Module 4 –

Memory Management

Introduction

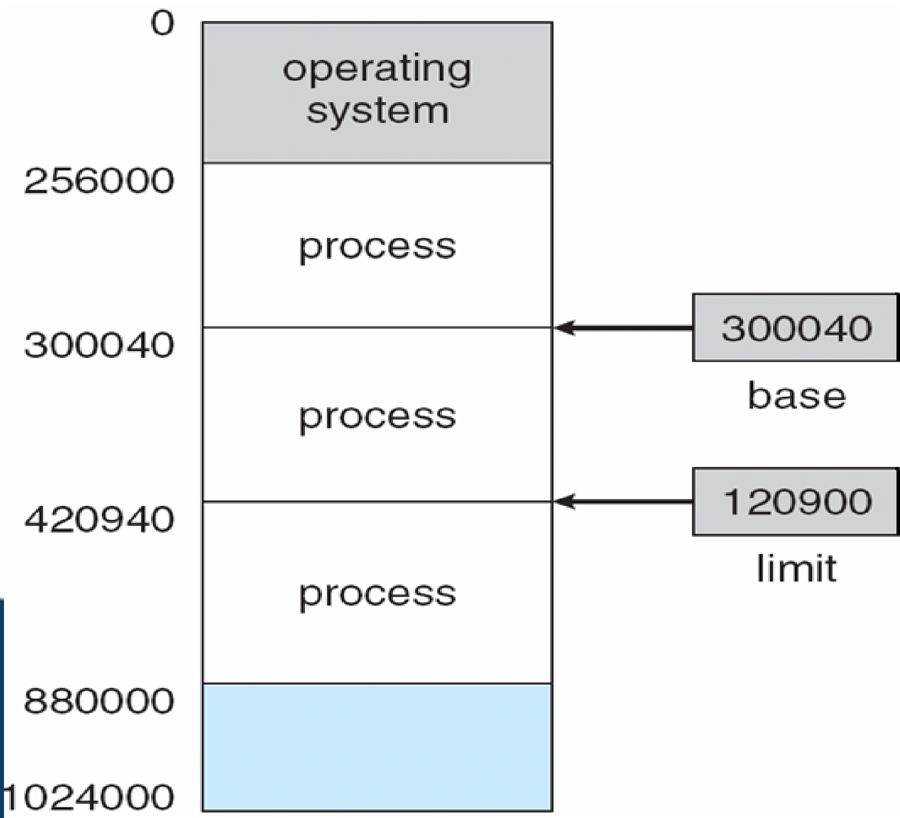
- All the programs are stored in the secondary memory.
- Programs are transferred from secondary to primary memory whenever they have to be executed
- Once programs are executed they have to be bring back to secondary memory from main memory
- This movement of programs between primary & secondary memory is done by **memory manager**
- Memory is an collection of large arrays of words or bytes.
- Each word or byte has its own address

Basic Hardware

- Running program can access only main memory & registers directly but cannot access secondary memory
- WKT, registers are faster than main memory.
- To improve the speed of main memory, cache is used in between CPU and main memory.
- We should not concentrate only on the faster access but we should concentrate on correct operation to protect OS from access by user processes and also to protect user processes from other user processes.
- This protection is provided by proper hardware implementation
- WKT, each process has its own memory space(code & data segment)

Basic Hardware …….contd

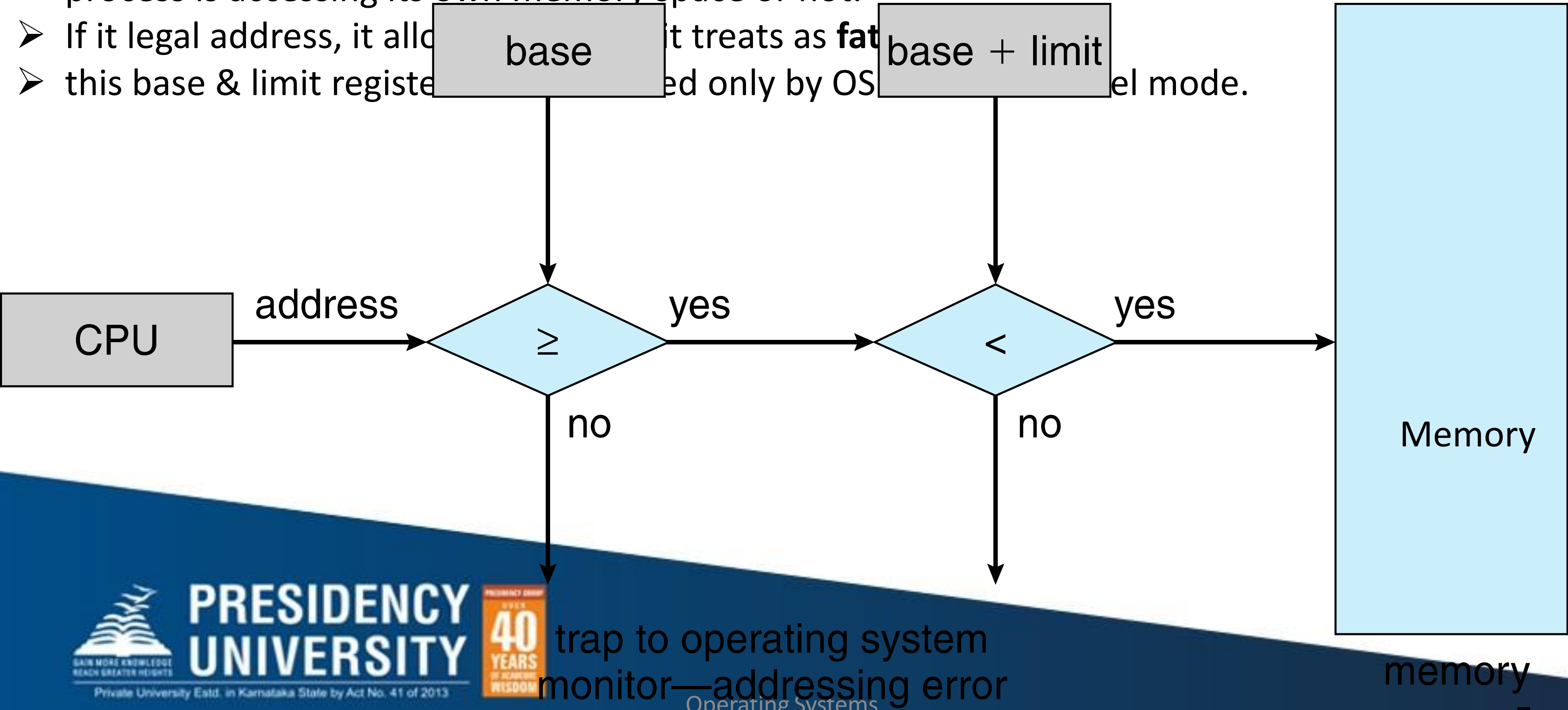
- Instructions of that process should access only from this memory space.
- That means, **processes should know the legal addresses that they can access.**
- OS provides 2 registers called **Base register** and **limit register** in order to specify the legal addresses (thereby protecting each process) that a process can access.
- Base register holds smallest legal physical address (300040)
- Limit register specify the size of the range(120900)



PRESIDENCY UNIVERSITY
GAIN MORE KNOWLEDGE REACH GREATER HEIGHTS
Private University Estd. in Karnataka State by Act No. 41 of 2013
PRESIDENCY GROUP OVER 40 YEARS OF ACADEMIC WISDOM

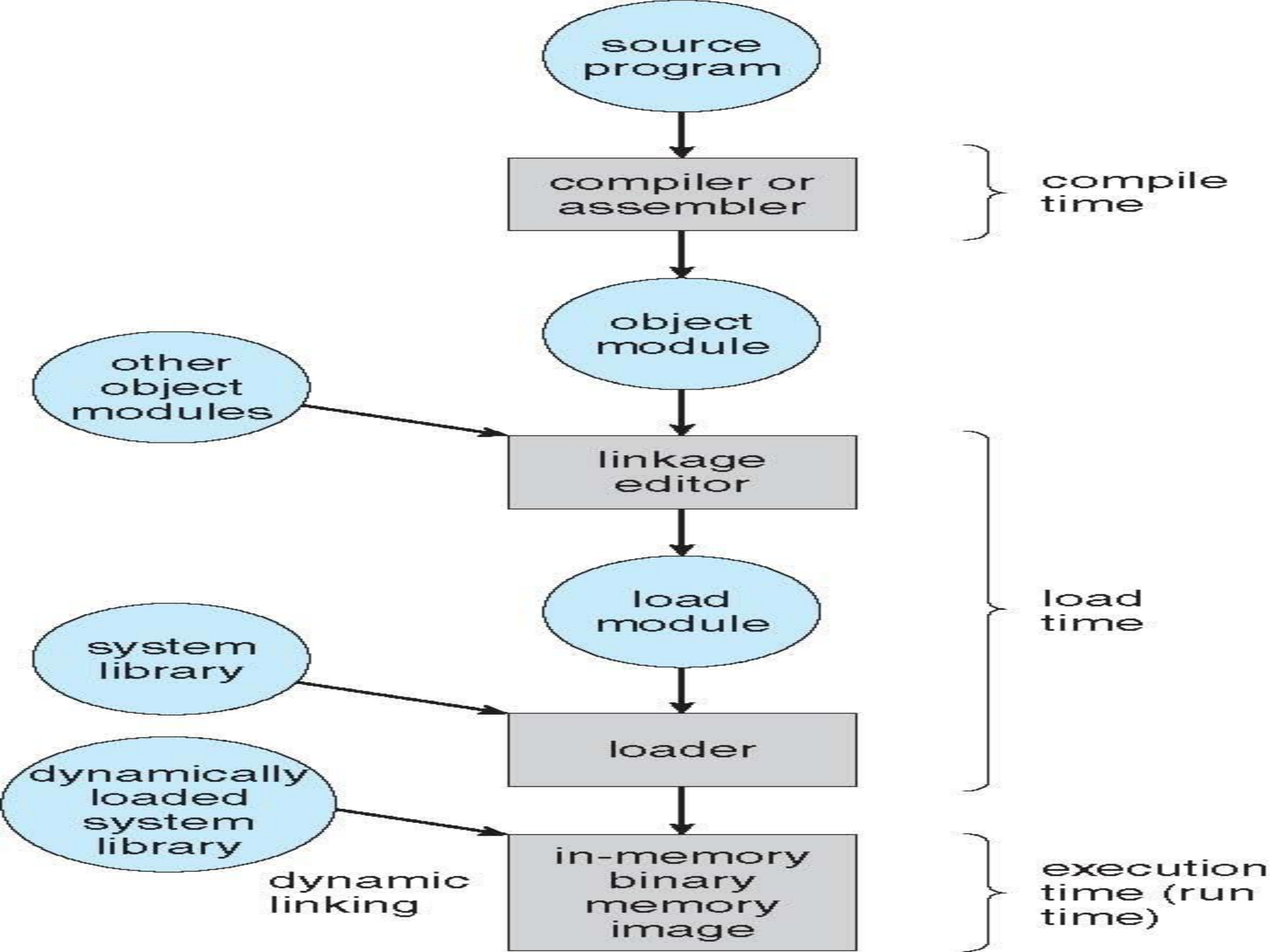
Basic Hardware ……contd

- That means, process 2 can legally access the addresses from 300040 to 420939.
- From 420940, process 3 starts
- Whenever a program is executed, CPU generates address of machine instructions & as well as data associated with that instruction.
- This address is compared with the base & limit registers in order to check whether that process is accessing its own memory space or not.
- If it legal address, it allo it treats as **fat**
- this base & limit registe ed only by OS el mode.



Address Binding

- Programs that are in main memory & waiting for CPU time, will be in **ready queue**
- User process may reside in any part of physical memory.
- Even though, memory address space starts from 00000, no user process occupies memory space from 00000 because user process has to go through several steps before executing.
- Addresses are mapped from one address space to another address space in these several steps.
- Mapping of addresses from one address space to another address space is called **address binding**
- Addresses in the source program are generally symbolic(such as count)


source program
compiler or assembler
compile time
object module
other object modules
linkage editor
load module
system library
load time
loader
dynamically loaded system library
dynamic linking
in-memory binary memory image
execution time (run time)

Address Binding...........contd

- Compiler will typically bind these symbolic addresses to Relocatable addresses
- Linkage editor or loader will in turn bind the Relocatable addresses to absolute addresses
- Binding of instructions & data to memory addresses can be done at any of the following 3 steps
- **<u>1) compile time</u>**

if we know at compile time, where the process will reside in memory, then absolute code can be generated

For example, if we know that, user process will reside at starting location R, then generated compile code will start at that location & extend up from there

If starting location changes(after some time, i.e., if there is a change in the source code), then it is necessary to recompile the code.

