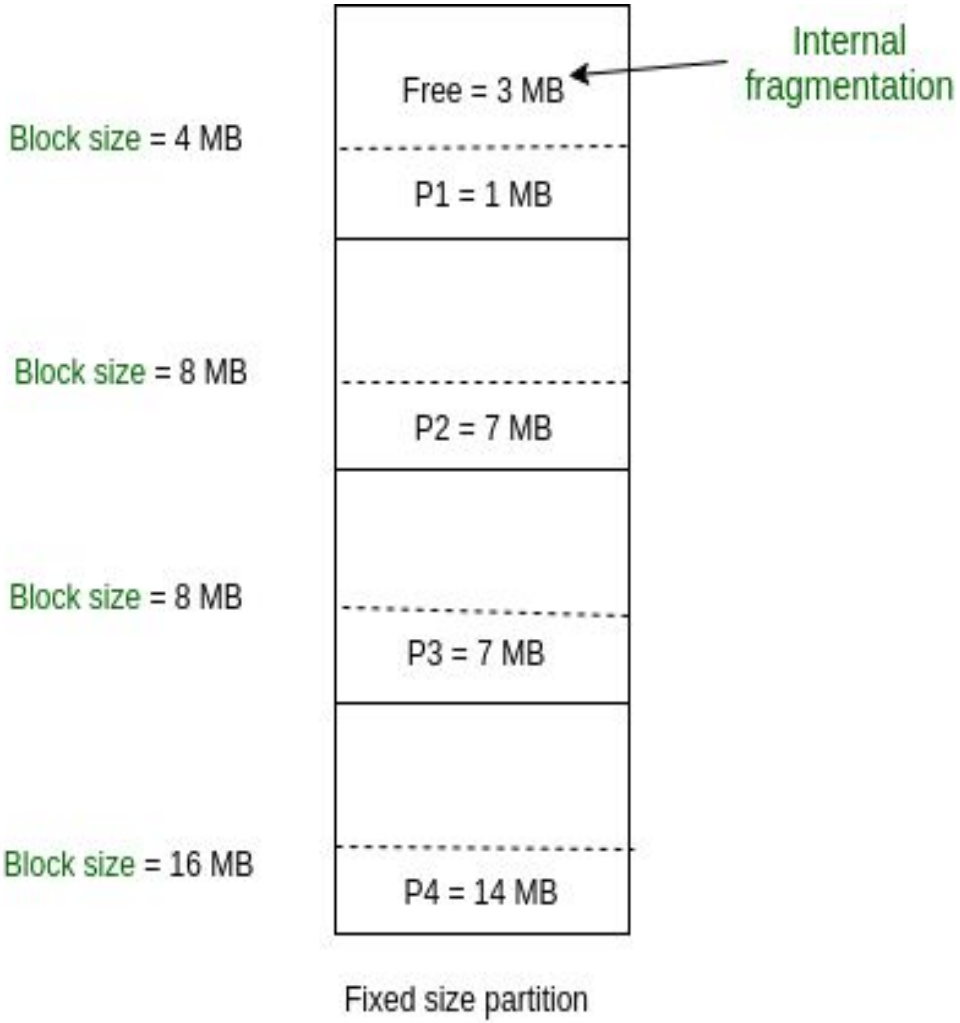
- Relocation registers used to protect user processes from each other, and from changing operating-system code and data
  - Base register contains value of smallest physical address
  - Limit register contains range of logical addresses – each logical address must be less than the limit register
  - MMU maps logical address *dynamically*
  - Can then allow actions such as kernel code being **transient** and kernel changing size

Hardware Support for Relocation and Limit Registers



Fixed Size Partitioning

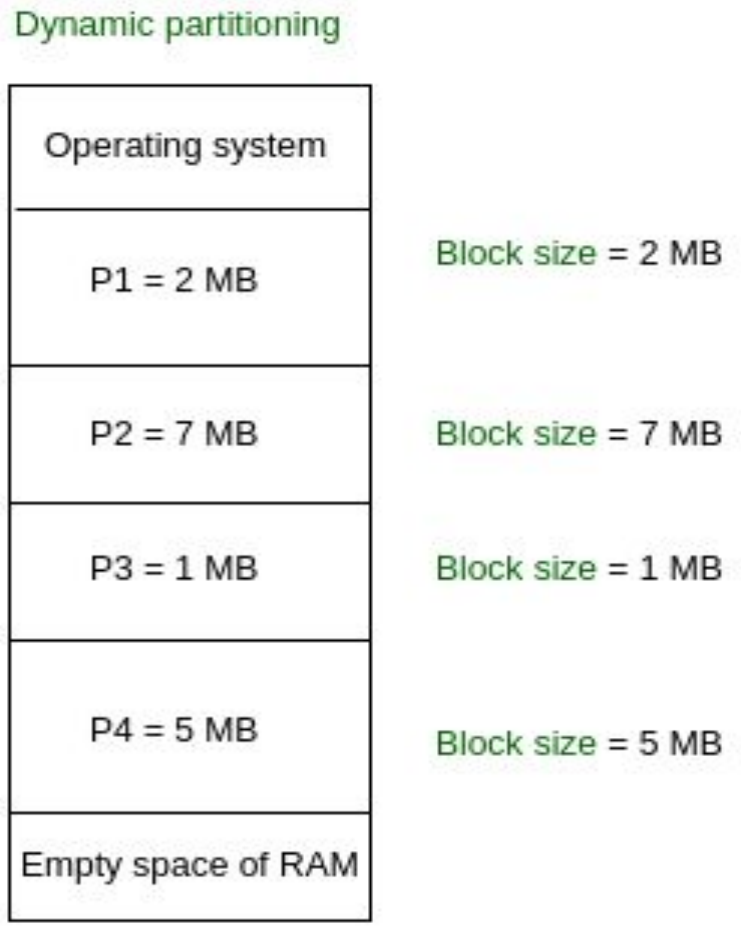
- This is the oldest and simplest technique used to put more than one processes in the main memory. In this partitioning, number of partitions (non-overlapping) in RAM are **fixed but size** of each partition may or **may not be same**.
- As it is **contiguous** allocation, hence no spanning is allowed. Here partition are made before execution or during system configure.+/



Fixed size partition

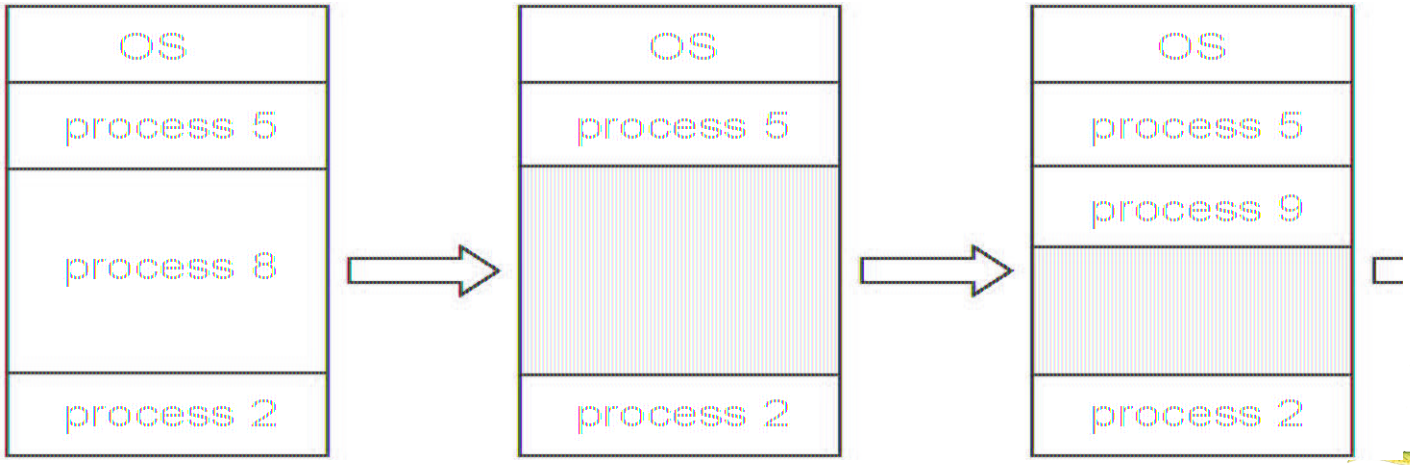
Variable Size Partitioning

- In contrast with fixed partitioning, partitions are not made before the execution or during system configure. Various **features** associated with variable Partitioning-
- Initially RAM is empty, and partitions are made during the run-time according to process's need instead of partitioning during system configure.
- The size of partition will be equal to incoming process.
- The partition size varies according to the need of the process so that the internal fragmentation can be avoided to ensure efficient utilization of RAM.
- Number of partitions in RAM is not fixed and depends on the number of incoming process and Main Memory's size.



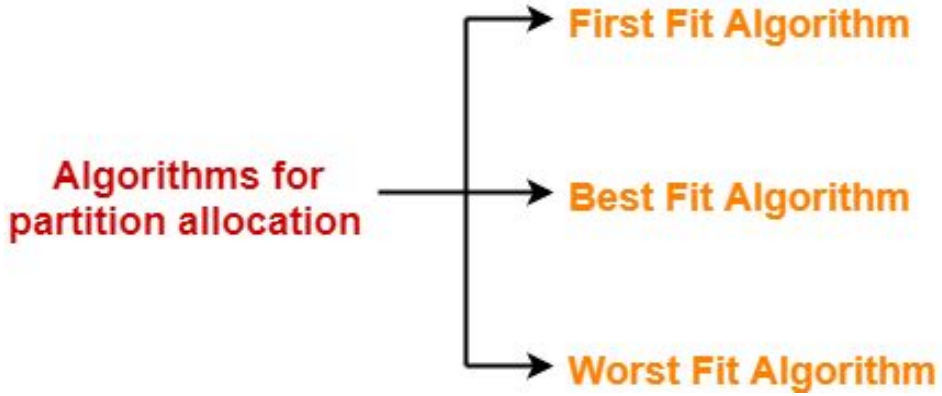
Multiple-partition allocation

- 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)



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
- **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

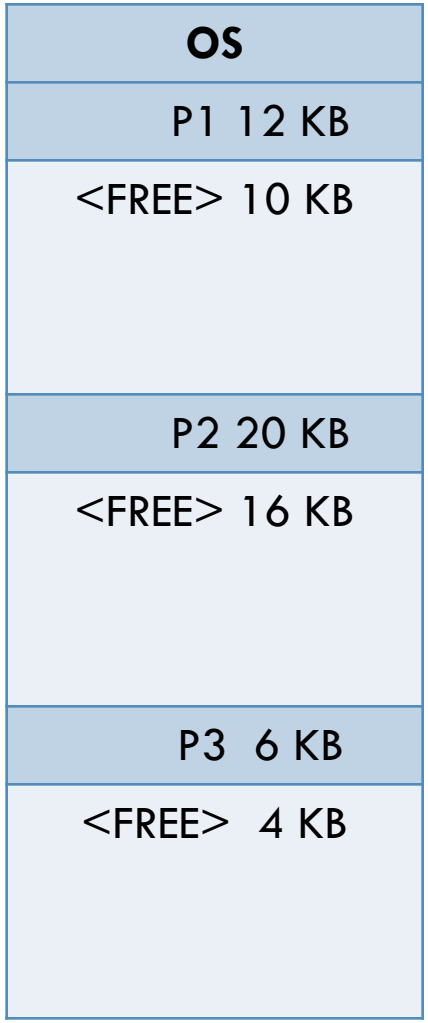
First-fit and best-fit better than worst-fit in terms of speed and storage utilization

First Fit

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

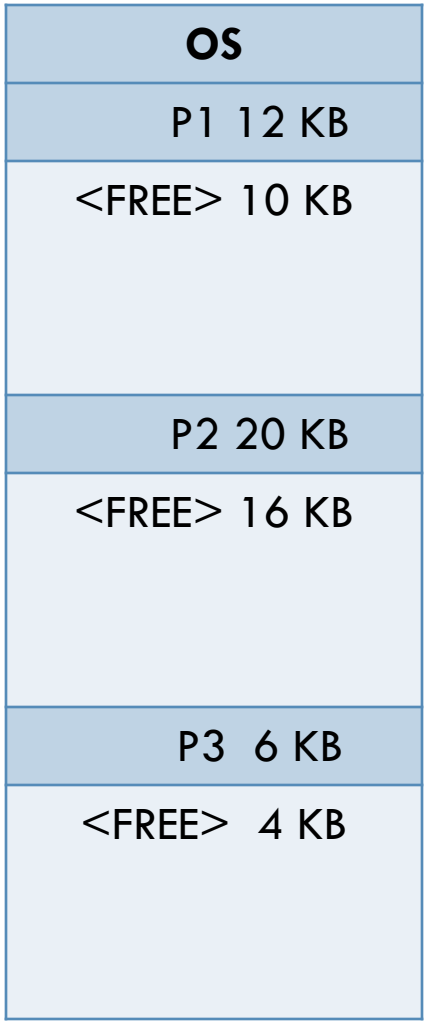
First Fit Example

Initial memory mapping



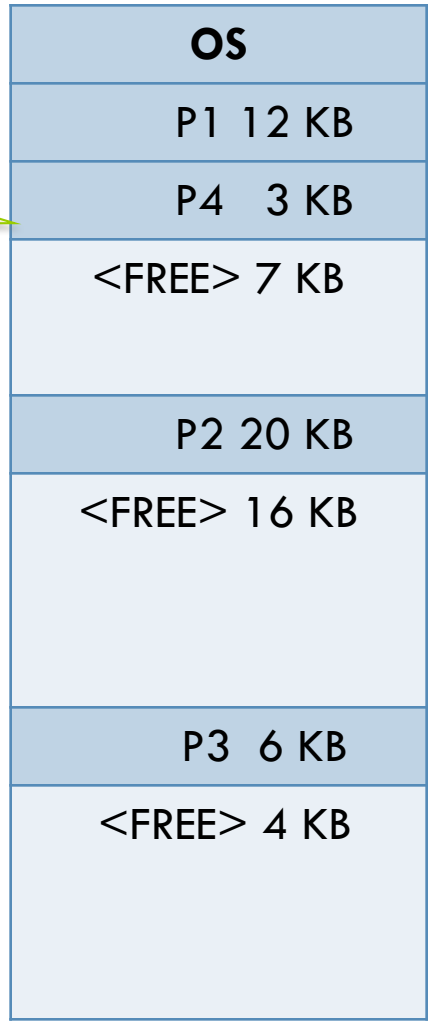
First Fit Example (Cont.)

