

Unit-4

# Memory Management

Operating System - 01CE1401

Dr. Krunal Vaghela

# Memory...

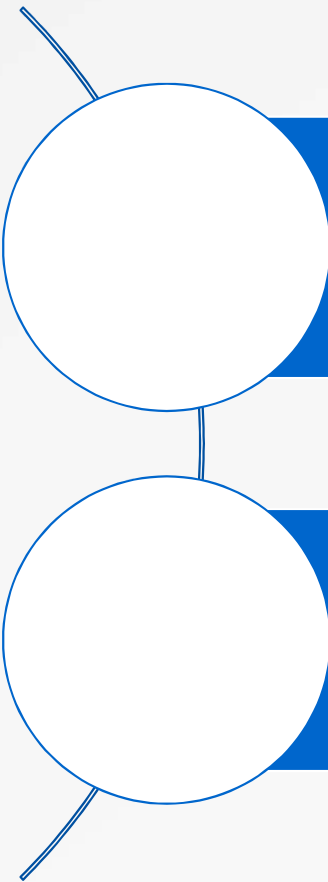


Second major component of computer system is memory after processors.

CPU can only access main memory and register.

Thus if we want to process some data then that data must be fetched into main memory from disk.

# Memory Management: Definition




As the main memory have limited space (generally less than the hard disk), it may causes some issues related to memory allocation and deallocation.

Is the task carried out by the OS and hardware to accommodate multiple processes in main memory

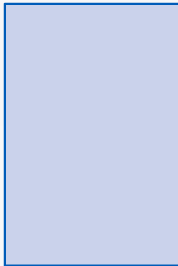
Definition: Computer memory is a physical device, capable to store information temporarily or permanently.

- There are various types of memory like,
  1. Random access memory(RAM) – volatile memory
  2. Read only memory(ROM) – non-volatile memory
  3. Programmable read only memory(PROM) –programmed by user, once it is programmed, the instruction can not changed.
  4. Erasable programmable read only memory(EPROM)- it can be programmed. To erase data from that it used ultra violet light.
  5. Electrically erasable programmable read only memory(EEPROM)- it erase the data by using electric filed. No need to use ultra violet light.


# MEMORY MANAGEMENT: function



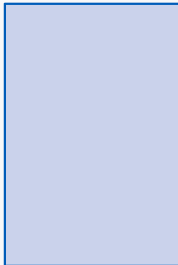
1. Keep track of what parts of memory are in use.



2. Allocate memory to processes when needed.



3. Deallocate when processes are done.



4. Swapping, or paging, between main memory and disk, when disk is too small to hold all current processes.

# Logical and Physical address map

## Logical address/ virtual address

- address generated by the CPU
- Range of logical address are limited to the size of processor.
- Example: 32-bit processor then address range would be up to  $2^{32}$  (4GB).

## Physical address

- Address seen by Main memory unit.
- Range is depend on the size of memory.

# Logical and Physical address map

## Logical address space

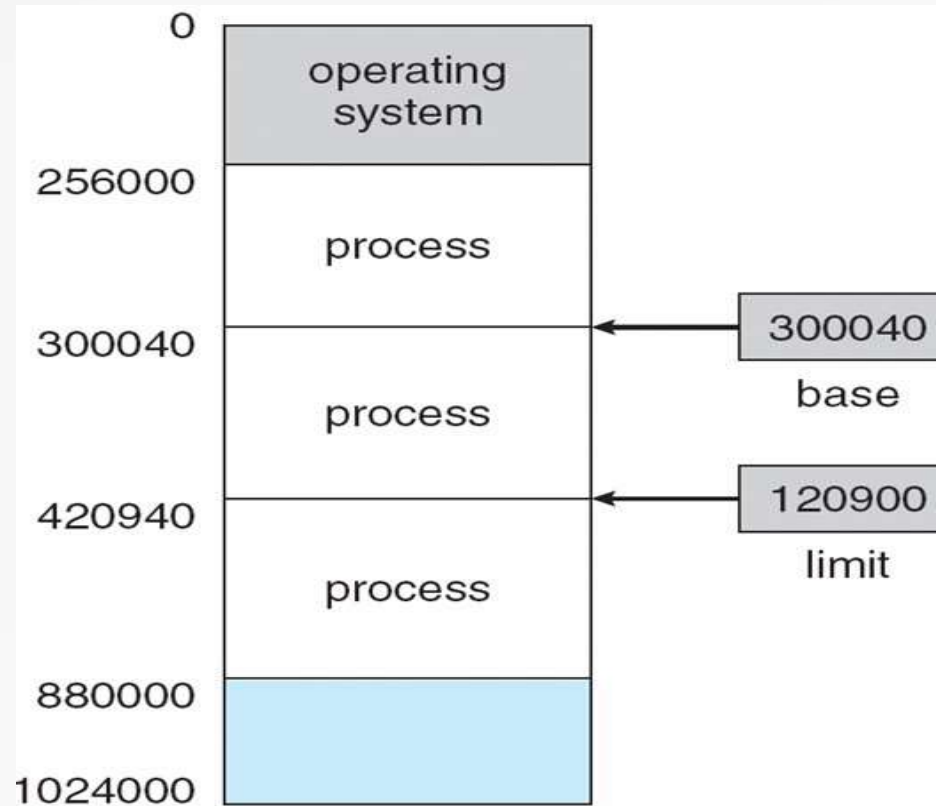
- The set of all logical addresses generated by a program is known as logical address space.

## Physical address space

- The set of all physical addresses corresponding to the logical addresses is known as physical address space.

# Logical and Physical address map

- **Hardware Address Protection**

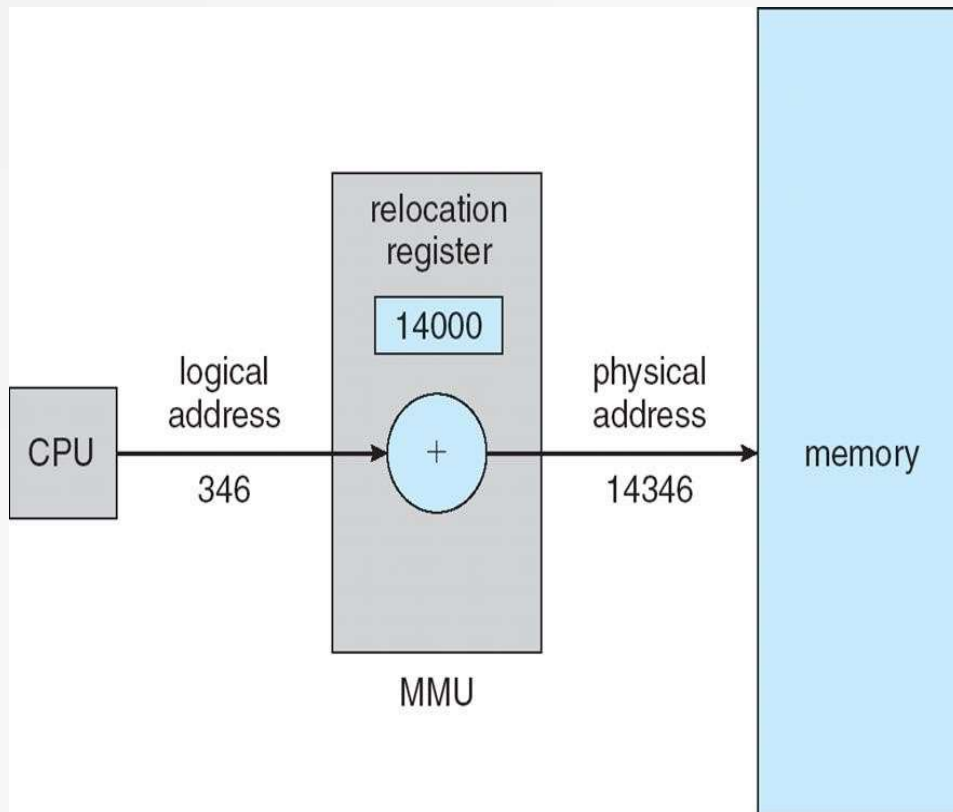




# Logical and Physical address map

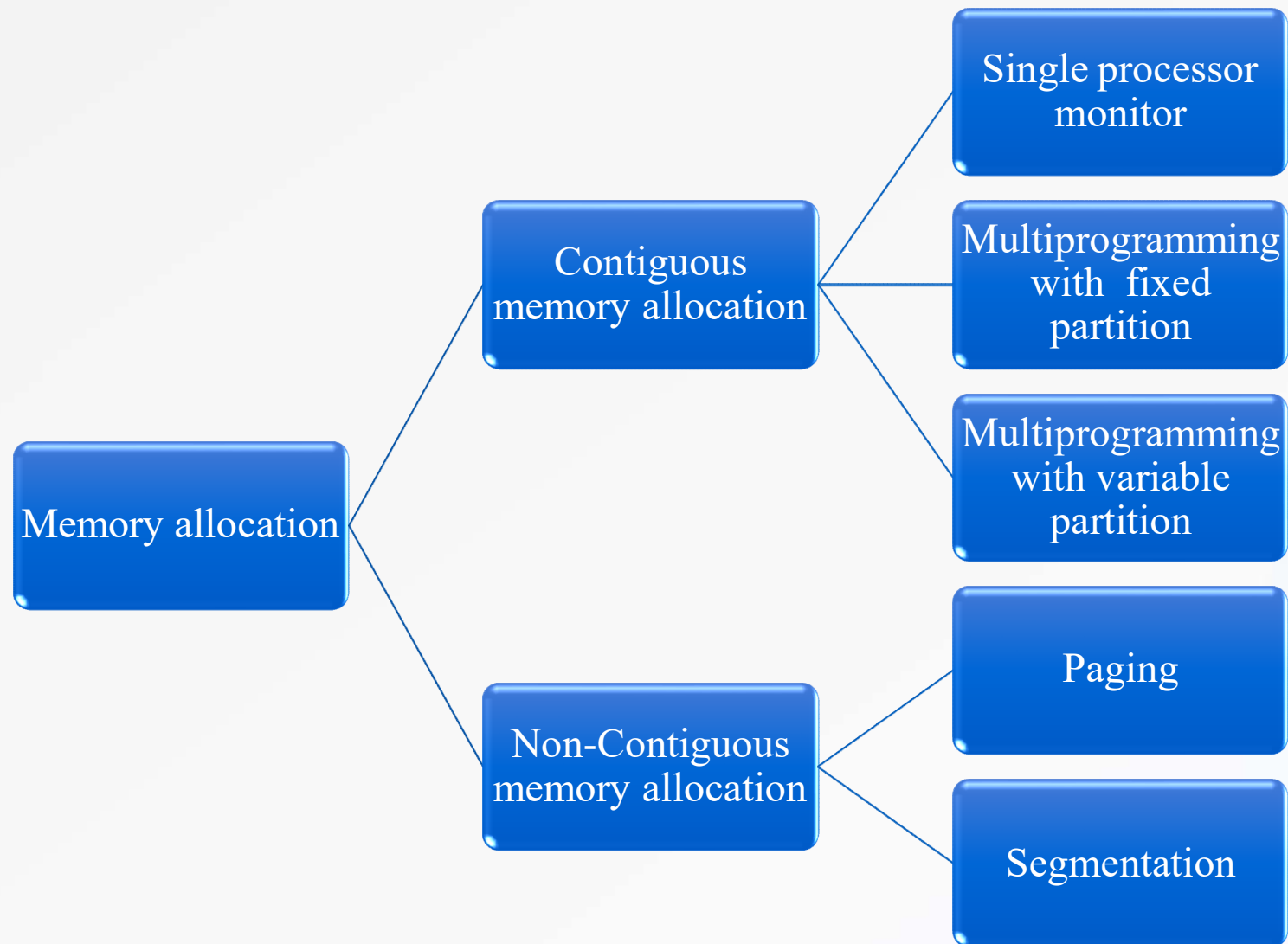
- For any process to get executed it should be in main memory.
- Thus whenever process (code, data, stack) fetched in main memory, it requires some storage space.
- In main memory physical address of process may not exactly match with the logical address that we need mapping between logical and physical address.
- Part of the OS manages the main memory is known as Memory Management Unit (MMU).

# Logical and Physical address map



Base register is now known as relocation register.

The value in the relocation register added to every address generated by a user process when process is sent to memory.



# Memory allocation

## Contiguous Memory allocation

- Simple and old method.
- Here each process occupies contiguous block of main memory.
- When process is brought in memory, a memory is searched to find out a chunk of free memory having enough size to hold a process.

# Memory allocation: Contiguous Memory allocation

## Single processor monitor

- Simplest possible schema.
- Only single process is allowed to run.
- Memory is only shared between single process and OS.

OS

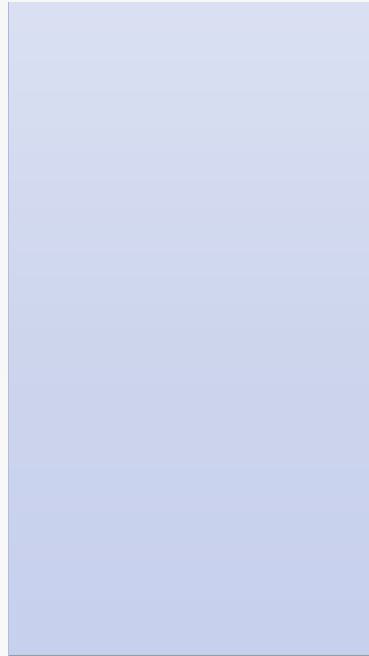
User Program

# Memory allocation: Contiguous Memory allocation

## Multi programming with fixed partition

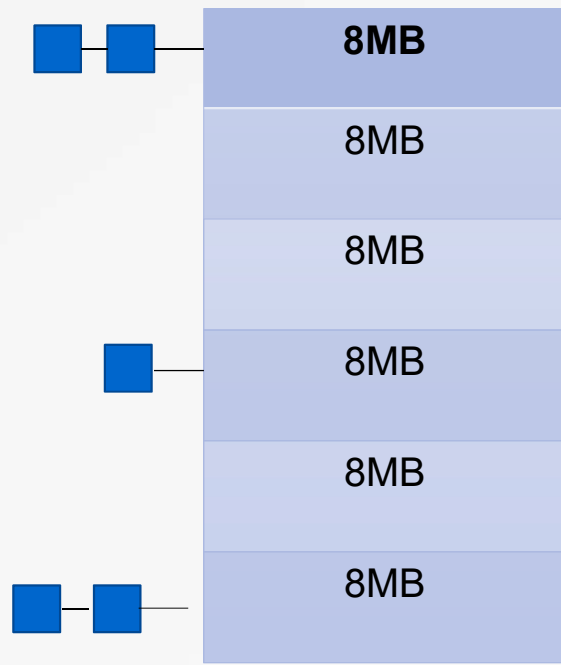
- Allows to *execute multiple* process simultaneously.
- Here, memory is divided into fixed size partition.
- Each partition may contain exactly one process.
- When a partition is free, process is selected from the input queue and it is loaded into free partition.

