

Memory Management

Prof. P.C.P. Bhatt



What is a Von-Neumann Architecture ?

- *Von Neumann* computing requires a program to reside in main memory to run.
- *Motivation* The main motivation for management of main memory comes from the fact we need to provide support for several programs which are memory resident as in a *multiprogramming environment*.



Main Memory Management - 1

Issues that prompt main memory management:

- *Allocation* : processes must be allocated *space* in the *main memory*.
- *Swapping, Fragmentation and Compaction* : If a program terminates or is moved out, it creates a *hole* in the *main memory*. The main memory is *fragmented* and needs to be *compacted* for organized allocation..



Main Memory Management - 2

➤ *Virtual Memory :*

VM support requires an address translation mechanism to map a *logical address* to the *physical address* to access the desired data or instruction.

➤ *IO Support :*

Most *block oriented devices* are recognized as specialized files. Their *buffers* need to be managed within main memory alongside other processes.



Main Memory Management - 3

- *Garbage Collection* : The area released by a process is usually not accounted for immediately by the processor – *its garbage!!*.
- *Compaction* or *garbage collection* is responsibility of the OS.
- *Protection* : checking for *illegal access* of data from another process's memory area.

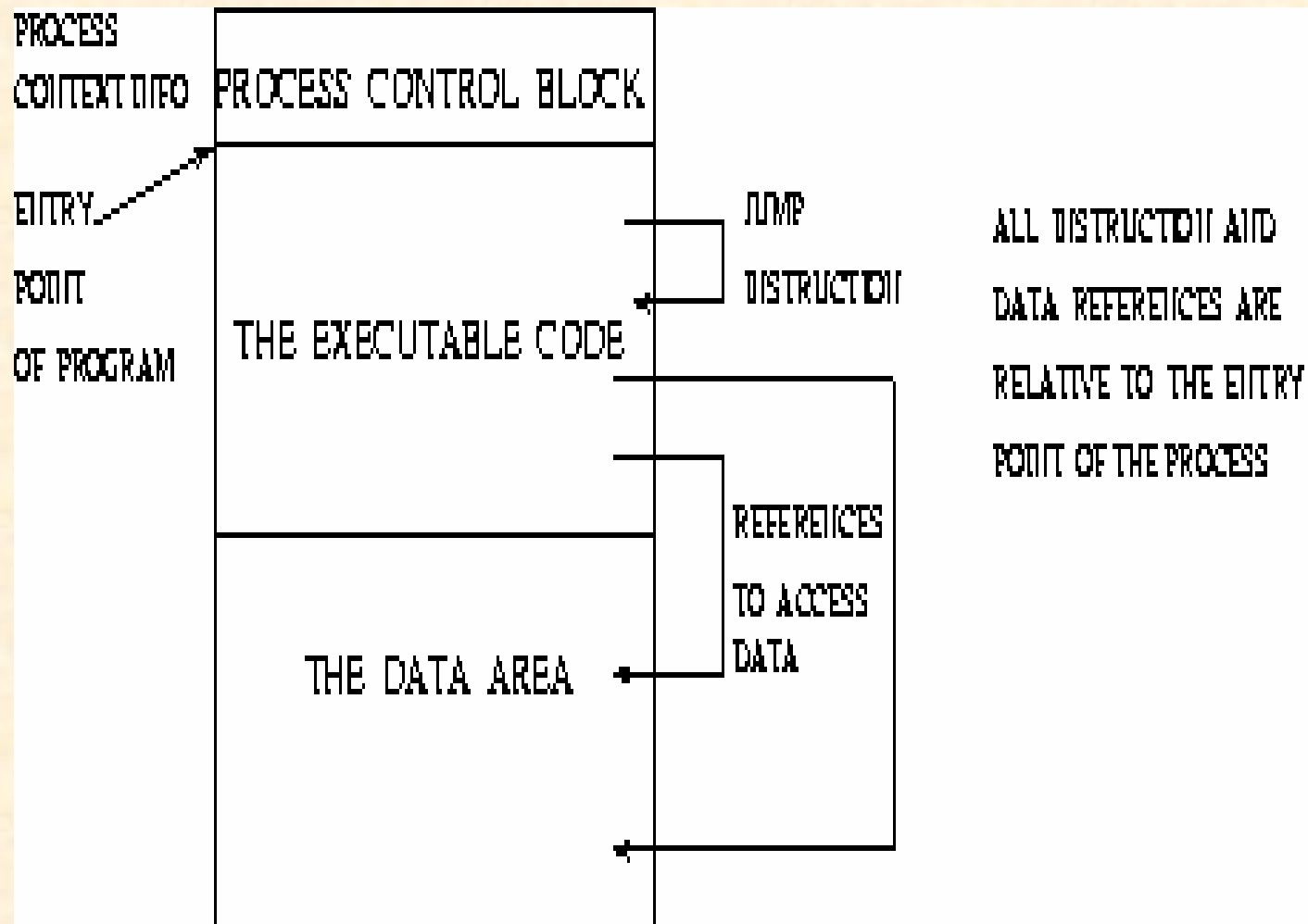


Memory Relocation Concept - 1

Why do we need relocatable Processes ?

- Consider a *linear map* (1-D view) of main memory.
- *Program Counter* is set to the *absolute address* of the first instruction of the program.
- *Data* can also fetched if we know its absolute address.
- In case that part of memory is currently in use then this program can not be run.
- Note if we have program instructions then we should be able to execute the program from starting at any location.

The Relocation Concept - 2





Memory Relocation Concept - 3

- With this flexibility, we can allocate *any area in the memory* to load this process.
- Note that this is most useful when processes *move in and out of main memory* – recall that a hole created by a process at the time of moving out of the main memory will not be available when it is brought into main memory again.