P4 of 3KB arrives



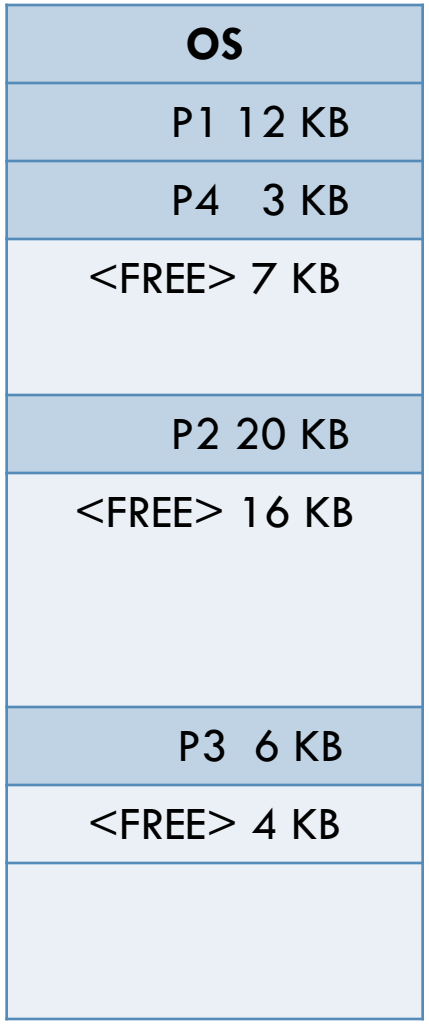
First Fit Example (Cont.)

P4 of 3KB loaded here by FIRST FIT



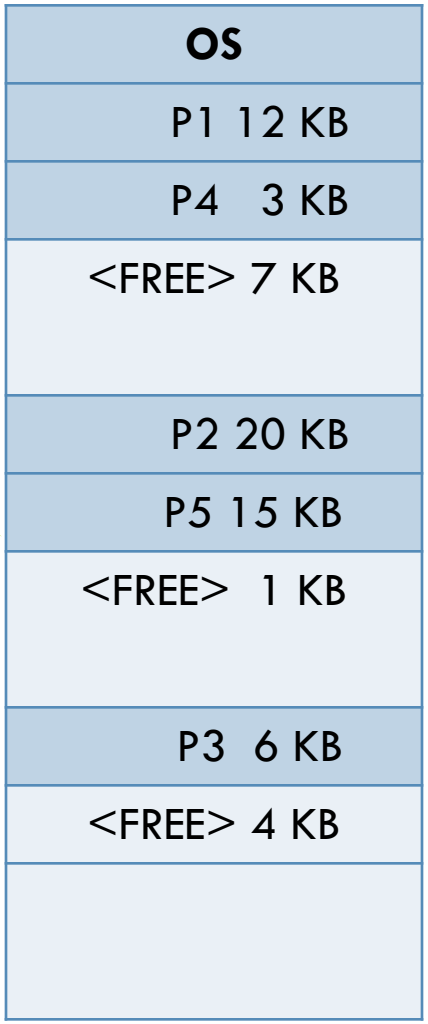
First Fit Example (Cont.)

P5 of 15KB arrives



First Fit Example (Cont.)

P5 of 15 KB loaded here by FIRST FIT

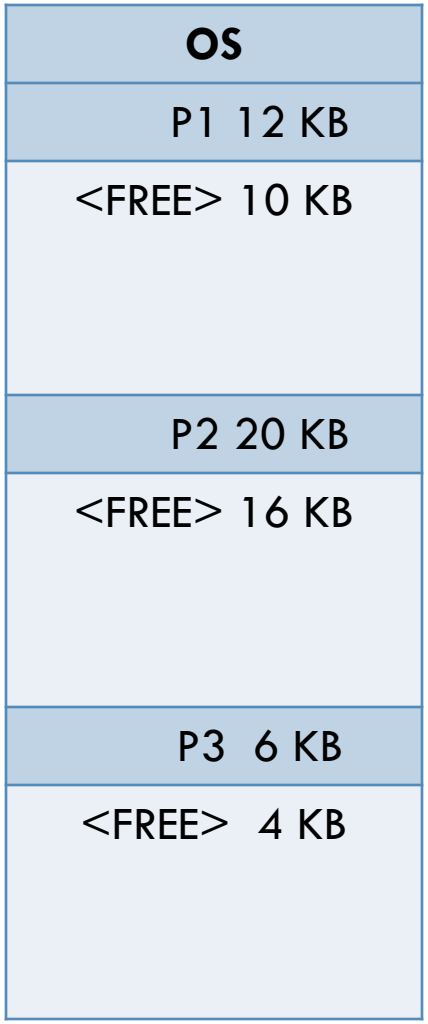


Best fit

- 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.

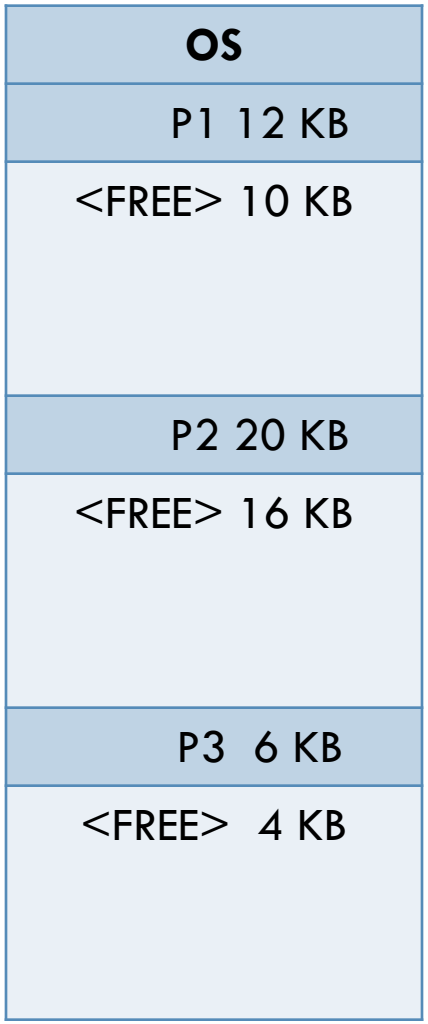
Best Fit Example

Initial memory mapping



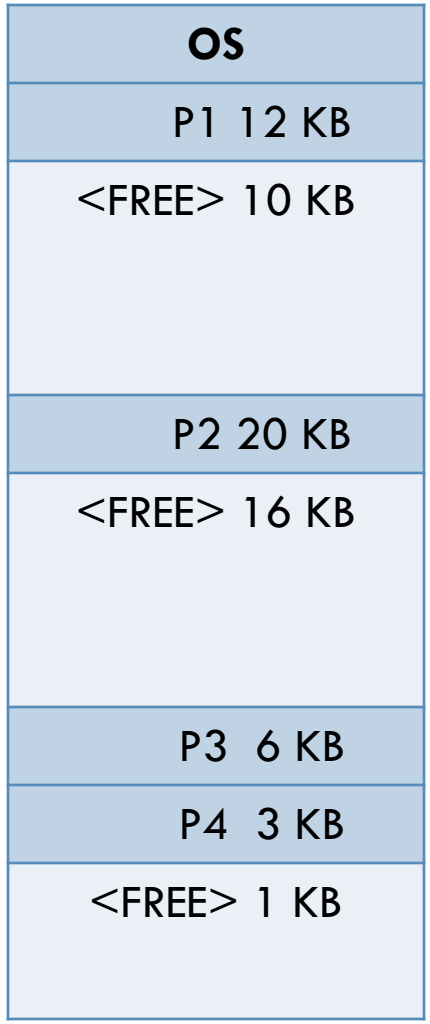
Best Fit Example (Cont.)

P4 of 3KB arrives



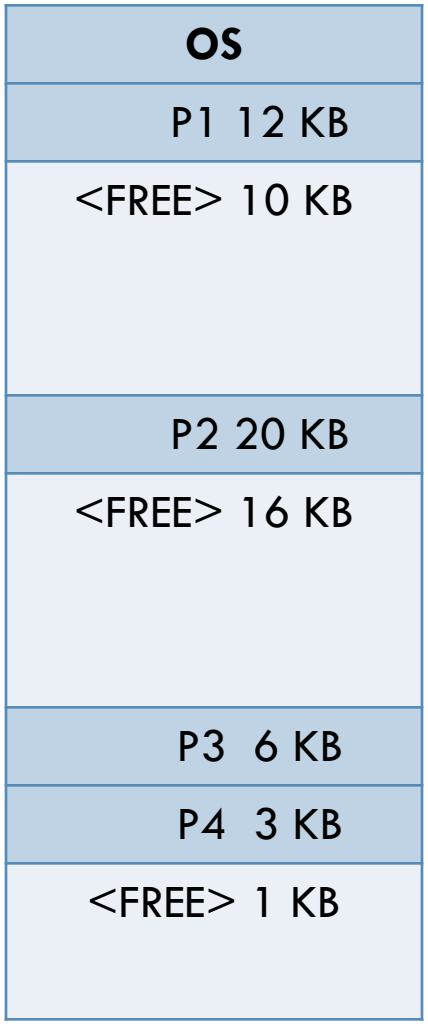
Best Fit Example (Cont.)

P4 of 3KB loaded here by BEST FIT



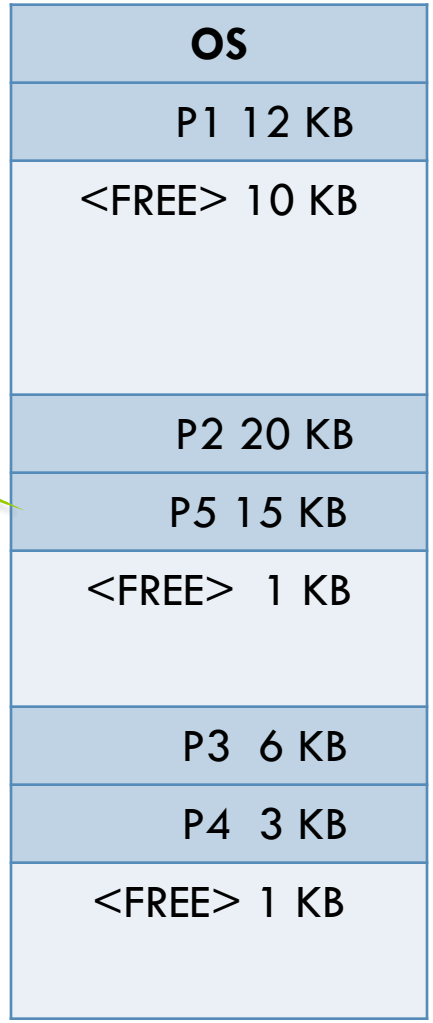
Best Fit Example (Cont.)

P5 of 15KB arrives



Best Fit Example (Cont.)