Address Binding………..contd

- **2) load time**

  if we don’t know, where the process will reside in memory at the compile time, then compiler must generate Relocatable code

  If the starting address changes, we need only reload the user code to incorporate this changed value.

  3) Execution time

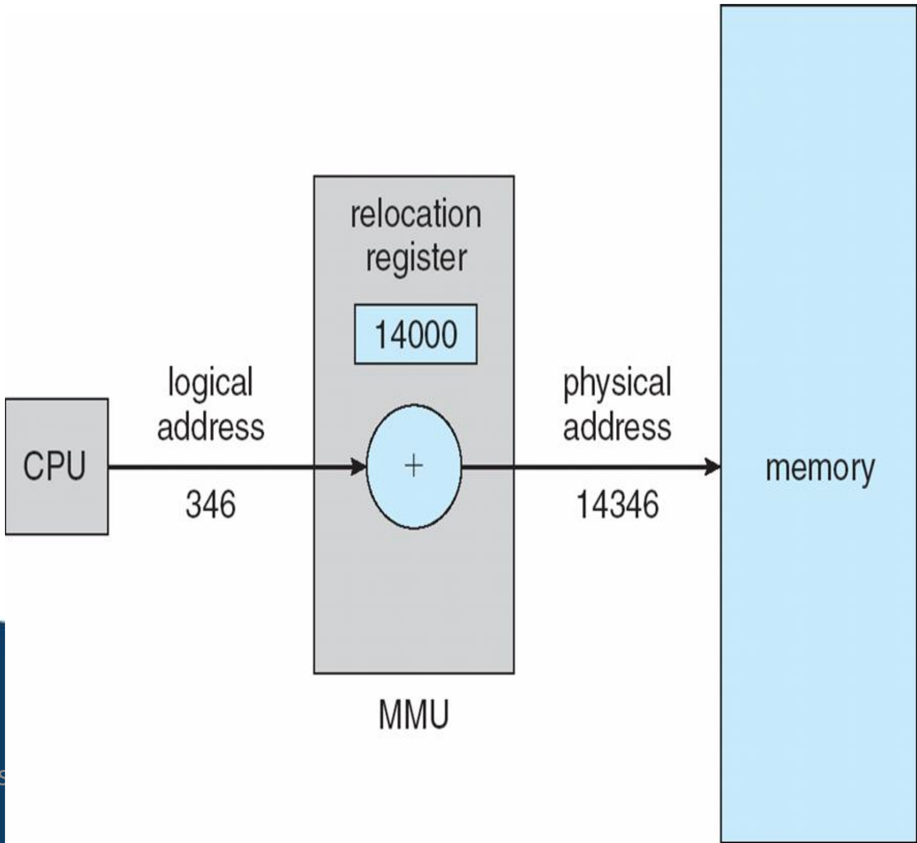
  If the process can be moved during its execution from one memory segment to another, then binding must be delayed until run time.

Logical versus physical address space

- When a user program is executed, CPU generates an address which us called **logical address**
- This executable file has to be loaded into the main memory unit in some address.
- So address seen by memory unit is called **Physical address.**
- During execution time, address binding scheme generates different logical & physical addresses. In this case, logical address is also called as **virtual address**
- Set of all logical addresses generated by a program is called **logical address space**
- Set of all physical addresses corresponding to these logical addresses is called **physical address space**

Logical versus physical address space

- So we need some mechanism/scheme to map the logical addresses into physical addresses during execution time
- Hardware device used for this purpose is called **memory-management-unit(MMU)**
- Whenever CPU executes user program, it generates logical address(346) which is added with the content of relocation register(14000).
- Then the resultant address(14346) will be sent to memory as physical address



Logical versus physical address space

- Range of logical addresses is **0 to max** and range of physical addresses is **R+0 to R+max** for base value R
- user program generates only logical addresses & thinks that, that process runs in locations 0 to max
- These logical addresses must be mapped to physical addresses before they are used.

Dynamic Loading

- A program is ready for execution when all of its data & instructions are loaded into the physical memory.
- Most of the times, size of the program is very large.
- To obtain better memory space utilization, **dynamic loading** is used.
- "All routines (user-defined & in-built functions) are kept on secondary storage (disk) in a Relocatable load format. Only main function is loaded into the physical memory & it is executed."
- "Functions/routines are loaded into the physical memory only when they are called"
- whenever a function/routine is called, calling function first checks whether it is already loaded or not. If it is not loaded, Relocatable linking, loader is called to load the desired function into memory. Then control is passed to the newly loaded routine. This concept is called as dynamic loading.
- The advantage of dynamic loading is that, an unused function is never loaded.

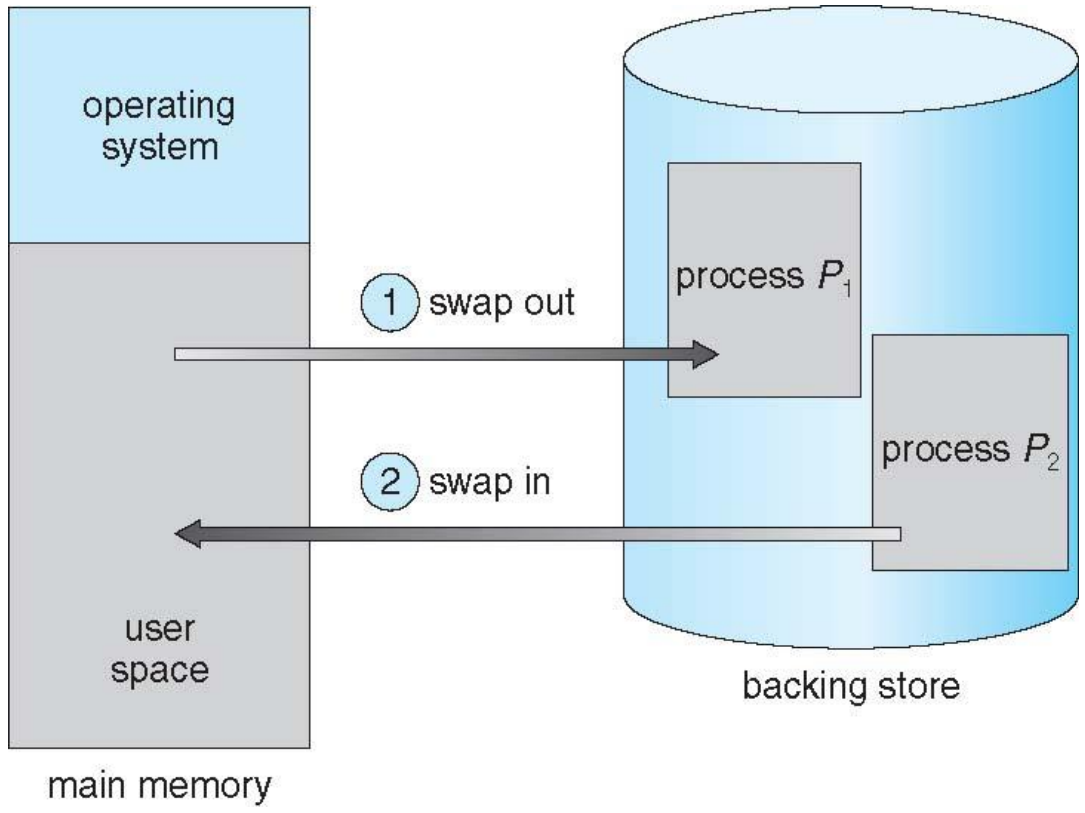
Dynamic Linking

- Dynamic linking is similar to dynamic loading
- In Dynamic linking, linking is postponed until execution time.
- This feature is usually used with the system libraries
- With dynamic linking, a **stub** is included in the image for each library routine reference.
- Stub is a small piece of code that indicates how to locate & how to load the library if the routine is not already present.
- When stub is executed, it checks whether the routine is already present in the memory or not. If it is not, then program loads the routine into memory
- Next time, when same routine is needed, then the library routine is executed directly without dynamic linking.
- Under this theme, all processes that uses a language library execute only one copy of the library code
- This feature of dynamic linking can be extended to library updates.

Swapping

- When a running process moves to waiting state, it no more requires CPU cycles.
- if it is waiting for long time, then no need to keep that waiting process in main memory.
- At this time, OS moves waiting process from memory into secondary storage (disk) which is called **swap out** & another process in secondary storage waiting for main memory allocation is moved into the main memory, which is called **swap in**.
- This procedure of swap-out & swap-in is called as **swapping**
- For example, assume a multiprogramming environment with a round-robin CPU scheduling algorithm.
- When a time-slice expires, memory manager swap-out the process that just finished & swap-in another process into the memory space that has been freed.
- In the meantime, CPU scheduler will allocate a time slice to some other process in the main memory.

Swapping

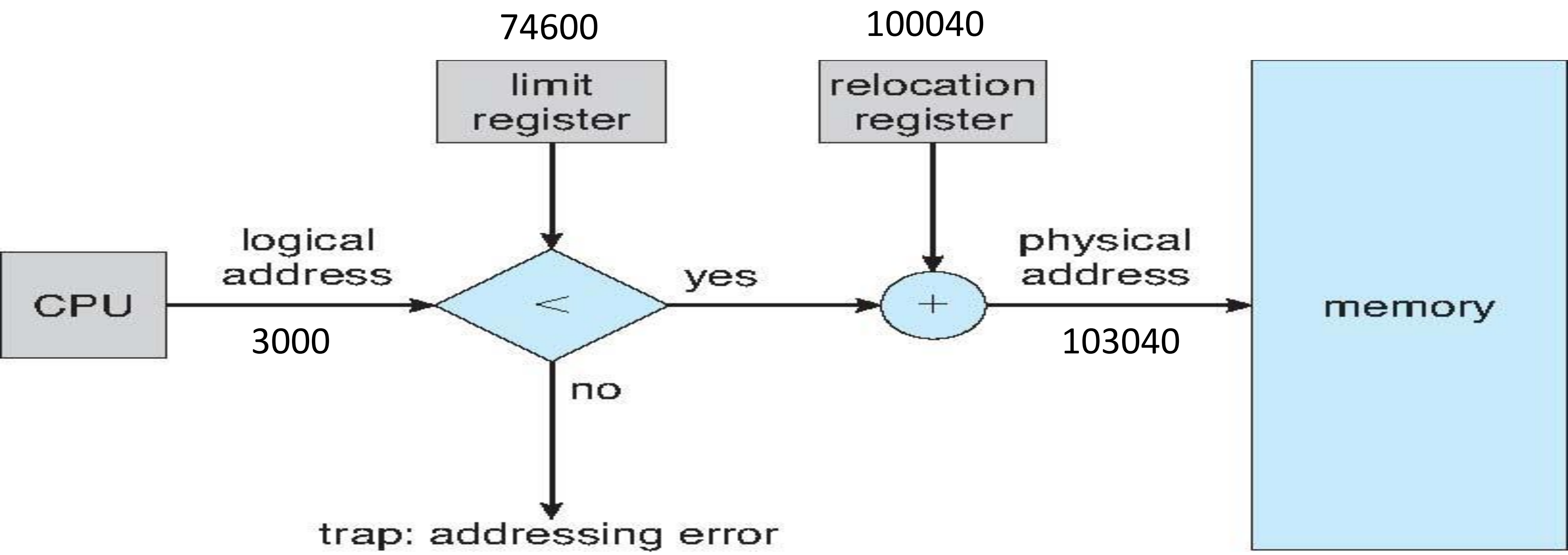


OS's swaps the processes which are completely idle & they are waiting for I/O device for long time
Or running I/O devices for long time & used very rarely.
Only such processes are swapped by OS

Contiguous memory allocation

- In contiguous memory allocation, each process is contained in a single contiguous section of memory
- **Memory mapping & protection** features can be provided by using relocation register & limit register
- relocation register contains value of physical address (say 100040)
- Each logical address generated by CPU should be less than the limit register
- Memory management unit(MMU) maps the logical address dynamically by adding the value in the relocation register
- This mapped address is sent to the main memory.

Contiguous memory allocation



For every logical addresses generated by CPU is checked against relocation & limit register, we can protect both the OS & the other user programs & data from being modified by this running process

Memory allocation

- Generally, main memory is divided into several fixed sized partitions
- Each partition contains exactly one process.
- Thus degree of multiprogramming depends on the number of partitions
- In this **multiple-partition method**, when a partition becomes free, a process is selected from the input queue & is loaded into free partition.
- When the process terminates, partition becomes available for another process.
- In the **variable-partition scheme**, OS keeps a table indicating which parts of the main memory are available & which are occupied
- Initially, all memory is available for user processes & is considered one large block of available memory called **hole**. So, we can say, main memory contains set of holes of various sizes (size of holes are depending on size of user programs).

