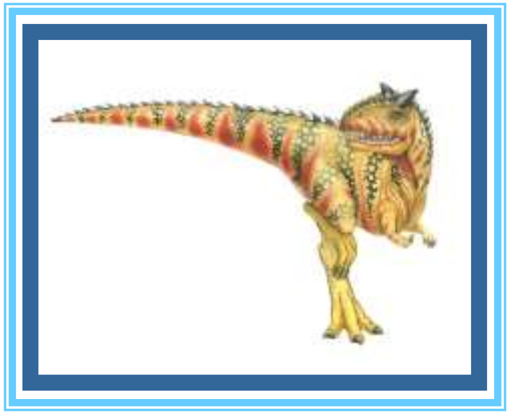
Chapter 8: Main Memory



In Chapter 6, we showed how the CPU can be shared by a set of processes.

As a result of CPU scheduling, we can improve both the utilization of the CPU and the speed of the computer’s response to its users.

To realize this increase in performance, however, we must keep several processes in memory—that is, we must share memory.

In this chapter, we discuss various ways to manage memory.

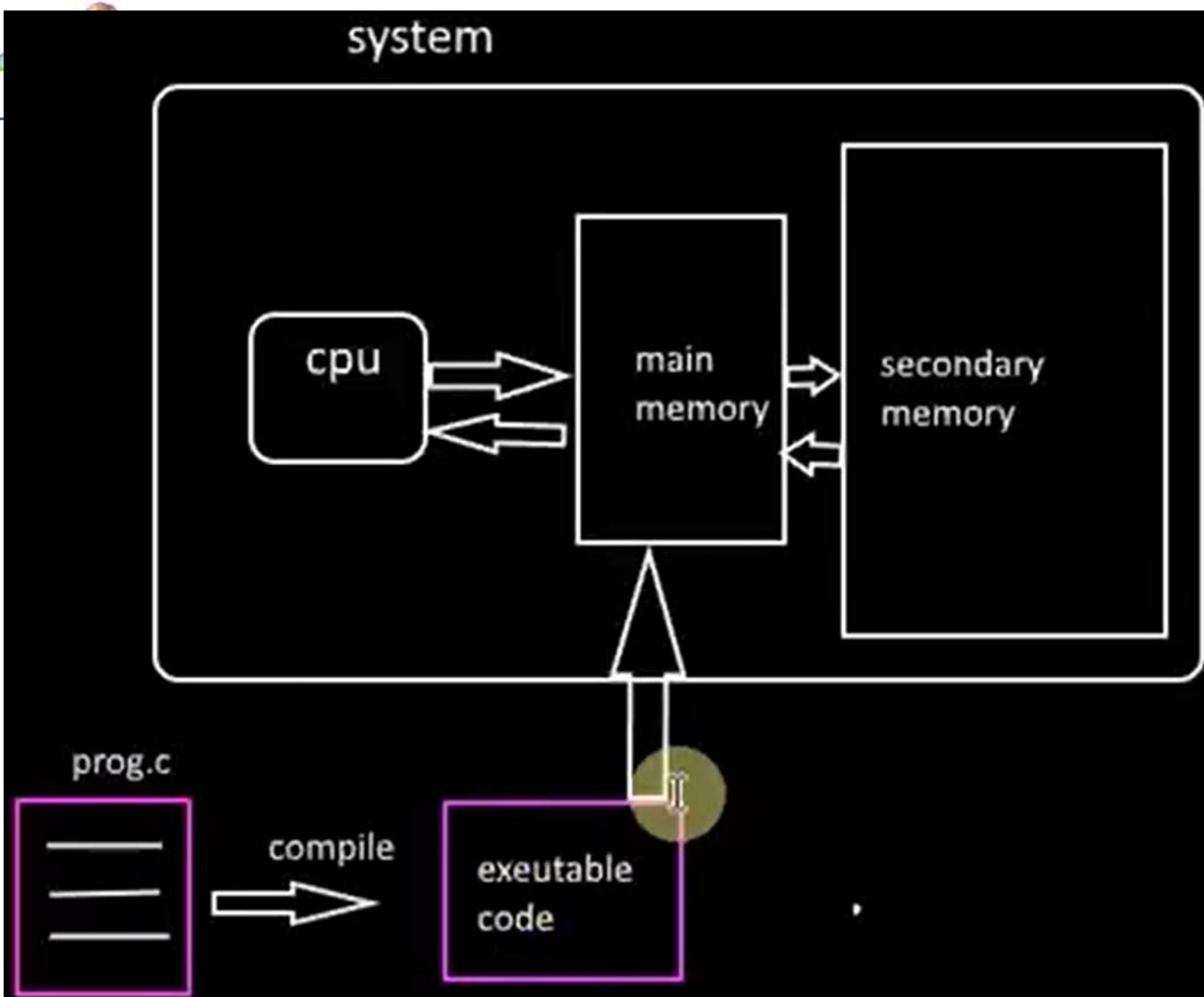
Question

Question:

When a program is executed, how does the CPU know what to do and what data to use

Background

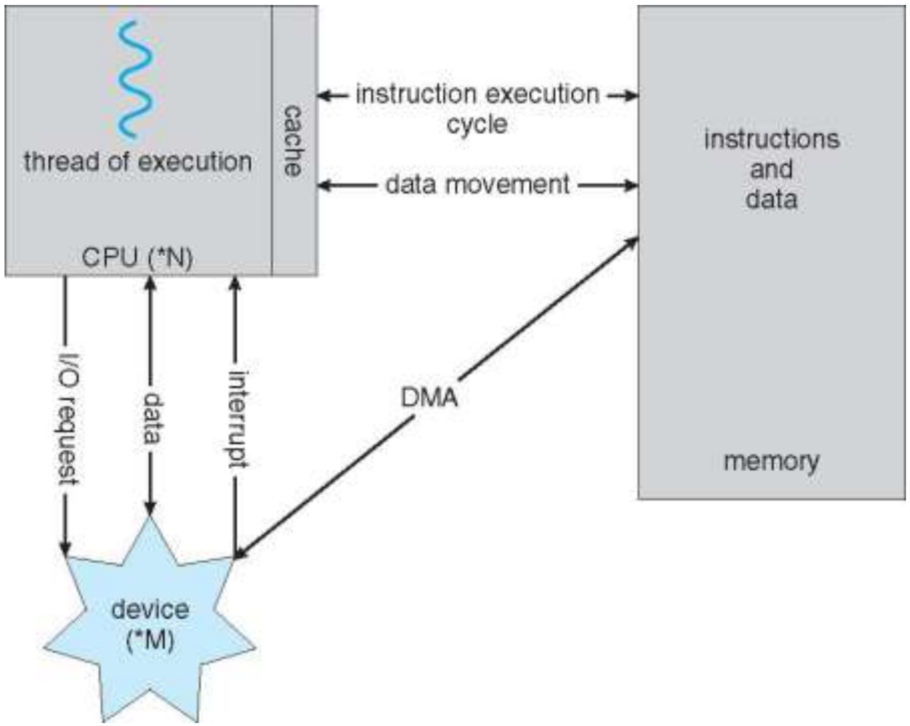
- Program must be brought (from **disk**) into **memory** and placed **within a process** for it to be run
- **Main memory** and **registers** are only storage CPU can access directly
- A typical **instruction-execution cycle**:
  - First fetches an instruction from memory.
  - The instruction is then decoded and may cause operands to be fetched from memory.
  - After the instruction has been executed on the operands, results may be stored back in memory.
- Memory unit only sees a stream of memory addresses. It does not know how they are generated (by the instruction counter, indexing, and so on) or what they are for (instructions or data).

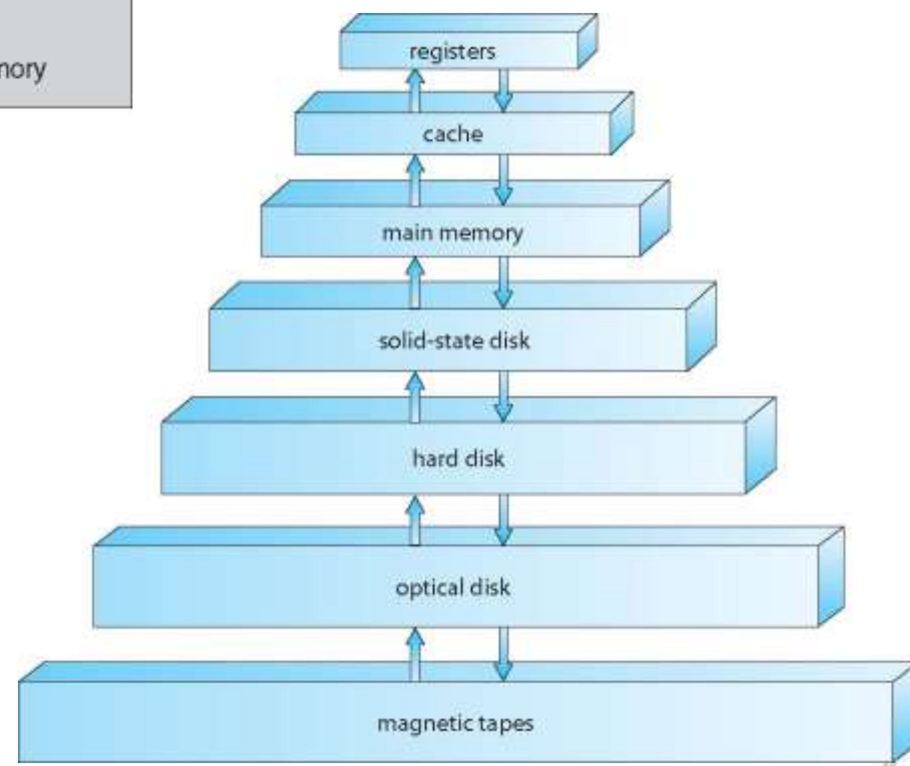

system
cpu
main
memory
secondary
memory
prog.c
compile
exeutable
code

Background (cont.)

- Register access in one CPU clock (or less)
- Main memory can take many cycles, causing a **stall** (because it does not have the data required to complete the instruction that it is executing)
- The remedy is to add fast memory between the CPU and main memory, typically on the CPU chip for fast access. --- **Cache** sits between main memory and CPU registers
- Protection of memory required to ensure correct operation
  - Need to differentiate **logic address space** and **physical address space**
    - For proper system operation we must protect the operating system from access by user processes. On multiuser systems, we must additionally protect user processes from one another

Background (cont.)





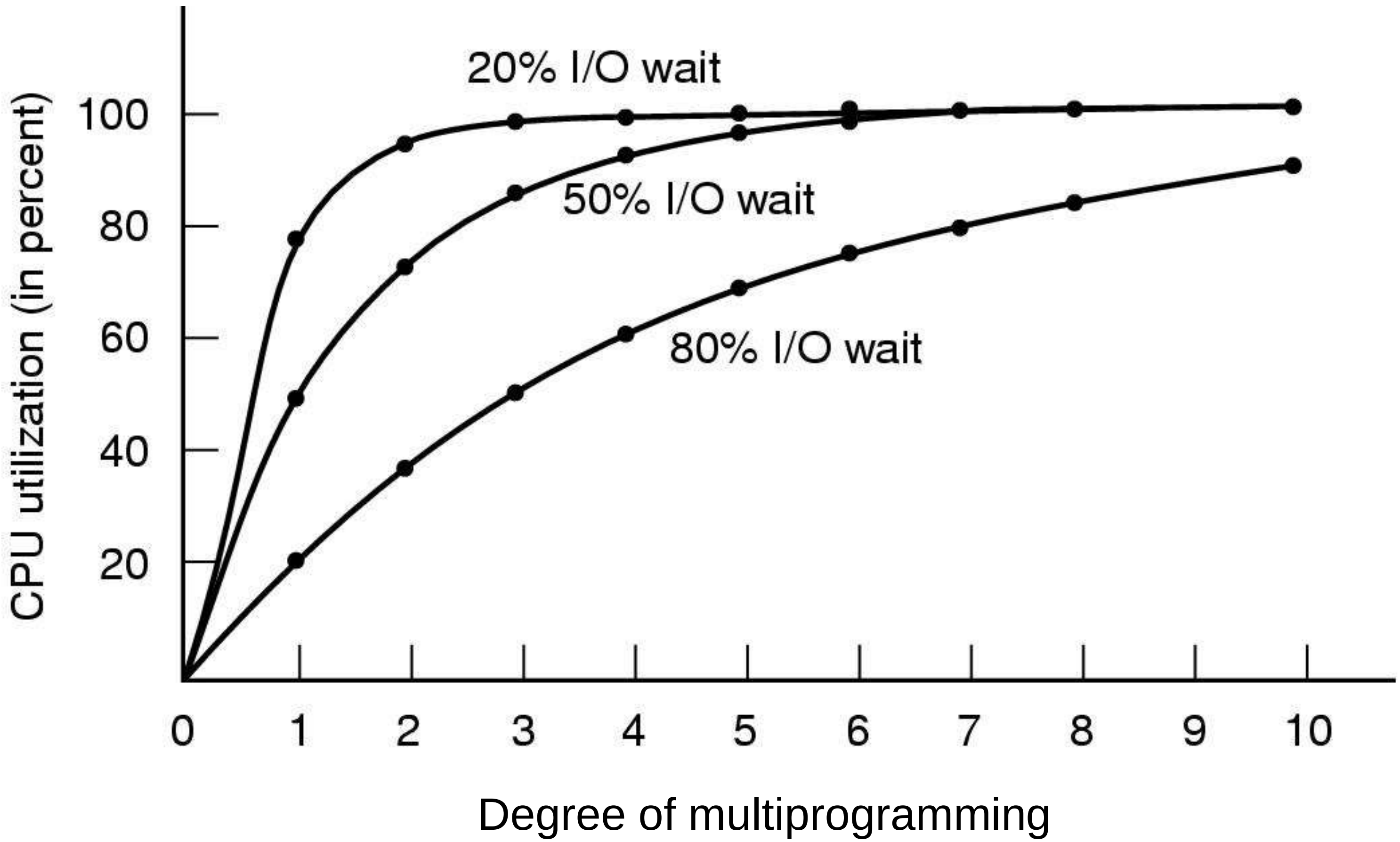
Memory Management

- Ideally programmers want memory that is
  - large
  - fast
  - non-volatile

- Memory hierarchy
  - small amount of fast, expensive memory – cache
  - some medium-speed, medium price – main memory
  - gigabytes of slow, cheap – disk storage

- Memory manager handles the memory hierarchy

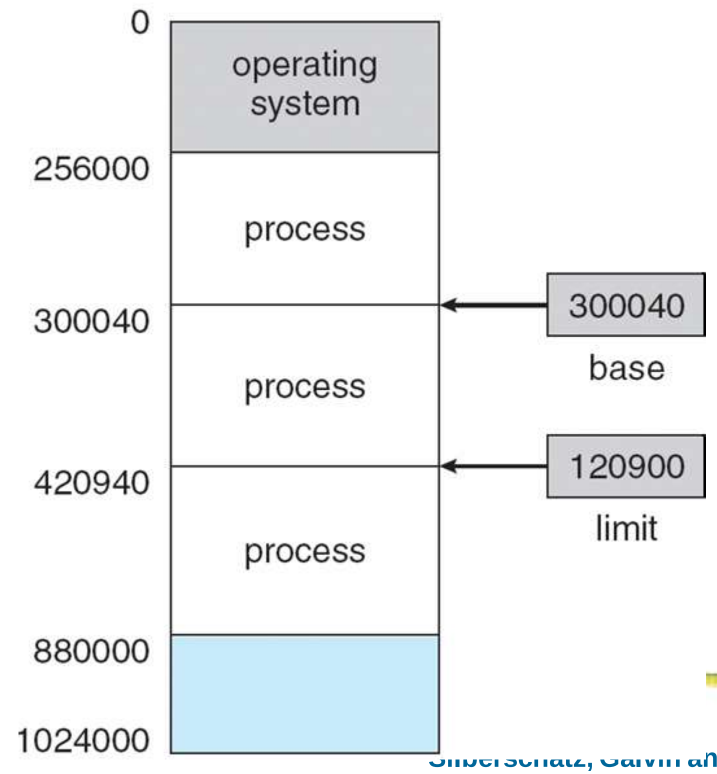
Modeling Multiprogramming



CPU utilization as a function of number of processes in memory

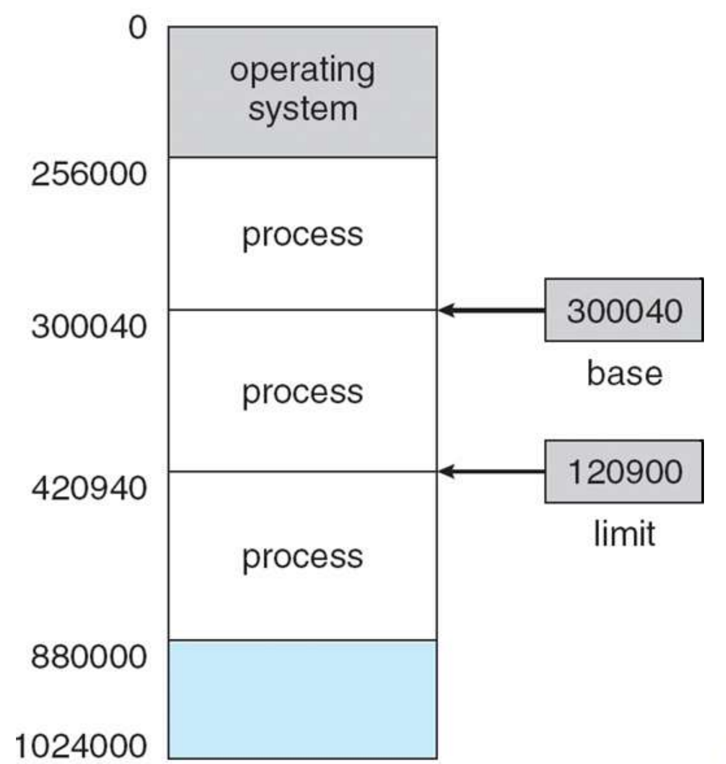
Base and Limit Registers

- The goal is to make sure that each process has a separate memory space
- Need to determine the range of legal addresses to ensure legal access
- A pair of **base** and **limit registers** define the **physical memory address**
  - Base register: holds the smallest legal physical memory address
  - Limit register: specifies the size of the range
- CPU must check every memory access generated in user mode to be sure it is between base and limit for that user



