Memory Management

Unit 4

Introduction

- During execution, programs and data must be in main memory (at least partially).
- Furthermore, to improve both utilization of the CPU and speed of its response to users, the computer system must keep several processes in memory; that is, we must share memory.
- Since main memory is usually too small to accommodate all the data and programs permanently, the computer system must provide secondary storage to back up main memory.
- **Memory management is the act of managing computer memory.**
- This involves providing ways to allocate portions of memory to programs at request, and freeing it for reuse when no longer needed.
- The management of main memory is critical to the computer system.

Introduction

The entire program and data of a process must be in main memory for the process to execute.

How to keep the track of processes currently being executed?
Which processes to load when memory space is available?
How to load the processes that are larger than main memory?
How do processes share the main memory?

OS component that is responsible for handling these issues is a *memory manager*.

Types of Memory:

Primary Memory (eg. RAM)

- Holds data and programs used by a process that is executing
- Type of memory that a CPU deals with directly.

Secondary Memory (eg. hard disk)

- Non-volatile memory used to store data when a process is not executing.

Memory Management Basics

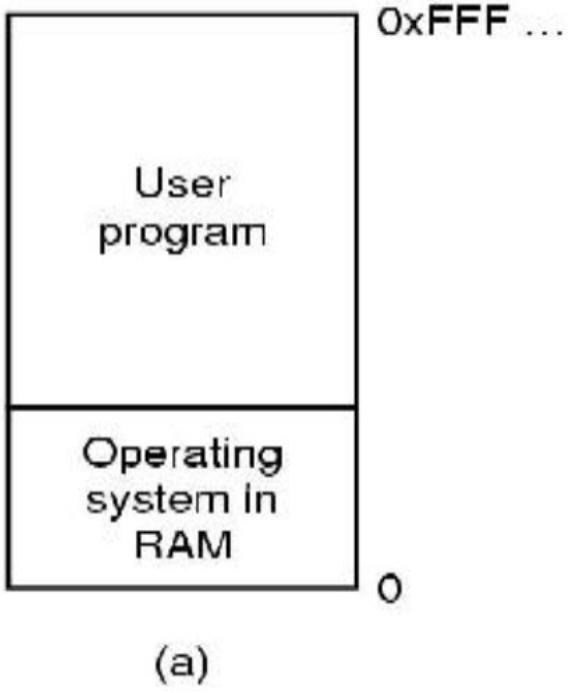
- Don't have infinite RAM
- Do have a memory hierarchy
  - Cache (fast)
  - Main(medium)
  - Disk(slow)
- Memory manager has the job of using this hierarchy to create an **abstraction** (illusion) of easily accessible memory
- Two important memory management function:
  - **Sharing**
  - **Protection**

Memory Management Basics

- In a **uniprogramming system**, Main memory is divided into two parts:
  - one part for the OS
  - one part for the program currently being executed.
- In a **multiprogramming system**, the user part of the memory must be further subdivided to accommodate multiple processes.
- The task of subdivision is carried out dynamically by the Operating System and is known as Memory Management

Monoprogramming Model

- Only one program at a time in main memory;
- Can use whole of the available memory.
- Since only one program or process resides in memory at a time, so sharing and protection is not an issue.
- However, the protection of OS program from the user code is must otherwise the system will crash down.
- This protection is done by a special hardware mechanism such as a dedicated register, called as a **Fence Register.** Also called Limit Register.



(a)

Monoprogramming Model

- The Fence Register is set to highest address occupied by the OS code.
- A memory address generated by user program to access certain memory location is first compared with the fence register's content.
- If the address generated is below the fence, it will be trapped and denied permission.
- Since the modification of fence register is considered as a privileged operation, therefore, only OS is allowed to make any changes to it.

Address Binding (Relocation)

- A program has to go through these three phases:
  - Compilation, Loading and Execution
- The problem that arises is that where should an OS store the results of programs after execution.
- A user specifies in his instruction where to store the result.

  i.e. x=(a+b)× (a-c)
- Such address given by the user, like x, are called symbolic or logical address.
- These addresses need to be mapped to real physical addresses in memory.
- The mechanism is called **address binding.**

Logical versus Physical Address Space

- An address generated by the CPU is commonly referred to as a **logical address**, whereas
- An address seen by the memory unit is commonly referred to as a **physical address**.
- The set of all logical addresses generated by a program is referred to as a logical address space;
- The set of all physical addresses corresponding to these logical addresses is referred to as a physical address space.
- The run-time mapping from virtual (logical) to physical addresses is done by the **memory management unit (MMU)**, which is a hardware device.

Program Relocation

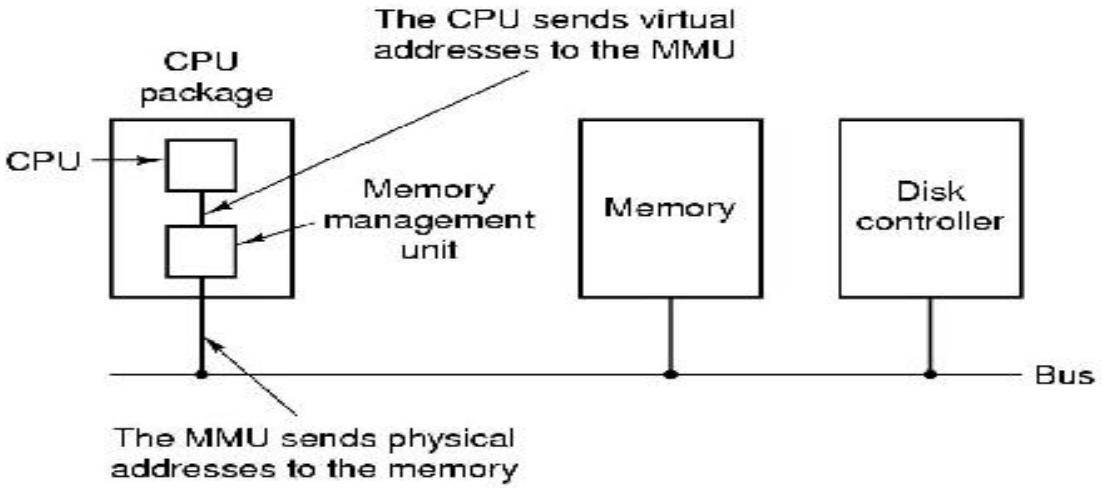
- Relocation is a mechanism to convert the logical address into a physical address.
- To do this, there is a special register in CPU called relocation register (also known as base register).
- Every address used in the program is relocated as:

  effective physical address= Logical address + Contents of Relocation register

- MMU is a special hardware which performs address binding, uses relocation scheme.

Memory Management Unit (MMU)

- MMU generates physical address from virtual address provided by the program by adding the virtual address to the address of relocation register.
- MMU maps virtual addresses to physical addresses and puts them on memory bus



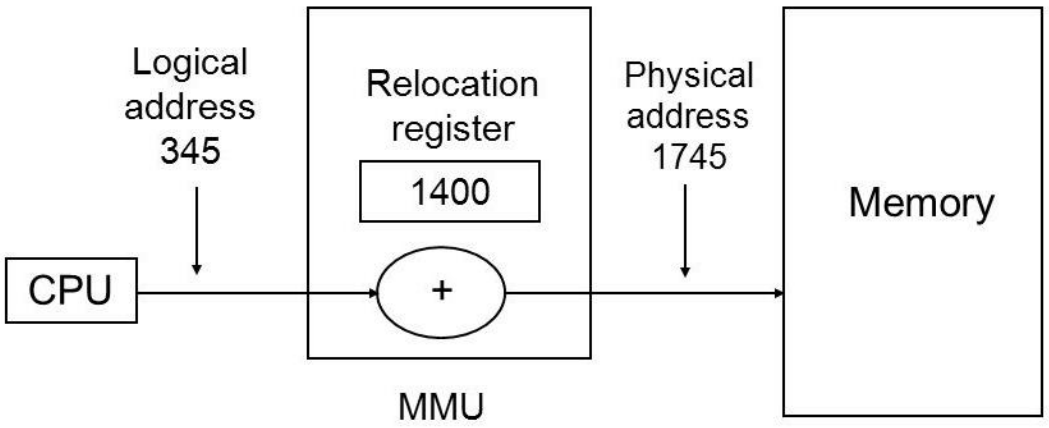
Two basic types of relocation:

Static Relocation:

- Formed during the loading of the program into memory by a loader.

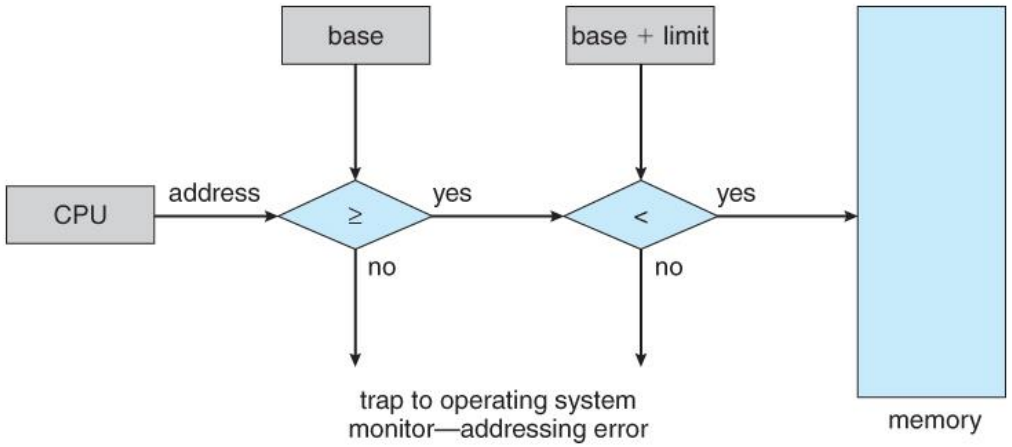
Dynamic Relocation:

- mapping from the virtual address space to physical address is performed at execution time.



Protection

- Providing security from unauthorized usage of memory.
- OS can protect the memory with the help of **base and limit registers**.
- **Base register** consist of the starting address of the next process,
- The **limit register** specifies the boundary of that job.
- That is why the limit register is also called a **fencing register.**
- Fig: Hardware protection mechanism



Memory Allocation Techniques

Two types:

Contiguous Storage Allocation

- Fixed Partition Allocation
- Variable Partition Allocation

Non-Contiguous

- Paging
- Segmentation

Contiguous Storage Allocation

- Main memory must accommodate both the operating system and the user processes
- Main memory is usually divided into two partitions:
  - o Resident operating system, usually held in low memory
  - o User processes then held in high memory
- In this case memory protection must be done.
- *Memory protection* means protecting the operating system from user processes and protecting user processes form one another.
- Memory protection can be done by using *relocation register*.
- Relocation registers (base register) used to protect user processes from each other, and from changing operating-system code and data.

Contiguous Storage Allocation

- *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 to physical address *dynamically*
- A memory resident program occupies a single contiguous block of physical memory.
- The memory is partitioned into block of different sizes to accommodate the programs.
- The partitioning may be:
  - **Fixed Partition allocation**
  - **Variable Partition allocation**

Fixed Partition allocation/Multiprogramming with fixed partition

- In multiprogramming environment, several programs reside in primary memory at a time and the CPU passes its control rapidly between these programs.
- One way to support multiprogramming is to divide the main memory into several partitions each of which is allocated to a single process.

Placement Algorithm with fixed Partitions

- Any process whose size is less than or equal to the partition size can be loaded into an available partition
- If all partitions are full, the operating system can swap a process out of a partition.
- A program may not fit in a partition. The programmer must design the program with overlays.

overlays

- The size of a process is limited to the size of physical memory.
- Process can be larger than the amount of memory allocated to it, a technique called overlays is sometimes used.
- The idea of overlays is to keep in memory only those instructions and data that are needed at any given time.
- When other instructions are needed, they are loaded into space that was occupied previously by instructions i.e. replacing those instructions that are no longer needed.

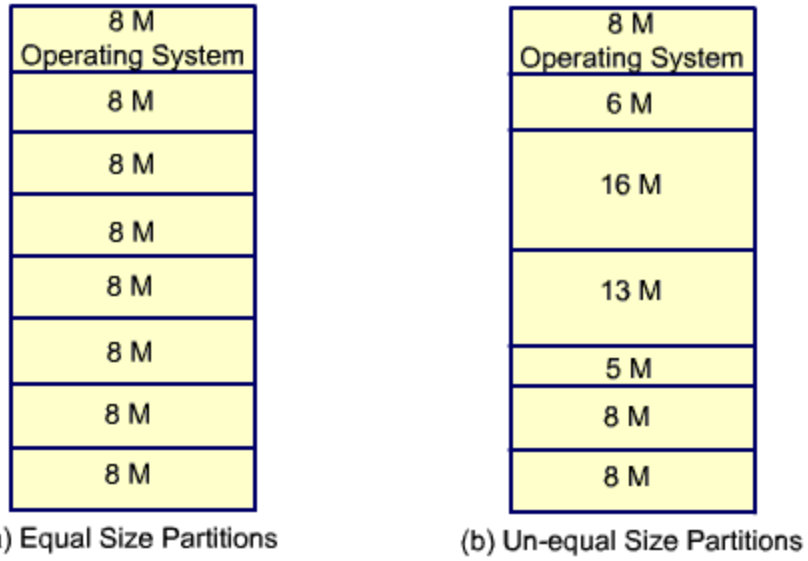
- Let us consider an example of overlays,
- Pass1 70K
- Pass 2 80K
- Symbol table 20K
- Common routines 30K
- To load everything at once, we would require 200K of memory.
- If only 150K is available, we cannot run our process.
- But pass 1 and pass 2 do not need to be in memory at the same time.
- We thus define two overlays:
- Overlay A is the symbol table, common routines, and pass 1, and overlay B is the symbol table, common routines, and pass 2.

Equal-size fixed partitions

- Because all partitions are of equal size, it does not matter which partition is used.

Unequal-size fixed partitions

- Can assign each process to the smallest partition within which it will fit.
- Processes are assigned in such a way as to minimize wasted memory within a partition.



(a) Equal Size Partitions

(b) Un-equal Size Partitions

Fixed partitioning Implemented in two ways:

- Dedicated partition for each process (Absolute translation)
- Maintaining a single queue (Relocatable translation)

Absolute Translation (Maintaining multiple input queues):

- In this scheme, separate input queue is maintained for each partition.
- When a process arrives it is put into the input queue for the smallest partition large enough to hold it.

Problem:

- If a process is ready to run and its partition is occupied then that process has to wait even if other partitions are available i.e. wastage of storage.

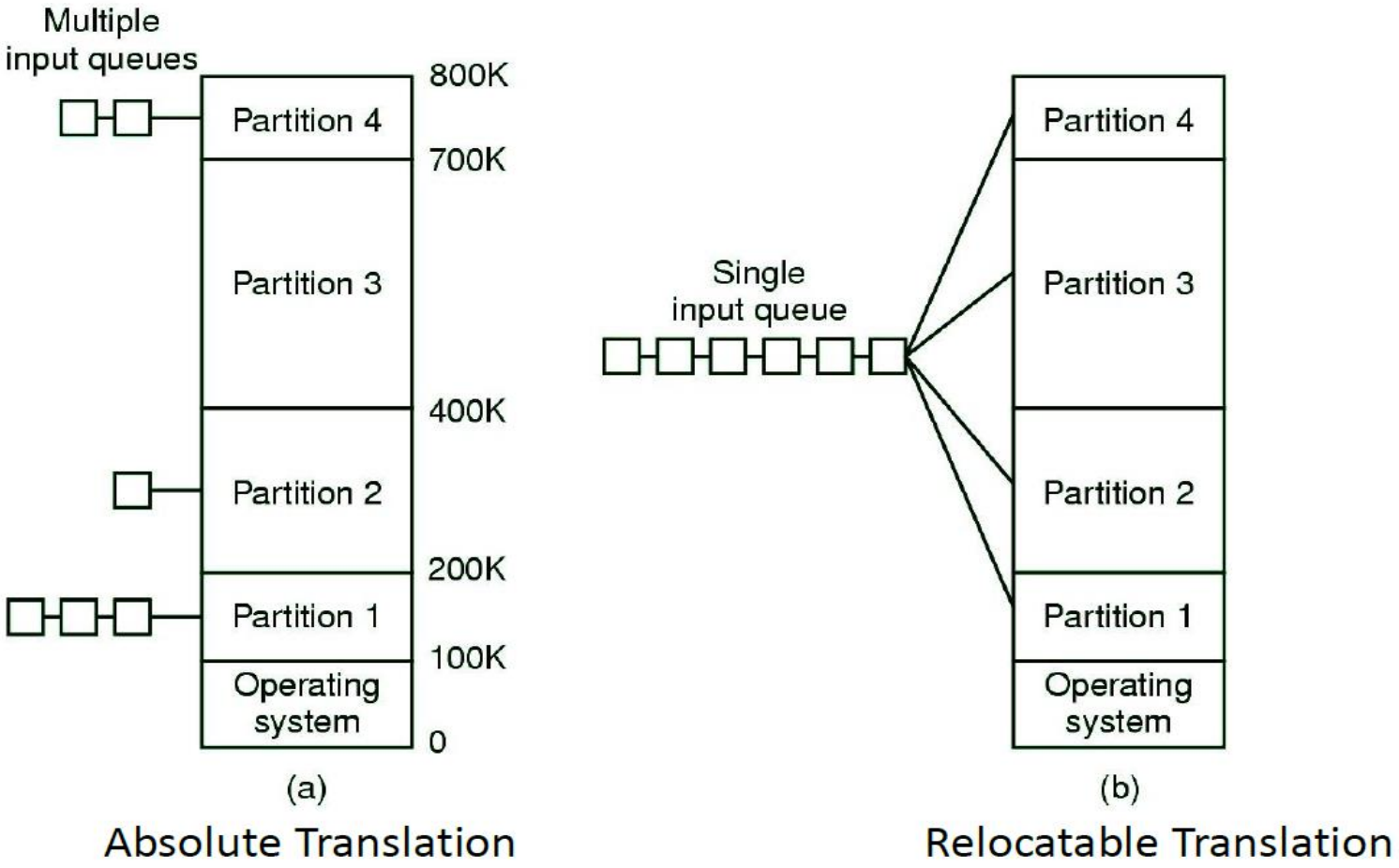
Relocatable Translation (Maintaining a single queue):

- Another strategy is to maintain a single queue for all partitions whenever a partition becomes free, the job closest to the front of the queue that fits in it could be loaded into the empty partition and run.
- Since it is undesirable to waste a large partition on a small job, a different strategy is to search the whole input queue whenever a partition becomes free and pick the largest job that fits.
- But this algorithm discriminates against small jobs as being unworthy of having a whole partition.

Problem:

- Eliminate the absolute problems but implementation is complex.
- Wastage of storage when many processes are small.

Relocatable Translation (Maintaining a single queue):



Advantages of fixed partition

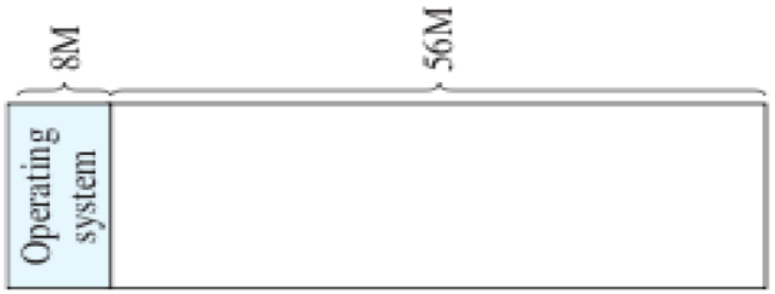
- Implementation of this allocation scheme is simple.
- It supports multiprogramming.