Memory allocation

- Once, programs are loaded into the main memory then they are competing for CPU.
- When a process completes its jobs, it releases its allocated memory.
- Now, OS should select another program from the input queue.
- While selecting the program from the input queue, OS should check the program size & as well as the size of available block of memory (hole).
- If size of memory block (hole) is lesser than the program size, then OS should skip that program & continue checking in the down of input queue.
- When a process arrives & requests for memory, OS search for a hole which is large enough for that process

Memory allocation

- If the hole is too large, it is splitted into 2 parts. One part is allocated to arriving process & another part is returned to the set of free holes.
- If the new returned hole is adjacent to other holes, these adjacent holes are merged to form one larger hole
- The most commonly used methods to select a free hole from the set of available holes are

1) First-Fit

Allocate the first hole that is big enough. Searching can start either at the beginning of the set holes or at the location where the previous first-fit search ended.

We can stop searching as soon as we find a free hole that is large enough

Memory allocation

- **<u>2) Best-Fit</u>**

  Allocate the smallest hole that is big enough.

  Here, entire list has to be searched if it is not sorted.

  This method produces the smallest leftover hole.

  <u>3) Worst-Fit</u>

  Allocate the largest hole.

  Again, entire list has to be searched if the list is not sorted,

  This method produces the largest leftover hole which may be more useful than smaller leftover hole from a best-fit method.

- First-fit & best-fit are better in terms of time and storage utilization. Generally, first-fit is faster.

Given memory partitions of 100k, 500k, 200k, 300k & 600k. Apply first fit & best fit algorithm to place 212k, 417k, 112k and 426k

Solution:

Mem partitions	First fit	Memory leakage
100k		
500k	212k	500k-212k=288k
200k	112k	200k-112k=88k
300k		
600k	417k	600k-417k=183k

Mem partitions	Best fit	Memory leakage
100k		
500k	417k	500k-417k=83k
200k	112k	200k-112k=88k
300k	212k	300k-212k=88k
600k	426k	600k-426k=174k

Fragmentation

- Both first fit & best fit methods suffer from external fragmentation
- Consider, memory is partitioned(fragmented) into large number of holes of small sizes.
- If any process requests for main memory then it is not possible to allocate sometimes because even though space is available they are not in contiguous.
- So it suffers from fragmentation.
- This fragmentation problem is severe especially when there is a free memory between every 2 processes
- Instead of having many small size holes, if they are in one big free block, we might be able to run several more processes
- If we use first fit or best fit, both affects the amount of fragmentation

Fragmentation

- External fragmentation may be minor or major problem which depends on total amount of memory storage & the average process size
- Memory fragmentation may be internal or external
- Consider a hole of size 18464 bytes. Suppose, if a next process requests 18462 bytes & if we allocate, we left with a hole of 2 bytes
- Now, overhead is to keep track of this small sized 2 byte hole
- Tracking such very small sized hole is complex than tracking large sized holes
- The general approach to avoid such internal fragmentation problem is to break the physical memory into fixed - size blocks & allocate memory in units based on block size.

Fragmentation

- With this approach, memory allocated to a process may be slightly larger than the requested memory.
- The difference between these 2 numbers is **internal fragmentation** - unused memory that is internal to a partition.
- One solution to the problem of external fragmentation is **compaction**.
- The goal is to shuffle the memory contents so as to place all free memory together in one large block.
- When compaction is possible, we must determine its cost. The simplest compaction algorithm is to move all processes toward one end of memory; all holes move in the other direction, producing one large hole of available memory.
- Hence, this scheme, is more expensive,

Paging

- In order to avoid compaction & external fragmentation, paging is used.
- Today, in most of the OS's, paging is used as memory management scheme that permits physical address space of a process to be **noncontiguous**,
- Basic method for implementing paging involves

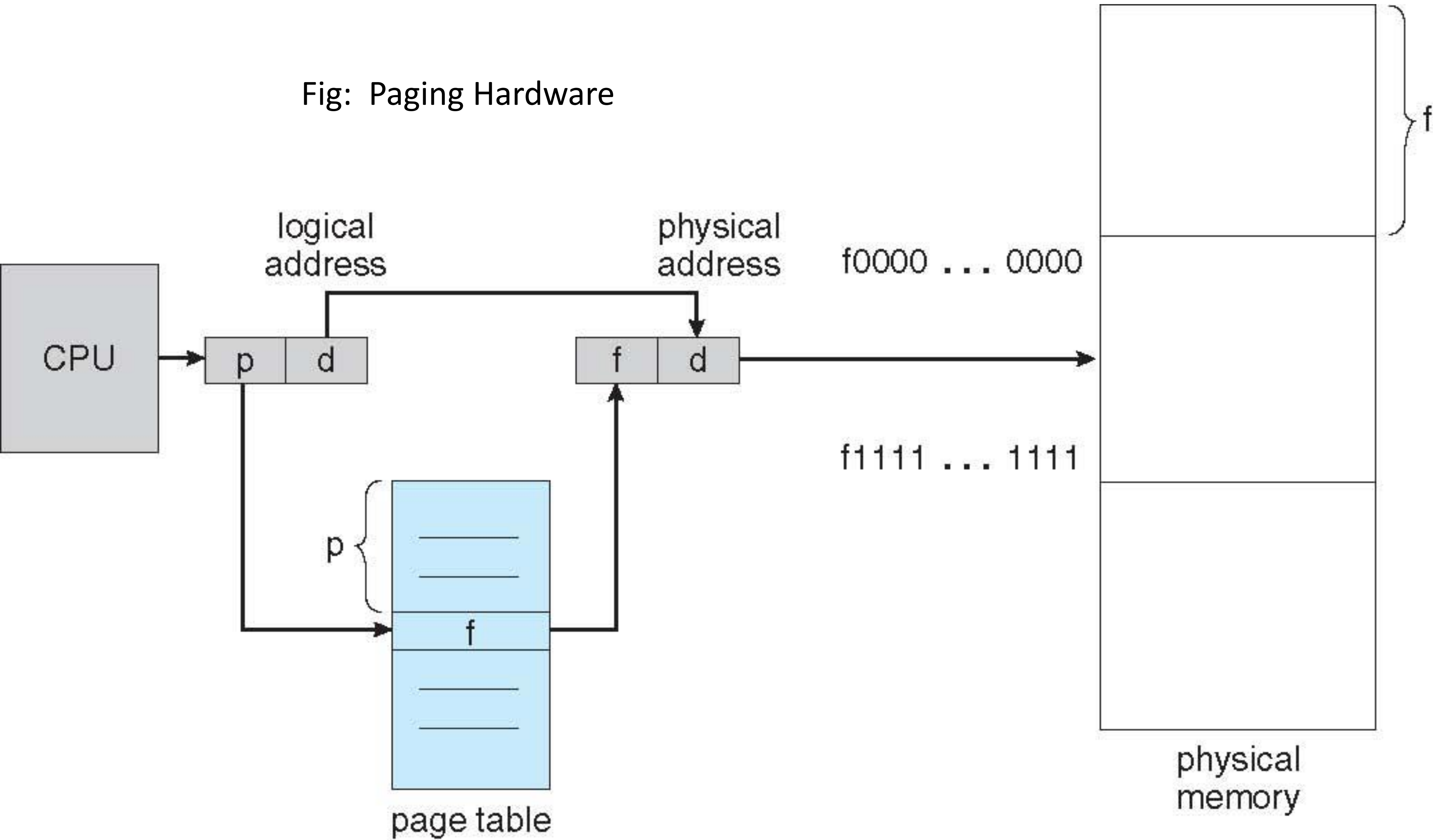
1) breaking physical memory into fixed-sized blocks called **frames**
2) breaking logical memory into blocks of same size called **pages**.

- When any process is to be executed, then all of its pages have to be loaded into the available memory frames (main memory) either from the file system or backing store.
- Frames in the memory may be contiguous or non- contiguous

Paging

- So, backing store also to be divided into fixed-sized blocks that are of the same size as the memory frames

Fig: Paging Hardware

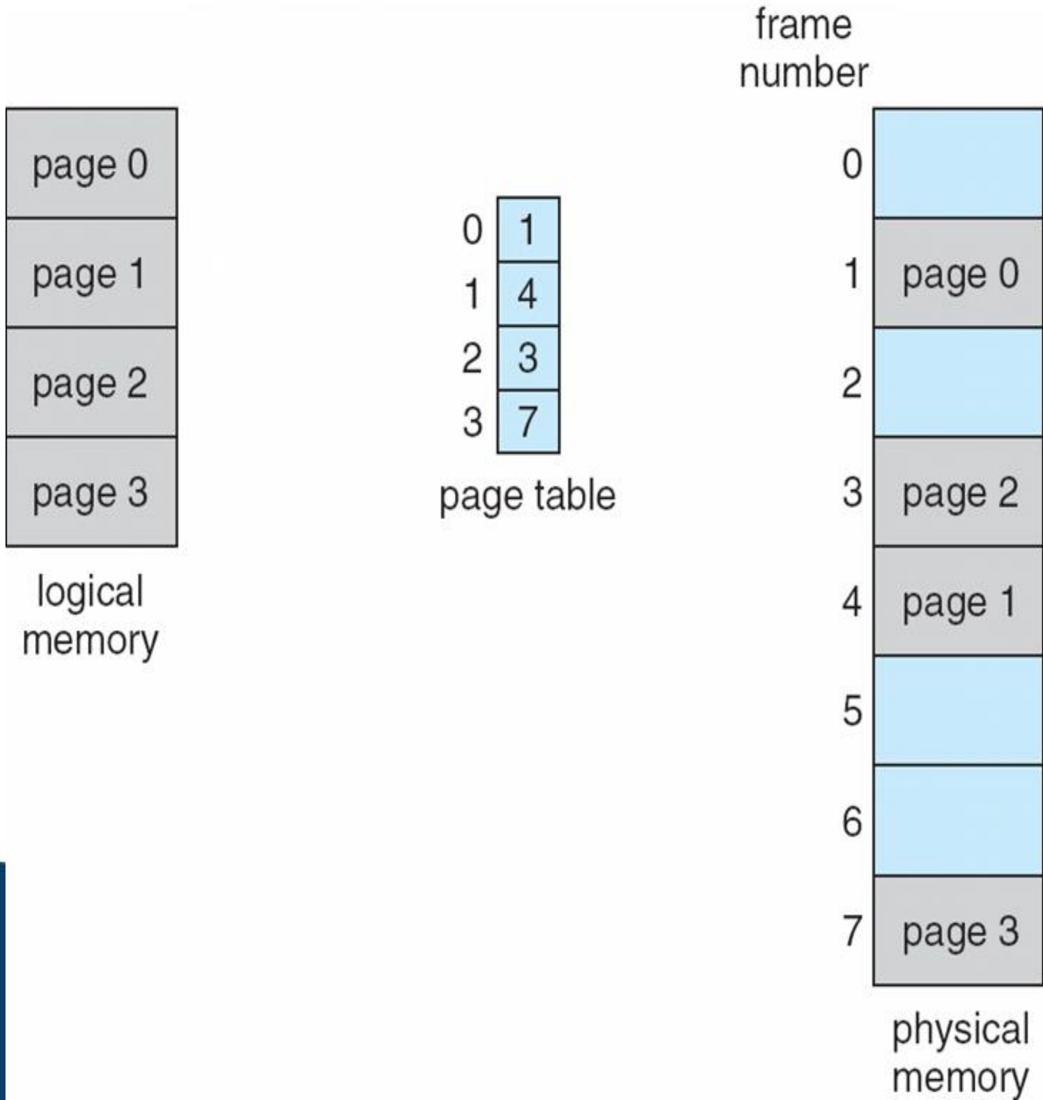


Paging

- WKT, whenever a program is executed, CPU generates an logical address related to that program(instruction).
- That logical address is having 2 parts
  1) page number (p)
  2) page offset (d)
- page number is used as an index into a index into a **page table**.
- page table contains base address of each page in physical memory
- Paging model of main memory (physical memory) is as shown in the fig (next slide)
- page size is defined by the hardware used in the system.
- Always page size is same as frame size.