# Memory allocation: Contiguous Memory allocation

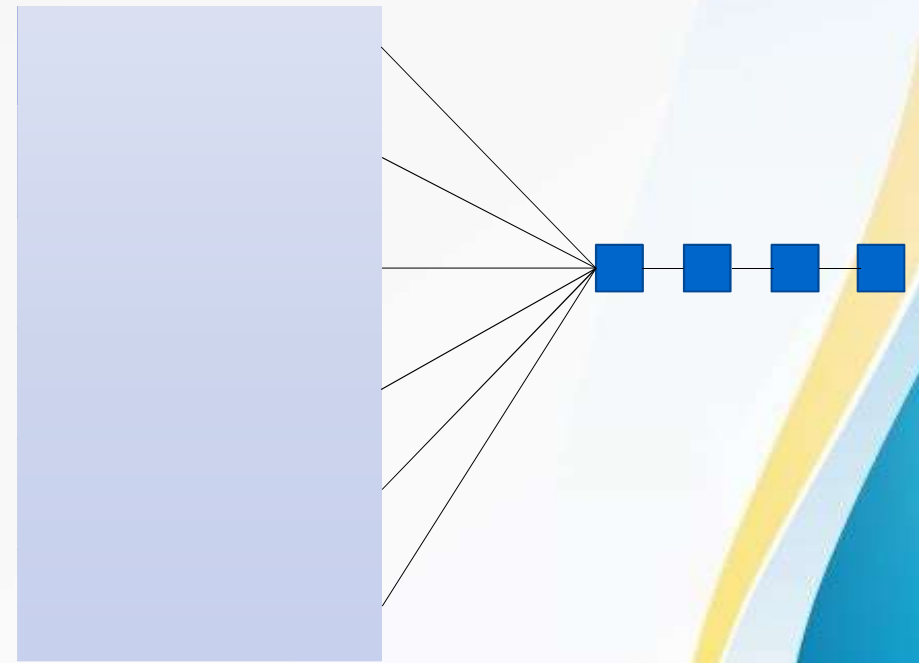


Multi programming with fixed partition

# Memory allocation: Contiguous Memory allocation



Fixed partition with multiple queue



Fixed partition with single queue



# Memory allocation: Contiguous Memory allocation

## Multi programming with fixed partition:

- *Advantage:*
  - Implantation is simple.
  - Processing overhead is low.
- *Disadvantage:*
  - Degree of multiprogramming is fixed because memory is divided into fixed partition and each partition can contain one process.
  - Causes Internal fragmentation because Partition are of fixed size.

# Memory allocation: Contiguous Memory allocation



## Multi programming with fixed partition:

- *Internal fragmentation:*
  - because Partition are of fixed size thus any space in a partition which is not used by process is wasted.
- *Example:*
  - process1 requires 4MB, Process 2 requires 8mb, Process 3 requires 6mb, Process 4 requires 3mb, Process 5 requires 8mb.
  - Here in total we have 8mb free in the memory but due to internal fragmentation we can not assign that to process 5.

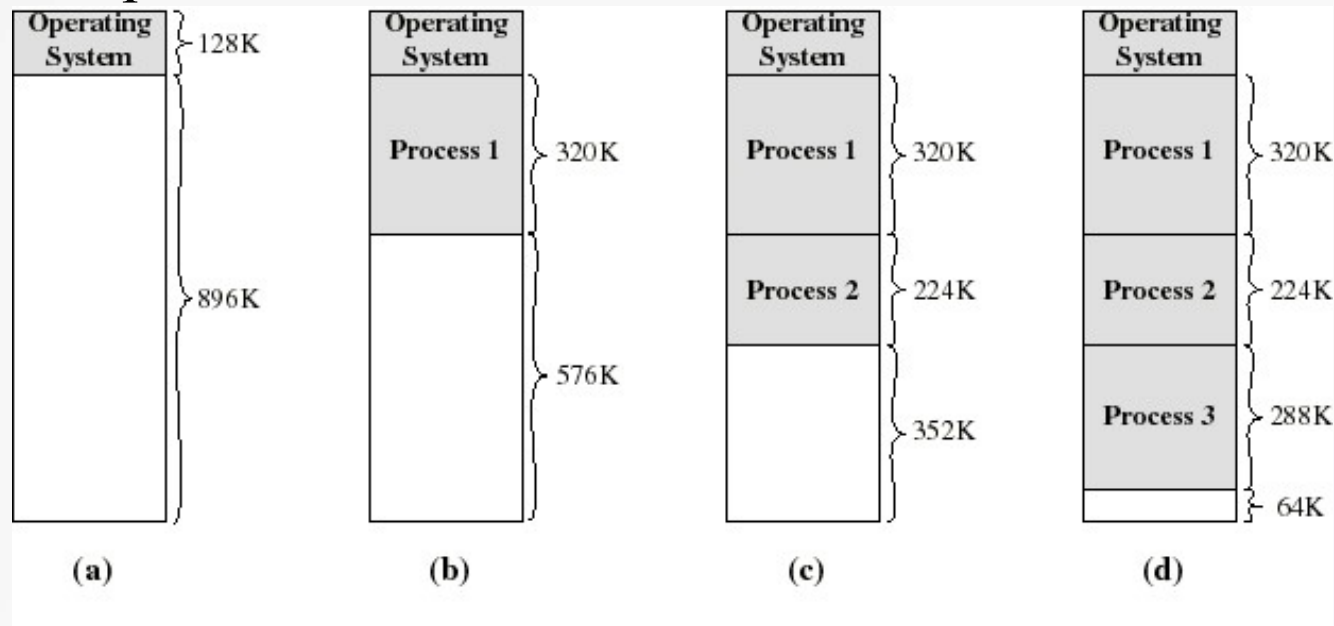
# Memory allocation: Contiguous Memory allocation

## Multi programming with variable/dynamic partition

- Here memory is not divided into fixed partition, also the number of partition is not fixed.
- Only required memory is allocated to process.
- Whenever any process enter in a system, a chunk of memory big enough to fit the process is found and allocated. And the remaining unoccupied space is treated as another free partition.
- When process get terminated it releases the space occupied and if that free partition is contiguous to another free partition then that both free partition can be merge.

# Memory allocation: Contiguous Memory allocation

- **Multi programming with variable/dynamic partition:**
  - Example:



# Memory allocation: Contiguous Memory allocation

- Example: process 1 needs 4mb, process 2 needs 8mb, process 3 needs 6mb, process 4 needs 3 mb, process 5needs 8 mb.

P1
4 MB
P2
8 MB
P3
6 MB
P4
3 MB
P5
8 MB

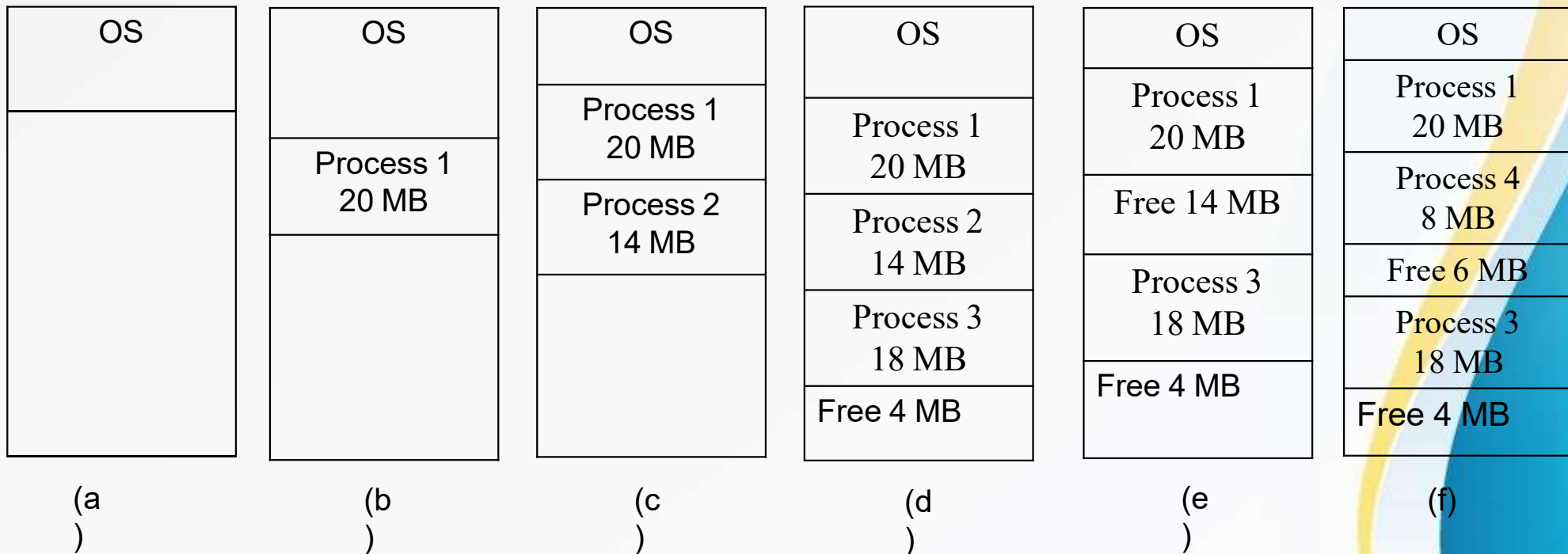
# Memory allocation: Contiguous Memory allocation

## Multi programming with variable/dynamic partition:

- *Advantage:*
  - Better utilization of memory
  - Degree of multiprogramming is not fix here.
- *Disadvantage:*
  - Causes External fragmentation:
    - Memory is allocated when process enters into system, and deallocated when terminates. This operation may leads to small holes in the memory.
    - This holes will be so small that no process can be loaded in it..
    - But total size of all holes may be big enough to hold any process.

# Memory allocation: Contiguous Memory allocation

- **Multi programming with variable/dynamic partition:**
  - Disadvantage: Causes External fragmentation:



# Memory allocation: Contiguous Memory allocation

- **Multi programming with variable/dynamic partition:**

- Disadvantage:

- Causes External fragmentation:

- (a). Initial state
      - (b). Process 1 enters.
      - (c). Process 2 enters.
      - (d). Process 3 enters.
      - (e). Process2 terminates.
      - (f). Process 4 enters
      - (g). Process 5 needs 10 MB.



# Memory allocation: Contiguous Memory allocation

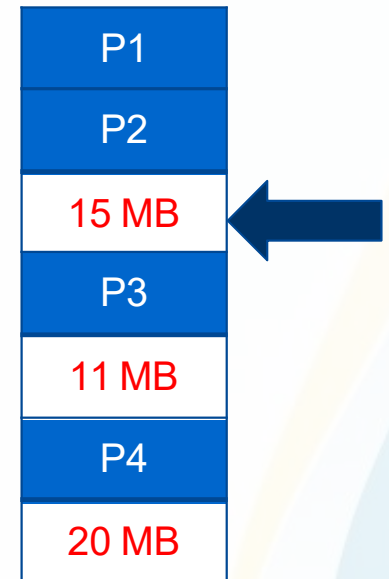
Multi  
programming  
with  
variable/dynamic  
partition

- Whenever any process enters in system, requires free memory to allocate memory to the process.
- For this purpose all the free memory is searched to find out chunk of required free memory.
- This procedure is instance of *general dynamic storage allocation problem*, which concern about how to satisfy request of size  $n$  from list of holes.
- There are basically three solutions: *first fit, best fit, worst fit*.

# Memory allocation: Contiguous Memory allocation

## Multi programming with variable/dynamic partition

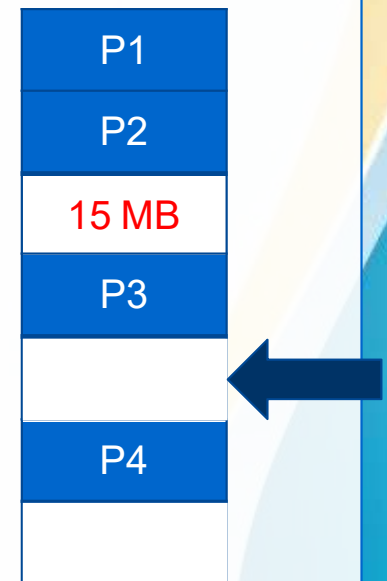
- First fit:
  - Searching start from the beginning of the memory.
  - Allocates the first hole that is big enough to hold process.
- Search time less.
  - May causes some external fragmentation.
  - Memory loss is high.
  - Example: processes P1, P2, P3, P4
- Are already in memory 3 free holes
- of 15MB, 11MB, 20MB.
- Now process P5 comes with needed
- memory is 10MB



# Memory allocation: Contiguous Memory allocation

## Multi programming with variable/dynamic partition

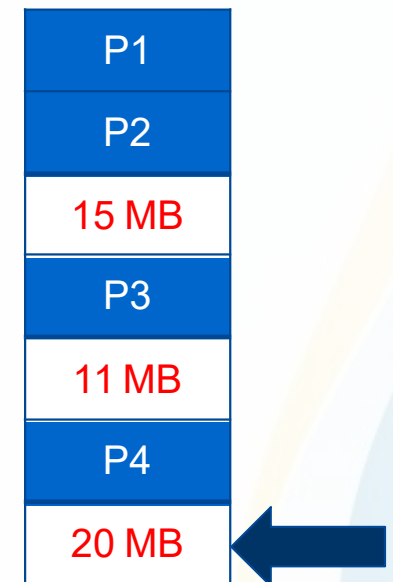
- Best fit:
  - Entire memory is searched to find smallest hole
  - which is large enough to hold the process.
  - Search time high.
  - Memory loss is less.
  - May cause more external fragmentation.
  - Example: processes P1, P2, P3, P4 are already in memory
- 3 free holes of 15MB, 11MB, 20MB.
- Now process P5 comes with needed memory is 10MB.