P5 of 15 KB loaded here by BEST FIT

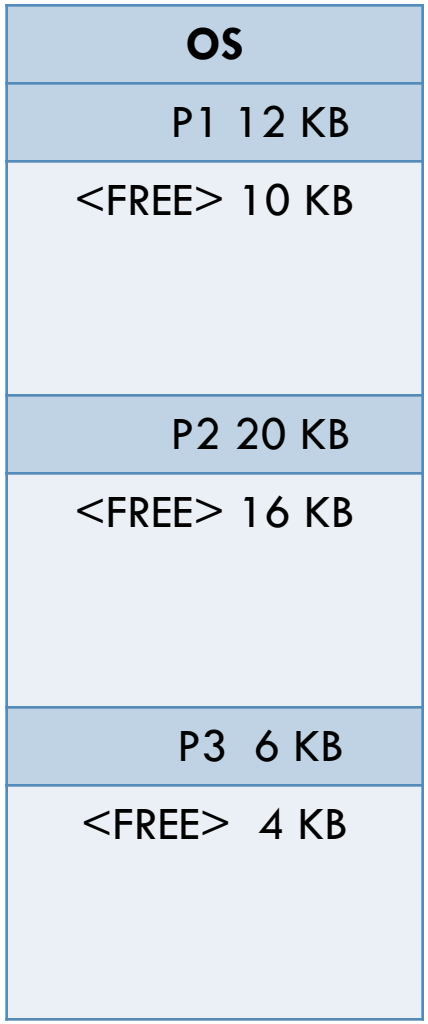


Worst Fit

- 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.

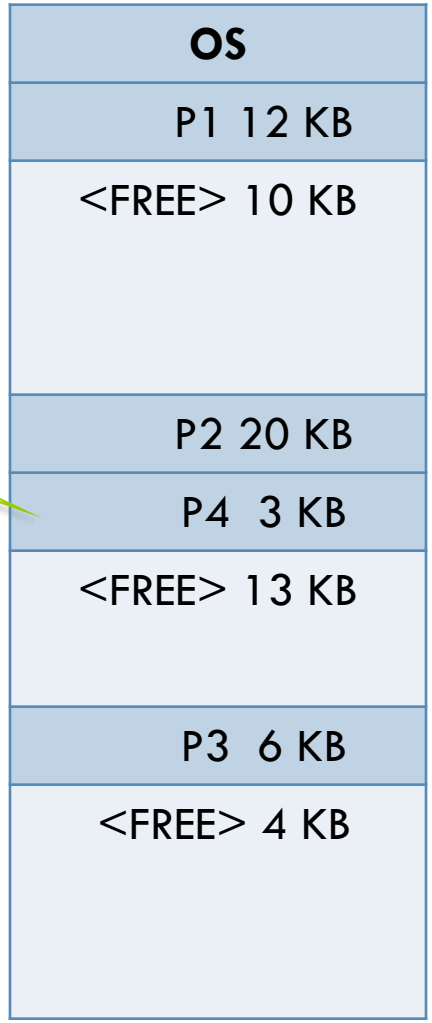
Worst Fit Example

Initial memory mapping



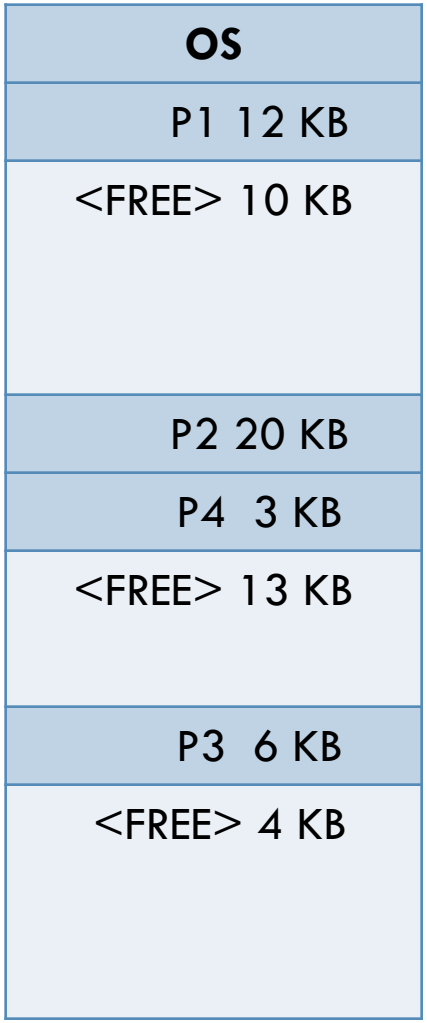
Worst Fit Example (Cont.)

P4 of 3KB
Loaded here by
WORST FIT



Worst Fit Example (Cont.)

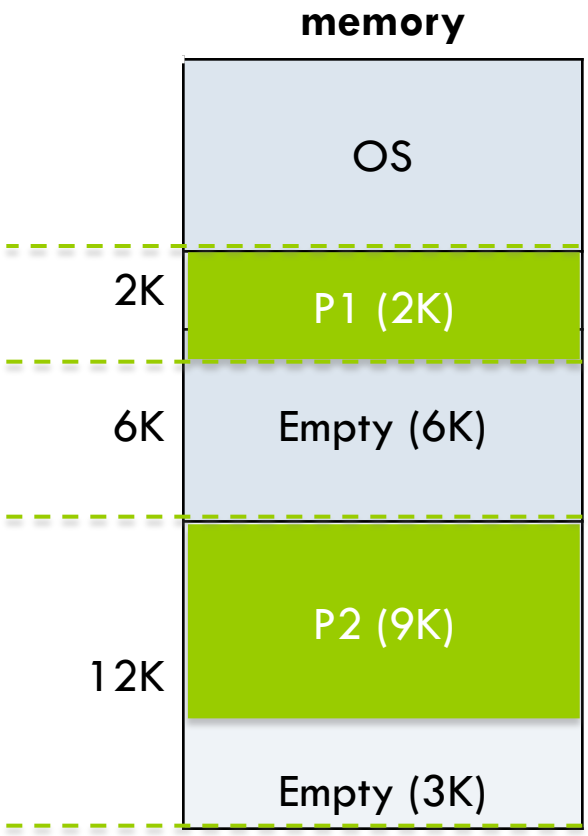
No place to load P5 of 15K



Fragmentation

- **External Fragmentation** – total memory space exists to satisfy a request, but it is not contiguous
- **Internal Fragmentation** – allocated memory may be slightly larger than requested memory; this size difference is memory internal to a partition, but not being used

Fragmentation



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

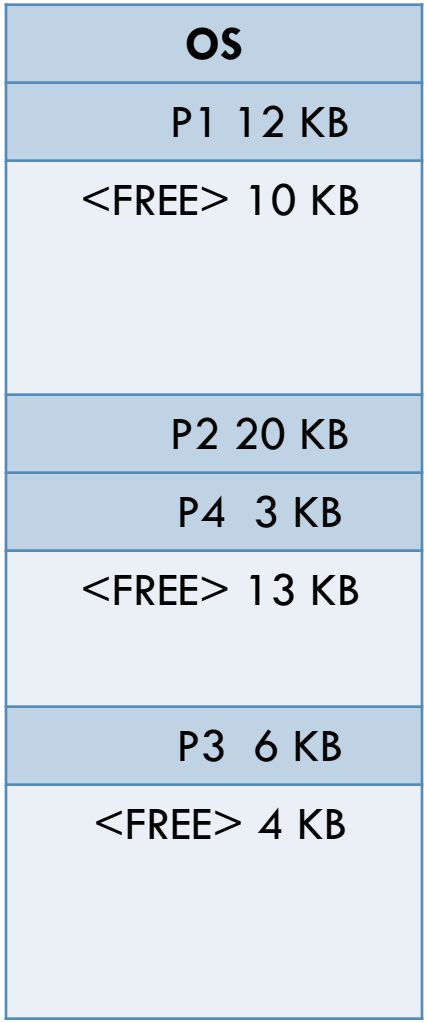
If a partition is being used by a process requiring some memory smaller than the partition size, then it is called an ***internal fragmentation.***

Compaction

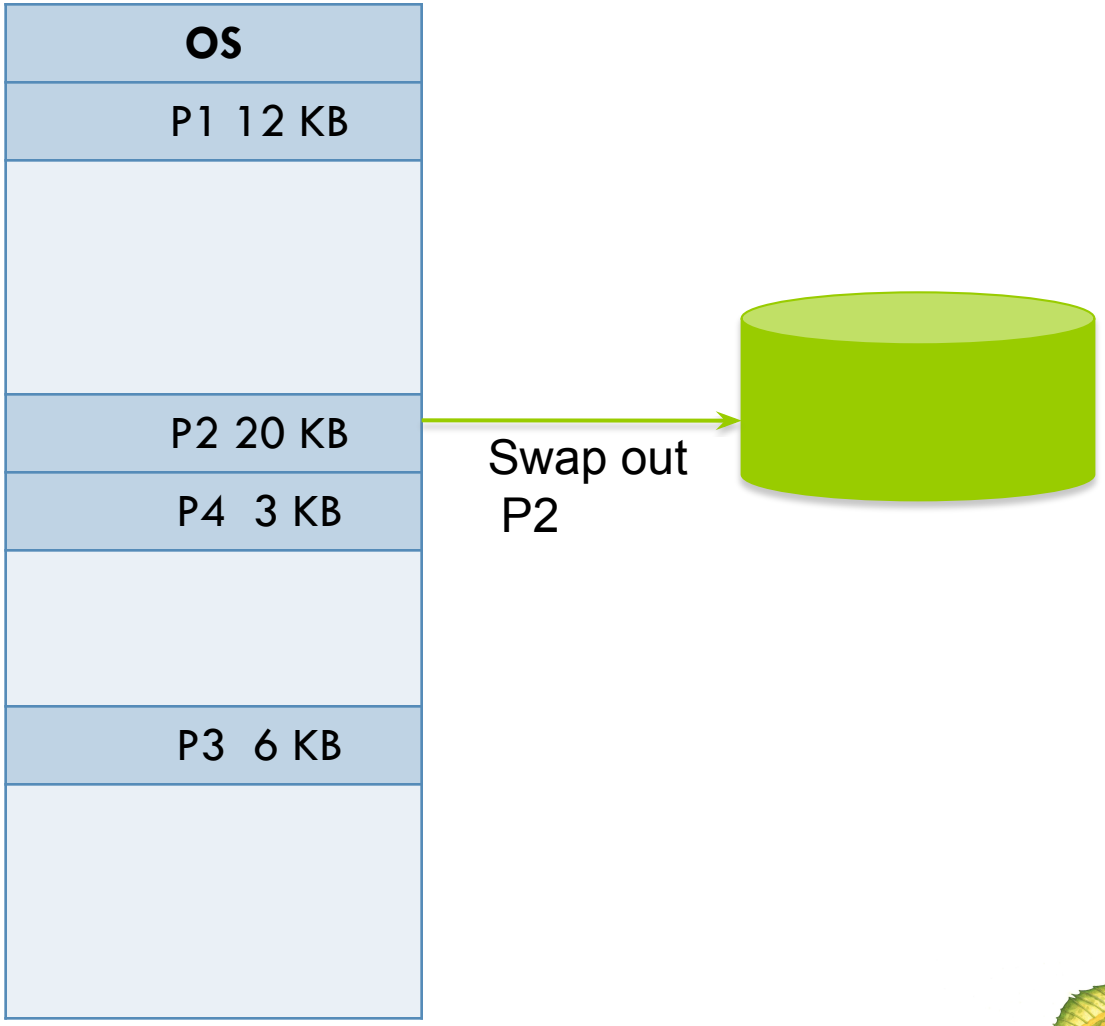
- Compaction is a method to overcome the external fragmentation problem.
- All free blocks are brought together as one large block of free space.
- Compaction requires dynamic relocation.
- Certainly, compaction has a cost and selection of an optimal compaction strategy is difficult.
- One method for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