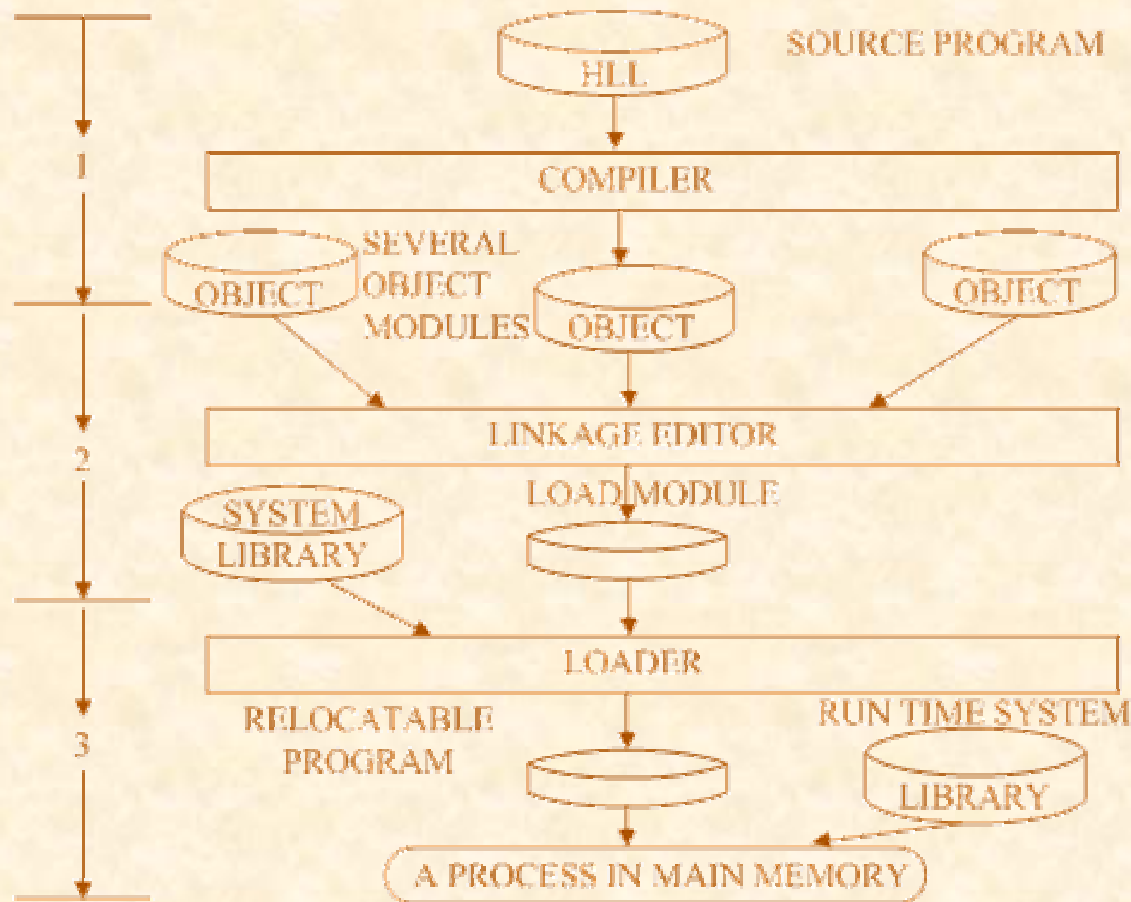


Compiler Generated Bindings

- The advantage of *relocation* can be seen in the light of *binding of addresses to variables* in a program.
- For a variable x in a program P , a *fixed address* allocation for x will mean that P can be run only when x is allocated the same memory again.

Linking and Loading Concepts - 1

Following the creation of a high level language (HLL) source program, there are three stages of processing before we can get a process as shown in figure below.





Linking and Loading Concepts - 2

- *Stage1*: In the first stage the *HLL source program* is compiled and an object code is produced. Technically, depending upon the program, this *object code may by itself* be sufficient to generate a relocatable process. However, many programs are compiled in *parts*, so this object code may have to link up with other object modules. At this stage the compiler may also *insert stub* at points where run time library modules may be linked.
- *Stage2*: All object modules which have sufficient linking information (generated by the compiler) for *static linking* are taken up for linking. The linking editor generates a *relocatable code*. At this stage, however, we still *do not replace* the *stubs* placed by compilers for a run time library link up.



Linking and Loading Concepts - 3

- *Stage 3* : The final step is to *arrange to make substitution for the stubs* with run time library code which is a *relocatable code*.

When all the three *stages are completed* we have an executable. When this executable is resident in the main memory it is a *runable* process.



The First Fit Policy of Memory Allocation

- We are following *FCFS (process management)* and *First Fit (memory allocation)* policies.
- First Fit main memory allocation policy is *very easy to implement* and is *fast in execution*.
- First Fit policy may leave *many small holes*.



Memory Allocation Policies

- Best Fit Policy *scans all* available *holes* and chooses the one with a *size closest* to the requirement.
- It requires a *scan of the whole memory* and is *slow*.
- Next Fit has a *search pointer* continues from where the previous search ended.
- Worst Fit method allocates the *largest hole*.
- First Fit and Next Fit are *fastest* and are hence *preferred* methods.
- Worst Fit is the *poorest* of all the four methods.
- To compare these policies, we shall examine the effect of using various policies on a given set of data next.



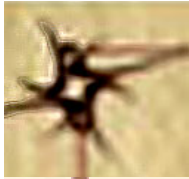
The Given Data For Policy Comparison

➤ The given Data:

- Memory available 20 units
- OS resides in 6 units
- User processes share 14 units.

➤ The user process data:

	P1	P2	P3	P4	P5	P6
Time of arrival	0	0	0	0	10	15
Processing time required	8	5	20	12	10	5
Memory required	3 units	7 units	2 units	4 units	2 units	2 units



FCFS Policy

Statement: *“Jobs are processed in the order they arrive”*.

FCFS Memory Allocation

Time units	Programs in Main memory	Programs on disk	Holes with sizes	Figure 4.3	Comments
0	P1, P2, P3	P4	H1=2	(a)	P4 requires more space than H1
5	P1, P4, P3		H1=2; H2=3	(b)	P2 is Finished P4 is loaded Hole H2 is created
8	P4, P3		H1=2; H2=3; H3=3	(c)	New hole created
10	P4, P3	P5			P5 arrives
10+	P5, P4, P3		H1=2; H2=3 H3=1	(d)	P5 is allocated P1's space
15	P5, P4, P3	P6	H1=2; H2=3; H3=1		P6 has arrived
15+	P5, P4, P6, P3		H1=2; H2=1; H3=1	(e)	P6 is allocated