Disadvantages of fixed partitioning

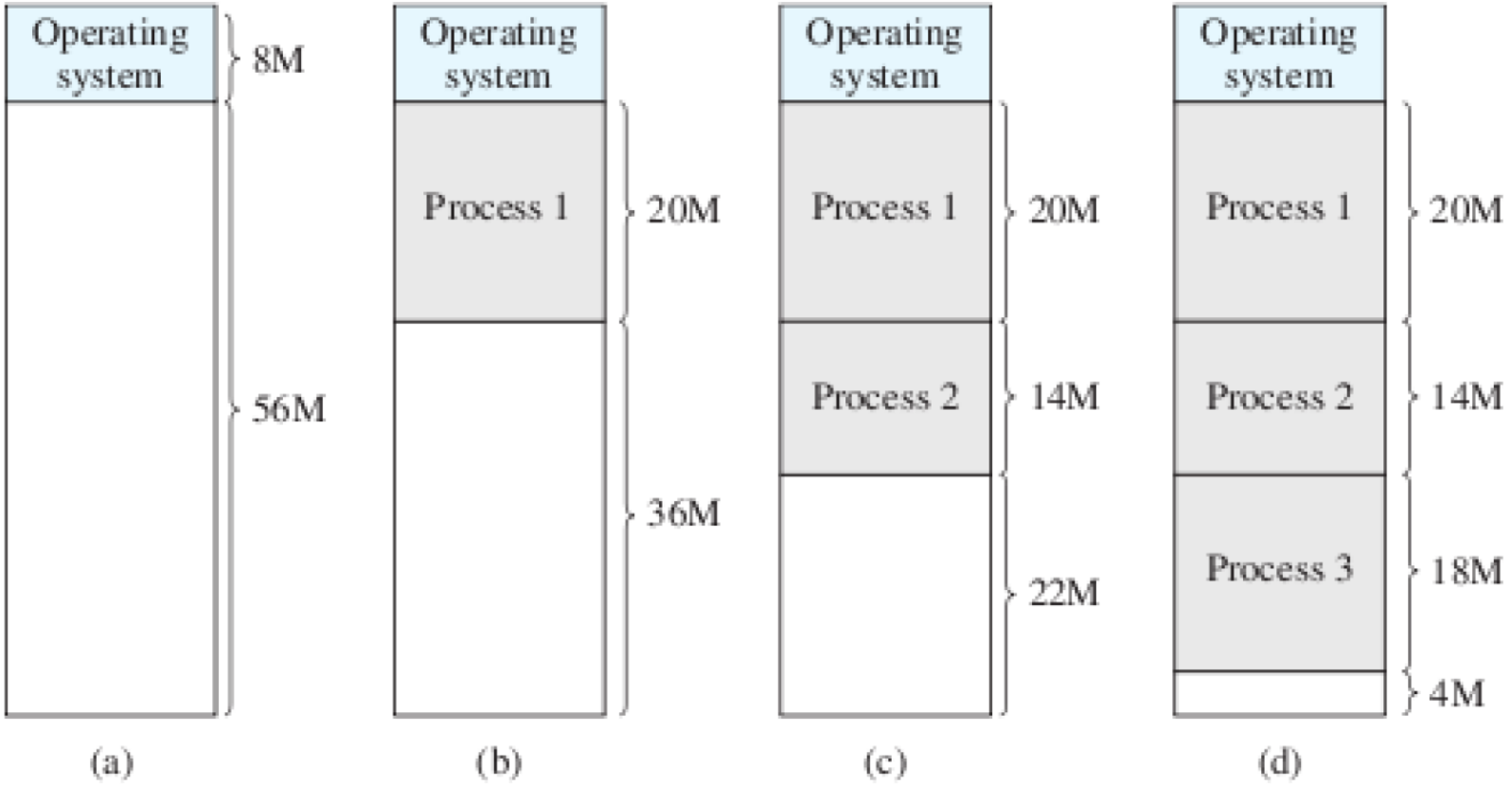
- No single program (process) may exceed the size of the largest partition in a given system.
- The Main memory use is inefficient. Any program, no matter how small, occupies an entire partition. This is called ***internal fragmentation***.
- Not suitable for systems in which process memory requirements not known ahead of time; i.e. timesharing systems.

Multiprogramming with Variable Partitions

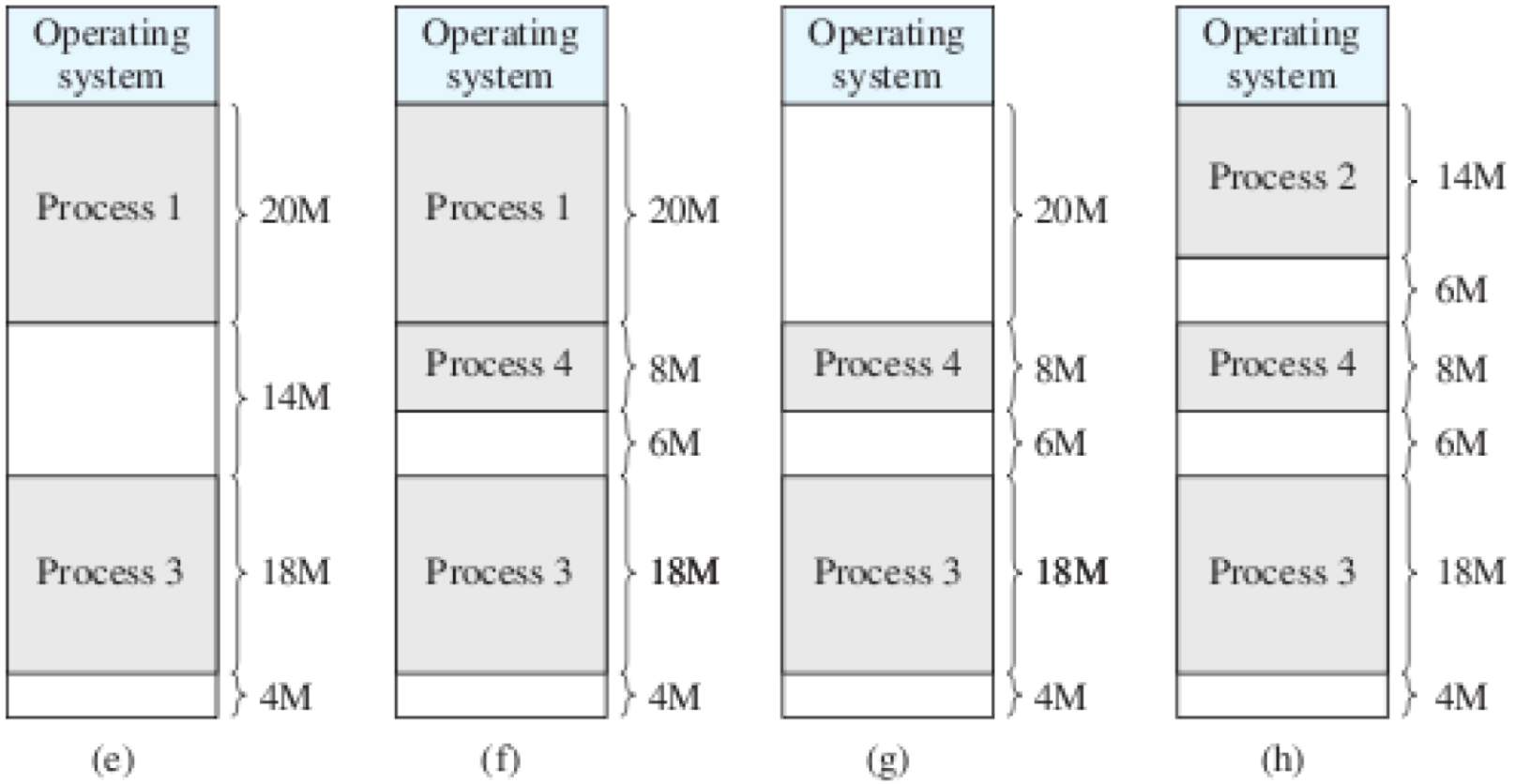
- To overcome some of the difficulties with fixed partitioning, an approach known as dynamic partitioning was developed.
- The partitions are of variable length and number.
- When a process is brought into main memory, it is allocated exactly as much memory as it requires and no more.
- An example, using 64 Mbytes of main memory, is shown in Figure



Coalescing vs Compaction



Coalescing vs Compaction



Coalescing vs Compaction

- Eventually it leads to a situation in which there are a lot of small holes in memory.
- As time goes on, memory becomes more and more fragmented, and memory utilization declines.
- This phenomenon is referred to as ***external fragmentation***, indicating that the memory becomes increasingly fragmented as shown in the above figure.
- In the above figure let us say that a process of size greater then 7M arise but this process cannot be allocated into the memory though there is availability of the memory in total is 6+6+4=16M.
- Following two activities should be taken place, to reduce wastage of memory:

(a) Coalescing

(b) Compaction

Coalescing vs Compaction

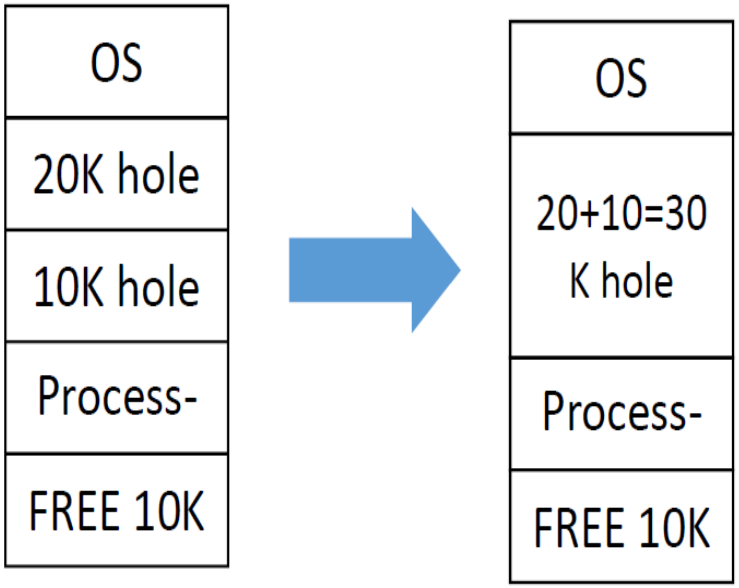
a) Coalescing

- The process of merging two adjacent holes to form a single larger hole is called coalescing.

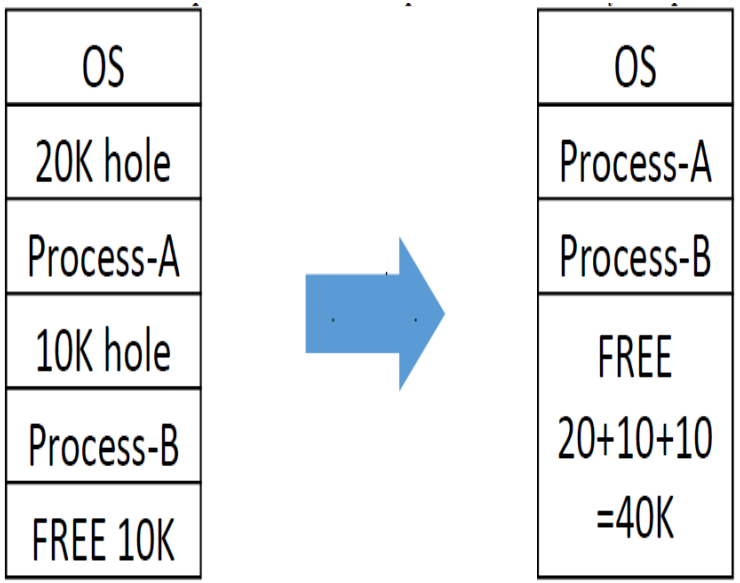
b) Compaction

- Even when holes are coalesced, no individual hole may be large enough to hold the job, although the sum of holes is larger than the storage required for a process.
- It is possible to combine all the holes into single big unit by moving all the processes downward as far as possible; this technique is called memory compaction.

Coalescing vs Compaction



(a) Coalescing



(b) Compaction

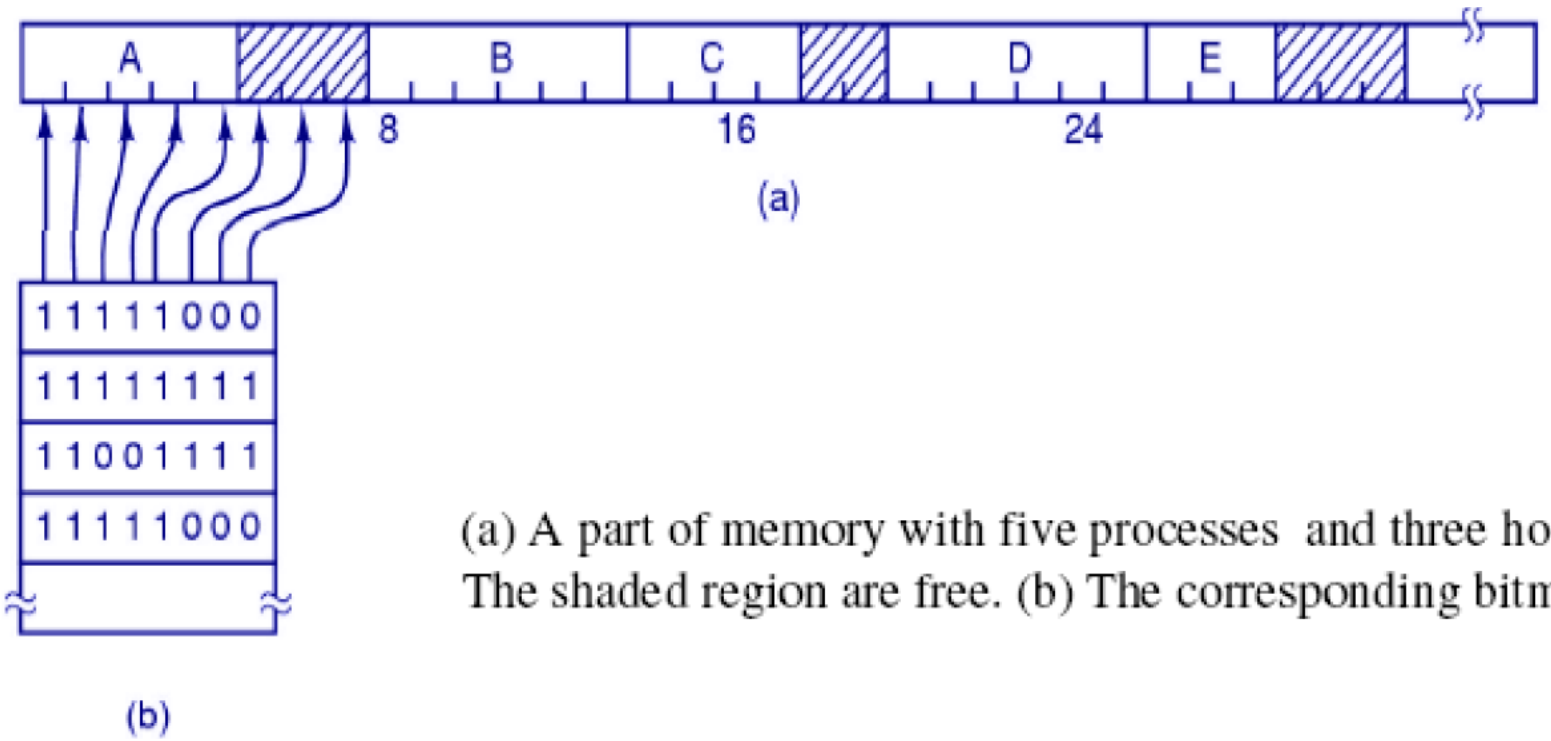
Fixed vs. Variable Partitioning

Fixed Partition	Variable partition
• It is the OS that decides the partition size only once at the system boot time.	• OS has to decide about partition size, every time a new process is chosen by long-term scheduler.
• Here, the degree of multiprogramming is fixed.	• Here, the degree of multiprogramming will vary depending on program size.
• It leads to internal fragmentation.	• It leads to external fragmentation.

Memory Management with Bitmaps:

- When memory is assigned dynamically, the operating system must manage it.
- With a bitmap, memory is divided up into allocation units, perhaps as small as a few words and perhaps as large as several kilobytes.
- Corresponding to each allocation unit is a bit in the bitmap, which is 0 if the unit is free and 1 if it is occupied (or vice versa).
- Figure below shows part of memory and the corresponding bitmap.
- The size of the allocation unit is an important design issue.
- The smaller the allocation unit, the larger the bitmap.

Memory Management with Bitmaps:



(a) A part of memory with five processes and three holes. The shaded region are free. (b) The corresponding bitmap.

Memory Management with Bitmaps:

Advantage

- A bitmap provides a simple way to keep track of memory words in a fixed amount of memory because the size of the bitmap depends only on the size of memory and the size of the allocation unit.

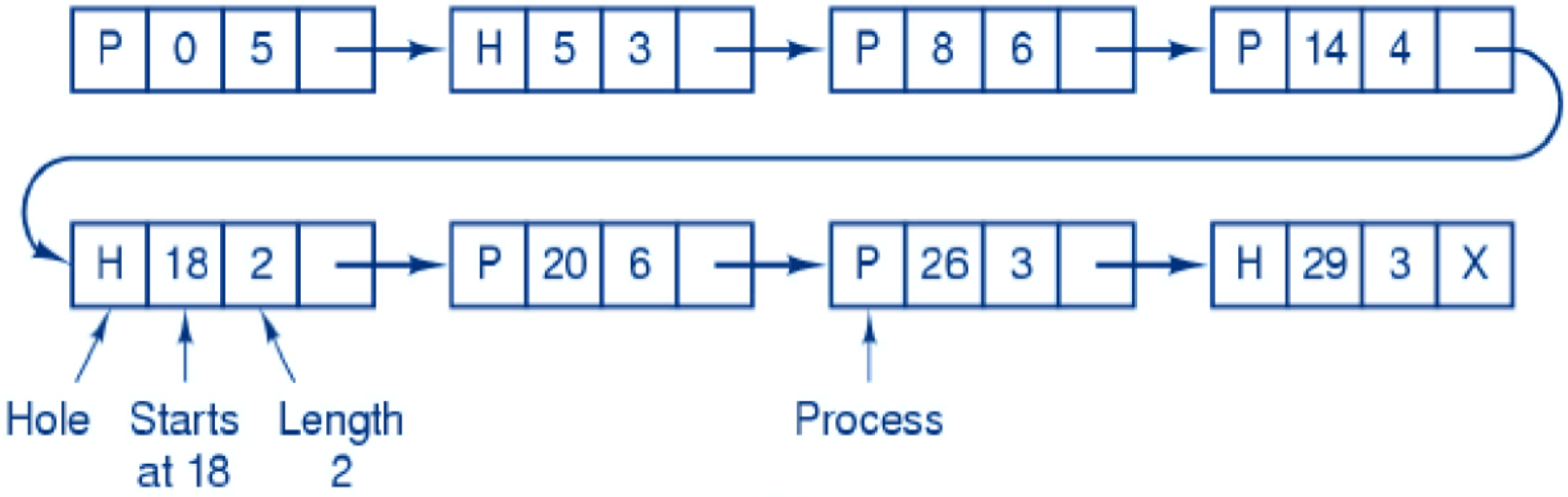
Disadvantage

- The main problem with it is that when it has been decided to bring a k unit process into memory, the memory manager must search the bitmap to find a run of k consecutive 0 bits in the map.
- Searching a bitmap for a run of a given length is a slow operation.

Memory Management with Linked Lists

- Another way of keeping track of memory is to maintain a linked list of allocated and free memory segments,
- Where a segment is either a process or a hole between two processes.
- Each entry in the list specifies a hole (H) or process (P), the address at which it starts, the length, and a pointer to the next entry.
- In this example, the segment list is kept sorted by address.
- Sorting this way has the advantage that when a process terminates or is swapped out, updating the list is straightforward.

Memory Management with Linked Lists



Memory Management with Linked Lists

- A terminating process normally has two neighbors (except when it is at the very top or very bottom of memory).
- These may be either processes or holes, leading to the four combinations shown in fig

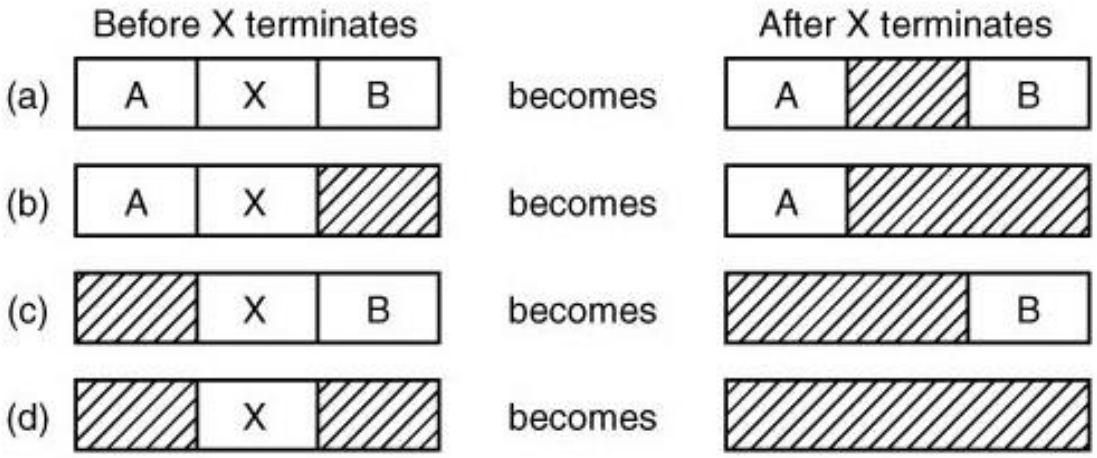


Fig: Four neighbor combinations for the terminating process, X.

Partition Selection Algorithms:

- When the processes and holes are kept on a list sorted by address, several algorithms can be used to allocate memory for a newly created process (or an existing process being swapped in from disk).
- We assume that the memory manager knows how much memory to allocate.

1. First fit:

- The memory manager allocates the first hole that is big enough.
- It stops the searching as soon as it finds a free hole that is large enough.
- The hole is then broken up into two pieces, one for the process and one for unused memory.

Advantages: it is a fast algorithm because it searches as little as possible.

Disadvantages: not good in terms of storage utilization.

Partition Selection Algorithms:

2.Next Fit:

- It works the same way as first fit, except that it keeps track of where it is whenever it finds a suitable hole.
- The next time it is called to find a hole, it starts searching the list from the place where it left off last time, instead of always at the beginning, as first fit does.

3. Best fit:

- Allocate the smallest hole that is big enough.
- Best fit searches the entire list and takes the smallest hole that is big enough to hold the new process.
- Best fit try to find a hole that is close to the actual size needed.

Advantages: more storage utilization than first fit.

Disadvantages:

- slower than first fit because it requires searching whole list at time.
- Creates a tiny hole that may not be used.