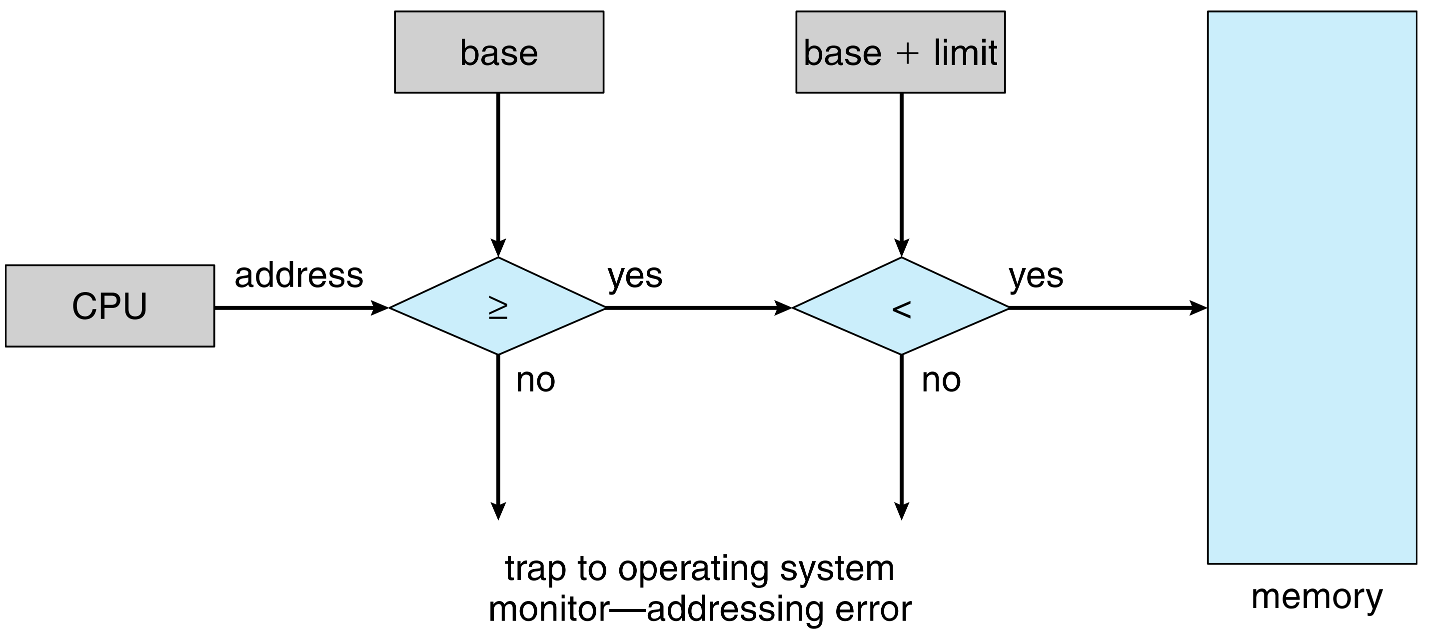
Base and Limit Registers

Separating per-process memory space protects the processes from each other and is fundamental to having multiple processes loaded in memory for concurrent execution.



Hardware Address Protection

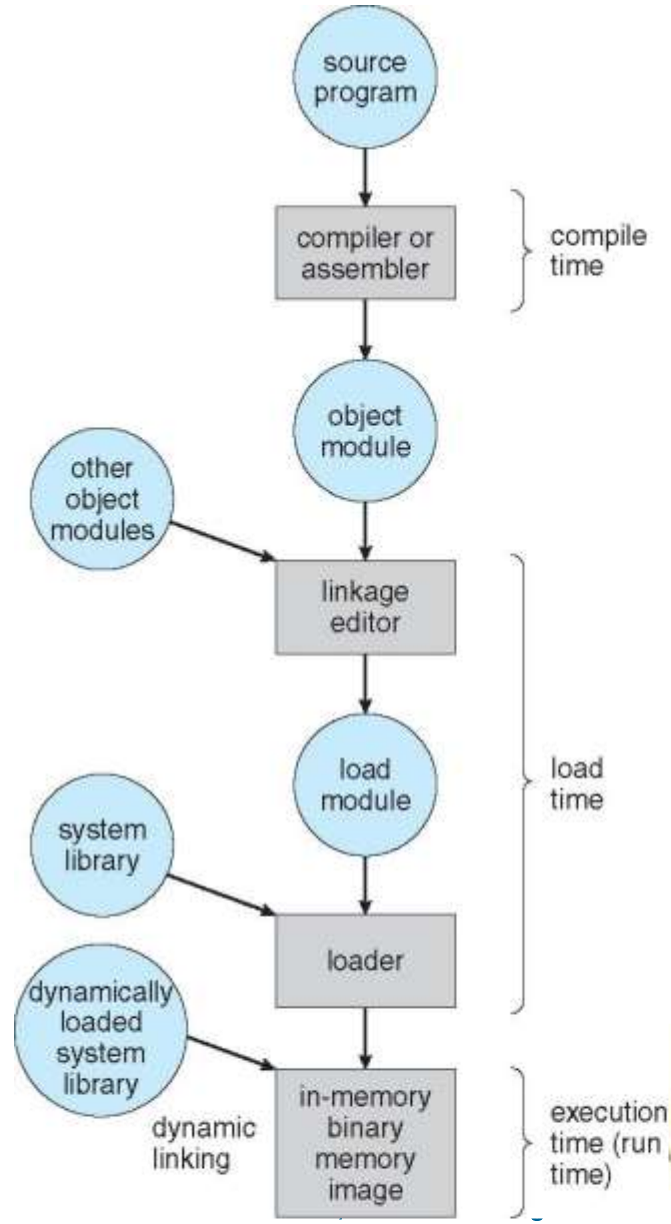
Having the CPU hardware compare every address generated in user mode with the registers



Prevents a user program from modifying the code or data structures of either the operating system or other users.

Address Binding

- Programs on disk, ready to be brought into memory to execute form an **input queue**
- In most cases, a user program goes through several steps—some of which may be optional—before being executed



Multistep Processing of a User Program

Address Binding

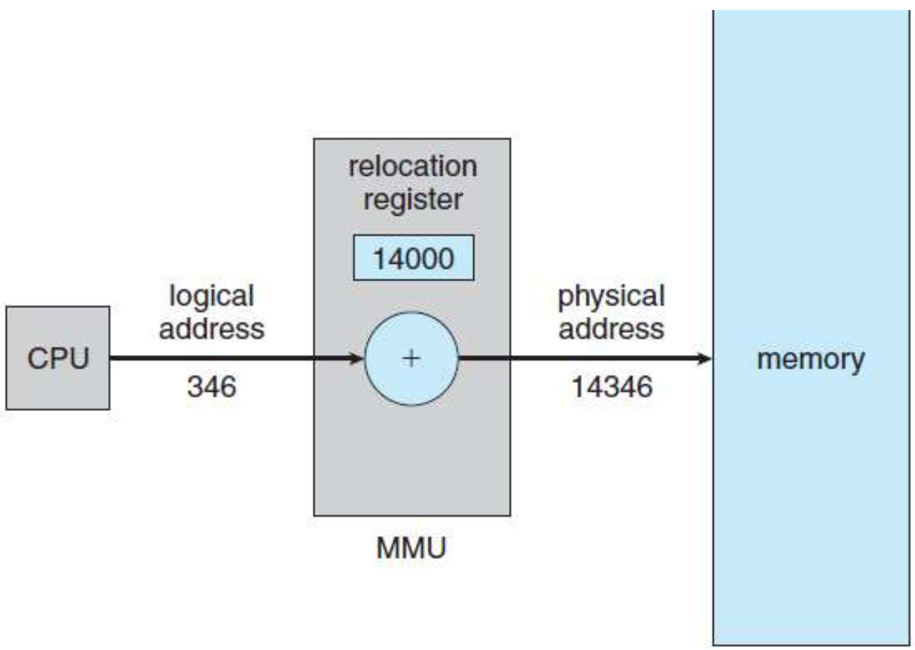
- Further, addresses represented in different ways at different stages of a program's life
  - Source code addresses usually **symbolic** (such as the variable *count*)
  - Compiler binds these **symbolic** address to **relocatable** addresses
    - i.e. "14 bytes from beginning of this module"
  - Linker or loader will bind **relocatable** addresses to **absolute** addresses
    - i.e. 74014
  - Each binding maps one address space to another

Logical vs. Physical Address Space

- **Logical address** – generated by the CPU; also referred to as virtual address
  - Tells how much memory a particular process will take, not tell what will the exact location of the process
- **Physical address** – address seen by the memory unit
  - The exact location of the process on physical memory
- **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)

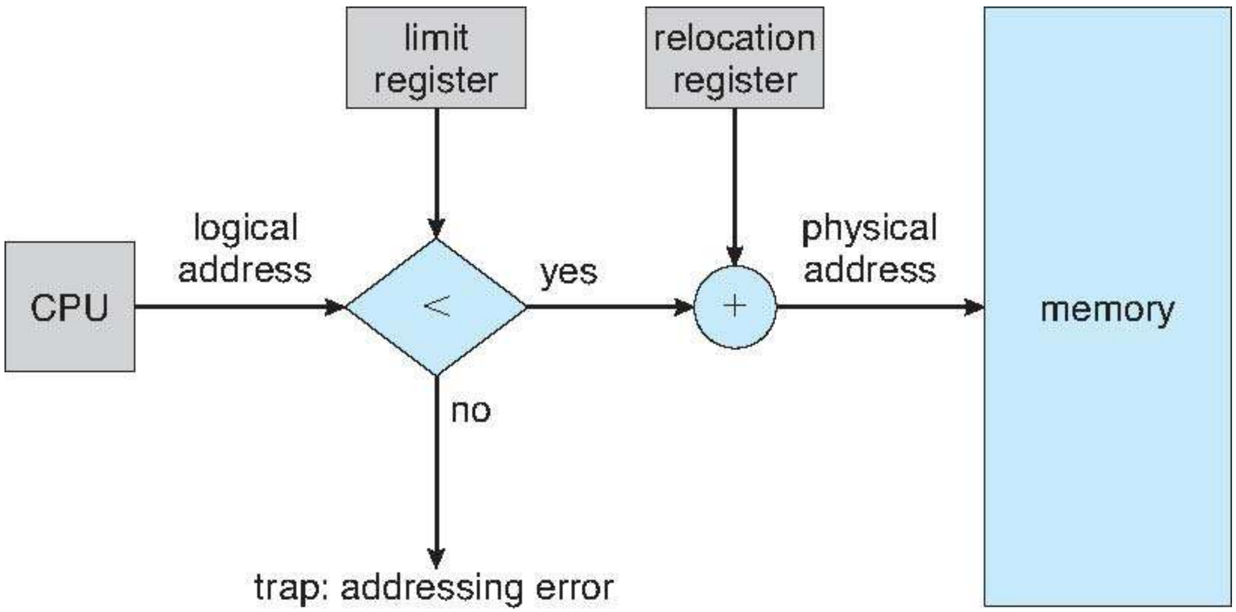
- **MMU**------**Hardware** device that **at run time** maps virtual address to physical address
- The user program deals with logical addresses; it never sees the real physical addresses
- To start, consider simple scheme where the value in the **relocation register** is added to every address generated by a user process at the time it is sent to memory
  - **Base register** now called **relocation register**



Memory Protection

- **Relocation** registers used to protect user processes from each other, from changing operating-system code and data
  - **Base** register contains value of smallest physical address
  - **Limit** register contains range of logical addresses – each logical address must be less than the limit register
  - **MMU** maps logical address *dynamically*

Hardware Support for Relocation and Limit Registers

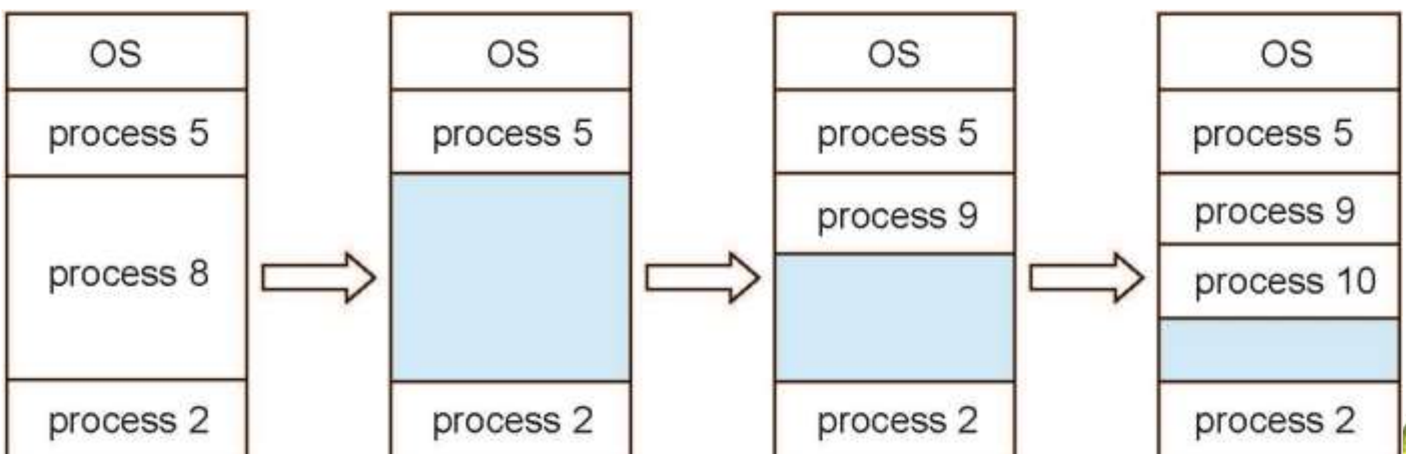


Multiple-partition allocation

- One of the simplest methods for allocating memory is to divide memory into several **fixed-sized** partitions ------ **Multiple-partition allocation**
  - Each partition may contain exactly one process
  - Degree of multiprogramming limited by number of partitions
  - In **multiple-partition**, when a partition is free, a process is selected from the input queue and is loaded into the free partition
  - When the process terminates, the partition becomes available for another process

Variable-partition allocation (cont.)

- In the **variable-partition scheme**, the operating system keeps a **table** indicating which parts of memory are available and which are occupied
  - **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

How to satisfy a request of size *n* from a list of free holes?

- **First-fit**: Allocate the ***first*** hole that is big enough
  - Stop searching as soon as we find a free hole that is large 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

Which one is faster?

Problems

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-

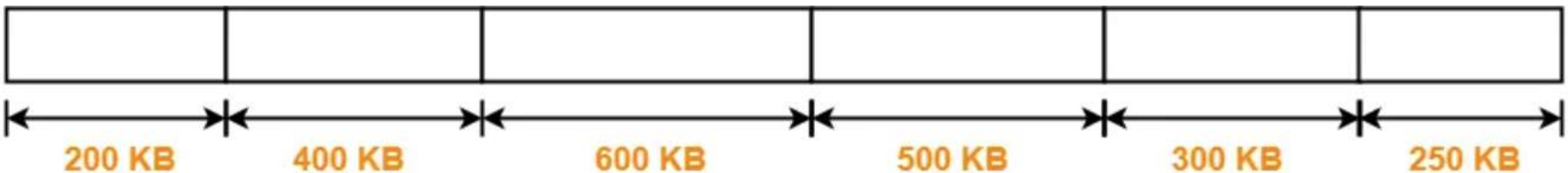
1. First Fit Algorithm
2. Best Fit Algorithm
3. Worst Fit Algorithm

Let us say the given processes are-

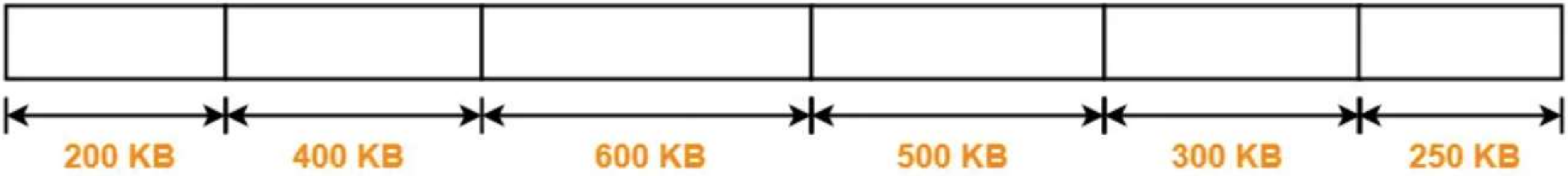
- Process P1 = 357 KB
- Process P2 = 210 KB
- Process P3 = 468 KB
- Process P4 = 491 KB

According to question,

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



Problems



Let us say the given processes are-

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

Step-04:

Fragmentation

- As processes are loaded and removed from memory, the free memory space is broken into little pieces.
- **External Fragmentation** – total memory space exists to satisfy a request, but it is not contiguous
  - Storage is fragmented into a large number of small holes
- **Internal Fragmentation**
  - A hole of 18,464 bytes, requests 18,462 bytes; so left with a hole 2 bytes
  - The overhead to keep track of this hole will be substantially larger than the hole itself
  - Solution: break the physical memory into fixed-sized blocks and allocate memory in units based on block size
  - Allocated memory may be slightly larger than requested memory; Unused memory that is internal to a partition

Fragmentation

- Reduce external fragmentation by **compaction**
  - **Shuffle** memory contents to place all free memory together in one large block