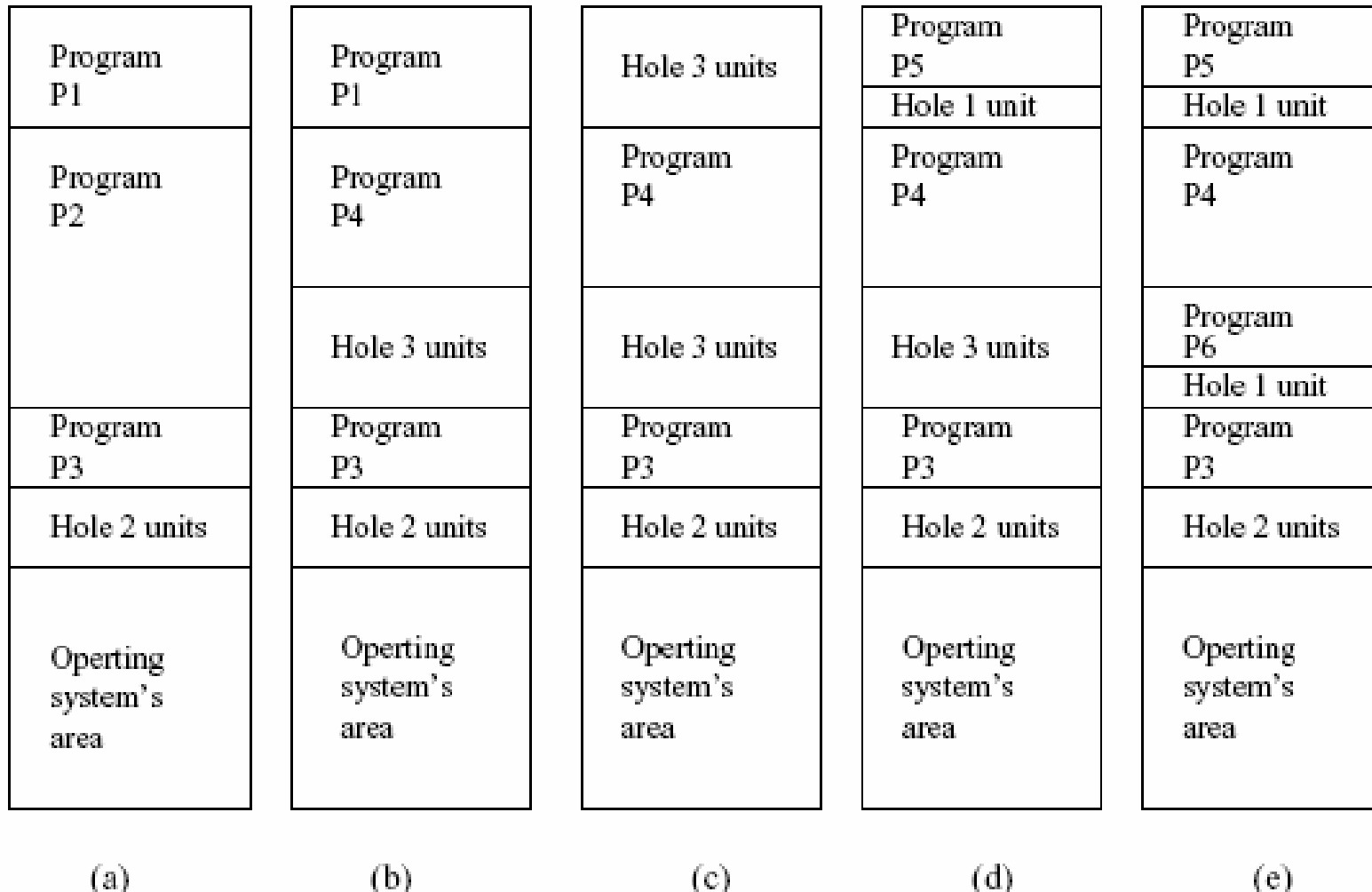


The First Fit Policy

- **Statement:** *“Jobs are processed always from one end and find the first block of free space which is large enough to accommodate the incoming process”.*
- **Given Data:**

	P1	P2	P3	P4	P5	P6
Time of arrival	0	0	0	0	10	15
Processing time required	8	5	20	12	10	5
Memory required	3 units	7 units	2 units	4 units	2 units	2 units

The First Fit Policy of Memory Allocation



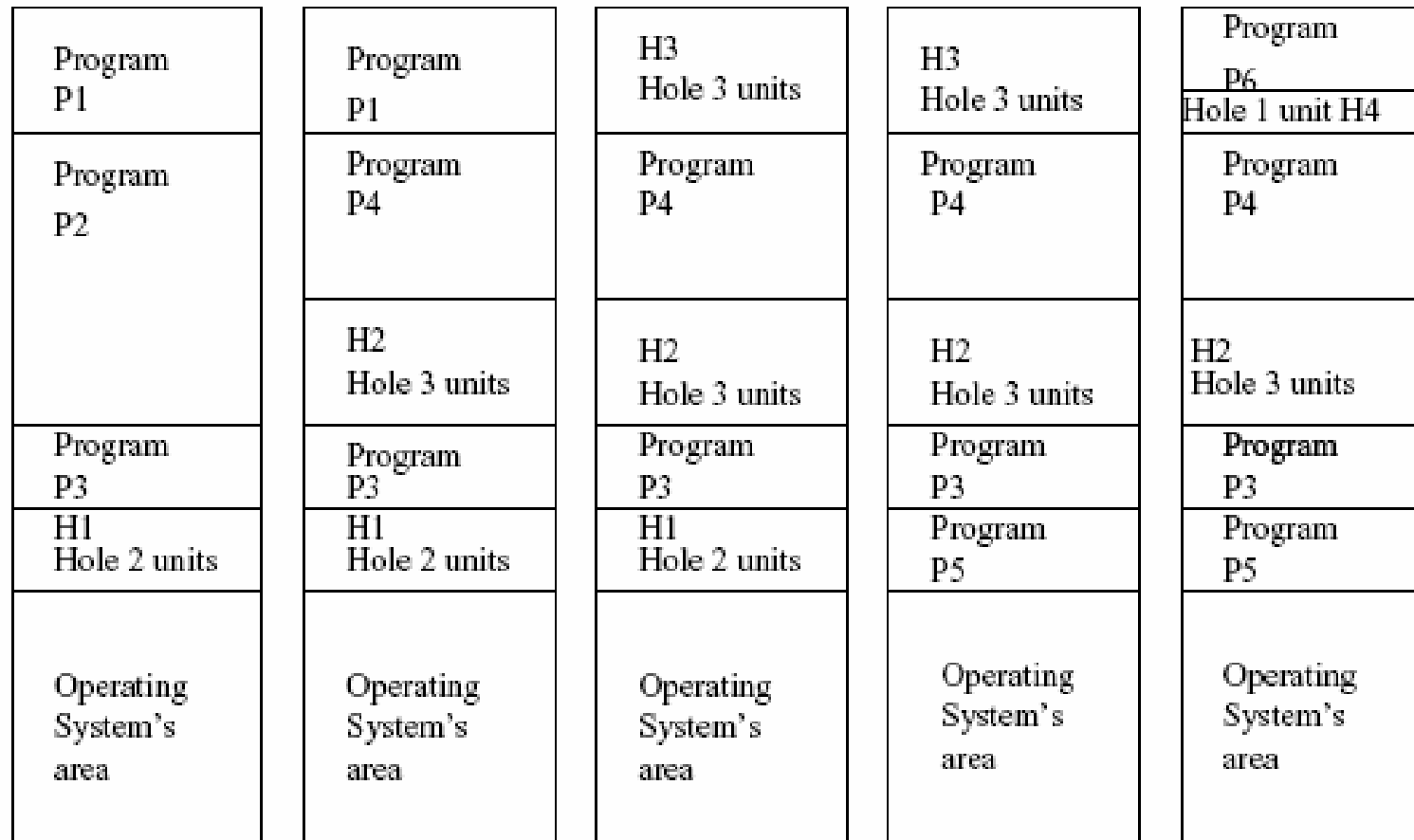


The Best Fit Policy

- Statement: *“Jobs are selected after scanning the main memory for all the available holes and having information about all the holes in the memory, the job which is closest to the size of the requirement of the process will be processed”.*
- Given Data:

	P1	P2	P3	P4	P5	P6
Time of arrival	0	0	0	0	10	15
Processing time required	8	5	20	12	10	5
Memory required	3 units	7 units	2 units	4 units	2 units	2 units

The Best Fit Policy of Memory Allocation



(a)

(b)

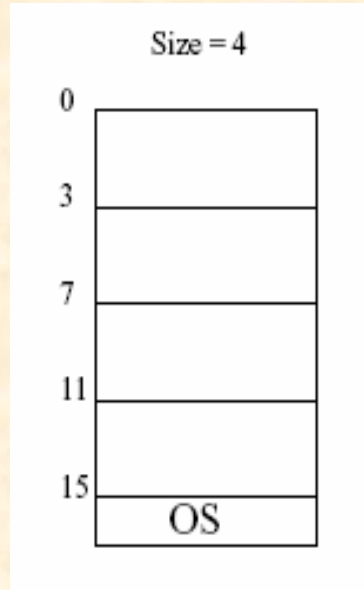
(c)

(d)

(e)

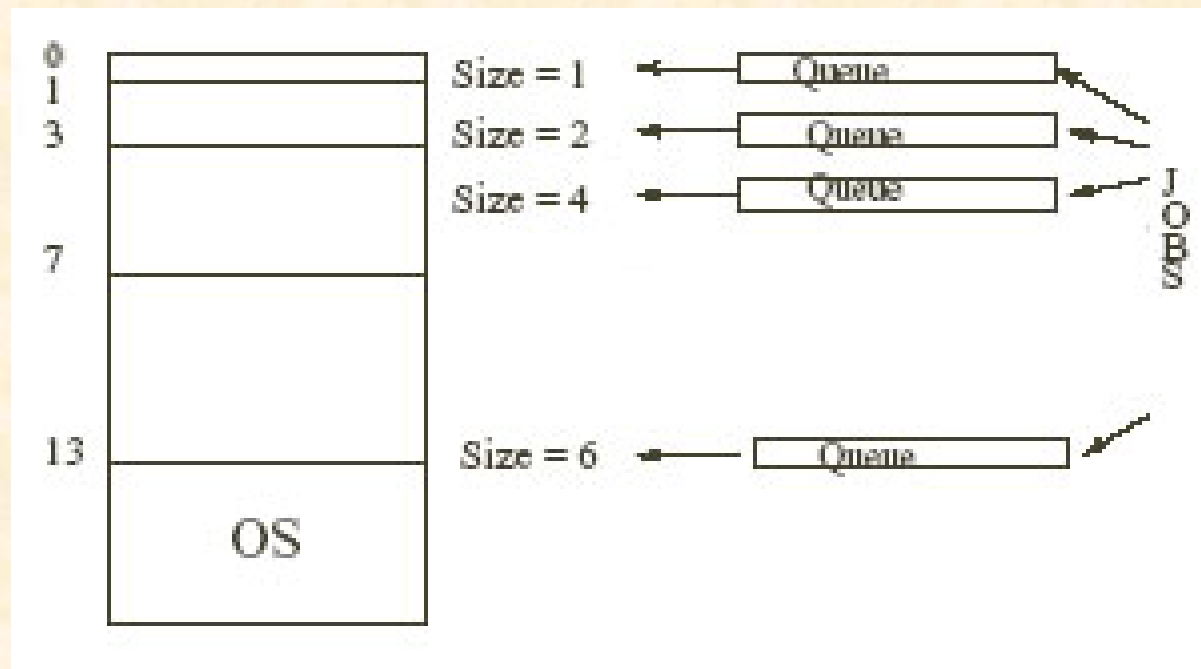
Fixed and Variable Partition - 1

- *Fixed Partition* : Memory is divided into chunks. For example, 4K/8K/16K Bytes. All of these are same size.
- *Allocation* : If a certain chunk can hold the program/data, then the allocation. If a chunk can not hold program/data then multiple chunks are allocated to accommodate the program/data.



Fixed and Variable Partition - 2

- *Variable Partition*: Memory is divided into chunks of various sizes. For Example there could be chunks of 8K, some may be 16 K or even more.
- *Allocation*: The program/Data are allocated to the chunks that can accommodate the incoming program/Data.





Buddy System - 1

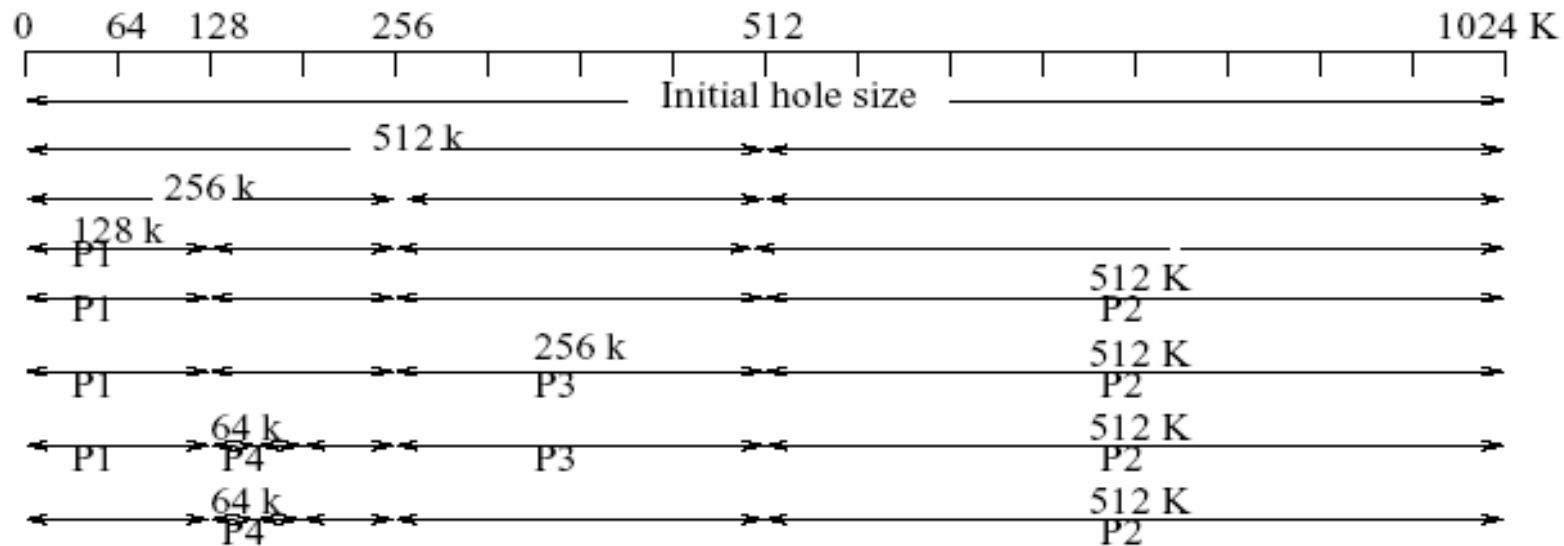
The *buddy system* of partitioning relies on the fact that space allocations can be conveniently handled in sizes of power of 2.

