Operating Systems (UNIT-4)

Memory Management - Objectives

To provide a detailed description of various ways of organizing memory hardware

To discuss various memory-management techniques, including paging and segmentation

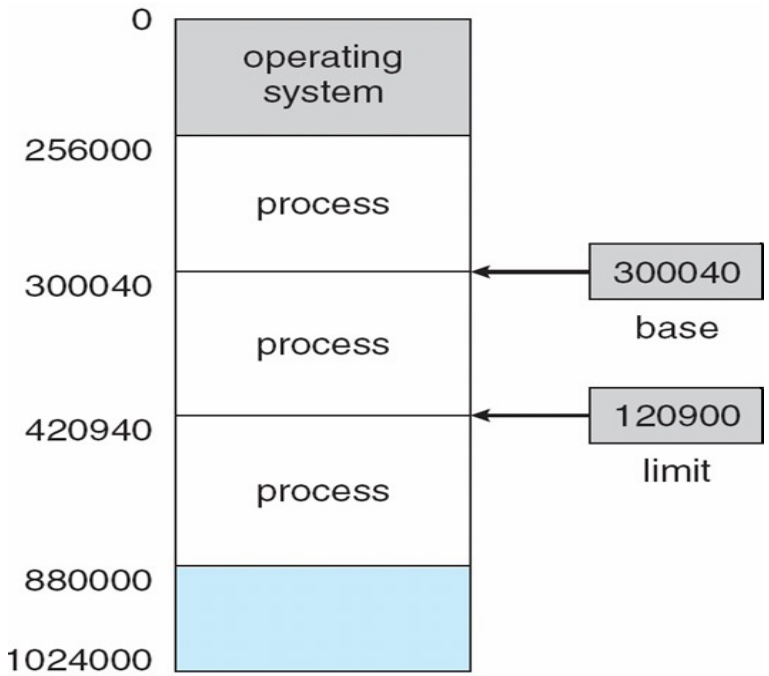
To provide a detailed description of the Intel Pentium, which supports both pure segmentation and segmentation with paging

Background

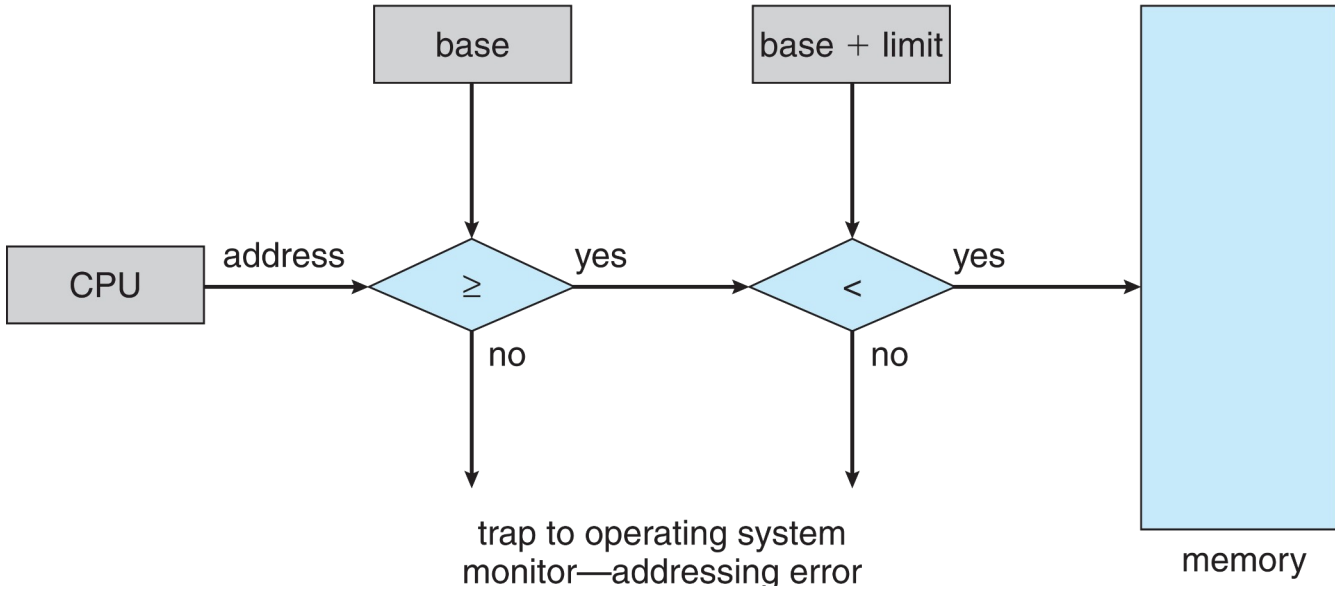
- Program must be brought (from disk) into memory and placed within a process for it to be run
- Main memory and registers are only storage CPU can access directly
- Memory unit only sees a stream of addresses + read requests, or address + data and write requests
- Register access in one CPU clock (or less).
- Main memory can take many cycles, causing a stall.
- Cache sits between main memory and CPU registers.
- Protection of memory required to ensure correct operation.

Base and Limit Registers

- A pair of **base** and **limit registers** define the logical address space.
- The **base register** holds the smallest legal physical memory address; the **limit register** specifies the size of the range.
- For example, if the base register holds300040 and the limit register is 120900, then the program can legally access all addresses from 300040 through 420939 (inclusive)



Hardware Address Protection



Protection of memory space is accomplished by having the CPU hardware compare every address generated in user mode with the registers.

Any attempt by a program executing in user mode to access operating-system memory or other users' memory results in a trap to the operating system, which treats the attempt as a **fatal error**

Address Binding

When you write the program it will be having the variables, each variables will be having the address

Most systems allow a user process to reside in any part of the physical memory.

Thus, although the address space of the computer may start at 00000,the first address of the user process need not be 00000.

In most cases, a user program goes through several steps—some of which may be optional—before being executed

Further, addresses represented in different ways at different stages of a program's life

Source code addresses usually symbolic

Compiled code addresses bind to re locatable addresses (Complier will not be knowing where the program is going to start or end, so it will just give the relative address)
i.e. " This variable is 14 bytes from beginning of this module" (then the generated compiler code will start at that location and extend up from there)

Linker or loader will bind relocatable addresses to absolute addresses (Because loader knows where it is going to load)

i.e. 74014

Each binding maps one address space to another

Binding of Instructions and Data to Memory

Address binding of instructions and data to memory addresses can happen at three different stages

- **Compile time**: If memory location known a priori, **absolute code** can be generated; must recompile code if starting location changes, Must generate **relocatable code** if memory location is not known at compile time
- **Load time**: Absolute address are binded
- **Execution time**: Binding delayed until run time if the process can be moved during its execution from one memory segment to another
  - Need hardware support for address maps (e.g., base and limit registers)

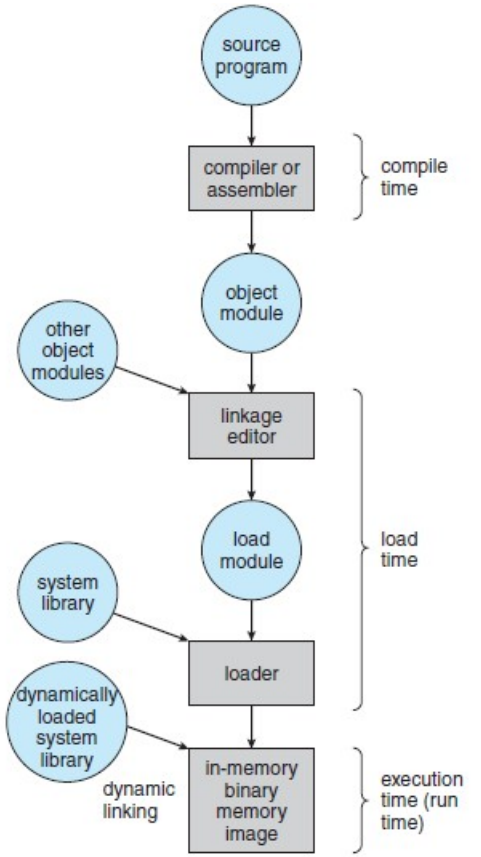


Figure 8.3 Multistep processing of a user program.

Logical vs. Physical Address Space

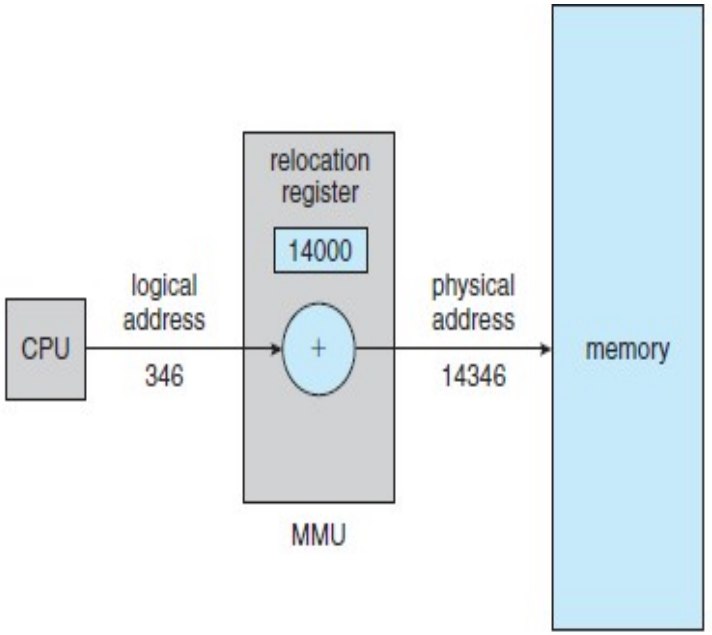
- The concept of a logical address space that is bound to a separate **physical address space** is central to proper memory management
  - **Logical address** – generated by the CPU; also referred to as **virtual address**
  - **Physical address** – address seen by the memory unit
- Logical and physical addresses are the same in compile-time and load-time address-binding schemes; logical (virtual) and physical addresses differ in execution-time address-binding scheme
- **Logical address space** is the set of all logical addresses generated by a program
- **Physical address space** is the set of all physical addresses generated by a program

Memory-Management Unit (MMU)

- MMU is the Hardware device that at **run time maps virtual to physical address**
- To start, consider simple scheme where the value in the relocation register(Base Register) is added to every address generated by a user process at the time it is sent to memory
- Eg: MS-DOS on Intel 80x86 used 4 relocation registers
- Like:(Code, Data, Stack and Extra Segment)
- Each of these relocation register have Base

Address

Physical Address : LA+Base register



Static and Dynamic Linking

Static

At the time of compilation, the complete programs will be compiled and linked without leaving any external program or module dependency.

Dynamic Linking

your compiler will compile the program and for all the modules which you want to include dynamically, only references will be provided and rest of the work will be done at the time of execution.

Dynamic Linking

- **Dynamically linked libraries** are system libraries that are linked to user programs when the programs are run.
- Dynamic linking –linking postponed until execution time
- Small piece of code, **stub**, used to locate the appropriate memory-resident library routine.
- Stub replaces itself with the address of the routine, and executes the routine
- Operating system checks if routine is in processes' memory address
  - If not in address space, add to address space
- Dynamic linking is particularly useful for libraries

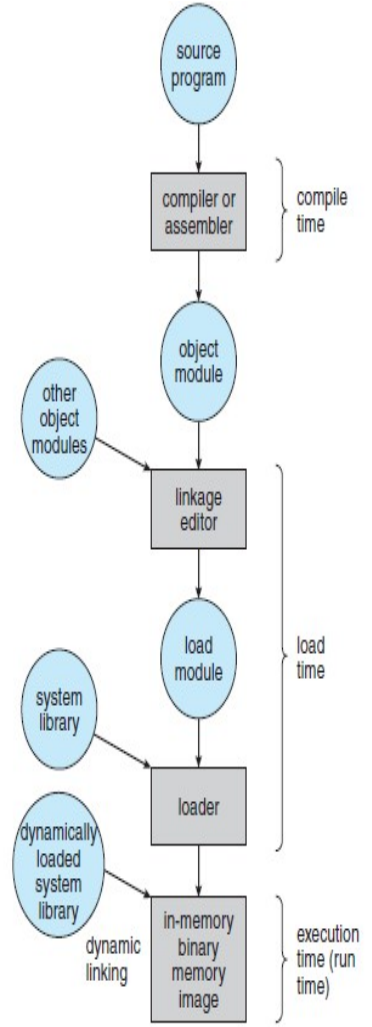
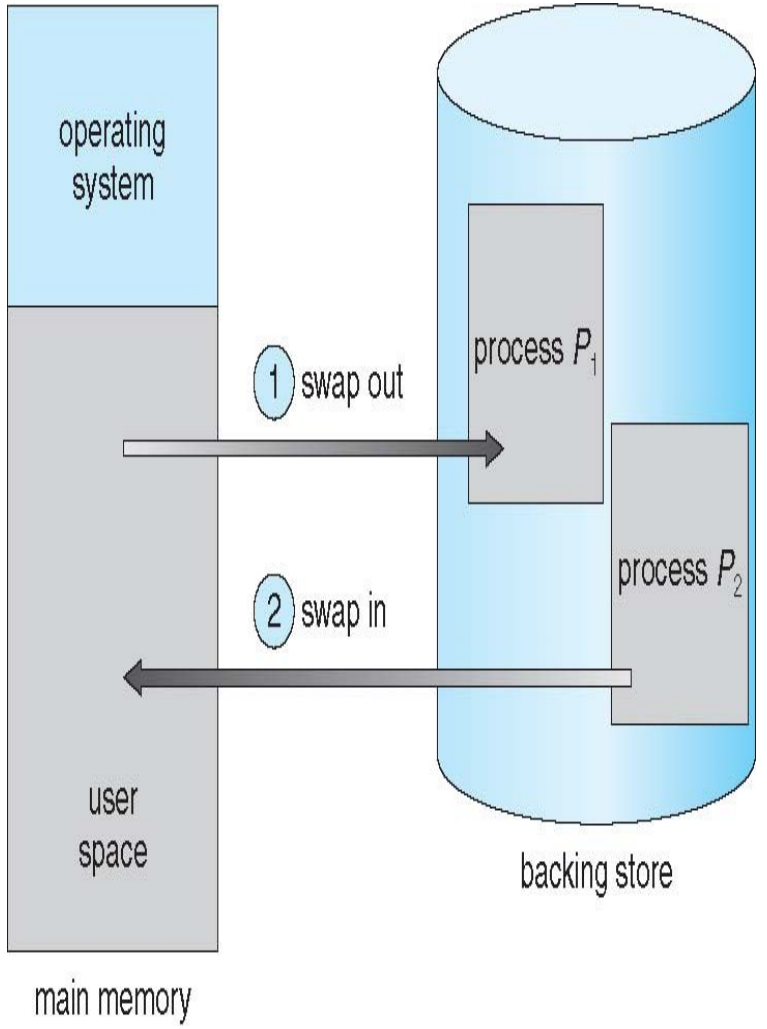


Figure 8.3 Multistep processing of a user program.

Swapping

- A process can be **swapped** temporarily out of memory to a backing store, and then brought back into memory for continued execution
- **Backing store** – fast disk large enough to accommodate copies of all memory images for all users; must provide direct access to these memory images
- **Roll out, roll in** – swapping variant used for priority-based scheduling algorithms; lower-priority process is swapped out so higher-priority process can be loaded and executed
- Major part of swap time is transfer time; total transfer time is directly proportional to the amount of memory swapped
- System maintains a **ready queue** of ready-to-run processes which have memory images on disk



Swapping

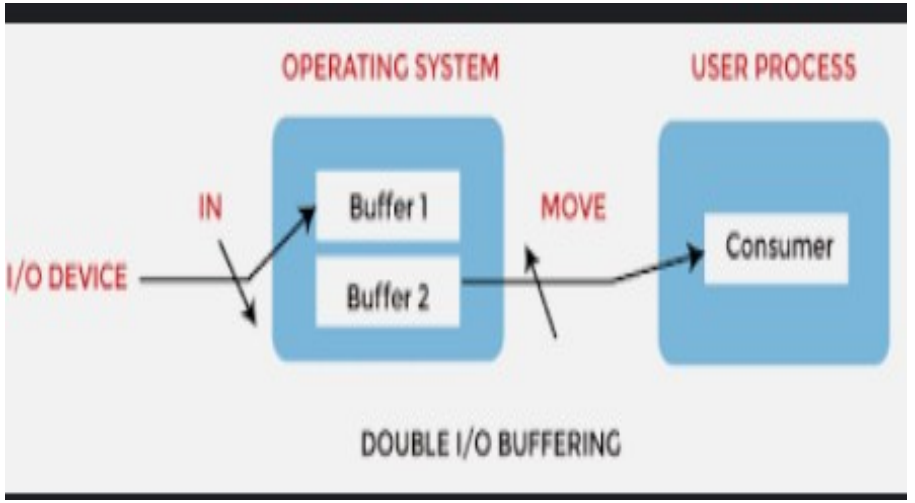
- Does the swapped out process need to swap back in to same physical addresses?
- You cannot just swap out the process and put that into same memory
- Depends on address binding method
- Plus consider pending I/O to from process memory space.(If some process is using the I/O devices, if it is swapped at that time, I/O devices may be yused by another process)
- Modified versions of swapping are found on many systems (i.e., UNIX, Linux, and Windows)
  - Swapping normally disabled
  - Started if more than threshold amount of memory allocated
  - Disabled again once memory demand reduced below threshold

Context Switch Time including Swapping

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

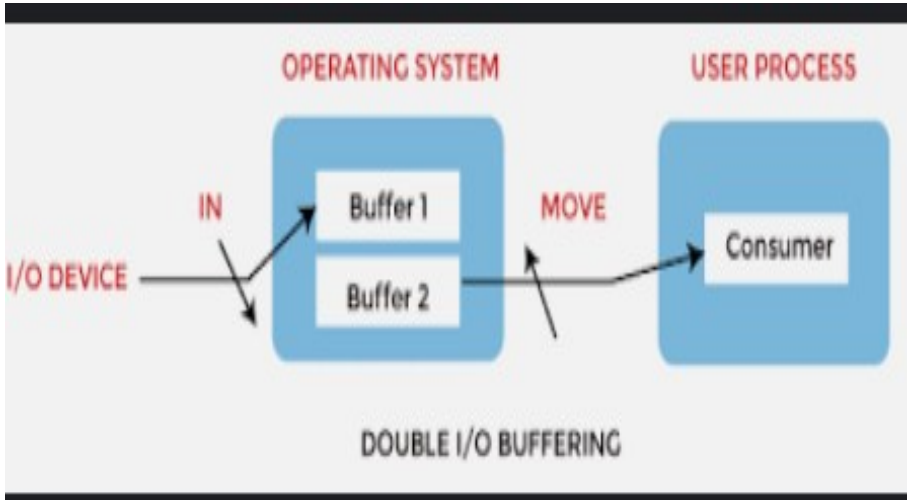
Context Switch Time including Swapping

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



Context Switch Time including Swapping

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

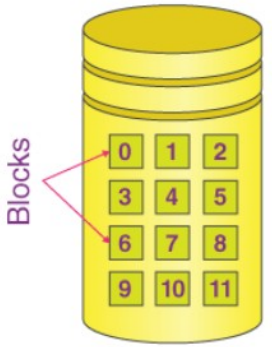


Swapping on Mobile Systems

- Swapping Not typically supported in Mobile Systems.
  - Mobiles are Flash memory based, which are
    - Small amount of space
    - Limited number of write cycles
    - Poor throughput between flash memory and CPU on mobile platform
- Instead use other methods to free memory if low
  - iOS ***asks*** apps to voluntarily relinquish allocated memory
    - Read-only data thrown out and reloaded from flash if needed
    - Failure to free can result in termination of app
  - Android terminates apps if low free memory, but first writes **application state** to flash for fast restart