Segmentation

- User's view of memory is not the same as the actual physical memory
- Dealing with memory in terms of its physical properties is **inconvenient** to both the operating system and the programmer
- What if the hardware could provide a memory mechanism that mapped the programmer's view to the actual physical memory
- **Segmentation** provides such a mechanism

User's View of a Program

- User's view of a program: a program is a collection of segments
  - A segment is a logical unit such as:

    main program

    procedure

    function

    method

    object

    local variables, global variables

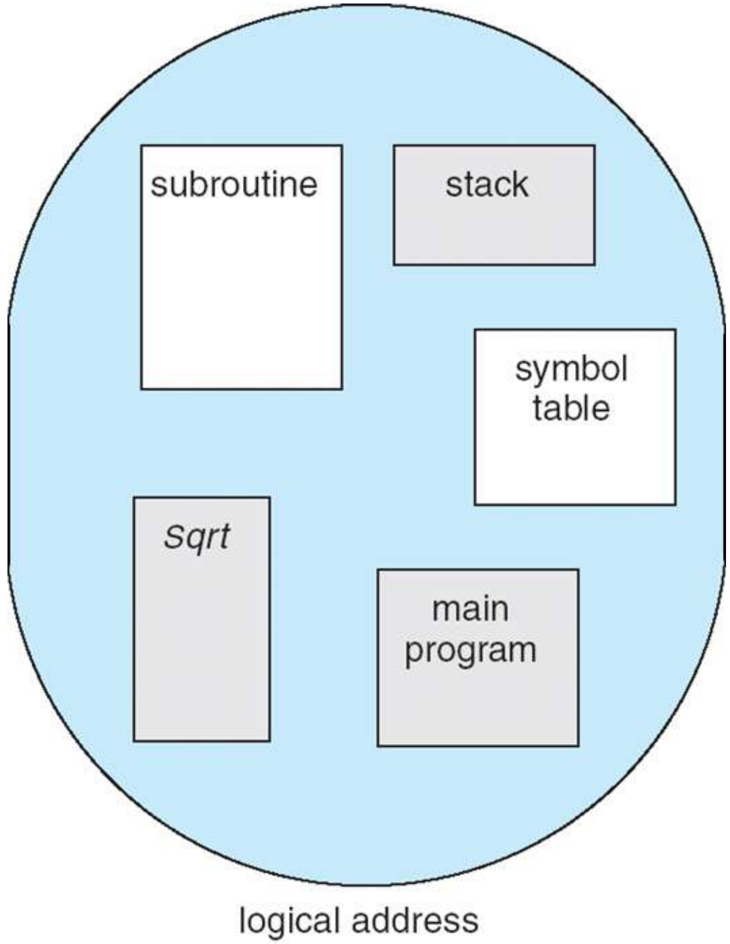
    common block

    stack

    symbol table

    arrays

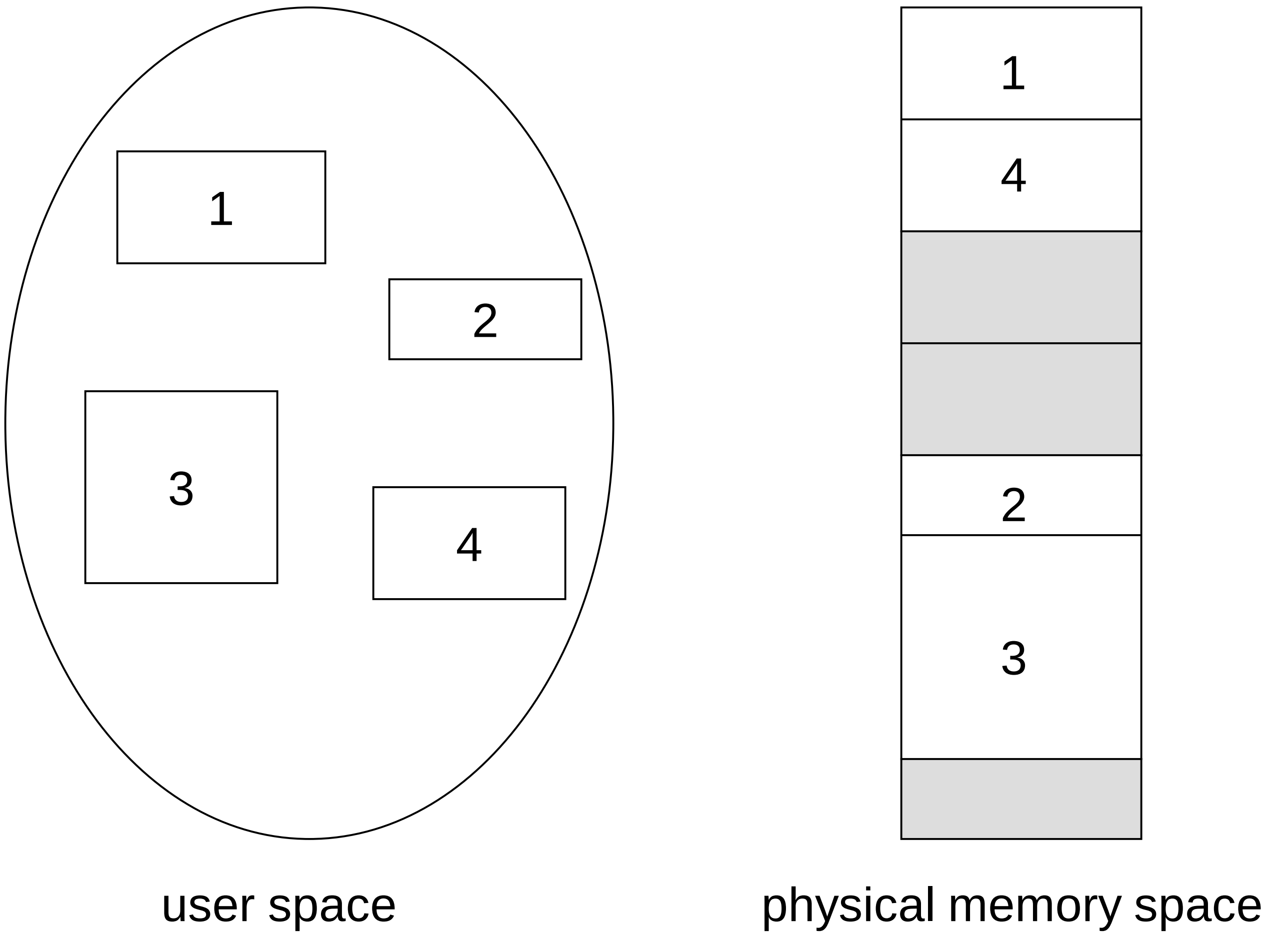
User's View of a Program (cont.)



No! view memory as a collection of segments

Do programmers think of memory as a linear array of bytes, some containing instructions and others containing data?

Physical View of Segmentation



- Memory-management scheme that supports programmer view of memory
  - Map the user's view to the actual physical memory

Segmentation Architecture

- A logical address space is a collection of segments
- Logical address of one segment specifies two tuple:

  $\langle \text{segment-number, offset within the segment} \rangle$

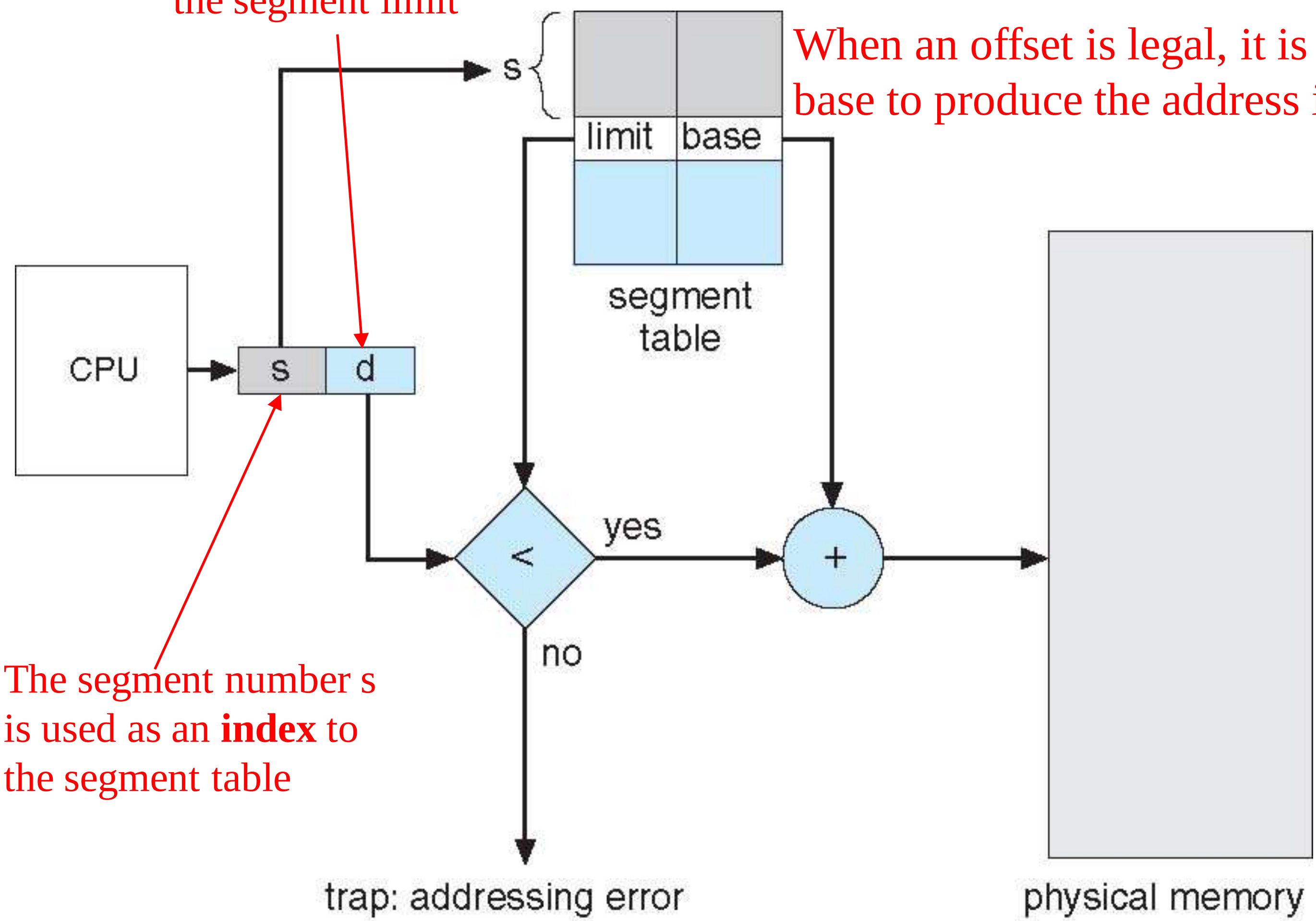
- Actual physical memory: still a one-dimensional sequence of bytes
- **Segment table** – maps two-dimensional logical addresses into physical address; each table entry has:
  - **base** – contains the starting physical address where the segments reside in memory
  - **limit** – specifies the length of the segment
- A segmentation example is shown next

Segmentation Hardware

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

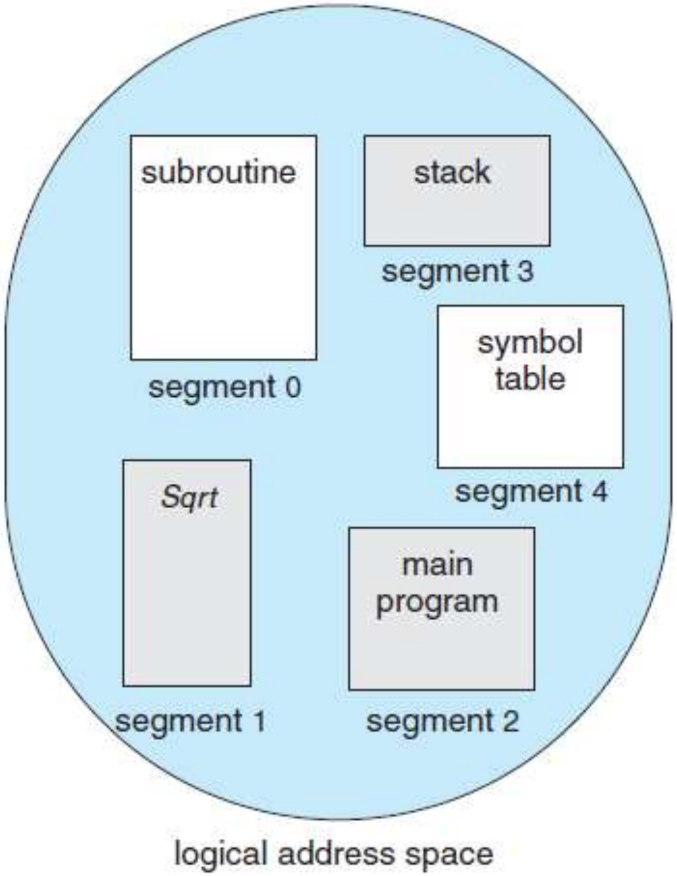
The offset d must be between 0 and the segment limit



The segment number s is used as an **index** to the segment table

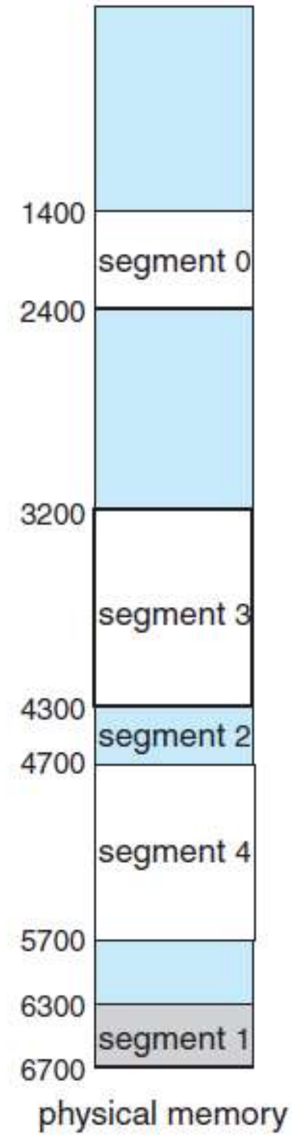
Logical address: *s*---segment number, *d*---offset

Example of Segmentation



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

segment table



A reference to segment 3, byte 852, where is it mapped to?

A reference to byte 1222 of segment 0, where is it mapped to?

Segment table:
base: the beginning address of the segment
limit: the length of that segment

Summary of Segmentation

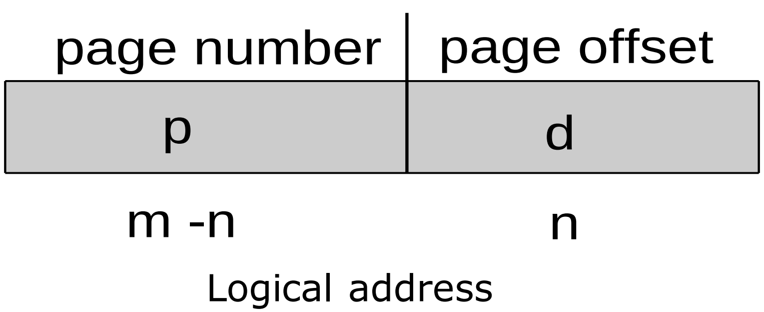
- A **non-contiguous** memory allocation scheme that supports the programmer view of memory
- Is there external fragmentation? or internal fragmentation?
- **Does not avoid external fragmentation!**

Paging

- Divide **physical memory** into **fixed-sized** blocks called **frames**
  - Size is power of 2
- Divide **logical memory** into blocks of **same size** called **pages**
  - Size is power of 2
- 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 logical memory
  - **Page offset** (***d***) – combined with base address to define the logical memory address that is sent to the memory unit

Address Translation Scheme

- The selection of a power of 2 as a page size makes **the translation** of a **logical address** into a **page number** and **page offset** particularly easy.
  - For given logical address space with size 2^m and page size 2^n
    - Then the high-order $m-n$ bits of a logical address designate the page number
    - and the n low-order bits designate the page offset.