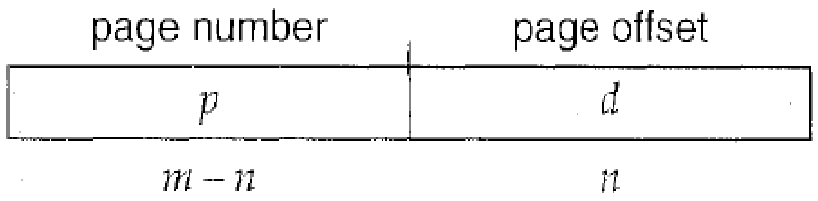
Paging

- Typically, page size is in terms of power of 2, varying between 512 bytes (2^9) & 16 MB (2^{14}), depending upon the computer architecture.
- In general, page size is 2^n



Paging

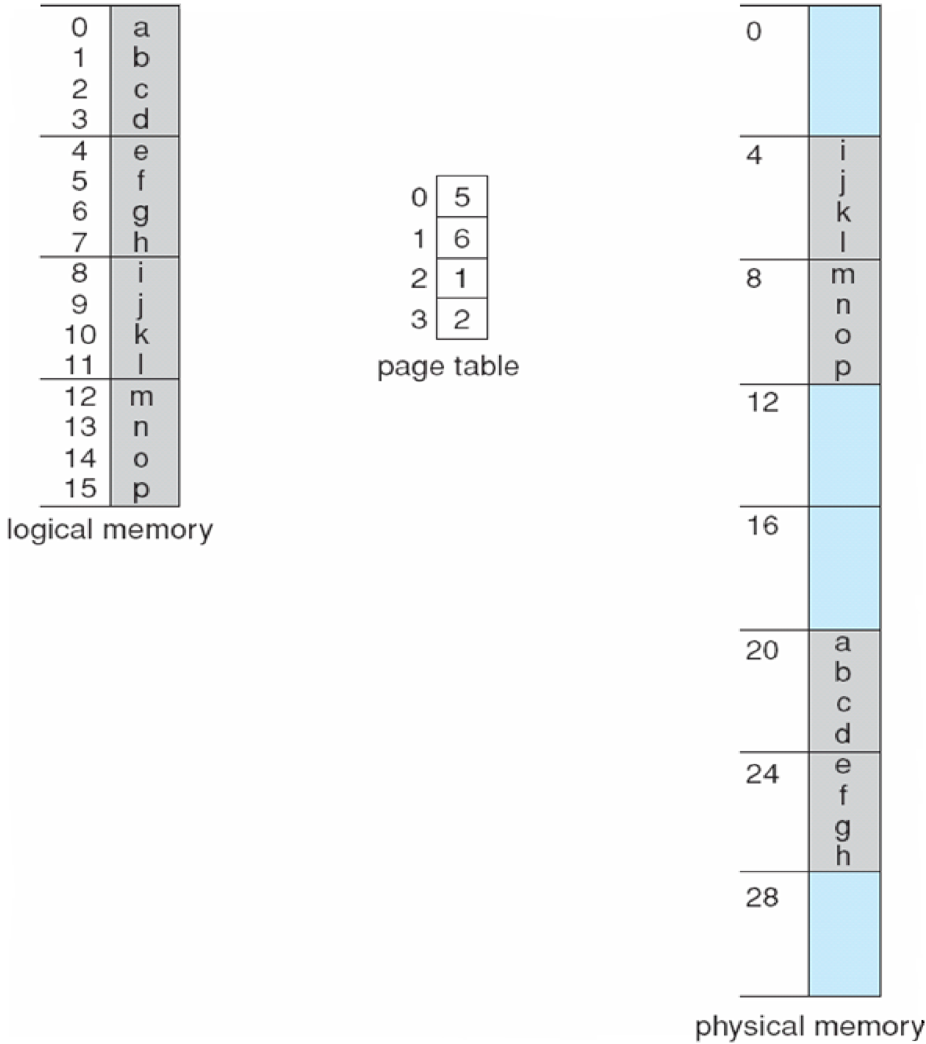
- If the size of the logical address space is 2^m & a page size is 2^n addressing units(bytes or words), then the higher order m-n bits of a logical address designate the page number & the n low order bits designate the page offset
- Thus logical address is given by



- P is an index to the page table
- d is the displacement within the page

Paging example for 32 byte memory with 4 byte pages

- For logical address, let us take n=2 and m=4
- Size od each page in logical memory= 2^2 = 4 bytes
- Total logical address space = 2^m = 2^4 = 16 bytes



Paging example for 32 byte memory with 4 byte pages

- Let us see, how user's view of memory can be mapped into physical memory

  1) logical address 0 is page0, offset0. from page table, we can find that page0 is in frame5

  Thus, logical address0 maps to physical address (5*4)+0=20

  2)) logical address 3 is page0, offset3. from page table, we can find that page0 is in frame5

  Thus, logical address3 maps to physical address (5*4)+3=23

  3)) logical address 13 is page3, offset1 maps to physical address (2*4)+1=9

- When paging scheme is used in our system, then there is **no external fragmentation** because any free frame can be allocated to any process that needs it

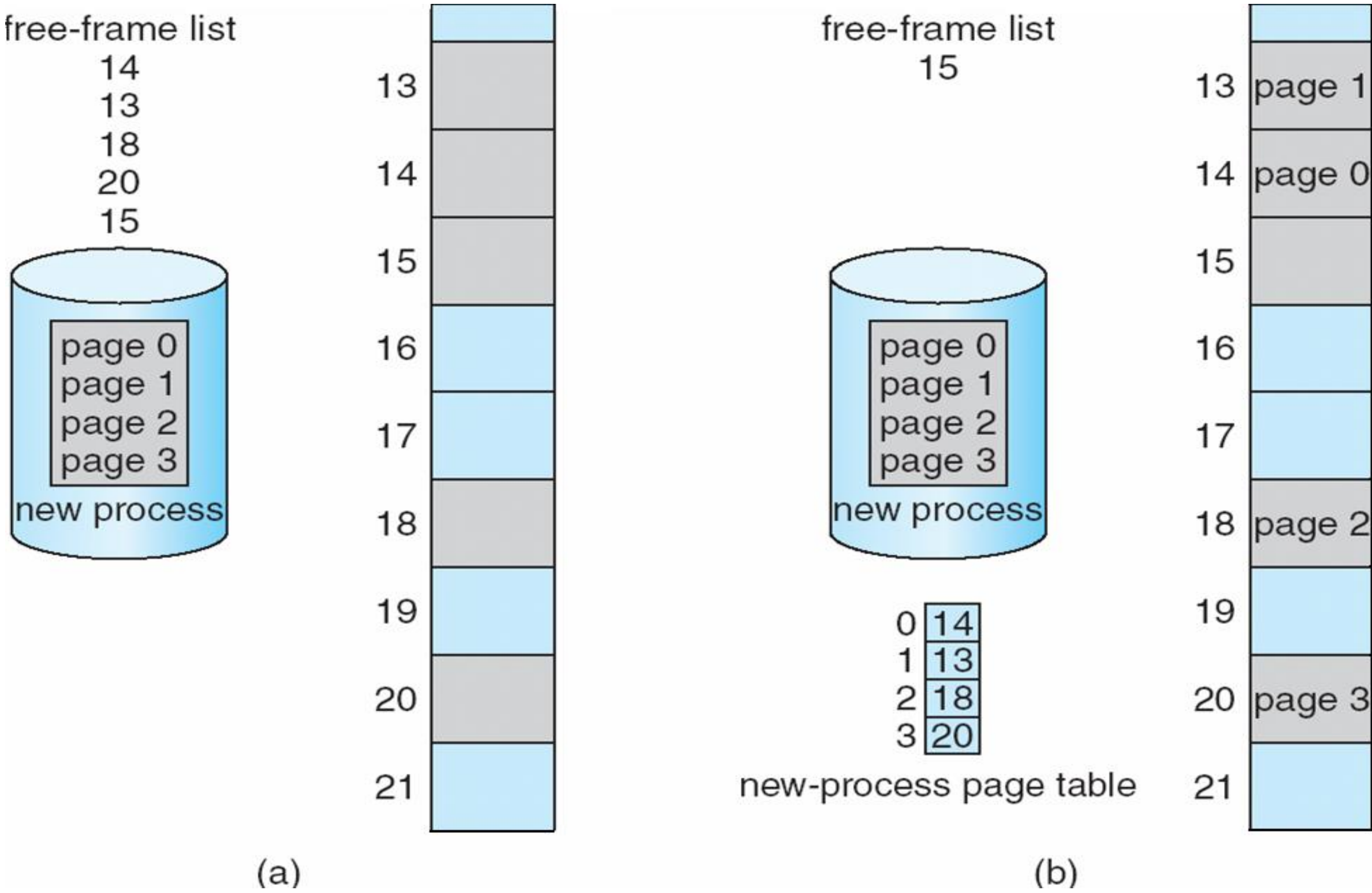
Paging example for 32 byte memory with 4 byte pages

- But **there will be some internal fragmentation.** In paging, we are allocating frames as a fixed units.
- If the memory needed by a process is fits within the frame, then there is no problem.
- Suppose, a process needs memory which is slightly more than the frame, then we need to allocate 2 frames to that process.
- Out of 2 frames, one is completely utilized & the last frame may not be used completely
- To avoid this internal fragmentation, page sizes are kept very small
- When page size is small, there will be more number of pages
- This leads to extra overhead of maintaining larger page table

Paging

- When a process arrives into the system for execution, it is expressed in terms of pages based on the process size.
- Let us consider a process which consists of n number of pages.
- In order to execute a process, all its pages has to be loaded into the physical memory.
- first page of the process is loaded into the one of available frame & that frame number is entered into the page table.
- Next page is loaded into another available frame & its frame number is entered in the page table & so on.
- This is as shown in the fig (next slide)

Paging



Before allocation

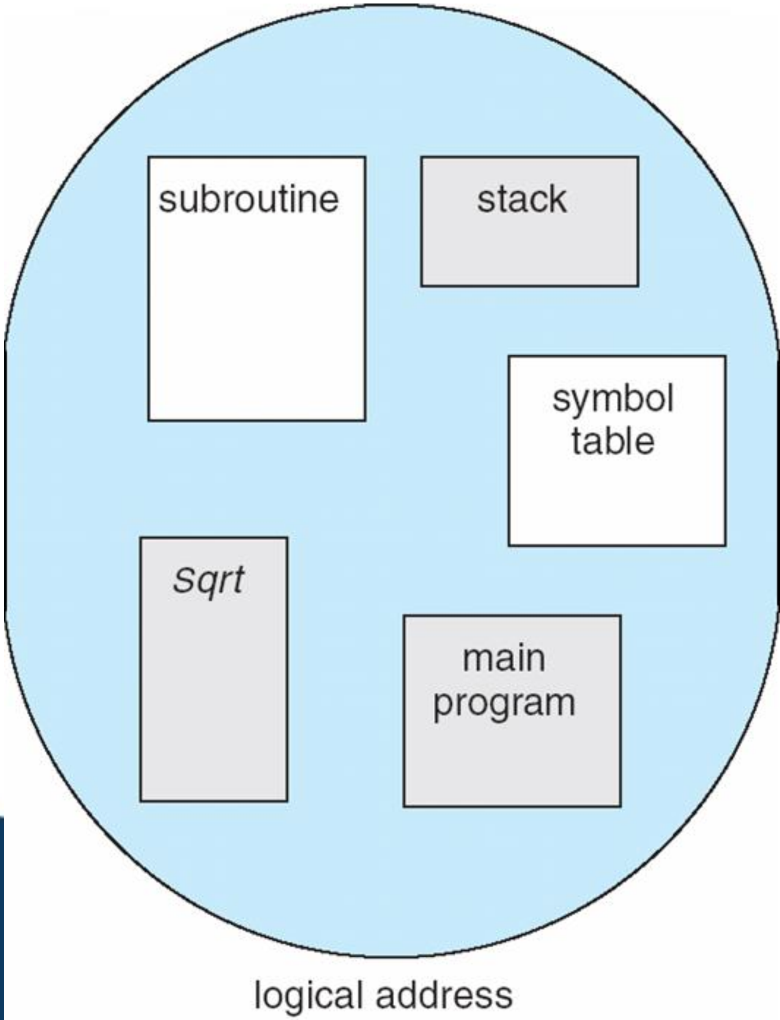
After allocation

Paging

- From paging scheme, it is clear that, program(process) doesn't fit into the physical memory in contiguous location
- Thus, user program is scattered throughout physical memory which also holds other programs
- Thus, we can define "**Paging** is a memory management scheme that permits physical address space of a process to be non-contiguous".