Contiguous Allocation

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



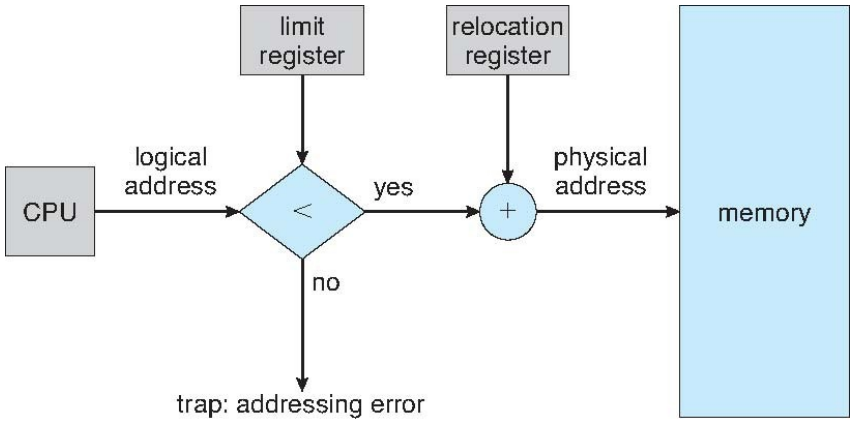
File Name	Start	Length	Allocated Blocks
abc.text	0	3	0, 1, 2
video.mp4	4	2	4, 5
jtp.docx	9	3	9, 10, 11

Directory

Contiguous Allocation

Contiguous Allocation

- Once the allocation is done we need to map back the address
- MMU does that job - maps the logical address *dynamically*



  - CPU generates the Logical address
  - Limit register contains range of logical addresses – each logical address must be less than the limit register , if yes it will be added to relocation register , the physical address is generated

Multiple-partition allocation

Multiple-partition allocation

Degree of multiprogramming limited by number of partitions

Variable-partition sizes for efficiency (sized to a given process' needs)

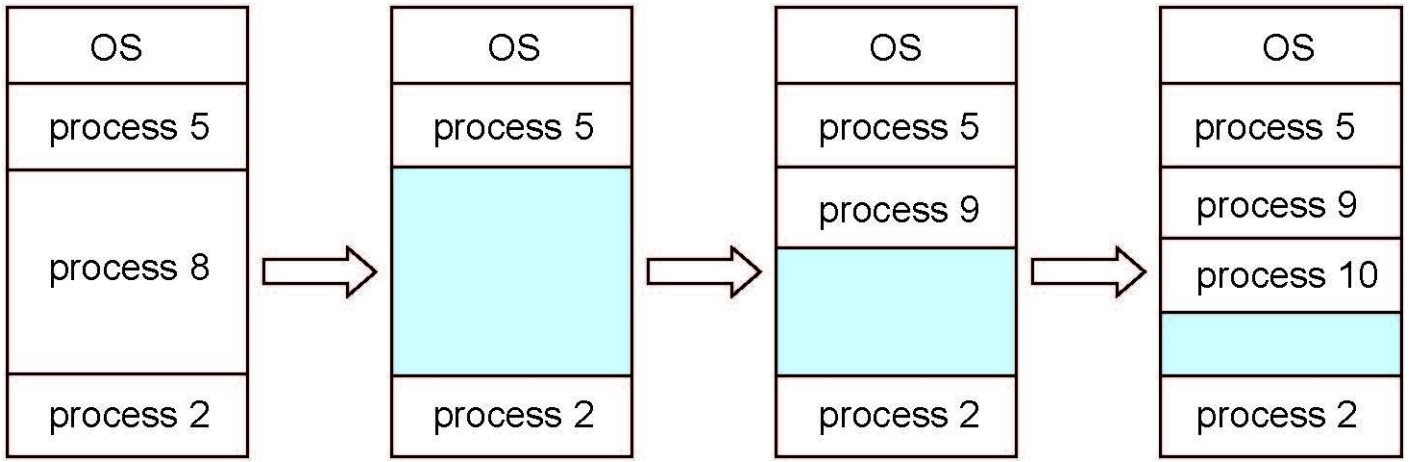
Hole – block of available memory; holes of various size are scattered throughout memory

When a process arrives, it is allocated memory from a hole large enough to accommodate it

Process exiting frees its partition, adjacent free partitions combined

Operating system maintains information about:

a) allocated partitions b) free partitions (hole)



Dynamic Storage-Allocation Problem

- **First-fit**: Allocate the ***first*** hole that is big enough
- **Best-fit**: Allocate the ***smallest*** hole that is big enough; must search entire list, unless ordered by size
  - Produces the smallest leftover hole
- **Worst-fit**: Allocate the ***largest*** hole; must also search entire list
  - Produces the largest leftover hole

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

Dynamic Storage-Allocation Problem

Consider six memory partitions of size 200 KB, 400 KB, 600 KB, 500 KB, 300 KB and 250 KB. These partitions need to be allocated to four processes of sizes 357 KB, 210 KB, 468 KB and 491 KB in that order.

Perform the allocation of processes using-

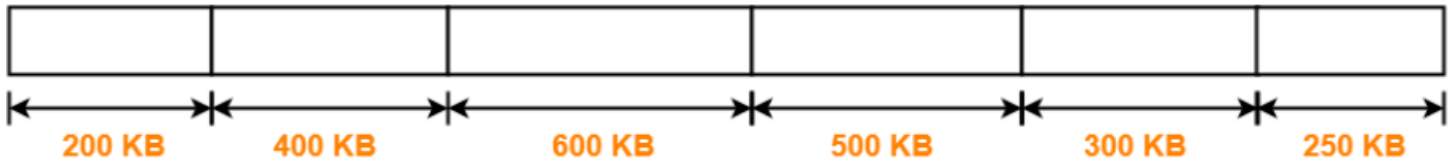
1.First Fit Algorithm

2.Best Fit Algorithm

3.Worst Fit Algorithm

Solution

The main memory has been divided into fixed size partitions as-



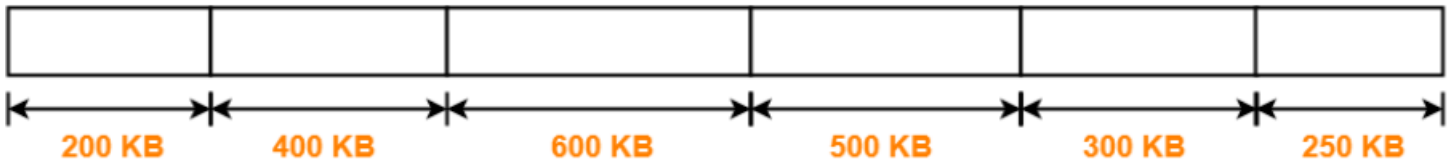
Dynamic Storage-Allocation Problem

1.First Fit Algorithm

Algorithm starts scanning the partitions serially.
When a partition big enough to store the process is found, it allocates that partition to the process

Solution

The main memory has been divided into fixed size partitions as-



Process P1 = 357 KB
Process P2 = 210 KB
Process P3 = 468 KB
Process P4 = 491 KB

Dynamic Storage-Allocation Problem

Step-01:

Step-02:

Step-03:

- Process P4 can not be allocated the memory.
- This is because no partition of size greater than or equal to the size of process P4 is available.

Dynamic Storage-Allocation Problem

2. Best Fit Algorithm

In Best Fit Algorithm,
Algorithm first scans all the partitions.
It then allocates the partition of
smallest size that can store the process.

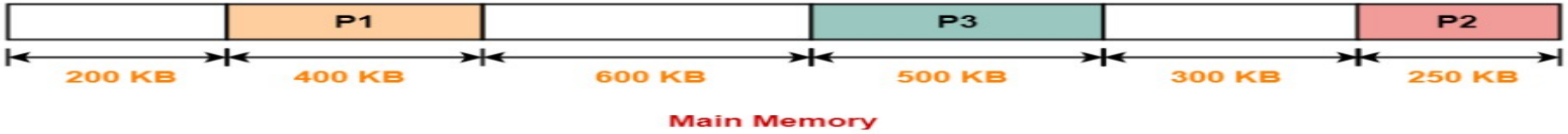
Solution ------

Process P1 = 357 KB
Process P2 = 210 KB
Process P3 = 468 KB
Process P4 = 491 KB

Step-01:

Step-02:

Step-03:



Step-04:

Dynamic Storage-Allocation Problem

3. Worst Fit Algorithm

In Worst Fit Algorithm,
Algorithm first scans all the partitions.
It then allocates the partition of largest size to the process.

The allocation of partitions to the given processes is shown below

Step-01:

Solution ------

Step-02:

Process P1 = 357 KB
Process P2 = 210 KB
Process P3 = 468 KB
Process P4 = 491 KB

Process P3 and Process P4 can not be allocated the memory.
This is because no partition of size greater than or equal to the size of process P3 and process P is available.

Fragmentation

As processes are loaded and removed from memory, the free memory space is broken into little pieces.

It happens after sometimes that processes cannot be allocated to memory blocks considering their small size and memory blocks remains unused. This problem is known as **Fragmentation.**

Internal Fragmentation
Memory block assigned to process is bigger. Some portion of memory is left unused, as it cannot be used by another process.
The internal fragmentation can be reduced by effectively assigning the smallest partition but large enough for the process

External Fragmentation
Total memory space is enough to satisfy a request or to reside a process in it, but it is not contiguous, so it cannot be used.

External fragmentation can be reduced by compaction or shuffle memory contents to place all free memory together in one large block

Fragmentation

Sr. No.	Key	Internal Fragmentation	External Fragmentation
1	Definition	When there is a difference between required memory space vs allotted memory space, problem is termed as Internal Fragmentation.	When there are small and non-contiguous memory blocks which cannot be assigned to any process, the problem is termed as External Fragmentation.
2	Memory Block Size	Internal Fragmentation occurs when allotted memory blocks are of fixed size.	External Fragmentation occurs when allotted memory blocks are of varying size.
3	Occurrence	Internal Fragmentation occurs when a process needs more space than the size of allotted memory block or use less space.	External Fragmentation occurs when a process is removed from the main memory.
4	Solution	Best Fit Block Search is the solution for internal fragmentation.	Compaction is the solution for external fragmentation.
5	Process	Internal Fragmentation occurs when Paging is employed.	External Fragmentation occurs when Segmentation is employed.

Segmentation

Segmentation is a memory management technique in which the memory is divided into the variable size parts. Each part is known as a segment which can be allocated to a process.

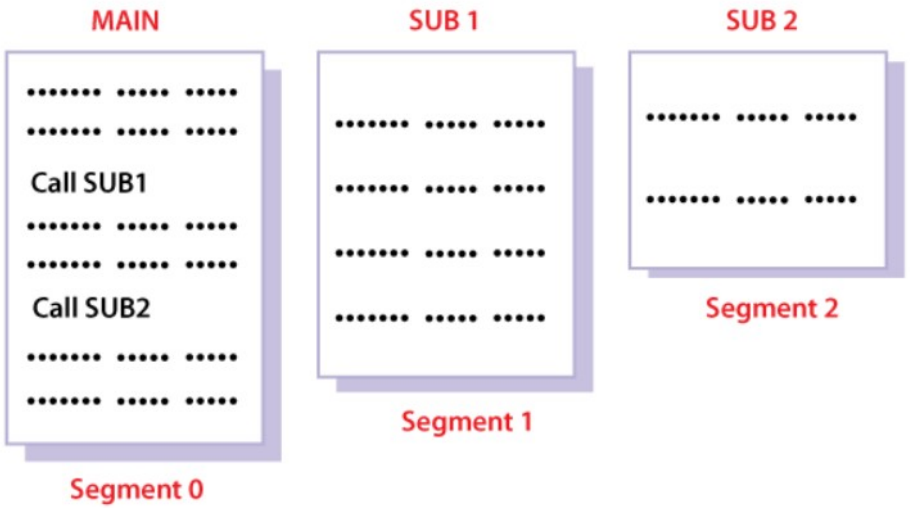
The details about each segment are stored in a table called a **segment table**.

Segment table is stored in one (or many) of the segments.

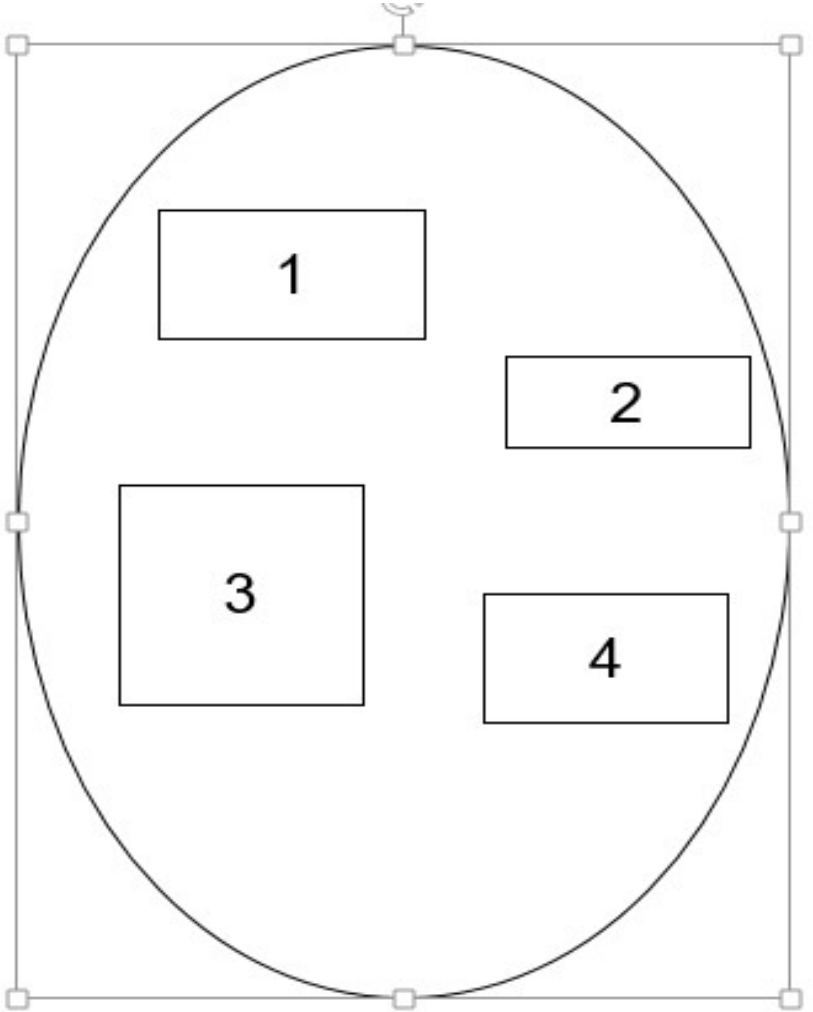
Each segment has a name and a length.

The addresses specify both the segment name and the offset within the segment.

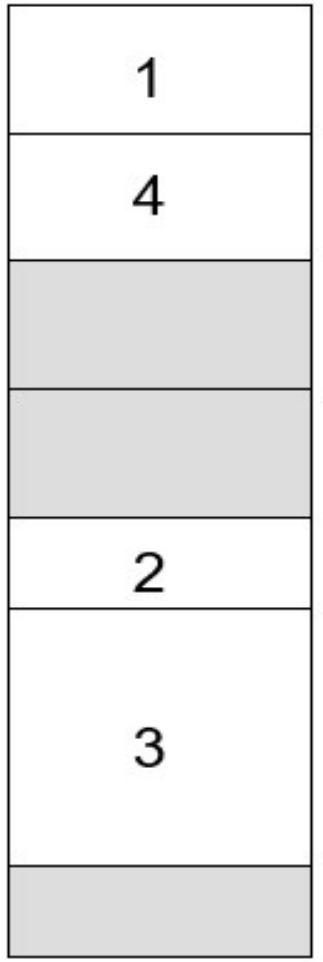
The programmer therefore specifies each address by two quantities: **a segment name and an offset**



Logical View of Segmentation



user space



physical memory space

Segmentation Architecture

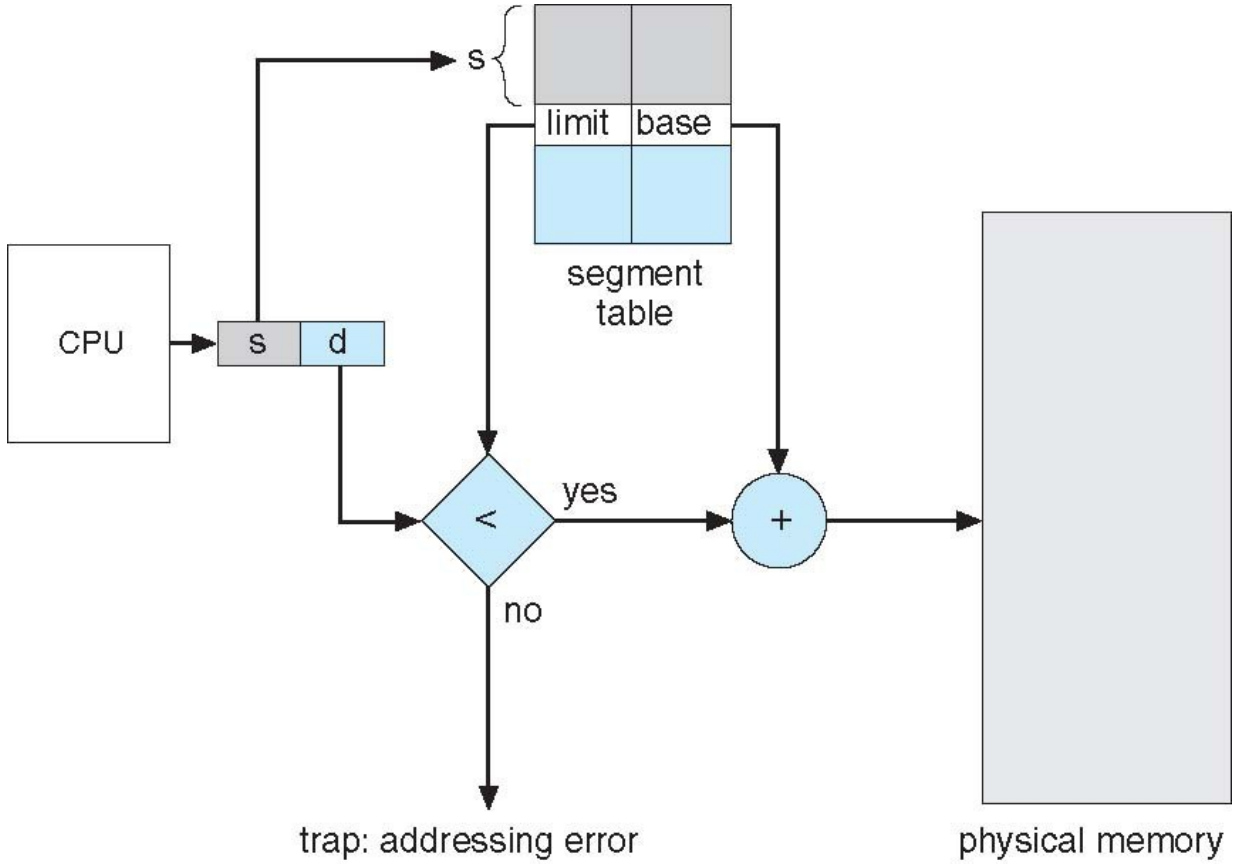
- Logical address consists of a two tuple:

  <segment-number, offset>,

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

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

Segmentation Hardware



A logical address consists of two parts: a segment number, s, and an offset into that segment, d.