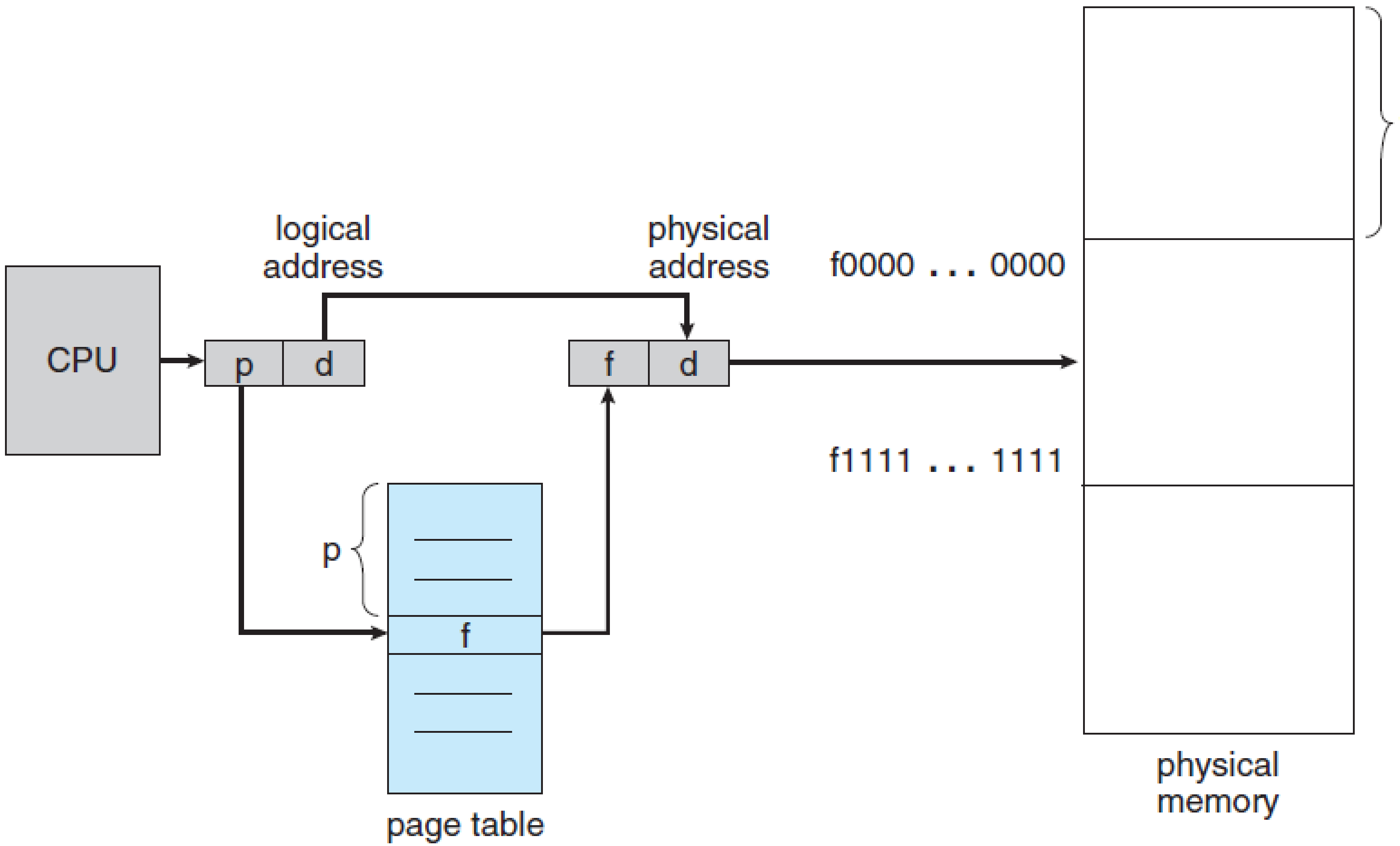


n=4 and m=16

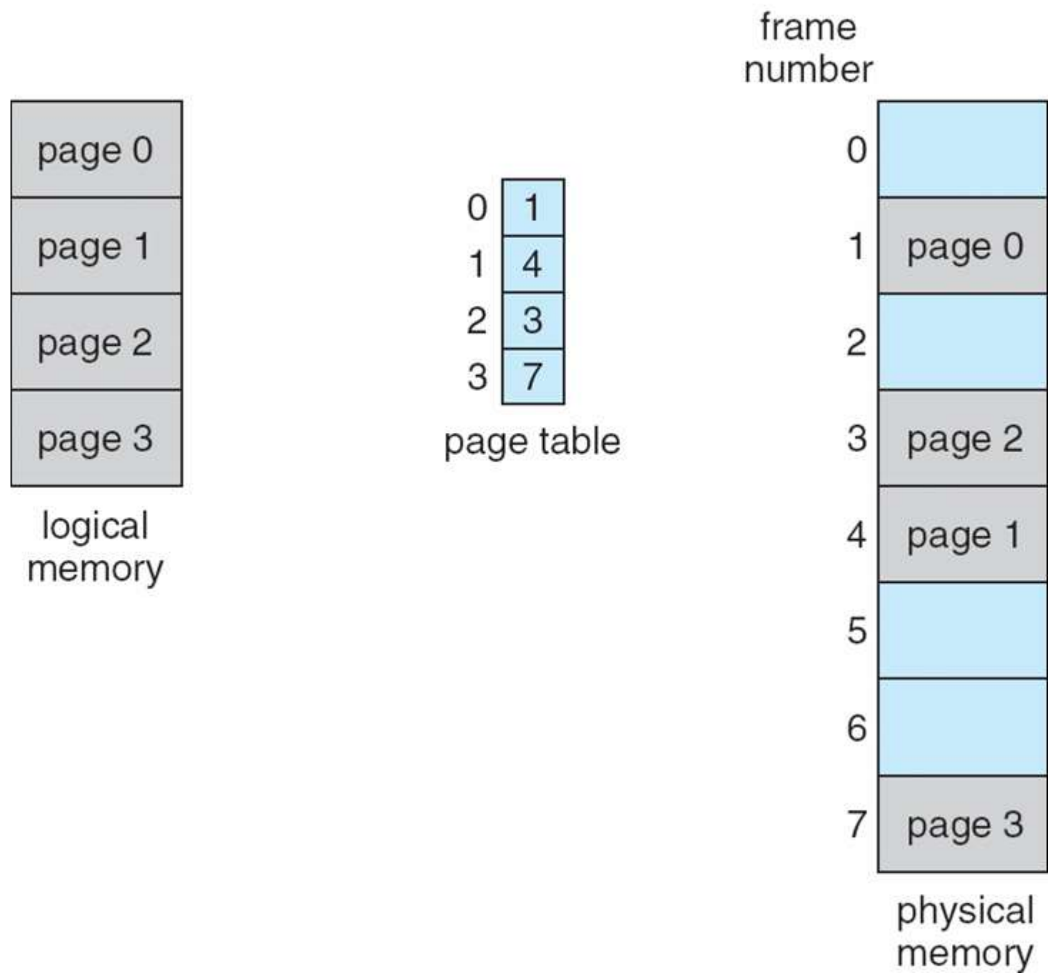
1. What's page size?
2. What's the number of pages?
3. What's the size of logical memory?

Paging Hardware

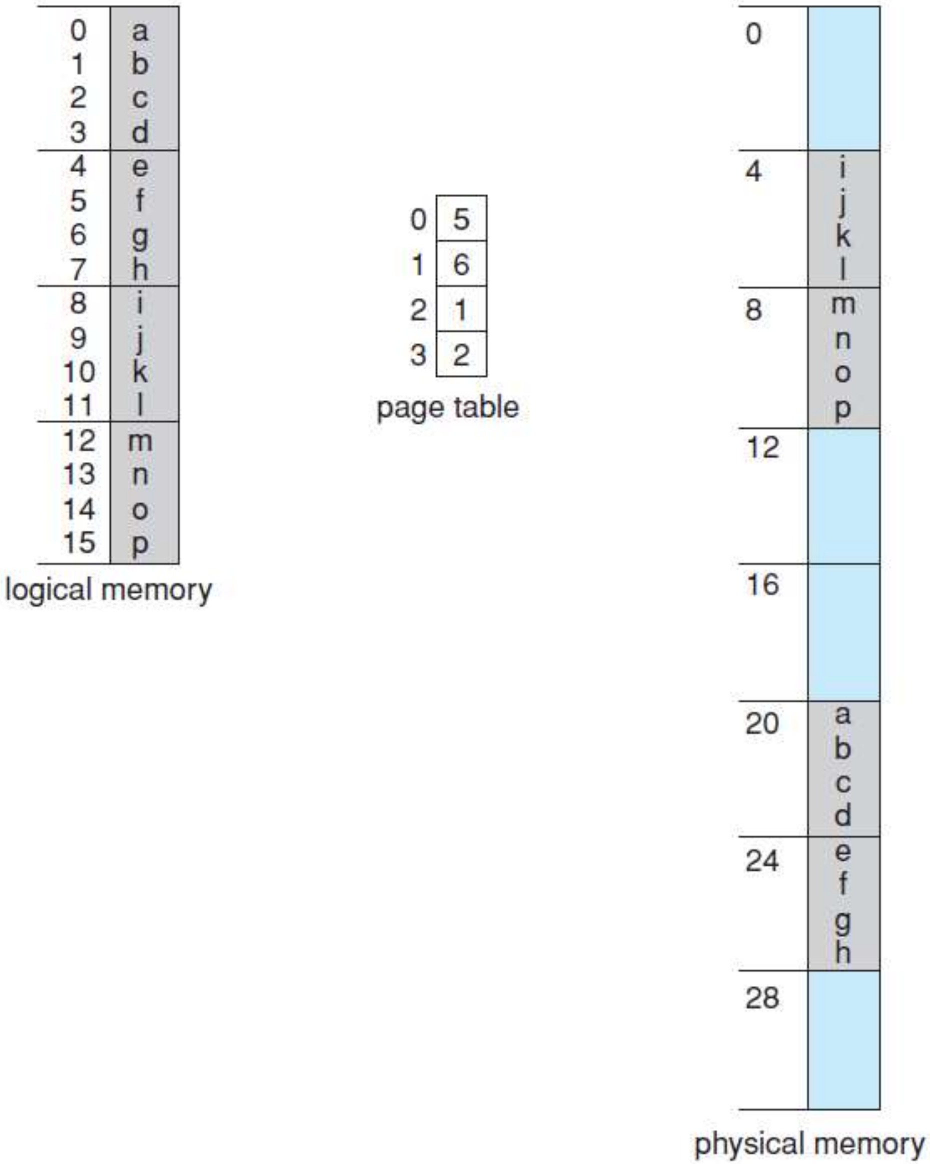
- To run a program of size ***N* pages**, need to find ***N* free frames** and load program
- Set up a **page table** to translate logical to physical addresses



Paging Model of Logical and Physical Memory



Paging Example (cont.)



Q1: What is the physical address for logical address 4

A1:

1. Logical address 5 is in page 1, offset 1
2. From page table, page 1 is frame 6
3. Thus, mapped to physical address 25 [= (6 × 4) + 1]

Q2: ……for logical address 3 13?

Paging Fragmentation Issue

- Is there any fragmentation issue for paging scheme?

- When we use a paging scheme, we have **no external fragmentation**
  - Any free frame can be allocated to a process that needs it
  - However, may have some **internal fragmentation**

Paging Fragmentation Issue (Cont.)

- An example, calculating internal fragmentation
  - Page size = 2,048 bytes
  - Process size = 72,766 bytes
  - 35 pages + 1,086 bytes
  - Have to allocate 36 frames: 36 = 35 + 1
  - Internal fragmentation of 2,048 - 1,086 = 962 bytes
- So small frame sizes desirable?
  - Overhead is involved in each page-table entry

Paging

- Process is allocated **physical memory** whenever the latter is available
  - Permits the physical address space of a process to be **non-contiguous**
- Paging is the clear separation between the programmer's view of memory and the actual physical memory
- By implementation, user process can only access its own memory

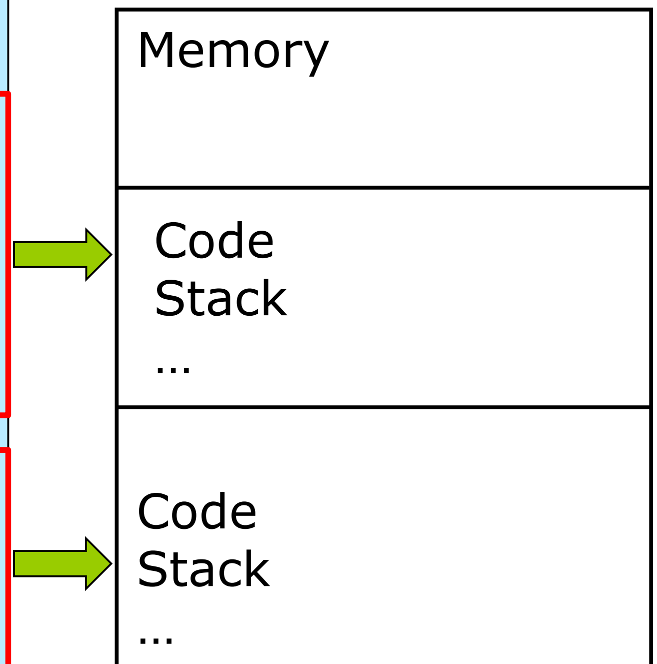
Segmentation V.S. Paging

```c
// C
bool checkPositive(int a,  int b){
   return  (a>=0  &&  b>=0);
}

int add(int x,  int  y){
   bool  chk = checkPositive(x, y);
   if (chk)
      return   x+y
   return -1;
}

int main() {
    int   a=5;   b=6;
    int   c = add(a,  b);
    printf("%d", c );
    return 0;
}
```

However, in **Paging**, everything is scattered in **various frames** in physical memory.

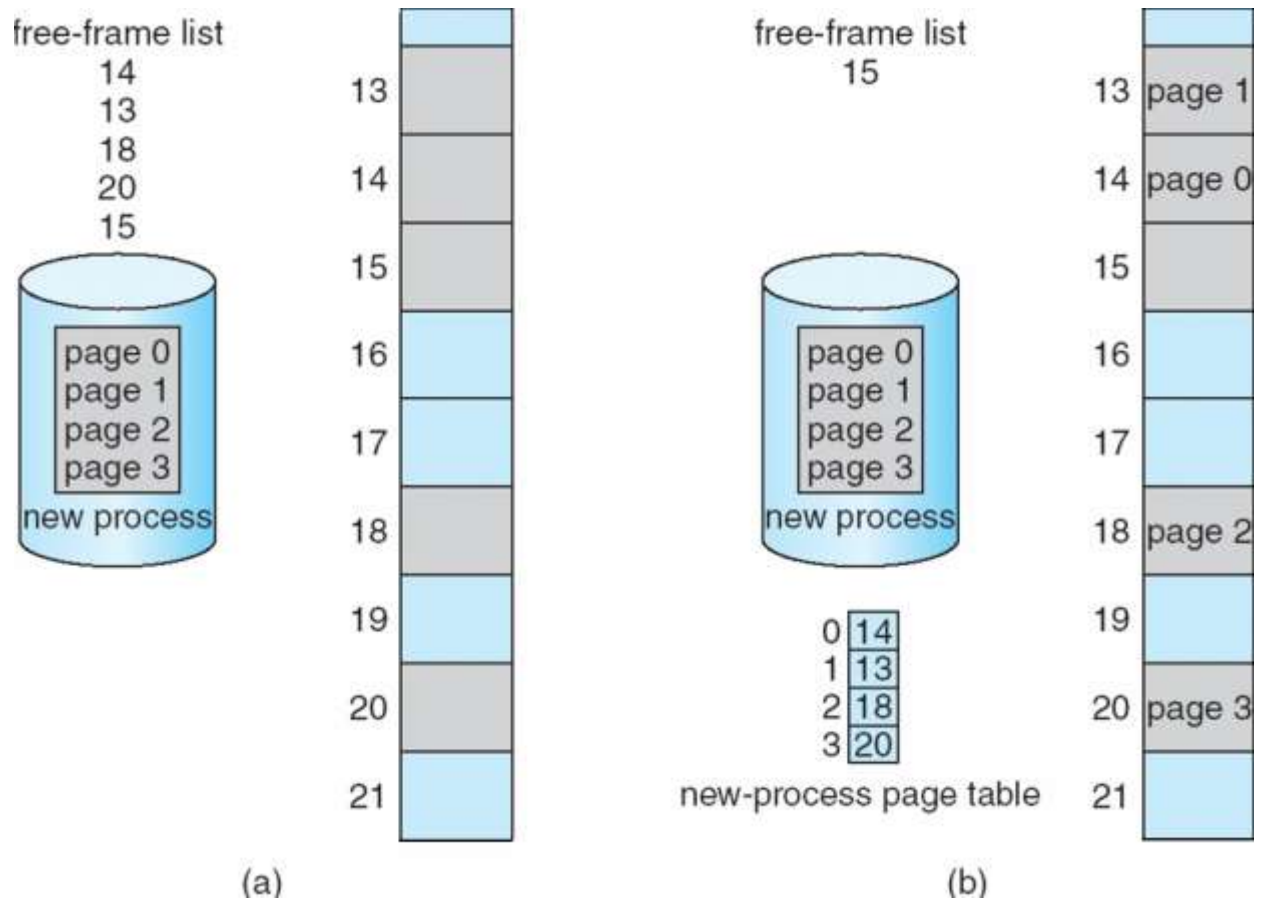


Then in **Segmentation**, the programmer can easily give a **logical address** to locate an element: "the fifth instruction of the memory module of function"

Segmentation V.S. Paging

- Each page of the process needs one frame. Thus, if the process requires *n* pages, at least *n* frames must be available in memory.
- If n frames are available, they are allocated to this arriving process.
- The first page of the process is loaded into one of the allocated frames, and the frame number is put in the page table for this process

Free Frames



Before allocation

After allocation

When one process is scheduled for CPU, where to find its page table?

Implementation of Page Table

Implementation of Page Table

- Page table is kept in **main memory**
- **Page-table base register** (**PTBR**) points to the page table
  - Changing page tables requires changing only this one register, substantially **reducing context-switch time**
- **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
- How to solve the two-memory-accesses problem?