Partition Selection Algorithms:

4. Worst fit:

- Allocate the largest hole.
- It search the entire list, and takes the largest hole, rather than creating a tinny hole it produces the largest leftover hole, which may be more useful.

Advantages:

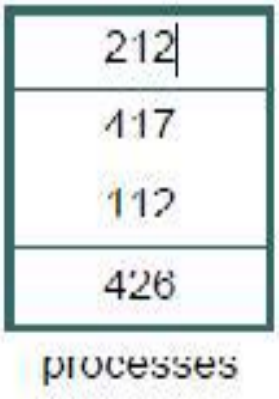
- some time it has more storage utilization than first fit and best fit.

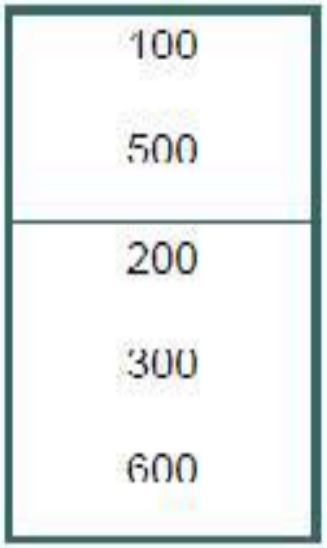
Disadvantages:

- not good for both performance and utilization.

Q: Given the memory partitions of 100K, 500K, 200K, 300K and 600K (in order), how would each of the First-fit, Best-fit, and Worst-fit algorithms place processes of 212K, 417K,112K, and 426K (in order)?

Which algorithm makes the most efficient use of memory?





- First-fit: search the list of available memory and allocate the first block that is big enough

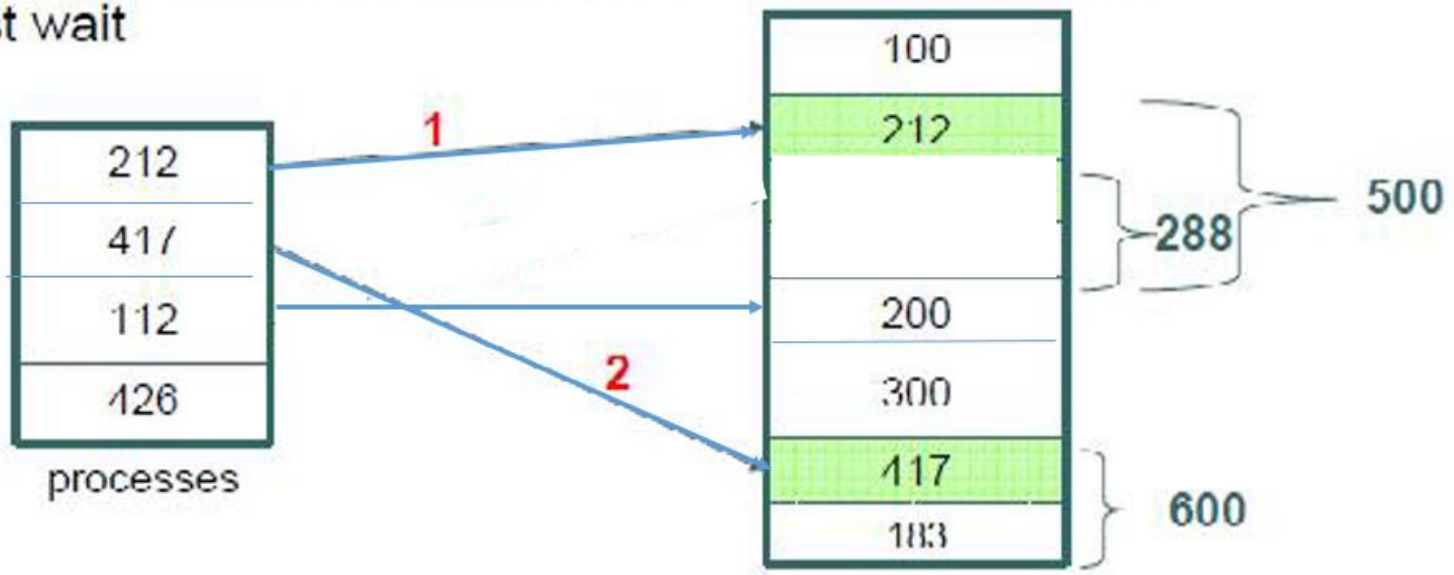
Processes placement:

212 K → 500 K partition

417 K → 600 K partition

112 K → 288 K partition

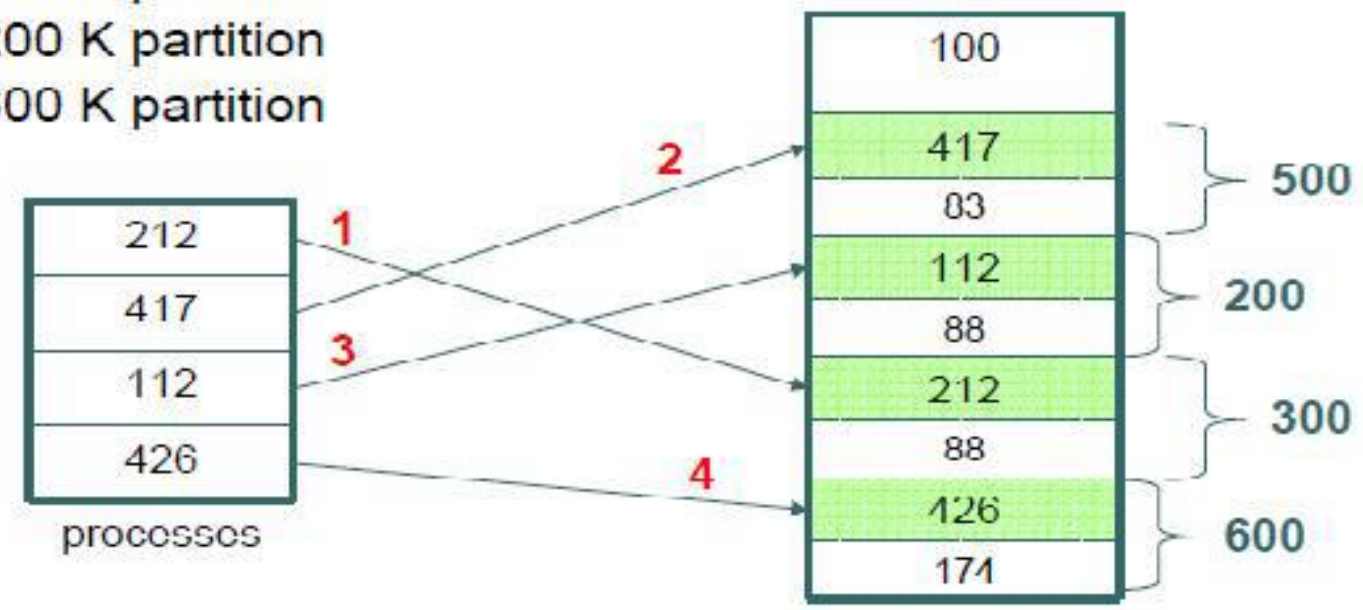
426 K must wait



- Best-fit: search the entire list of available memory and allocate the smallest block that is big enough

Processes placement:

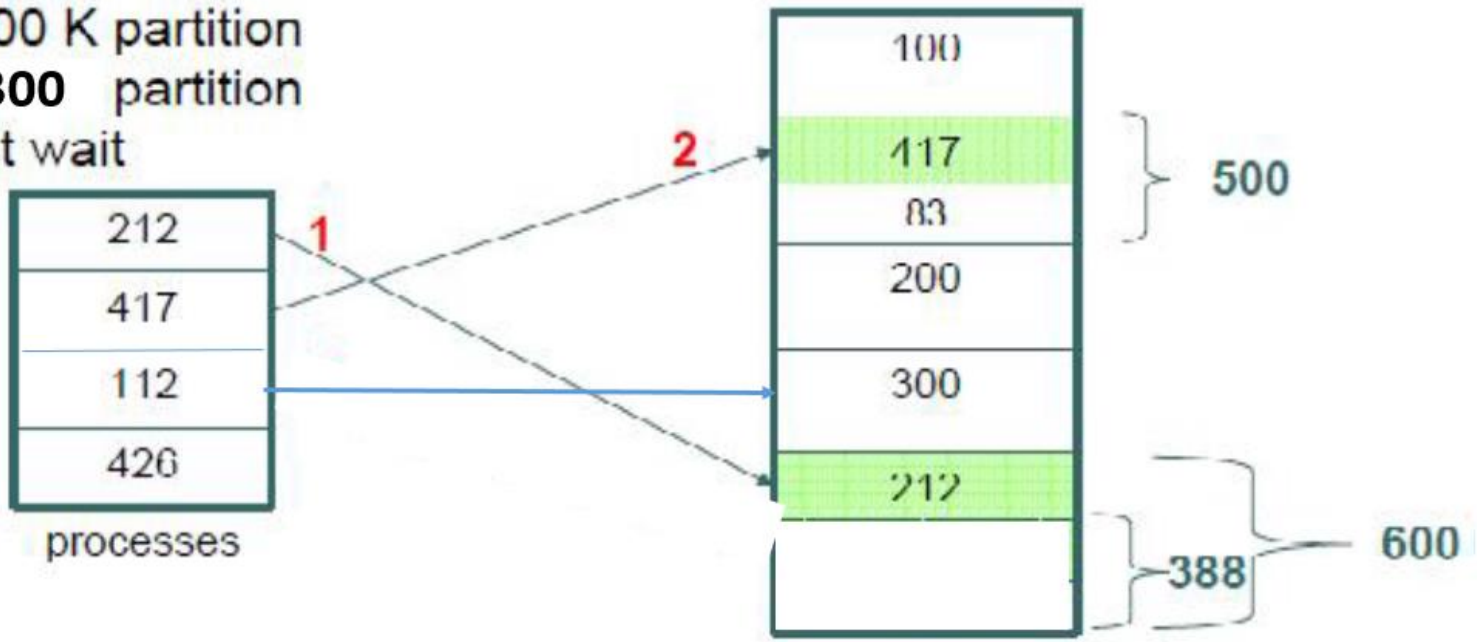
212 K → 300 K partition
417 K → 500 K partition
112 K → 200 K partition
426 K › 600 K partition



- Worst-fit: search the entire list of available memory and allocate the largest block.

Processes placement:

212 K → 600 K partition
417 K → 500 K partition
112 K → **300** partition
426 K must wait



Q: Consider a swapping system in which memory consists of the following hole sizes in memory order: 10K, 4K, 20K, 18K, 7K, 9K, 12K, and 15K. Which hole is taken for successive segment requests of:

I) 12K

II) 10K

III) 9K

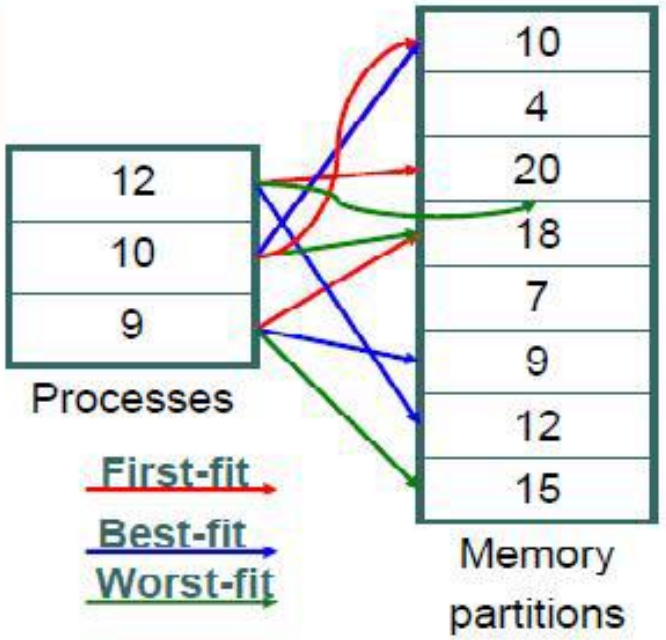
for

(a) First-fit?

(b) Best-fit?

(c) Worst-fit?

	First-Fit	Best-Fit	Worst-Fit
12	20	12	20
10	10	10	18
9	18	9	15



Memory Management with Buddy-system:

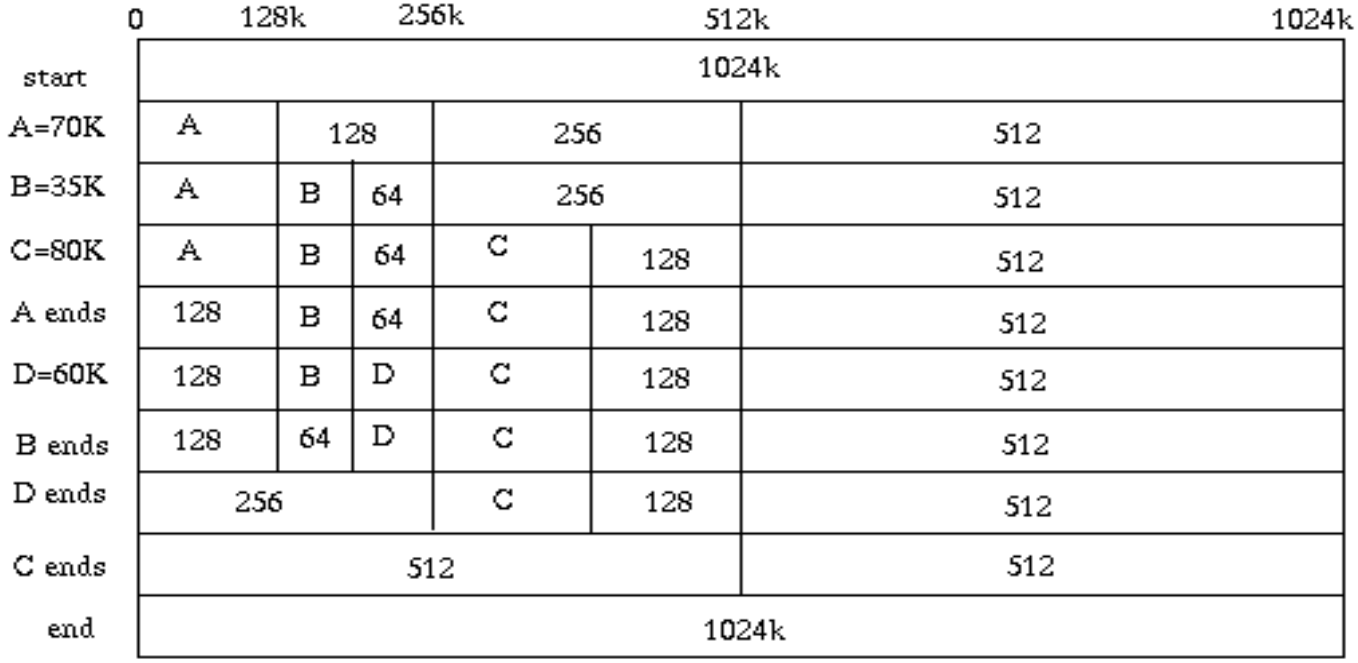
- Both fixed and dynamic partitioning schemes have drawbacks.
- A fixed partitioning scheme limits the number of active processes and may use space inefficiently if there is a poor match between available partition sizes and process sizes.
- A dynamic partitioning scheme is more complex to maintain and includes the overhead of compaction.
- An interesting compromise is the buddy system .
- In a buddy system, memory blocks are available of size 2^K words, $L \leq K \leq U$, where,
- 2^L = smallest size block that is allocated
- 2^U =largest size block that is allocated; generally 2^U is the size of the entire memory available for allocation.

Memory Management with Buddy-system:

- In a buddy system, the entire memory space available for allocation is initially treated as a single block whose size is a power of 2.
- When the first request is made, if its size is greater than half of the initial block then the entire block is allocated.
- Otherwise, the block is split in two equal companion buddies.
- If the size of the request is greater than half of one of the buddies, then allocate one to it.
- Otherwise, one of the buddies is split in half again.
- This method continues until the smallest block greater than or equal to the size of the request is found and allocated to it.
- In this method, when a process terminates the buddy block that was allocated to it is freed.
- Whenever possible, an unallocated buddy is merged with a companion buddy in order to form a larger free block.

Memory Management with Buddy-system:

- Two blocks are said to be companion buddies if they resulted from the split of the same direct parent block.
- The following fig. illustrates the buddy system at work, considering a 1024k (1-megabyte) initial block and the process requests as shown at the left of the table.



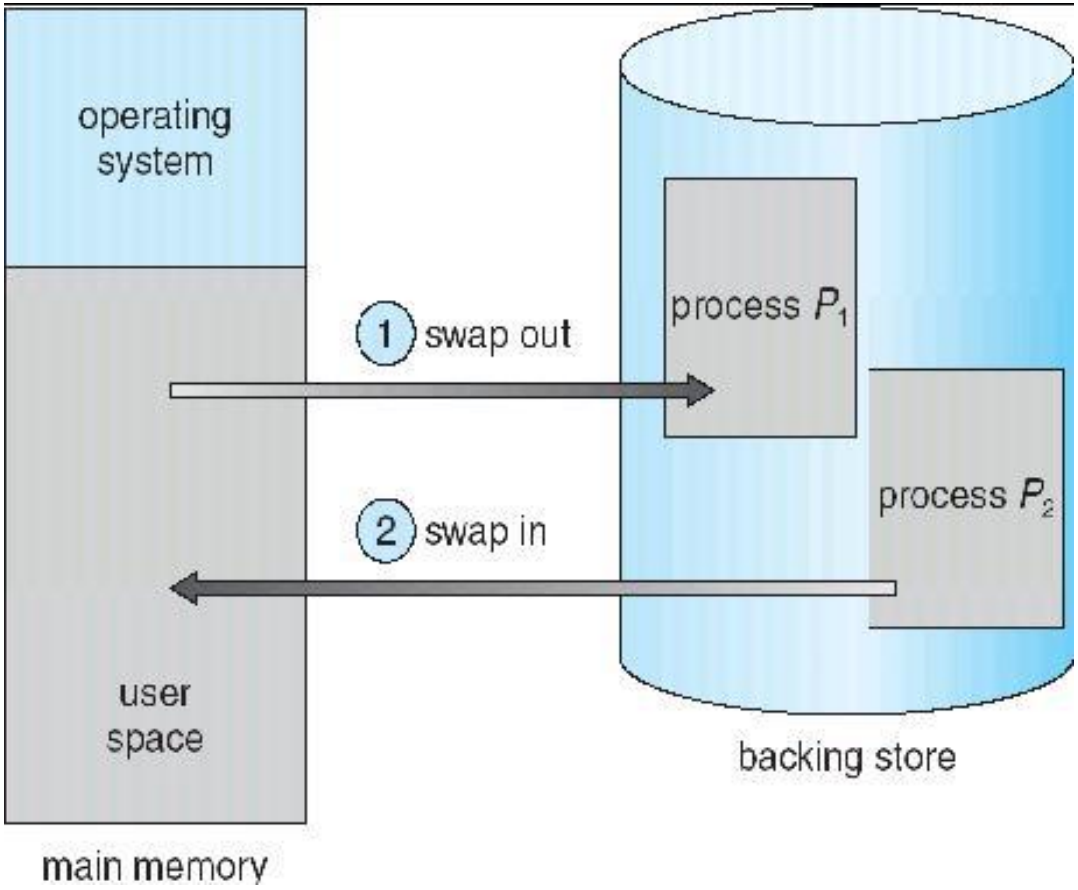
Swapping

- If there is not enough main memory to hold all the currently active processes, the excess processes must be kept on the disk and brought in to run dynamically.
- Swapping consists of moving processes from main memory and disk.
- Relocation may be required during swapping.
- **Backing store is** disk large enough to accommodate copies of all memory images which provides 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
- A process, can be swapped temporarily out of memory to a backing store, and then brought back into memory for continued execution.

Roll out, roll in



Roll out, roll in

- A variant of this swapping policy is used for priority-based scheduling algorithms.
- If a higher-priority process arrives and wants service, the memory manager can swap out the lower-priority process so that it can load and execute the higher priority process.
- When the higher priority process finishes, the lower- priority process can be swapped back in and continued.
- This variant of swapping is sometimes called rollout, roll in.
- A process is swapped out will be swapped back into the same memory space that it occupies previously.
- If binding is done at assembly or load time, then the process cannot be moved to different location.
- However, If execution-time binding is being used, then it is possible to swap a process into a different memory space.

Roll out, roll in

- Assume a multiprogramming environment with a round robin CPU-scheduling algorithm.
- When a quantum expires, the memory manager will start to swap out the process that just finished, and to swap in another process to the memory space that has been freed.
- When each process finishes its quantum, it will be swapped with another process.
- The context-switch time in such a swapping system is fairly high.

Non-contiguous Memory allocation:

- Fragmentation is a main problem in contiguous memory allocation.
- We have seen a method called compaction to resolve this problem.
- System efficiency gets reduced. So, a better method to overcome the fragmentation problem is to make our logical address space noncontiguous.
- Consider a system in which before applying compaction, there are holes of size 1K and 2K. If a new process of size 3K wants to be executed then its execution is not possible without compaction.
- An alternative approach is divide the size of new process P into two chunks of 1K and 2K to be able to load them into two holes at different places.