# Memory allocation: Contiguous Memory allocation

## Multi programming with variable/dynamic partition

- Worst fit:
  - Entire memory is searched
  - to find largest hole which is big enough
  - to hold the process.
  - Search time high.
  - Example: processes P1, P2, P3, P4 are already
  - in memory 3 free holes of 15MB, 11MB, 20MB.
  - Now process P5 comes with needed memory is 10MB.



# Memory allocation: Contiguous Memory allocation

## Solution of external fragmentation

- One of the solution can be define as compaction:
  - Shuffle memory contents to place all free memory together in one large block.
  - Compaction is possible only if using relocatable logical addresses.
- Other solution of fragmentation is paging and segmentation.

# Non-Contiguous Memory allocation



This method is used by most modern OS.

The diagram illustrates the concept of non-contiguous memory allocation through three vertically stacked blue rectangular boxes. Each box is preceded by a white circle with a blue outline. The circles are connected by thin blue lines, suggesting a sequence of points or steps. The first circle is at the top, the second in the middle, and the third at the bottom. The text within each box describes a characteristic of non-contiguous memory allocation.

Here logical address space of process is divided into partition.

Physical address space will not be contiguous.

- Two methods:
  - Paging and Segmentation

# Non-Contiguous Memory allocation : Paging

## Principle of operation

- Physical memory is divided into fixed sized block called *frames*.
- Logical address space is divided into blocks of fixed size called *pages*.
- Page and frame will be of same size.
- Whenever a process needs to get execute on CPU, its pages are moved from hard disk to available frame in main memory.

# Non-Contiguous Memory allocation : Paging

## Principle of operation

- Thus here memory management task is:
  - to find free frame in main memory,
  - allocate appropriate frame to the page,
  - keeping track of which page belong to which frame.
- OS maintains a table called *page table*, for each process.
- Page table is index by page number and stores the information about frame number.



page 0
page 1
page 2
page 3

logical  
memory

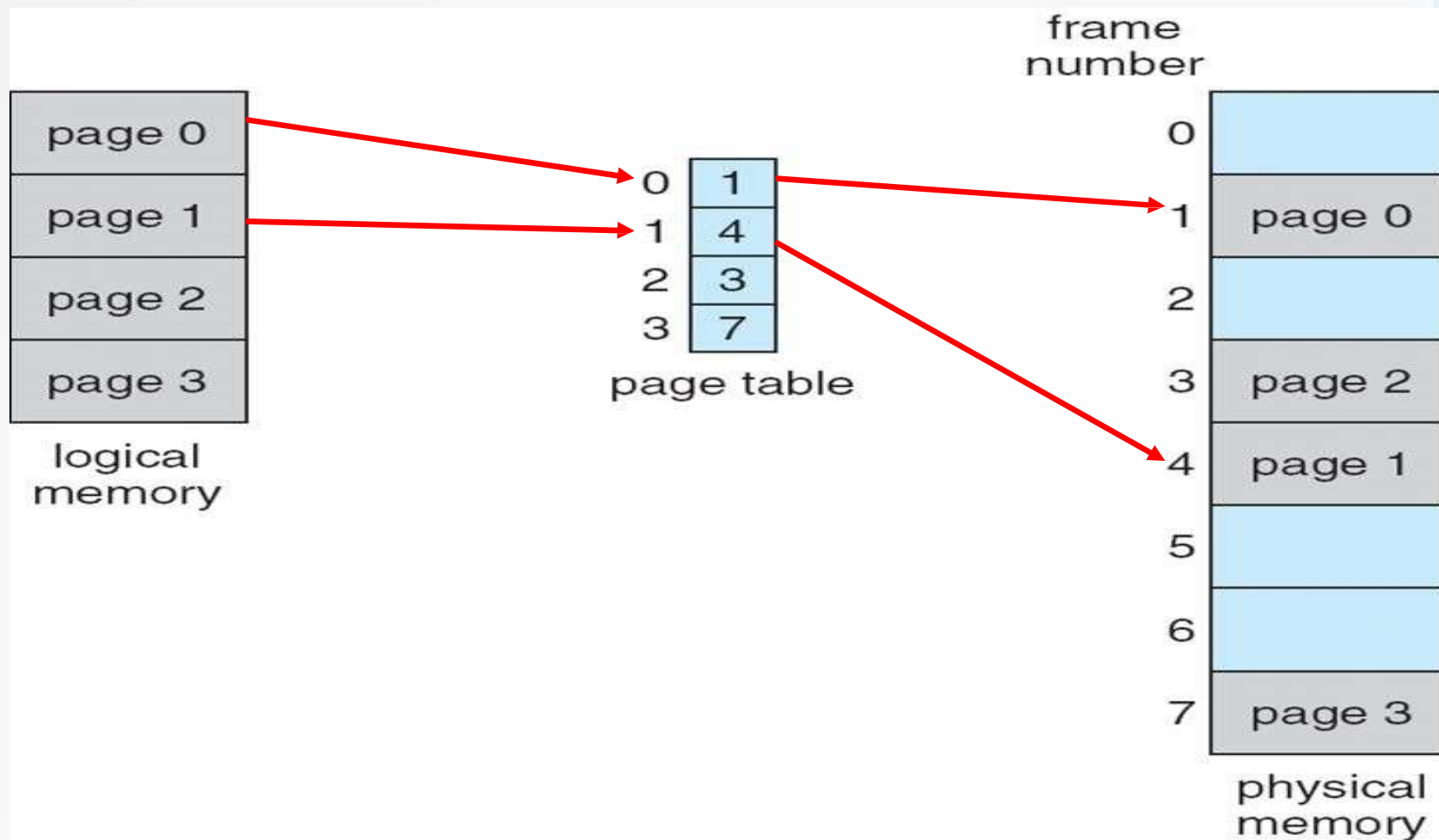
0	1	Page number	Frame number
1	4		
2	3		
3	7		

page table

frame  
number

0	
1	page 0
2	
3	page 2
4	page 1
5	
6	
7	page 3

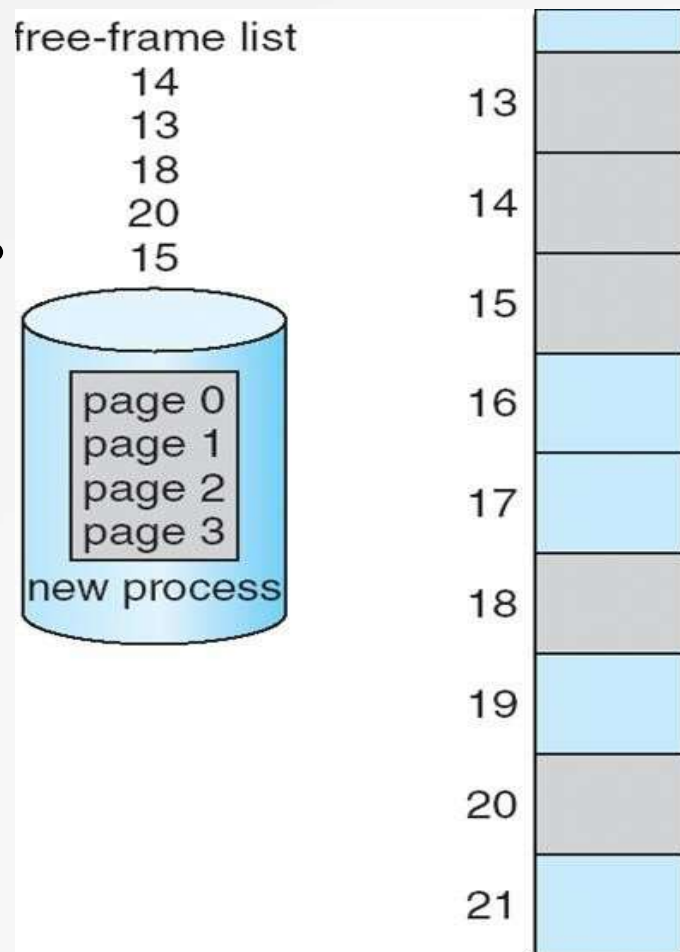
physical  
memory



# Non-Contiguous Memory allocation

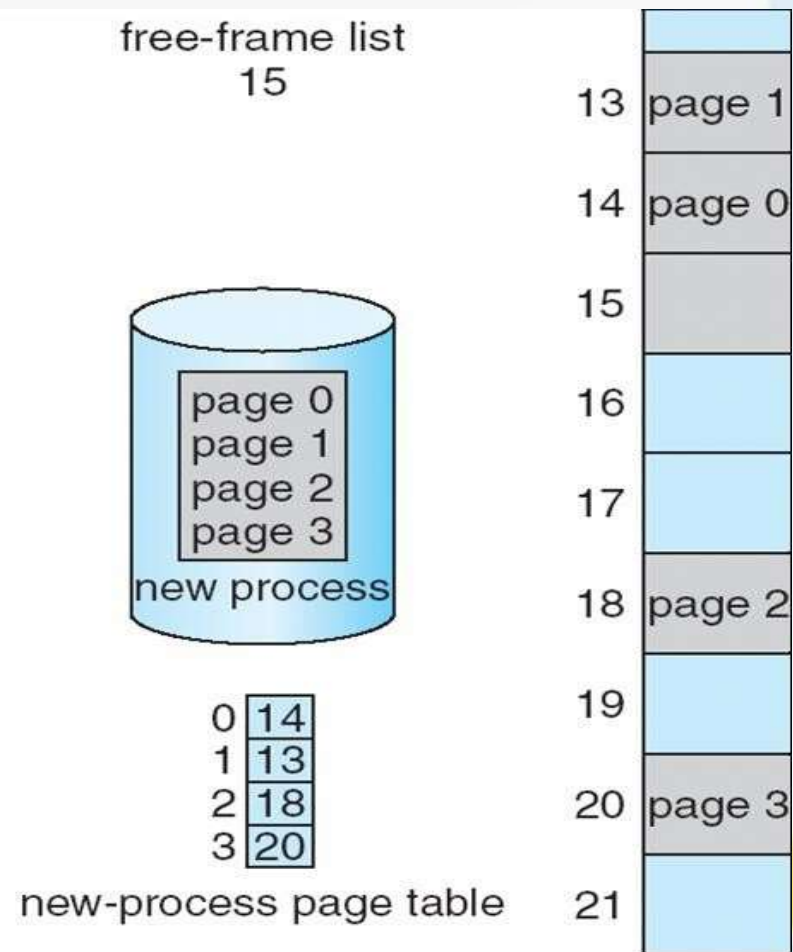
## Page allocation in Paging system

- Here one *free frame list* is maintain.
- When a process arrives in the system to be executed, size of process is expressed in terms of number of pages.
- Each page of the process needs one frame.
- Thus if process requires  $n$  pages then  $n$  frames must be free in memory.



(a)

Before allocation



(b)

After allocation

# Paging: Hardware support

Logical address is divided into two parts:

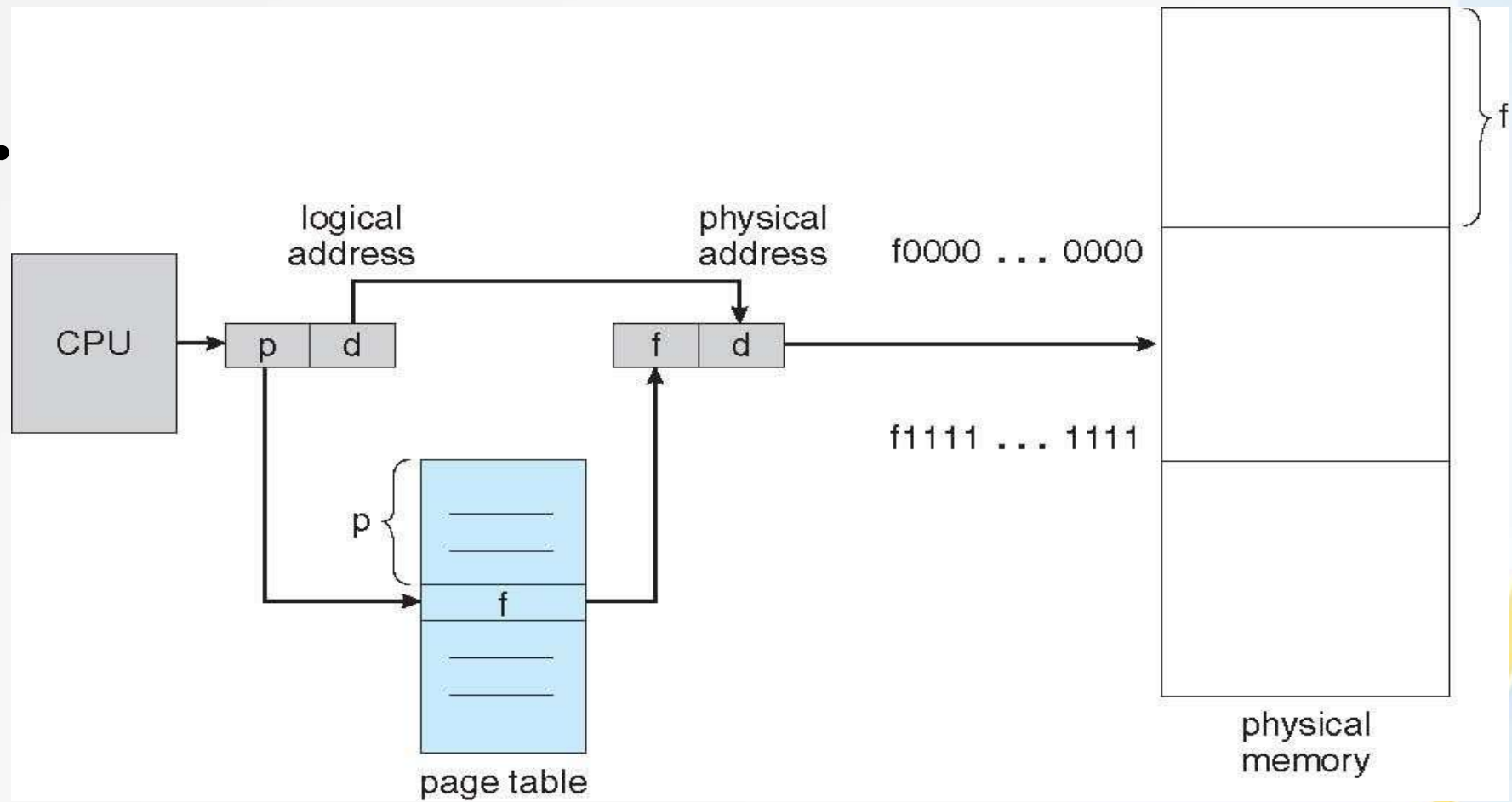
- Page number (p)
- Offset (d)

Offset part of both the address is always same.