Implementation of Page Table

- How to solve the two-memory-accesses problem

- 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**)

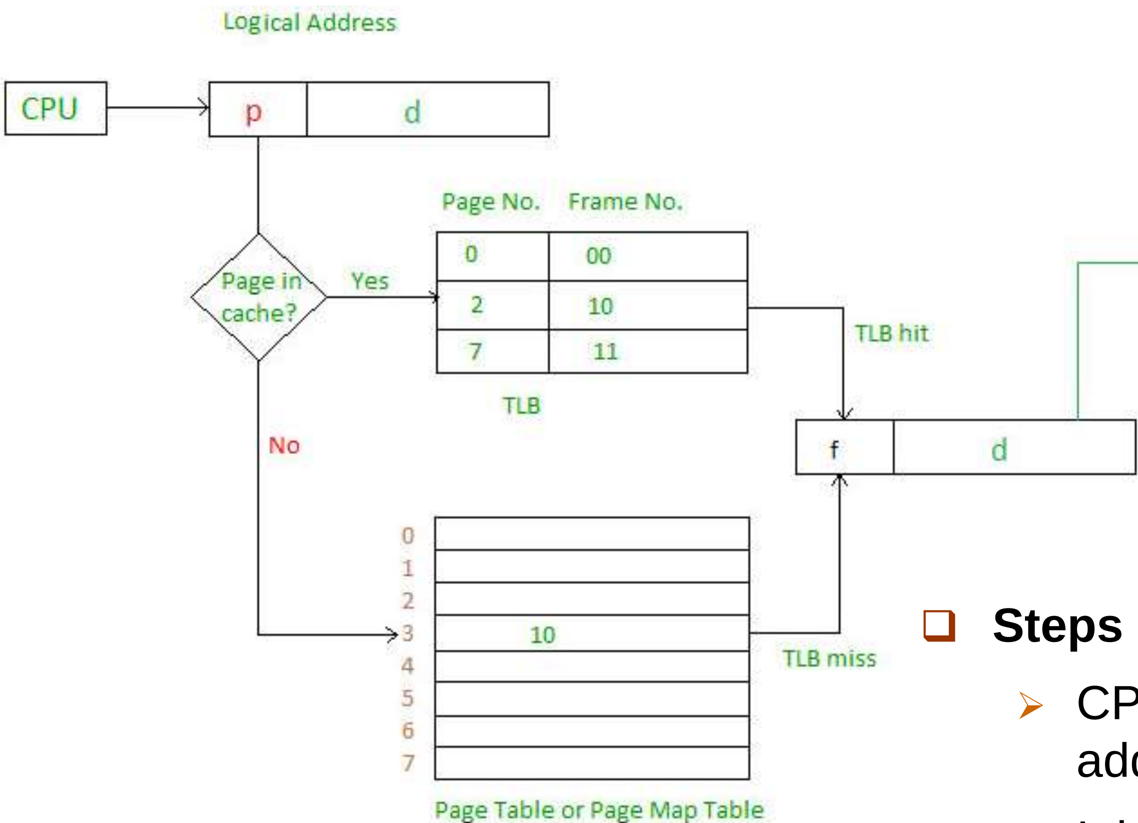
Implementation of Page Table

- TLB contains page table entries that have been **most recently used.**
  - TLBs typically small (64 to 1,024 entries)

Valid	Virtual page	Modified	Protection	Page frame
1	140	1	RW	31
1	20	0	R X	38
1	130	1	RW	29
1	129	1	RW	62
1	19	0	R X	50
1	21	0	R X	45
1	860	1	RW	14
1	861	1	RW	75

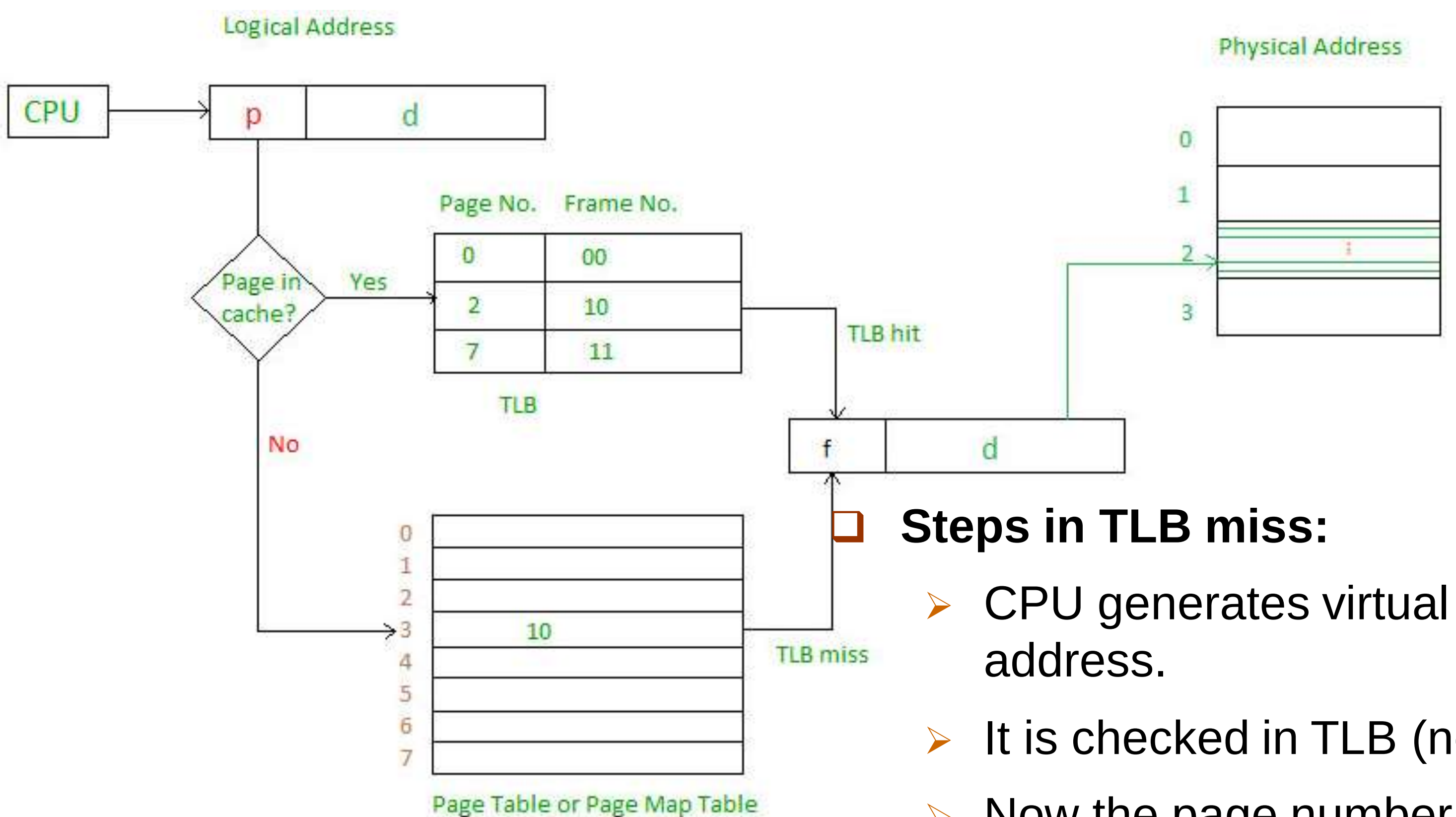
A TLB to speed up paging

Implementation of Page Table



- **Steps in TLB hit:**
  - CPU generates virtual (logical) address.
  - It is checked in TLB (present).
  - Corresponding frame number is retrieved, which now tells where in the main memory page lies.

Implementation of Page Table



- **Steps in TLB miss:**
  - CPU generates virtual (logical) address.
  - It is checked in TLB (not present).
  - Now the page number is matched to page table residing in main memory
  - Corresponding frame number is retrieved, which now tells where in the main memory page lies.
  - The TLB is updated

Implementation of Page Table

Effective memory access time (EMAT) : TLB is used to reduce effective memory access time as it is a high-speed associative **cache**.

$$\mathbf{EMAT} = \boldsymbol{h*(c+m) + (1-h)*(c+2m)}$$

where, h = hit ratio of TLB

m = Memory access time

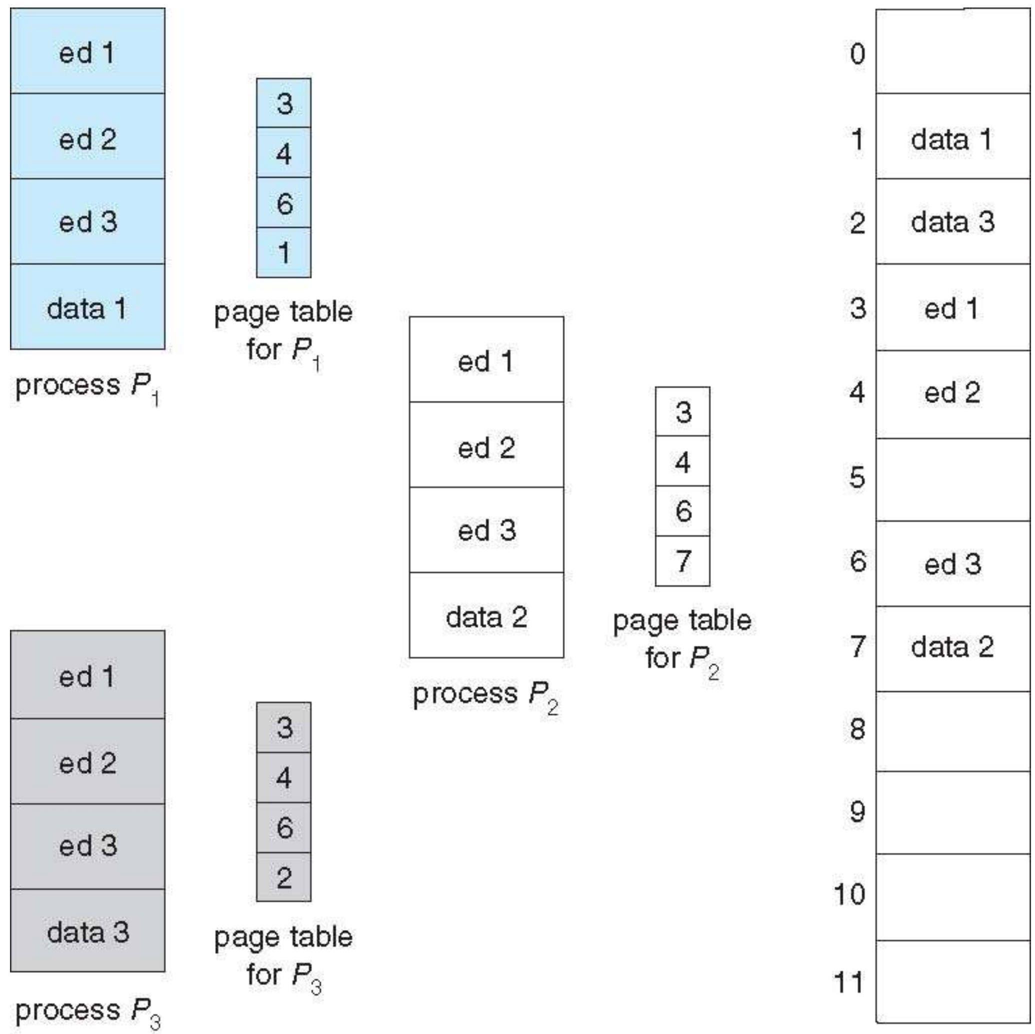
c = TLB access time

Shared Pages

- Consider a system that supports 40 users, each of whom executes a text editor
  - If the text editor consists of 150 KB of code and 50 KB of data space
  - Then we need 8,000 KB to support the 40 users

- **Shared code**
  - One copy of read-only code shared among processes (i.e., text editors, compilers, window systems)
    - non-self-modifying code; never changes during execution
  - Similar to multiple threads sharing the same process space
  - Two or more processes can execute the same code at the same time
  - Only one copy of the editor need be kept in physical memory

Shared Pages Example



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} bits)
  - Page table would have 1 million entries (2^{32} / 2^{12})
    - That amount of memory used to cost a lot
    - Don't want to allocate that **contiguously** in main memory

- Hierarchical Paging
- Hashed Page Tables
- Inverted Page Tables

Single-Level Page Tables

Virtual Address (VA): 32 bits

Offset field in VA: 12 bits

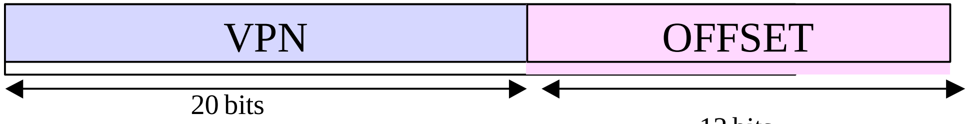
Virtual Page Number field in VA: 32 - 12 = 20 bits

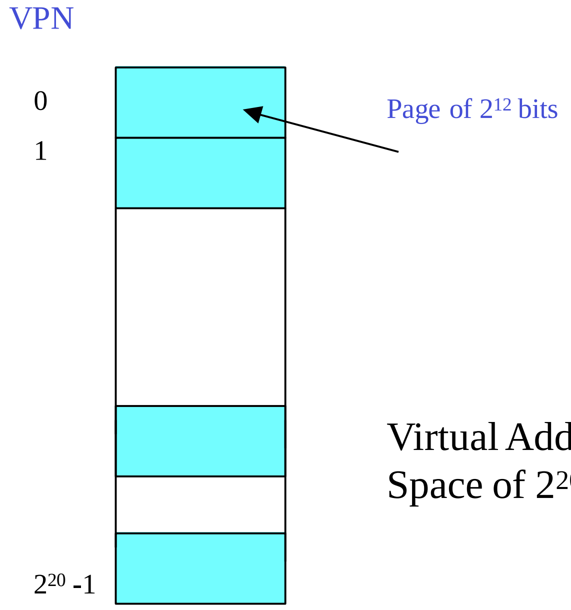
Virtual Address Space: 2^{32} bits

Page Size: 2^{12} bits = 4Kb

Number of Virtual Pages: $2^{32} / 2^{12} = 2^{20}$

VA:





Virtual Address
Space of 2^{20} pages

Single-Level Page Tables

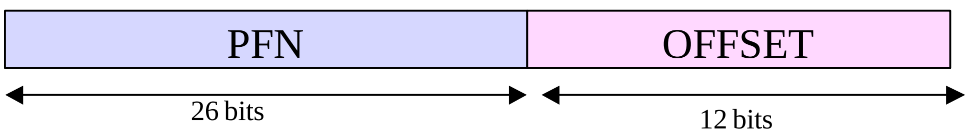
PA:

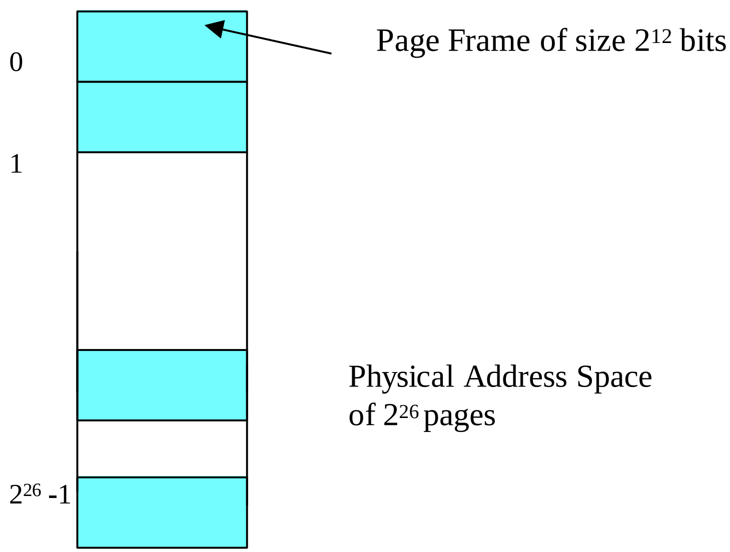
Physical Address (PA): 38 bits
Offset field in PA: 12 bits