Compaction

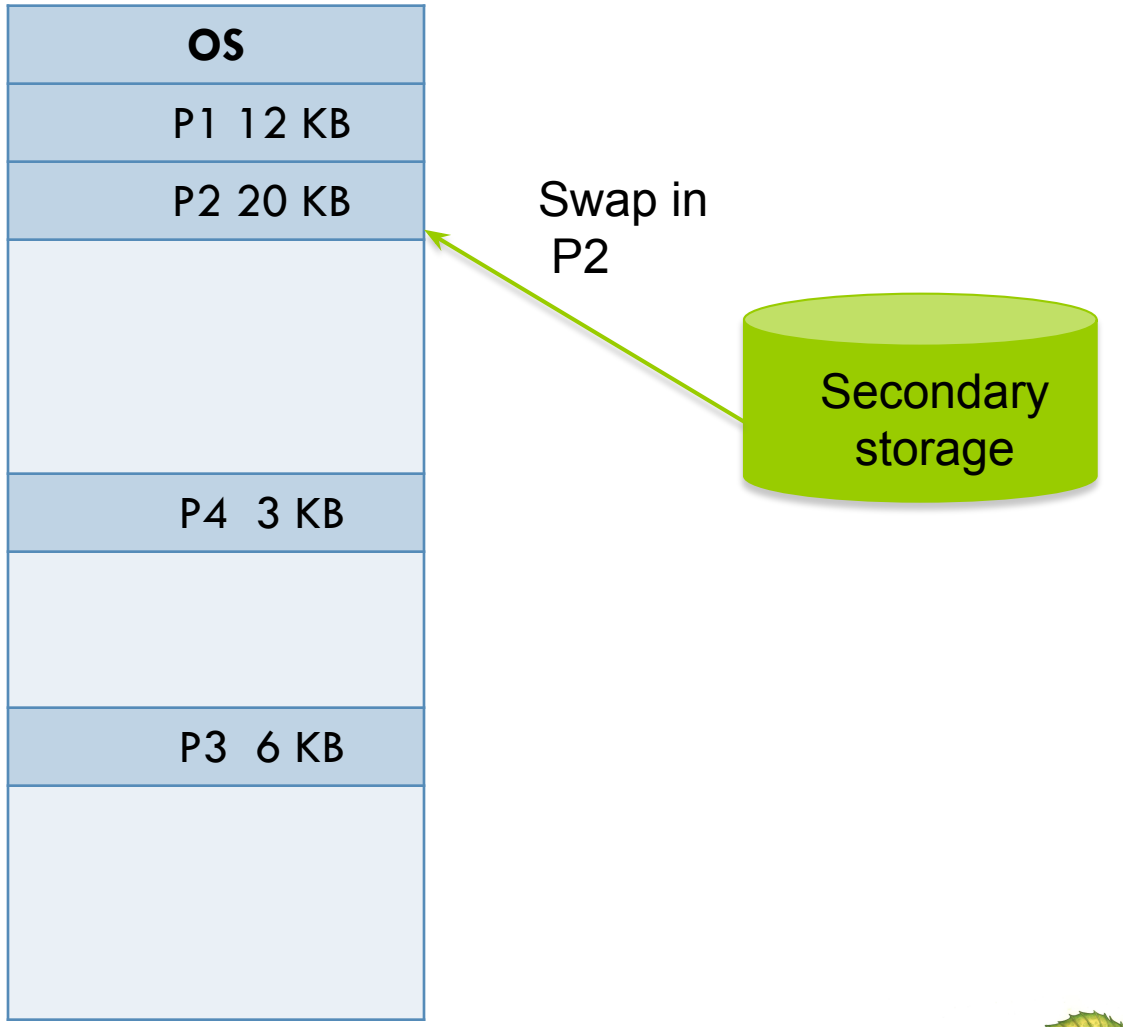
Memory mapping before compaction



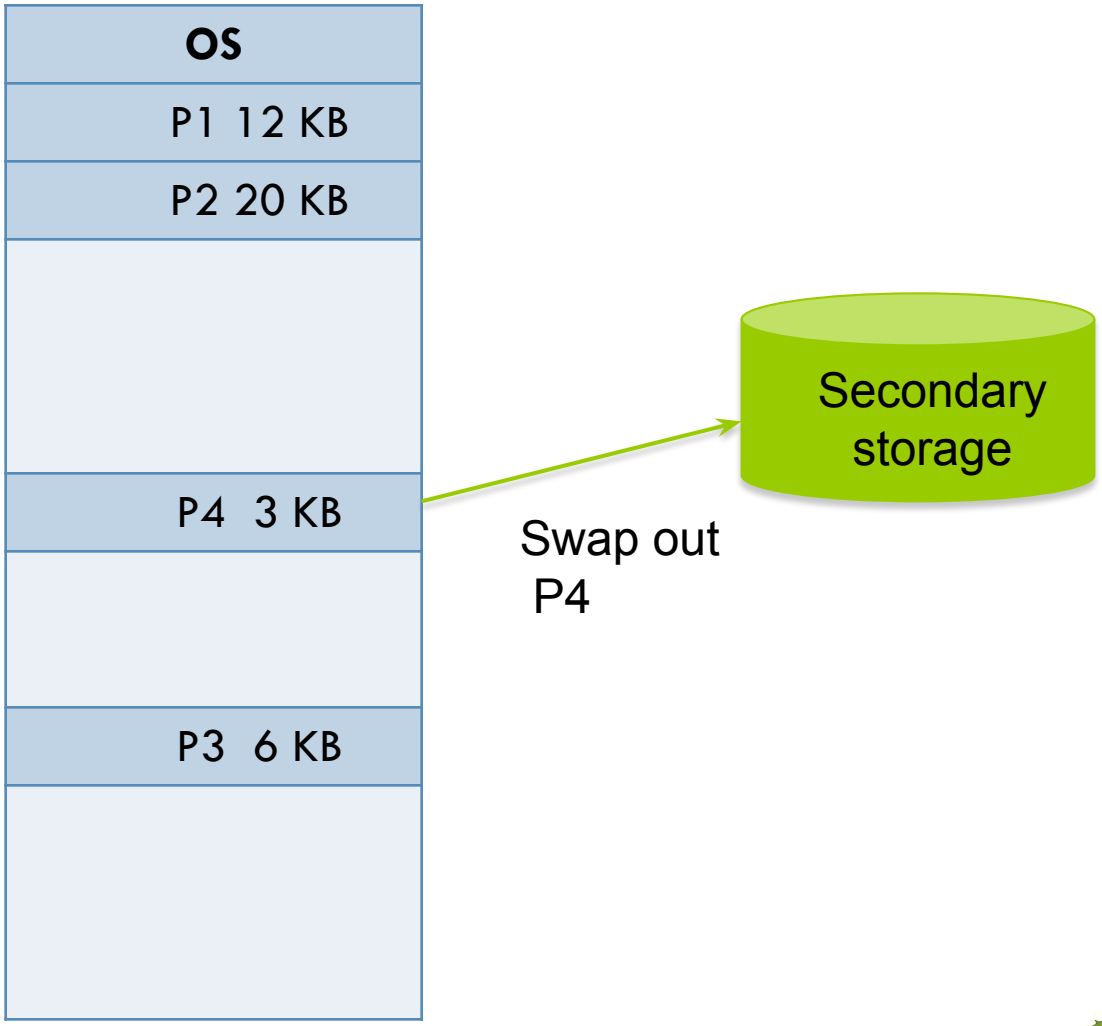
Compaction Example



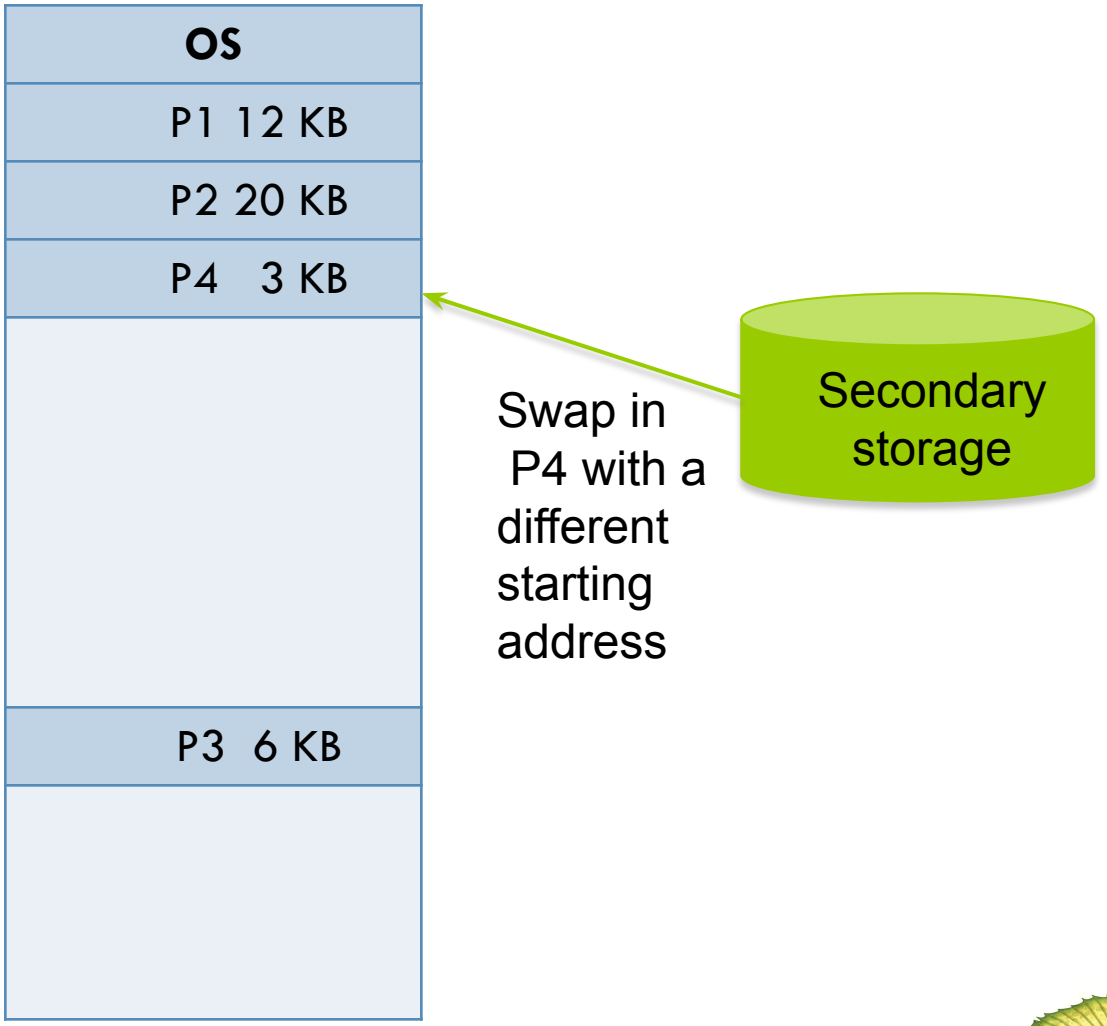
Compaction (Cont.)



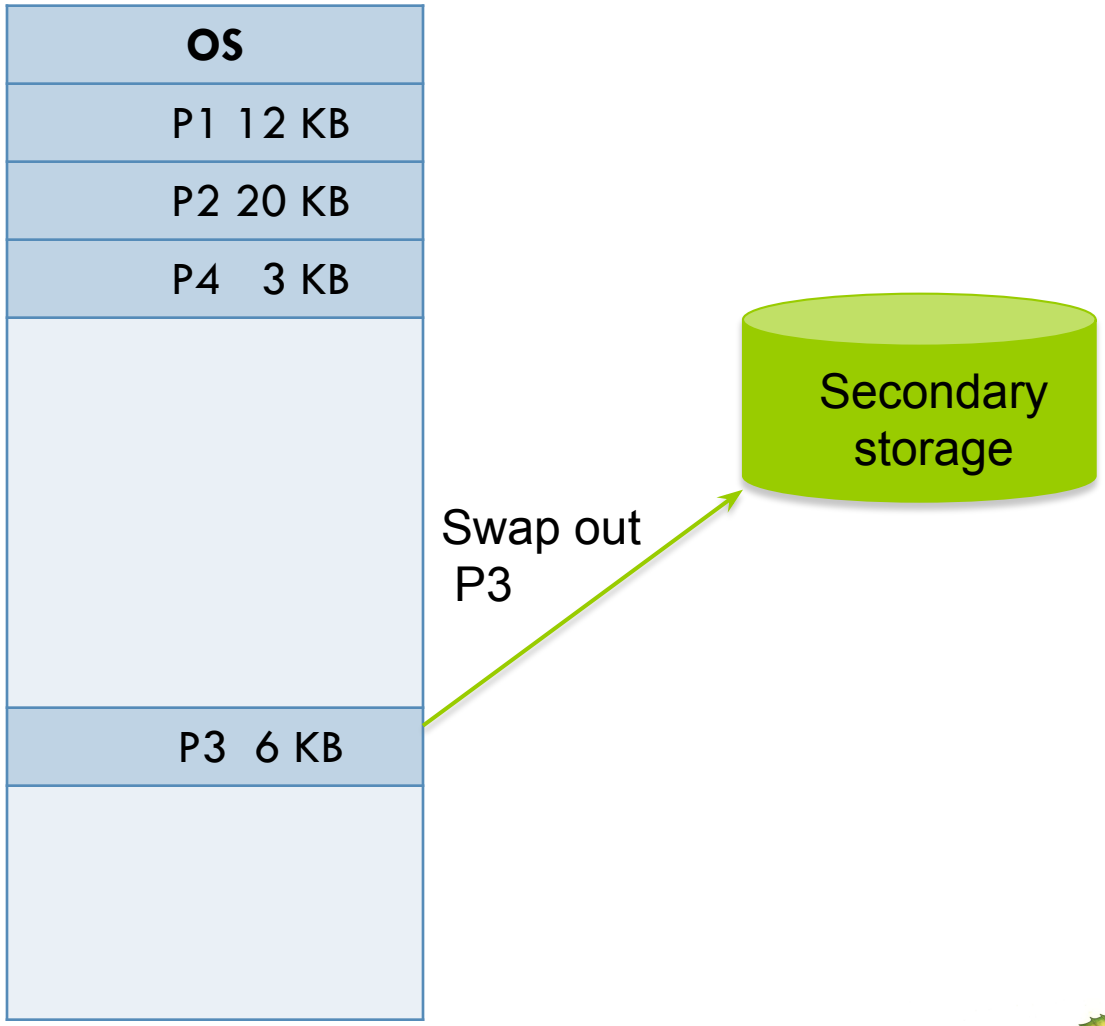
Compaction Example (Cont.)