Non-contiguous Memory allocation:

1. If the chunks have to be of same size for all processes ready for the execution then the memory management scheme is called **PAGING.**
2. If the chunks have to be of different size in which process image is divided into logical segments of different sizes then this method is called **SEGMENTATION.**
3. If the method can work with only some chunks in the main memory and the remaining on the disk which can be brought into main memory only when its required, then the system is called **VIRTUAL MEMORY MANAGEMENT SYSTEM.**

Virtual memory

- It is desirable to be able to execute a process whose logical address space is larger than the available physical address space.
- The programmer can make such a process executable by restructuring it using overlays, but doing so is generally a difficult programming task.
- Virtual memory is a technique to allow a large logical address space to be mapped onto a smaller physical memory.
- Virtual memory allows extremely large process to be run, and also allows the degree of multiprogramming to be raised, increasing CPU utilization.
- Further, it frees application programmers from worrying about memory availability.

Virtual memory

- The basic idea behind virtual memory is that the combined size of the program, data, and stack may exceed the amount of physical memory available for it.
- The operating system keeps those parts of the program currently in use in main memory, and the rest on the disk(swap area called virtual memory).
- For example, a 512-MB program can run on a 256-MB machine by carefully choosing which 256 MB to keep in memory at each instant, with pieces of the program being swapped between disk and memory as needed.

Virtual memory

- Virtual storage is not a new concept; this concept was devised by fortherigham in 1961 and used in Atlas computer system.
- But the common use in OS is the recent concept, all microprocessor now support virtual memory.
- Virtual memory can be implemented by two most common used methods:

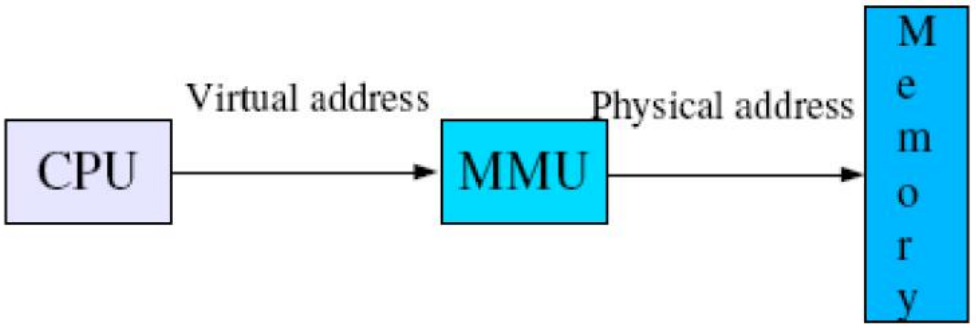
❑ Paging

❑ Segmentation

❑ Or mixed strategy of both.

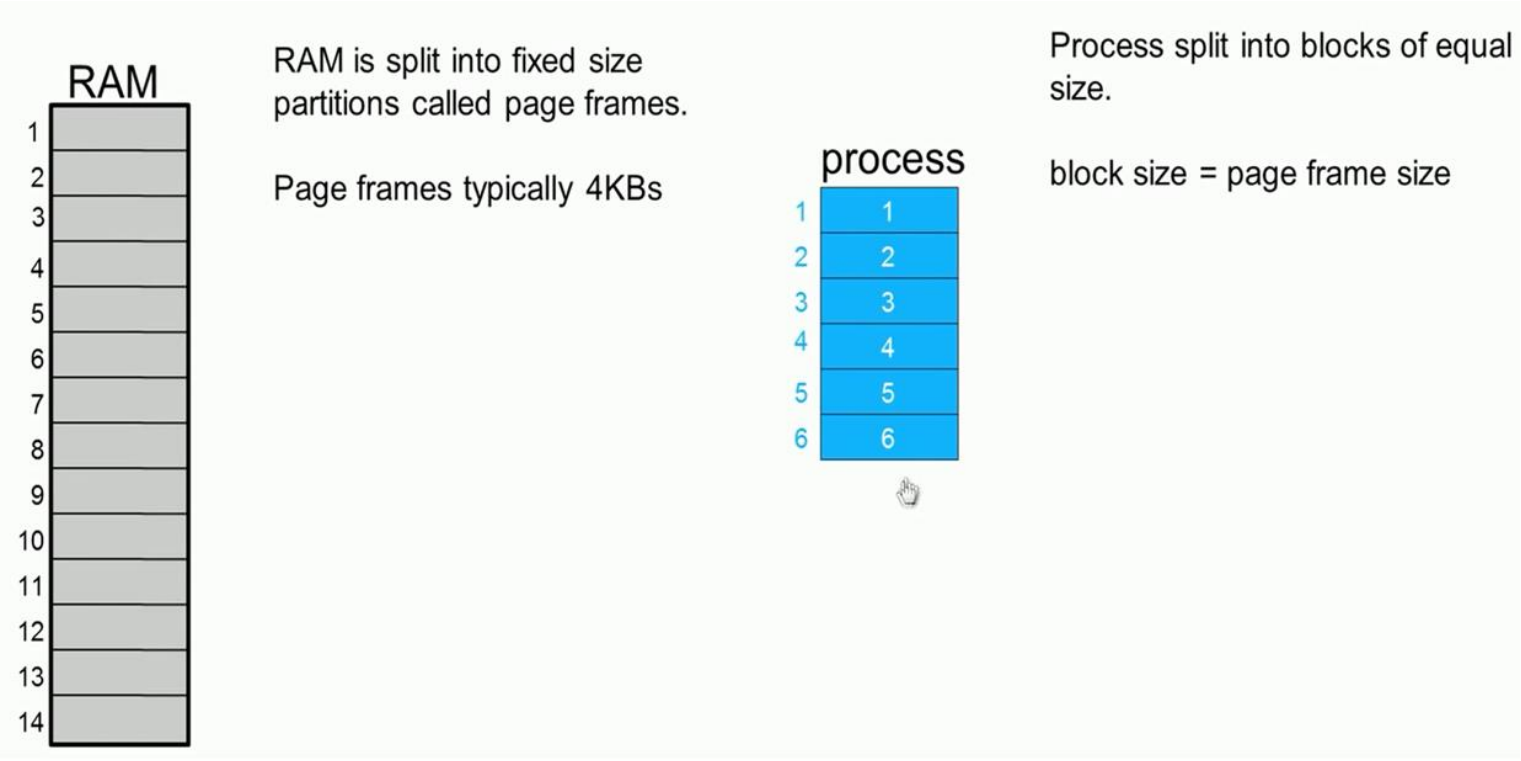
Paging

- Most virtual system uses a techniques called paging that permits the physical address space of a process to be non-contiguous.
- Program-generated addresses are called **virtual addresses** and form the **virtual address space.**
- When virtual memory is used, the virtual addresses requested by cpu do not go directly to the memory bus.
- Instead, they go to an MMU (Memory Management Unit) that maps the virtual addresses onto the physical memory addresses as illustrated in Fig



Paging

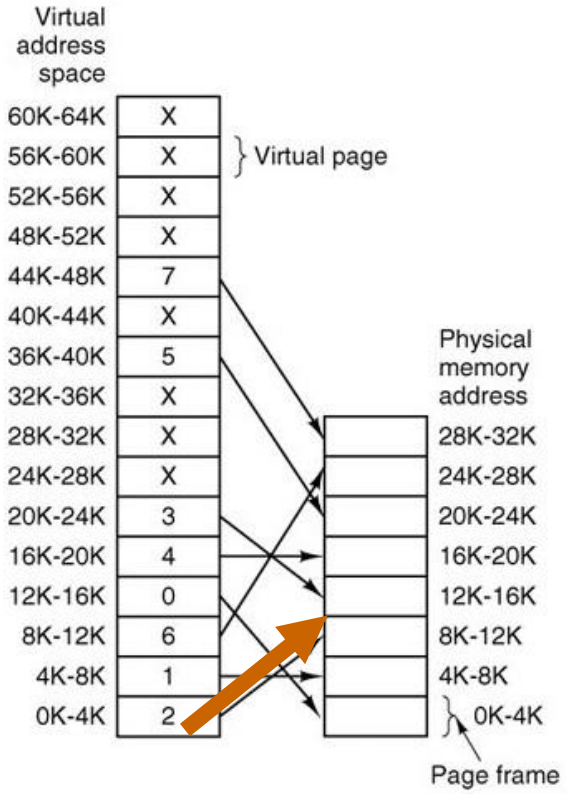
- The basic method for implementing paging involves breaking physical memory into fixed size block called **frames** and breaking logical memory into blocks of the same size called **pages.**



Paging

- A very simple example of how this mapping works is shown in Fig. below.
- In this example, we have a computer that can generate 16-bit addresses, from 0 up to 64K. These are the virtual addresses.
- This computer, however, has only 32 KB of physical memory, so although 64-KB programs can be written, they cannot be loaded into memory in their entirety and run.
- With 64 KB of virtual address space and 32 KB of physical memory, we get 16(64/4KB, since page frame size is typically of 4KB) virtual pages and 8 page frames. Transfers between RAM and disk are always in units of a page.

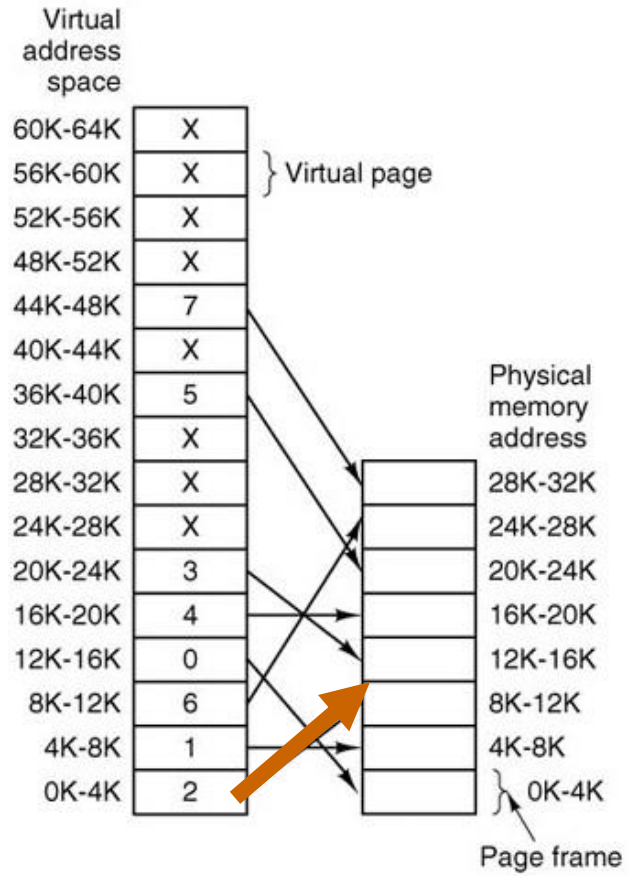
Paging



- When the program tries to access address 0 (see the virtual address space), for example, using the instruction **MOV REG,0**
- virtual address 0 is sent to the MMU.
- The MMU sees in page table that this virtual address falls in page 0 (0 to 4095), which according to its mapping is page frame 2 (8192 to 12287 in the physical address).

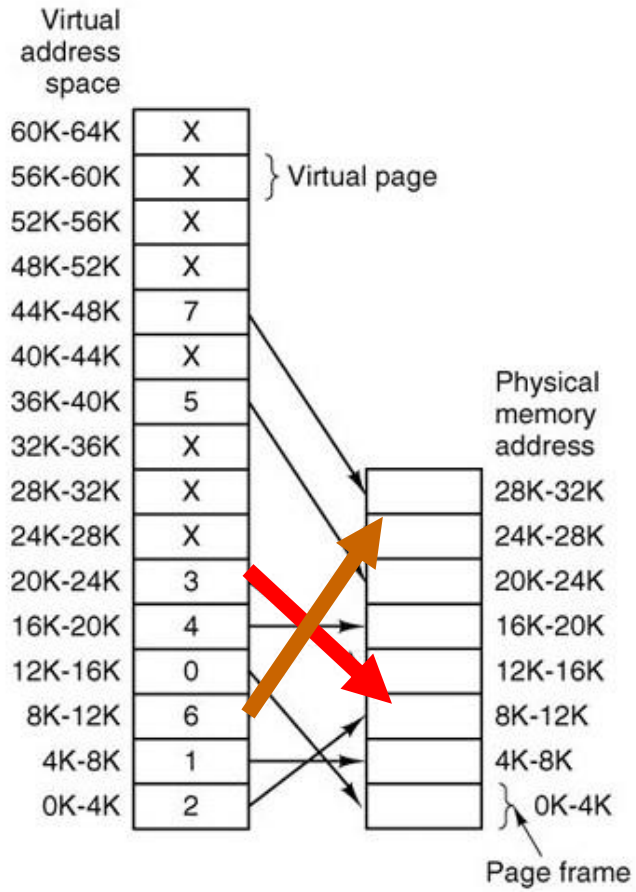
Fig:The relation between virtual addresses and physical memory addresses.

Paging



- It thus transforms the address to 8192 and outputs address 8192 onto the bus.
- The memory knows nothing at all about the MMU and just sees a request for reading or writing address 8192, which it honors.
- Thus, the MMU has effectively mapped all virtual addresses between 0 and 4095 onto physical addresses 8192 to 12287.

Paging



Similarly, an instruction **MOV REG, 8192** is effectively transformed into **MOV REG,24576.**

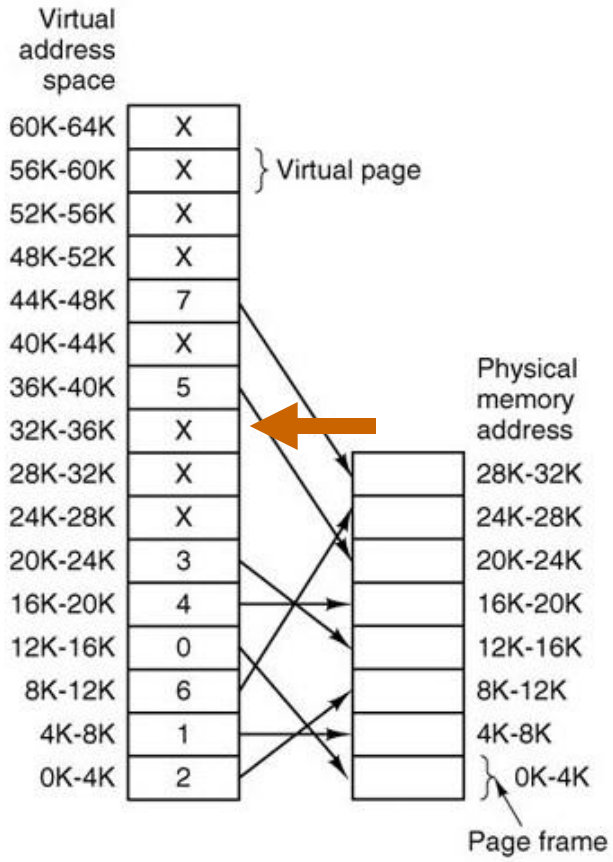
Because virtual address 8192 is in virtual page 2 and this page is mapped onto physical page frame 6 (physical addresses 24576 to 28671).

- As a third example, virtual address 20500 is 20 bytes from the start of virtual page 5 (virtual addresses 20480 to 24575) and maps onto physical address 12288 + 20 = 12308.

PAGE Fault:

- A **page fault** is a trap to the software raised by the hardware when a program accesses a page that is mapped in the virtual address space, but not loaded in physical memory.
- In the typical case the operating system tries to handle the page fault by making the required page accessible at a location in physical memory or kills the program in the case of an illegal access.
- The hardware that detects a page fault is the memory management unit in a processor.
- The exception handling software that handles the page fault is generally part of the operating system.
- What happens if the program tries to use an unmapped page, for example, by using the instruction **MOV REG,32780**

PAGE Fault:



- Which is within virtual page 8 (starting at 32768)
- The MMU notices that the page is unmapped (indicated by a cross in the figure) and causes the CPU to trap to the operating system. This trap is called a **page fault**.
- The operating system picks a little-used page frame and writes its contents back to the disk.
- It then fetches the page just referenced into the page frame just freed, changes the map, and restarts the trapped instruction.

PAGE Fault:

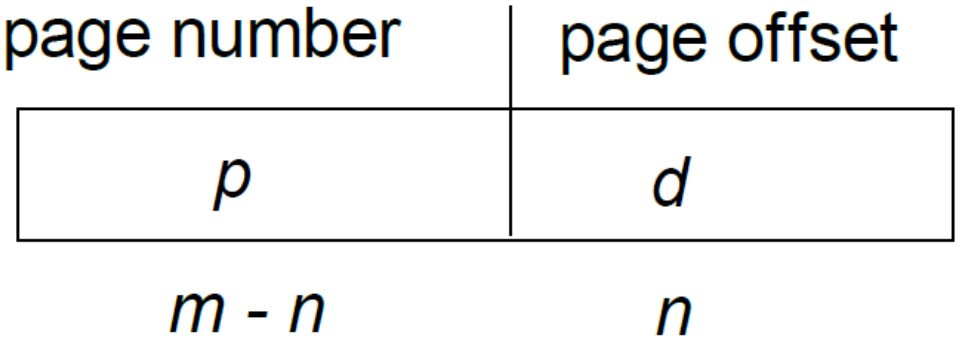
- In computer storage technology, a **page** is a fixed-length block of memory that is used as a unit of transfer between physical memory and external storage like a disk,
- And a **page fault** is an interrupt (or exception) to the software raised by the hardware, when a program accesses a page that is mapped in address space, but not loaded in physical memory.
- An interrupt that occurs when a program requests data that is not currently in real memory.
- The interrupt triggers the operating system to fetch the data from a virtual memory and load it into RAM.
- An invalid page fault or page fault error occurs when the operating system cannot find the data in virtual memory.
- This usually happens when the virtual memory area, or the table that maps virtual addresses to real addresses, becomes corrupt.

Paging Hardware:

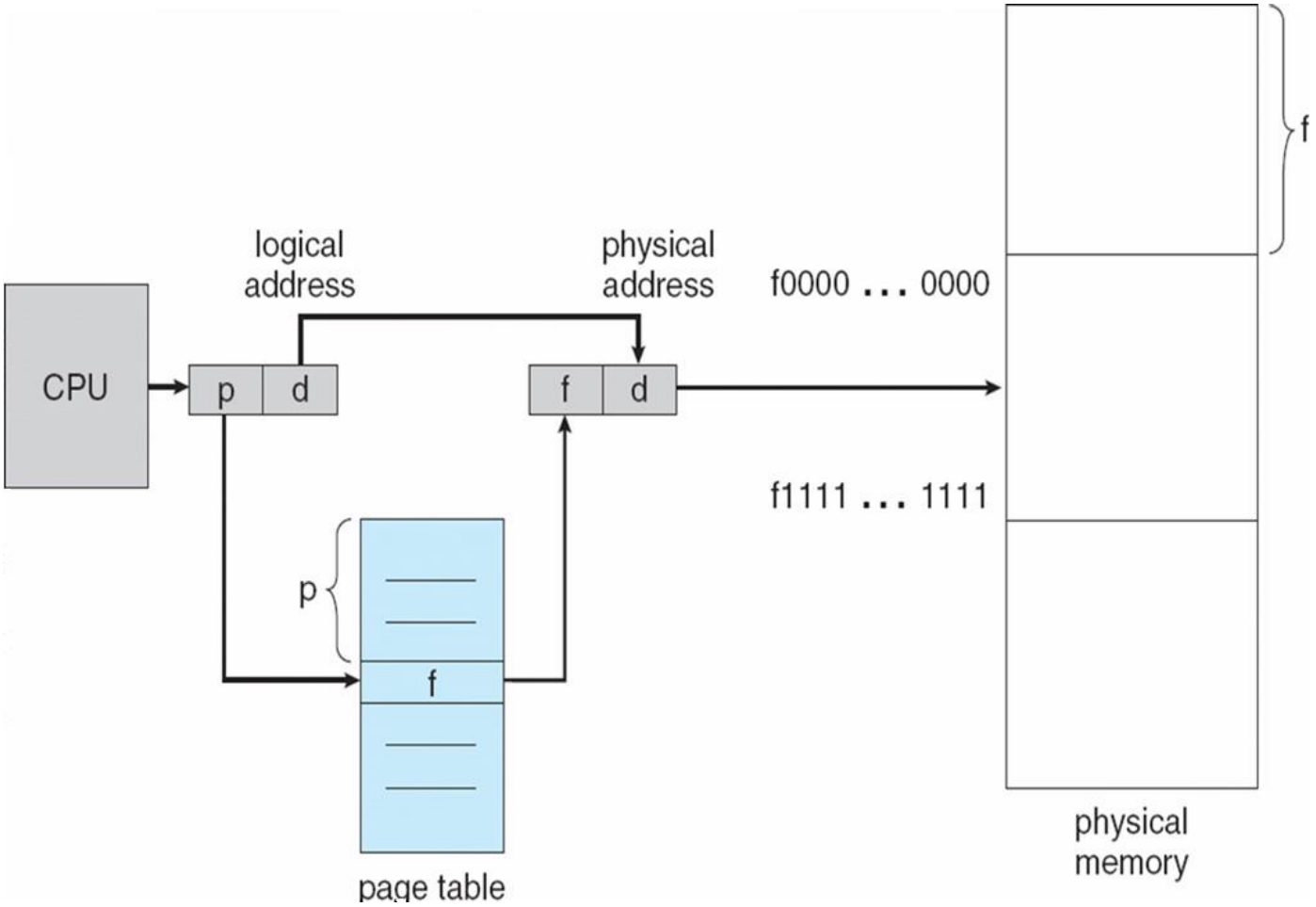
- The hardware support for the paging is as shown in fig below.
- Every address generated by the CPU is divided into two parts:

  a **page number(p)** and a **page offset (d)**

- **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.