Physical address is also divided in tow parts:

- Frame number (f)
- Offset (d)


Frame number can be searched form page number and offset will provide the exact location of that object.




# Paging: Hardware support




Page table can be implemented in several ways.



One of the simplest method is to implement it with the collection of register.



Most of the computer needs their page table of large size for that schema using register for page table is not feasible.



Rather page table is kept in main memory. This may cause high access time.

# Paging: Hardware support

Two overlapping blue squares, one slightly offset to the left and top of the other, serving as a decorative element.

Thus Morden OS uses variations of  
Paging which is

- Translation look aside buffer
- Hierarchical paging



# Paging: structure of page table

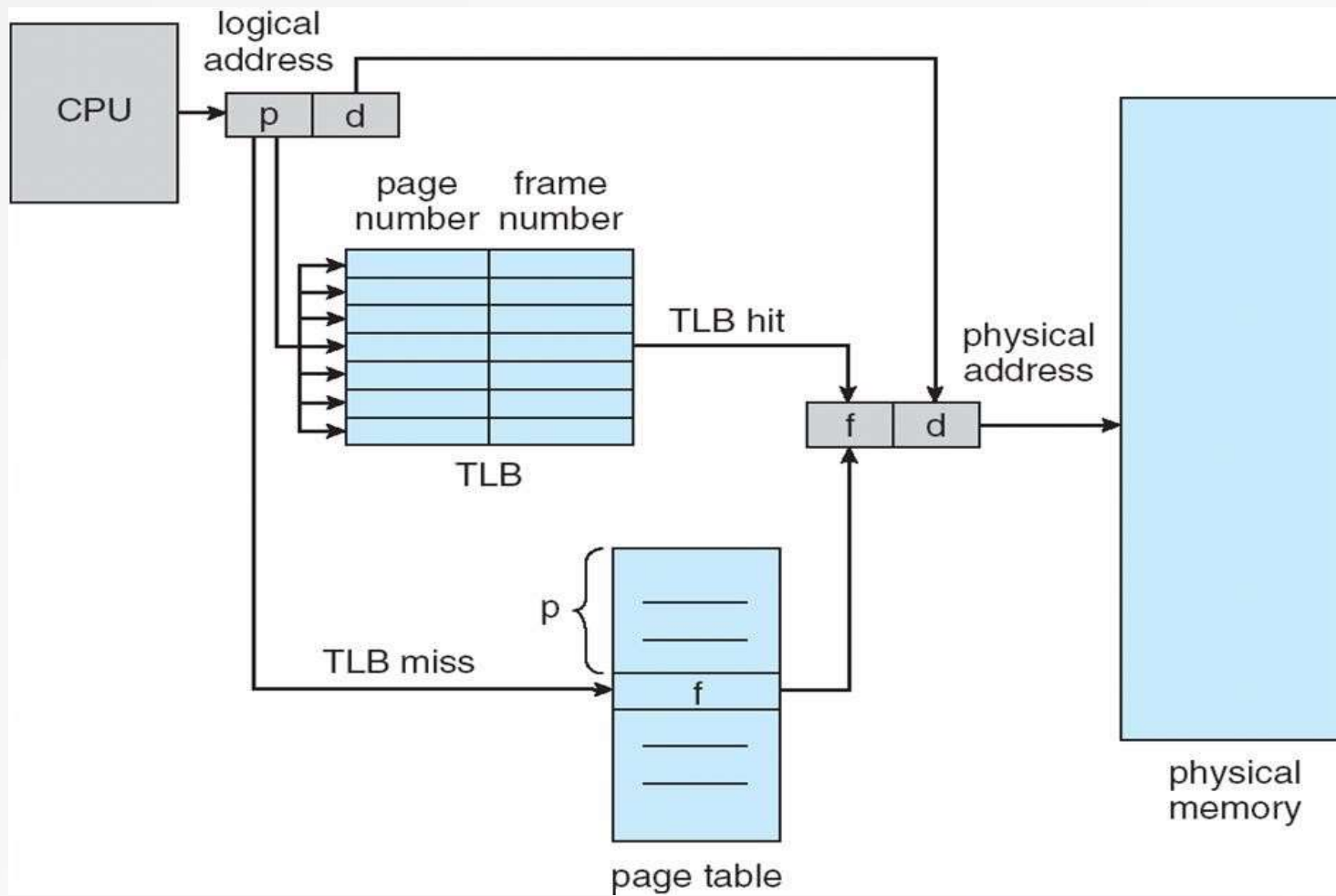
## 1. Translation look aside buffer

- Used to overcome the slower access problem.
- TLB *is page table cache*, which is implemented in fast associative memory.
- Its cost is high so capacity is limited, thus only subset of page table is kept in memory.
- Each TLB contains a page number and a frame number where the page is stored in the memory.

# Paging: structure of page table

## 1. Translation look aside buffer

- Working:
  - Whenever a logical address is generated, the page number of logical address is searched in the TLB.
  - If the page number is found then it is known as *TLB hit*. In this case corresponding frame number is fetched from TLB entry and used to get physical address. The whole task may take a bit longer than it would if an unmapped memory reference were used.
  - If a match is not found then it is termed as *TLB miss*, in this case a memory reference to that page must be made. page table is used to get the frame number. And this entry is moved to TLB.



# Paging: structure of page table

- Effective Access Time calculation:
  - Effective access time = TLB HIT (C+M) + TLB MISS (C+2M) nanosecond.

# Paging: structure of page table

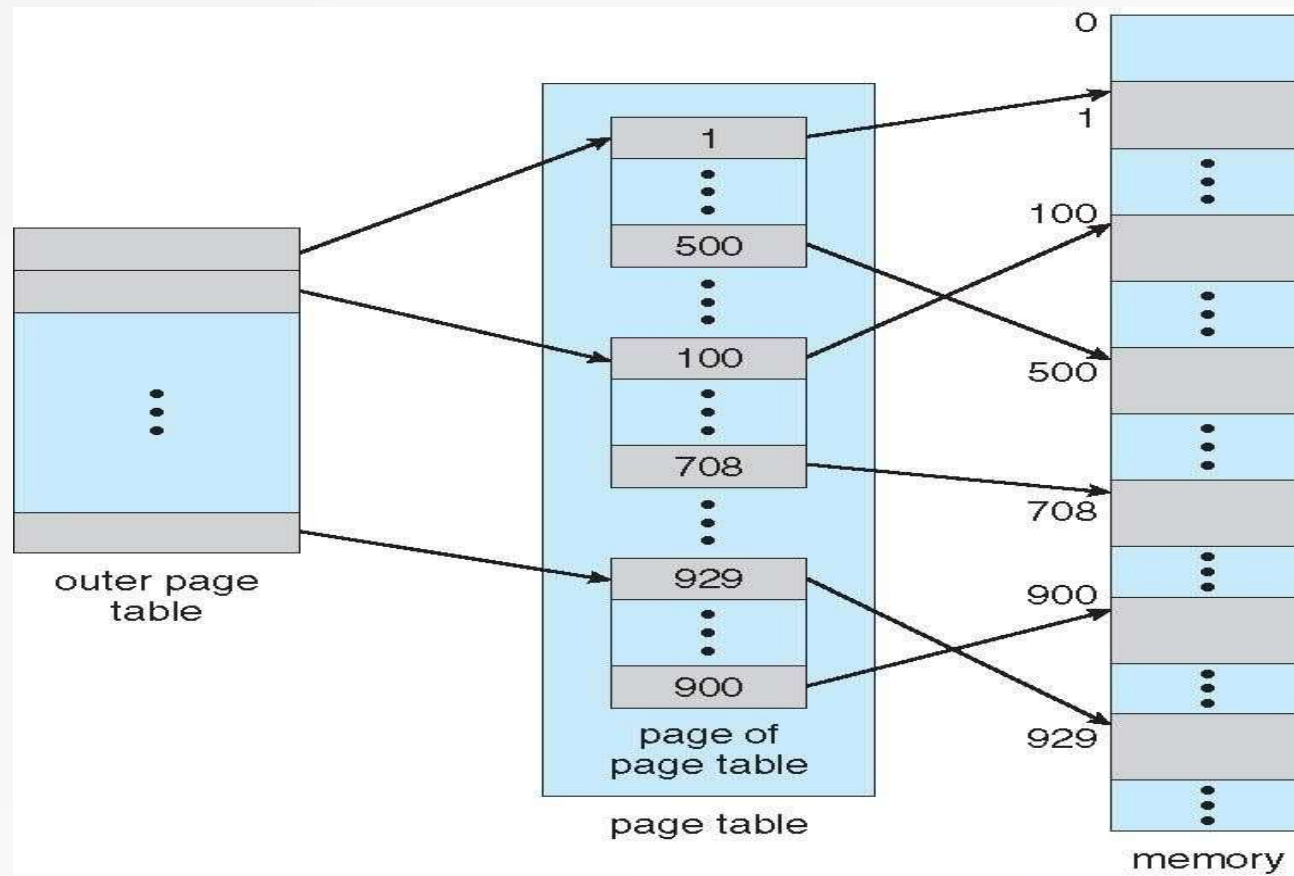
## 2. Hierarchical Paging

- Most modern OS supports large logical address space. In such case the page table itself become excessive large to store in contiguous space.
- Example system with 32-bit ( $2^{32}$ ) logical address space, if the page size is 4kb ( $4096=2^{12}$ ), Then page table size would be 1 million entry.
- One of the solution is to use two level paging algorithm, in which page table itself is also page.

# Paging: structure of page table

## 2. Hierarchical Paging

- Here the page table is divided into various inner page table. Also the extra outer page table is maintained.
- The outer table contains very limited entry, and points to the inner page table.
- And the inner page table in turn gives actual frame number.
- Here logical address is divided into three parts:
  - Page number (p1) (outer page table page number)
  - Page number (p2) (page table page number)
  - Offset (d)



# Paging: structure of page table

## 2. Hierarchical Paging

- Advantage:
  - All the inner pages need not be in memory simultaneously, thus reduce memory overhead.
- Disadvantage:
  - Extra memory access is required in each level of paging. Here total three memory access is required to get data.



# Disadvantages of paging

## Additional memory reference

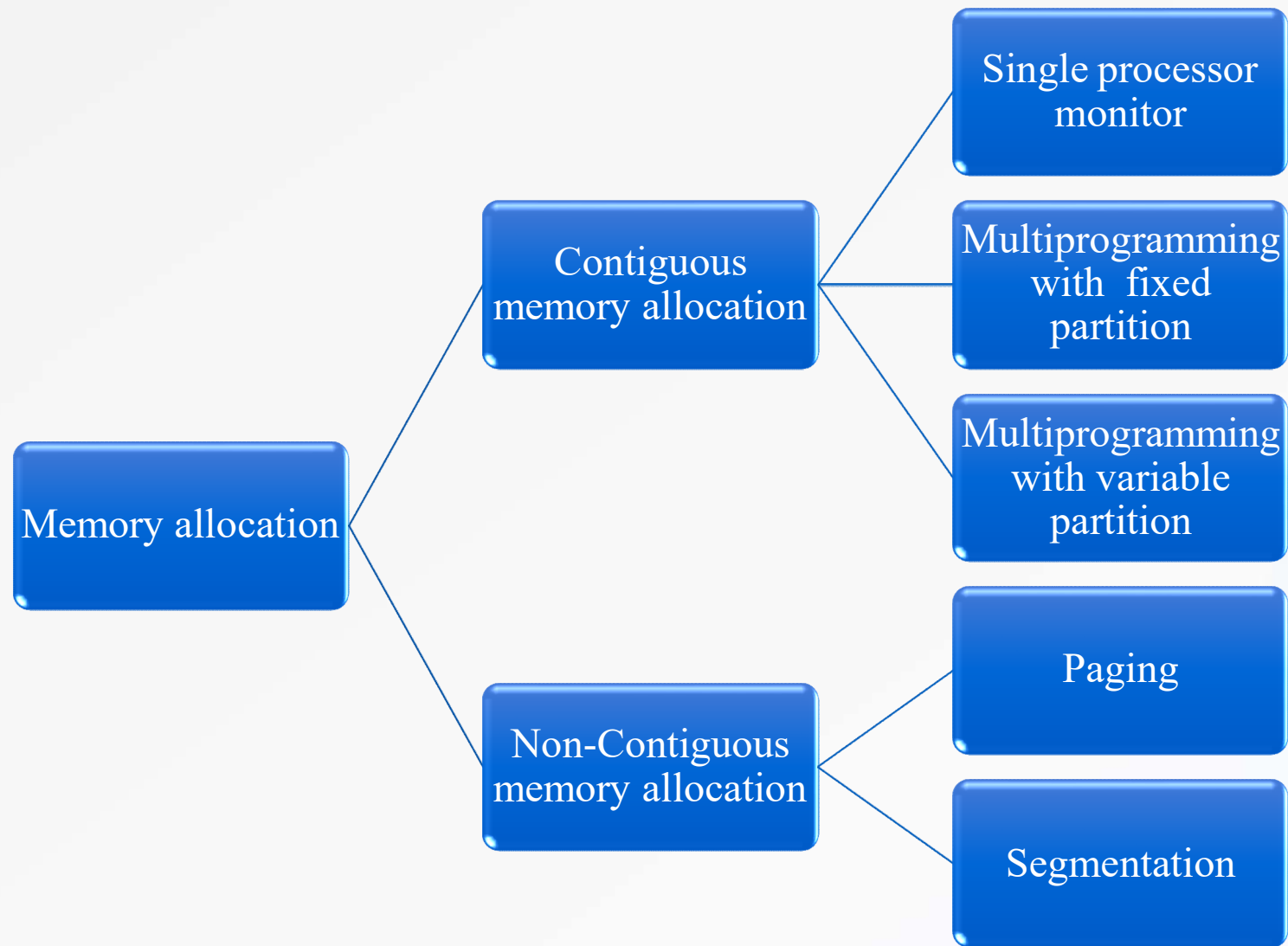
- Its required to read information from page table.
- Every instruction requires two memory accesses: one for page table, and one for instruction or data.