The segment number is used as an index to the segment table.
The offset d of the logical address must be between 0 and the segment limit.

If it is not, we trap to the operating system (logical addressing attempt beyond end of segment).

When an offset is legal, it is added to the segment base to produce the address in physical memory of the desired byte

Paging

Paging is a memory management technique in which process address space is broken into blocks of the same size called pages.

Divide logical memory into blocks of same size called pages

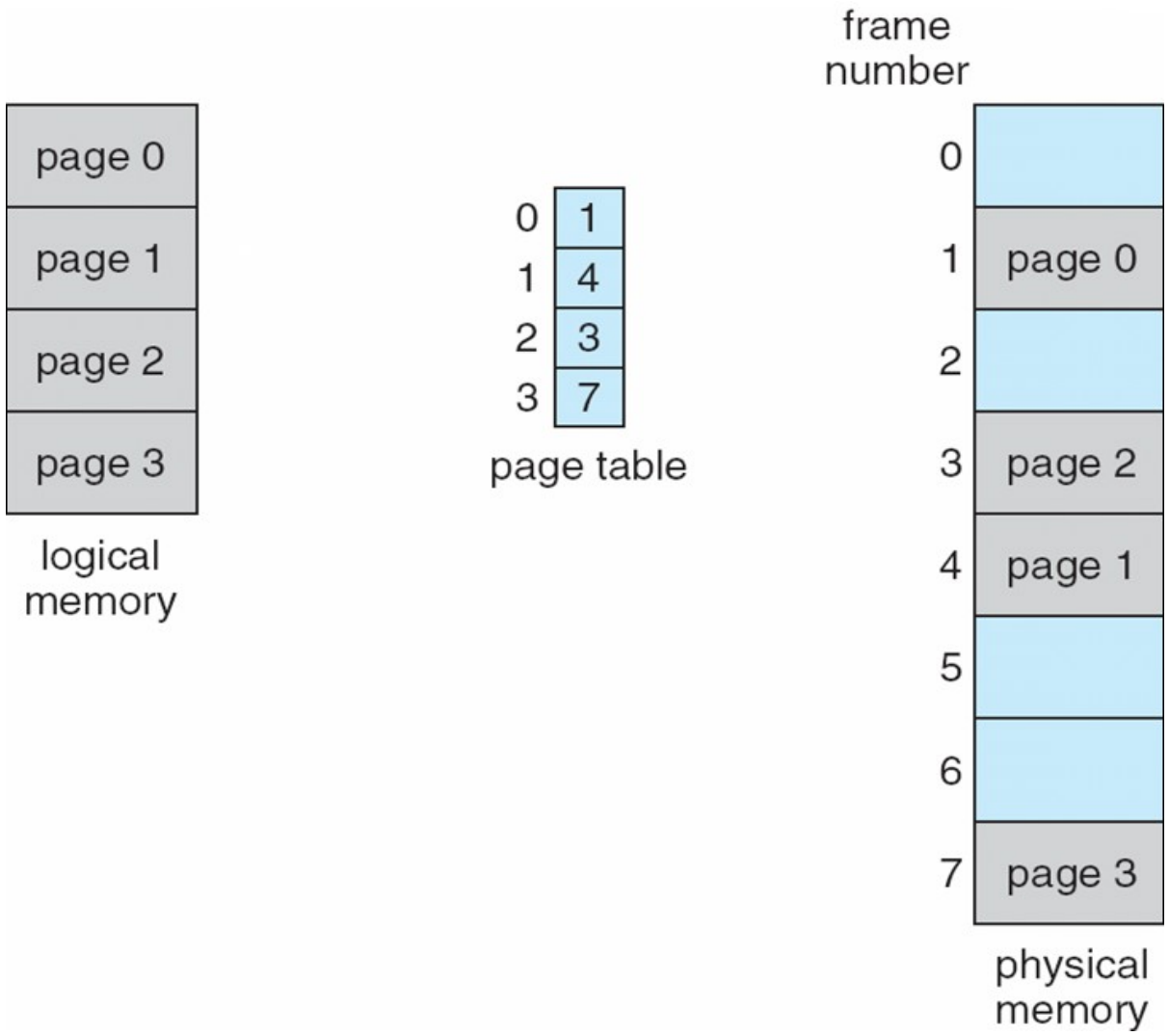
Divide physical memory (Main Memory) into fixed-sized blocks called frames

Size is power of 2, between 512 bytes and 16 Mbytes

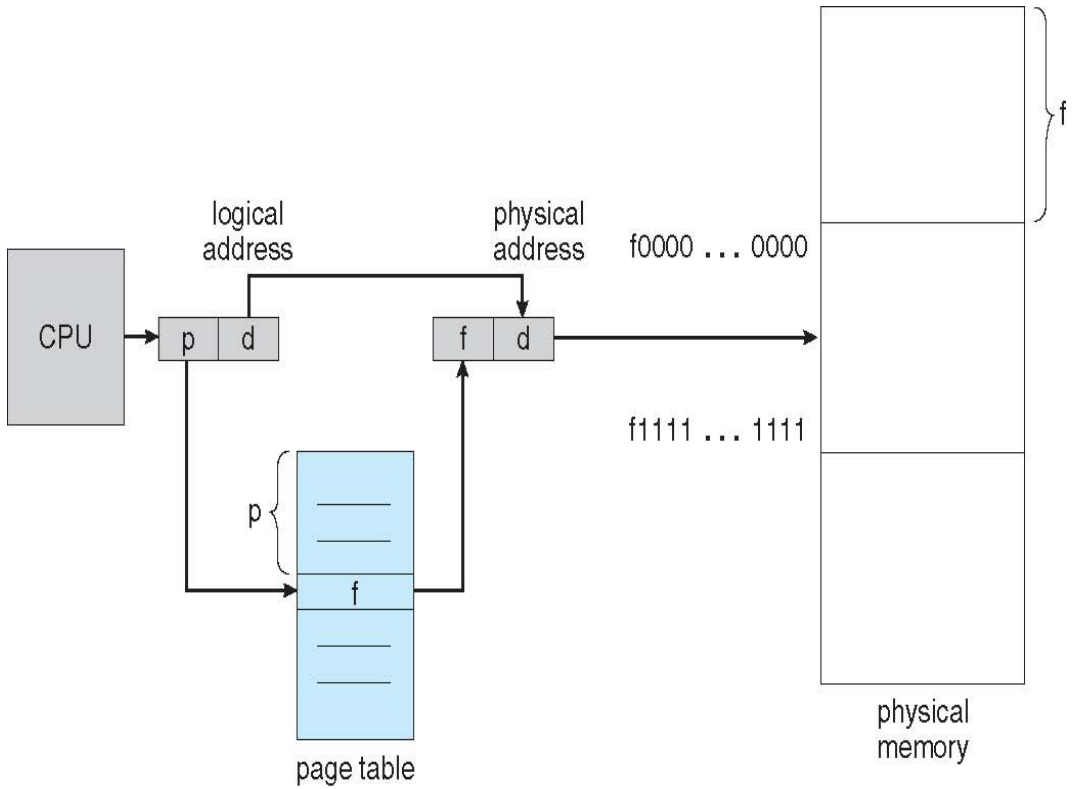
To run a program of size N pages, need to find N free frames and load program

Set up a **page table** to translate logical to physical addresses. (Page Table takes, page as input and frames as output)

Paging Model of Logical and Physical Memory



Paging Hardware



Logical Address generated by CPU is divided into:

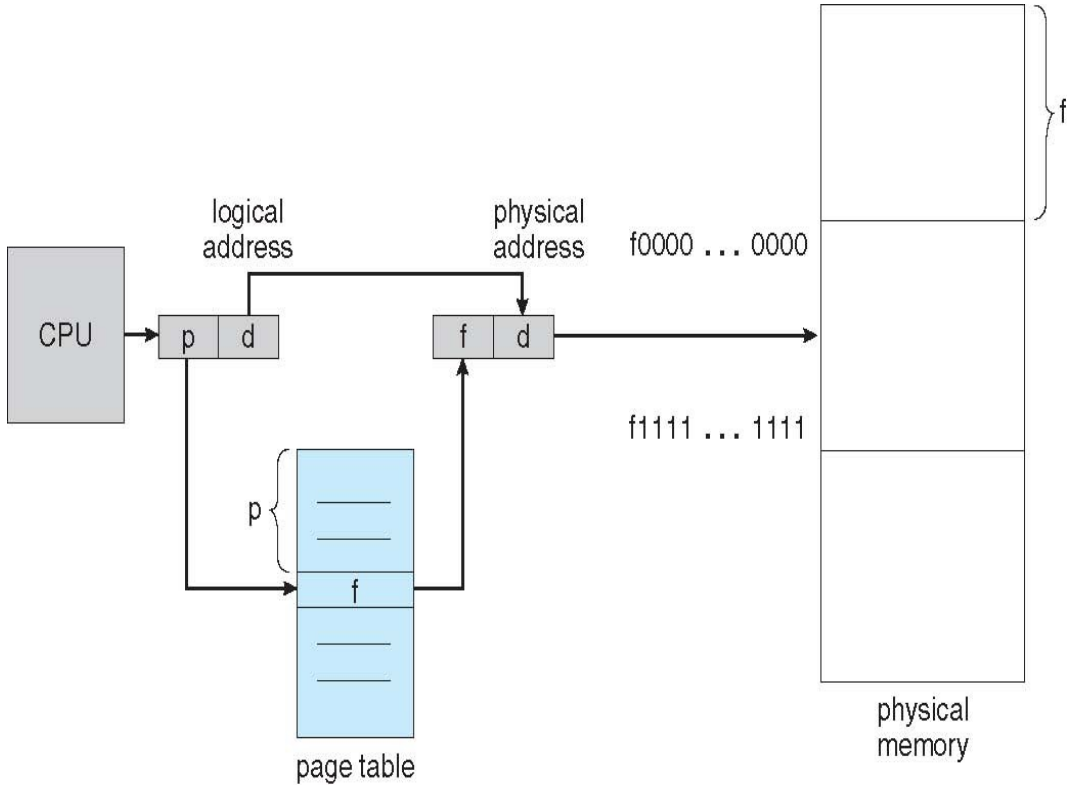
- **Page number** (*p*) – used as an index into a **page table** which contains base address of each page in physical memory
- **Page offset** (*d*) – combined with base address to define the physical memory address that is sent to the memory unit

The page number is given as input to the Page Table

The Page Table gives **frames** as the output

Then it will be combined with the offset.

Paging Hardware



Logical Address generated by CPU is divided into:

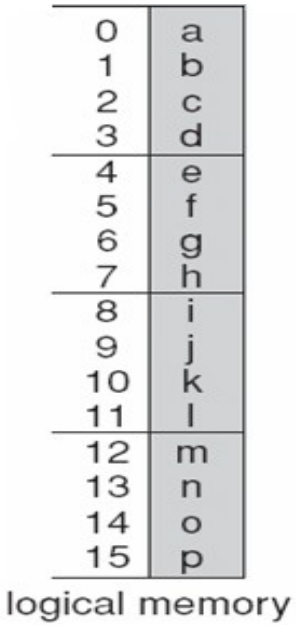
- **Page number** (*p*) – used as an index into a **page table** which contains base address of each page in physical memory
- **Page offset** (*d*) – combined with base address to define the physical memory address that is sent to the memory unit

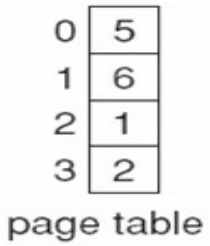
The page number is given as input to the Page Table

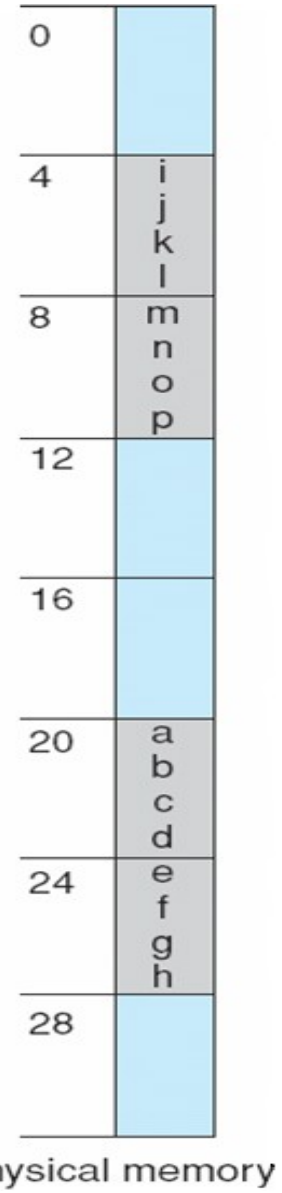
The Page Table gives **frames** as the output

Then it will be combined with the offset.

Paging Example with 32byte memory and 4 byte pages







Eg : Page Number 0 is stored in frame number 5

Logical address 0

Page 0, Offset 0

Frame*Length+Offset = 5*4+0 = 20, so a is stored at 20

Logical address 1

Frame*Length+Offset = 5*4+1 = 21, so b is stored at 21

Implementation of Page Table

Page table is kept in main memory

Page-table base register (**PTBR**) points to the page table

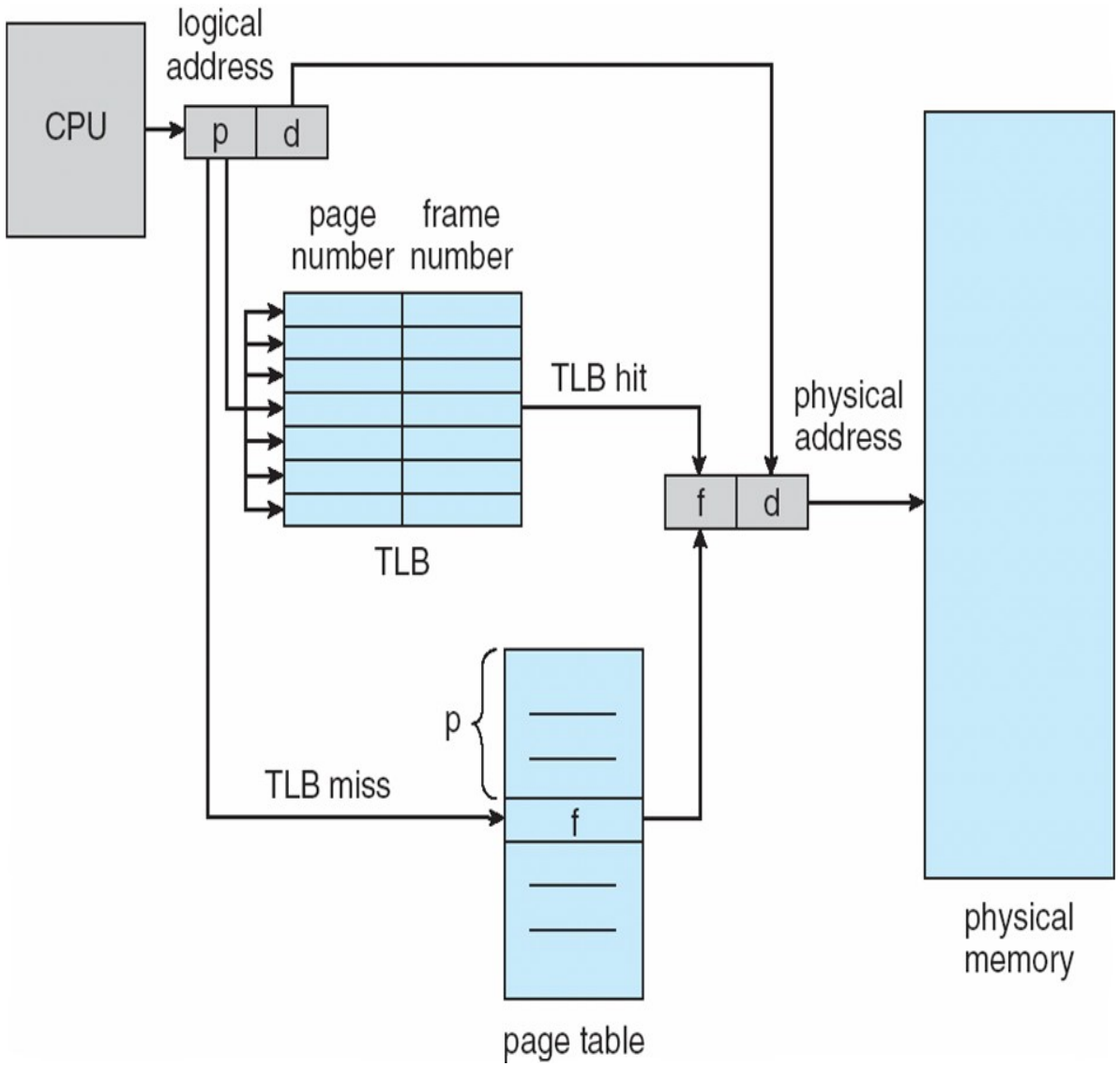
Page-table length register (**PTLR**) indicates size of the page table

In this scheme every data/instruction access requires **two memory accesses**

One for the page table and one for the data / instruction

The two memory access problem can be solved by the use of a special fast-lookup hardware cache called **associative memory** or **translation look-aside buffers** (**TLBs**)

Paging Hardware With TLB



Each entry in the TLB consists of two parts:
a key (or tag) and a value.