There are two ways in which the buddy system allocates space.

- Suppose we have a hole which is the closest power of two. In that case, that hole is used for allocation.
- In case we do not have that situation then we look for the next power of 2 hole size, split it in two equal halves and allocate one of these.

Because we always split the holes in two equal sizes, the two are “*buddies*”. Hence, the name buddy system.

Buddy System - 2



We assume that initially we have a space of $1024 K$. We also assume that processes arrive and are allocated following a time sequence as shown in figure.

In the figure we assume the requirements as $(P1:80K);(P2:312K);(P3:164 K); (P4:38 K)$. These processes arrive in the order of their index and $P1$ and $P3$ finish at the same time



Buddy System - 3

- With *1024 K* or (*1M*) storage space we split it into buddies of *512 K*, splitting one of them to two *256 K* buddies and so on till we get the right size. Also, we assume scan of memory from the beginning. We always use the first hole which accommodates the process.
- Otherwise, we split the next sized hole into buddies. Note that the buddy system begins search for a hole as if we had a *number of holes of variable sizes*. In fact, it turns into a dynamic partitioning scheme if we do not find the best-fit hole initially.
- The buddy system has the advantage that it minimizes the *internal fragmentation*. In practice, some Linux flavors use it. Earlier systems offered by Burroughs used this scheme.

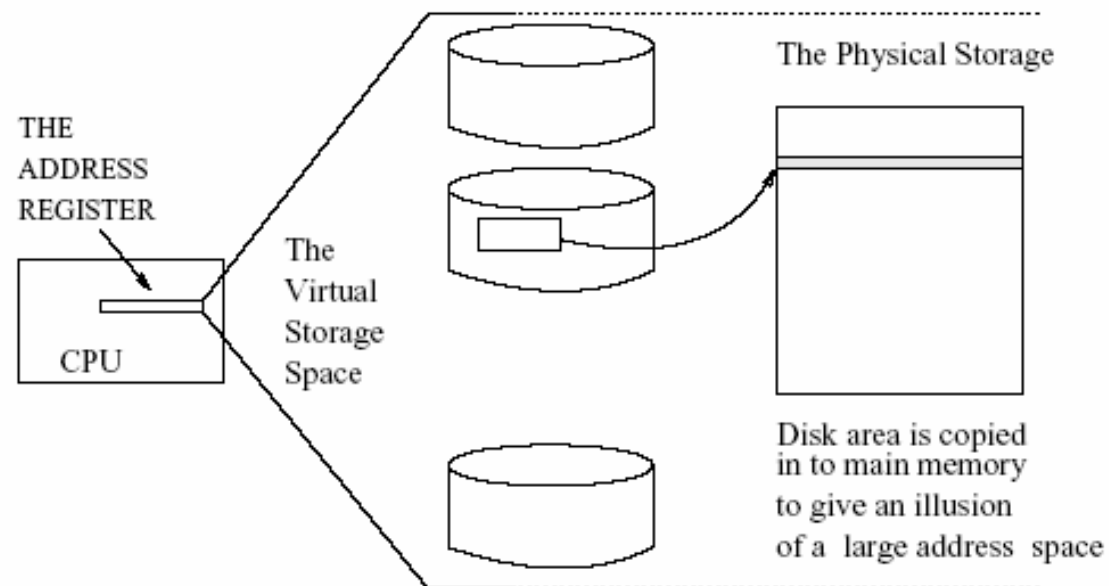


Concept of Virtual Storage - 1

- The *directly addressable main memory* is limited and is quite small in comparison to the *logical addressable space*.
- The actual size of main memory is referred as the *physical memory*. The logical addressable space is referred to as *virtual memory*.
- The concept of *virtual storage* is to give an impression of a large addressable storage space without necessarily having a *large primary memory*
- The basic idea is to offer a *seamless extension* of primary memory into the space within the secondary memory. The address register *generate addresses* for a space much larger than the primary memory.
- The notion of virtual memory is a bit of an illusion. The OS *supports and makes* this illusion possible.

Virtual Storage - 2

- The OS creates this illusion by *copying chunks of disk memory* into the *main memory* as shown in figure. In other words, the processor is fooled into believing that it is *accessing a large addressable space*. Hence, the name virtual storage space. The disk area may map to the virtual space requirements and even beyond.





Virtual Memory: Paging-1

- Once we have addressable segments in the *secondary memory*-we need to bring it within the *main memory* for physical access for process. Often mechanism of *paging* helps.
- Paging is like *reading a book*. At any time we do not need all pages-except the ones we are reading. The analogy suggest that pages we are reading are in the main memory and the rest can be in the secondary memory.



Virtual Memory: Paging-2

- The primary idea is to always keep focus on that area of memory from which instructions are executed. Once that area is identified – it is loaded into the primary memory into the fixed size pages.
- To enable such a loading, page sizes have to be defined and observed for both primary as well as secondary memory.
- Paging support *locality of reference* for *efficient access*
- For instance we have location of reference during execution of *while or for loop* or a call to a procedure.