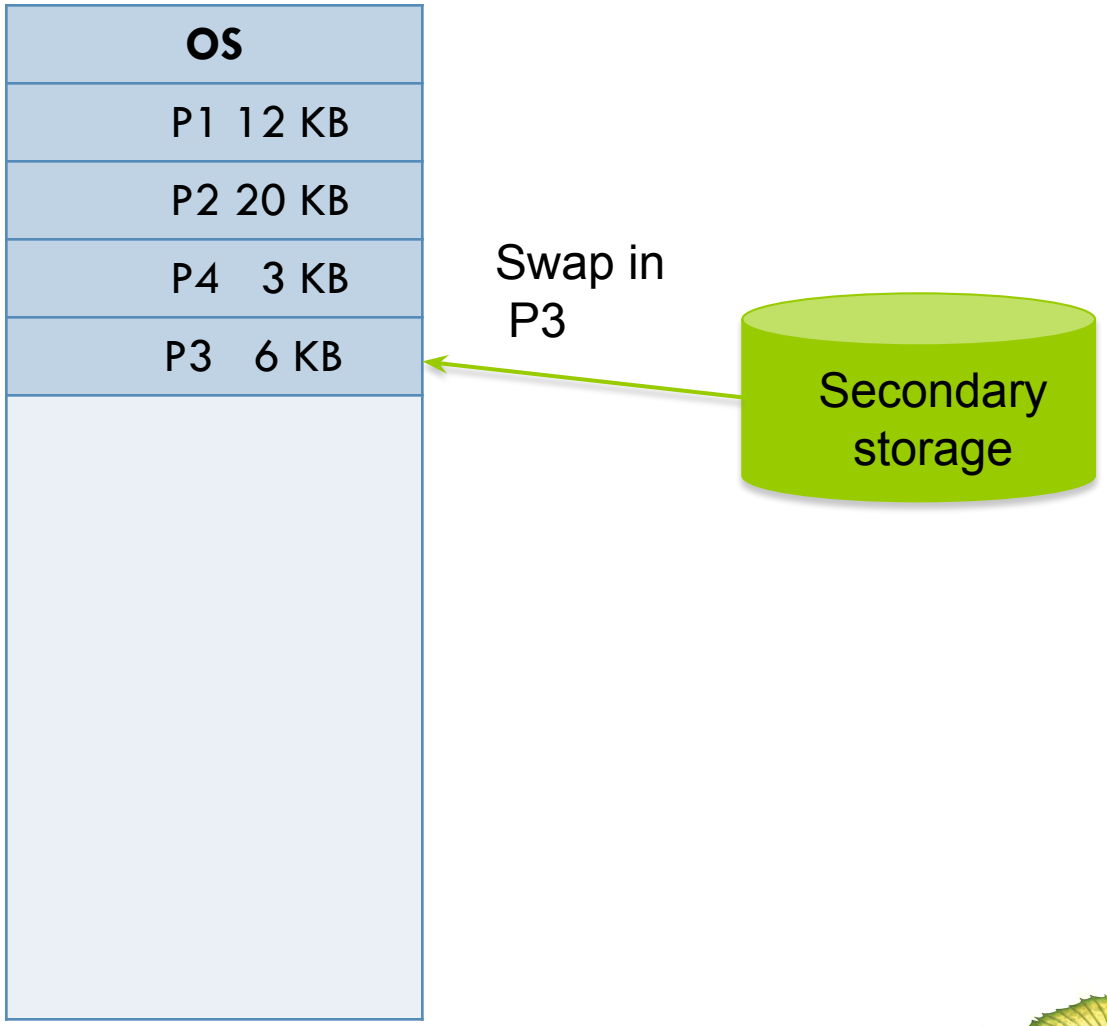
Compaction Example (Cont.)



Compaction Example (Cont.)

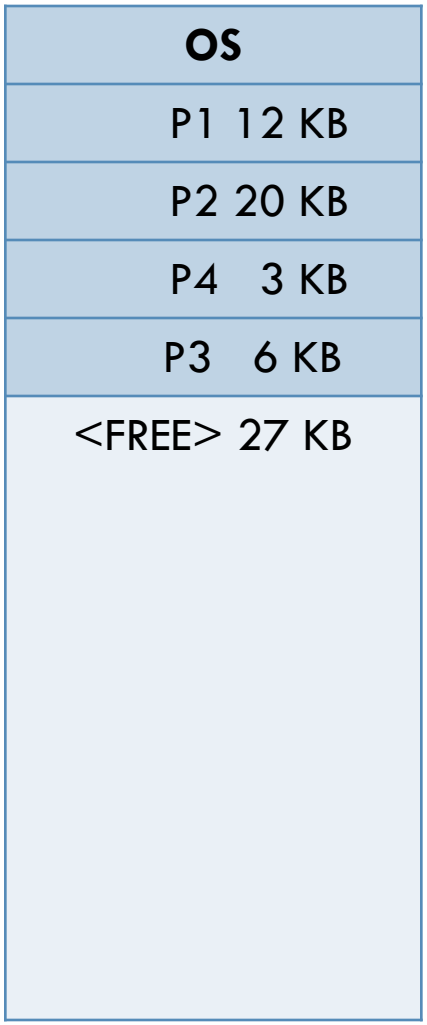


Compaction Example (Cont.)



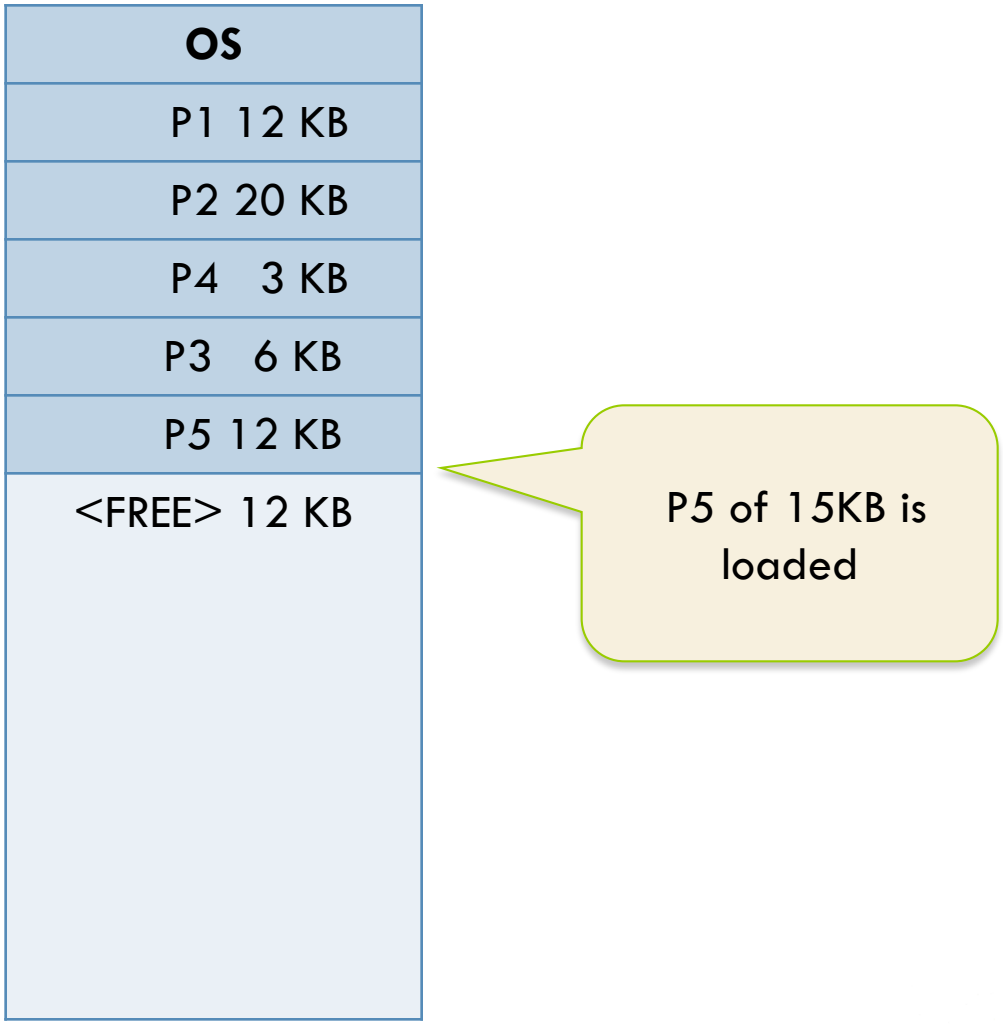
Compaction Example (Cont.)

Memory mapping after compaction



Now P5 of 15KB can be loaded here

Compaction Example (Cont.)



How does user view Memory

A process is divided into Segments. The chunks that a program is divided into are not necessarily to be of the same sizes are called segments. It is a non- Contiguous memory allocation scheme to processes.

Segmentation gives user's view of the process which paging does not give. Here the user's view is mapped to physical memory.

Thus the differentiation comes between the logical and physical memory.

Do users view memory as linear array of bytes some containing instruction and other containing data???

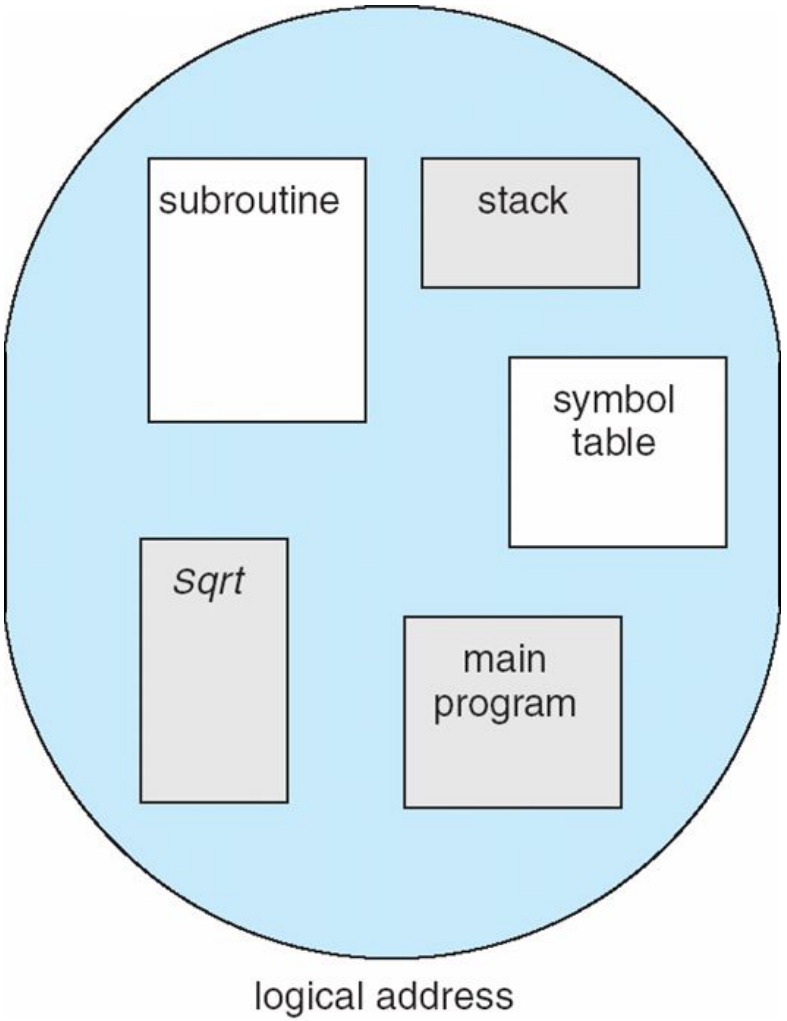
No…Rather they would see it as collection of

Segmentation

- Memory-management scheme that supports user view of memory
- A program is a collection of segments
  - A segment is a logical unit such as:

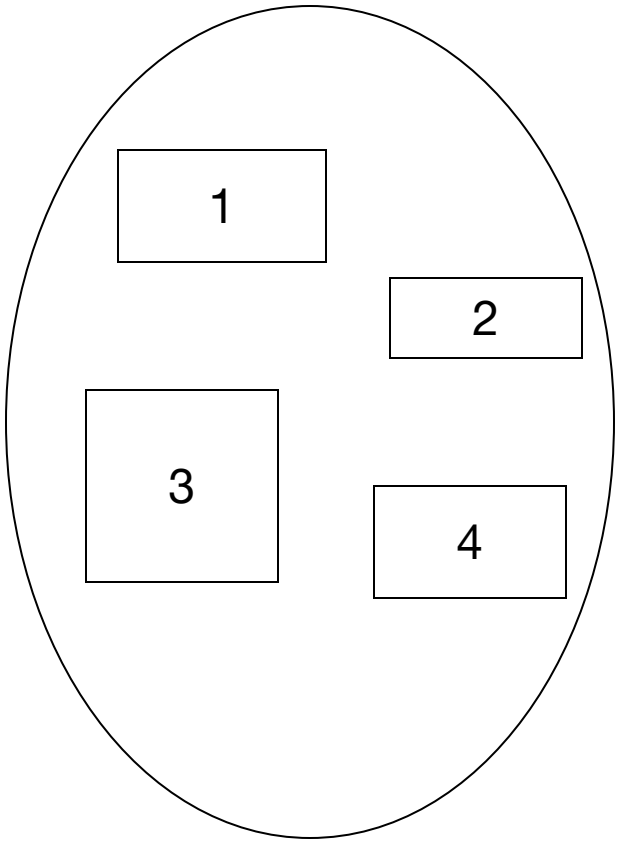
main program
procedure
function
method
object
local variables, global variables
common block
stack
symbol table
arrays

Segmentation

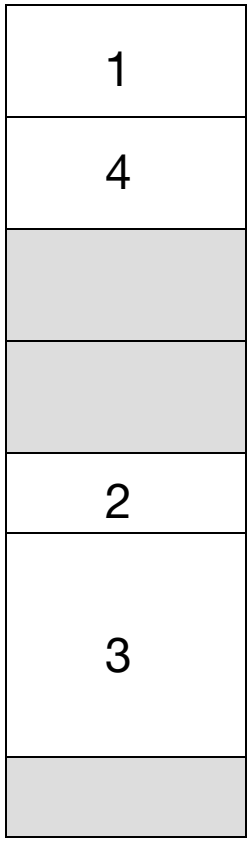


- User View of logical memory
  - Linear array of bytes
  - A collection of variable-sized entities
    - User thinks in terms of "subroutines", "stack", "symbol table", "main program" which are somehow located somewhere in memory.
- Segmentation supports this user view. The logical address space is a collection of segments.

Logical View of Segmentation



user space



physical memory space

Segmentation

- Although the user can refer to objects in the program by a two-dimensional address, the actual physical address is still a one-dimensional sequence
- Thus, we need to map the segment number
- This mapping is effected by a **segment table**
- In order to protect the memory space, each entry in segment table has a **segment base** and a **segment limit**

Segmentation Architecture

- Logical address consists of a two tuple:

  <segment-number, offset>,

- **Segment table** – maps two-dimensional physical addresses; each table entry has:
  - **base** – contains the starting physical address where the segments reside in memory
  - **limit** – specifies the length of the segment
- **Segment-table base register (STBR)** points to the segment table's location in memory
- **Segment-table length register (STLR)** indicates number of segments used by a program;

  segment number ***s*** is legal if ***s*** < **STLR**

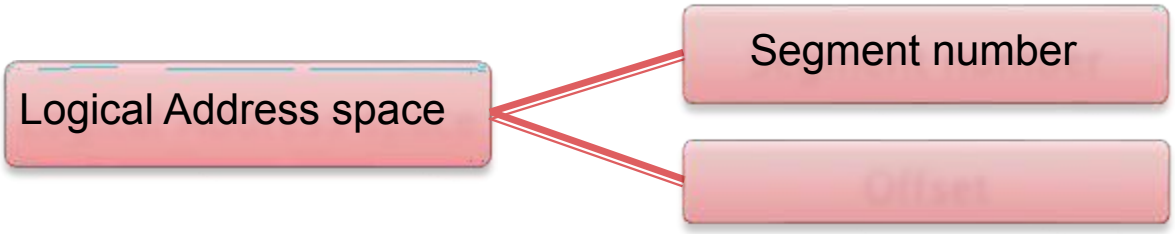
Segmentation Architecture

- **Segments are variable-sized**
  - Dynamic memory allocation required (first fit, best fit, worst fit).
- **External fragmentation**
  - In the worst case the largest hole may not be large enough to fit in a new segment.
- **Each process has its own segment table**
  - Each process has its own page table. The size of the segment table is determined by the number of segments, whereas the size of the page table depends on the total amount of memory occupied.
- Segment table located in the main memory

Segmentation Architecture (Cont.)

- Protection
  - With each entry in segment table associate:
    - validation bit = 0 ⇒ illegal segment
    - read/write/execute privileges
- Protection bits associated with segments; code sharing occurs at segment level
- Since segments vary in length, memory allocation is a dynamic storage-allocation problem

Logical Addressing in Segmentation



The mapping of the logical address to the physical address is done with the help of the segment table.

the length of the segment

SEGMENT TABLE

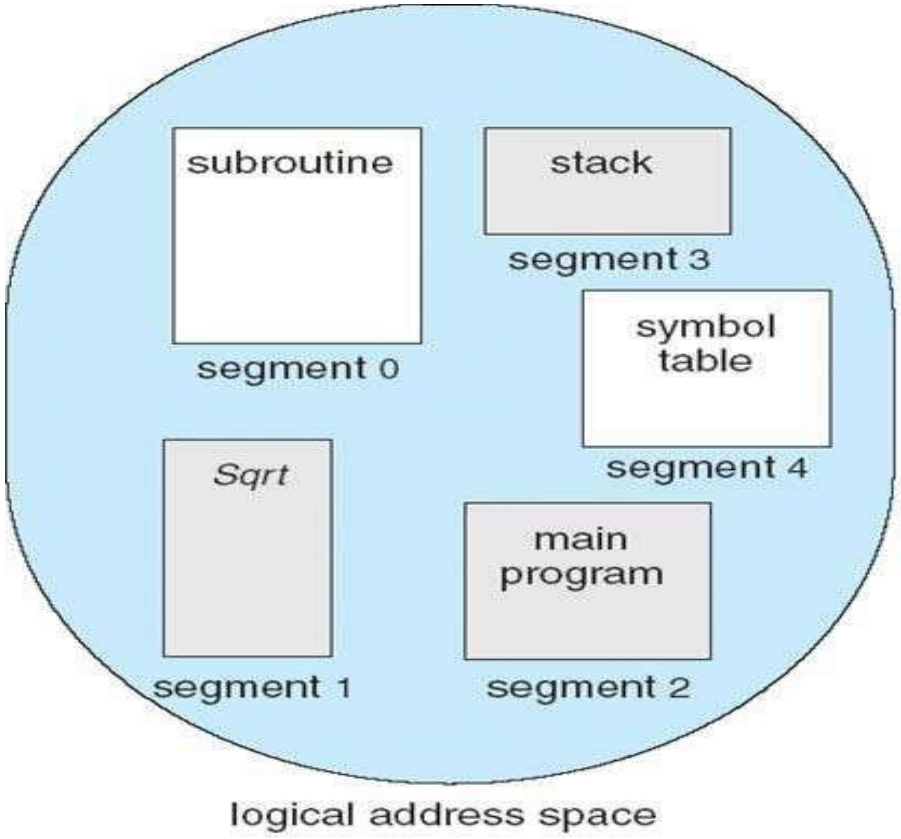
Segment Limit	Segment Base	Other bits

starting address of the corresponding segment in main memory

A bit is needed to determine if the segment is already in main memory (P)
Another bit is needed to determine if the segment has been modified since it was loaded in main memory (M)

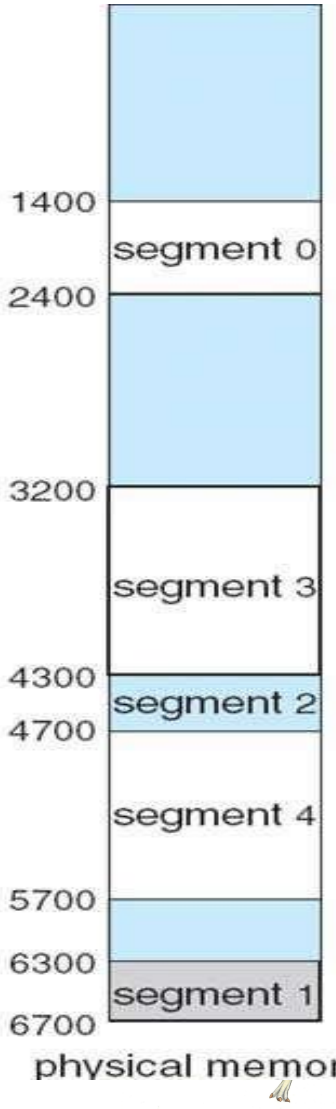
Example of Segmentation

A segmentation example is shown in the following diagram

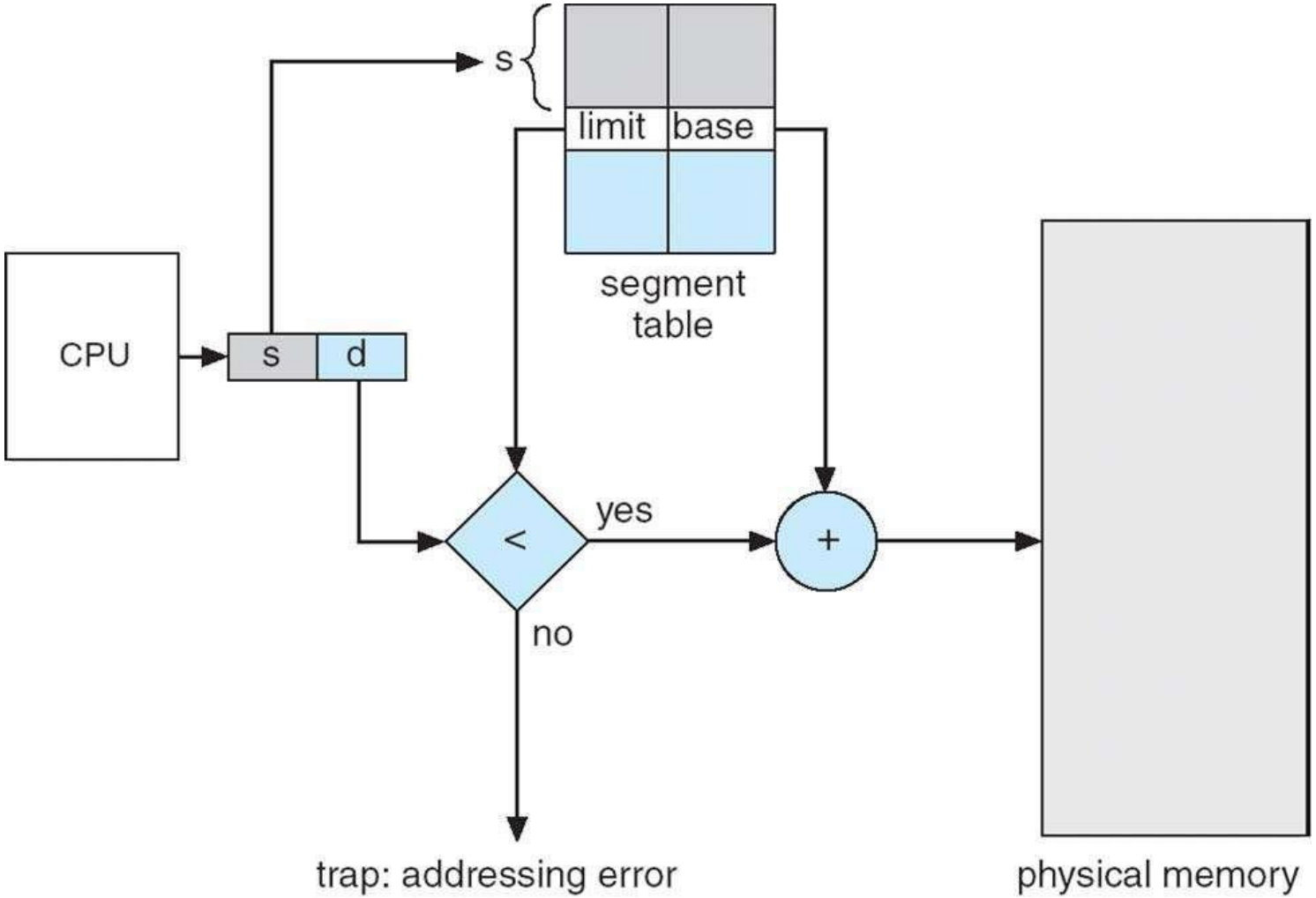


	limit	base
0	1000	1400
1	400	6300
2	400	4300
3	1100	3200
4	1000	4700

segment table



Segmentation Hardware



Protection and Sharing

- Segmentation lends itself to the implementation of protection and sharing policies
- Each entry has a base address and length so inadvertent memory access can be controlled
- Sharing can be achieved by segments referencing multiple processes
- Two processes that need to share access to a single segment would have the same segment name and address in their segment tables.