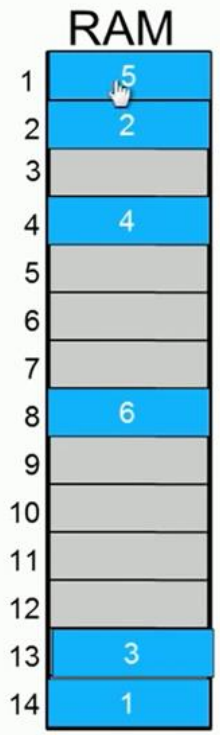
Paging Hardware:

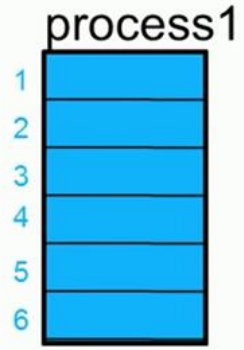


Page Table

- For each process, page table stores the number of frame, allocated for each page.
- The purpose of the page table is to map virtual pages into page frames by the MMU.
- Mathematically the page table is a function with the virtual page number as argument and the physical frame number as result.
- Using the result of this function the virtual page field in a virtual address can be replaced by a page frame field, thus forming a physical memory address.

Page Table





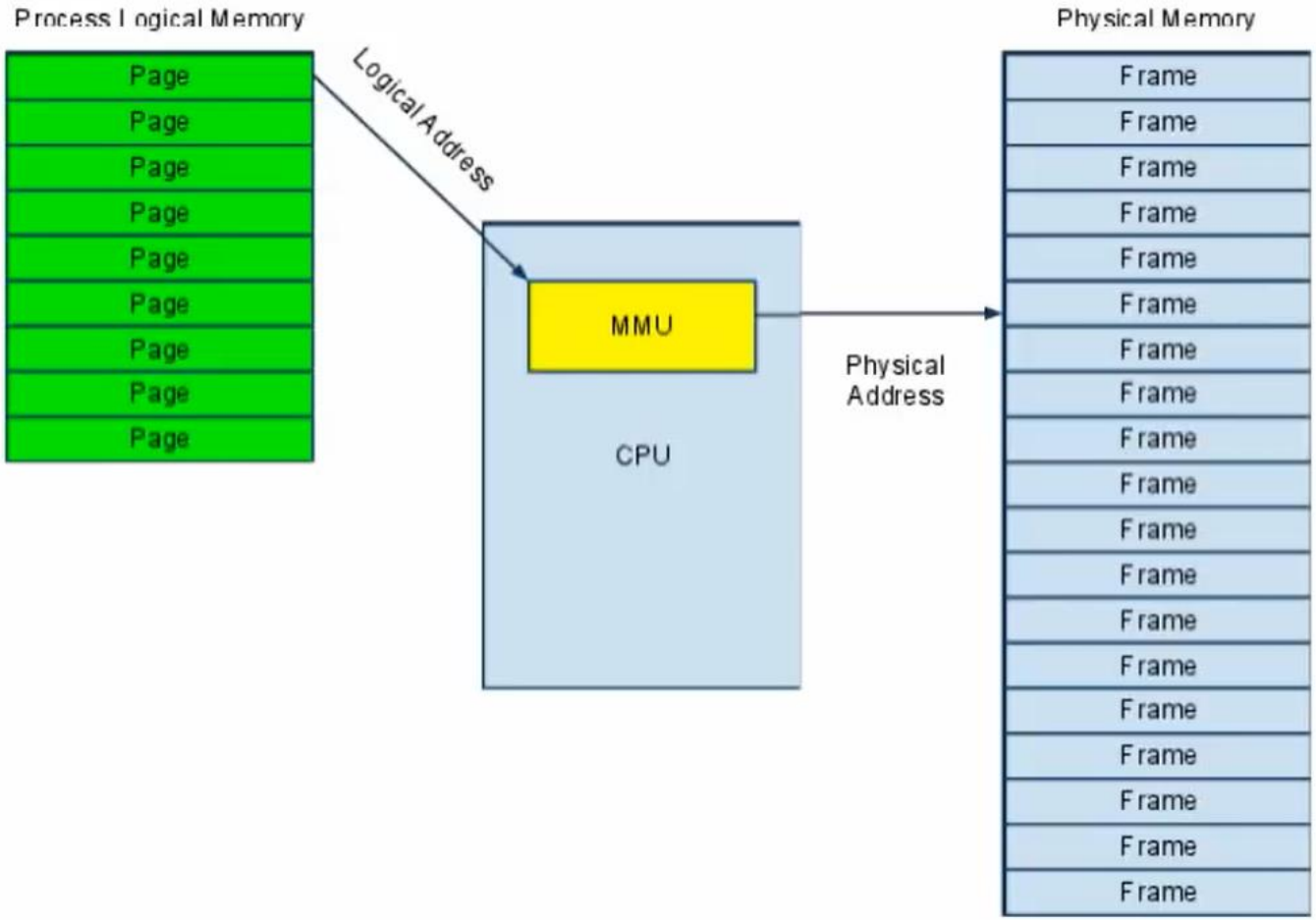
process page table

block	page frame
1	14
2	2
3	13
4	4
5	1
6	8

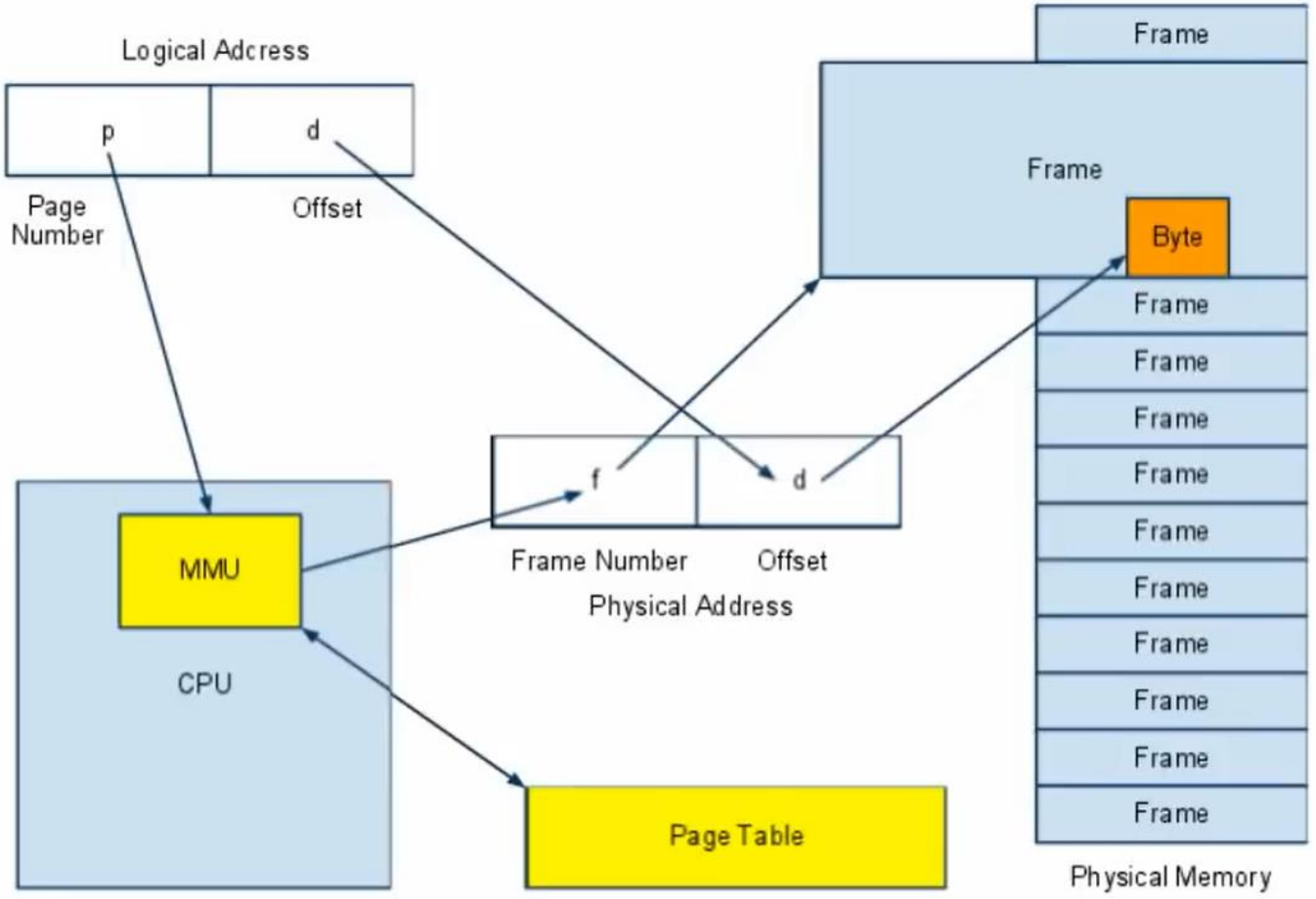
Because of the page table, blocks need not be in contiguous page frames

Every time a memory location is accessed, the processor looks into the page table to identify the corresponding page frame number.

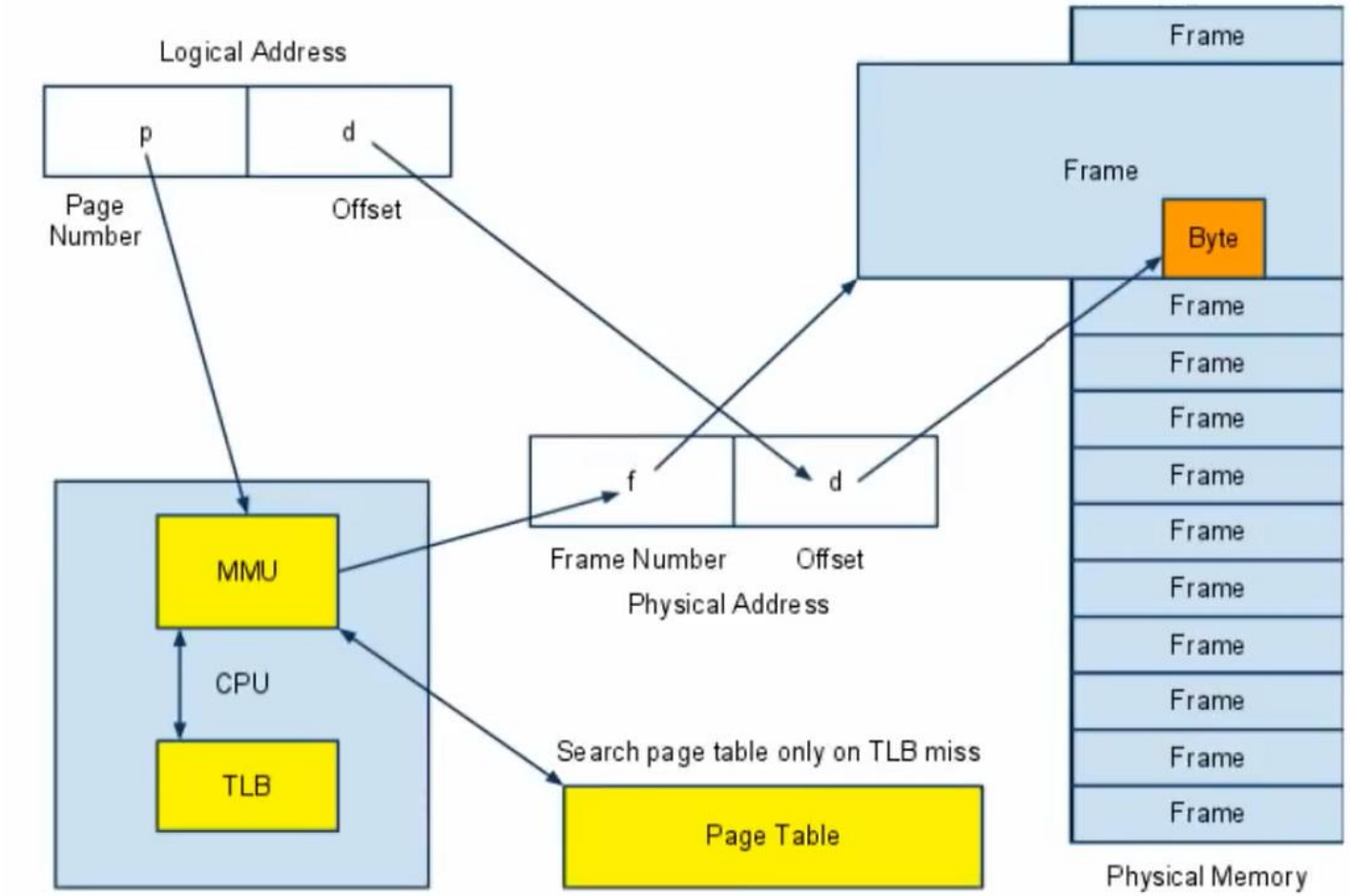
Page Translation



MMU Address Translation



TLB-Assisted Translation



TLB-Assisted Translation

- The problem with earlier MMU translation would have caused to access the memory twice(one for looking of the page table into the kernel space region in the physical memory and again in the memory itself after getting the appropriate mapping of the page into the frame). Which makes slower execution.
- To solve the above problem a different hardware TLB(translation lookaside buffer) cache is embedded into the CPU. This time MMU checks first into the TLB if it is found then TLB Hit occurs and no need to access the memory for getting the page table information.
- TLB is also called **associative memory** sometimes.

Page Replacement Algorithm

- When a page fault occurs, the operating system has to choose a page to remove from memory to make room for the page that has to be brought in.
- The page replacement is done by swapping the required pages from backup storage to main memory and vice-versa.
- If the page to be removed has been modified while in memory, it must be rewritten to the disk to bring the disk copy up to date.
- If, however, the page has not been changed (e.g., it contains program text), the disk copy is already up to date, so no rewrite is needed.
- The page to be read in just overwrites the page being evicted.

Page Replacement Algorithm

- Each operating system uses different page replacement algorithms.
- To select the particular algorithm, the algorithm with lowest page fault rate is considered.

1. Optimal page replacement algorithm
2. Not recently used page replacement
3. First-In, First-Out page replacement
4. Second chance page replacement
5. Clock page replacement
6. Least recently used page replacement
7. Most frequently used etc.

1.Optimal Page Replacement

- The algorithm has lowest page fault rate of all algorithm.
- This algorithm state that: **Replace the page which will not be used for longest period of time** i.e future knowledge of reference string is required.
- Often called Balady's Min Basic idea: Replace the page that will not be referenced for the longest time.
- Impossible to implement.

Q). If we consider reference string 7,0,1,2,0,3,0,4,2,3,0,3,2,1,2,0,1,7,0,1 and number of frames allocated =3. Using optimal page replcement algorithm to find fault number .

7	0	1	2	0	3	0	4	2	3	0	3	2	1	2	0	1	7	0	1
7	7	7	2	2	2	2	2	2	2	2	2	2	2	2	2	2	7	7	7
	0	0	0	0	0	0	4	4	4	0	0	0	0	0	0	0	0	0	0
		1	1	1	3	3	3	3	3	3	3	3	1	1	1	1	1	1	1
*	*	*	*	+	*	+	*	+	+	*	+	+	*	+	+	+	*	+	+

Here : * = Fault

\+ = hit

Total number of page fault = 9

page fault ratio = 9/20 = 0.45

2. First In First out Page replacement algorithm

- The oldest page in the physical memory is the one selected for replacement.
- Very simple to implement.
- Keep a list on a page fault, the page at the head is removed and the new page added to the tail of the list.
- When a page must be replaced, the oldest page is chosen
- To determine the number of page faults for a particular reference string and page replacement algorithm, we also need to know the number of page frames available.
- As the number of frames available increase, the number of page faults will decrease.

Issues:

Poor replacement policy

FIFO doesn't consider the page usage.

Q). If we consider reference string 7,0,1,2,0,3,0,4,2,3,0,3,2,1,2,0,1,7,0,1 and number of frames allocated =3. Using FIFO page replcement algorithm to find fault number .

7	0	1	2	0	3	0	4	2	3	0	3	2	1	2	0	1	7	0	1
7	7	7	2	2	2	2	4	4	4	0	0	0	0	0	0	0	7	7	7
	0	0	0	0	3	3	3	2	2	2	2	2	1	1	1	1	1	0	0
		1	1	1	1	0	0	0	3	3	3	3	3	2	2	2	2	2	1
*	*	*	*	+	*	*	*	*	*	*	+	+	*	*	+	+	*	*	*

Here : * = Fault

\+ = hit

Total number of page fault = 15

page fault ratio = 15/20

3. LRU Algorithm

- LRU replacement associates with each page the time of that page's last use.
- When a page must be replaced, LRU chooses that page that has not been used for the longest period of time.

Counter implementation

- Every page entry has a counter; every time page is referenced through this entry, copy the clock into the counter.
- When a page needs to be changed, look at the counters to determine which are to change.

Q). If we consider reference string 7,0,1,2,0,3,0,4,2,3,0,3,2,1,2,0,1,7,0,1 and number of frames allocated =3. Using LRU page replcement algorithm to find fault number .

7	0	1	2	0	3	0	4	2	3	0	3	2	1	2	0	1	7	0	1
7	7	7	2	2	2	2	4	4	4	0	0	0	1	1	1	1	1	1	1
	0	0	0	0	0	0	0	0	3	3	3	3	3	3	0	0	0	0	0
		1	1	1	3	3	3	2	2	2	2	2	2	2	2	2	7	7	7
*	*	*	*	+	*	+	*	*	*	*	+	+	*	+	*	+	*	+	+

Here : * = Fault

\+ = hit

Total number of page fault = 12

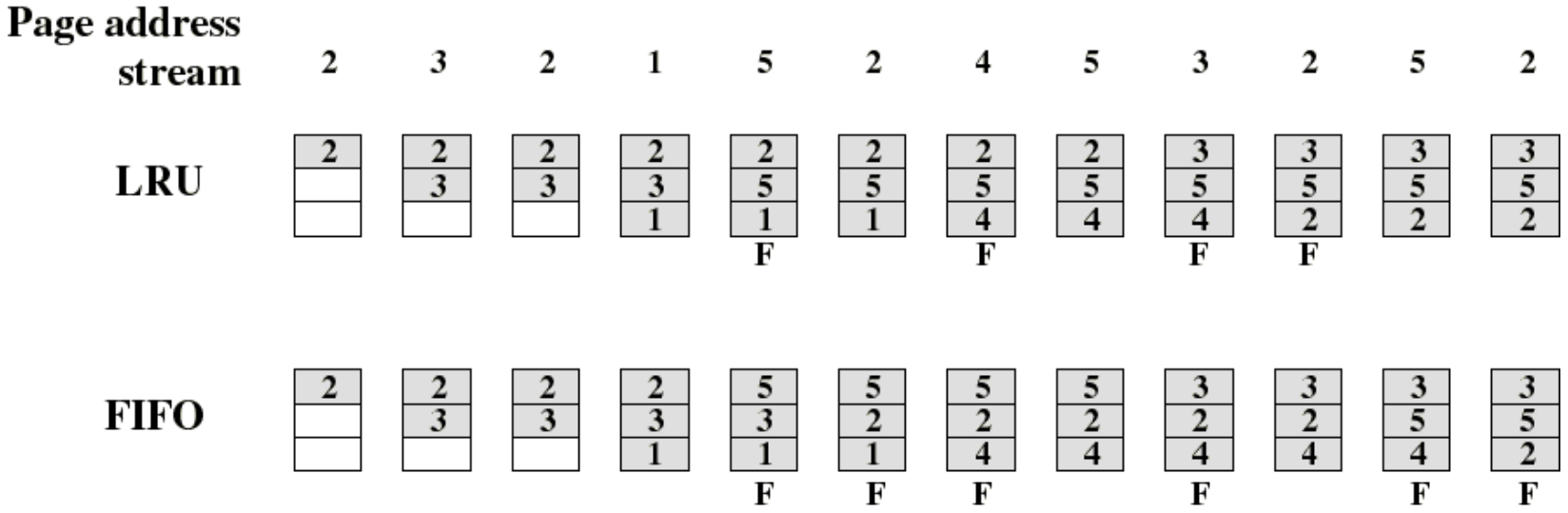
page fault ratio = 12/20

Comparison of OPT with LRU

Page address stream	2	3	2	1	5	2	4	5	3	2	5	2
OPT	2	2	2	2	2	2	4	4	4	2	2	2
		3	3	3	3	3	3	3	3	3	3	3
				1	5	5	5	5	5	5	5	5
					F		F			F		
LRU	2	2	2	2	2	2	2	2	3	3	3	3
		3	3	3	5	5	5	5	5	5	5	5
				1	1	1	4	4	4	2	2	2
					F		F		F	F		

A process of 5 pages with an OS that fixes the resident set size to 3

Comparison of FIFO with LRU



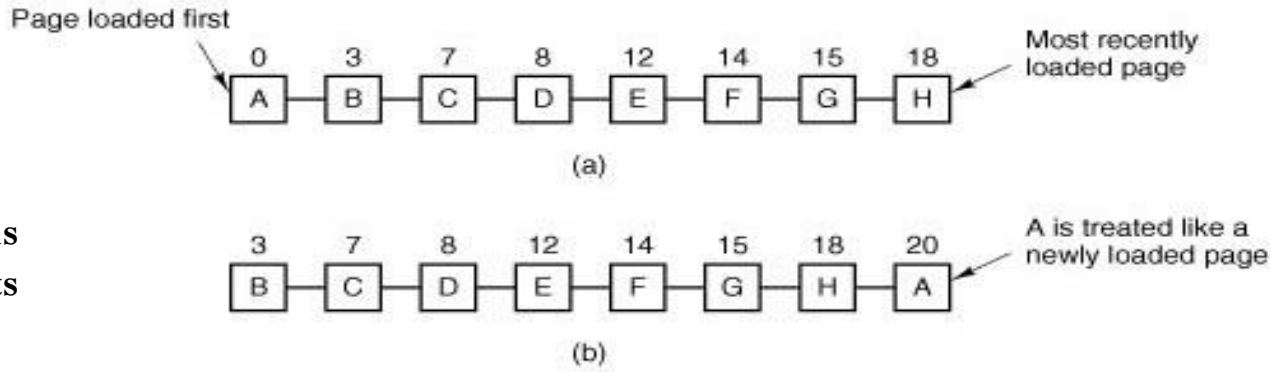
LRU recognizes that pages 2 and 5 are referenced more frequently than others but FIFO does not.