## Size of page table

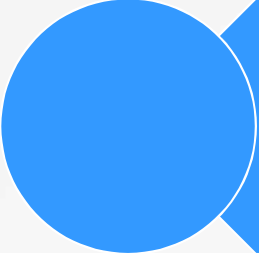
- Page tables are too large to keep it in main memory.
- Page table contain all pages in logical address space thus larger process page table will be large.

## Internal fragmentation

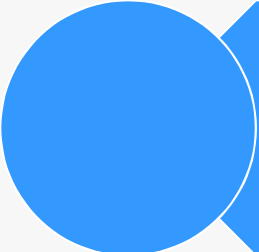
- A process size may not be exactly of the page size.
- So some space would remain unoccupied in the last page of a process. This result in internal fragmentation.



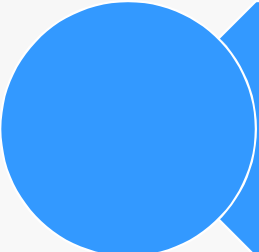
# Segmentation



In paging we refer to have same size of pages, which again turns in “internal fragmentation”



So, to use variable scheme in size , segmentation can be used.



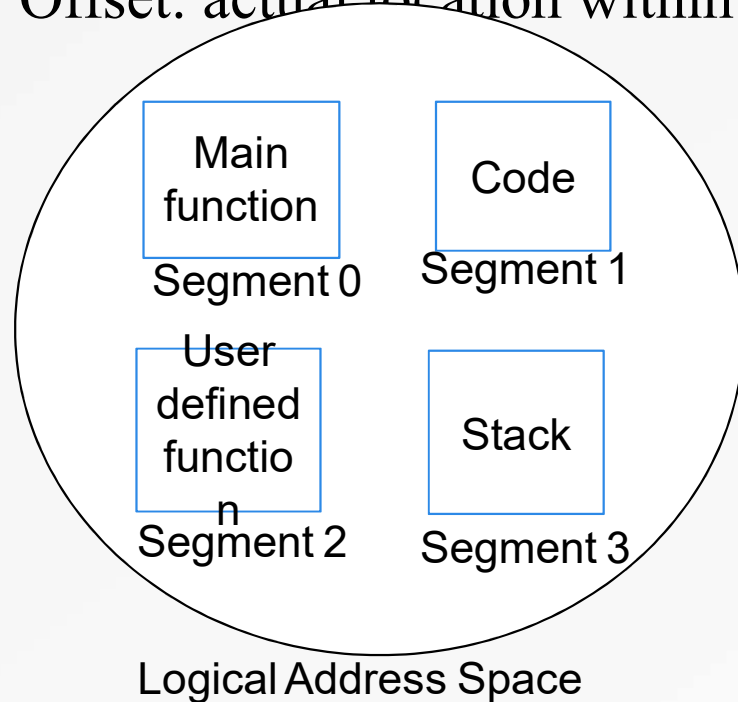
Here the logical address space of a process is *divided into blocks of varying size*, called segments.

# Segmentation

- Each segment contains a logical unit of process.
- When ever a process is to be executed, its segments are moved from secondary storage to the main memory.
- Each segment is allocated a *chunk of free memory of the size equal to that segment*.
- OS maintains one table known as *segment table*, for each process. It includes size of segment and location in memory where the segment has been loaded.

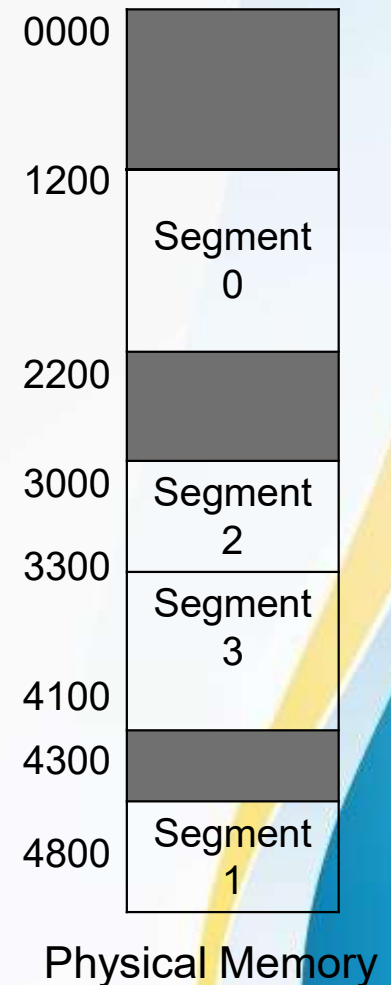
# Segmentation

- Logical address is divided into two parts:
  - 1). Segment number: identifier for segment.
  - 2). Offset: actual location within a segment.

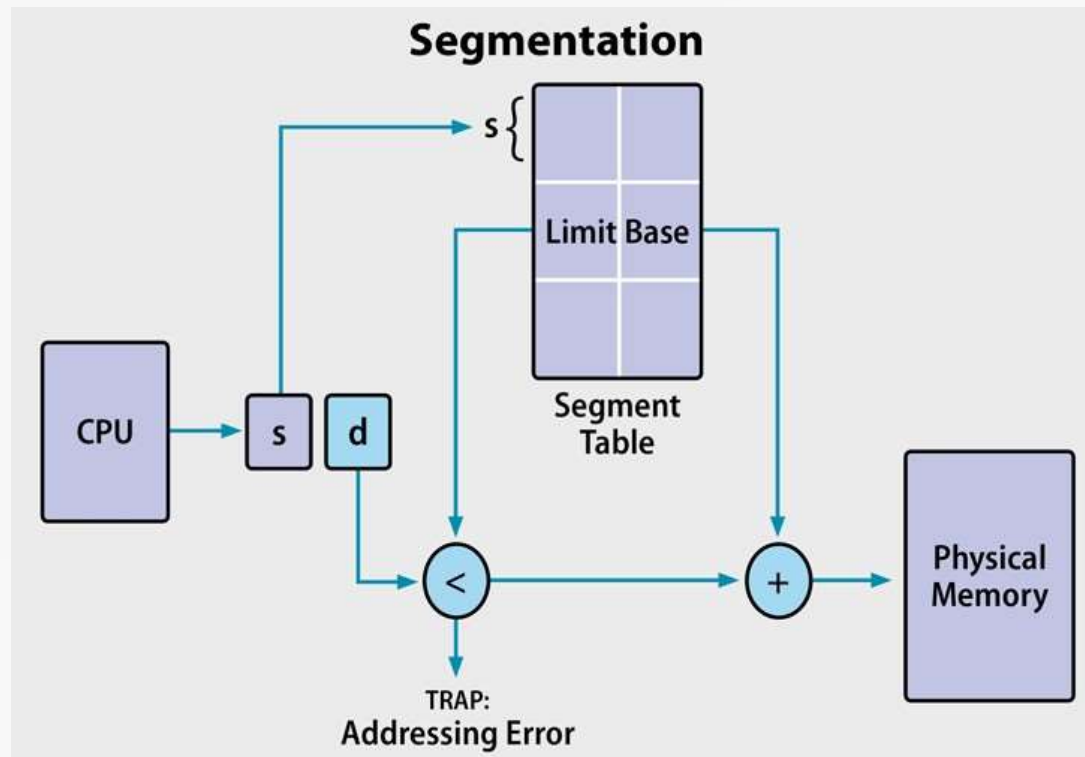


	Limit	Base
0	1000	1200
1	500	4300
2	300	3000
3	800	3300

Segment Table



# Segmentation



# Swapping

A process, in the swapping method is replaced by a different process for some time, when the time gets over, the process back to the main memory.

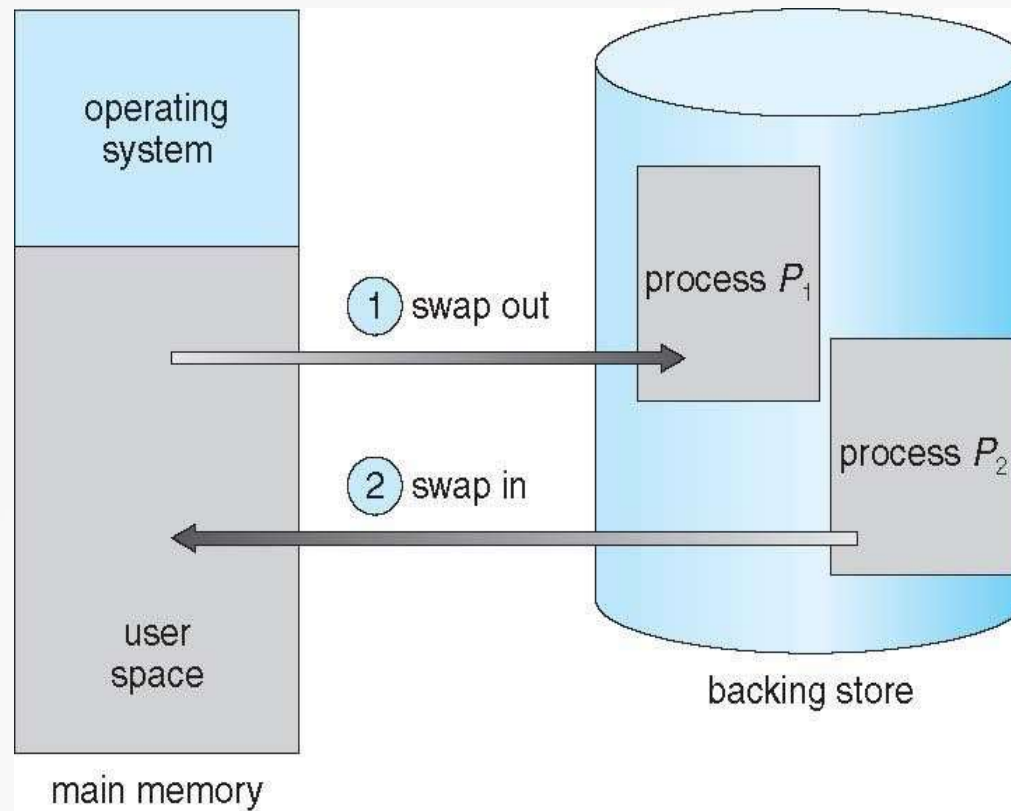
Swapping is technique in which process are moved *between main memory and secondary memory* or disk.

Operation of moving process from memory to swap area is called "*swap out*". And moving from swap area to memory is known as "*swap in*".

For ex. Multiprogramming with Round Robin.

Each process swapped out after completion of TQ, and a different process gets the same address space which was previously carried out by last process.

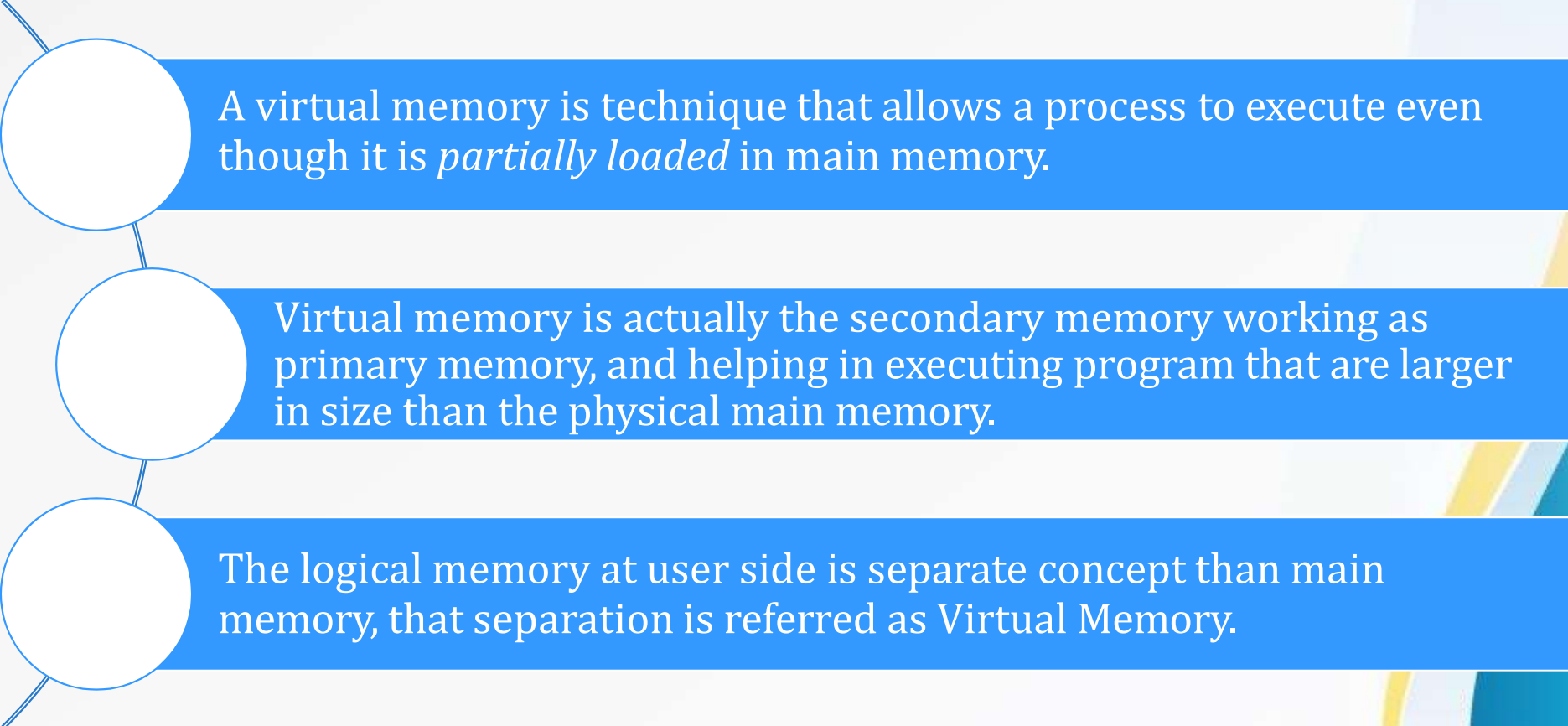
# Swapping





- Swapping is also used in priority based CPU scheduling algorithms.
- Suppose process with low-priority is under execution and any higher priority process arrive for execution.
- So in that case memory manager can swap the executing process with higher priority process.
- When execution of higher priority process completed, then it again swapped out and low priority process can again swapped in and carried out its further execution.
- Limitation of swapping: Increase context- switching time.