Key means page number and value means frame number.

TLB have data of frequently accessed pages

When the associative memory is presented with an item, the item is compared with all keys simultaneously.

If the item is found, the corresponding value field is returned.

P is given as input to the TLB, if the corresponding P is available in TLB, then the respective frame number is assigned (TLB HIT).

If the P is not found in TLB(TLB Miss), it will be stored in the page table, which resides in the main memory

Memory Protection

Memory protection implemented by associating protection bit with each frame to indicate if read-only or read-write access is allowed

Can also add more bits to indicate page execute-only, and so on

Valid-invalid bit attached to each entry in the page table:

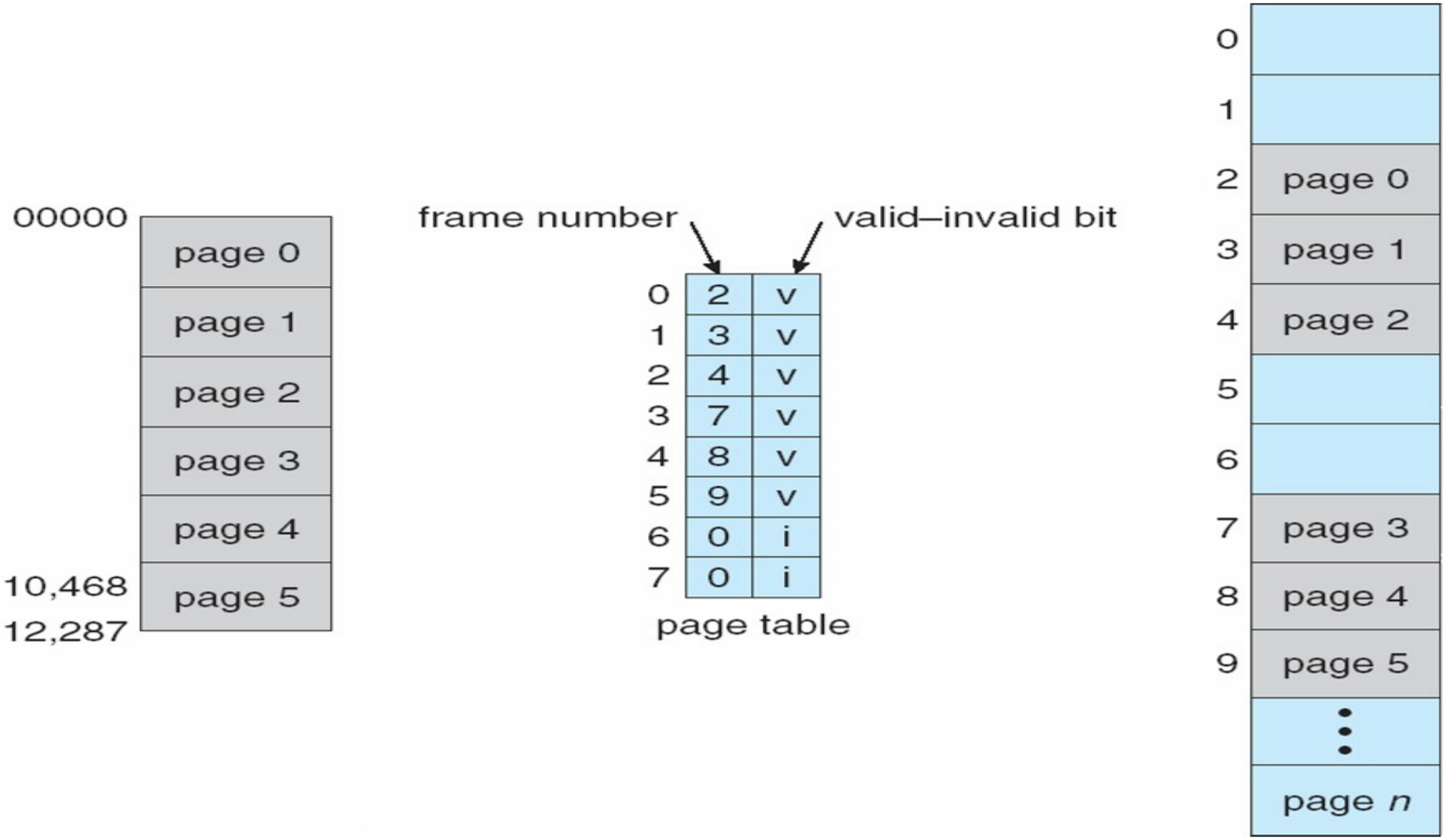
"valid" indicates that the associated page is in the process' logical address space, and is thus a legal page

"invalid" indicates that the page is not in the process' logical address space

Or use **page-table length register** (**PTLR**)

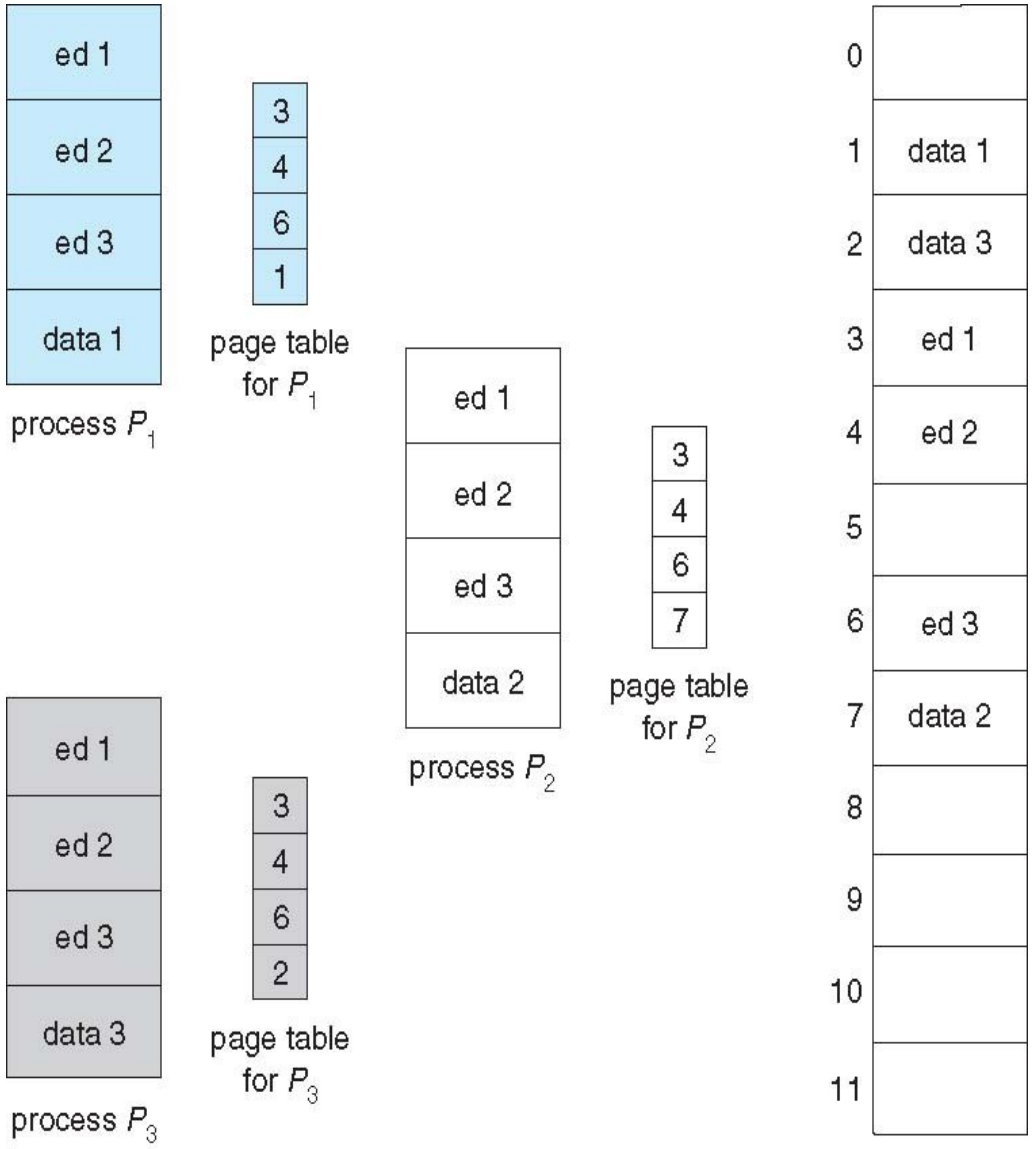
Any violations result in a trap to the kernel

Memory Protection



Shared Pages

- **Shared code**
  - **One copy of read-only** (**reentrant**) code shared among processes (i.e., text editors, compilers, window systems)
  - Similar to multiple threads sharing the same process space
  - Also useful for interprocess communication if sharing of read-write pages is allowed
- **Private code and data**
  - Each process keeps a separate copy of the code and data
  - The pages for the private code and data can appear anywhere in the logical address space



Structure of the Page Table

Memory structures for paging can get huge using straight-forward methods

- Consider a 32-bit logical address space as on modern computers
- Page size of 4 KB (2^{12})
- Page table would have 1 million entries (2^{32} / 2^{12}), it is huge data we cant keep that into the same page table.

There are 3 types of Structures of the Page Table

- Hierarchical Paging
- Hashed Page Tables
- Inverted Page Tables

Multilevel paging is a paging scheme where there exists a hierarchy of page tables.

Need –

The need for multilevel paging arises when-

The size of page table is greater than the frame size.

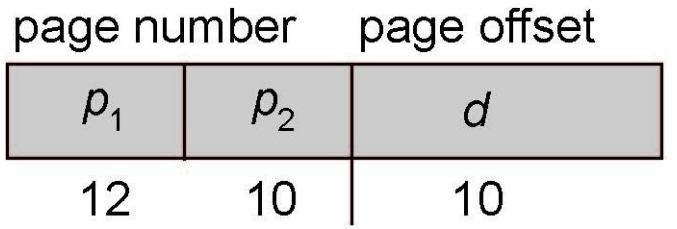
As a result, the page table can not be stored in a single frame in main memory.

Working-

- **The page table having size greater than the frame size is divided into several parts.**
- **The size of each part is same as frame size except possibly the last part.**
- **The pages of page table are then stored in different frames of the main memory.**
- **To keep track of the frames storing the pages of the divided page table, another page table is maintained.**
- **As a result, the hierarchy of page tables get generated.**
- **Multilevel paging is done till the level is reached where the entire page table can be stored in a single frame.**

Two-Level Paging Example

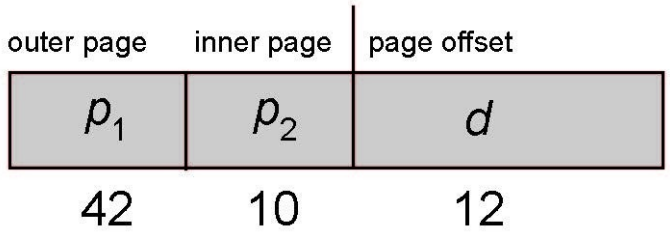
- A logical address (on 32-bit machine with 1K page size) is divided into:
  - a page number consisting of 22 bits
  - a page offset consisting of 10 bits
- Since the page table is paged, the page number is further divided into:
  - a 12-bit page number
  - a 10-bit page offset
- Thus, a logical address is as follows:



- where p_1 is an index into the outer page table, and p_2 is the displacement within the page of the inner page table
- Known as **forward-mapped page table**

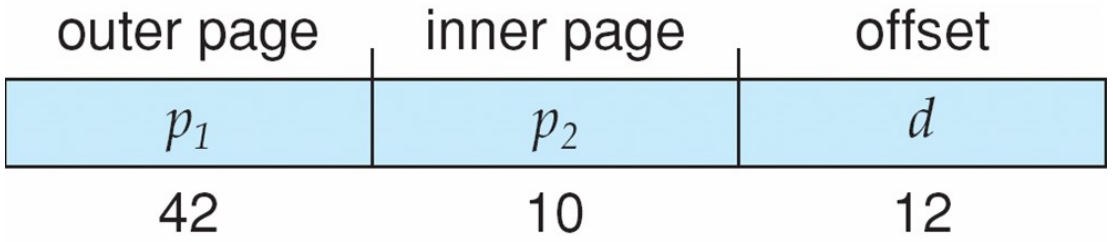
64-bit Logical Address Space

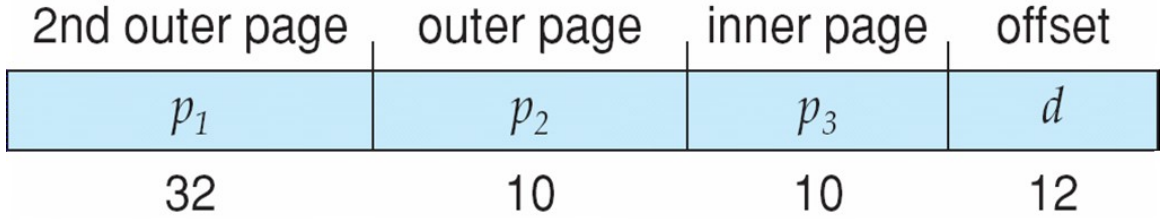
- Even two-level paging scheme not sufficient
- If page size is 4 KB (2^{12})
  - Then page table has 2^{52} entries
  - If two level scheme, inner page tables could be 2^{10} 4-byte entries
  - Address would look like



  - Outer page table has 2^{42} entries or 2^{44} bytes
  - One solution is to add a 2^{nd} outer page table
  - But in the following example the 2^{nd} outer page table is still 2^{34} bytes in size
    - And possibly 4 memory access to get to one physical memory location

Three-level Paging Scheme





Hashed Paging

This page table has a hash value being the virtual page number.

And each entry in the hash table contains a linked list of elements that hash to the same location (to handle collisions).

Each element consists of three fields:
(1) The virtual page number,
(2) The value of the mapped page frame, and
(3) A pointer to the next element in the linked list.

Address translation process for hash table:

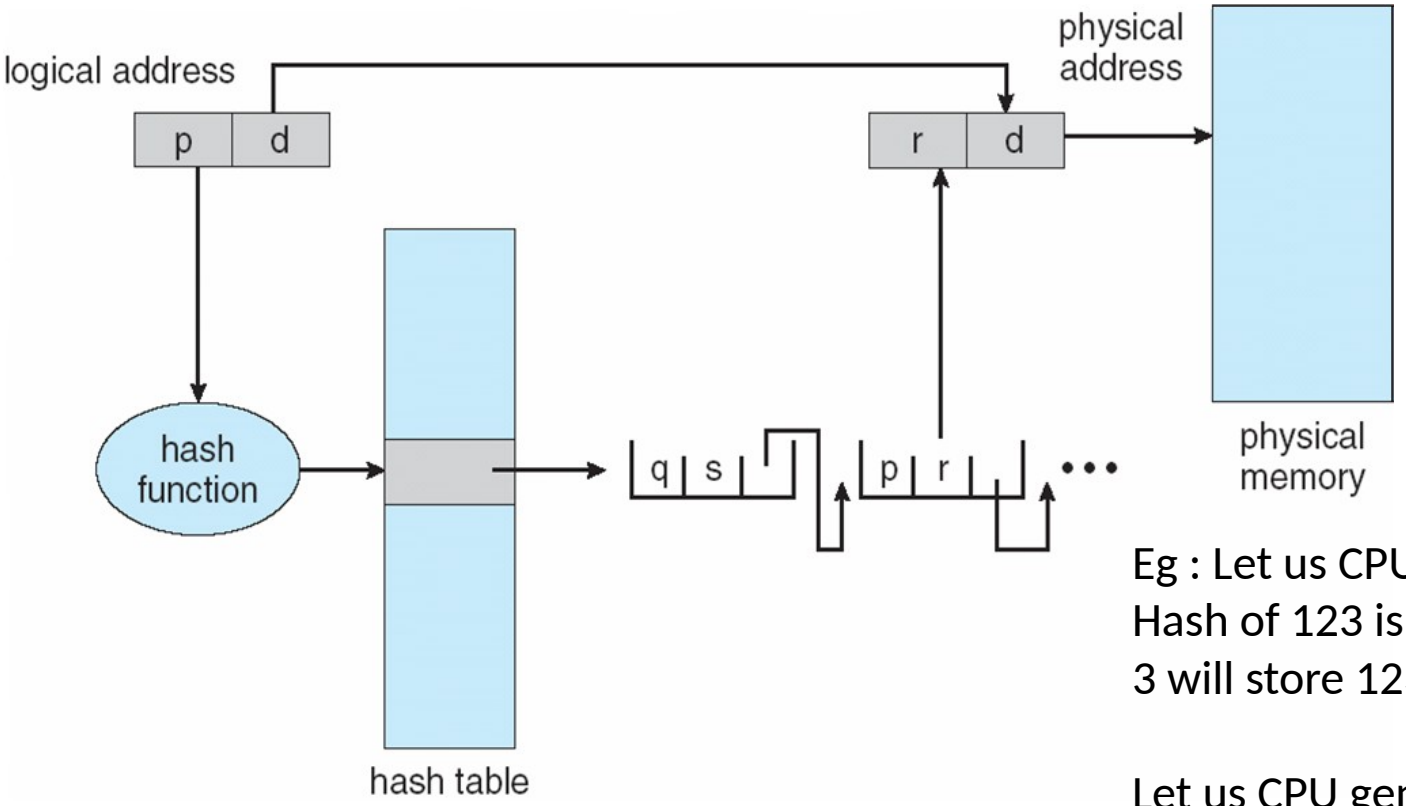
1. Page number in Logical Address (generated by CPU) is hashed into the hash table.

2. The page number of the logical address is compared with the first element (field 1: virtual page number) in the linked list.

If both page numbers are matched then retrieve the corresponding frame number (field 2 in the linked list) to from the physical address.

If both page numbers do not match then subsequent entries in the linked list searched for a matching virtual page number.

Hashed Page Table



Eg : Let us CPU generates the Logical Address 123
Hash of 123 is 3
3 will store 123(q),5(s),pointer to the next element

Let us CPU generates the Logical Address 893
Hash of 893 is 3
3 will store 893(q),5(s),pointer to the next element.

Now again when 893 is generated , it will searched, frame number will be taken from that combined with offset and mapped.