Mapping the Pages - 1

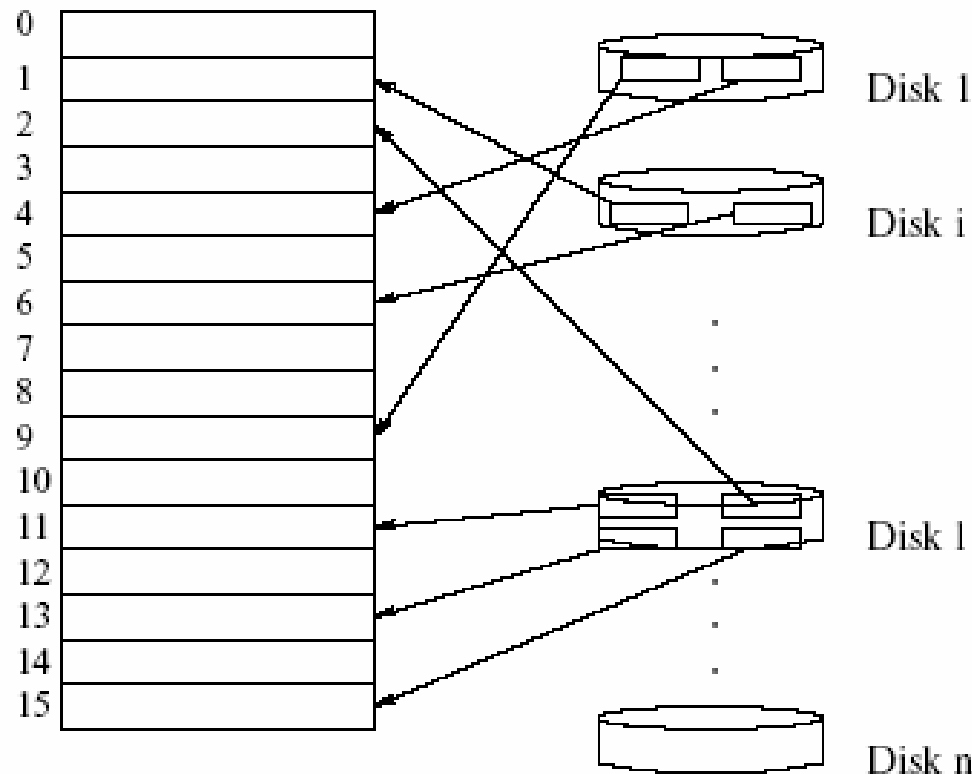
- Paging stipulates that main memory is partitioned into *frames* of sufficiently small sizes.
- Also, we require that the virtual space is divided into *pages* of the same size as the frames.
- This equality facilitates movement of a *page* from anywhere in the virtual space (on disks) to a *frame* anywhere in the physical memory.
- The capability to map “*any page*” to “*any frame*” gives a lot of flexibility of operation.



Mapping the Pages - 2

- Division of main memory into frames is *like fixed partitioning*. So keeping the frame size small helps to keep the internal fragmentation small.
- Paging supports *multi-programming*. In general there can be many processes in main memory, each with a different number of pages. To that extent, paging is like *dynamic variable partitioning*.

Paging: Implementation



Process 1 may be in Disk 1 occupying 20 pages. Pages 2 and 3 at virtual address 20 and 21 are mapped to main memory location 9 and 4.

Process 6 may be in Disk i Occupying 30 pages. Pages 17 and 18 at virtual address 39 and 40 are mapped to main memory location 1 and 6.

Process 29 may be in Disk 1 occupying 4 pages, 77 to 80 are mapped to main memory.

The OS maintains a list of free pages in main memory for allocating a new free page.

		PAGE TABLES FOR PROCESSES P1, .. P6, .. P29 ..										
LOGICAL		20	21	.	39	40	.	77	78	79	80	
PHYSICAL		9	4	.	1	6	.	11	2	13	15	
		← P1 →			← P6 →			← P29 →				

Paging: Replacement - 1

- When a page is no longer needed it can be replaced.
- Consider an example shown in figure process P29 has all its pages present in main memory.
- Process P6 does not have all its pages in main memory. If a page is present we record 1 against its entry. The OS also records if a page has been referenced to read or to write. In both these cases a reference is recorded.

Page tables for processes P1, .. P6, .. P29 ...

	← P1 →	← P6 →	← P29 →
Logical	20 21	. . 39 40	. . 77 78 79 80 .
Physical	9 4 .	1 6	. 11 2 13 15
Present	0 1 1 0	0 0 1 1 0	1 1 1 1 1 1 1
Referenced	1 1	0 1	1 0 1 0
Modified	1 0	0 0	0 0 1 0
Protection	rw- rw-		r-- r--

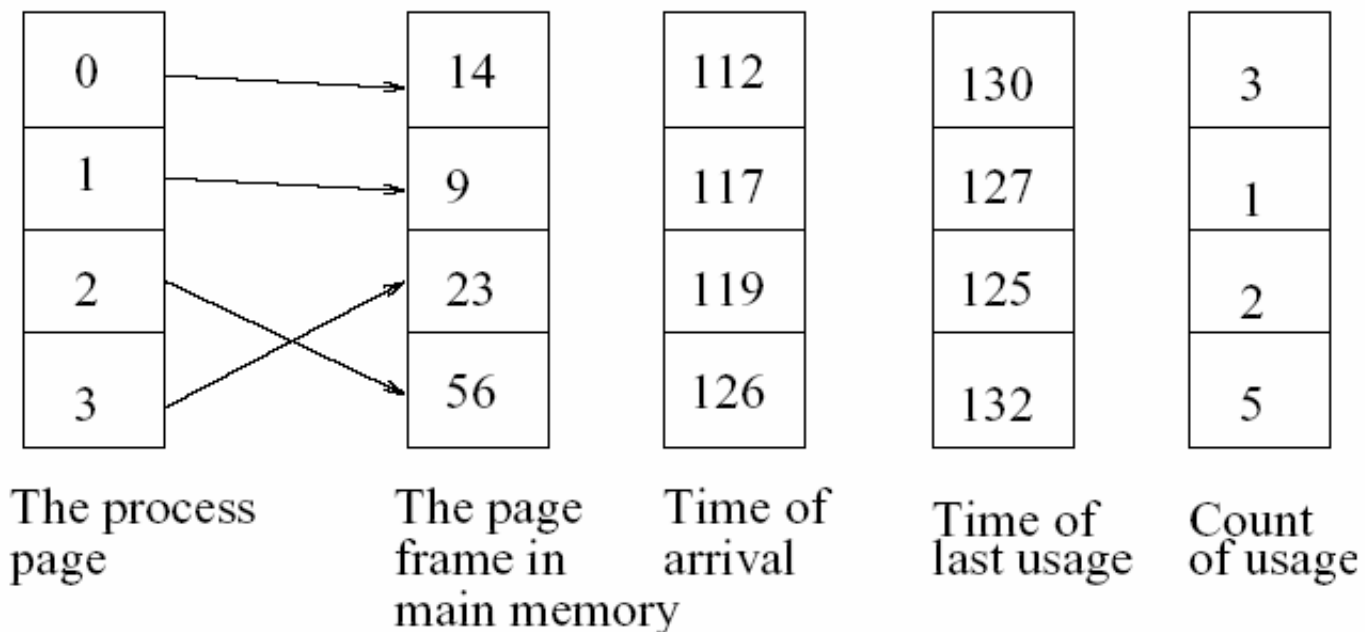


Paging: Replacement - 2

- If a page frame is written into, then *a modified bit* is set. In our example, frames 4, 9, 40, 77, 79 have been referenced and page frames 9 and 13 have been modified.
- Sometimes OS may also have some information about protection using *rwe* information. If a reference is made to a certain virtual address and its corresponding page is not present in main memory, then we say a *page fault* has occurred.
- Typically, a page fault is followed by moving in a page. However, this may require that we move a page out to create a space for it. Usually this is done by using an appropriate *page replacement policy* to ensure that the throughput of a system does not suffer. We shall next see how a page replacement policy can affect performance of a system.