# Virtual Memory



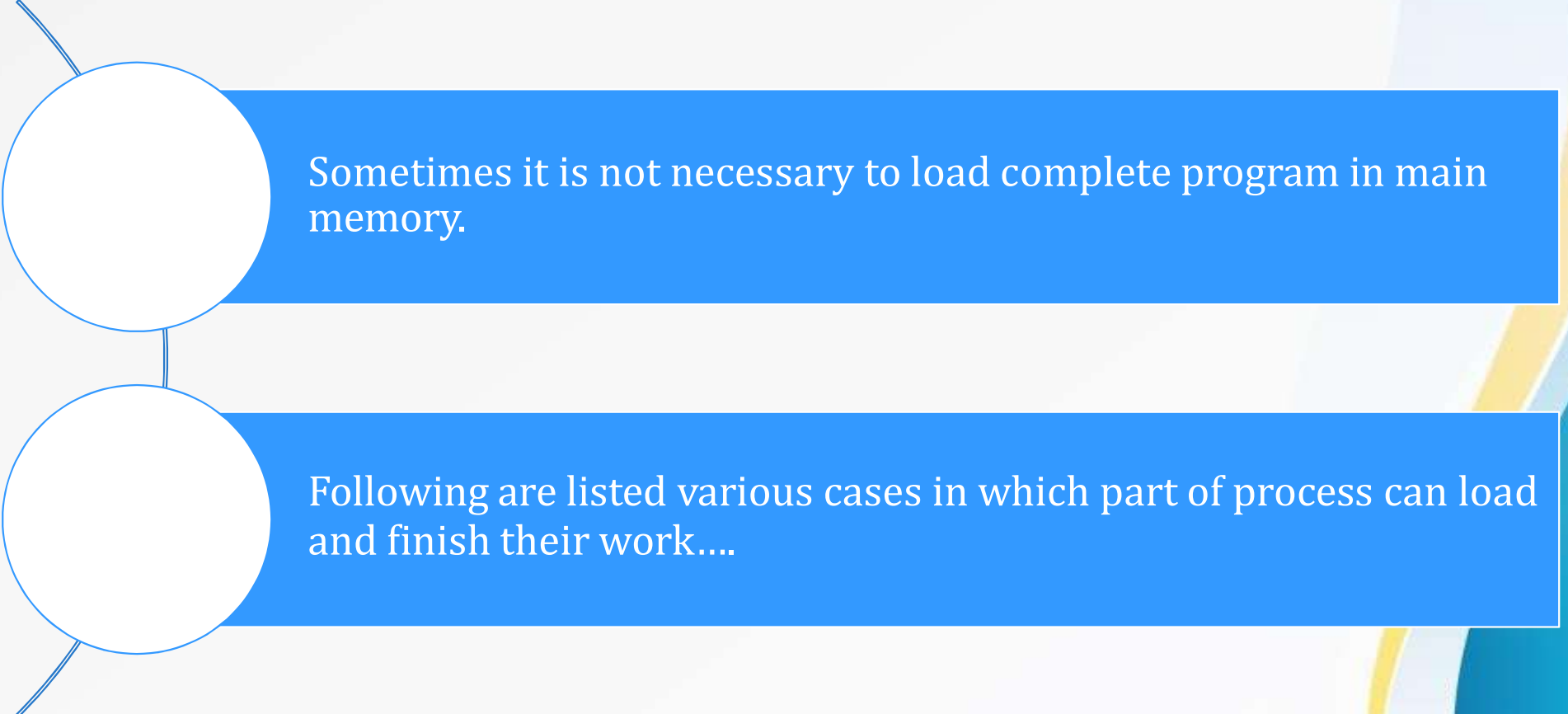
A virtual memory is technique that allows a process to execute even though it is *partially loaded* in main memory.

The diagram consists of three white circles arranged vertically on the left side. Each circle is connected by a thin blue line to a corresponding blue rectangular text box on the right. The circles are empty, and the lines extend slightly from the top and bottom of the circles.

Virtual memory is actually the secondary memory working as primary memory, and helping in executing program that are larger in size than the physical main memory.

The logical memory at user side is separate concept than main memory, that separation is referred as Virtual Memory.

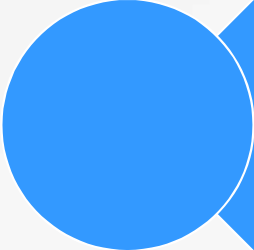
# Virtual Memory



Sometimes it is not necessary to load complete program in main memory.

Following are listed various cases in which part of process can load and finish their work....

## Continued...



*Error handling routines are used only when an error occurred in the data or computation.*

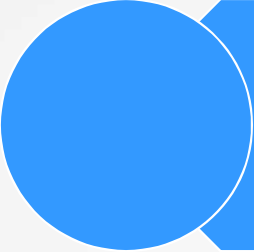


*Some program feature are optional to use*



*Some table require small memory slot, still we have allocated larger space to them*

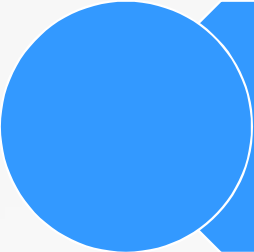
# Virtual Memory: Advantage of Virtual Memory



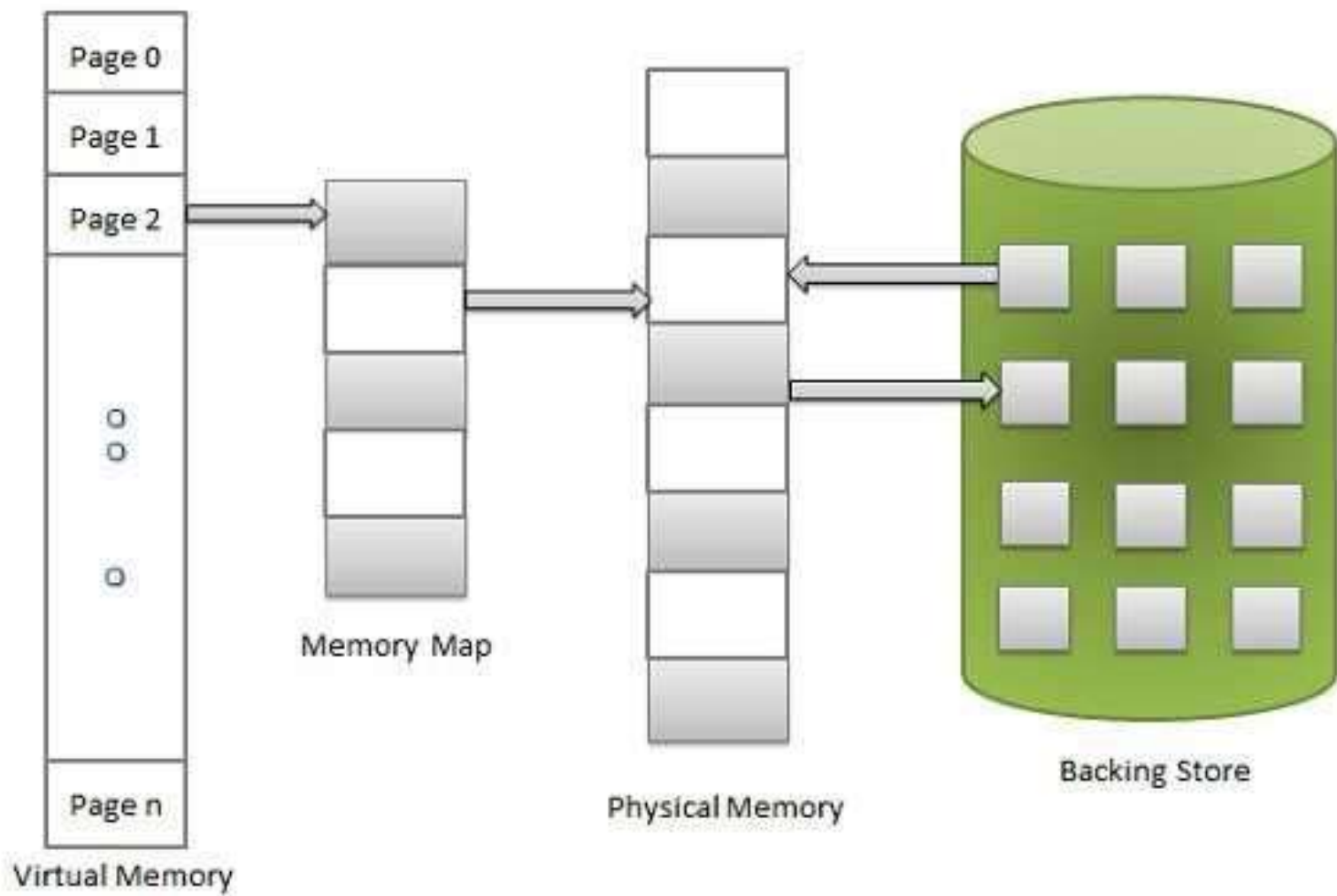
*Less number of I/O* would be needed to load or swap each user program into memory.



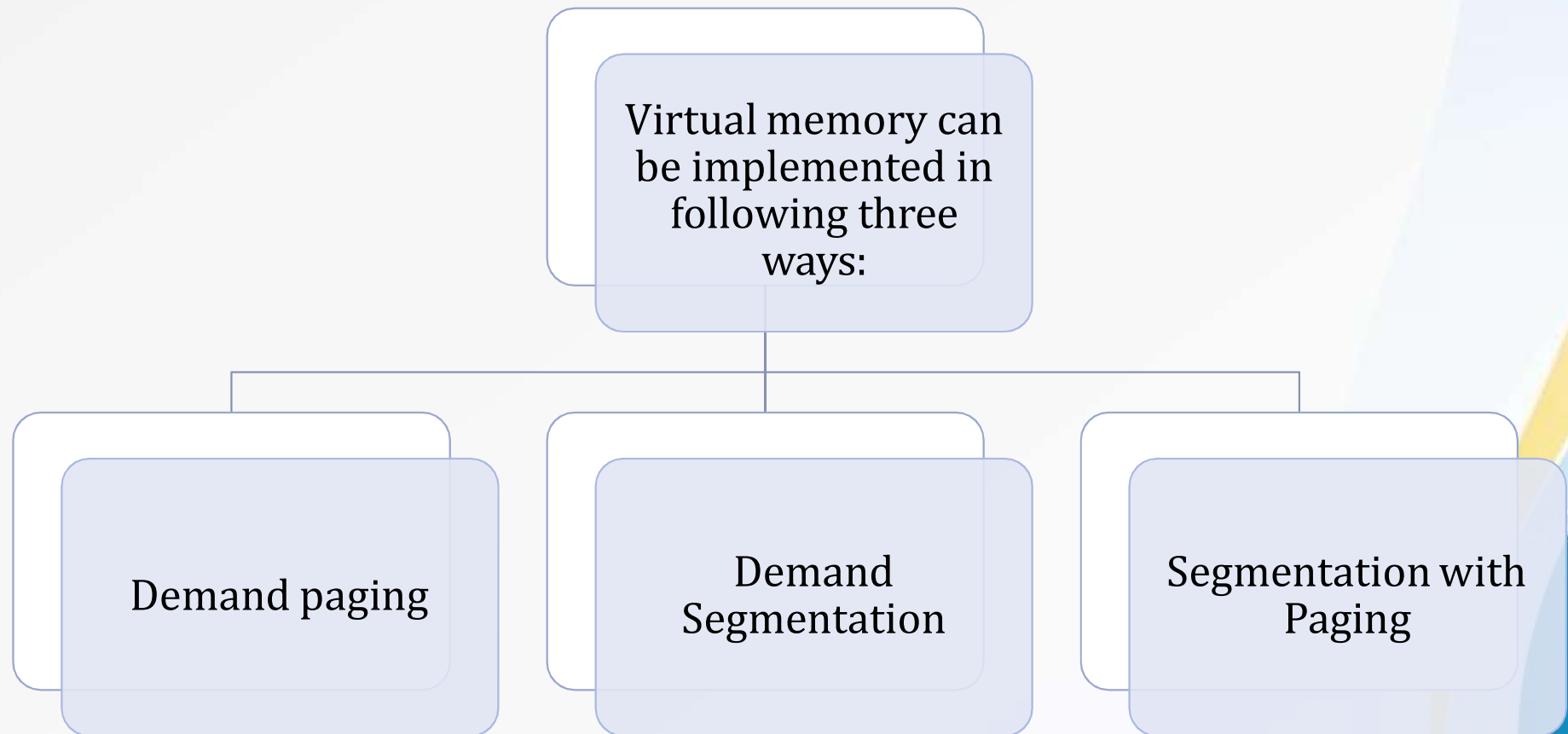
User would be able to write programs for an extremely large virtual address space.



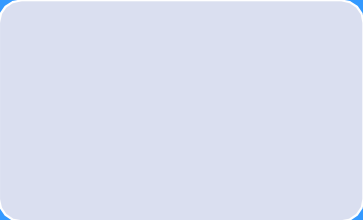
Each user program could take less physical memory, *more programs could be run the same time*, with a corresponding increase in CPU utilization and throughput.