Segmentation

- Most of the users thinks that their programs are stored in main memory in a variable sized multiple segments, where each segment is dedicated to a particular use such as code segment, data segment, stack, heap, etc,
- Here, each of these segments is of variable length.
- Elements within a segment are identified by their offset from the beginning of segment
- Segmentation is a memory management scheme that supports user view of memory



Segmentation

- Segmentation is a technique to break memory into logical pieces where each piece represents a group of related information.
- For example, data segments or code segments for each process, data segment for OS and so on
- Unlike paging, segments are having varying sizes & thus segmentation eliminates internal fragmentation
- A logical address space is a collection of segments.
- Each segment has a name & length

Segmentation

- Addresses generated by CPU is divided into

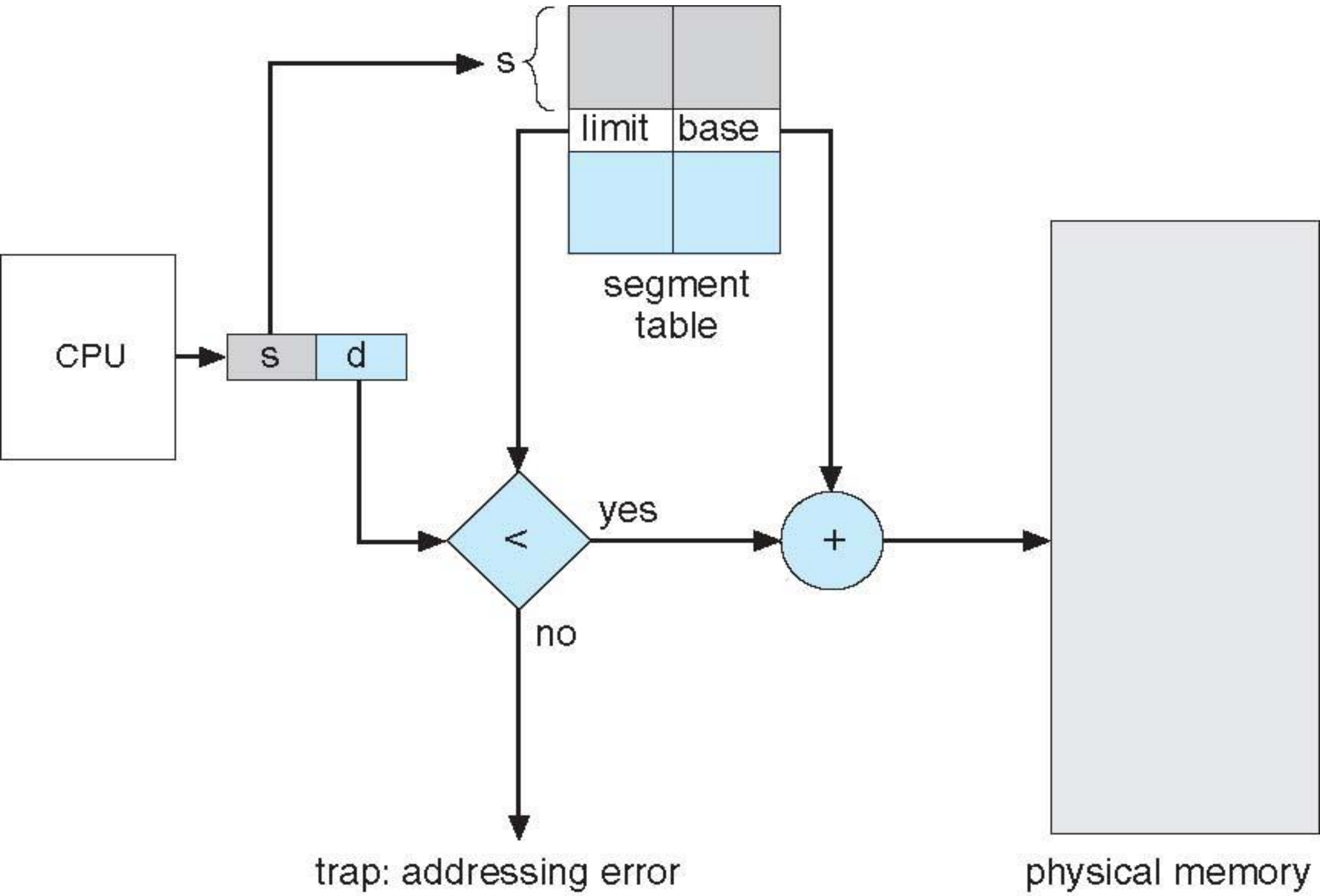
  1) **segment number(s):-** used as an index into a segment table which contains base address of each segment in physical memory & a limit of segment

  2) segment offset (d):-it is first checked against limit & then is combined with base address to define the physical memory address
- But, segments are referred by its numbers than its name
- Thus logical address consists of 2 tuple **< segment-number, offset>**

Segmentation-hardware

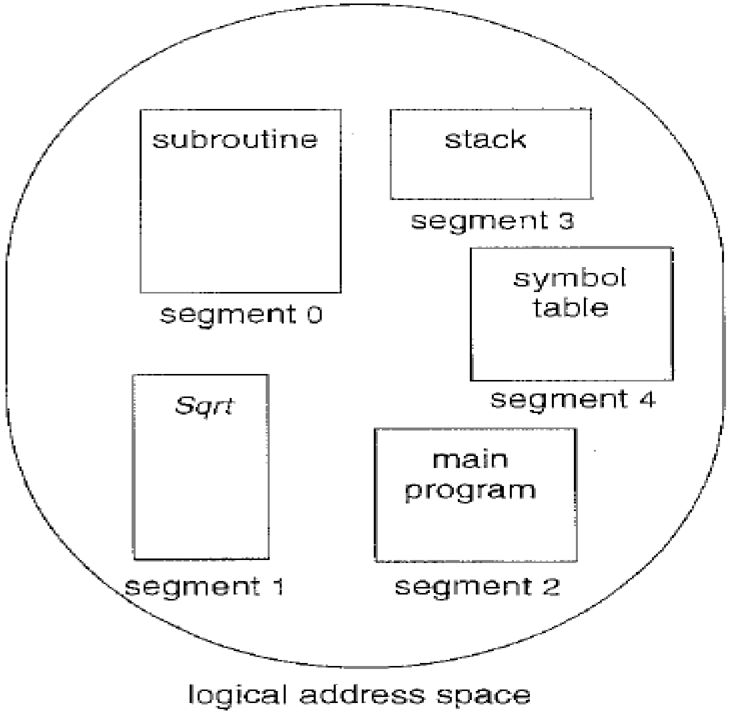
- User refers the objects in the program by 2 dimensional address, But, physical address is in one-dimensional address.
- Thus, we need an hardware implementation to map 2 dimensional user defined address into one-dimensional physical address.
- This mapping is done by using segment table.
- Each entry in the segment table has a segment base & segment limit.
- Segment base contains starting address of physical address where the segment resides in main memory & segment limit specifies the length of the segment. This is as shown in fig (next slide)

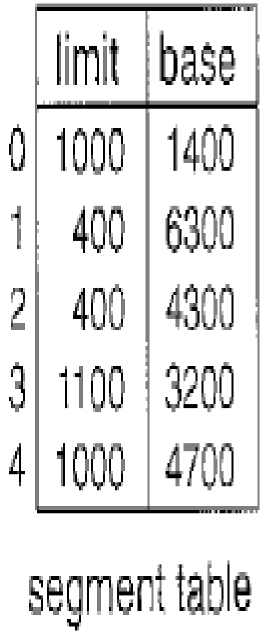
Segmentation-hardware



Segmentation-hardware

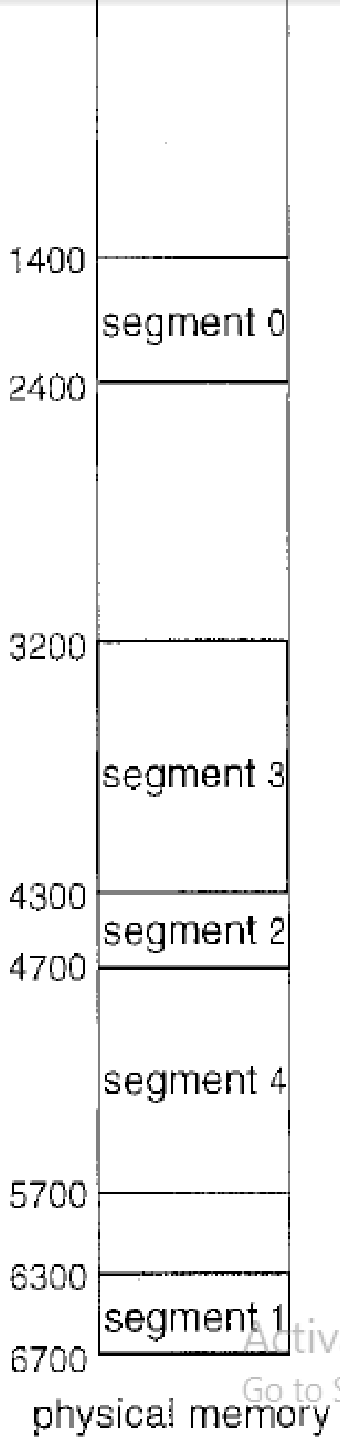
- Logical address generated by CPU consists of 2 parts
  1) segment number(s)   2) segment offset(d)
- Offset d must be between 0 & segment limit. If it is not, it is considered as addressing error(i.e., beyond segment)
- When an offset is legal, it is added to the segment base to produce the address in physical memory of the desired byte.
- For example, consider the situation shown





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

segment table



Segmentation-hardware

- There are 5 segments numbered through o to 4 of any One program.
- Segments are stored in physical memory as shown(prev slide)
- Segment 2 is 400 bytes long & begins at location 4300. Thus, reference to byte 53 of segment 2 is mapped onto location a 4300 +53=4353
- A reference to segment 3, byte 852, is mapped to 3200 (base of segment 3) +852=4052
- A reference byte 1222 of segment 0 would result in trap to the OS, as this segment is only 1000 bytes long.

Module 4 –

Virtual Memory Management

Virtual Memory Management-Introduction

- Main goal of memory management is to keep processes in the main memory simultaneously to allow multiprogramming
- **Virtual memory** is a technique that allows the execution of processes that are not completely in memory
- For most of the processes it is not necessary to store all their pages or atleast not all the pages at once.
- This is for the following reasons

1) error handling code is not needed unless that specific error occurs, some of which are quite rare

2) most of the times, small portion of array is used

3) certain features of certain programs are rarely used

Virtual Memory Management-Introduction

- So, it is not necessary to laod all the pages of a program into main memory but it needs only to load the page that will be used at that time
- Thus, the ability to load only the portions of processes that were actually needed has several benefits

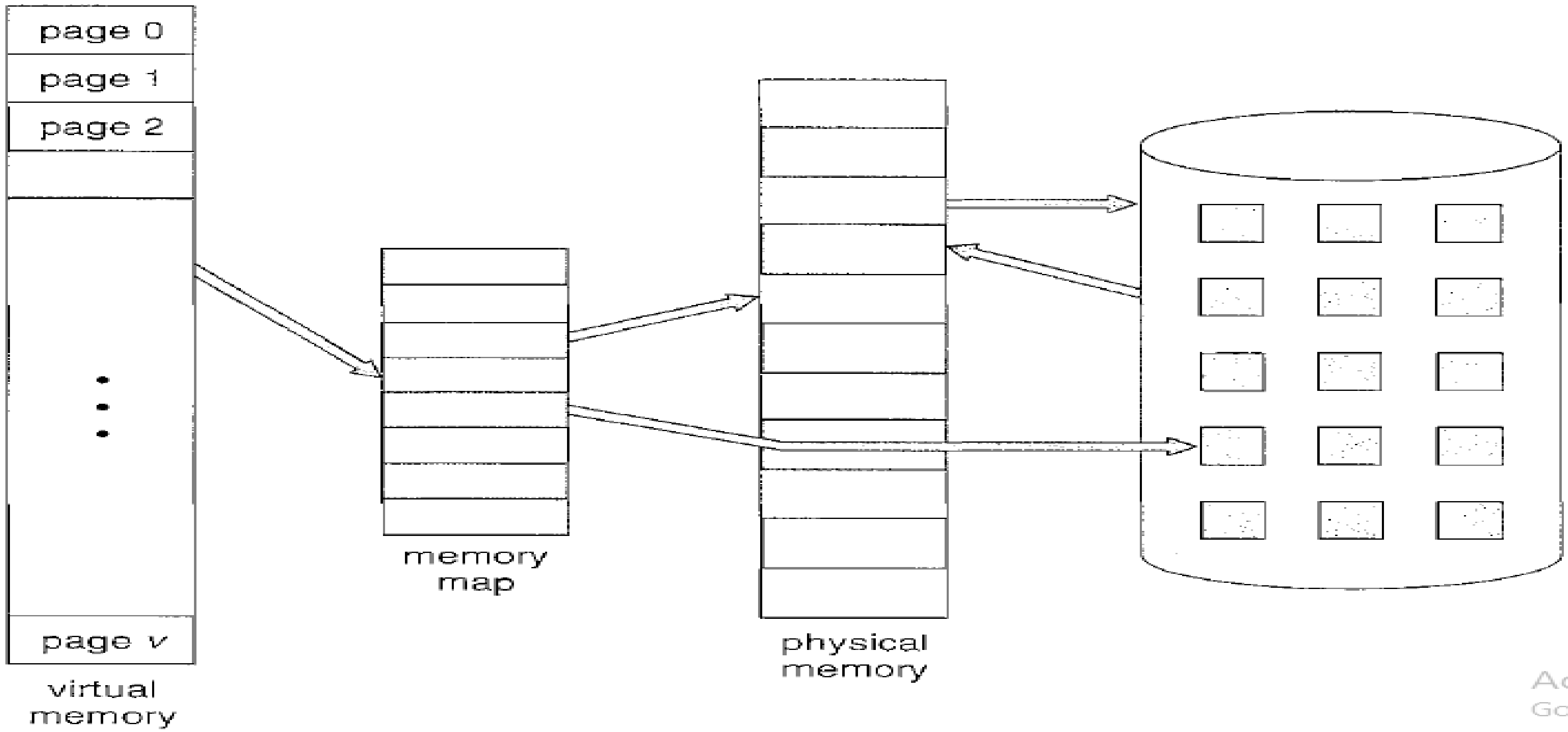
1) programs can be written in any large size even more than the size of main memory