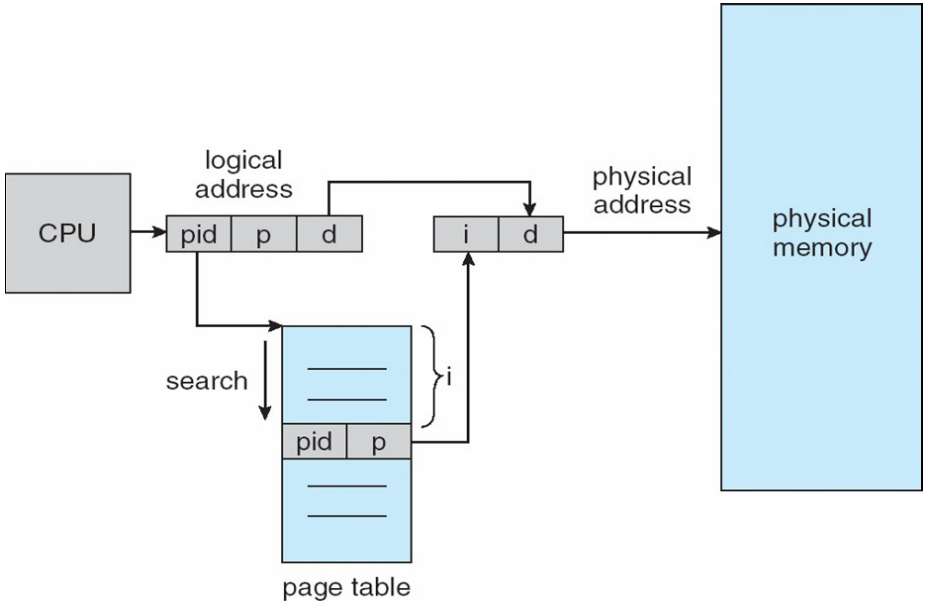
Inverted Page Table

Let us assume that we have a process of the 100 pages, but main memory have only 10 pages. (Only 10 frames are available)

The Page table of that process, will have 100 pages but among 100 pages, only 10 will get frames, so 90 will be unused

In order to overcome this problem we will use the concept of **Inverted Page Table.**

Inverted Page Table is the Global Page Table, it contains only that many number of pages as main memory. Rather than each process having a page table and keeping track of all possible logical pages, track all physical pages

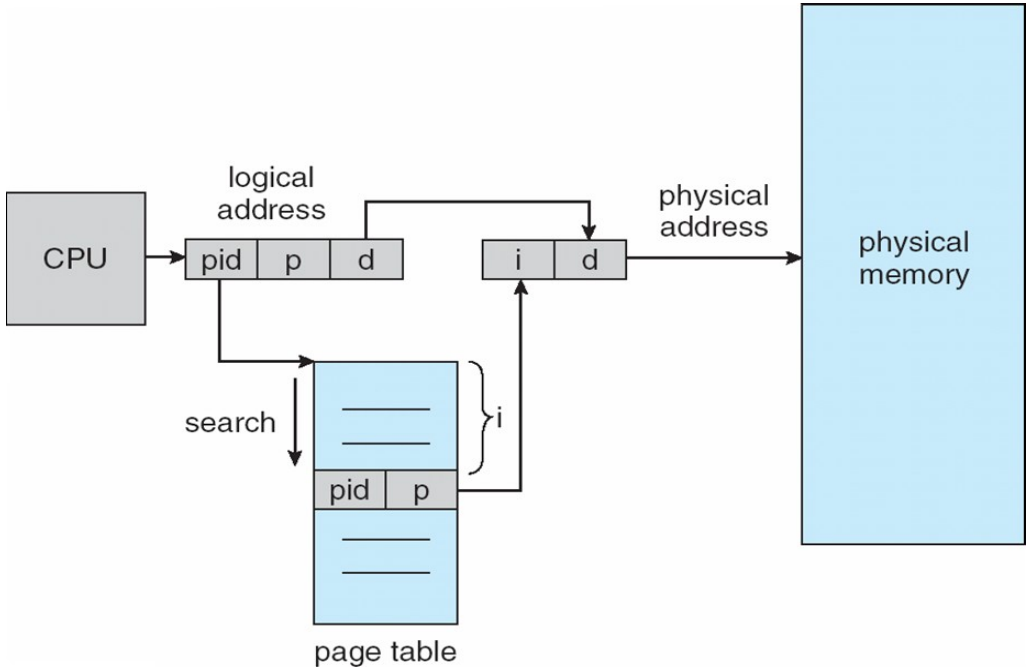


Inverted Page Table

CPU generates the Logical address

Here we have pid, which is used in order to determine for which process the page number belongs

The pid and p will be given as input to the page table and that generates frame number as the output , which will be combined with the offset to generate the Physical address

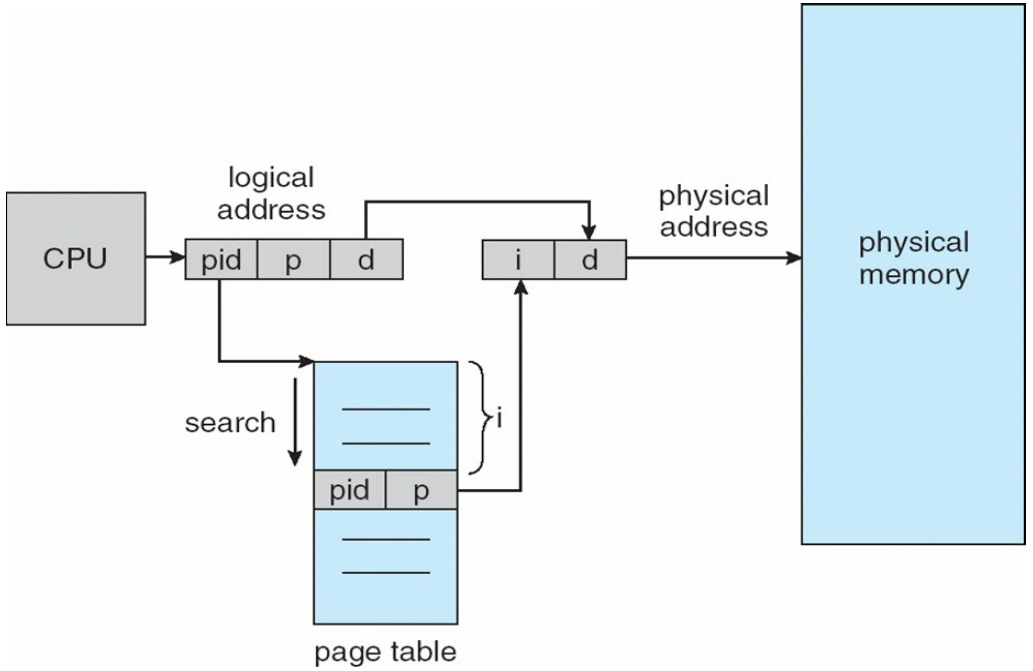


Inverted Page Table

CPU generates the Logical address

Here we have pid, which is used in order to determine for which process the page number belongs

The pid and p will be given as input to the page table and that generates frame number as the output , which will be combined with the offset to generate the Physical address



Virtual Memory

Virtual memory is a mechanism used to manage memory using hardware and software.
It is a part of the secondary storage that gives the user an illusion that it is a part of the main memory.

It helps in running multiple applications with low main memory and increases the degree of multiprogramming in systems. It is commonly implemented using demand paging.

Virtual memory is a part of the system's secondary memory that acts and gives us a feel as if it is a part of the main memory.

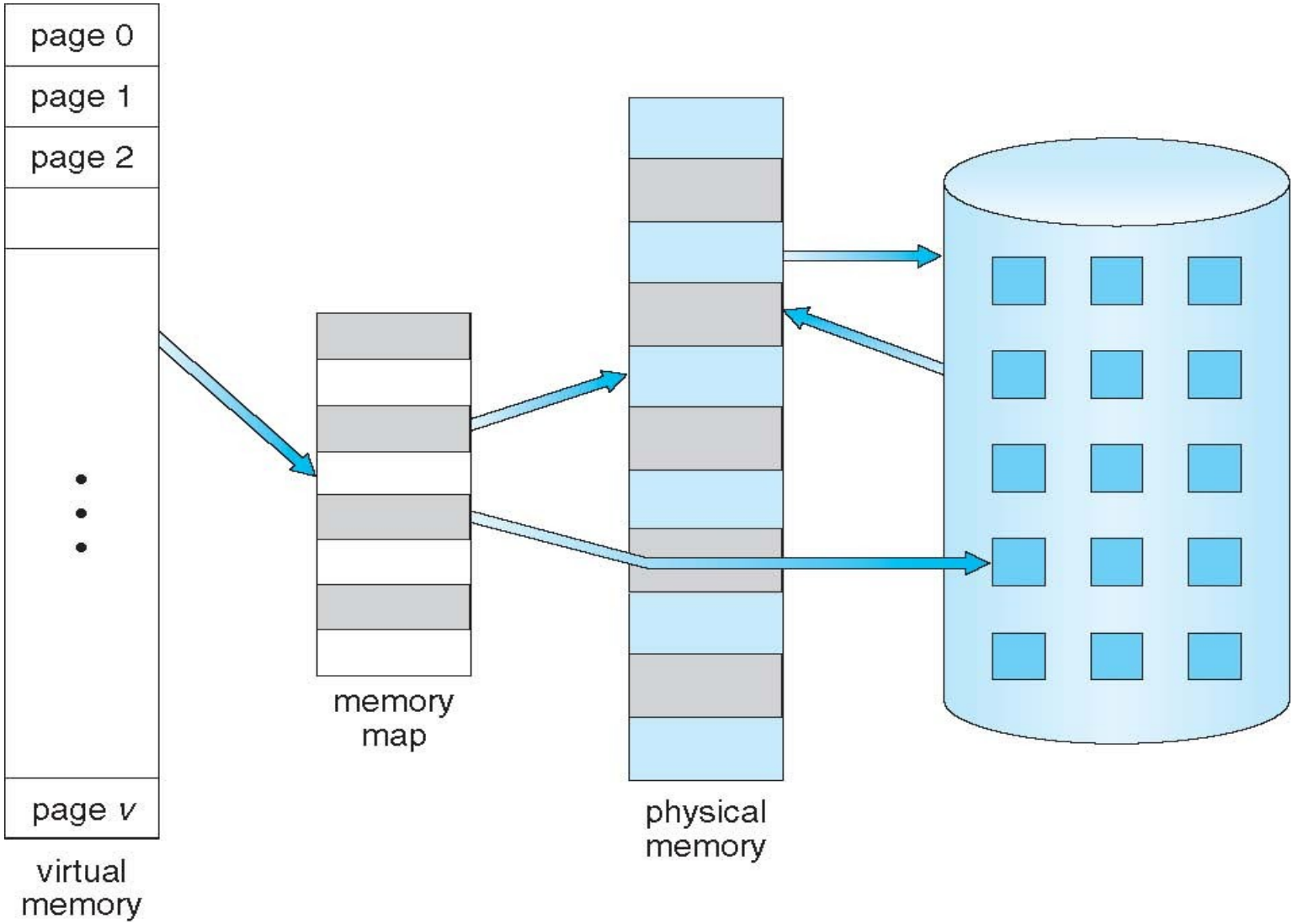
Virtual memory allows a system to execute heavier applications or multiple applications simultaneously without exhausting the RAM (Random Access Memory).

In particular, the system can behave as if its total RAM resources were equal to the whole amount of physical RAM plus the complete amount of virtual RAM
Only part of the program needs to be in memory for execution

Logical address space can therefore be much larger than physical address space

- **Allows address spaces to be shared by several processes**
- **Allows for more efficient process creation**
- **More programs running concurrently**
- **Less I/O needed to load or swap processes**

Virtual Memory That is Larger Than Physical Memory

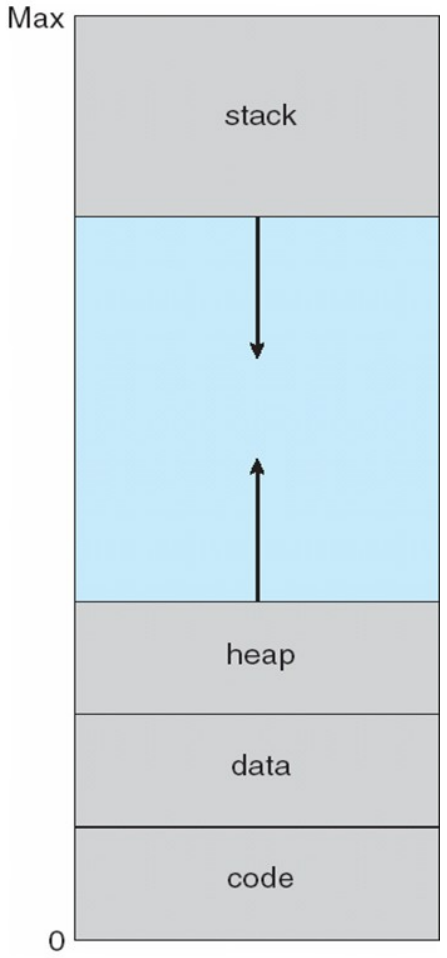


Virtual-address Space

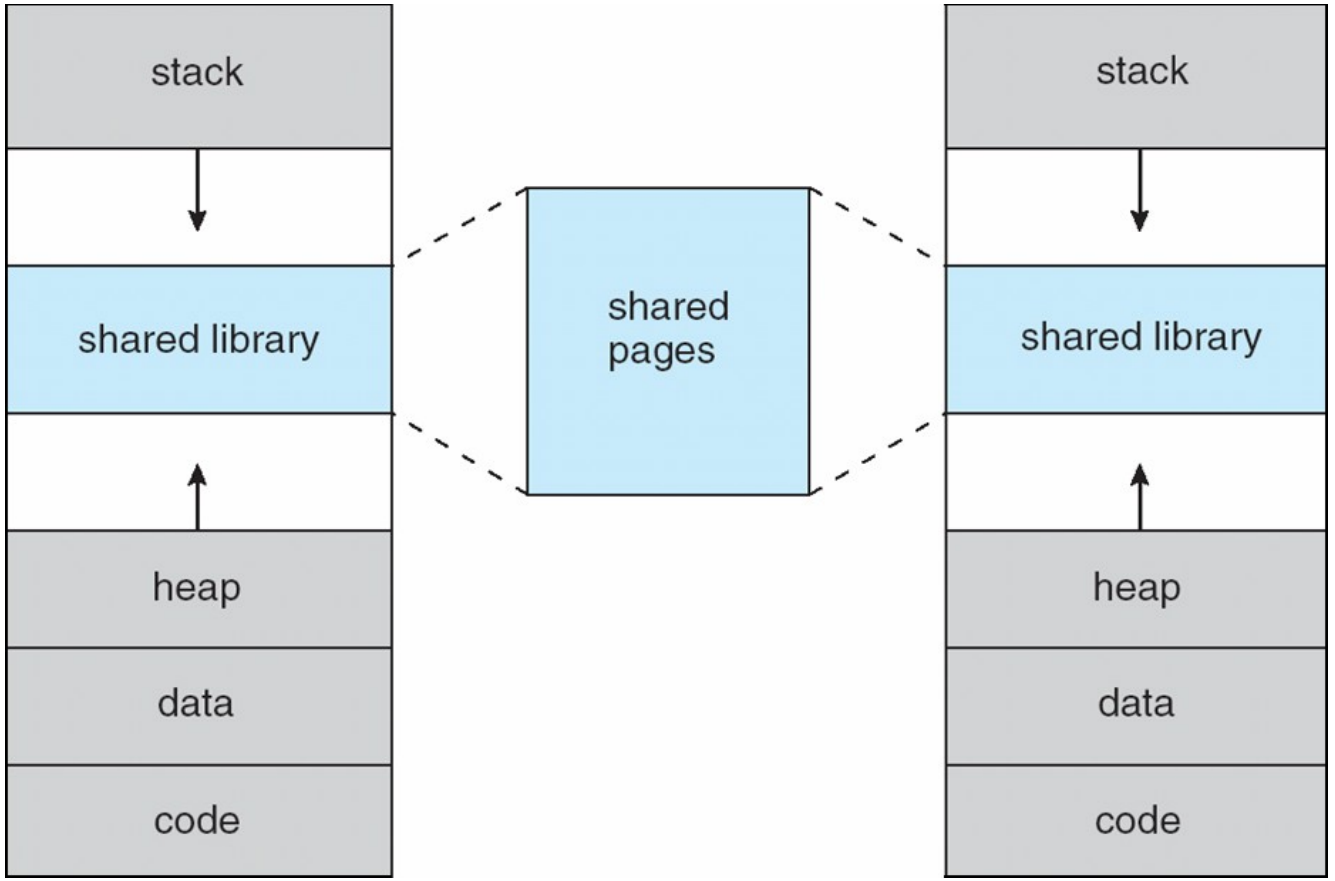
The **virtual address space** of a process refers to the logical (or virtual) view of how a process is stored in memory.

the heap to grow upward in memory as it is used for dynamic memory allocation and Stack grow inwards

- System libraries shared via mapping into virtual address space
- Shared memory by mapping pages read-write into virtual address space
- Pages can be shared during `fork()`, speeding process creation