Page Replacement Policies - 1

- Towards understanding page replacement policies we shall consider a simple example of a process P which gets an allocation of four pages to execute.
- Further, we assume that the OS collects some information (depicted in figure) about the use of these pages as this process progresses in execution.





Page Replacement Policies - 2

Let us examine the information depicted in figure in some detail to determine how this may help in evolving a page replacement policy.

Note that we have the following information available about P .

- *The time of arrival of each page:* We assume that the process began at some time with value of time unit 100. During its course of progression we now have pages that have been loaded at times 112, 117, 119, and 120.
- *The time of last usage:* This indicates when was a certain page last used. This entirely depends upon which part of the process P is being executed at any time
- *The frequency of use:* We have also maintained the frequency of use over some fixed interval of time T in the immediate past. This clearly depends upon the nature of control flow in process P .



Page Replacement Policies - 3

Based on the previous pieces of information if we now assume that at time unit 135 the process P experiences a page-fault, what should be done. Based on the choice of the policy and the data collected for P , we shall be able to decide which page to swap out to bring in a new page.

- *FIFO policy*: This policy simply removes pages in the order they arrived in the main memory. Using this policy we simply remove a page based on the time of its arrival in the memory. Clearly, use of this policy would suggest that we swap page located at 14 as it arrived in the memory earliest.



Page Replacement Policies - 4

➤ *LRU policy*: LRU expands to least recently used. This policy suggests that we remove a page whose last usage is farthest from current time. Note that the current time is 135 and the least recently used page is the page located at 23. It was used last at time unit 125 and every other page is more recently used. So page 23 is the least recently used page and so it should be swapped if LRU replacement policy is employed.

➤ *NFU policy*: NFU expands to not frequently used. This policy suggests to use the criterion of the count of usage of page over the interval T . Note that process P has not made use of page located at 9. Other pages have a count of usage like 2, 3 or even 5 times. So the basic argument is that these pages may still be needed as compared to the page at 9. So page 9 should be swapped.



Page Hit and Page Miss

When we find that a page frame reference is in the main memory then we have a *page hit* and when page fault occurs we say we have a *page miss*.

Now let us consider the following two cases when we have 50% and 80% page hits. We shall compute the average time to access.

Case 1 : With 50% page hits the average access time is

$$((10+40) \times 0.5) + (10+190) \times 0.5 = 125 \text{ time units}$$

Case 2 : With 80% page hits the average access time is

$$((10+40) \times 0.8) + (10+190) \times 0.2 = 80 \text{ time units}$$

Clearly, the *case 2 is better*. The OS designers attempt to offer a page replacement policy which will try to minimize the page miss. It is not unusual to be able to achieve over 90% page hits when the application profile is very well known.



Thrashing - 1

- The page replacement policy may result in a situation such that two pages alternatively move in and out of the resident set. Clearly, this is an undesirable situation.
- Note that because pages are moved between main memory and disk, this has an enormous overhead. This can adversely affect the throughput of a system.
- The drop in the level of system throughput resulting from frequent page replacement is called *thrashing*.

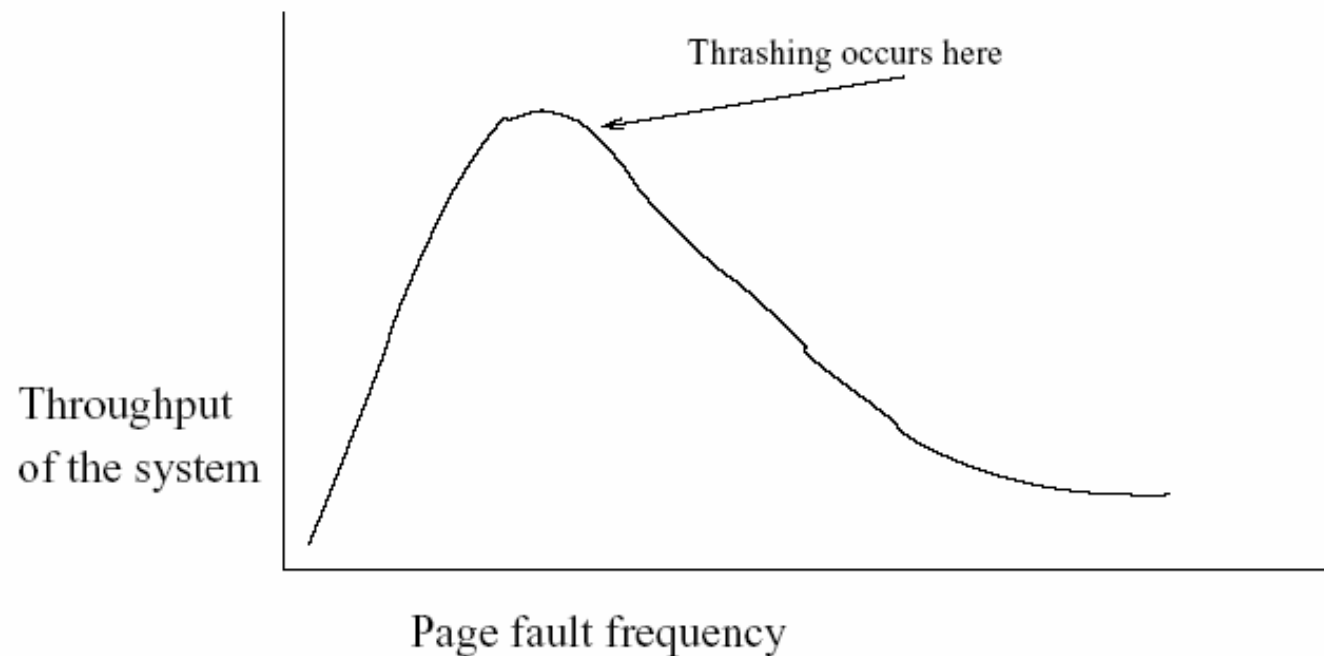


Thrashing - 2

- Note that the page size influences the number of pages and hence it determines the number of resident sets we may support. With more programs in main memory or more pages of a program we hope for better locality of reference. This is seen to happen (at least initially) as more pages are available.