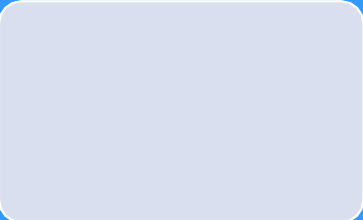
# Virtual Memory: Hardware and control structures




# Demand Paging



A set of technique is provided by the virtual memory to execute the program that is not present in entirely on the memory. *Demand Paging* is a common example of virtual memory system.



Demand paging is similar to a *paging system with swapping* where processes may reside in secondary memory. When we want to execute a process, we swap it into the main memory.



Rather than swapping entire process into memory we use *lazy swapper*. A lazy swapper never swaps page into memory, unless that page will be needed.

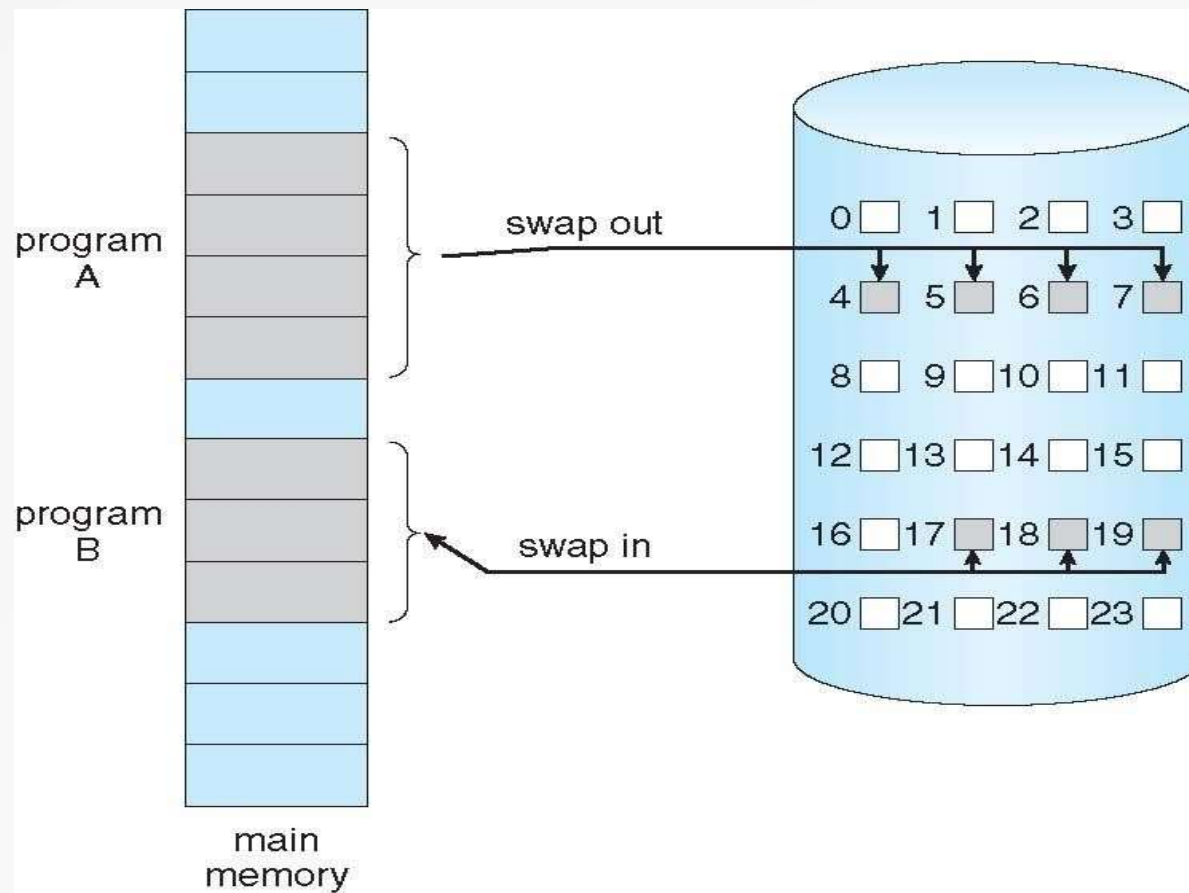


# Demand Paging

If some process is needs to be swap in, then pager(page table handler) brings only those pages which will be used by process.

Thus avoid reading unused pages and decrease swap time and amount of physical memory needed.

# Demand Paging



# Demand Paging: Page Fault

If a process tries to access a page that is not in main memory then it causes page fault.

Pager will generate trap to the OS, and tries to swap in.

Page table includes the valid-invalid bit for each page entry.

If the bit is valid then page is currently available in to the memory.

If it is set to invalid then page is either invalid or not present in main memory.


# Demand Paging: Page Fault

whenever logical address is generated the system operates as follows:


Page table is searched if validity bit is set to valid then physical address is determined.

If the page validity is invalid then OS read that page into free frame from disk.


# Locality of reference




Locality is set of pages that are actively used together. A program generally composed of different localities.



Example, when function is called it defines new locality, and we exit the function the process leaves its locality. Process may return to this locality later.



Locality of reference (often abbreviated simply as locality) is an extremely important concept in systems.



The basic idea is that programs don't just access memory randomly, a program tends to use small amount of data for a while and then move on to another small chunk of data.

# Locality of reference

There are two main kinds of locality:

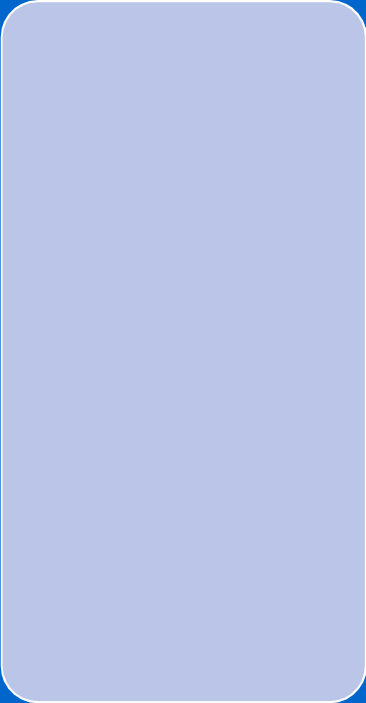
## ***Temporal Locality:***

Temporal locality is the tendency for accesses to a particular data to be clustered in time. e.g. if you used it a little while ago, you're probably going to use it again soon.

## ***Spatial Locality:***

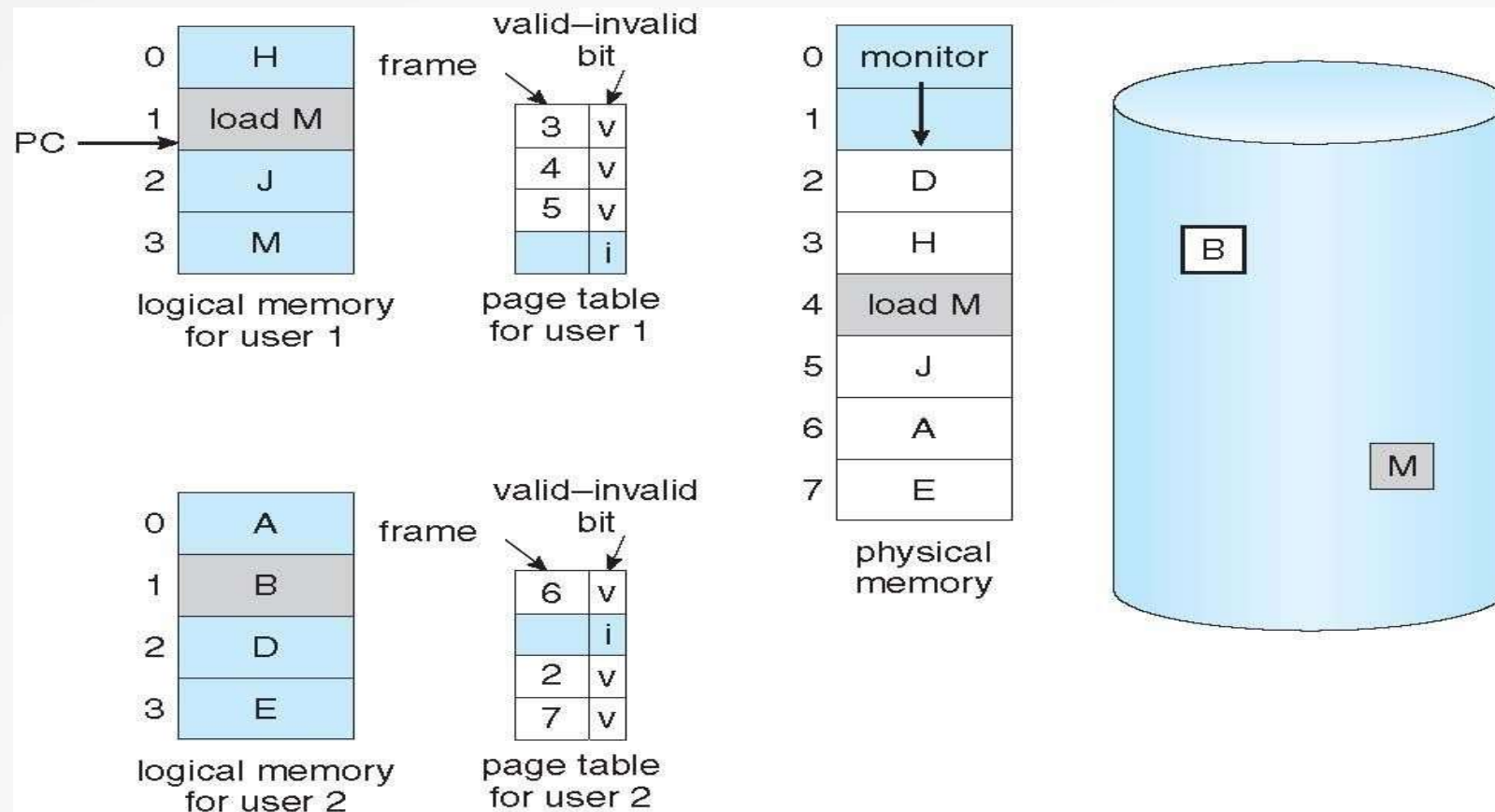
Spatial locality is the tendency for related bits of data to be kept near each other in memory. For example, in an array it is common to access element  $n$ , then element  $n+1$ , then  $n+2$ , etc. Therefore, these elements should be located near each other in memory.

## Dirty page/Dirty bit



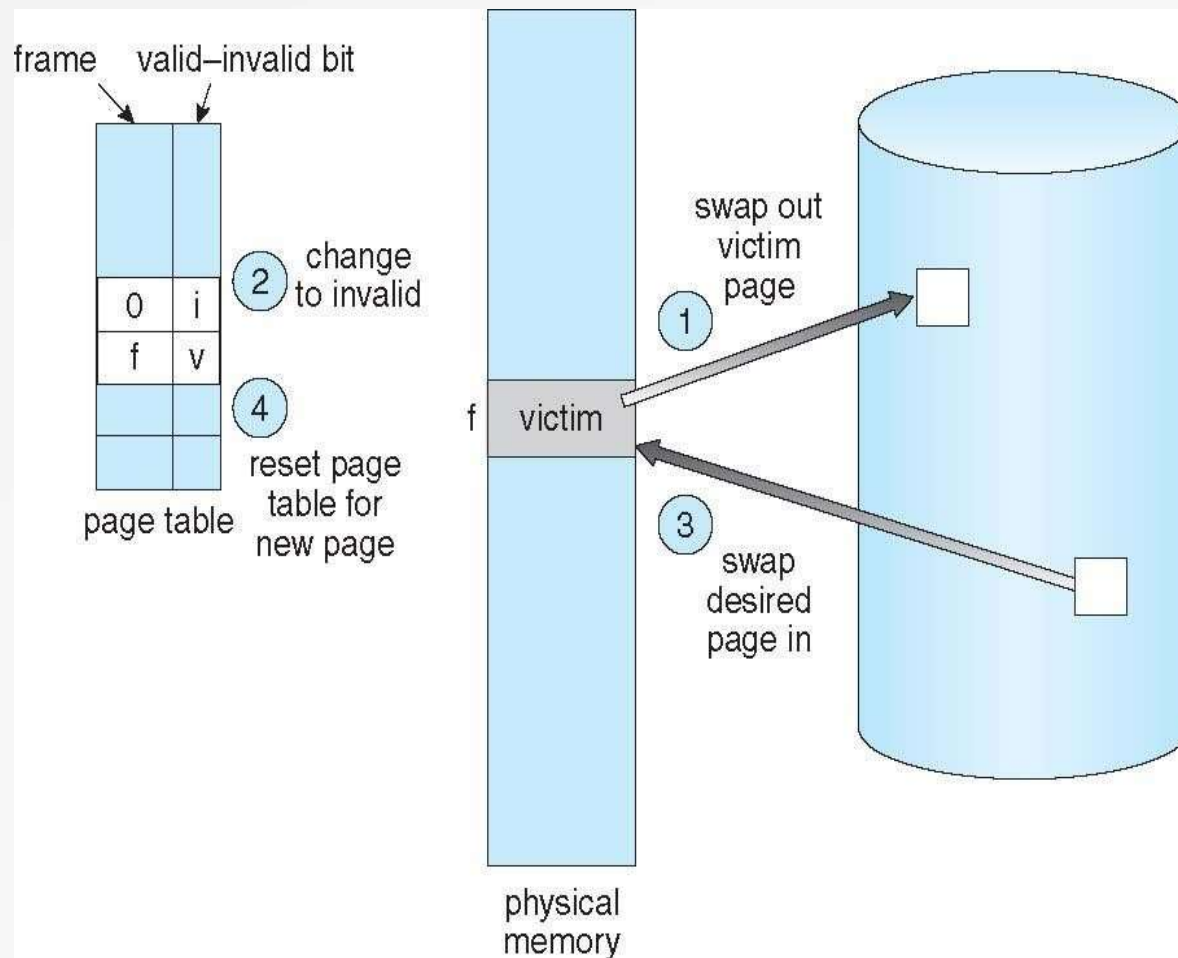
***Dirty bit:*** when a bit is modified by the CPU and not written back to the storage, it is called as a dirty bit. This bit is present in the memory cache or on virtual memory.