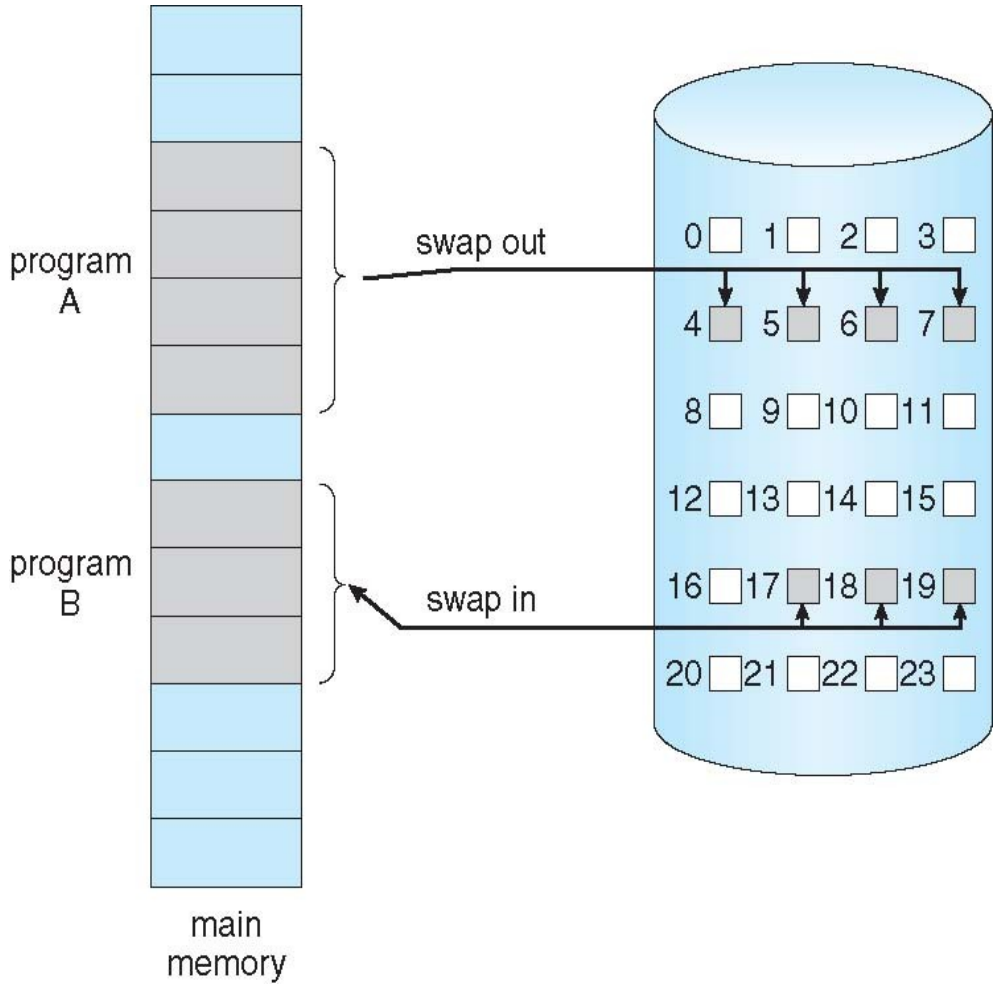


Shared Library Using Virtual Memory



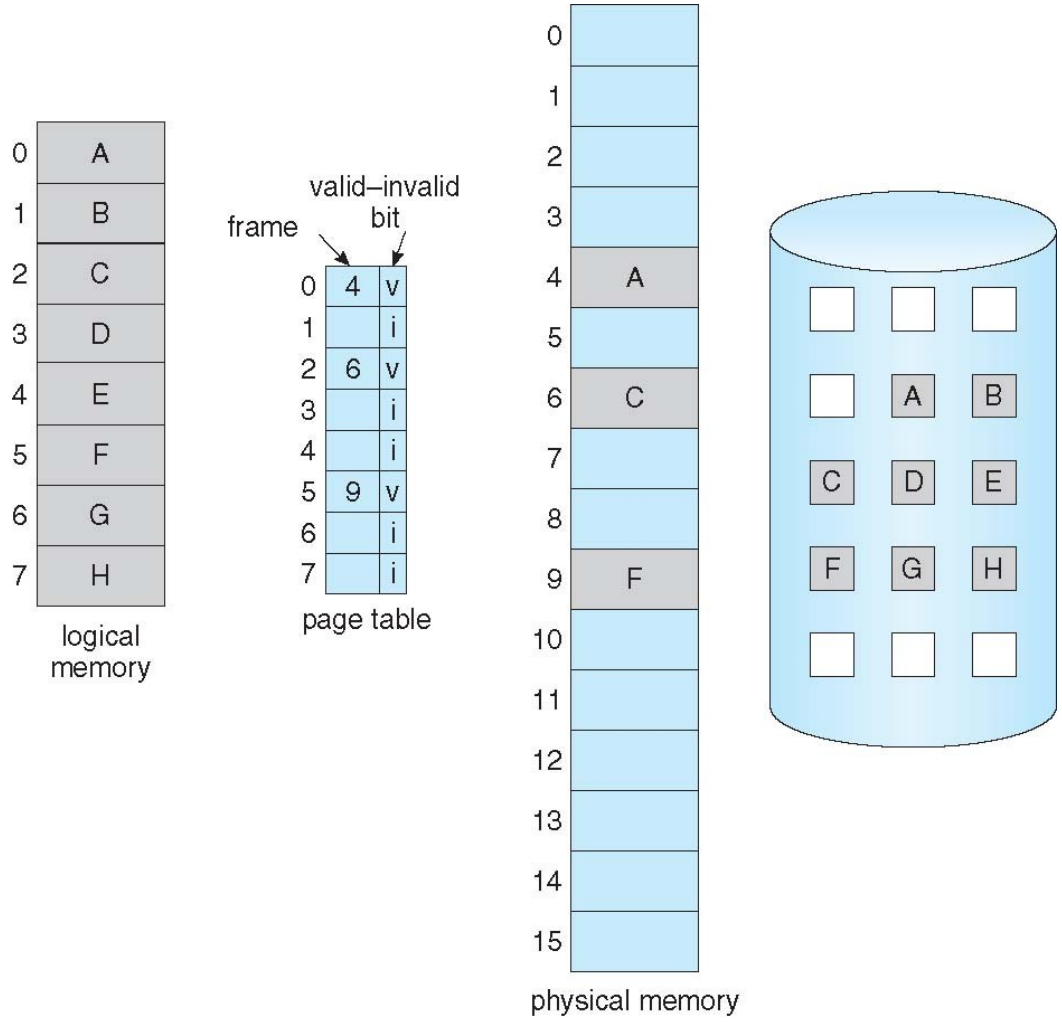
Demand Paging

- Could bring entire process into memory at load time
- Or bring a page into memory only when it is needed
  - Less I/O needed, no unnecessary I/O
  - Less memory needed
  - Faster response
  - More users
- Similar to paging system with swapping (diagram on right)
- Page is needed ⇒ reference to it
  - invalid reference ⇒ abort
  - not-in-memory ⇒ bring to memory
- Lazy swapper – never swaps a page into memory unless page will be needed
  - Swapper that deals with pages is a pager



Page Table When Some Pages Are Not in Main Memory

With each page table entry a valid–invalid bit is associated (**v** ⇒ in-memory – **memory resident**, **i** ⇒ not-in-memory)

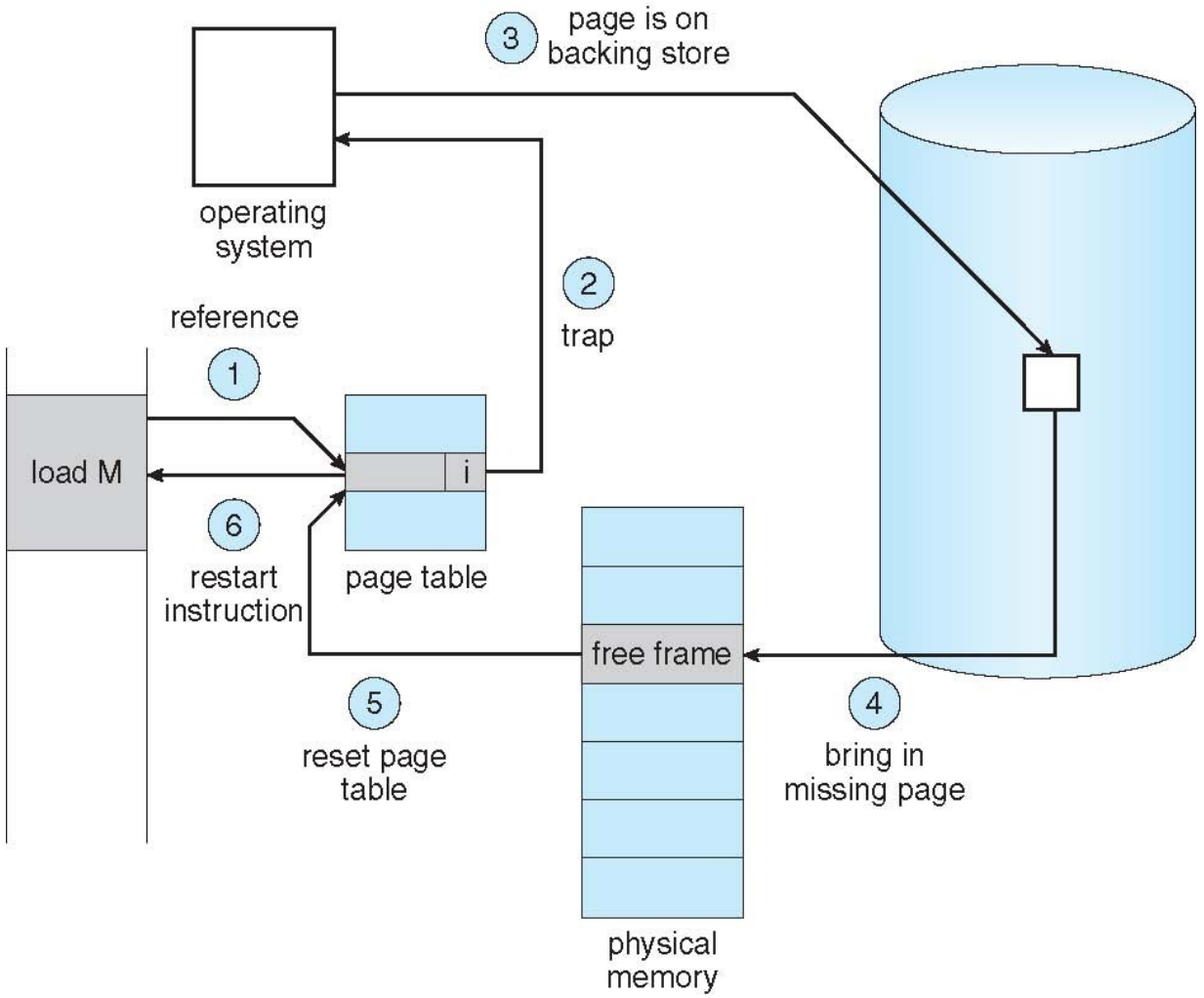


Page Fault

A page fault is trap that occurs when the requested page is not loaded into the memory,

in other words, when a program tries to access a chunk of memory that does not exist in physical memory (main memory) causes a page fault.

Steps in Handling a Page Fault



When a page fault occurs, the following sequence of events happens:

1.At first, an internal table is created to process whether the reference was valid or invalid memory access.

2.The system gets terminated if the reference becomes invalid, if not, the page will be paged in.

3.After checking for validity, the free-frame list searches the system for a free frame.

4.The disk operation will be scheduled to get the required page from the disk.

5.The page table of the process will be updated with a new frame number, and the invalid bit will be changed after the completion of the I/O operation. Now it is a valid page reference.

6.Restart these steps upon finding any more page faults

Aspects of Demand Paging

- Extreme case – start process with *no* pages in memory
  - OS sets instruction pointer to first instruction of process, non-memory-resident -> page fault
  - And for every other process pages on first access
  - **Pure demand paging**
- Actually, a given instruction could access multiple pages -> multiple page faults
  - Consider fetch and decode of instruction which adds 2 numbers from memory and stores result back to memory
  - Pain decreased because of **locality of reference**
- Hardware support needed for demand paging
  - Page table with valid / invalid bit
  - Secondary memory (swap device with **swap space**)
  - Instruction restart

Performance of Demand Paging

- Stages in Demand Paging (worse case)

1. Trap to the operating system
2. Save the user registers and process state
3. Determine that the interrupt was a page fault
4. Check that the page reference was legal and determine the location of the page on the disk
5. Issue a read from the disk to a free frame:
    1. Wait in a queue for this device until the read request is serviced
    2. Wait for the device seek and/or latency time
    3. Begin the transfer of the page to a free frame
6. While waiting, allocate the CPU to some other user
7. Receive an interrupt from the disk I/O subsystem (I/O completed)
8. Save the registers and process state for the other user
9. Determine that the interrupt was from the disk
10. Correct the page table and other tables to show page is now in memory
11. Wait for the CPU to be allocated to this process again
12. Restore the user registers, process state, and new page table, and then resume the interrupted instruction

Performance of Demand Paging

- Three major activities
  - Service the interrupt – careful coding means just several hundred instructions needed
  - Read the page – lots of time
  - Restart the process – again just a small amount of time
- Page Fault Rate $0 \le p \le 1$
  - if $p = 0$ no page faults
  - if $p = 1$, every reference is a fault
- Effective Access Time (EAT)

$$EAT = (1 - p) \times \text{memory access} + p\ (\text{page fault overhead} + \text{swap page out} + \text{swap page in})$$

Demand Paging Example

- Memory access time = 200 nanoseconds
- Average page-fault service time = 8 milliseconds
- EAT = (1 – p) x 200 + p (8 milliseconds)

  = (1 – p x 200 + p x 8,000,000

  = 200 + p x 7,999,800
- If one access out of 1,000 causes a page fault, then

  EAT = 8.2 microseconds.

  This is a slowdown by a factor of 40!!
- If want performance degradation < 10 percent
  - 220 > 200 + 7,999,800 x p
    20 > 7,999,800 x p
  - p < .0000025
  - < one page fault in every 400,000 memory accesses

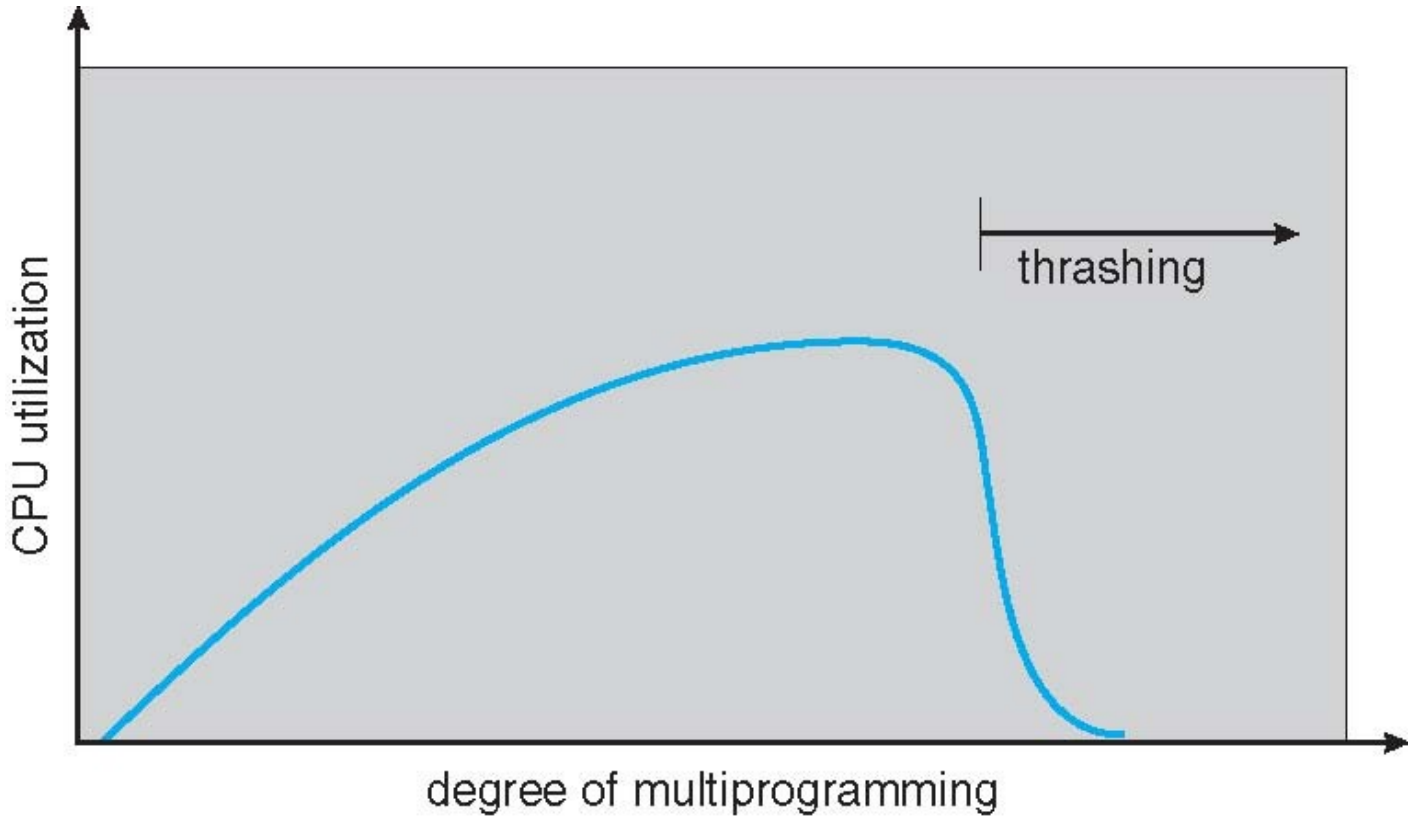
Demand Paging Optimizations

- Swap space I/O faster than file system I/O even if on the same device
  - Swap allocated in larger chunks, less management needed than file system
- Copy entire process image to swap space at process load time
  - Then page in and out of swap space
  - Used in older BSD Unix
- Demand page in from program binary on disk, but discard rather than paging out when freeing frame
  - Used in Solaris and current BSD
  - Still need to write to swap space
    - Pages not associated with a file (like stack and heap) – **anonymous memory**
    - Pages modified in memory but not yet written back to the file system
- Mobile systems
  - Typically don't support swapping
  - Instead, demand page from file system and reclaim read-only pages (such as code)

Thrashing

- If a process does not have “enough” pages, the page-fault rate is very high
  - Page fault to get page
  - Replace existing frame
  - But quickly need replaced frame back
  - This leads to:
    - Low CPU utilization
    - Operating system thinking that it needs to increase the degree of multiprogramming
    - Another process added to the system
- **Thrashing** ≡ a process is busy swapping pages in and out

Thrashing



Demand Paging and Thrashing

- Why does demand paging work?
  Locality model – In Short amount of time , instructions are executed repatadely
  - Process migrates from one locality to another
  - Localities may overlap

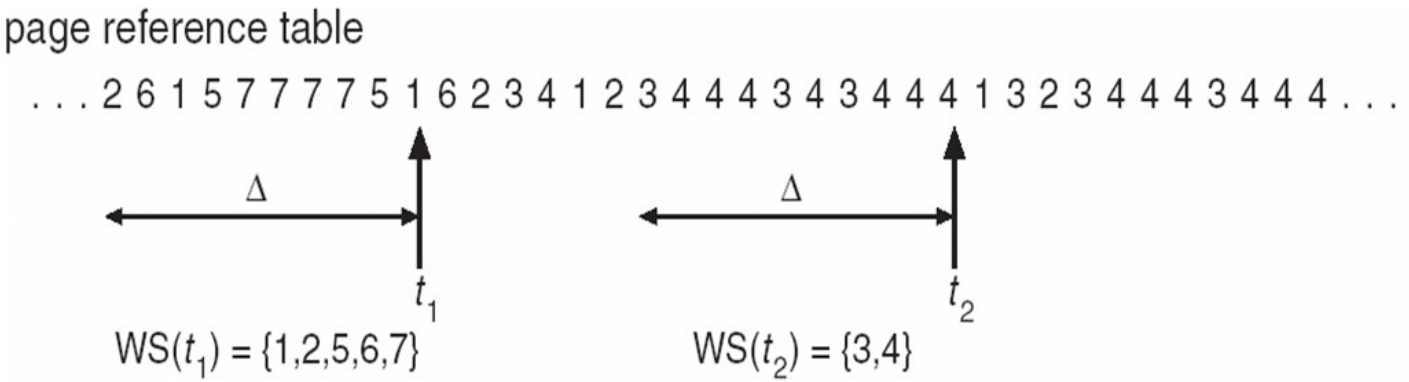
- Why does thrashing occur?
  Σ size of locality > total memory size
  - Limit effects by using local or priority page replacement