4. The Second Chance Page Replacement Algorithm:

- A simple modification to FIFO that avoids the problem of throwing out heavily used page.
- It inspects the reference (R) bit If it is 0, the page is both old and unused, so it is replaced immediately.
- If the R bit is 1, the bit is cleared, the page is put onto the end of the list of pages, and its load time is updated as though it had just arrived in memory. Then the search continues.

If r=0 then page A is removed since A is oldest page

If r=1 then page A is not removed it gets second chance



Q). If we consider reference string 2,3,2,1,5,2,4,5,3,2,5,2 and number of frames allocated =3. Using Second chance page replcement algorithm to find fault number

2	3	2	1	5	2	4	5	3	2	5	2
2(0)	2(0)	2(1)	2(1)	2(0)	2(1)	2(0)	2(0)	3(0)	3(0)	3(0)	3(0)
	3(0)	3(0)	3(0)	5(0)	5(0)	5(0)	5(1)	5(0)	5(0)	5(1)	5(1)
			1(0)	1(0)	1(0)	4(0)	4(0)	4(0)	2(0)	2(0)	2(1)
*	*	+	*	*	+	*	+	*	*	+	+

Here : * = Fault

\+ = hit

Total number of page fault = 7

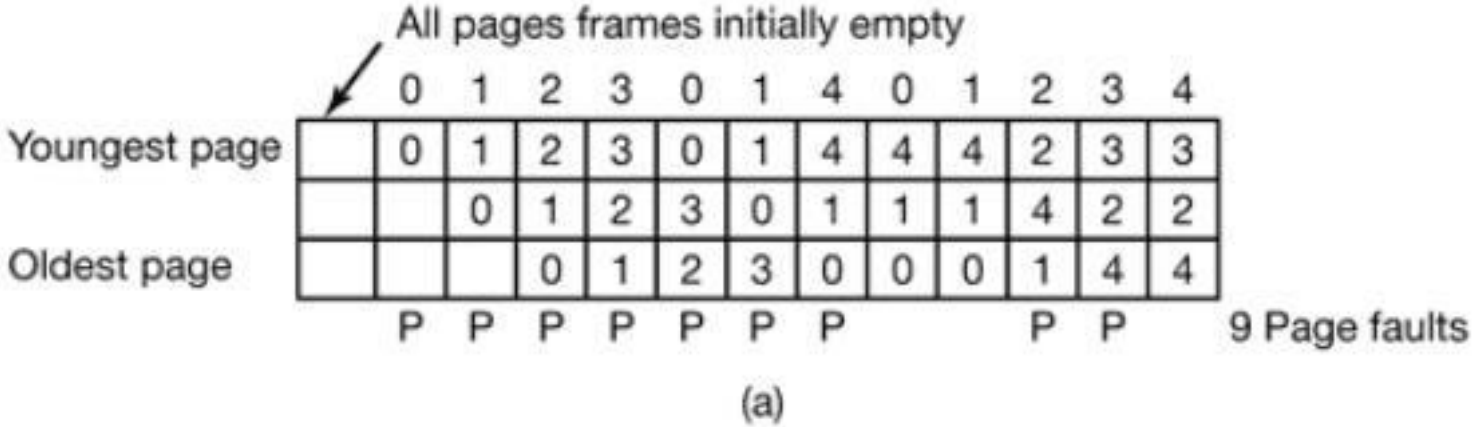
page fault ratio = 7/12

5.Belady's Anomaly

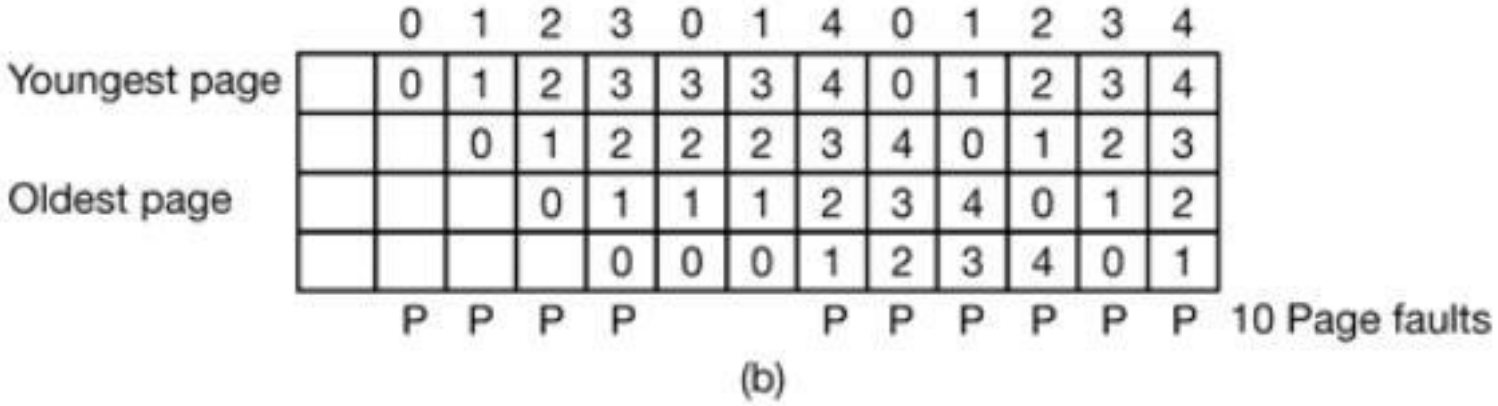
- Intuitively, it might seem that the more page frames the memory has, the fewer page faults a program will get.
- Surprisingly enough, this is not always the case.
- Belady discovered a counter example, in which FIFO caused more page faults with four page frames than with three.
- This strange situation has become known as **Belady's anomaly**.
- It is illustrated in Fig. for a program with five virtual pages, numbered from 0 to 4. The pages are referenced in the order

Reference String: 0 1 2 3 0 1 4 0 1 2 3 4

5.Belady's Anomaly



(a)



(b)

5. Belady's Anomaly uCSIT-4

Q). If we consider reference string 7,0,1,2,0,3,0,4,2,3,0,3,2,1,2,0,1,7,0,1 and number of frames allocated =3. Using Belady's anomaly FIFO page replcement algorithm to find fault number .

7	0	1	2	0	3	0	4	2	3	0	3	2	1	2	0	1	7	0	1
7	7	7	2	2	2	2	4	4	4	0	0	0	0	0	0	0	7	7	7
	0	0	0	0	3	3	3	2	2	2	2	2	1	1	1	1	1	0	0
		1	1	1	1	0	0	0	3	3	3	3	3	2	2	2	2	2	1
*	*	*	*	+	*	*	*	*	*	*	+	+	*	*	+	+	*	*	*

Here : * = Fault

\+ = hit

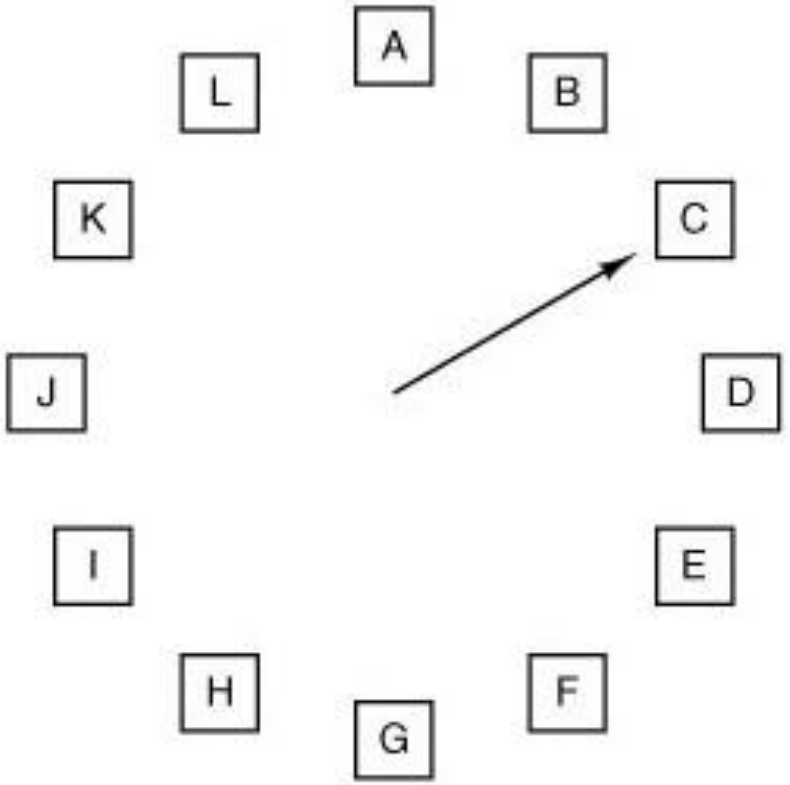
Total number of page fault = 15

page fault ratio = 15/20

6.The Clock Page Replacement Algorithm

- Keep all the page frames on a circular list in the form of a clock, as shown in Fig below.
- A hand points to the oldest page. When a page fault occurs, the page being pointed to by the hand is inspected.
- If its R bit is 0, the page is evicted, the new page is inserted into the clock in its place, and the hand is advanced one position.
- If R is 1, it is cleared and the hand is advanced to the next page.
- This process is repeated until a page is found with R = 0

6.The Clock Page Replacement Algorithm



When a page fault occurs, the page the hand is pointing to is inspected. The action taken depends on the R bit:

R = 0: Evict the page

R = 1: Clear R and advance hand

6.The Clock Page Replacement Algorithm

Algorithm

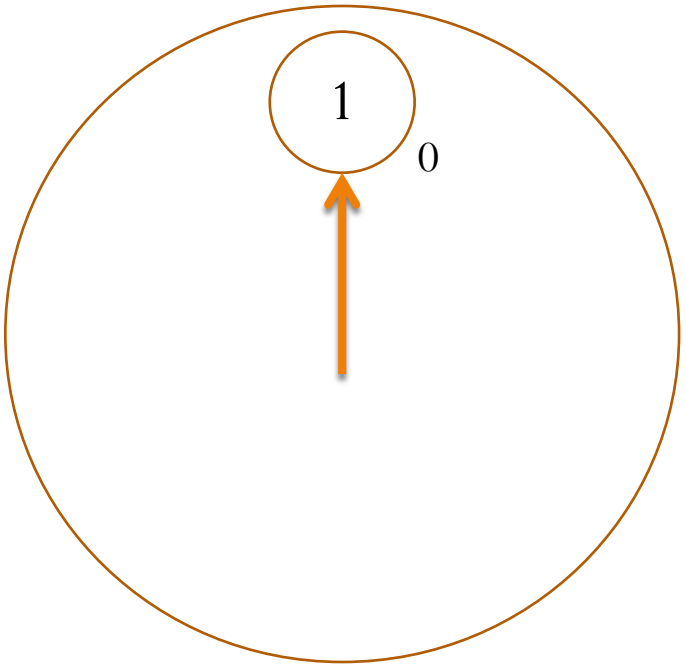
1. Begin
2. Read new page, say P
3. If p is available in clock (circular linked list)
   - 3.1. Page hit occurs, then
   - 3.2. Turn reference bit to 1 and do nothing else
4. Else
   - 4.1. Page miss occurs, then
   - 4.2. If its R bit is 0, the page is evicted, the new page is inserted into the clock in its place, and the hand is advanced one position.
   - 4.3. If R is 1, it is cleared and the hand is advanced to the next page. The process is repeated until a page is found with R=0.

Example:

Input References : {1,2,3,4,3,1,2,1,5,4}

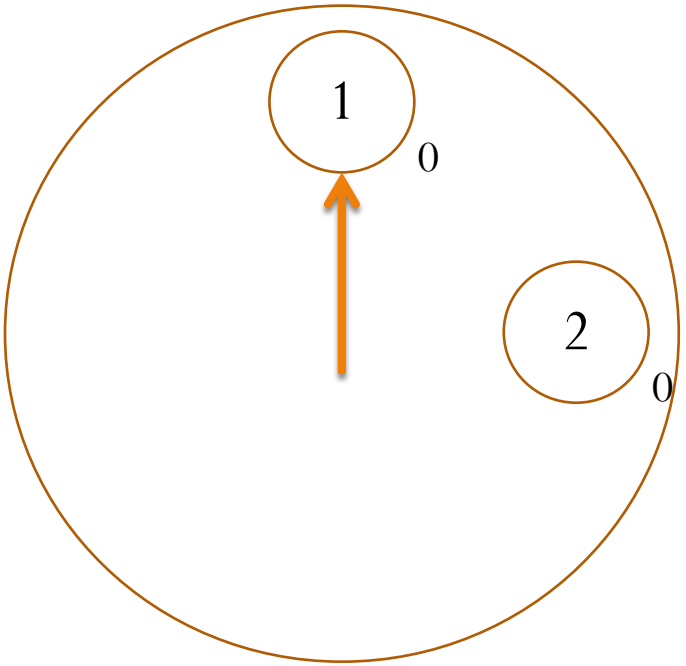
NO. of allocation : 3

1) 1



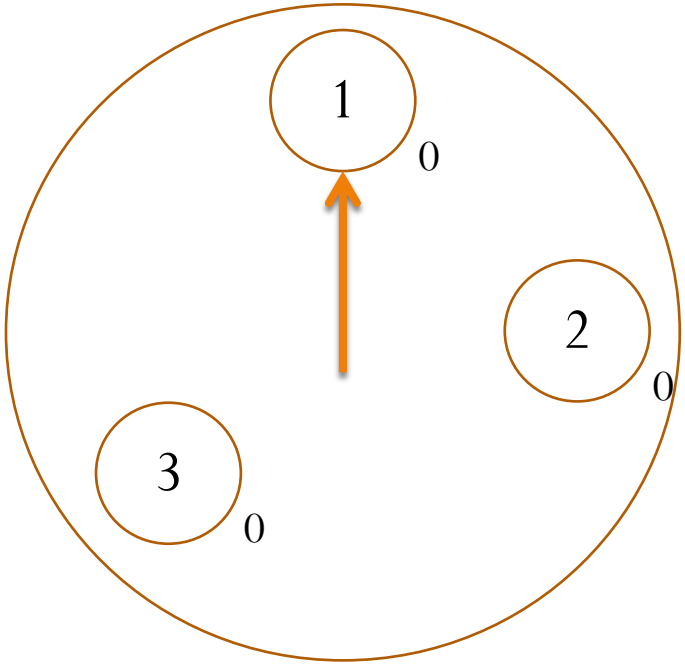
*

2) 2



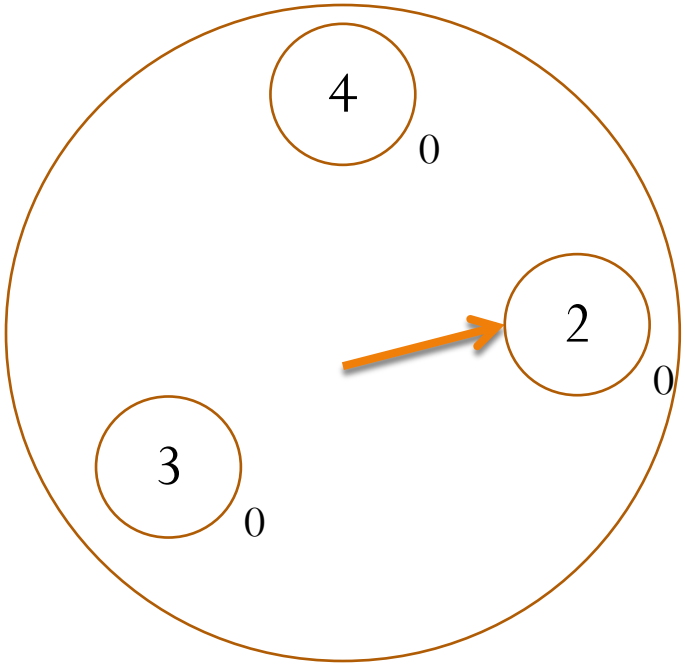
*

3) 3



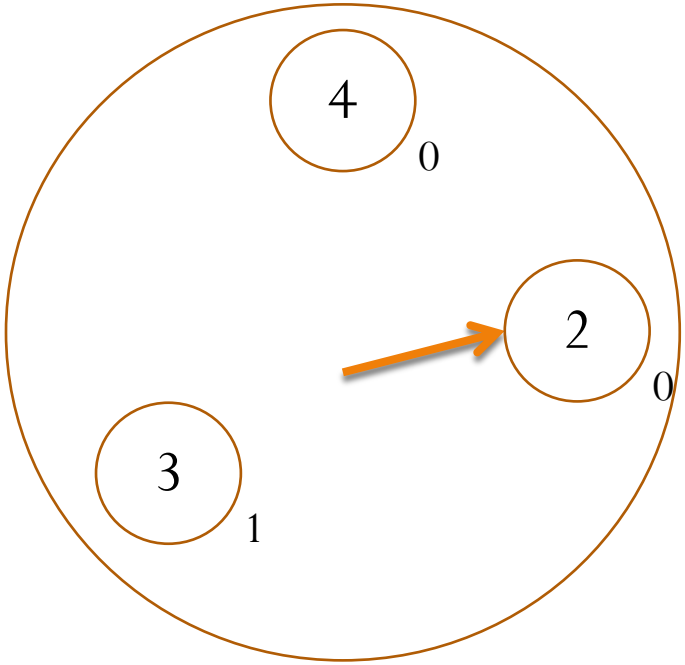
*

4) 4



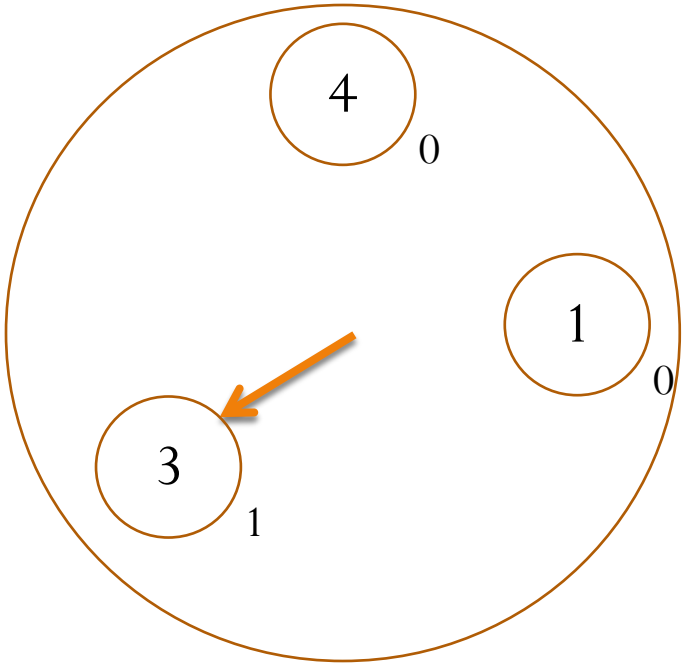
*

5) 3



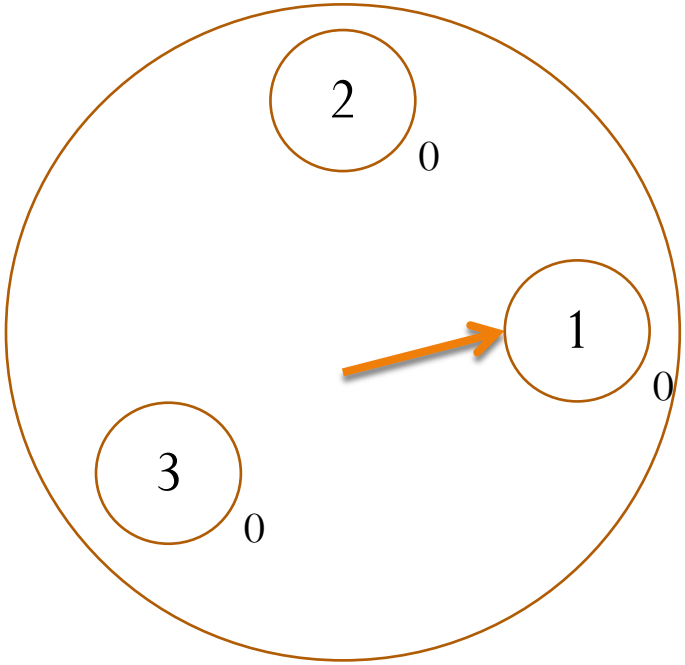
+

6) 1



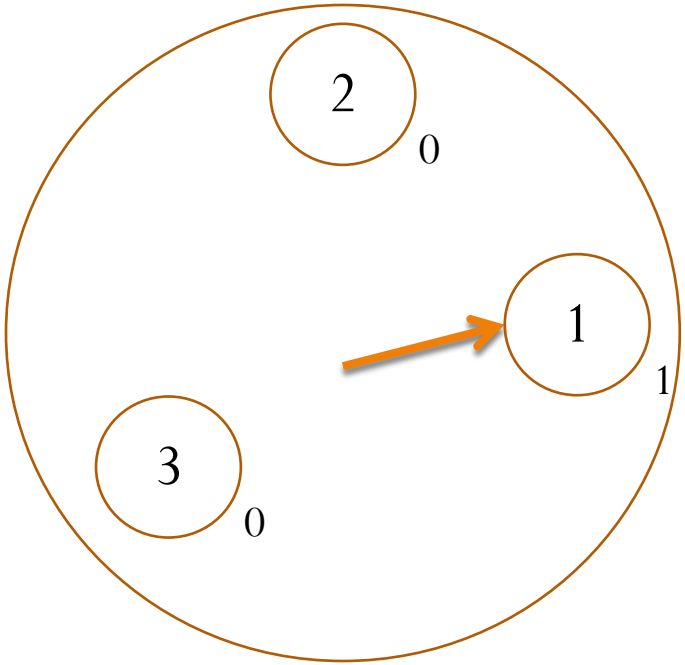
*

7) 2



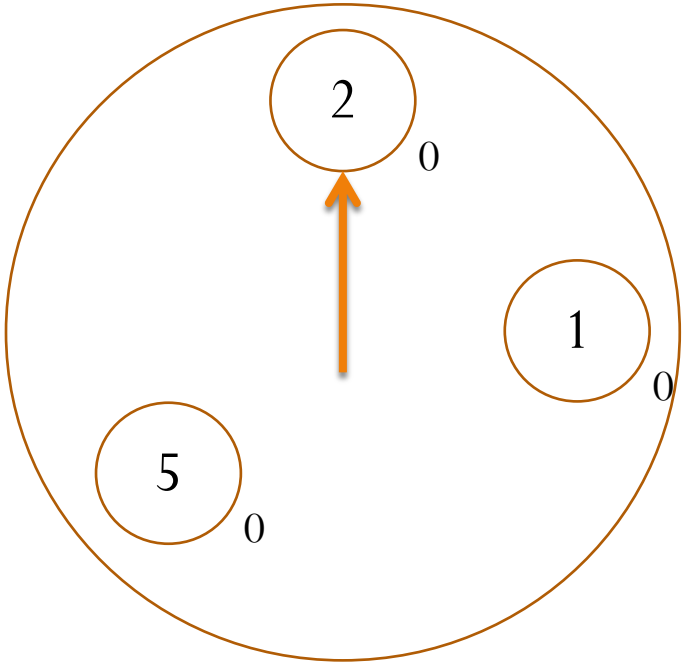
*

8) 1



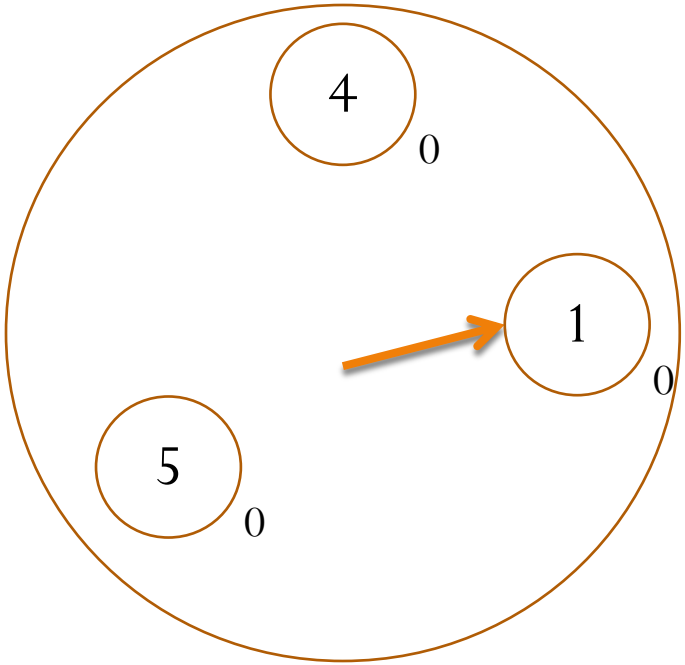
+

9) 5



*

10) 4



*

Total Page fault = 8

Segmentation

- A user program can be subdivided using segmentation, in which the program and its associated data are divided into a number of *segments*.
- Segmentation is the 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.

Programs are a collection of logical modules

```
static unsigned int blk_flush_policy(unsigned int fflags, struct request *rq
{
        unsigned int policy = 0;

        if (blk_rq_sectors(rq))
                policy |= REQ_FSEQ_DATA;

        return policy;
}

static unsigned int blk_flush_cur_seq(struct request *rq)
{
        return 1 << ffz(rq->flush.seq);
}

static void blk_flush_restore_request(struct request *rq)
{
        rq->bio = rq->biotail;
        rq->end_io = rq->flush.saved_end_io;
}

static bool blk_flush_queue_rq(struct request *rq, bool add_front)
{
        if (rq->q->mq_ops) {
                struct request_queue *q = rq->q;

                blk_mq_add_to_requeue_list(rq, add_front);
                blk_mq_kick_requeue_list(q);
                return false;
        }
}

static bool blk_flush_complete_seq(struct request *rq,
                                   struct blk_flush_queue *fq,
                                   unsigned int seq, int error)
{
        struct request_queue *q = rq->q;
        struct list_head *pending = &fq->flush_queue[fq->flush_pending_idx];
        bool queued = false, kicked;

        BUG_ON(rq->flush.seq & seq);
        rq->flush.seq |= seq;
}

static void flush_end_io(struct request *flush_rq, int error)
{
        struct request_queue *q = flush_rq->q;
        struct list_head *running;
        bool queued = false;
        struct request *rq, *n;
        unsigned long flags = 0;
```

Logical modules : such as global data, stack, heap, functions, classes, namespaces, etc.

Virtual memory does not split programs into logical modules, instead splits programs into fixed size blocks.

Programs are a collection of logical modules

```
static unsigned int blk_flush_policy(unsigned int fflags, struct request *rq
{
        unsigned int policy = 0;

        if (blk_rq_sectors(rq))
                policy |= REQ_FSEQ_DATA;

        return policy;
}
```

```
static unsigned int blk_flush_cur_seq(struct request *rq)
{
        return 1 << ffz(rq->flush.seq);
}
```

```
static void blk_flush_restore_request(struct request *rq)
{
        rq->bio = rq->biotail;
        rq->end_io = rq->flush.saved_end_io;
}
```

```
static bool blk_flush_queue_rq(struct request *rq, bool add_front)
{
        if (rq->q->mq_ops) {
                struct request_queue *q = rq->q;

                blk_mq_add_to_requeue_list(rq, add_front);
                blk_mq_kick_requeue_list(q);
                return false;
        }
}
```

```
static bool blk_flush_complete_seq(struct request *rq,
                                   struct blk_flush_queue *fq,
                                   unsigned int seq, int error)
{
        struct request_queue *q = rq->q;
        struct list_head *pending = &fq->flush_queue[fq->flush_pending_idx];
        bool queued = false, kicked;

        BUG_ON(rq->flush.seq & seq);
        rq->flush.seq |= seq;
}
```

```
static void flush_end_io(struct request *flush_rq, int error)
{
        struct request_queue *q = flush_rq->q;
        struct list_head *running;
        bool queued = false;
        struct request *rq, *n;
        unsigned long flags = 0;
```

Logical modules : such as global data, stack, heap, functions, classes, namespaces, etc.

Virtual memory does not split programs into logical modules, instead splits programs into fixed size blocks.

Segmentation can be used to split program into segments that are more logical.

The segment size could range from a few bytes to the maximum size (4GB in 32 bit Intel machines)

Segmentation

- It is not required that all segments of all programs be of the same length, although there is a maximum segment length.
- As with paging, a logical address using segmentation consists of two parts, in this case a *segment number* and an *offset*.
- Because of the use of unequal-size segments, segmentation is similar to dynamic partitioning.
- In segmentation, a program may occupy more than one partition, and these partitions need not be contiguous.
- Segmentation eliminates internal fragmentation but, like dynamic partitioning, it suffers from external fragmentation.
- However, because a process is broken up into a number of smaller pieces, the external fragmentation should be less.
- Whereas paging is invisible to the programmer, segmentation usually visible and is provided as a convenience for organizing programs and data.

Segmentation

- When a process enters the Running state, the address of its segment table is loaded into a special register used by the memory management hardware.
- A logical address consists of two parts: a segment number, *s*, and an offset into the segment, *d*.
- The segment number is used as an index into the segment table.
- The offset must be between 0 and the segment limit. If the offset is legal it is added to the segment base to produce the address in physical memory of the desired byte.

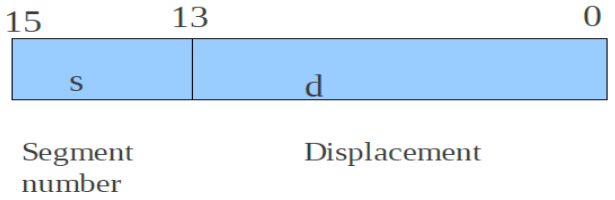
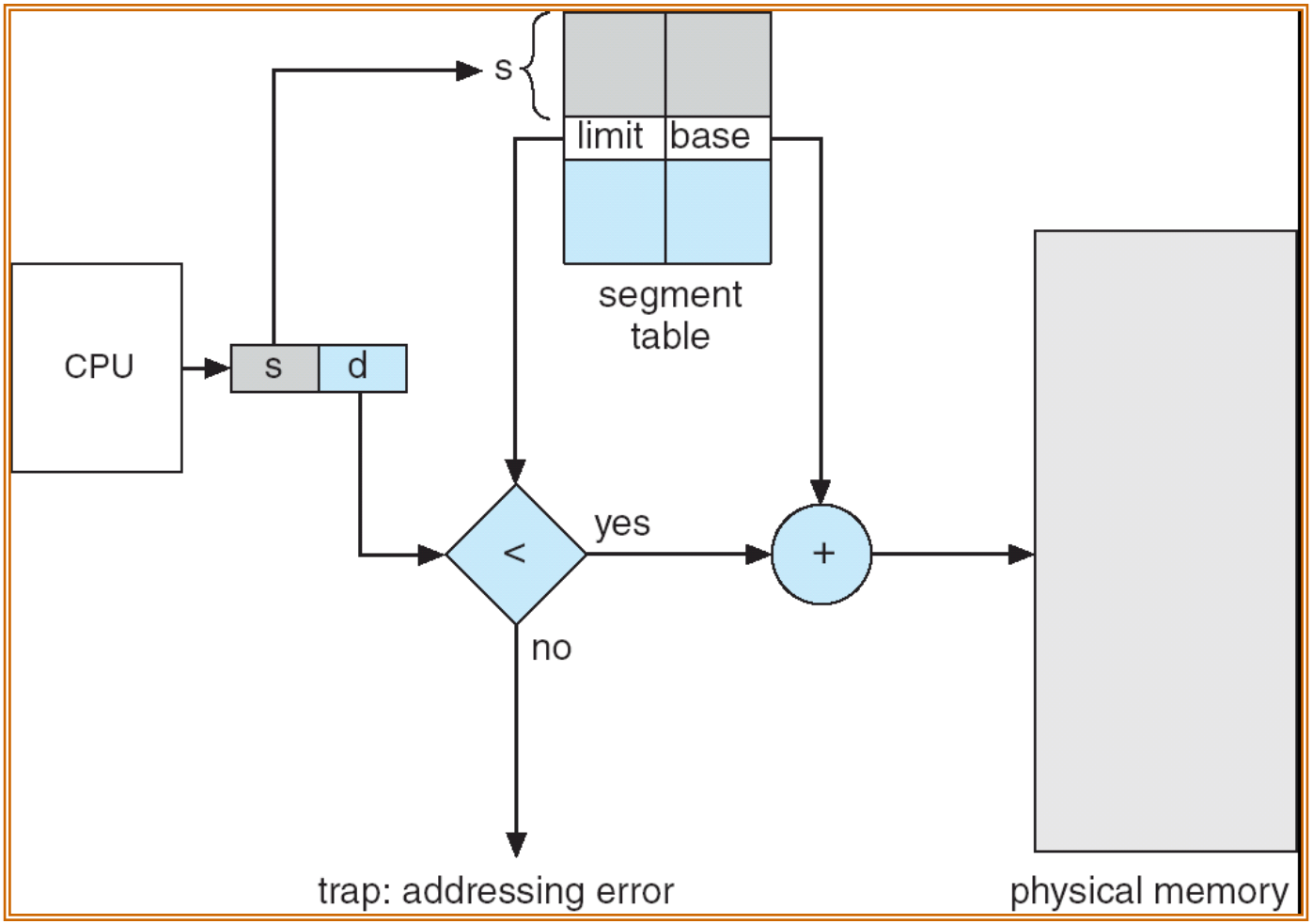


Fig:Virtual address space or logical address space (16 bit)

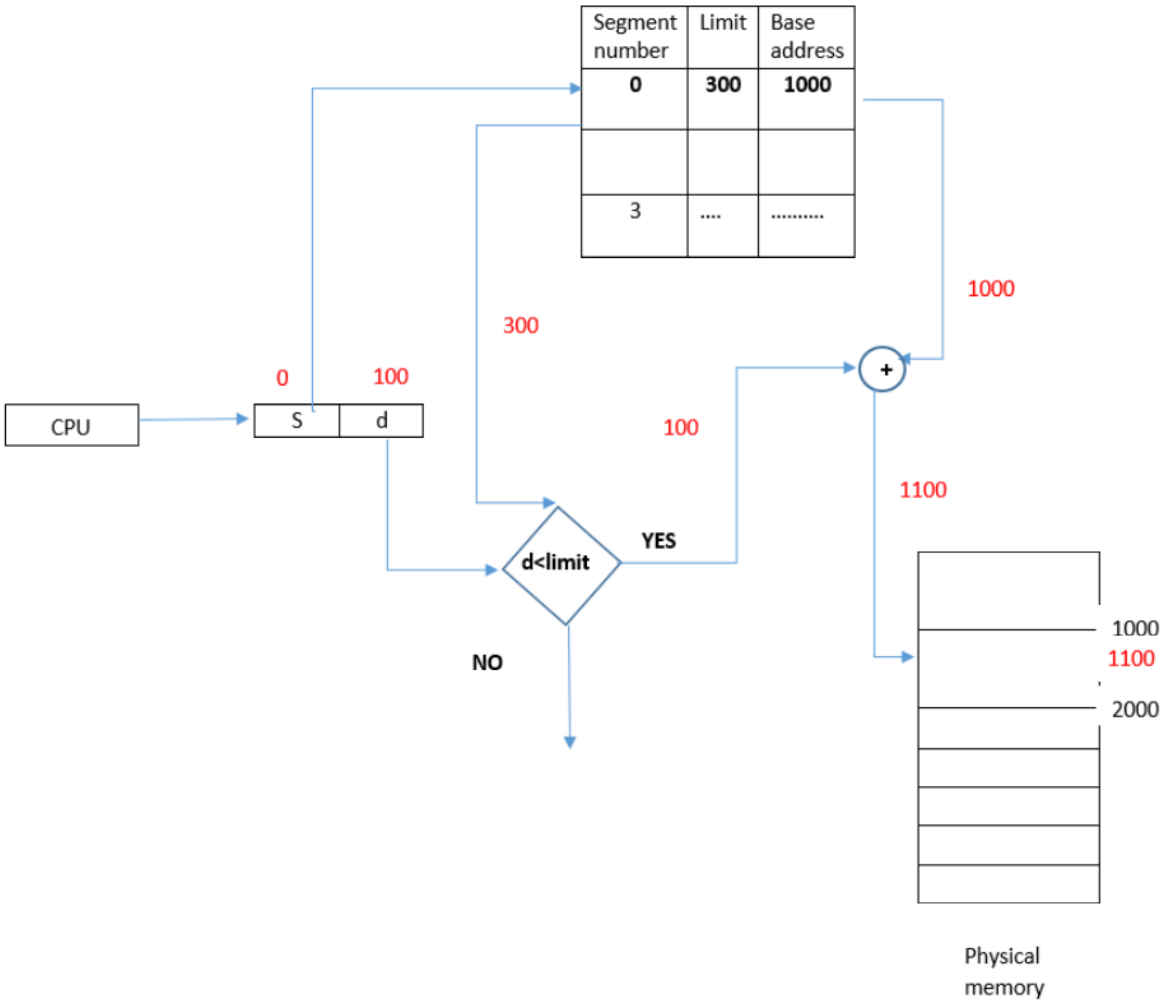
Segment table

- The segmentation hardware consists **segment table**.
- Each entry in the segment table has a segment *base* and a segment *limit*.
- The segment base contains the starting physical address where the segment resides in memory, whereas the segment limit specifies the length of the segment.
- The segment table is thus essentially an array of base-limit register pair.

Segment table



Segment table

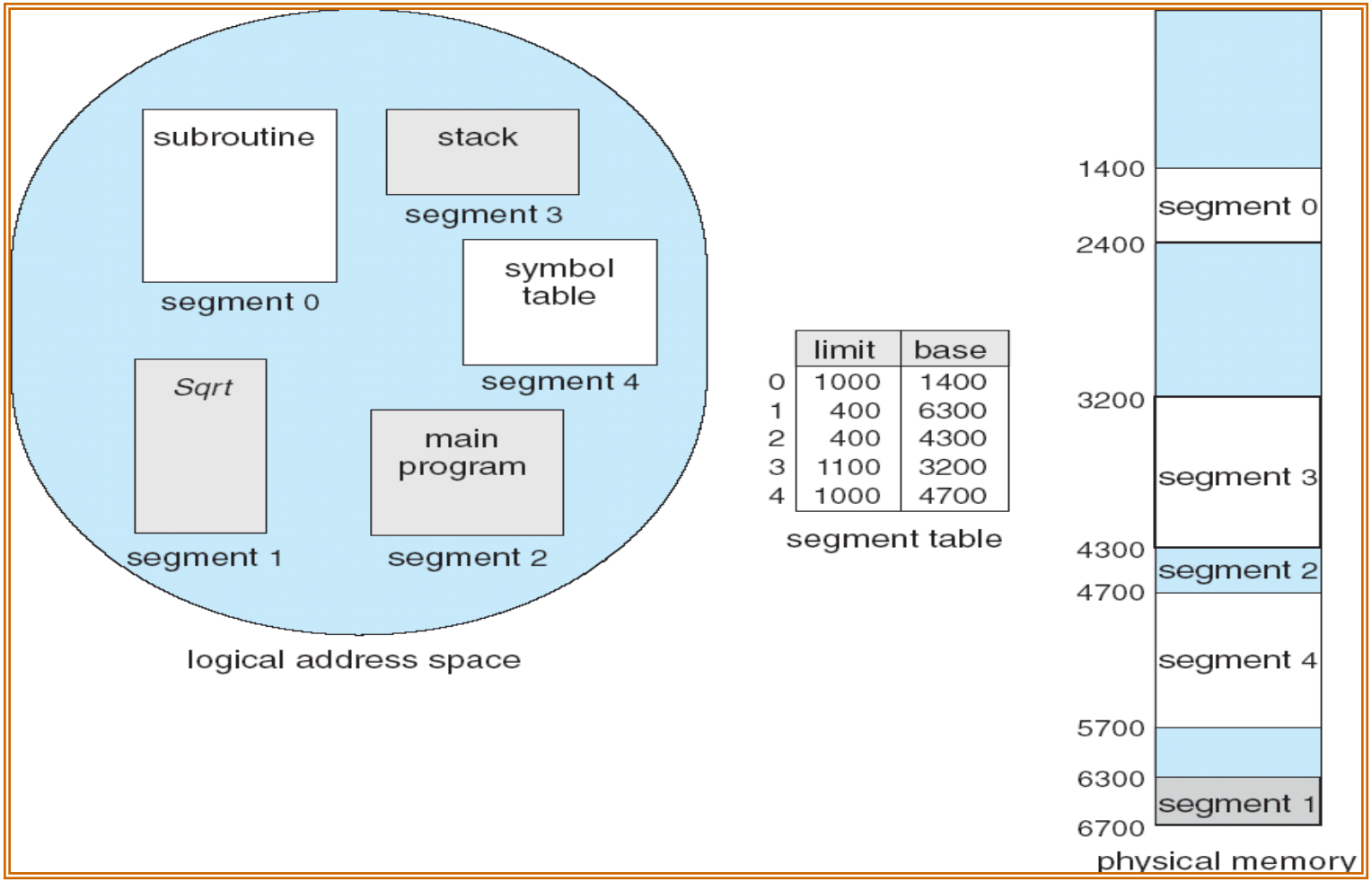


Segment table

The following steps are needed for address translation

- Extract the segment number as the leftmost n bits of the logical address.
- Use the segment number as an index into the process segment table to find the starting physical address of the segment.
- Compare the offset, expressed in the rightmost m bits, to the length of the segment.
- If the offset is greater than or equal to the length, the address is invalid.
- The desired physical address is the sum of the starting physical address of the segment plus the offset.
- Segmentation and paging can be combined to have a good result.