2) since, each process is using only fraction of their total address space, there is more memory left for other programs, improving CPU utilization & system throughput

3) less I/O would be needed to load or swap user programs into mermory, so each user program would run faster

Virtual Memory Management-Introduction

- Virtual memory involves the separation of logical memory as perceived by users from physical memory.
- This separation allows an extremely large virtual memory to be provided for programmers when only a smaller memory is available



Demand Paging

- Genearlly, virtual memory is implemented using demand paging
- Consider, how an executable program might be loaded from disk into memory. There are 2 options

1) load the entire program into physical memory during its execution time

problem with this approach is that, we may not initially need the entire program in memory

2) load the pages only as they are needed-----**demand paging**

pages are only loaded when they are demanded during program execution. Pages that are never demanded, are never loaded

Demand Paging

- Demand paging is similar to paging method with swapping
- In **paging,** all the pages of a process are loaded into the main memory before it is get executed
- Rather than swapping entire process(all pages), we use **lazy swapper**
- When a process is swapped in, its pages are not swapped in all at once. Rather, they are swapped in only when process needs them i.e., on demand. This is called **lazy swapper**

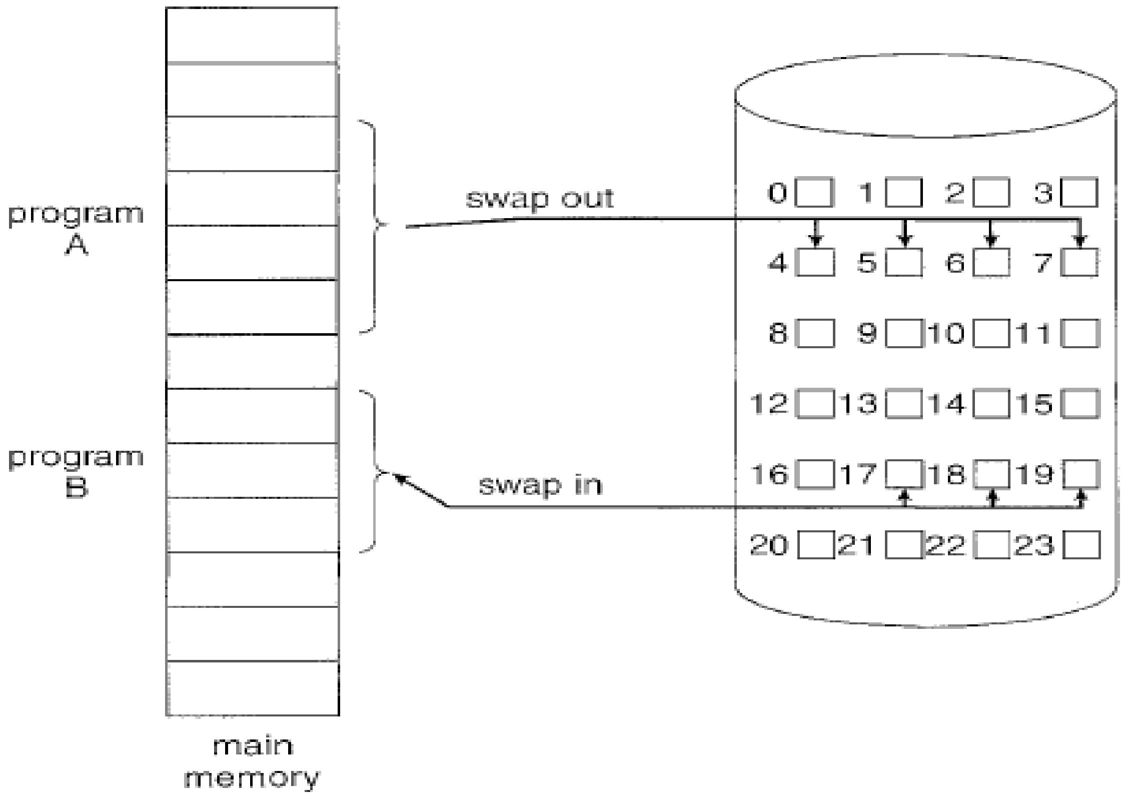
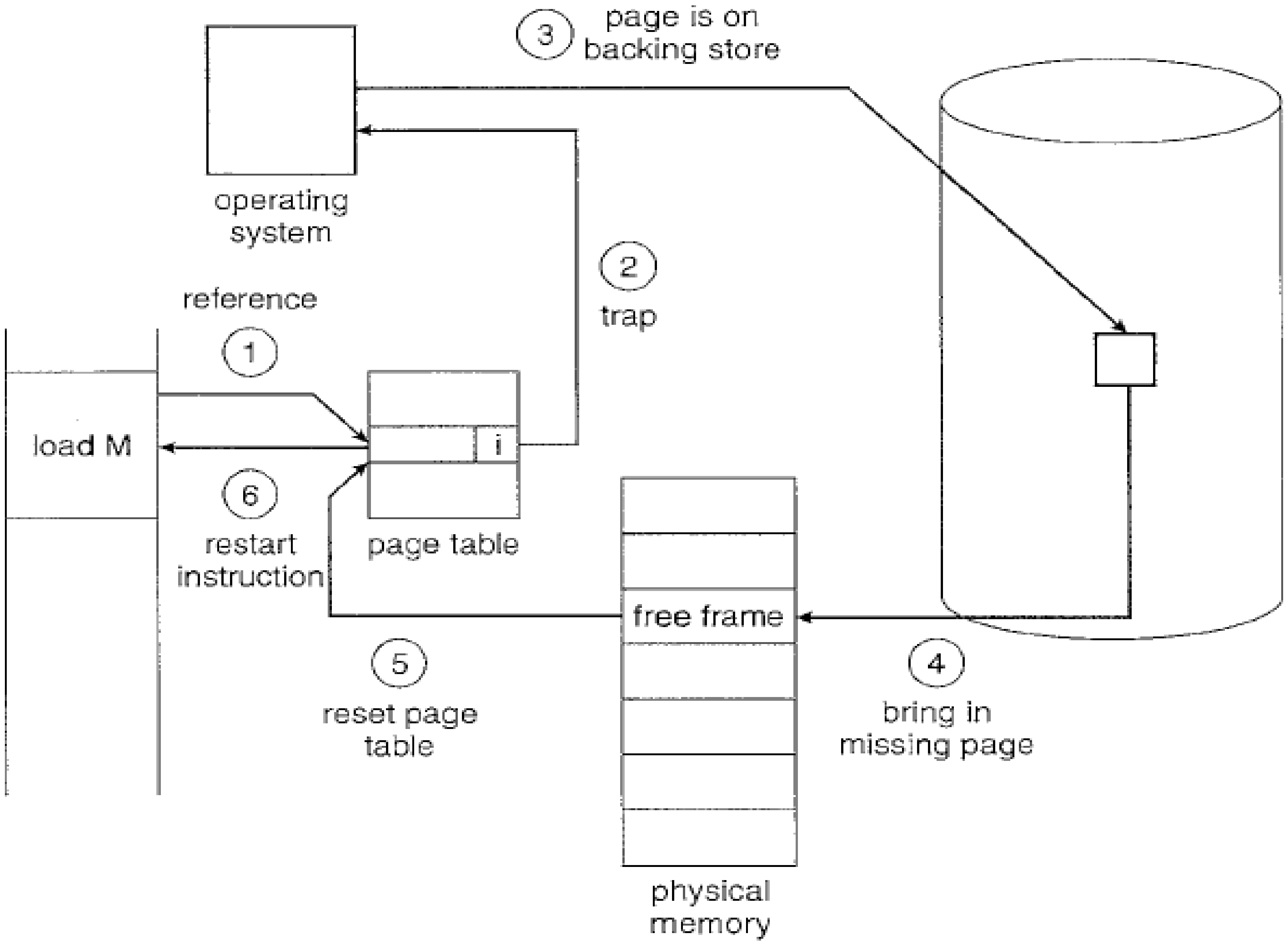


Figure 9.4 Transfer of a paged memory to contiguous disk space.

Demand Paging----Basic Concepts

- Basic idea behing demand paging is that to load only those pages which are demanded by the process. This is done through **pager**
- Pager guesses the pages of a process which may be used by the running process & only those pages are loaded into memory.
- Remaining pages of the process wil be in disk only.
- So we need some hardware support to distinguish between the pages that are in memory & pages that are in disk. For this **valid-invalid bit** scheme is used
- If this bit is set to "valid", then the corresponding page is legal & is in memory
- If this bit is set to "invalid", then the corresponding page is not legal (may be not in the logical address space of the process or legal but is not loaded into the memory)
- If process only accessing the pages that are loaded in memory, then process runs exactly as if all the pages were loaded into the memory
- If any process tries to access a page which is invalid, then OS treats it as **page fault**

Steps in handling a page fault



Demand Paging----Basic Concepts

1) Memory address of an instruction is checked first, to make sure that, whether it was a valid(v) or invalid(i) memory request
2) If the reference was valid, that particular page is there but it was not loaded & hence paged-in

   if the reference was invalid, the process is terminated
3) If reference is valid & page is not loaded, then OS will find the free frame from the free-frame list
4) Missed page will be copied from backing store into the free frame of phsyical memory
5) Missed page number & its corresponding frame number are inserted/included into the page table. (updating the page table)
6) Restart the instruction that was interrupted by the trap.

Demand Paging----Basic Concepts

- When the OS sets the instruction pointer to the first instruction of the process & which is not in the memory, then page fault occurs.
- After missed page is brought into memory, process continue to execute.
- After loading all the pages like this(after page fault), process can execute with no more faults
- This scheme is called as **pure demand paging**
- pure demand paging says, NO pages are swapped-in (loaded) until they are requested by page faults

Copy-on-write

- Whenever a fork() system call is executed, child process is created & its parent process address space is copied to child process
- parent process address space means parent process pages are copied to its child processes
- Since, both the parent & child processes are running same pages, no need to create separate frames, instead of that they can be shared
- But sharing is having some disadvantages. So we go for another technique called **copy-on-write**
- In **copy-on-write,** initially parent & child process share the same pages
- These shared pages are marked as **copy-on-write** pages. This means, if any one process tries to write/modify/manipulate the shared page then at that time, copy of the shared page is created.
- Till any one tries to modify, pages are shared (hence, the name, **copy** the pages **on writing)**

Copy-on-write

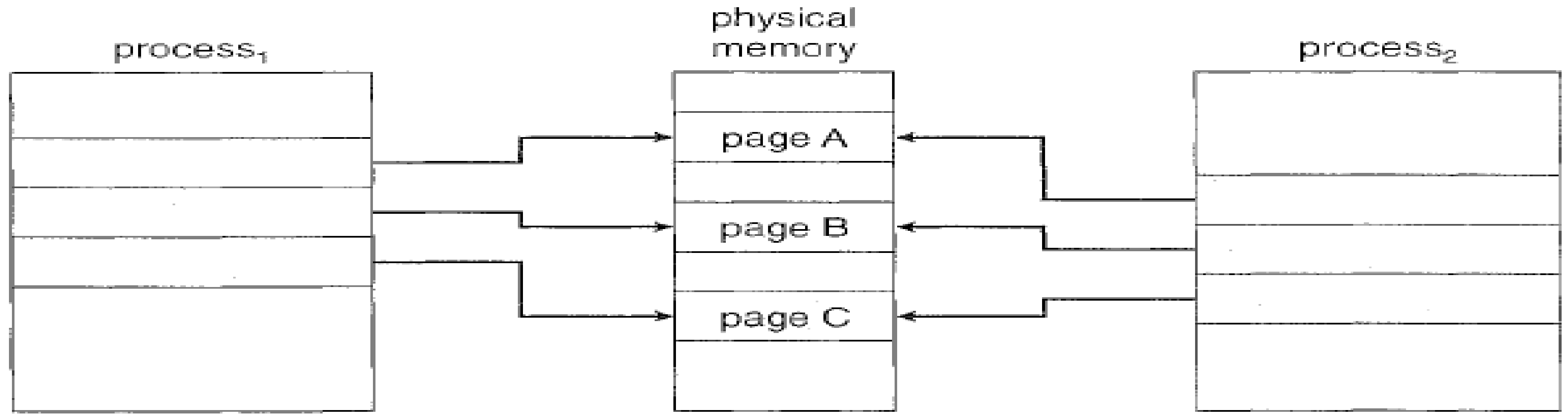


Figure 9.7 Before process 1 modifies page C.