Working-Set Model

- $\Delta \equiv$ working-set window $\equiv$ a fixed number of page references – Specfies the size of working set
  Example: 10,000 instructions
- WSS_i (working set of Process P_i) =
  total number of pages referenced in the most recent Δ (varies in time)
  - if Δ too small will not encompass entire locality (Number of Page Fault Increases)
  - if Δ too large will encompass several localities
  - if $\Delta = \infty \Rightarrow$ will encompass entire program
- $D = \Sigma\ WSS_i \equiv$ total demand frames
  - Approximation of locality
- if $D > m \Rightarrow$ Thrashing
- Policy if D > m, then suspend or swap out one of the processes

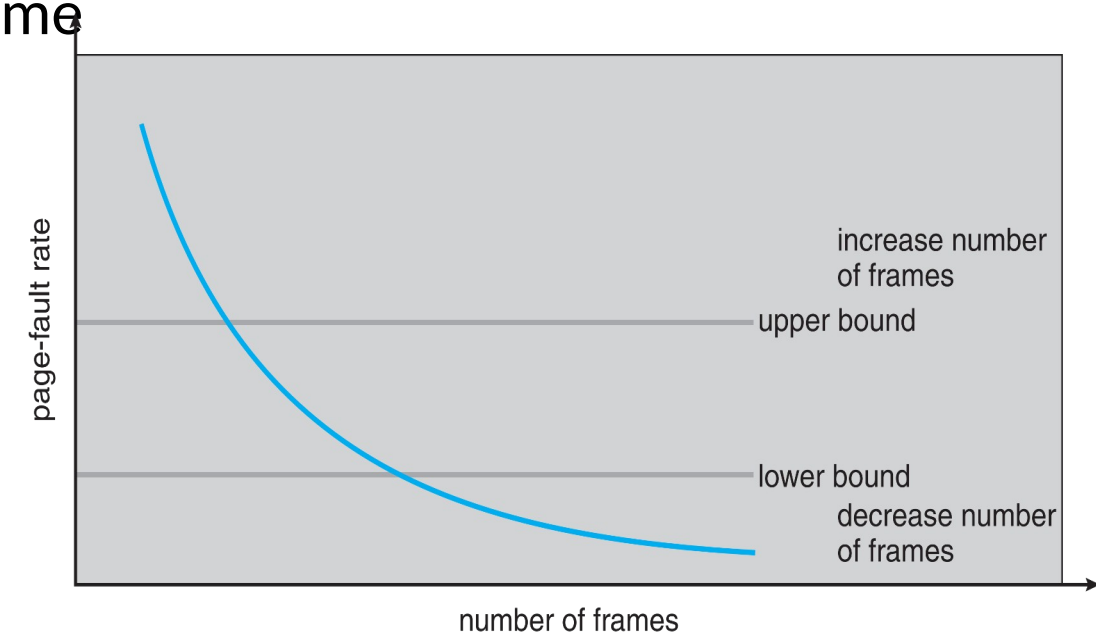
Working-Set Model

- $\Delta \equiv$ working-set window $\equiv$ a fixed number of page references
  Example: 10,000 instructions
- WSS_i (working set of Process P_i) =
  total number of pages referenced in the most recent Δ (varies in time)
  - if Δ too small will not encompass entire locality
  - if Δ too large will encompass several localities
  - if $\Delta = \infty \Rightarrow$ will encompass entire program
- $D = \Sigma\ WSS_i \equiv$ total demand frames
  - Approximation of locality
- if $D > m \Rightarrow$ Thrashing : (m = main memory frames)
- Policy if D > m, then suspend or swap out one of the processes



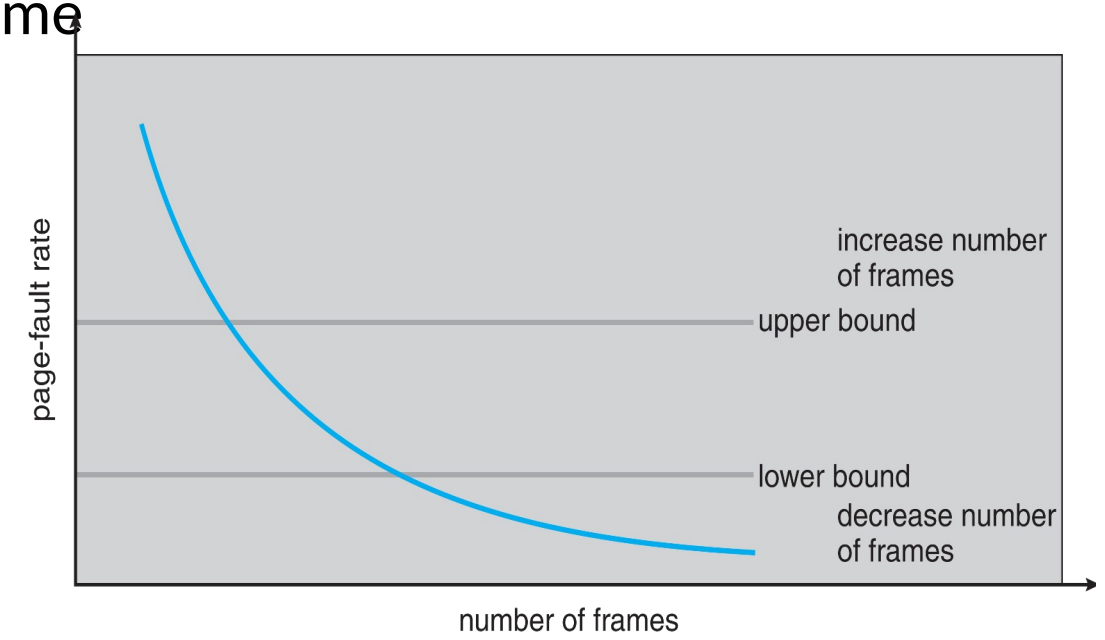
Page-Fault Frequency

- More direct approach than WSS
- Establish “acceptable” **page-fault frequency** (**PFF**) rate and use local replacement policy
  - If actual rate too low, process loses frame
  - If actual rate too high, process gains frame



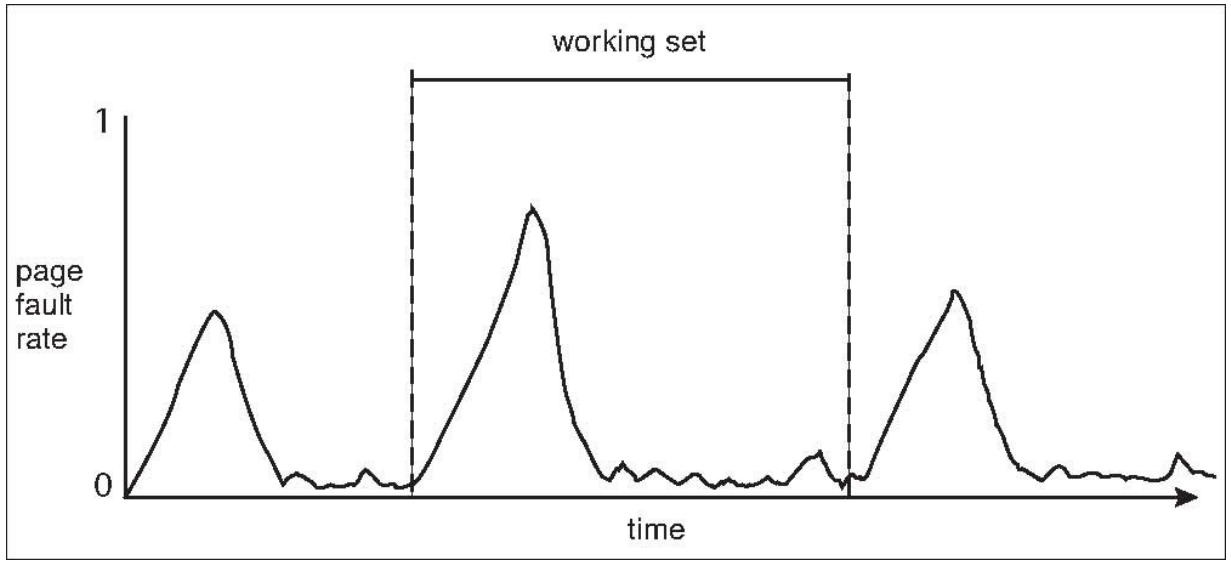
Page-Fault Frequency

- More direct approach than WSS
- Establish “acceptable” **page-fault frequency** (**PFF**) rate and use local replacement policy
  - If actual rate too low, process loses frame
  - If actual rate too high, process gains frame



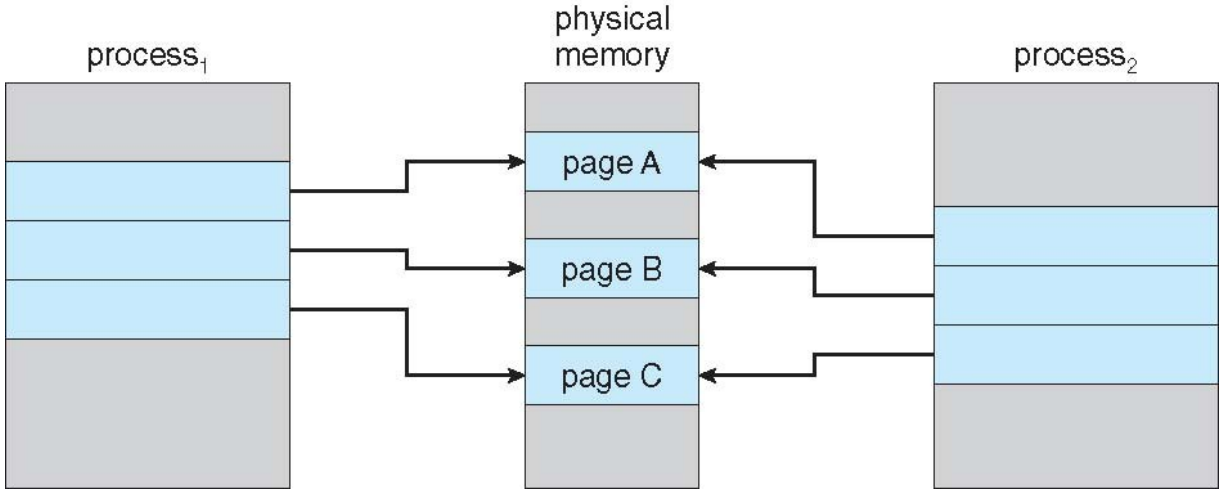
Working Sets and Page Fault Rates

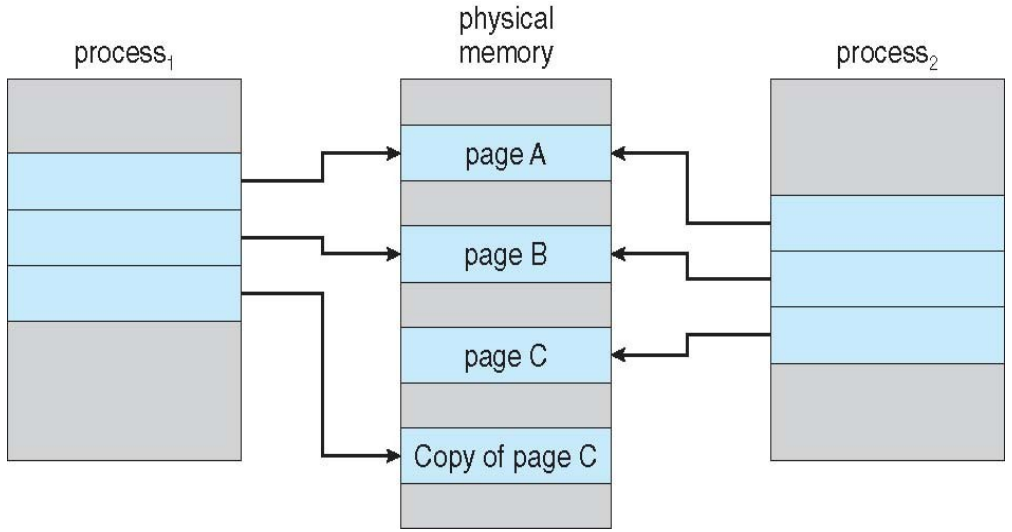
- Direct relationship between working set of a process and its page-fault rate
- Working set changes over time
- Peaks and valleys over time



Copy and Write

- **Copy-on-Write** (COW) allows both parent and child processes to initially ***share*** the same pages in memory
  - If either process modifies a shared page, only then is the page copied
- COW allows more efficient process creation as only modified pages are copied
- In general, free pages are allocated from a **pool** of **zero-fill-on-demand** pages
  - Pool should always have free frames for fast demand page execution
    - Don't want to have to free a frame as well as other processing on page fault





Page Replacement

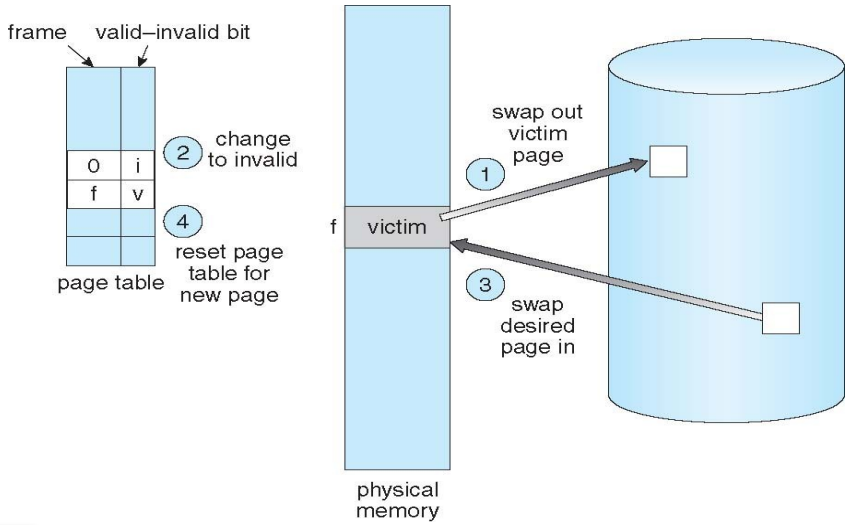
Whenever the process wants to execute the page, it will check whether the page is available in Main Memory or not.

If it is there , it will execute.

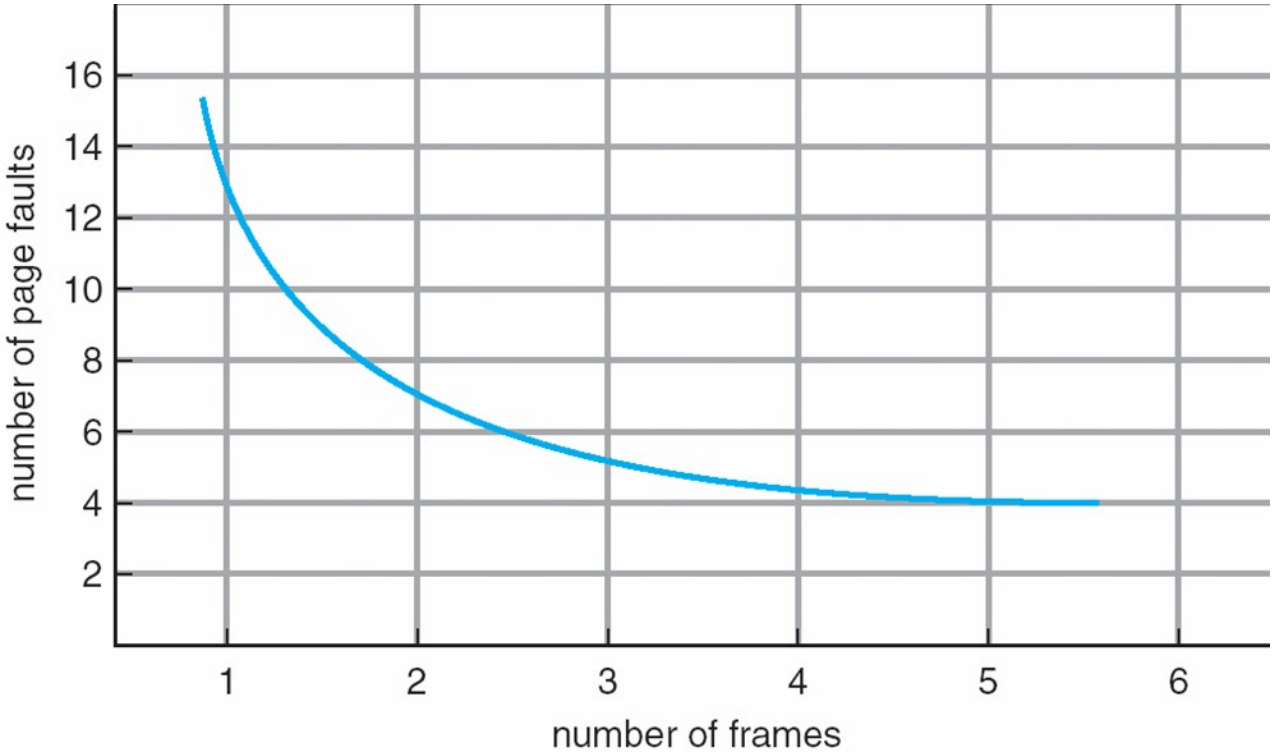
If the page is not there it should be swapped in from the secondary memory to main memory.

If the main memory is full, some pages has to be swapped out to secondary memory.

Swapped out pages data has to be updated in the Page table.



Graph of Page Faults Versus The Number of Frames



First-In-First-Out (FIFO) Algorithm

The page that is there for the longest period of the time in the main memory will be replaced.