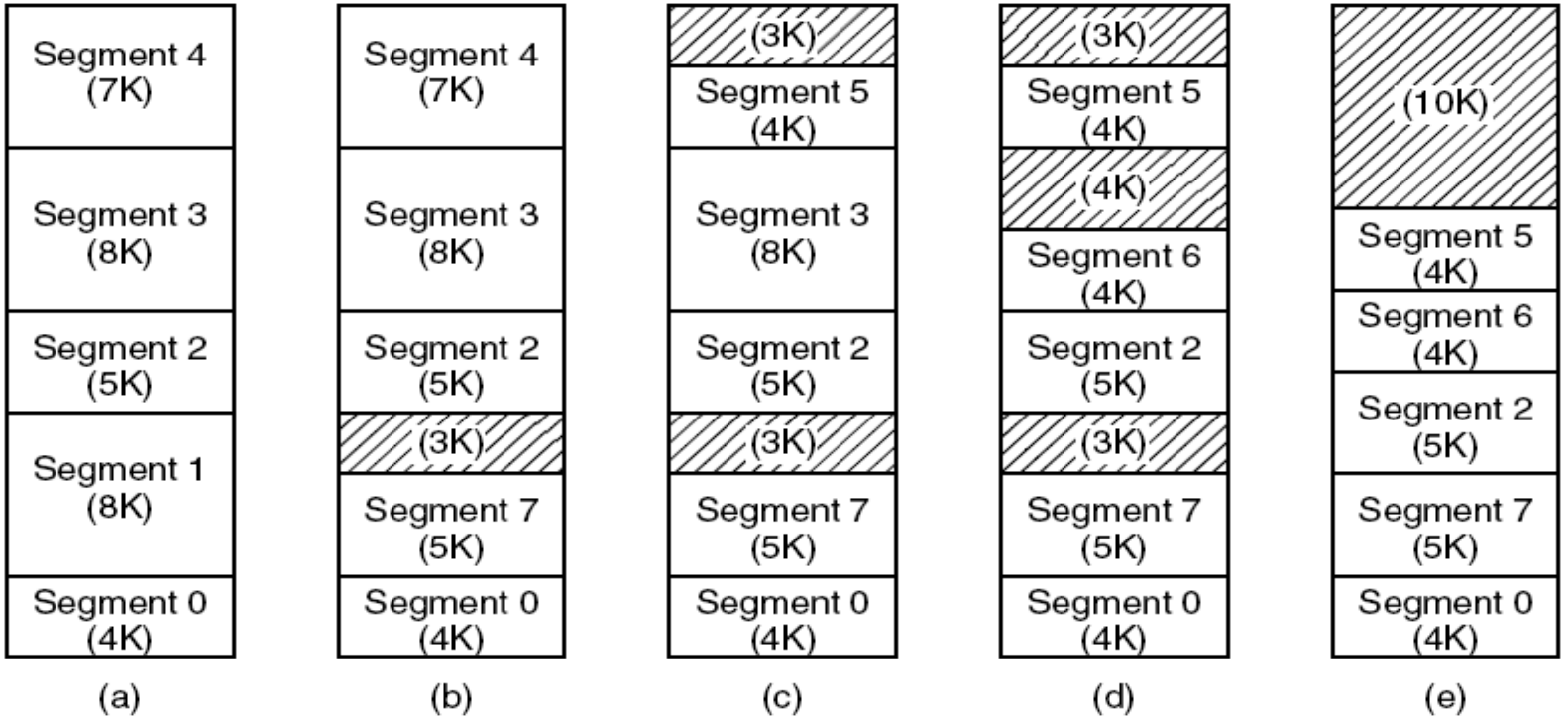
Segment table



Segment table

- Like paging, segmentation is also a memory-management scheme that permits the physical-address space of a process to be noncontiguous.
- However, in segmentation, segments are of variable length, whereas pages are all the same size.
- Segmentation may cause external fragmentation, when all blocks of free memory are too small to accommodate a segment.
- In this case, the processes may simply have to wait until more memory (or at least a larger hole) becomes available, or until compaction creates a larger hole.

Implementation of Pure Segmentation



(a)-(d) Development of checkerboarding(external fragmentation).

(e) Removal of the checkerboarding by compaction.

Paging Vs Segmentation

Consideration	Paging	Segmentation
Need the programmer be aware that this technique is being used?	No	Yes
How many linear address spaces are there?	1	Many
Can the total address space exceed the size of physical memory?	Yes	Yes
Can procedures and data be distinguished and separately protected?	No	Yes
Can tables whose size fluctuates be accommodated easily?	No	Yes
Is sharing of procedures between users facilitated?	No	Yes
Why was this technique invented?	To get a large linear address space without having to buy more physical memory	To allow programs and data to be broken up into logically independent address spaces and to aid sharing and protection

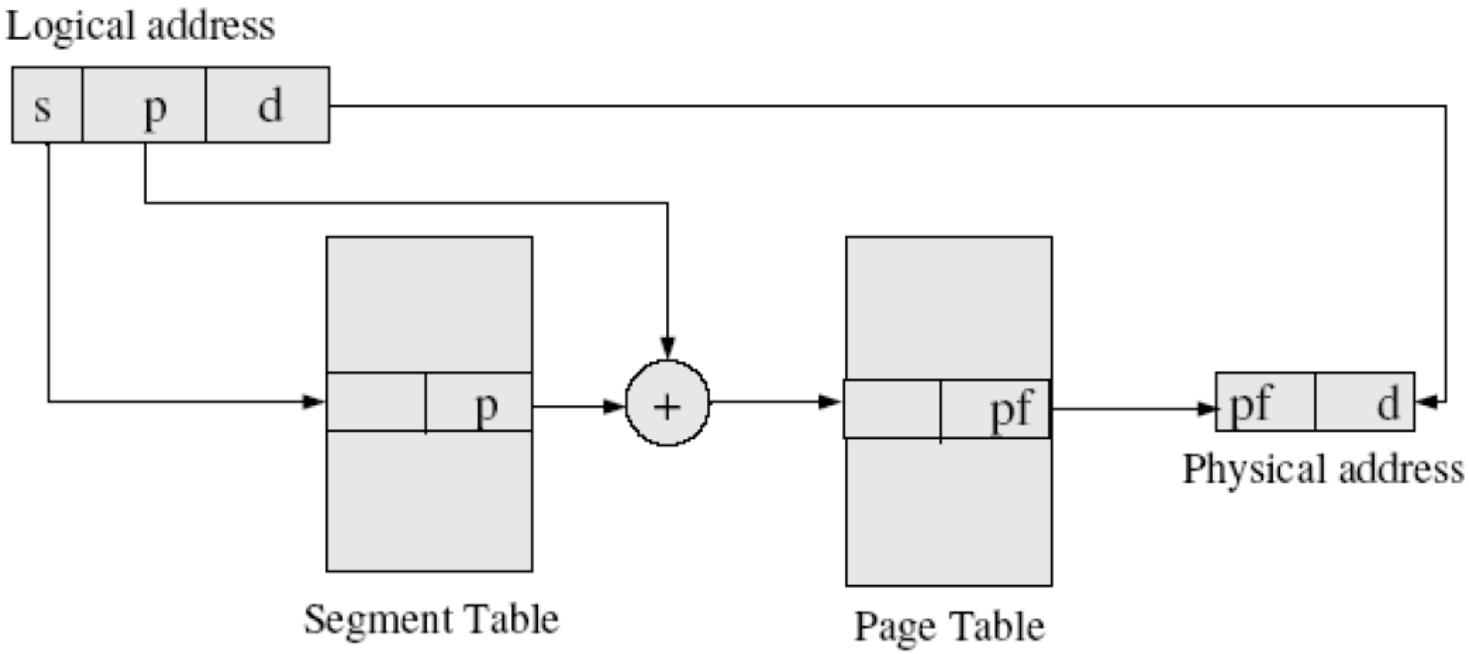
Segmentation with paging

- Both paging and segmentation have advantages and disadvantages.
- We can merge these two methods to improve on each.
- This combination is used in Intel 386 (and later) architecture
- If the segments are large, say large enough then the physical memory it may be inconvenient or even impossible to keep them in main memory in their entirely.
- This leads to the idea of paging them, so that only those pages are actually needed have to be in the physical memory.

Segmentation with paging

- Segmentation can be combined with paging to provide the facility of paging with the protection and sharing capabilities of segmentation.
- As with segmentation, the logical address specifies the segment and the offsets within the segment.
- When paging is added, the segment offset is further divided into page number and page offset.
- The segment table entry contains the address of the segment's page table.

Segmentation with paging



Examples:

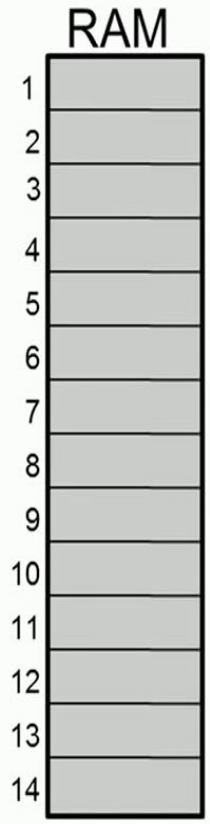
The Intel Pentium:

The Intel Pentium 80386 and later architecture uses segmentation with paging memory management.
The maximum number of segments per process is 16K, and each segment can be large as 4 GB 32-bit words. The page size is 4K fixed. It use two-level paging scheme.

Multics:

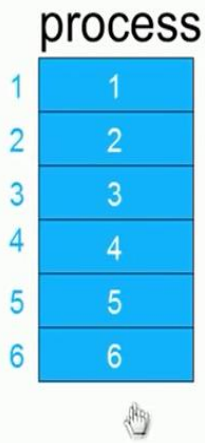
It has 256K independent segments, and each up to 64K 36-bit words. The page size is 1K words or small.

Virtual Memory



RAM is split into fixed size partitions called page frames.

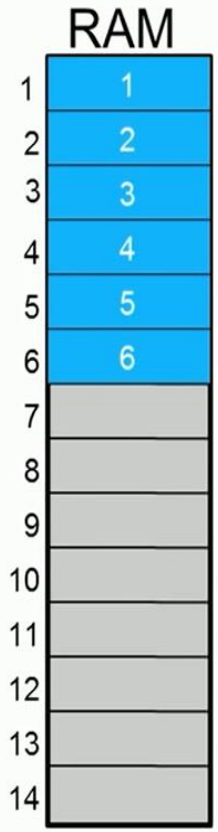
Page frames typically 4KBs



Process split into blocks of equal size.

block size = page frame size

Virtual Memory

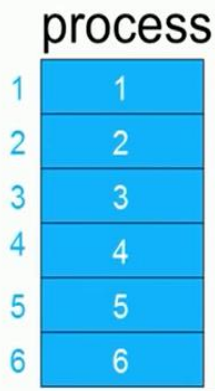


RAM is split into fixed size partitions called page frames.

Page frames typically 4KBs

Process split into blocks of equal size.

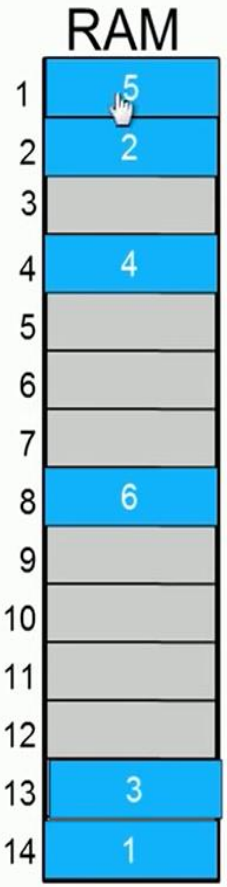
block size = page frame size

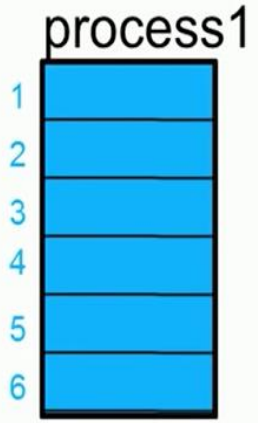


Per process page table (stored in RAM)

block	page frame
1	1
2	2
3	3
4	4
5	5
6	6

Virtual Memory





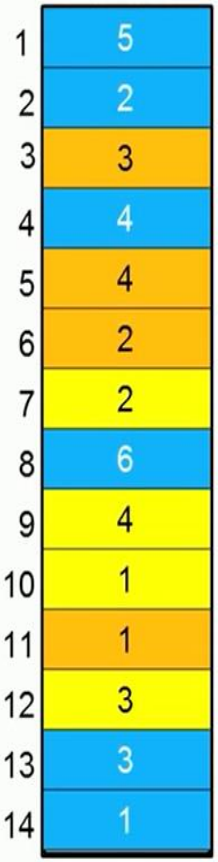
process page table

block	page frame
1	14
2	2
3	13
4	4
5	1
6	8

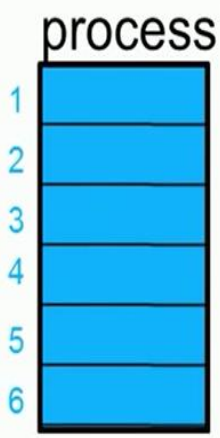
Because of the page table, blocks need not be in contiguous page frames

Every time a memory location is accessed, the processor looks into the page table to identify the corresponding page frame number.

Virtual Memory

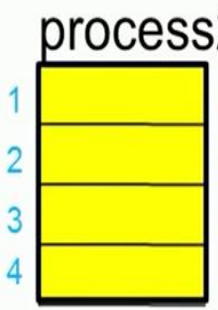


Blocks from Several processes can share pages in RAM simultaneously



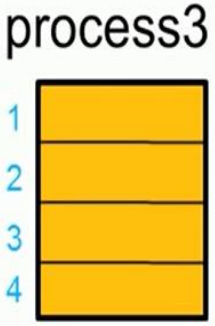
process page table

block	page frame
1	14
2	2
3	13
4	4
5	1
6	8



process page table

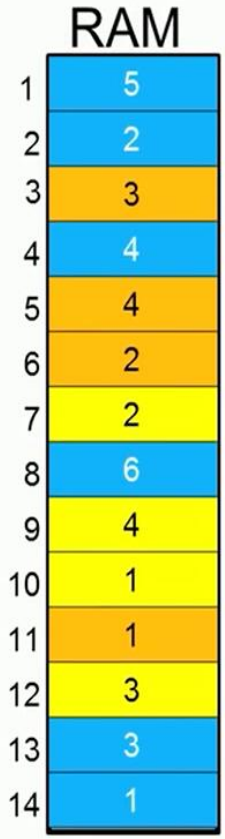
block	page frame
1	10
2	7
3	12
4	9

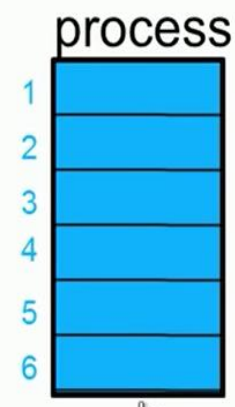


process page table

block	page frame
1	11
2	6
3	3
4	5

Virtual Memory





process page table

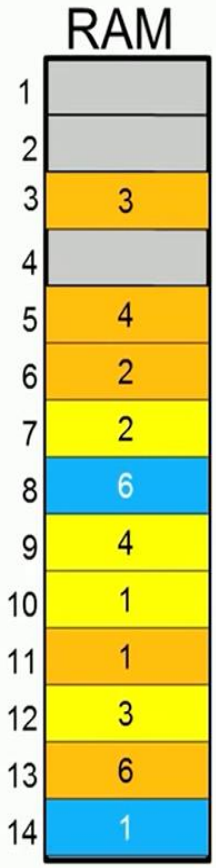
block	page frame
1	14
2	2
3	13
4	4
5	1
6	8

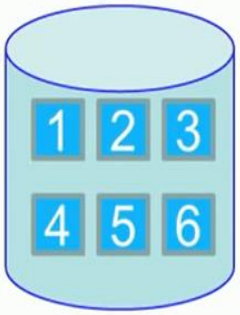
Do we really need to load all blocks into memory before the process starts executing?

No.

Not all parts of the program are accessed simultaneously.
Infact, some code may not even be executed.

Demand Paging





process page table in RAM

block	page frame	P
1	14	1
2		0
3		0
4		0
5		0
6	8	1

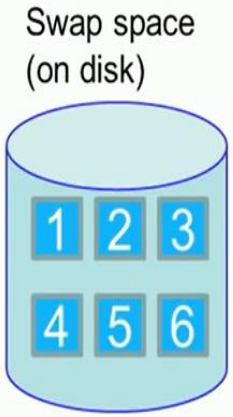
← present bit

Pages are loaded from disk to RAM, only when needed.

A 'present bit' in the page table indicates if the block is in RAM or not.

If (present bit = 1){ block in RAM}
else {block not in RAM}

Demand Paging



process page table in RAM

block	page frame	P
1	14	1
2		0
3		0
4		0
5	1	1
6	8	1

← present bit

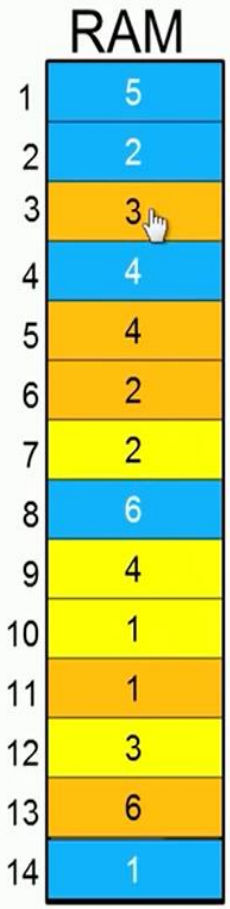
Pages are loaded from disk to RAM, only when needed.

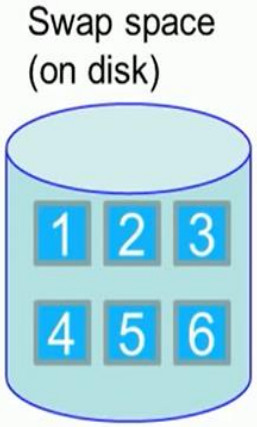
A 'present bit' in the page table indicates if the block is in RAM or not.

If (present bit = 1){ block in RAM}
else {block not in RAM}

If a page is accessed that is not present in RAM, the processor issues a page fault interrupt, triggering the OS to load the page into RAM and mark the present bit to 1

Demand Paging





process page table in RAM

block	page frame	P
1	14	1
2	2	1
3		0
4	4	1
5	1	1
6	8	1

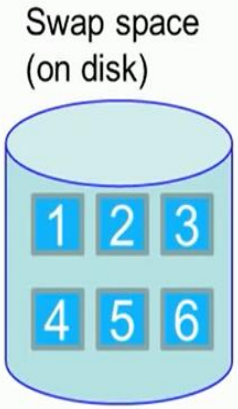
If there are no pages free for a new block to be loaded, the OS makes a decision to remove another block from RAM.

This is based on a replacement policy, implemented in the OS.

Some replacement policies are
* First in first out
* Least recently used
* Least frequently used

The replaced block **may** need to be written back to the swap (swap out)

Demand Paging



process page table in RAM

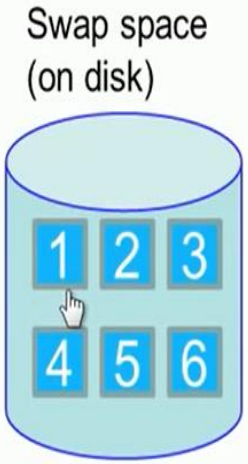
block	page frame	P
1	14	1
2	2	1
3		0
4	4	1
5	1	1
6	8	1

If there are no pages free for a new block to be loaded, the OS makes a decision to remove another block from RAM.

This is based on a replacement policy, implemented in the OS.

Some replacement policies are
* First in first out
* Least recently used
* Least frequently used

The replaced block **may** need to be written back to the swap (swap out)



process page table in RAM

block	page frame	P
1	14	0
2	2	1
3	14	1
4	4	1
5	1	1
6	8	1

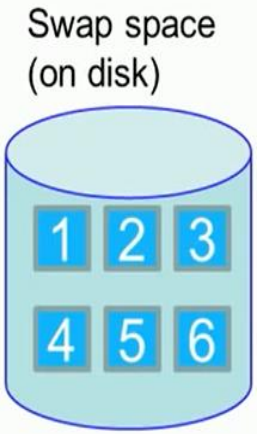
If there are no pages free for a new block to be loaded, the OS makes a decision to remove another block from RAM.

This is based on a replacement policy, implemented in the OS.

Some replacement policies are
* First in first out
* Least recently used
* Least frequently used

The replaced block **may** need to be written back to the swap (swap out)

Demand Paging



process page table in RAM

block	page frame	P	D
1	14	0	1
2	2	1	1
3	14	1	0
4	4	1	1
5	1	1	0
6	8	1	1

The **dirty bit**, in the page table indicates if a page needs to be written back to disk

If the dirty bit is 1, indicates the page needs to be written back to disk.

Finished Unit 4