Physical Address Space: 2^{38} bits
Page Size: 2^{12} bits = 4Kb

Page Frame Number field in PA: 38 - 12 = 26 bits

Number of Physical Pages: $2^{38} / 2^{12} = 2^{26}$

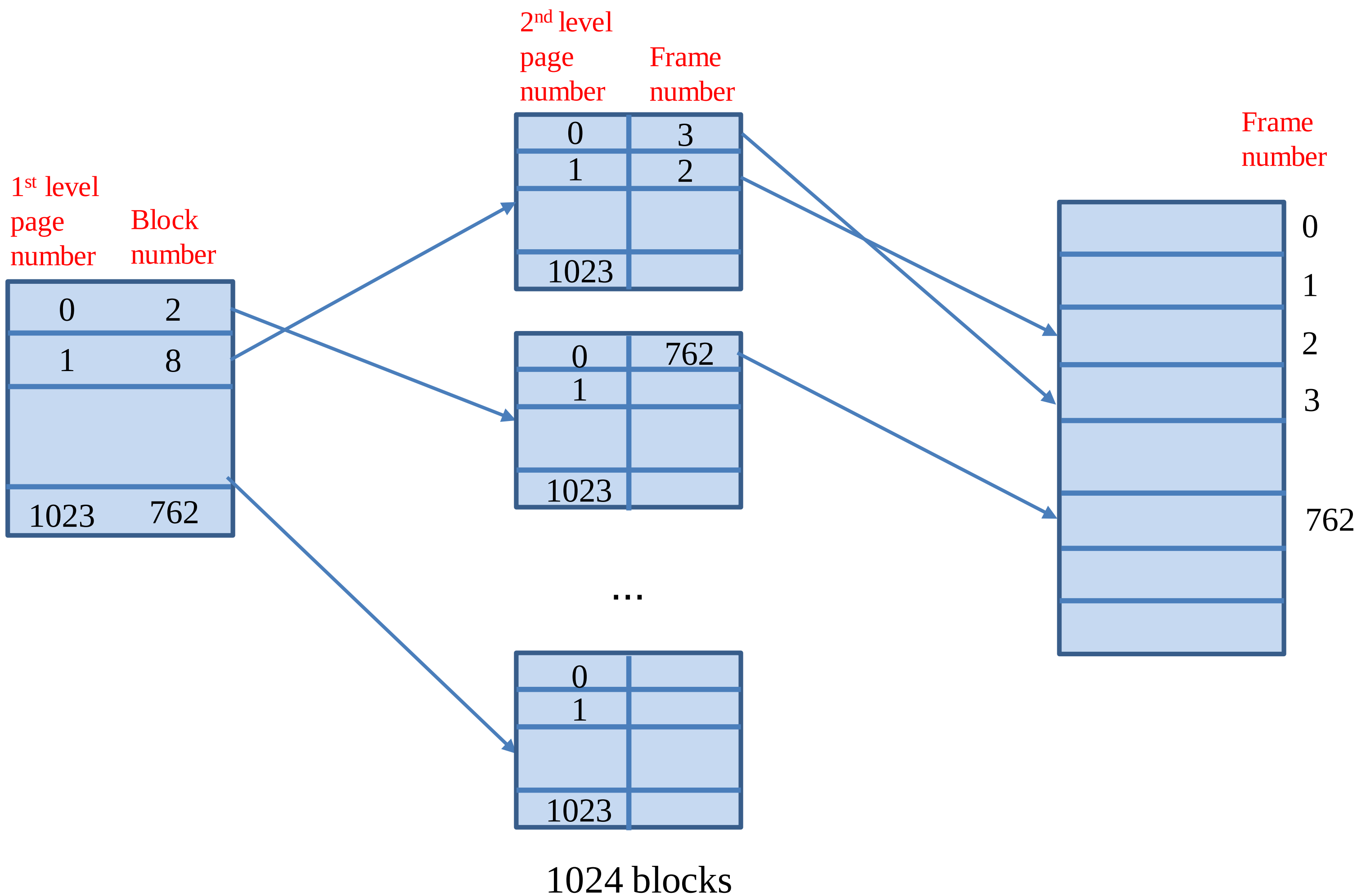


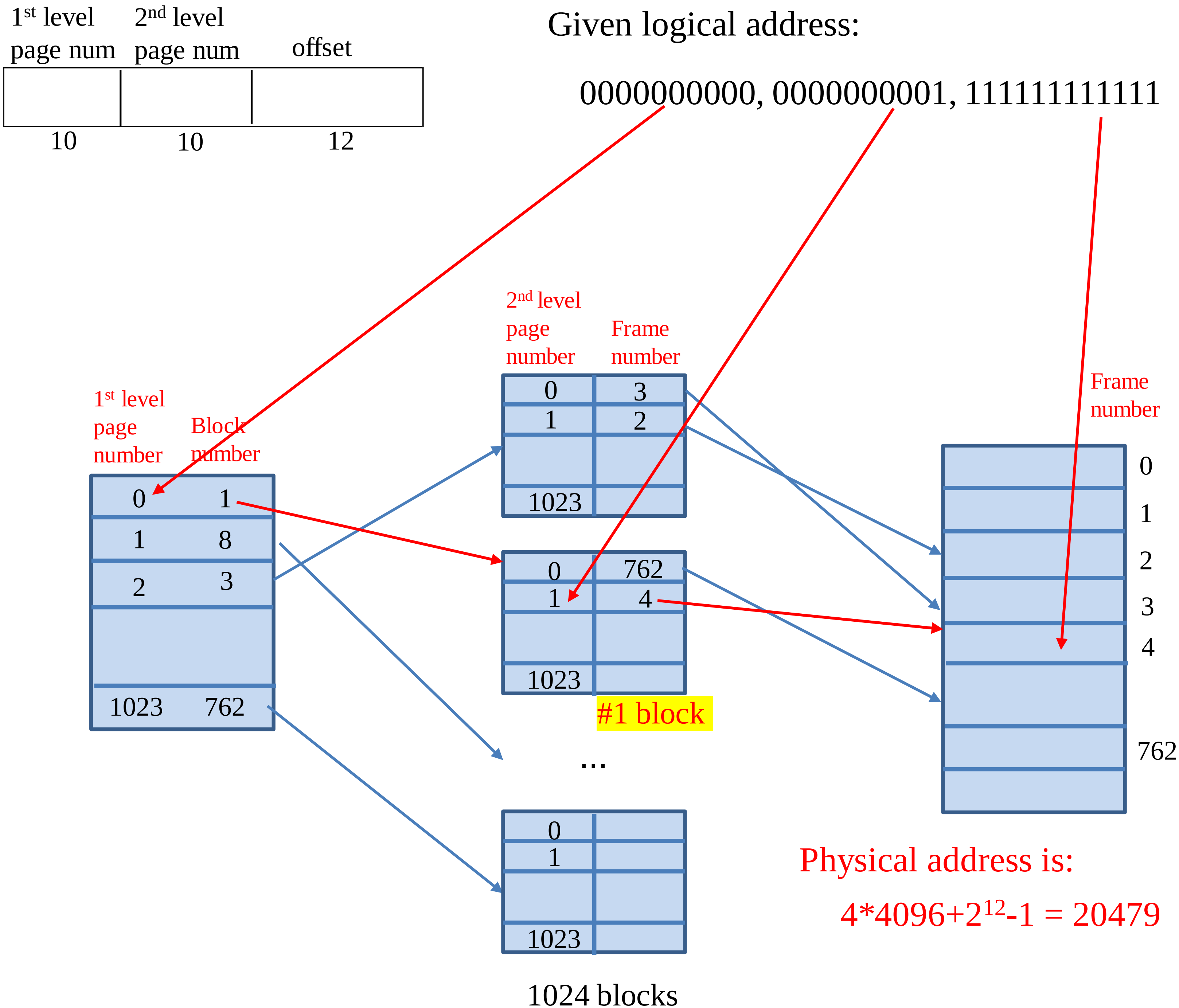


Two-Level Page Tables

- Break up Page Table into fixed-size blocks of the same size as a page
- In example: Each block is 4Kb and Page Table is 4Mb

  So we will have 4Mb/4Kb = 2^{10} = 1024 such blocks





1st level page num
2nd level page num
offset
10
10
12
Given logical address:
0000000000, 0000000001, 111111111111
1st level page number
Block number
0 1
1 8
2 3
1023 762
2nd level page number
Frame number
0 3
1 2
1023
0 762
1 4
1023
#1 block
...
0
1
1023
1024 blocks
Frame number
0
1
2
3
4
762
Physical address is:
4*4096+2^12-1 = 20479

1st level page num	2nd level page num	offset
10	10	12

Given logical address:

000000010, 0000000001, 111111111110

1st level page number	Block number
0	1
1	8
2	3
…	
1023	762

2nd level page number	Frame number
0	3
1	2
…	…
1023	

#1 block

…

2nd level page number	Frame number
0	762
1	3
…	…
1023	

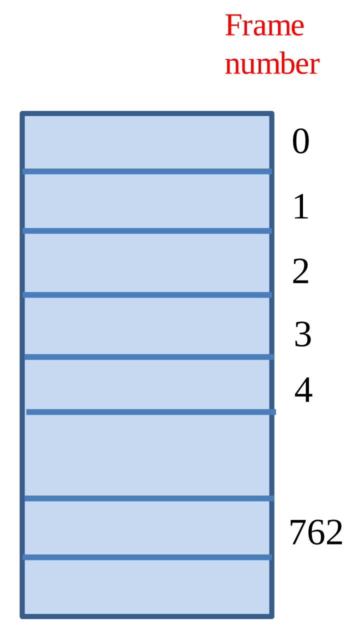
#3 block

…

2nd level page number	Frame number
0	
1	
1023	

#1023 block

1024 blocks



Two-Level Page Tables

What is the advantage to introduce the 2nd -level page tables?

Do not need to store the entire 2nd level Page Table as a **contiguous** array

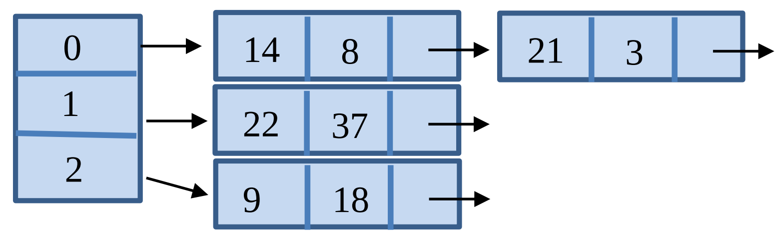
Hashed Page Tables

- Given page number: 14, 21, 35, 49, 9, 22
- Then the page table is:

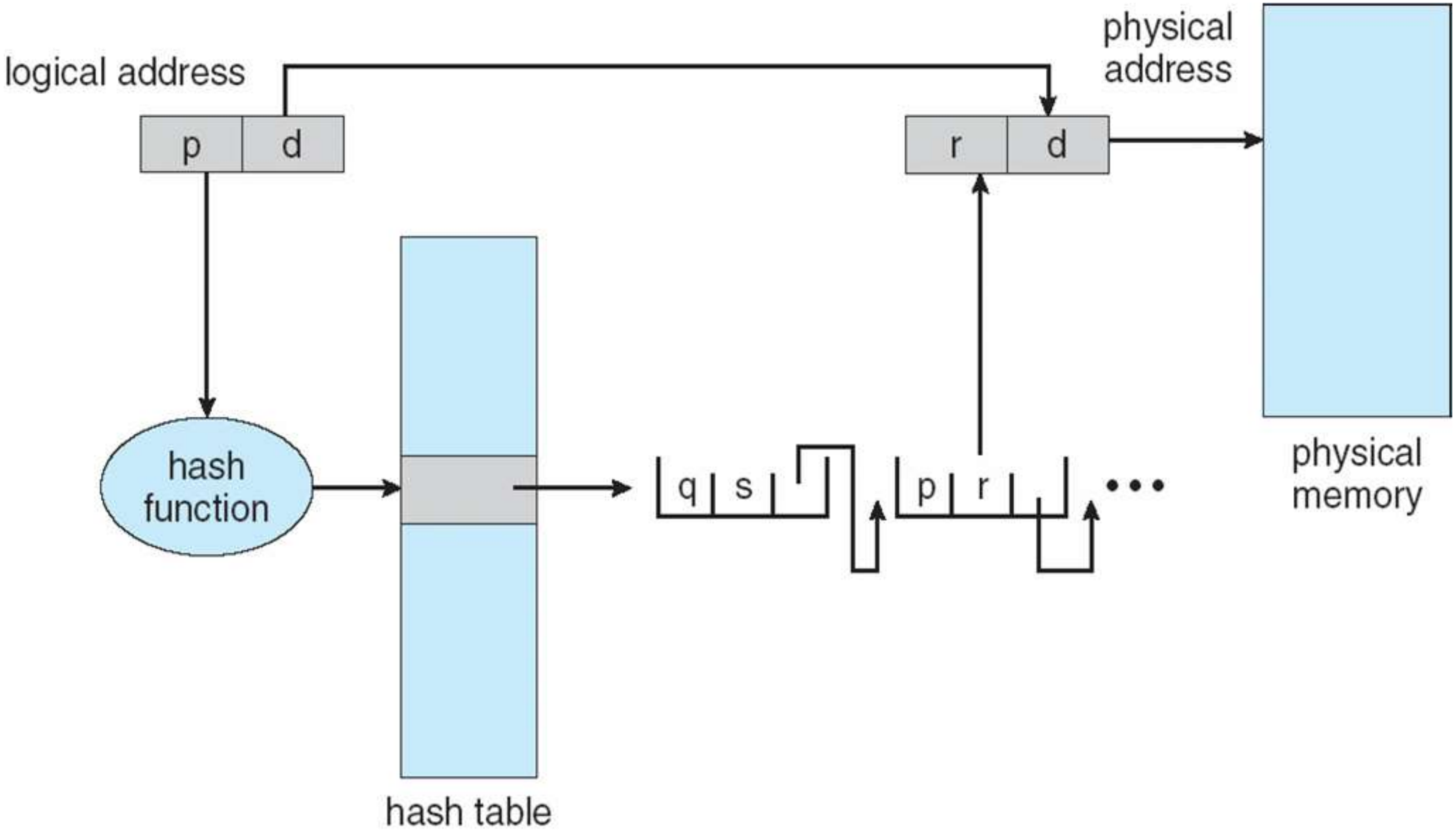
page number	Frame number
35	1
14	8
21	3
9	18
22	37
49	762

Is it possible to reduce the size of the page table via better organizing these entries?

- 14%7 = 0
- 21%7 = 0
- 35%7 = 0
- 49%7 = 0
- 9%7 = 2
- 22%7 =1



Hashed Page Table



Hashed Page Tables

- The **page number** is hashed into a **page table**
  - This page table contains **a chain of** elements hashing to the same location
- Each element contains
  - (1) the **page number**
  - (2) the value of the mapped **frame number**
  - (3) a **pointer** to the next element
- Page numbers are compared in this chain searching for a match
  - If a match is found, the corresponding physical frame is extracted

Page Table Size Grows Dramatically

- Most operating systems implement **a separate page table for each process**
  - When a process is vast in size and takes up a lot of virtual memory, the **page table size** grows dramatically.
- Example: A process of size 2 GB with:
  - Page size = 512 Bytes
  - Size of page table entry = 4 Bytes, then
  - Number of pages in the process = 2 GB / 512 B = 2^{22}
  - Page Table Size = $2^{22} * 2^{2} = 2^{24}$ bytes
- When numerous processes are operating in an OS simultaneously, page tables take up a significant amount of memory.

Page Table Size Grows Dramatically

- **Multilevel paging** strategies increase the amount of space necessary for storing page tables.

- The amount of memory occupied by page tables can be a significant **overhead**, which is always undesirable because main memory is always a limited resource.

Inverted Page Table

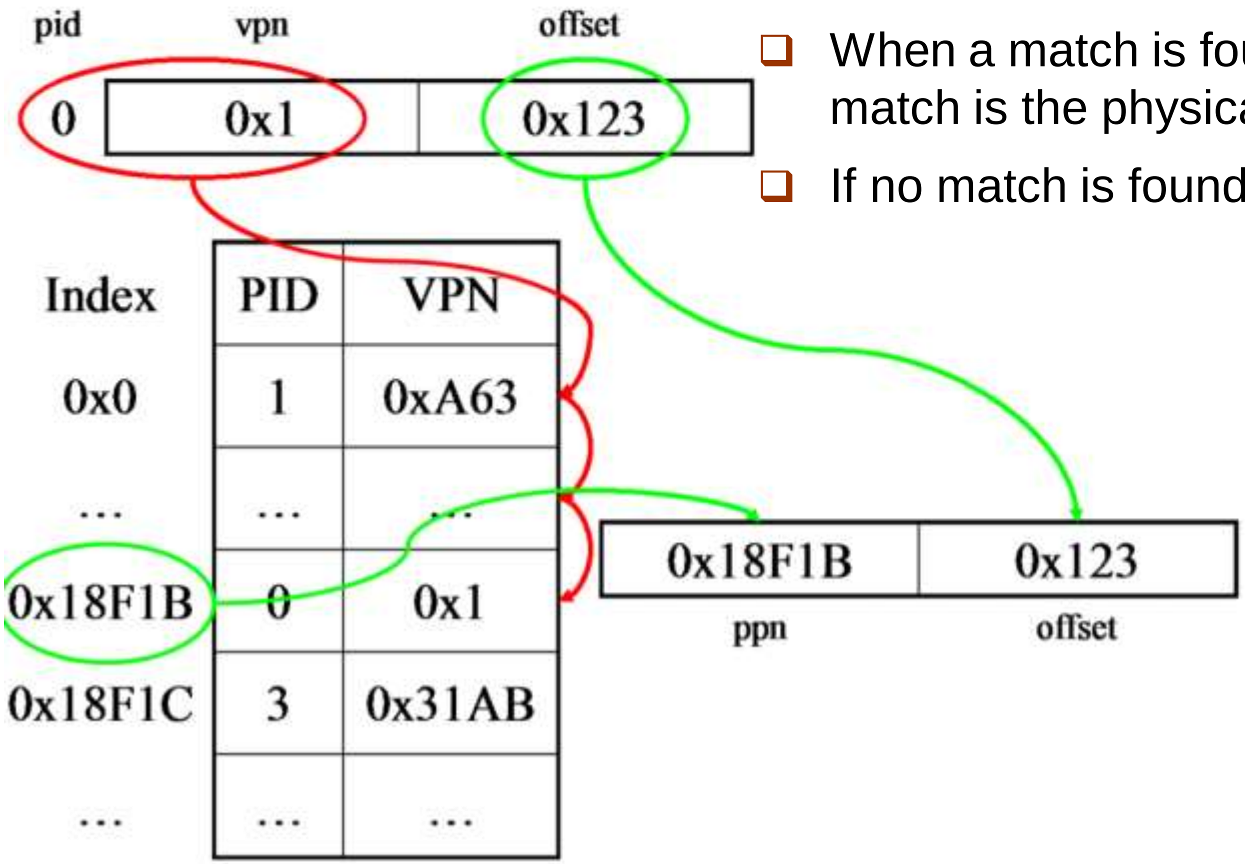
- Rather than each process having a page table and keeping track of all possible logical pages, track all **physical pages**

- **One entry for each real frame of memory**
  - The number of page table entries is reduced to the number of frames available in physical memory.
  - A single-page table represents all processes' paging information.