Thrashing - 3

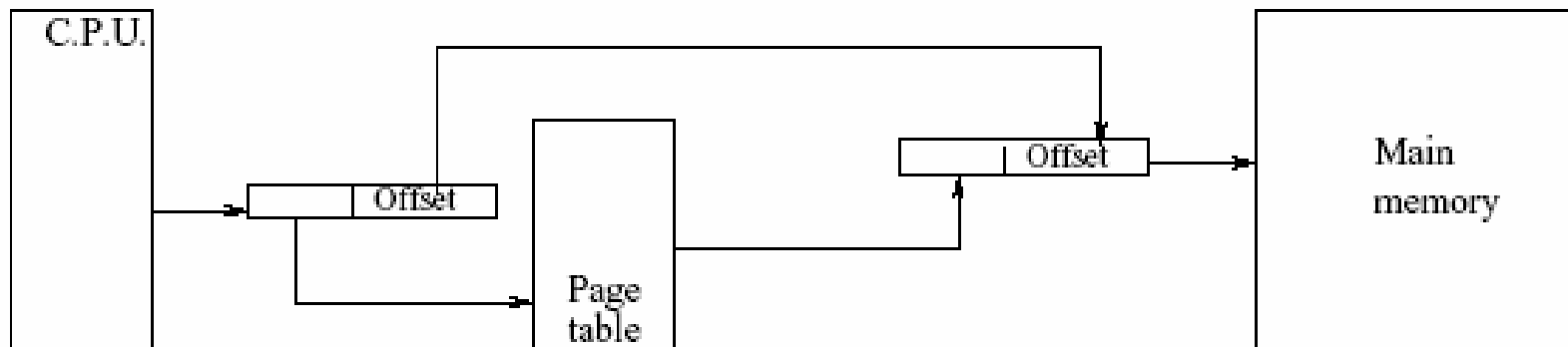
Page-faults result in frequent disk IO. With more disk IO the throughput drops. At this point we say *thrashing* has occurred. Note: the basic advantage of higher throughput from a greater level of utilisation of processor and more effective multi-programming does not accrue any more. The advantages derived from locality of reference and multi-programming begins to vanish and thrashing manifests as shown in figure.



Paging: HW Support - 1

Recall the point we need HW within CPU to support paging. The CPU generates a logical address which must get translated to a physical address. In Figure we indicate the basic address generation and translation.

Address generation and translation



The Offset is same because the page and frame size are same

The page table provides the mapping of virtual page to frame number



Paging: HW Support - 2

- The *sequence of steps* in generation of address is as follows:
 - ✓ The *process generates a logical address*. This address is *interpreted in two parts*.
 - ✓ The *first part of the logical address identifies the virtual page*.
 - ✓ The *second part of the logical address gives the offset within the page*.



Paging: HW Support - 3

- ✓ The *first part* is used as an *input to the page table* to find out the following:
 - ❖ Is the page in the main memory;
 - ❖ What is the page frame number for this virtual page;
- ✓ The *page frame number* is the *first part* of the physical memory address.
- ✓ The *offset* is the *second part* of the correct physical memory location.



Paging: HW Support - 4

- A *page fault* is generated if the page is not in the physical memory – *trap*.
- The *trap* suspends the regular sequence of *operations* and brings the required page from disc to main memory.



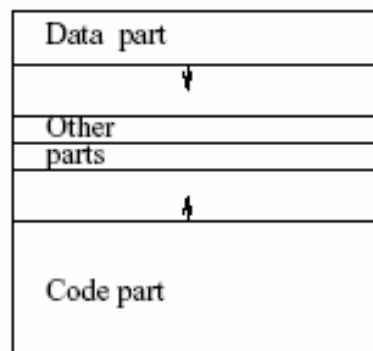
Segmentation - 1

- Segmentation also *supports virtual memory* concept.
- One view of segmentation could be that each part like its *code segment*, its *stack requirements* (of data, nested procedure calls), its different *object modules* etc. has a contiguous space. This view is *uni-dimensional*.

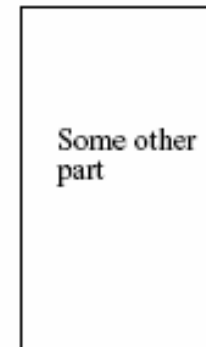
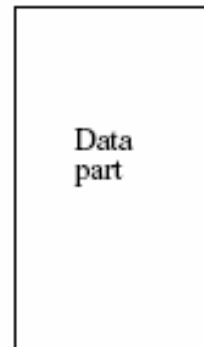
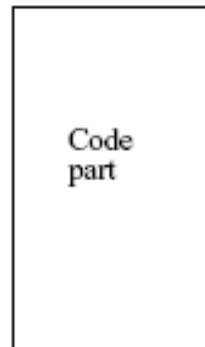
Segmentation - 2

- Each segment has *requirements that vary over time* – *stacks grow or shrink*, memory requirements of object and data segments may change during the process's lifetime.

We, therefore, have a *two dimensional view* of a process's memory requirement - each process segment *can acquire a variable amount of space over time*.



Varying address space





Segmentation - 3

Segmentation is similar to paging, except that we have a segment table look ups to identify address values.

Comparing segmentation and paging :

- Paging offers the *simplest mechanism* to effect virtual addressing.
- Paging *suffers from internal fragmentation*, segmentation *from external fragmentation*.



Segmentation - 4

- Segmentation affords *separate compilation* of each segment with a view to link up later.
- A user may develop a code segment and *share* it amongst many applications.
- In paging, a process address space is linear and hence, *uni-dimensional*. For segmentation each procedure and data segment has its own *virtual space mapping*. Therefore this offers a *greater degree of protection*.



Segmentation - 5

- In case a program's address space *fluctuates* considerably, paging may result in *frequent page faults*. Segmentation offers no such problems.
- Paging partitions a program and data space *uniformly* and hence *simpler to manage; difficult to distinguish* data and program space. In segmentation, space required is *partitioned according to logical division* of program segments.



Segmentation and Paging

- In practice, there are segments for the code(s), data and stack.
- Each segment carries the *raw information* as well.
- Usually, stack and data have *read and write* permissions only; code has *read and execute* permissions only.

Segmentation & Paging:

A clever scheme with advantages of both would be segmentation with paging. In such a scheme each segment would have a descriptor with its pages identified. Such a scheme is shown in figure.