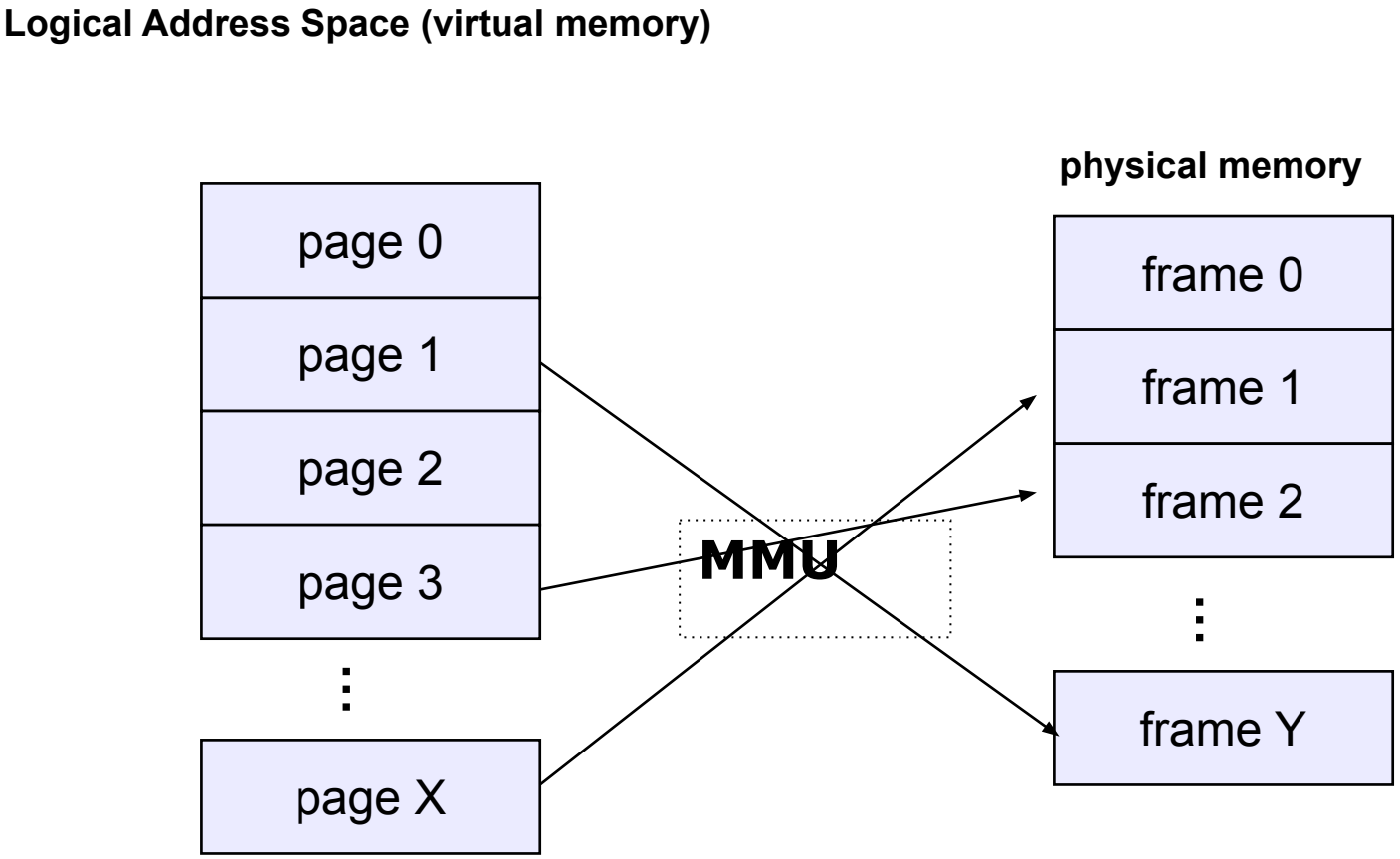
Disadvantages of Segmentation

- External fragmentation.
- Costly memory management algorithm
- Unequal size of segments is not good in the case of swapping.

Paging

- Physical address space of a process can be noncontiguous; process is allocated physical memory whenever the latter is available
- Memory partitions are same fixed size.
- The mapping from virtual logical to physical address is done by the memory management unit (MMU) which is a hardware device and this mapping is known as paging technique.
- The Physical Address Space is conceptually divided into a number of fixed-size blocks, called **frames**.
- The Logical address Space is also splitted into fixed-size blocks, called **pages**.
- Page Size = Frame Size

Address Mapping



Paging

- **Frames:**

  Divide physical memory into fixed-sized blocks called **frames**

- **Pages:**

  Divide logical memory into blocks of same size called **pages**

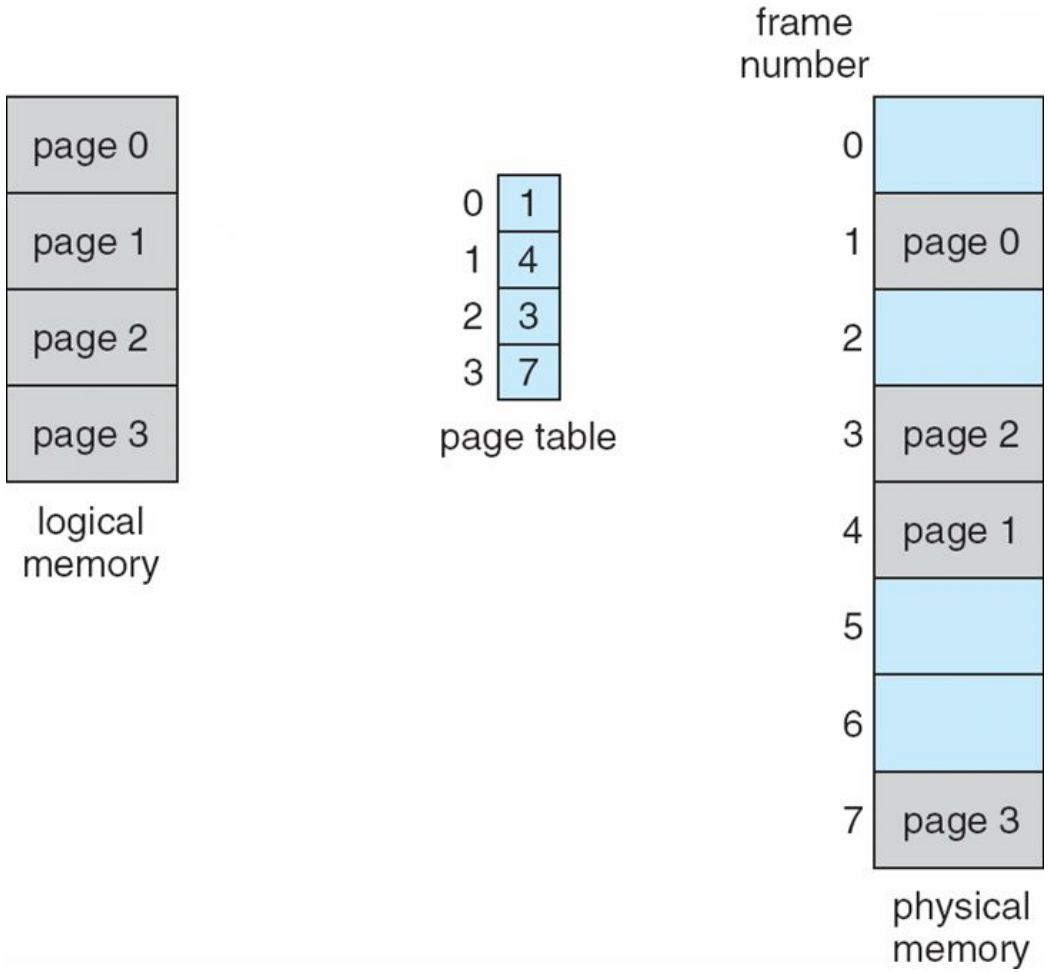
- **Page Table:**

  Set up a **page table** to translate logical to physical addresses

- *To run a program of size **N** pages, need to find **N** free frames and load program*

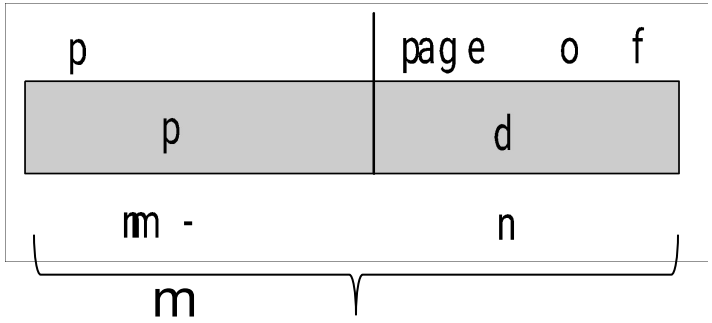
- **Advantage of Paging:**
  - Easy to use memory management algorithm
  - No need for external Fragmentation
  - Swapping is easy between equal-sized pages and page frames.
- **Disadvantages:**
  - Still have Internal fragmentation
  - Overhead to maintain data structure of page table

Paging Model of Logical and Physical Memory



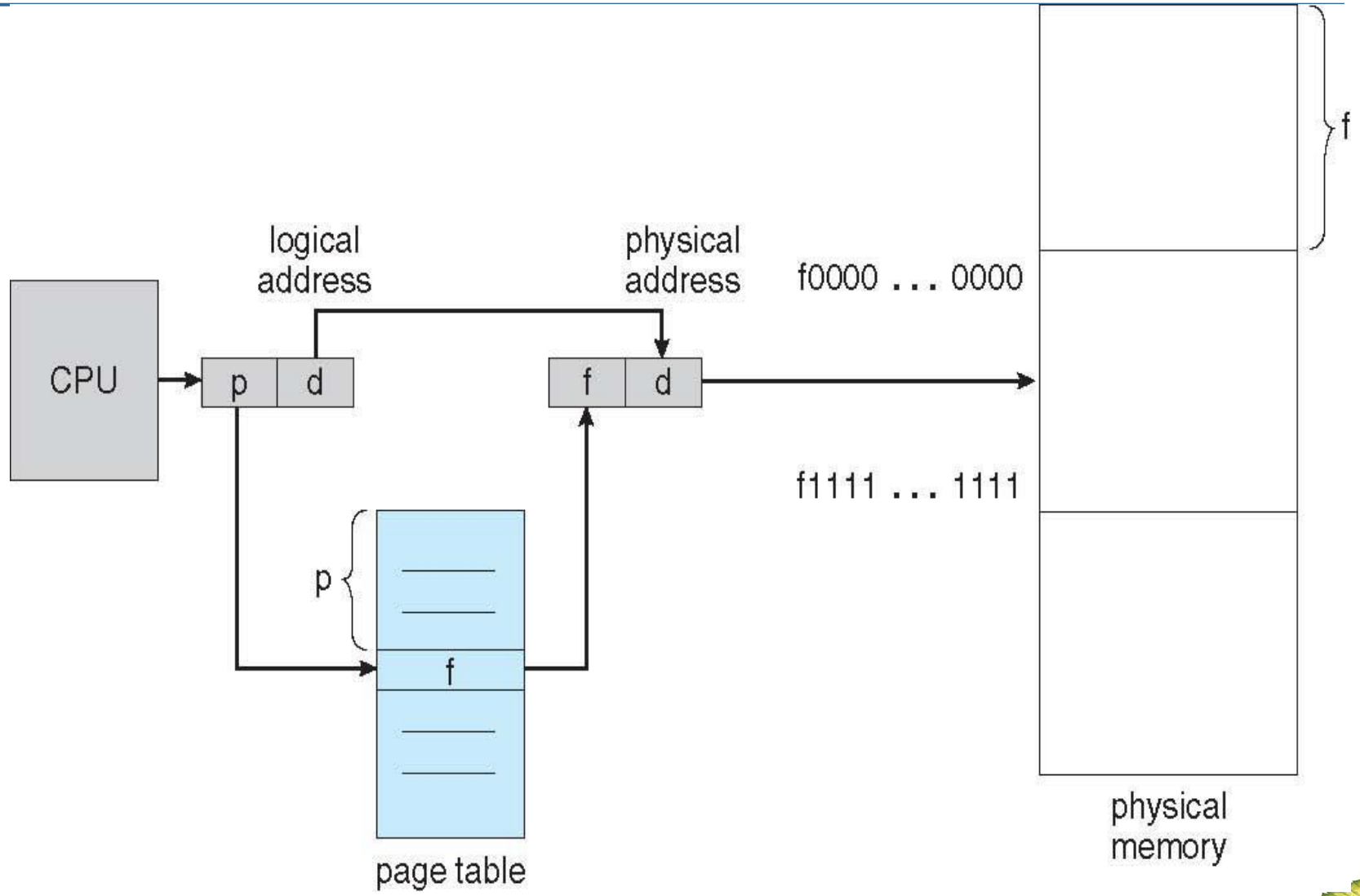
Address Translation Scheme

- Address generated by CPU is divided into:
  - **Page number (*p*)** – used as an index into a **page table** which contains base address of each page in physical memory
  - **Page offset (*d*)** – combined with base address to define the physical memory address that is sent to the memory unit