# Page Replacement policies: Need





# Page Replacement policies: Basics



1. Find the location of the desired page on disk.
2. Find a free frame:
  - If there is a free frame, use it.
  - If there is no free frame, use a page replacement algorithm to select a **victim frame**
    - Write victim frame to disk if dirty
3. Bring the desired page into the (newly) free frame; update the page and frame tables
4. Continue the process by restarting the instruction that caused the trap

## Page replacement algorithm:

- 1. FIFO- First In First Out
- 2. Optimal Page Replacement
- 3. LRU- Least Recently Used
- 4. Second Chance

# Belady's (or FIFO) Anomaly

# First in First Out (FIFO)

The simplest page replacement algorithm.

When the page must be replaced the *oldest page* is chosen.

one *FIFO queue* is maintained to hold all pages in memory.

Replace the page which is at the top of the queue and add new pages from rear end (tail) of the queue.

# Belady's (or FIFO) Anomaly

	time →											
Access pattern	0 1 2 3 0 1 4 0 1 2 3 4											
Physical memory (3 page frames)	0	0	0	1	2	3	0	0	0	1	4	4
		1	1	2	3	0	1	1	1	4	2	2
			2	3	0	1	4	4	4	2	3	3
	time →											
Access pattern	0 1 2 3 0 1 4 0 1 2 3 4											
Physical memory (4 page frames)	0	0	0	0	0	0	1	2	3	4	0	1
		1	1	1	1	1	2	3	4	0	1	2
			2	2	2	2	3	4	0	1	2	3
				3	3	3	4	0	1	2	3	4

9 page faults!

10 page faults!

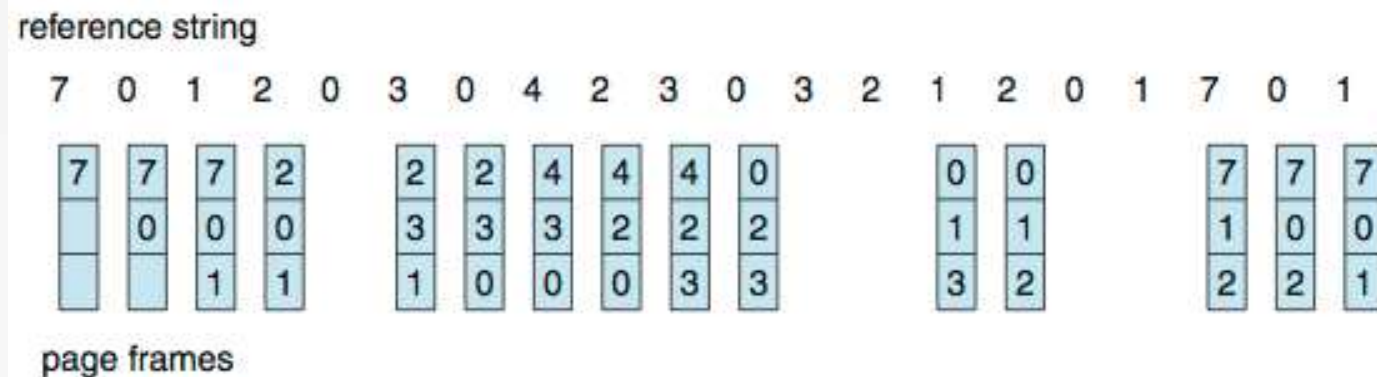
**Certain page reference patterns actually cause more page faults when number of page frames allocated to a process is increased**

## First in First Out (FIFO)

- Reference string: **7,0,1,2,0,3,0,4,2,3,0,3,0,3,2,1,2,0,1,7,0,1**
- 3 frames (3 pages can be in memory at a time per process)

# First in First Out (FIFO)

- Reference string: **7,0,1,2,0,3,0,4,2,3,0,3,2,1,2,0,1,7,0,1**
- 3 frames (3 pages can be in memory at a time per process)



- 15 page faults

## First in First Out (FIFO)

- Reference string: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5
- 3 frames (3 pages can be in memory at a time per process)



# First in First Out (FIFO)

## Advantage

- Very simple.
- Easy to implement.

## Disadvantage

- A page fetched into memory a long time ago may have now fallen out of use.
- This reasoning will often be wrong, because there will often be regions of program or data that are heavily used throughout the life of a program.
- Those pages will be repeatedly paged in and out by the FIFO algorithm.

## Review - FIFO

- If FIFO page replacement is used with four page frames and eight pages, how many page faults will occur with the reference string

0 1 7 2 3 2 7 1 0 3

- if the four frames are initially empty?

Answer: FIFO yields 6 page faults

# Optimal Page Replacement



Its best page replacement policy.

The diagram consists of three white circles arranged vertically on the left side. Each circle is connected to a blue rectangular box on the right by a thin blue line. The circles have small blue lines extending from their top-left and bottom-left edges, giving them a hand-drawn appearance. The blue boxes contain white text describing the Optimal page replacement policy.

The Optimal policy selects for replacement the page that will *not be used for longest period* of time.

Impossible to implement (need to know the future) but serves as a standard to compare with the other algorithms we shall study.

## One more example

- Reference string : 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5 (check next pages)
- 4 frames example

1	<del>1</del>	4
2	2	
3	3	
4	<del>4</del>	5

6 page faults

# Optimal Page Replacement

- Reference string: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5
- 3 frames (3 pages can be in memory at a time per process)

# Example

- Reference string: **7,0,1,2,0,3,0,4,2,3,0,3,0,3,2,1,2,0,1,7,0,1**
- 3 frames (3 pages can be in memory at a time per process)

# Example

reference string

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						7		
	0	0	0		0		4		0		0						0		
		1	1		3		3		3		1						1		

page frames

# Optimal Page Replacement : How can we do better?



Need an approximation of how likely each frame is to be accessed in the future

If we base this on past behavior we got a way to track future behavior

Tracking memory accesses requires hardware support to be efficient



# Optimal Page Replacement

## Advantage:

- Lowest page faults.
- Can Improves performance of system as it reduces number of page faults so requires less swapping.

## Disadvantage:

- Very difficult to implement.

# Not recently used page replacement



Two more bits added in to page table *reference and dirty bits*

- Each page table entry (and TLB entry!) has a
- Referenced bit - page is in used(recent access)
- Dirty / modified bit - set when page is written



Idea: use the information contained in these bits to drive the page replacement algorithm

# Not recently used page replacement

- When a page fault occurs...
- Categorize each page...
  - Class 1:      Referenced = 0   Dirty = 0
  - Class 2:      Referenced = 0   Dirty = 1
  - Class 3:      Referenced = 1   Dirty = 0
  - Class 4:      Referenced = 1   Dirty = 1
- Choose a victim page from class 1, class 2, class 3 and class 4.

## Second Chance (SC)/ Clock Algorithm

A simple modification to FIFO that avoids the problem of throwing out a heavily used page

Use one more bit *reference bit* R. initially the value of R bit is 0, if page is reference then set bit as 1.

## Second Chance (SC)



The diagram illustrates the logic of the Second Chance (SC) algorithm using four circular nodes connected by lines. The first node leads to a blue text box. The second node leads to another blue text box. The third node leads to a third blue text box. The fourth node leads to a final blue text box. The flow is sequential, starting from the top and moving downwards.

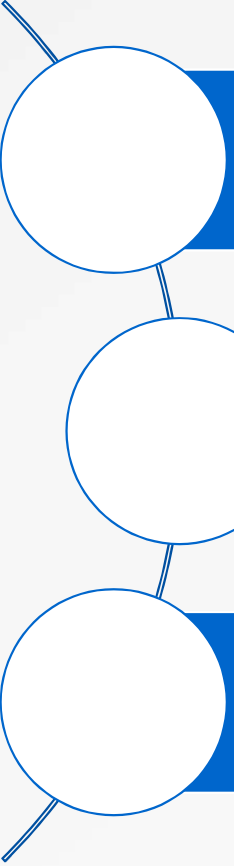
If  $R$  is 0, the page is both *old and unused*, so it is replaced immediately.

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.

Thus, a page that is given a second chance will not be replaced until all other pages have been replaced (or given second chances).

In addition, if a page is used often enough to keep its reference bit set, it will never be replaced.

# Least Recently used (LRU)



It is based on the observation that if pages that have been *heavily used in the last few instructions will probably be heavily used again in the next few.*

Conversely, pages that have not been used for ages will probably remain unused for a long time.

This idea suggests a realizable algorithm: when a page fault occurs, throw out the page that has been *unused for the longest time.*

## One more example

- Reference string: 1, 2, 3, 4, 1, 2, **5**, 1, 2, **3**, **4**, **5**(check past pages)

1	1	1	1	<b>5</b>
2	2	2	2	2
3	<b>5</b>	5	<b>4</b>	4
4	4	<b>3</b>	3	3

8 page faults

## Least Recently used (LRU)

Reference string: **7,0,1,2,0,3,0,4,2,3,0,3,0,3,2,1,2,0,1,7,0,1**  
3 frames (3 pages can be in memory at a time per process)



## Least Recently used (LRU)

reference string

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		4	4	4	0		1		1		1
	0	0	0		0		0	0	3	3		3		0		0
		1	1		3		3	2	2	2		2		2		7

page frames

**Total Page Faults = 12**

# Least Recently used (LRU)



## Implementation

- Every time a page is accessed, record a *timestamp* of the access time
- When choosing a page to evict, scan over all pages and throw out page with *oldest timestamp*

## Least Recently used (LRU)

Time		0	1	2	3	4	5	6	7	8	9	10
Requests			<i>c</i>	<i>a</i>	<i>d</i>	<i>b</i>	<i>e</i>	<i>b</i>	<i>a</i>	<i>b</i>	<i>c</i>	<i>d</i>
Page Frames	0	<i>a</i>	<i>a</i>	<i>a</i>	<i>a</i>	<i>a</i>	<i>a</i>	<i>a</i>	<i>a</i>	<i>a</i>	<i>a</i>	<i>a</i>
	1	<i>b</i>	<i>b</i>	<i>b</i>	<i>b</i>	<i>b</i>	<i>b</i>	<i>b</i>	<i>b</i>	<i>b</i>	<i>b</i>	<i>b</i>
	2	<i>c</i>	<i>c</i>	<i>c</i>	<i>c</i>	<i>c</i>	<i>e</i>	<i>e</i>	<i>e</i>	<i>e</i>	<i>e</i>	<i>d</i>
	3	<i>d</i>	<i>d</i>	<i>d</i>	<i>d</i>	<i>d</i>	<i>d</i>	<i>d</i>	<i>d</i>	<i>d</i>	<i>c</i>	<i>c</i>
Faults							•				•	•
Time page last used			<i>a</i> = 2 <i>b</i> = 4 <i>c</i> = 1 <i>d</i> = 3					<i>a</i> = 7 <i>b</i> = 8 <i>e</i> = 5 <i>d</i> = 3		<i>a</i> = 7 <i>b</i> = 8 <i>e</i> = 5 <i>c</i> = 9		

**Total Page Faults = 7**

## Least Recently used (LRU)

### Implementation

```
graph LR; A[Implementation] --- B[Counter implementation]; A --- C[Stack implementation]; A --- D[Hardware matrix implementation];
```

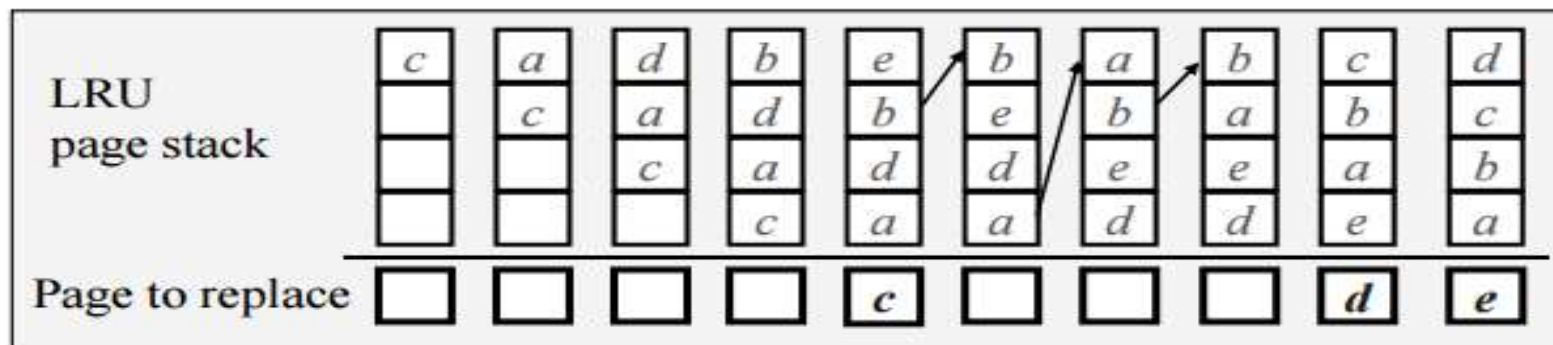
Counter implementation

Stack implementation

Hardware matrix  
implementation

## Least Recently used (LRU) : Stack implementation

Time	0	1	2	3	4	5	6	7	8	9	10
Requests		<i>c</i>	<i>a</i>	<i>d</i>	<i>b</i>	<i>e</i>	<i>b</i>	<i>a</i>	<i>b</i>	<i>c</i>	<i>d</i>
Page Frames	0	<i>a</i>	<i>a</i>	<i>a</i>	<i>a</i>	<i>a</i>	<i>a</i>	<i>a</i>	<i>a</i>	<i>a</i>	<i>a</i>
	1	<i>b</i>	<i>b</i>	<i>b</i>	<i>b</i>	<i>b</i>	<i>b</i>	<i>b</i>	<i>b</i>	<i>b</i>	<i>b</i>
	2	<i>c</i>	<i>c</i>	<i>c</i>	<i>c</i>	<i>e</i>	<i>e</i>	<i>e</i>	<i>e</i>	<i>e</i>	<i>d</i>
	3	<i>d</i>	<i>d</i>	<i>d</i>	<i>d</i>	<i>d</i>	<i>d</i>	<i>d</i>	<i>d</i>	<i>c</i>	<i>c</i>
Faults						•				•	•



Add new page in top and remove form bottom.

# Least Recently used (LRU)

## Advantage

- Feasible to implement.
- Not as simple as FIFO but not that much complex to implement .

## Disadvantage

- It requires additional data structure.