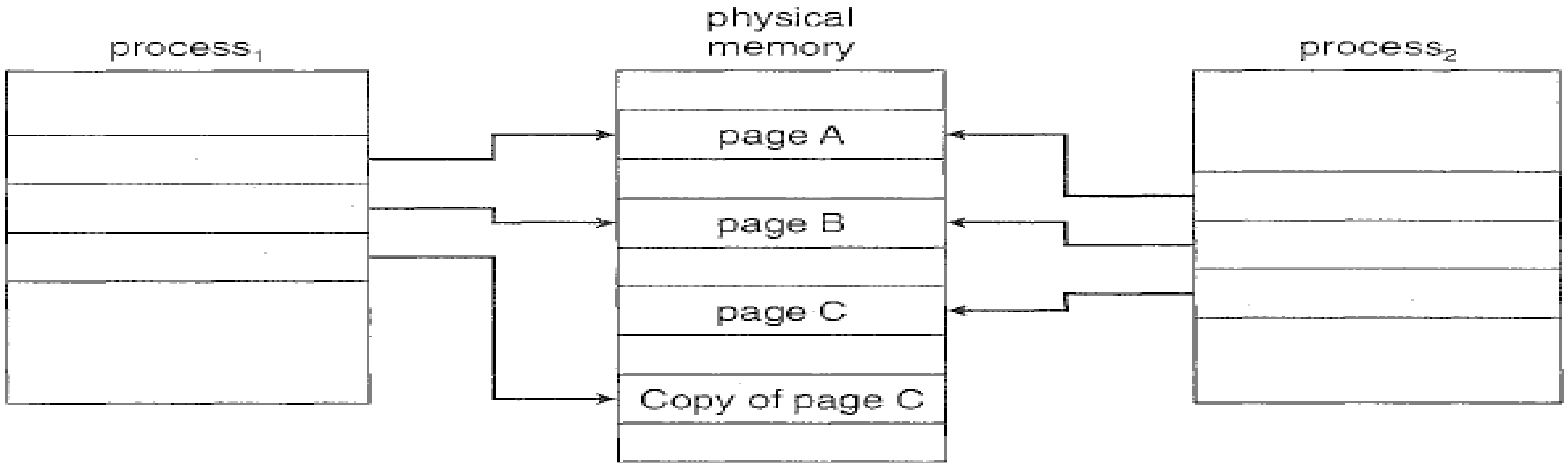


Figure 9.8 After process 1 modifies page C.

Page replacement

- From demand paging & pure-demand paging, it is clear that pages are not loaded into frames until it is demanded
- If a process of 10 pages actually uses only half of them, then demand paging saves 5 pages that are never used
- Assume, 12 processes are in ready queue. Each process is of 10 pages. If we use paging concept, then we need to load all the pages.
- Assume, there are totally 40 frames are free in memory. With paging, we can load only 4 process
- If we use demand paging, we can load 8 process(5 pages of each process).
- There by **we can increase the degree of multiprogramming**
- Eventhough we are increasing the degree of multiprogramming, we are **over-allocating** the memory
- It is also possible that, when suddenly all the processes needs all of its pages, then 40 more frames are required but no frames are free

Page replacement

- Over-allocation of memory manifests itself as follows:

1) while a process is executing, a page fault occurs(because we are not loaded all the pages)

2) OS determines where the desired page is residing on the disk but then finds that there are no free frames

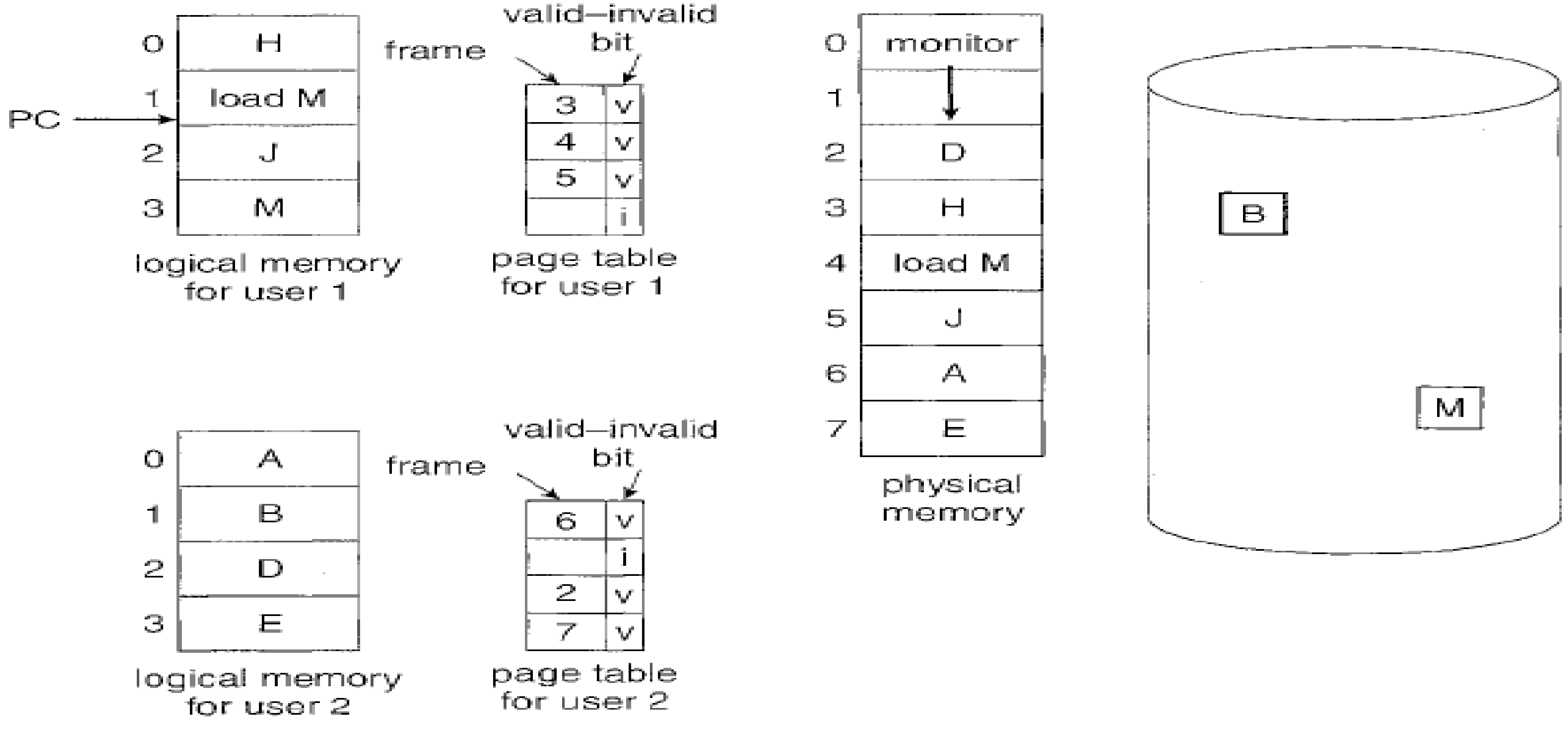


Figure 9.9 Need for page replacement.

Page replacement

- If process needs some more pages to be loaded & there are no free frames, then OS has several options

  1) OS can terminate the user process itself(which is requesting more). But it is not a good option

  2) adjust the memory used by I/O buffering, etc, to free up some frames for user processes

  3) put the process which is requesting more pages into waiting queue until some frames become free

  4) swap some process out of memory completely, freeing up it page frames. But this reduces the degree of multiprogramming

  5) find some page/frame in the memory that is not being used right now or from long time & swap out only that page from memory & then that free frame can be allocated to the requested process. This is known as **page replacement** and is the most common solution

Basic Page replacement

- If there is no free frame, find frame which is not currently being used & free it.
- Once, we free a frame of any process then its corresponding page table should be updated indicating that page is no more in the memory
- This page replacement should be done, when page fault occurs & there is no free frame
- So, we modify the page fault service routine to include page replacement

  1) find the location of the desired page on the disk

  2) find a free frame

     a) if there is a free frame, use it

     b) if there is no free frame, use page replacement algorithm to select **victim frame**

     c) write the victim frame to the disk; change the page & frame tables accordingly

  3) read the desired page into newly freed frame; change page & frame tables

  4) restart the user process

Basic Page replacement

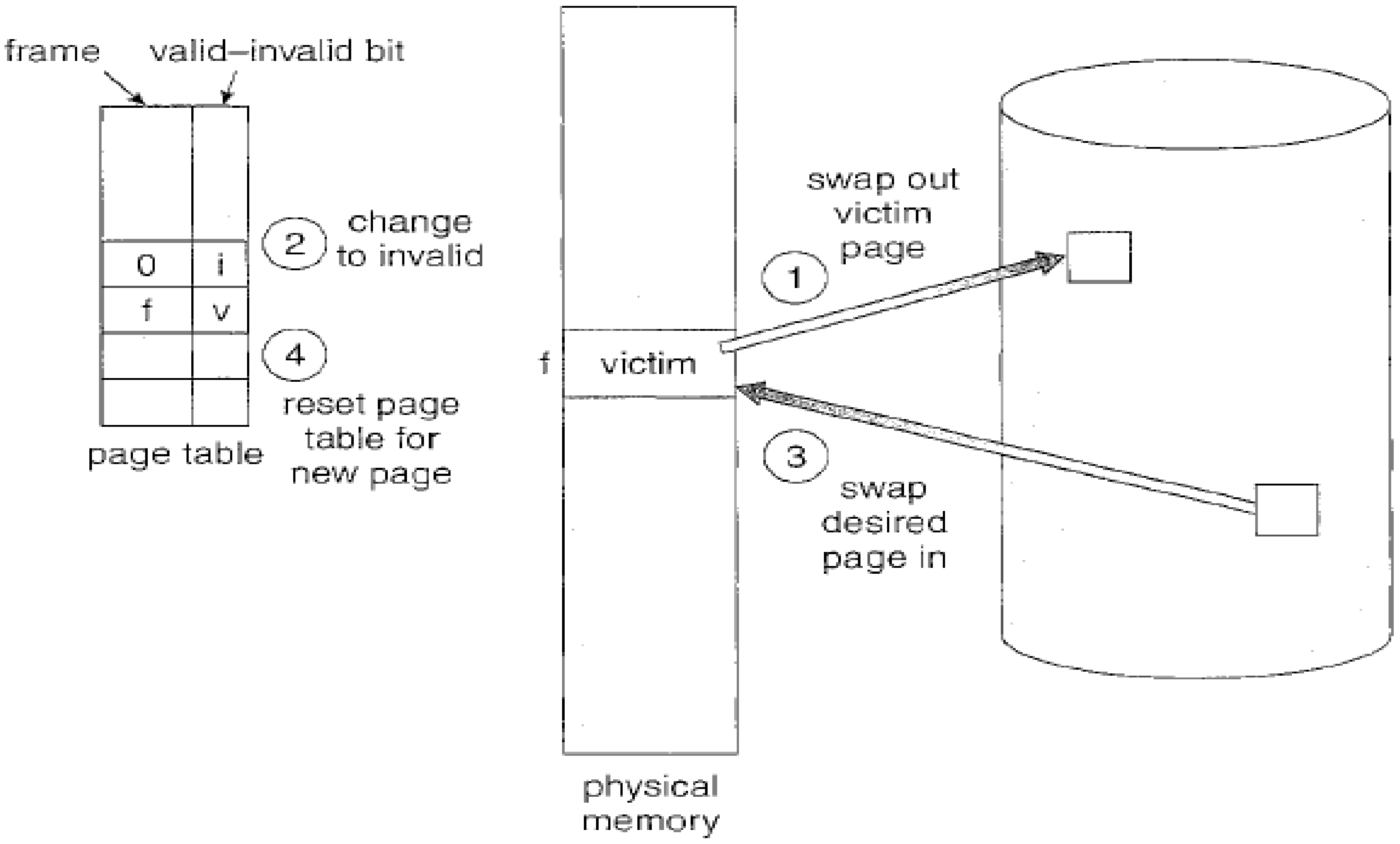


Figure 9.10 Page replacement.