- Since the table is shared, each entry must contain the process ID of the page owner

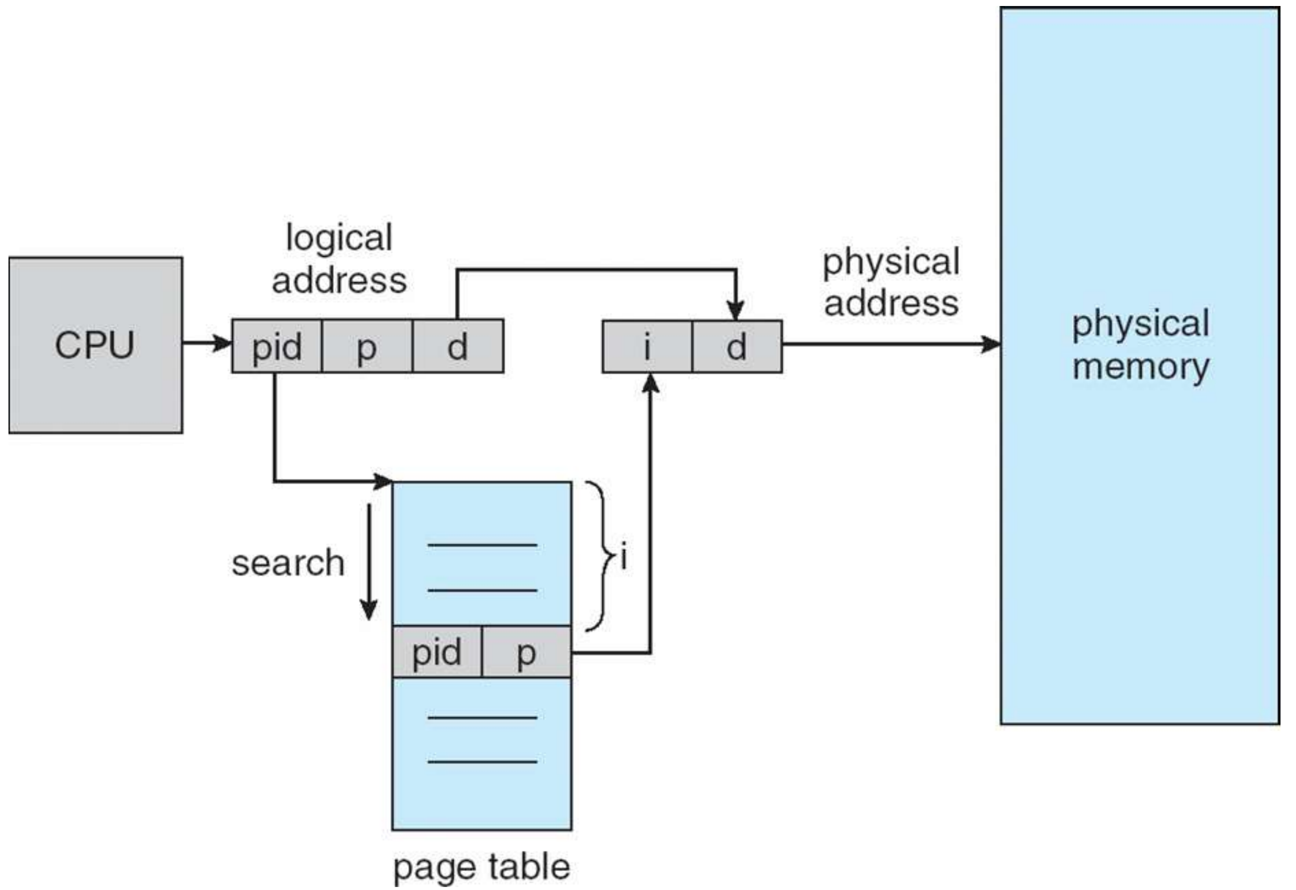
- And since physical pages are now mapped to logical, each entry contains a logical page number instead of a physical.

Inverted Page Table



- In order to translate a logical address, the **page number** and **process ID** are compared against each entry, traversing the array sequentially.
- When a match is found, the index of the match is the physical address.
- If no match is found, a page fault occurs.

Inverted Page Table Architecture



Why can the inverted page table save memory usage?

If we have two processes which both have 4 pages, we would have 8 entries in two different tables pointing from virtual to physical address:

Process 1:	Process 2:
[0] = 1	[20] = 14
[1] = 5	[21] = 55
[2] = 63	[22] = 11
[3] = 0	[25] = 9

If we would use inverted page tables we would only have one big table pointing it the other way around. But in size they equal.

[0] = <p1 | 3>
[1] = <p1 | 0>
[5] = <p1 | 1>
[9] = <p2 | 25>
[11]= <p2 | 22>
[14]= <p2 | 20>
[55]= <p2 | 21>
[63]= <p1 | 2>

Virtual Memory

In order to execute any process, it is **not** necessary that the **whole** process should present in the main memory at the given time.

The process can also be executed if **only some pages** are present in the main memory at any given time.

But, how can we decide beforehand **which page should be present** in the main memory at a particular time and **which should not be there**?

Background

- Code needs to be in memory to execute, but entire program rarely used
  - error handling code
  - large data structures
  - unusual routines
  - Certain features of certain programs are rarely used.

Background

- Entire program code not needed **at same time**
- Consider ability to execute **partially-loaded program**
  - Program no longer constrained by **limits of physical memory**
  - Each program takes less memory while running -> **more programs** run at the same time
    - Increased CPU utilization and throughput with no increase in response time or turnaround time
  - **Less I/O** needed to load or swap programs into memory -> each user program runs faster

Background (Cont.)

- **Virtual memory** – **separation** of user logical memory from physical memory
  - 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

Background (Cont.)

- But, how can we decide beforehand which page should be present in the main memory at a particular time and which should not be there?

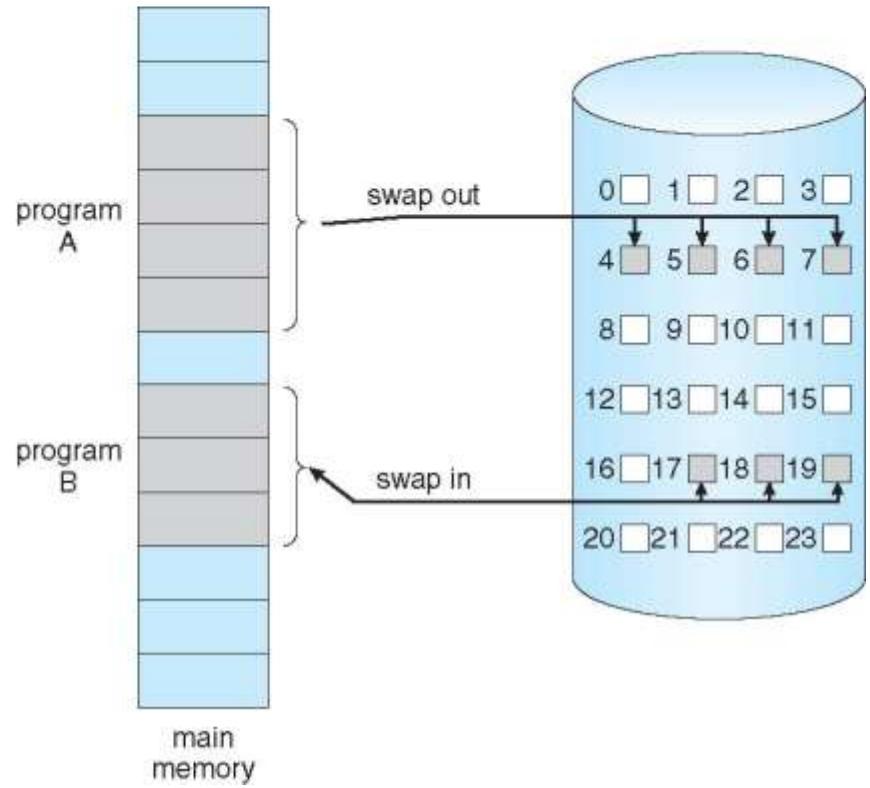
- Virtual memory can be implemented via:
  - **Demand paging**

Demand Paging

- We should not load any page into the main memory until required or we should keep all the pages in secondary memory until demanded.
- **Demand paging** is a technique used in virtual memory systems where the pages are brought in the main memory only when required or demanded by the CPU.
- 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

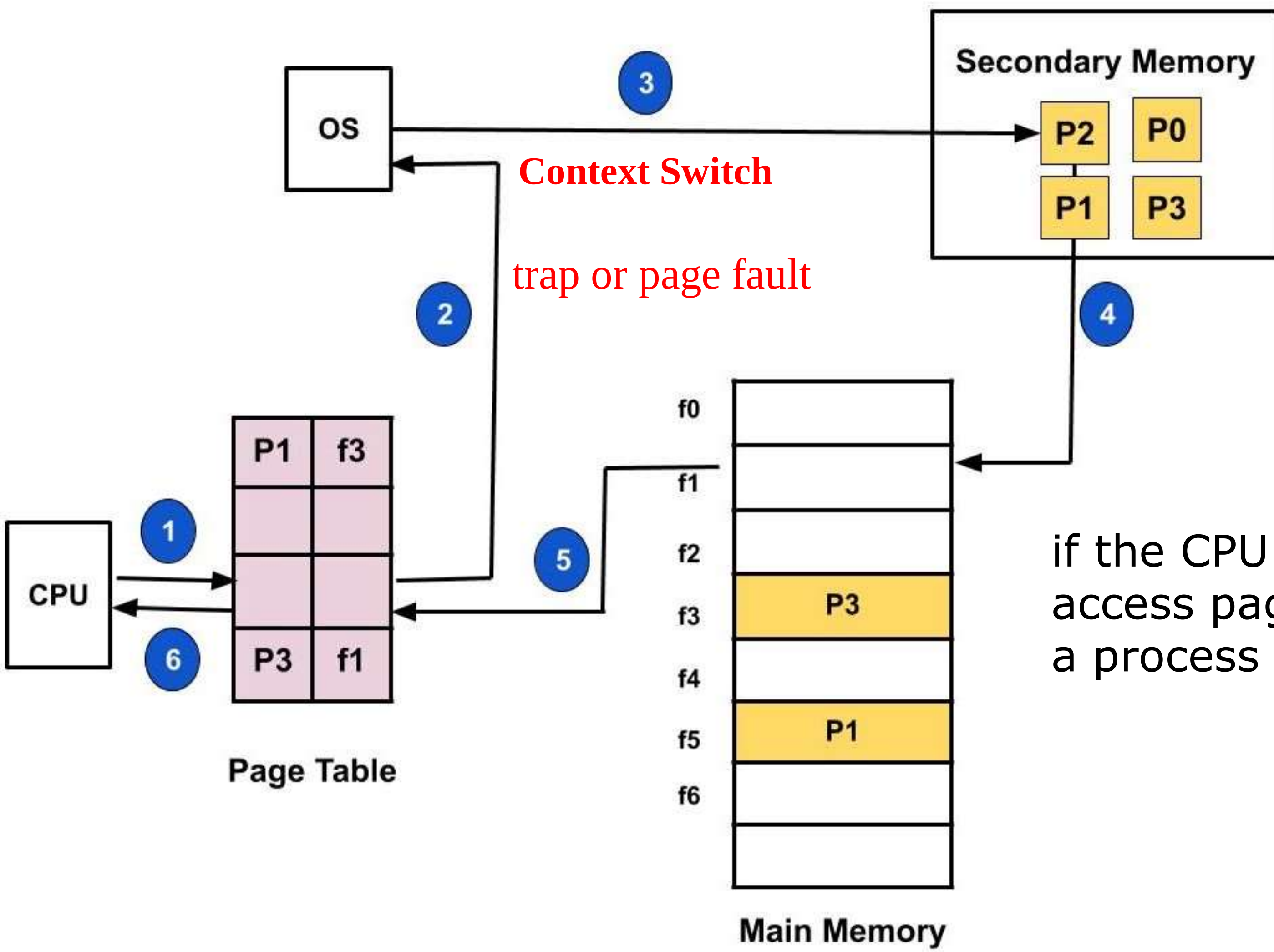
Demand Paging

- Similar to paging system with **swapping**
- 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



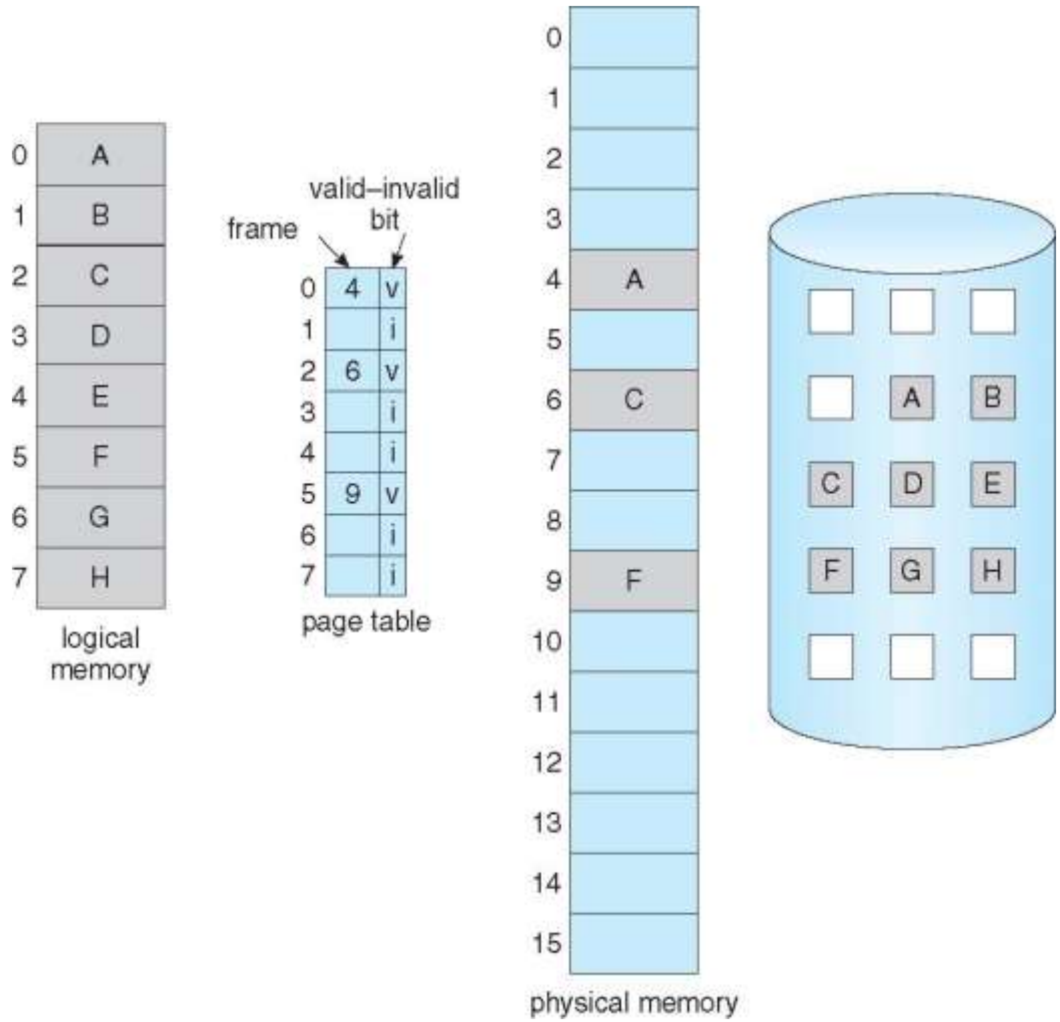
Demand Paging

- Suppose we have to execute a process P having four pages as P0, P1, P2, and P3. Currently, in the page table, we have page P1 and P3.



if the CPU wants to access page P2 of a process P

Page Table When Some Pages Are Not in Main Memory

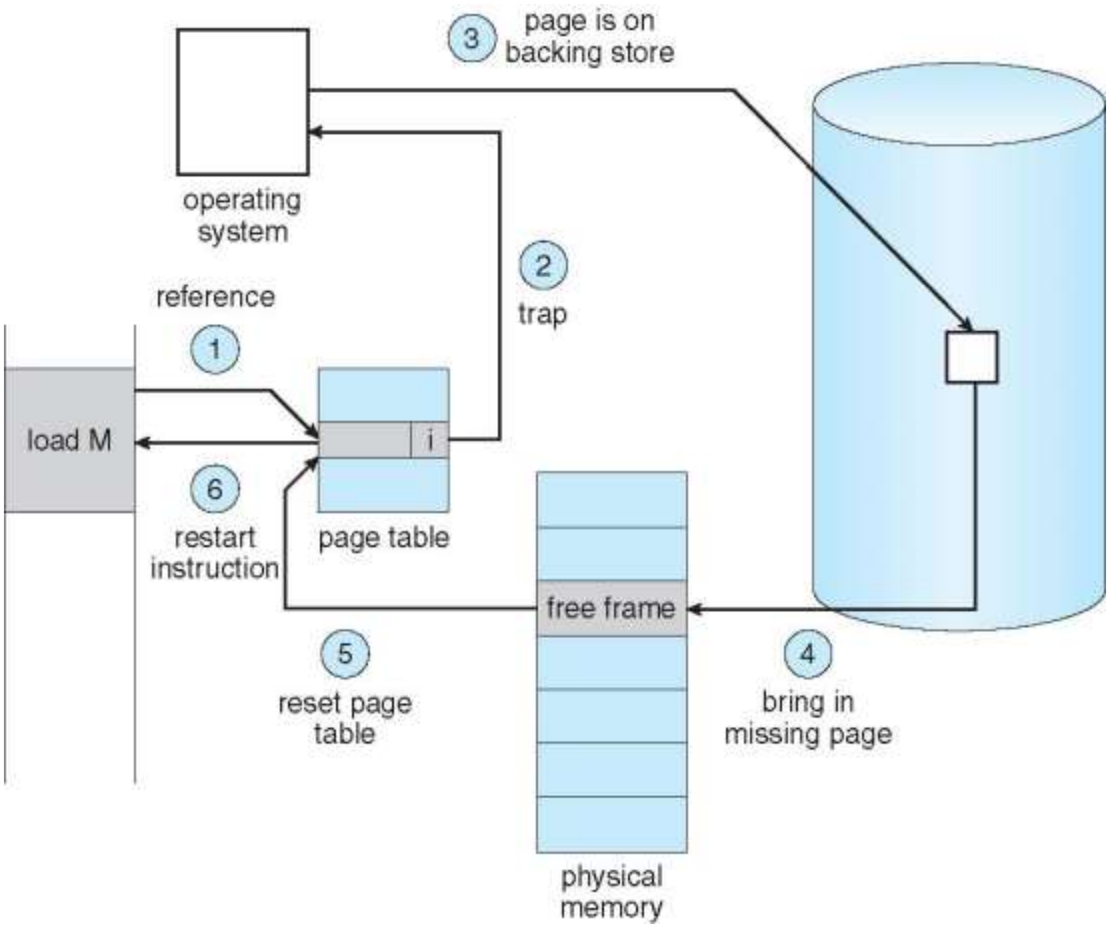


Page Fault

- **Page Fault** – A page fault happens when a running program accesses a memory page that is mapped into the virtual address space, **but not loaded in physical memory**

1. Operating system looks at page table to decide:
   - Invalid reference $\Rightarrow$ abort
   - Just not in memory
2. Find free frame
3. Swap page into frame via scheduled disk operation
4. Reset tables to indicate page now in memory
   Set validation bit = **v**
5. Restart the instruction that caused the page fault

Steps in Handling a Page Fault



Since actual physical memory is much **smaller** than virtual memory, **page faults happen**.

In case of page fault, Operating System might have to replace one of the existing pages with the newly needed page.

Different **page replacement algorithms** suggest different ways to decide which page to replace.