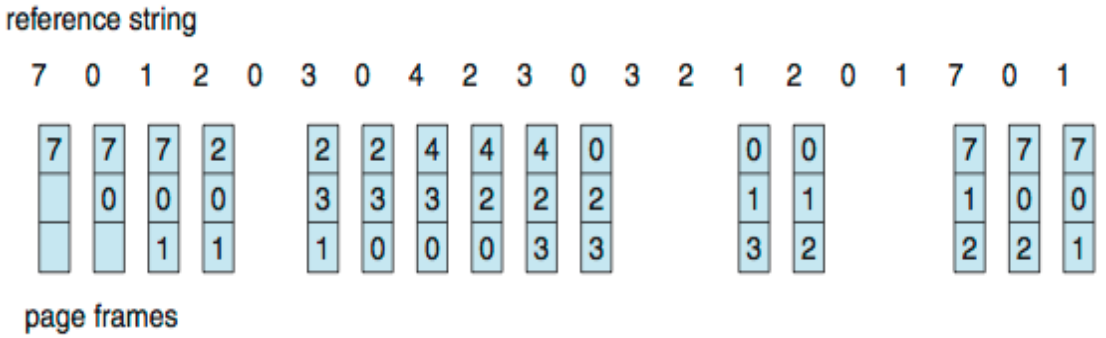
Eg

Consider the Empty Main Memory

Assume main memory can hold max 3 frames at the same time

Reference string: 7,0,1,2,0,3,0,4,2,3,0,3,0,3,2,1,2,0,1,7,0,1

The OS request for Page 7, it is not there in the main memory(We assumed it to be empty), page fault occurs, Page 7 will be brought from the secondary memory, next 0 the same process and it continues



15 page faults occurred

First-In-First-Out (FIFO) Algorithm- Belady's Anomaly

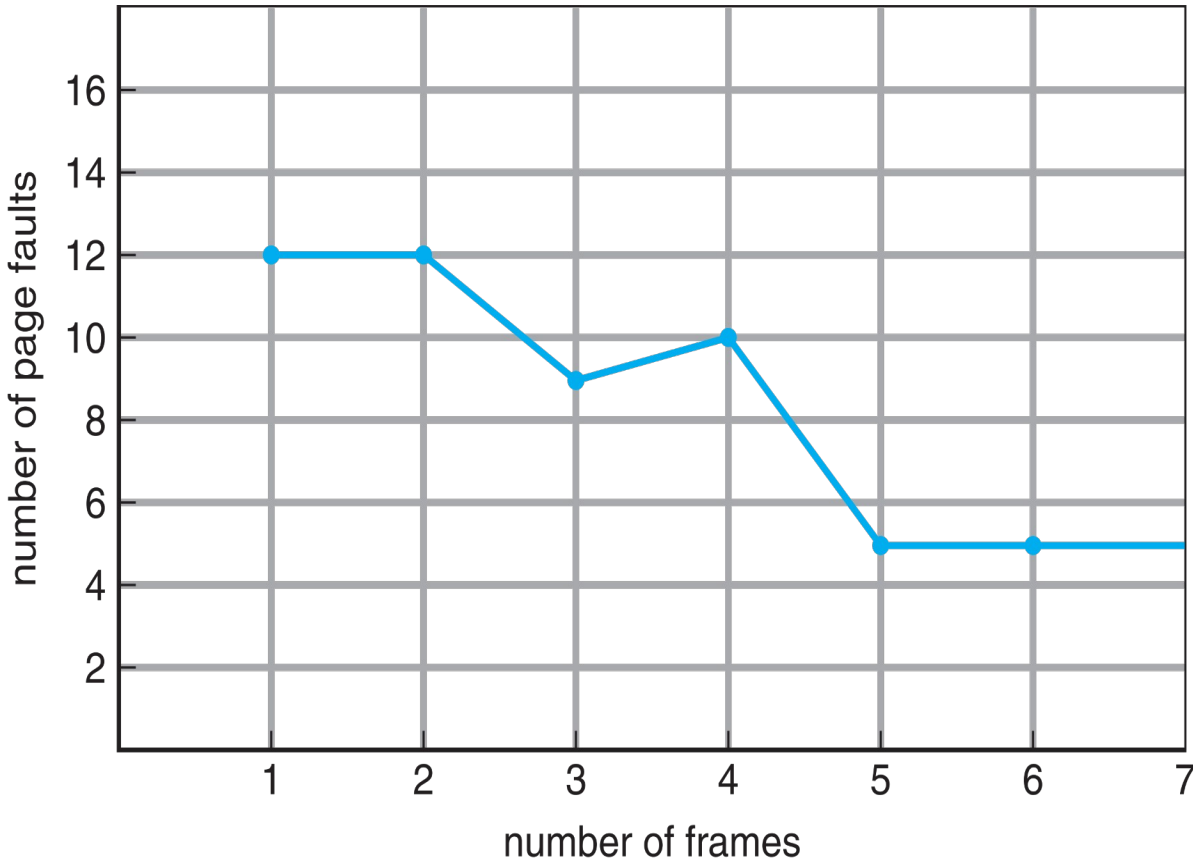
If we increase the number of frames, the number of the page faults will also be increased.

Let us consider the example 1,2,3,4,1,2,5,1,2,3,4,5

The number of page faults by considering 3 frames -9

The number of page faults by considering 4 frames-10

Though we increase the number of frames , the page faults will not decrease.



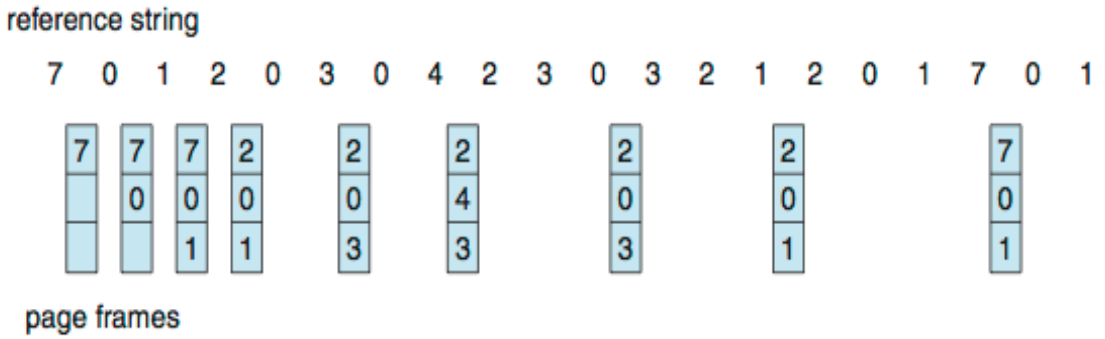
Optimal Algorithm

- Replace page that will not be used for longest period of time in future.

Consider the Empty Main Memory

Assume main memory can hold max 3 frames at the same time

Reference string: 7,0,1,2,0,3,0,4,2,3,0,3,0,3,2,1,2,0,1,7,0,1



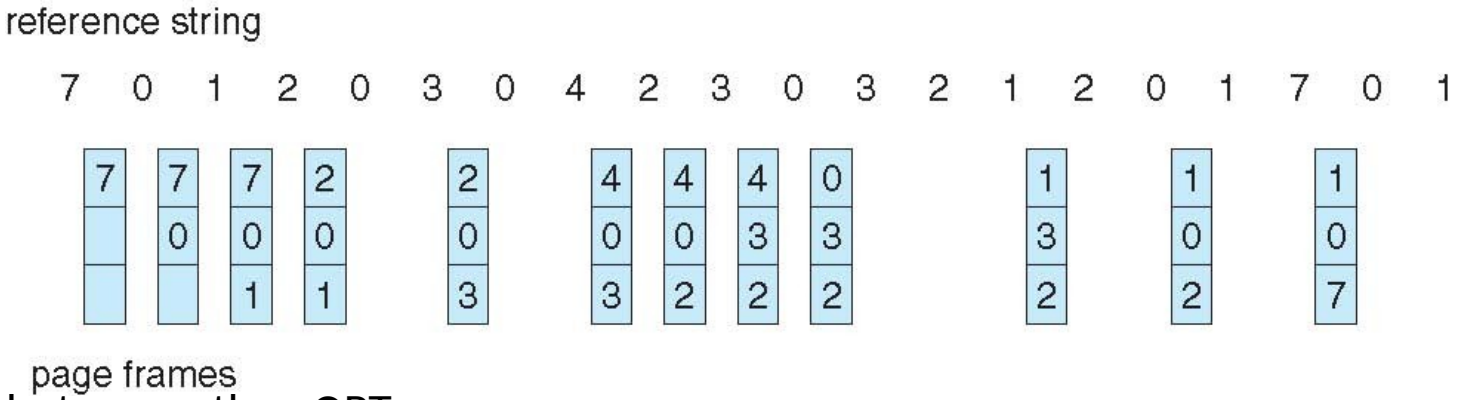
The number of page faults by considering 3 frames -9

Least Recently Used (LRU) Algorithm

Use past knowledge rather than future
Replace page that has not been used in the most amount of time
Associate time of last use with each page

Consider the Empty Main Memory
Assume main memory can hold max 3 frames at the same time

Reference string:
7,0,1,2,0,3,0,4,2,3,0,3,0,3,2,1,2,0,1,7,0,1



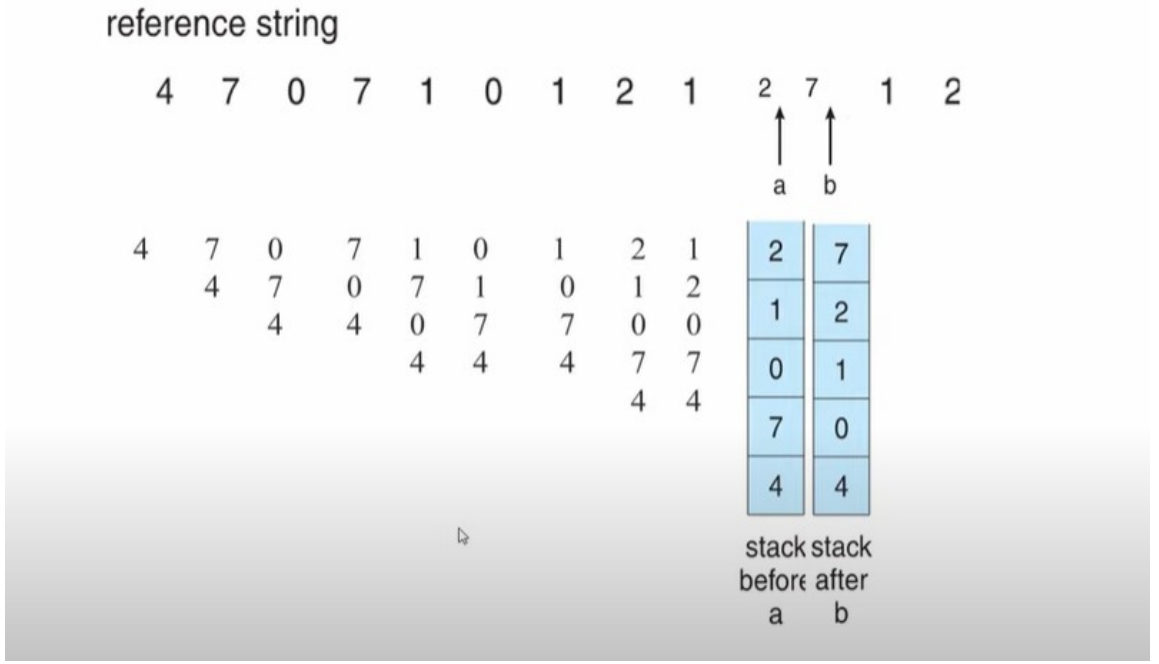
12 faults – better than FIFO but worse than OPT
Generally good algorithm and frequently used
Assignment :Reference string: 2,3,2,1,5,2,4,5,3,2,5,2

Least Recently Used (LRU) Algorithm - Implementation

- *Counter implementation*
  - Every page entry has a time of used field; every time page is referenced through this entry, that data is stored in the counter
  - When a page needs to be changed, look at the counters to find smallest value
    - Search through table needed
- *Stack implementation*
  - Keep a stack of page numbers in a double link form:
  - Page referenced:
    - move it to the top
    - requires 6 pointers to be changed
  - But each update more expensive
  - No search for replacement
- LRU and OPT are cases of **stack algorithms** that don't have Belady's Anomaly

Least Recently Used (LRU) Algorithm - Implementation

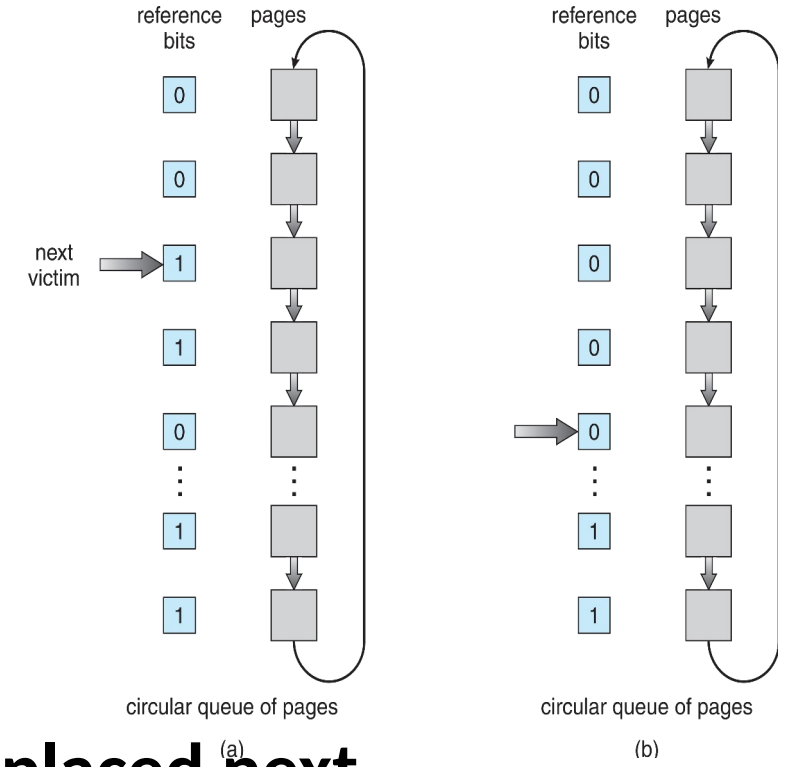
- *Stack implementation*
  - Keep a stack of page numbers in a double link form:
  - Most reference page number will be at top
  - Least referenced will be at bottom
  - Page referenced:
    - move it to the top
    - requires 6 pointers to be changed
  - But each update more expensive
  - No search for replacement



LRU Approximation Algorithms

- LRU needs special hardware and still slow
- **Reference bit**
  - With each page associate a bit, initially = 0
  - When page is referenced bit set to 1
  - Replace any with reference bit = 0 (if one exists)
    - Drawback: We do not know the order.
- **Second-chance algorithm**
  - Generally it uses FIFO, plus hardware-provided reference bit
  - **Clock** replacement
  - If page to be replaced has
    - Reference bit = 0 -> replace it
    - reference bit = 1 then page will be given the second page
      - set reference bit 0, leave page in memory
      - replace next page, subject to same rules

LRU Approximation Algorithms- Second Chance Algorithm



A pointer indicates which page has to be replaced next.

A pointer is pointing to 1, it will set that to 0, move to next.

By this the oldest page will be replaced.

Enhanced Second-Chance Algorithm

- Improve algorithm by using reference bit and modify bit (if available) in concert
- Take ordered pair (reference, modify)

1.(0, 0) neither recently used not modified – best page to replace

2.(0, 1) not recently used but modified – not quite as good, must write out before replacement

3.(1, 0) recently used but clean – probably will be used again soon

4.(1, 1) recently used and modified – probably will be used again soon and need to write out before replacement

- When page replacement called for, use the clock scheme but use the four classes replace page in lowest non-empty class
  - Might need to search circular queue several times

Counting Algorithms

- Keep a counter of the number of references that have been made to each page
  - Not common
- **Lease Frequently Used** (**LFU**) **Algorithm**: replaces page with smallest count
- **Most Frequently Used** (**MFU**) **Algorithm**: based on the argument that the page with the smallest count was probably just brought in and has yet to be used

Allocation of Frames

- Equal allocation – For example, if there are 100 frames (after allocating frames for the OS) and 5 processes, give each process 20 frames
  - Keep some as free frame buffer pool
- Proportional allocation – Allocate according to the size of process
  - Dynamic as degree of multiprogramming, process sizes change

s_i = size of process p_i

$S = \sum s_i$

m = total number of frames

a_i = allocation for $p_i = \frac{s_i}{S} \times m$

$m = 64$

$s_1 = 10$

$s_2 = 127$

$a_1 = \frac{10}{137} \times 62 \approx 4$

$a_2 = \frac{127}{137} \times 62 \approx 57$

Allocation of Frames

If the total number of available frames is 100, and there are 2 processes one of 2pages and the other of 10 pages then how much of memory would be proportionally allocated to each of these processes?

s_i = size of process p_i

$S = \sum s_i$

m = total number of frames

a_i = allocation for $p_i = \frac{s_i}{S} \times m$

$m = 64$

$s_1 = 10$

$s_2 = 127$

$a_1 = \frac{10}{137} \times 62 \approx 4$

$a_2 = \frac{127}{137} \times 62 \approx 57$

Priority Allocation

Use a proportional allocation scheme using priorities rather than size

If process P_i generates a page fault,

- **select for replacement one of its frames**
- **select for replacement a frame from a process with lower priority number**

Global vs. Local Allocation

Global replacement – process selects a replacement frame from the set of all frames; one process can take a frame from another

- But then process execution time can vary greatly
- But greater throughput so more common

Local replacement – each process selects from only its own set of allocated frames

- More consistent per-process performance
- But possibly underutilized memory

Assuming a 1-KB page size, what are the page numbers and offsets for the following address references (provided as decimal numbers): a. 3085 b. 42095 c. 215201 d. 650000

The given page size is = 1KB .
1 KB= 1024 B = 2 ^10 B .
Nothing is given therefore it means that the addresses are byte addressable .
Number of bits reserved for the page offset is =10 .
This means in total 10 LSB will be used to refer to the page offset .

Now ,
A>>> 3085 .(in decimal)
In binary 3085 = 110000001101 in binary .
The LSB 10 bits are the page offset and the rest 2 bit is the page number .In decimal the page number =3 and the page offset = 13 .

Effective Access Time

Effective Access Time =

Hit ratio of TLB x { Access time of TLB + Access time of main memory }

+

Miss ratio of TLB x { Access time of TLB + (L+1) x Access time of main memory }

where L = Number of levels of page table

Consider a single level paging scheme with a TLB. Assume no page fault occurs. It takes 20 ns to search the TLB and 100 ns to access the physical memory. If TLB hit ratio is 80%, the effective memory access time is _____ msec.

Effective Access Time

Consider a single level paging scheme with a TLB. Assume no page fault occurs. It takes 20 ns to search the TLB and 100 ns to access the physical memory. If TLB hit ratio is 80%, the effective memory access time is _____ msec.

TLB Miss ratio

- = 1 – TLB Hit ratio
- = 1 – 0.8
- = 0.2
- **Calculating Effective Access Time-**
-
- Substituting values in the above formula, we get-
- Effective Access Time
- = 0.8 x { 20 ns + 100 ns } + 0.2 x { 20 ns + (1+1) x 100 ns }
- = 0.8 x 120 ns + 0.2 + 220 ns
- = 96 ns + 44 ns
- = 140 ns
- Thus, effective memory access time = 140 ns.

Consider a logical address space of 64 pages of 1024 words each, mapped onto a physical memory of 32 frames. a. How many bits are there in the logical address? b. How many bits are there in the physical address?

Answer:

Method1:

a) m =???

Size of logical address space $= 2^m =$ # of pages × page size

$2^m = 64 \times 1024$

$2^m = 2^6 \times 2^{10}$

$2^m = 2^{16}$ »» **m=16 bit**

b)

Let (**x**) is number of bits in the physical address

x =???

Size of physical address space = 2^x

Size of physical address space = # of frames × frame size

(frame size = page size)

Size of physical address space = 32×1024

$2^x = 2^5 \times 2^{10}$

$2^x = 2^{15}$

»» number of required bits in the physical address=**x =15 bit**

End of Chapter Unit-4