Basic Page replacement

- When no frames are free, 2 page transfers are required(one for moving out the victim page into disk & another one is to bring requested page from disk)
- This situation doubles the page-fault service time & also increases the effective access time.
- We can reduce this overhead by using **modify bit** also called as **dirty bit**
- Each page or frame has modify bit along with protection bit
- Usually, we copy the page from logical memory into frame. Initially, modify bit is 0, indicating there is no change in the content i.e., both page & frame are same
- If modify bit of frame is 1, indicates that frame is modfified/changed & need to change the page also.
- When we select a page for replacement, we examine its modify bit
- If modify bit is not set, then page is not changed & hence no need to update the corresponding page in the disk
- If modify bit is set, then page is changed & hence first we need to update those changes in the disk also before that page has been replaced

Basic Page replacement

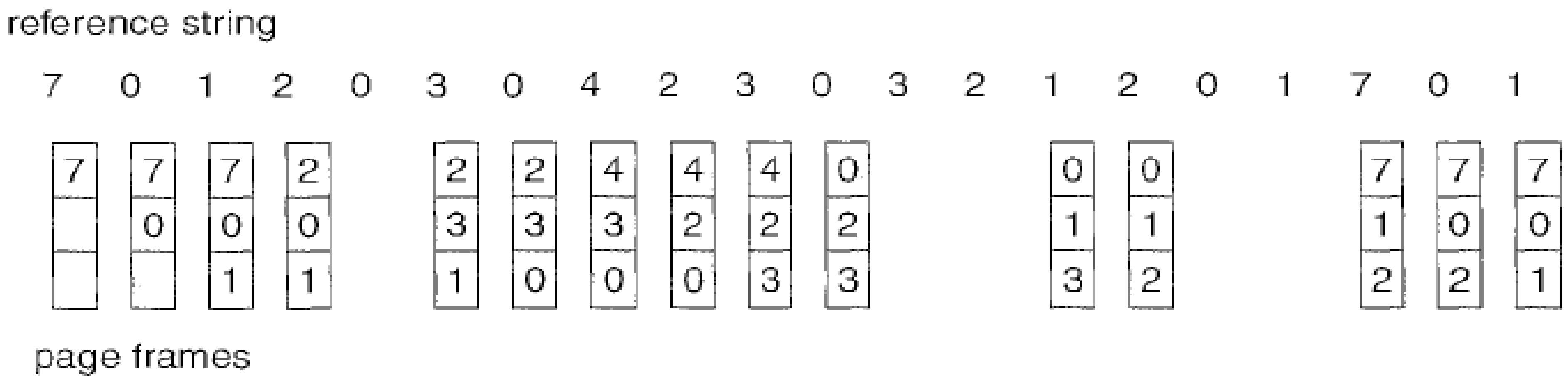
- There are 2 major problems to implement demand paging & those 2 problems have to be solved. They are
  1) frame-allocation algorithm
  2) page-replacement algorithm
- frame-allocation algorithm deals with, deciding how many frames are allocated to each process
- page-replacement algorithm dals with, how to select a page for replacement when there are no free frames are available
- In order to check performance of these 2 algorithms, we use **reference string**
- Algorithms are evaluated using a given string of memory accesses known as **reference string**
- Algorithms which produces less page fault then those algorithms are called efficient algorithms

Page replacement

- Reference string can be generated in several ways. Most commonly used 3 methods are
  1) by using random generators
  2) by using specifically designed sequences
  3) by tracing the given system & there by recording the addresses of each memory reference
- To find number of page faults for a particular reference string & page replacement algorithm, we should know the number of available frames.
- If the number of free frames are more, then number of page faults were less.
- Page faults will be more when less number of free frames
- There are several page replacement algorithm & for all of them we are using same reference string 7,0,1,2,0,3,0,4,2,3,0,3,2,1,2,0,1,7,0,1
- For all page replacement algorithms we consider memory with 3 frames

FIFO Page replacement

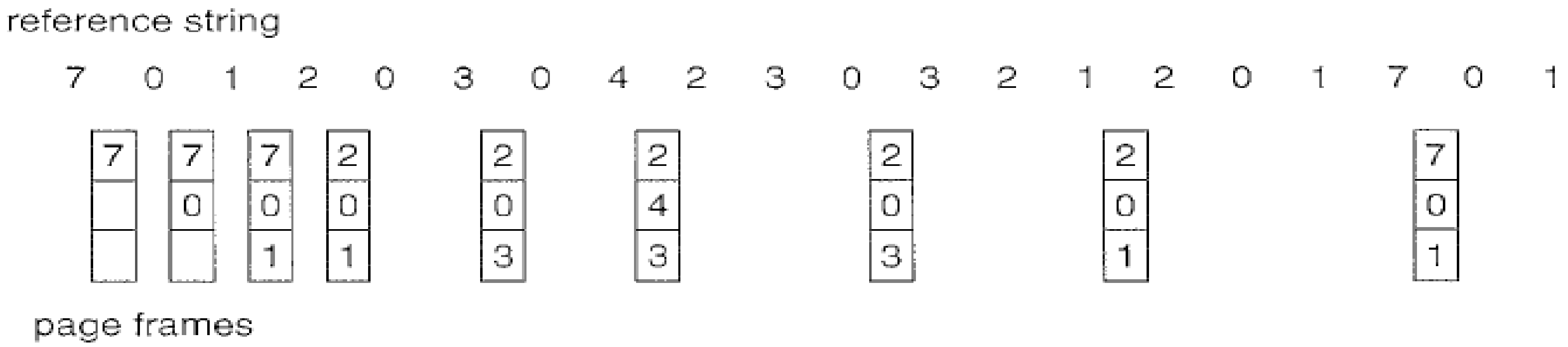
- This algorithm attaches time(page is brought into memory) with every page.
- When a page is to be replaced, it selects & replaces the oldest page(which came first)



- This algorithm is easy to understand & also to implement
- But its performance is not always good

Optimal Page replacement

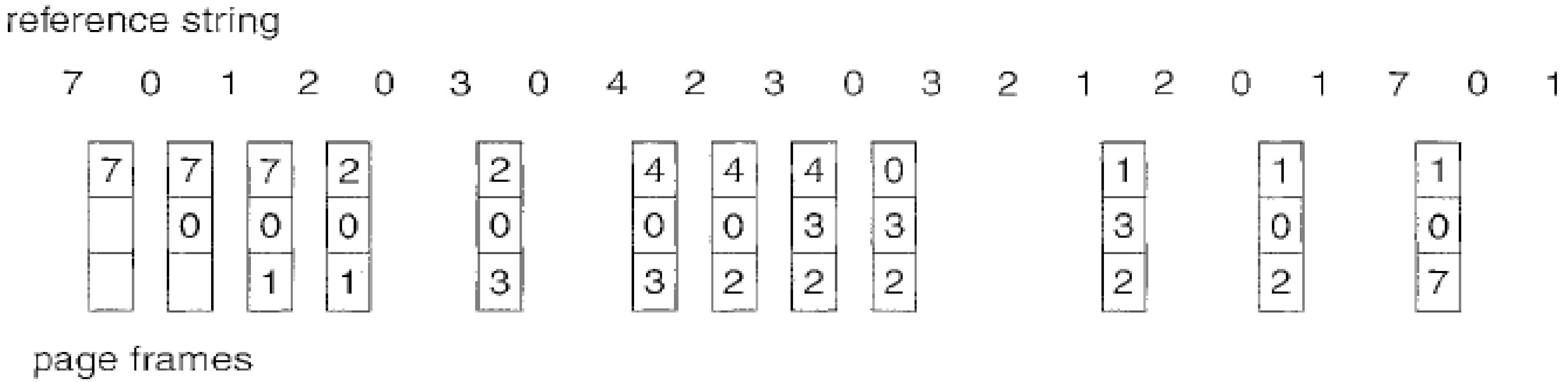
- This algorithm has lowest page fault rate.
- It never suffer from Belady's anomaly.
- This algorithm says "replace the page which is not going to be used in future for long time"
- It guarantees the lowest possible page fault rate for a fixed number of frames



- 9 page faults, first 3 are unavoidable, we can say only 6 page faults
- Difficult to implement because it requires future knowledge of reference string

LRU Page replacement

- Optimal page replacement algorithm looks time in forward, whereas LRU page replacement algorithm looks time backward
- It says, “replace the page which is not being used from long time”
- It replaces the page which is not used most recently, **Least Recently Used**.



- 12 page faults
- It is often used as a page replacement algorithm & is considered to be good

LRU Page replacement

- The major problem is how to implement exactly.
- Generally, there are 2 approaches to implement LRU
  1) counters
  2) stack
- **Counter** is used for every page & is incremented when corresponding page is accessed
- In order to find which page is least recently used, just search the entire page table for the page with the smallest counter value
- Whenever a page is accessed, it will be placed at the top of the **stack**. Always, least recentlu used page will be in the bottom
- Since, it requires removing the objects from middle of the stack, doubly linked list is the recommended data structure
- Both implementation of LRU requires hardware support, either for incrementing counter or for managing stack, as these opeartions must be performed for every memory access

Thrashing

- If any process does not have enough frames than what it actually, then it results in page fault very quickly
- Assume a process needs 12 frames but it got only 5 frames
- Whenever it tries to access 6th frame, page fault occurs
- At this point of time, this process has to replace any one of its 5 frames only
- So, withoutany choice, that process will replace any one of its frequently used frames
- If that process needs that frame again, then page-fault occurs again. Like this page fault will appear again and again
- This high paging activity is called **thrashing**
- A process will be in thrashing, if it is spending more time in paging than executing

Thrashing

- As the degree of multiprogramming increases, then CPU utilization will be in its peak till stack is full/main memory is full(i.e., all frames are busy in executing many processes)
- once main memory is full, if we increase the degree of multiprogramming slightly then system performance will decrease drastically

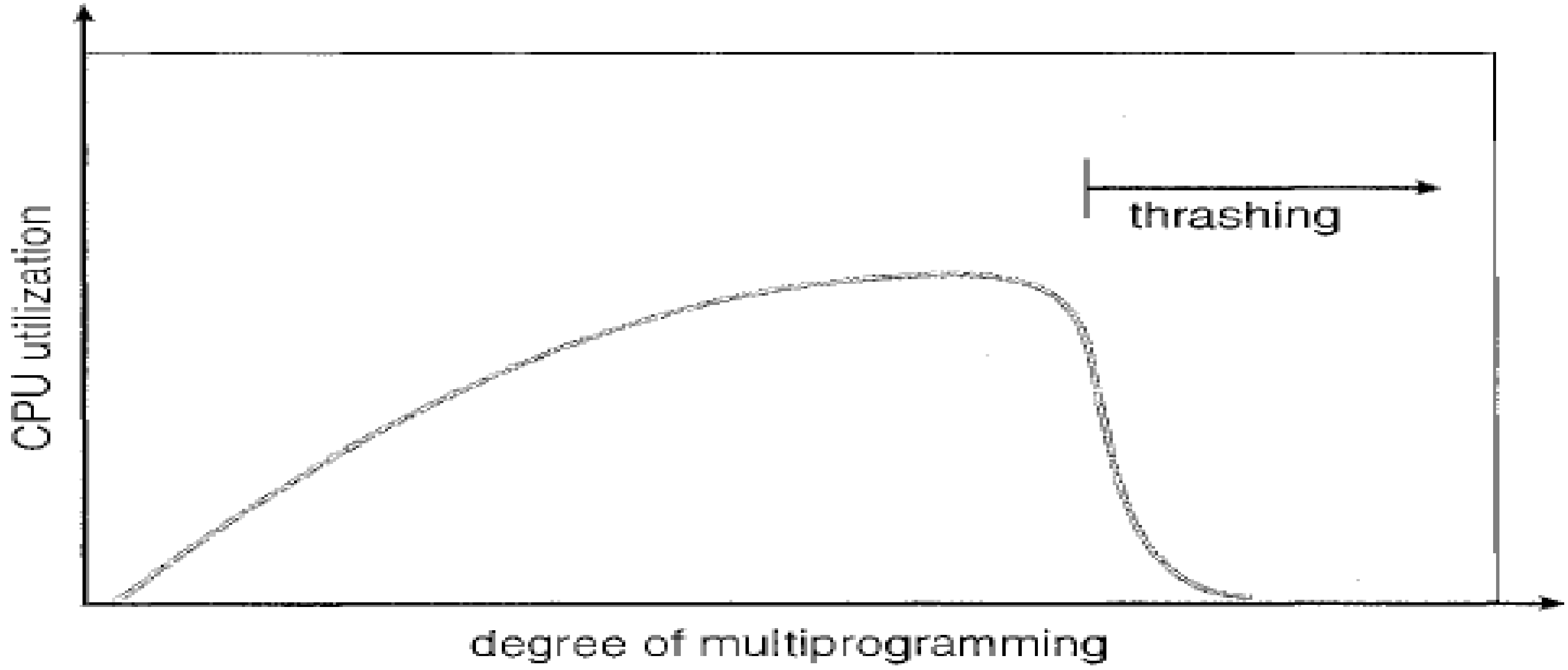


Figure 9.18 Thrashing.