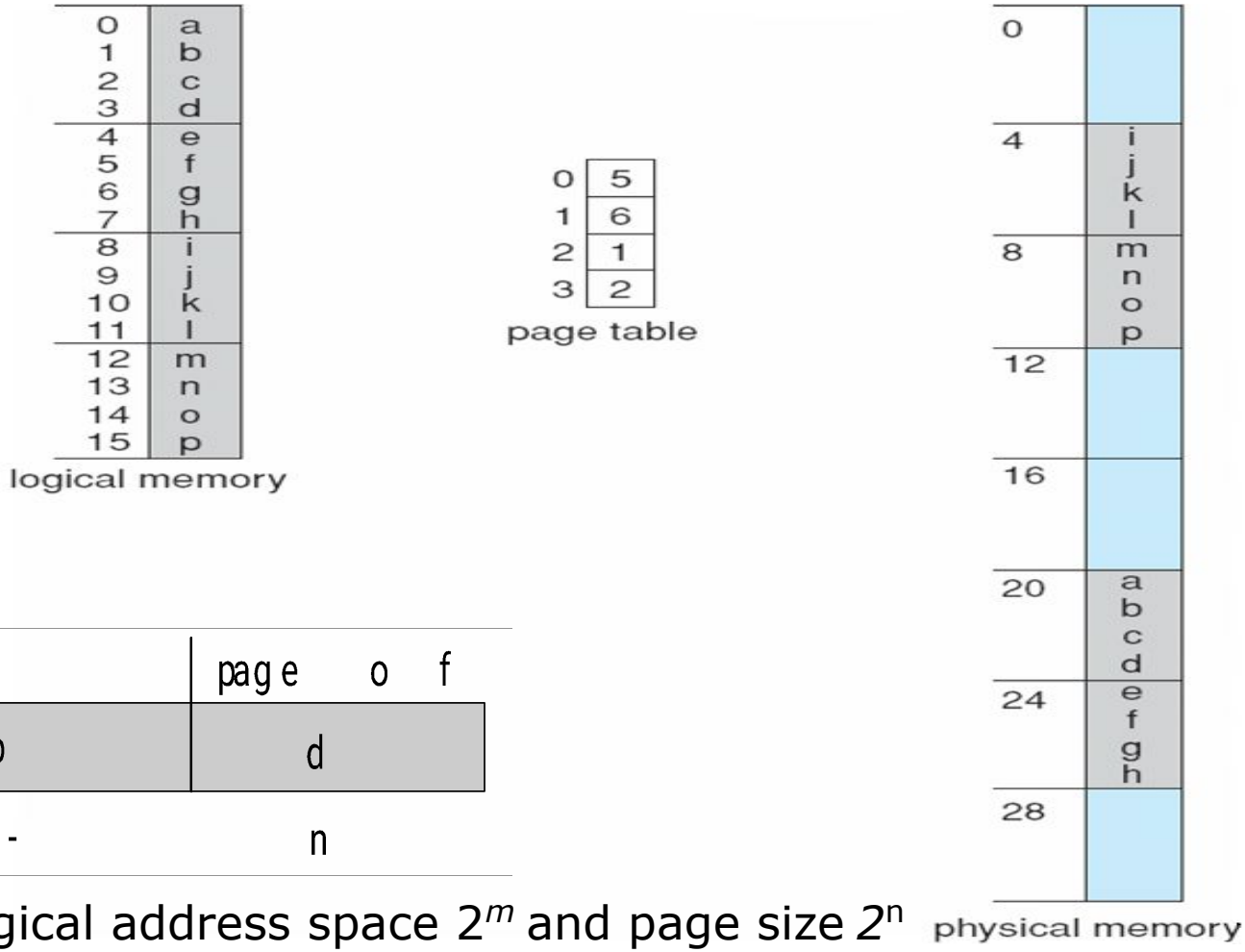


  - For given logical address space 2^m and page size 2^n

Paging Hardware



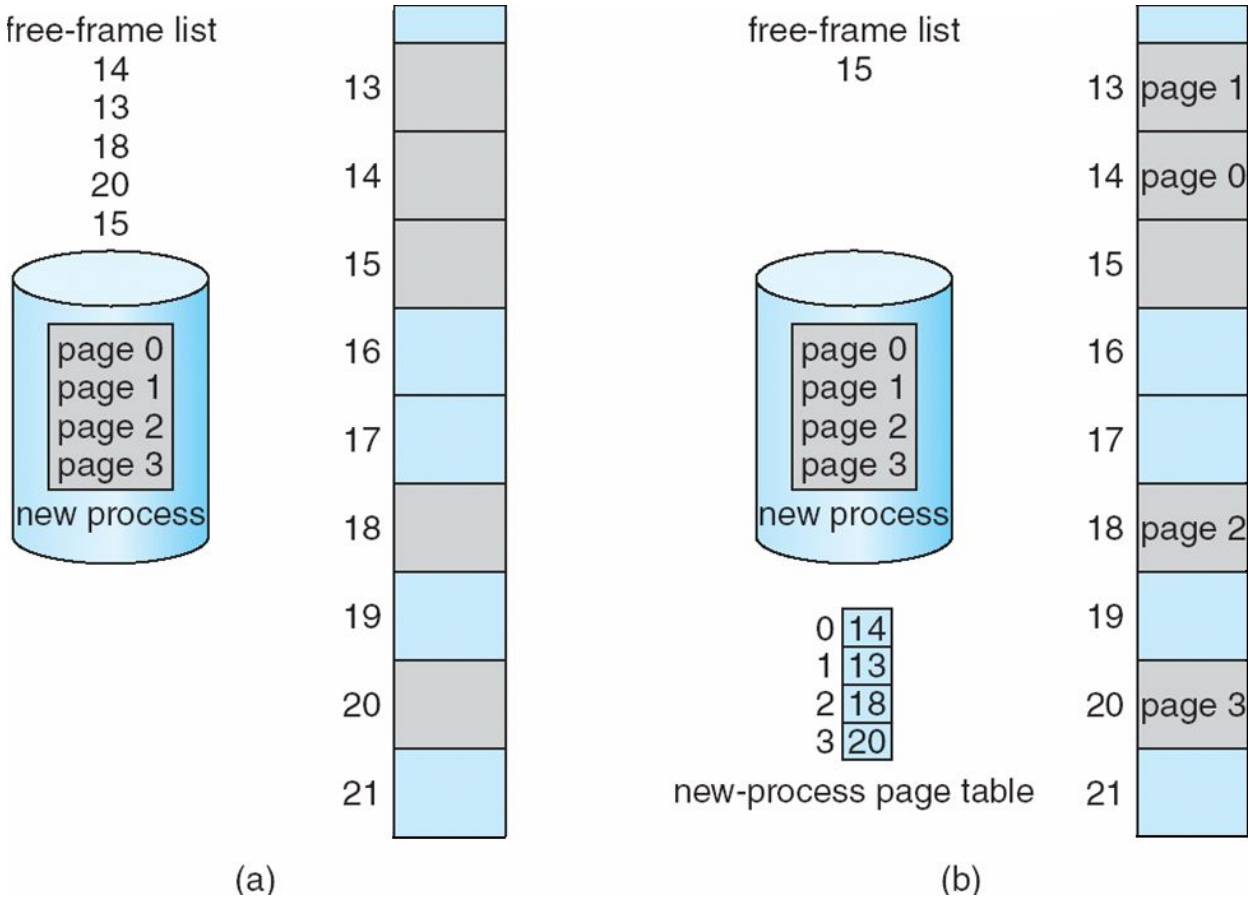
Paging Example



logical address space 2^m and page size 2^n

n=2 and m=4 16-byte logical memory and page size 4-byte

Free Frames



Before allocation

After allocation

Implementation of Page Table

- Page table (data structure) is kept in main memory
- **Page-table base register** (**PTBR**) points to the page table
- **Page-table length register** (**PTLR**) indicates size of the page table
- In this scheme every data/instruction access requires two memory accesses
  - One for the page table and one for the data / instruction
- The hardware implementation of page table can be done by using dedicated registers. But the usage of register for the page table is satisfactory only if page table is small. If page table contain large number of entries then we can use TLB(translation Look-aside buffer), a special, small, fast look up hardware cache.
- The two-memory access problem can be solved by the use of a special fast-lookup hardware cache called **associative memory** or **translation look-aside buffers** (**TLBs**)

Implementation of Page Table (Cont.)

- Some TLBs store **address-space identifiers** (**ASIDs**) in each TLB entry – uniquely identifies each process to provide address-space protection for that process
  - Otherwise need to flush at every context switch
- TLBs typically small (64 to 1,024 entries)
- On a TLB miss, value is loaded into the TLB for faster access next time
  - Replacement policies must be considered
  - Some entries can be **wired down** for permanent fast access

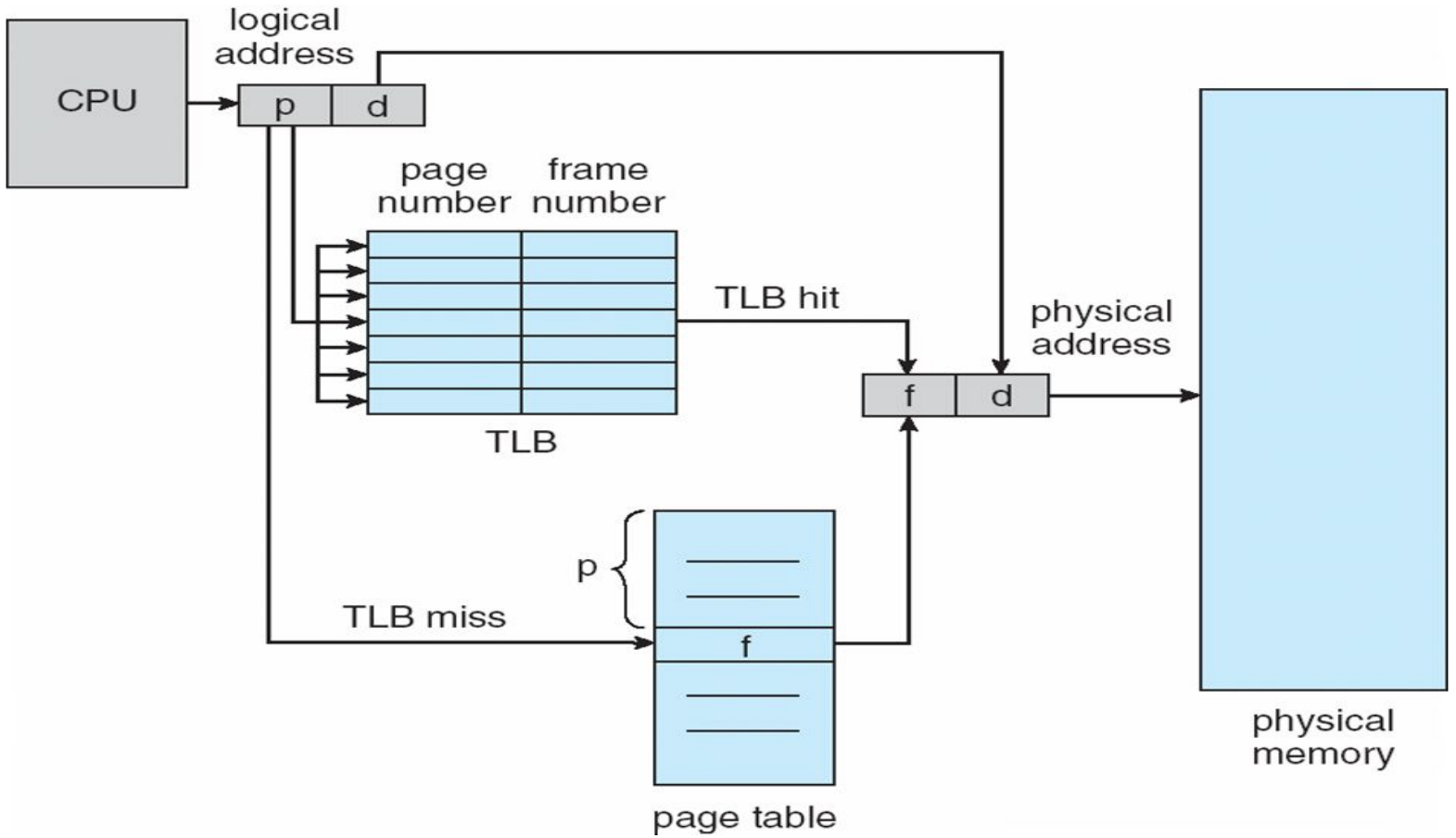
Associative Memory

- Associative memory – parallel search

Page #	Frame #

- Address translation (p, d)
  - If p is in associative register, get frame # out
  - Otherwise get frame # from page table in memory

Paging Hardware With TLB



Paging Protection

- The paging process should be protected by using the concept of insertion of an additional bit called Valid/Invalid bit. Paging Memory protection in paging is achieved by associating protection bits with each page.
- These bits are associated with each page table entry and specify protection on the corresponding page.