The target for all algorithms is to reduce the number of page faults.

Aspects of Demand Paging

- Advantages
  - It **increases** the degree of multiprogramming as many processes can be present in the main memory at the same time.
  - There is a **more efficient** use of memory as processes having size more than the size of the main memory can also be executed using this mechanism because we are not loading the whole page at a time.

- Disadvantages
  - Individual program face extra latency when they access a page for the first time
  - Programs running on low-cost, low-power embedded systems may not have a memory management
  - Memory management with page replacement algorithms becomes slightly more complex

What Happens if There is no Free Frame?

- **Used up** by process pages
- Also in demand from the kernel, I/O buffers, etc

- **Page replacement** – find some page in memory, but not really in use, page it out
  - Algorithm – terminate? swap out? replace the page?
  - Performance – want an algorithm which will result in minimum number of page faults

- Same page may be brought into memory several times

Basic Page Replacement

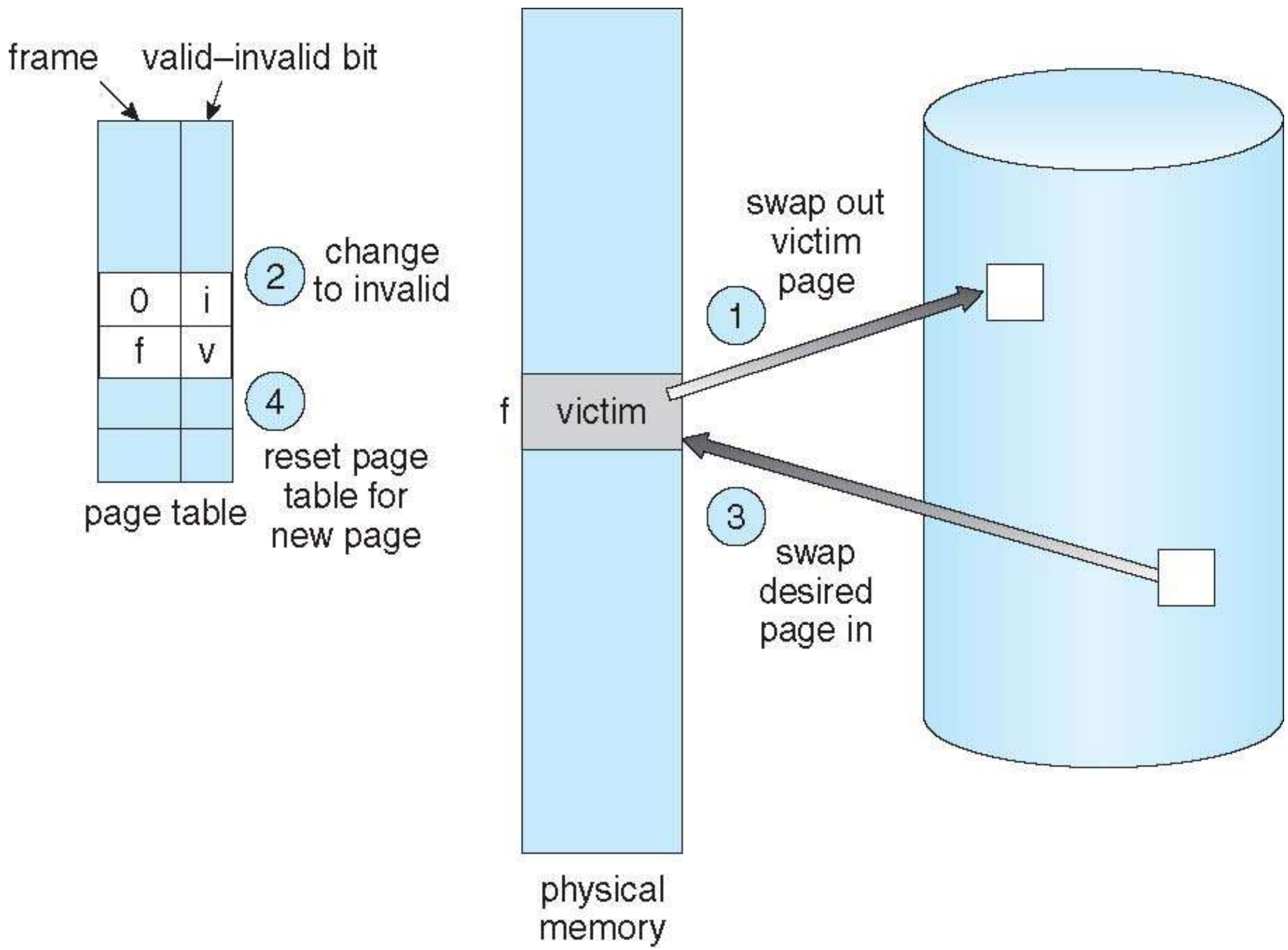
1. Find the location of the desired page on disk

2. Find a free frame:
   - If there is a free frame, use it
   - If there is no free frame, use a page replacement algorithm to select a **victim frame**

3. Bring the desired page into the (newly) free frame; update the page and frame tables

4. Continue the process by restarting the instruction that caused the trap

Page Replacement



- Want lowest **page-fault rate** on both first access and reaccess

First-In-First-Out (FIFO) Algorithm

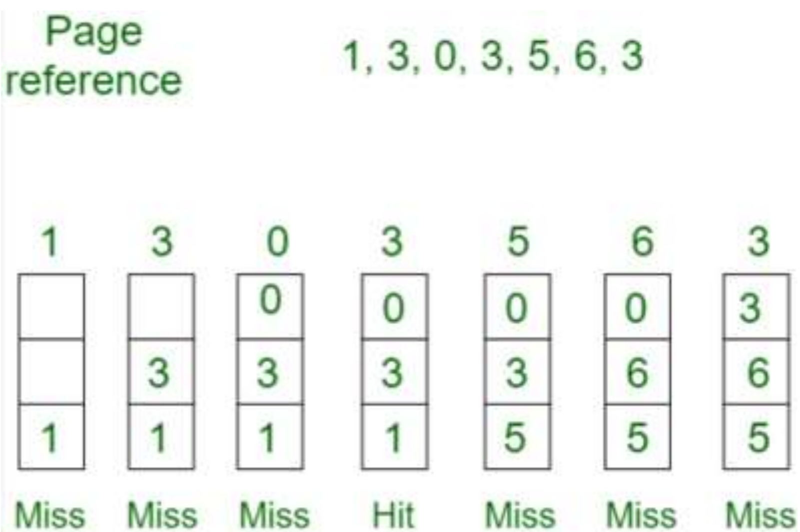
- This is the simplest page replacement algorithm.

- In this algorithm, the operating system keeps track of all pages in the memory in a queue, the **oldest page** is in the **front** of the queue.

- When a page needs to be replaced, page in the front of the queue is selected for removal.

First-In-First-Out (FIFO) Algorithm

- Consider page reference string 1, 3, 0, 3, 5, 6.
- 3 frames (3 pages can be in memory at a time per process)
- Find number of page faults.



Initially all slots are empty, so when 1, 3, 0 came they are allocated to the empty slots —> **3 Page Faults.**
when 3 comes, it is already in memory so —> **0 Page Faults.**
Then 5 comes, it is not available in memory, so it replaces the oldest page slot i.e 1. —>**1 Page Fault.**
6 comes, it is also not available in memory, so it replaces the oldest page slot i.e 3 —>**1 Page Fault.**
Finally, when 3 come it is not available, so it replaces 0 **1 page fault**

Least Recently Used (LRU) Algorithm

- Replace page that has not been used in the **most amount of time**

Page reference 7,0,1,2,0,3,0,4,2,3,0,3,2,3

No. of Page frame - 4

7	0	1	2	0	3	0	4	2	3	0	3	2	3
			2	2	2	2	2	2	2	2	2	2	2
		1	1	1	1	1	4	4	4	4	4	4	4
	0	0	0	0	0	0	0	0	0	0	0	0	0
7	7	7	7	7	3	3	3	3	3	3	3	3	3
Miss	Miss	Miss	Miss	Hit	Miss	Hit	Miss	Hit	Hit	Hit	Hit	Hit	Hit

Total Page Fault = 6

Here LRU has same number of page fault as optimal but it may differ according to question.

Problem 1

Calculate the size of memory if its address consists of 22 bits and the memory is 2-byte addressable.

We have-

- Number of locations possible with 22 bits = 2^{22} locations
- It is given that the size of one location = 2 bytes

Thus, Size of memory

= 2^{22} x 2 bytes

= 2^{23} bytes

= 8 MB

Problem 2

Calculate the number of bits required in the address for memory having size of 16 GB. Assume the memory is 4-byte addressable.

Solution-

Let ‘n’ number of bits are required. Then, Size of memory = 2^n x 4 bytes.

Since, the given memory has size of 16 GB, so we have-

2^n x 4 bytes = 16 GB

2^n x 4 = 16 G

$2^n \times 2^2 = 2^{34}$

$2^n = 2^{32}$

$\therefore$ n = 32 bits

Problem 3

Consider a system with byte-addressable memory, 32 bit logical addresses, 4 kilobyte page size and page table entries of 4 bytes each. The size of the page table in the system in megabytes is _____.

1. 2
2. 4
3. 8
4. 16

Problem 3: Solution

Given-

- Number of bits in logical address = 32 bits
- Page size = 4KB
- Page table entry size = 4 bytes

Process Size-

Number of bits in logical address = 32 bits

Thus,

Process size

= 2^{32} B

= 4 GB

Number of pages the process is divided

= Process size / Page size

= 4 GB / 4 KB

= 2^{20} pages

Problem 3: Solution

Process Size-

Number of bits in logical address = 32 bits

Thus,

Process size

$= 2^{32}$ B

= 4 GB

Number of pages the process is divided

= Process size / Page size

= 4 GB / 4 KB

$= 2^{20}$ pages

Page table size

= Number of entries in page table x Page table entry size

$= 2^{20}$ x 4 bytes

= 4 MB

Problem 4

Consider a machine with 64 MB physical memory and a 32 bit virtual address space. If the page size is 4 KB, what is the approximate size of the page table?

A. 16 MB
B. 8 MB
C. 2 MB
D. 24 MB

Problem 4: Solution

Given-

Size of main memory = 64 MB
Number of bits in virtual address space = 32 bits
Page size = 4 KB

We will consider that the memory is byte addressable.

Size of main memory = 64 MB = 2^{26} B

Thus, Number of bits in physical address = 26 bits

Problem 4: Solution

Number of frames in main memory

= Size of main memory / Frame size

= 64 MB / 4 KB

= 2^{26} B / 2^{12} B

= 2^{14}

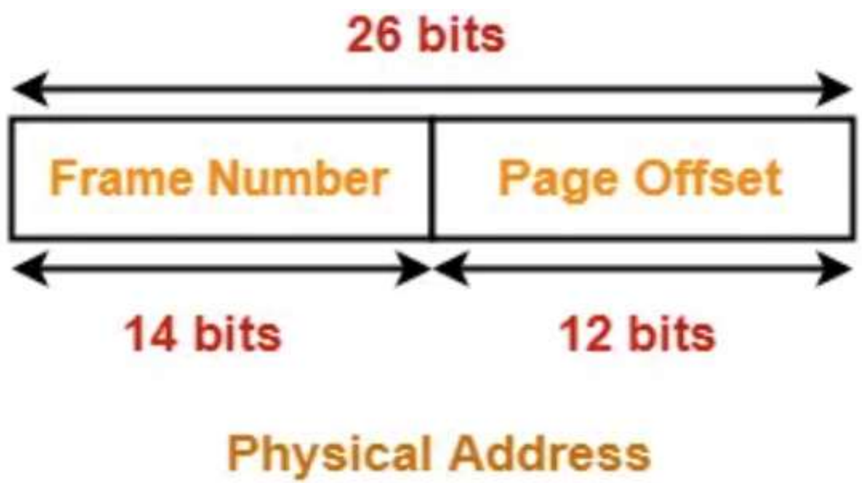
Thus, Number of bits in frame number = 14 bits

Problem 4: Solution

We have, Page size = 4 KB = 2^{12} B

Thus, Number of bits in page offset = 12 bits

So, Physical address is



Problem 4: Solution

Number of bits in virtual address space = 32 bits

Thus, Process size = 2^{32} B = 4 GB.

Number of pages the process is divided = Process size / Page size

= 4 GB / 4 KB

= 2^{20} pages

Thus, Number of entries in page table = 2^{20} entries

Problem 4: Solution

Page table size

= Number of entries in page table $\times$ Page table entry size

= Number of entries in page table $\times$ Number of bits in frame number

$= 2^{20} \times 14$ bits

$= 2^{20} \times 16$ bits (Approximating 14 bits $\approx$ 16 bits)

$= 2^{20} \times 2$ bytes

= 2 MB

Thus, Option (C) is correct.

Problem 5

In a virtual memory system, size of virtual address is 32-bit, size of physical address is 30-bit, page size is 4 Kbyte and size of each page table entry is 32-bit. The main memory is byte addressable. Which one of the following is the maximum number of bits that can be used for storing protection and other information in each page table entry?

A. 2
B. 10
C. 12
D. 14

Problem 5: Solution

Given:

Number of bits in virtual address = 32 bits

Number of bits in physical address = 30 bits

Page size = 4 KB

Page table entry size = 32 bits

Number of frames in main memory

= Size of main memory / Frame size

= 2^{30} B / 2^{12} B

= 2^{18}

Thus, Number of bits in frame number = 18 bits

Problem 5: Solution

Maximum number of bits that can be used for storing protection and other information

= Page table entry size – Number of bits in frame number

= 32 bits – 18 bits

= 14 bits

Thus, Option (D) is correct.

Problem 6

Consider a single level paging scheme. The virtual address space is 4 MB and page size is 4 KB. What is the maximum page table entry size possible such that the entire page table fits well in one page?

Problem 6: Solution

Number of pages the process is divided
= Process size / Page size
= 4 MB / 4 KB
= 2^{10} pages

Let page table entry size = B bytes

Page table size

= Number of entries in the page table $\times$ Page table entry size

= Number of pages the process is divided $\times$ Page table entry size

= $2^{10} \times$ B bytes

Problem 6: Solution

According to the above condition, we must have-

$2^{10} \times$ B bytes <= 4 KB

$2^{10} \times B <= 2^{12}$

$B <= 4$

Thus, maximum page table entry size possible = 4 bytes.

Problem 7

A paging scheme uses a Translation Lookaside buffer (TLB). A TLB access takes 10 ns and a main memory access takes 50 ns. What is the effective access time (in ns) if the TLB hit ratio is 90% and there is no page fault?

A. 54
B. 60
C. 65
D. 75

Problem 7: Solution

Given:

TLB access time = 10 ns
Main memory access time = 50 ns
TLB Hit ratio = 90% = 0.9

TLB Miss ratio

= 1 – TLB Hit ratio

= 1 – 0.9

= 0.1

Problem 7: Solution

Effective Access Time

$= 0.9 \times \{ 10 \text{ ns} + 50 \text{ ns} \} + 0.1 \times \{ 10 \text{ ns} + 2 \text{ x } 50 \text{ ns} \}$

$= 0.9 \times 60 \text{ ns} + 0.1 \times 110 \text{ ns}$

$= 54 \text{ ns} + 11 \text{ ns}$

$= 65 \text{ ns}$

Thus, Option (C) is correct.

Problem 8

Consider a system using paging scheme where-

Logical Address Space = 4 GB
Physical Address Space = 16 TB
Page size = 4 KB

How many levels of page table will be required?

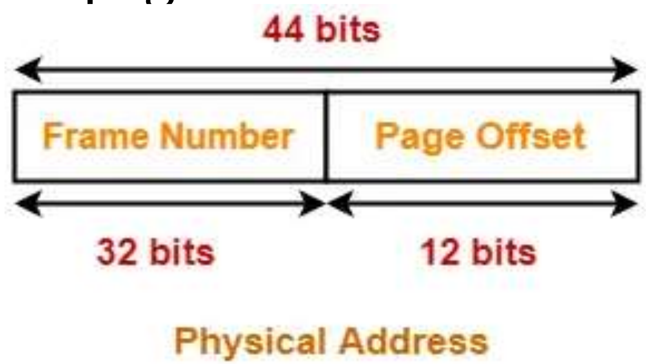
Problem 8: Solution

Logical Address Space = 4 GB
Physical Address Space = 16 TB
Page size = 4 KB

Size of main memory = Physical Address Space = 16 TB = 2^{44} B

Number of frames = Size of main memory / Frame size
= 16 TB / 4 KB = 2^{32} frames

Page size = 4 KB = 2^{12} B
Thus, Number of bits in page offset = 12 bits



Problem 8: Solution

Logical Address Space = 4 GB
Physical Address Space = 16 TB
Page size = 4 KB

Number of pages = Process size / Page size = 4GB / 4KB= 2^{20} pages

Inner Page Table Size-

Inner page table keeps track of the frames storing the pages of process.

Inner Page table size
= Number of entries in inner page table × Page table entry size
= Number of pages × Number of bits in frame number
= $2^{20} \times 32$ bits = $2^{20} \times 4$ bytes
= 4 MB

Problem 8: Solution

Inner Page table size = 4 MB

Now, we can observe-

- The size of inner page table is greater than the frame size (4 KB).
- Thus, inner page table can not be stored in a single frame.
- So, inner page table has to be divided into pages.

Number of pages the inner page table is divided
= Inner page table size / Page size
= 4 MB / 4 KB
= 2^{10} pages

Now, these $\mathbf{2^{10}}$ pages of inner page table are stored in different frames of the main memory.

Problem 8: Solution

Now, these **2^{10}** pages of inner page table are stored in different frames of the main memory.

Number of page table entries in one page of inner page table
= Page size / Page table entry size
= Page size / Number of bits in frame number
= 4 KB / 32 bits
= 4 KB / 4 B
= 2^{10}

One page of inner page table contains 2^{10} entries.
Thus, number of bits required to search a particular entry in one page of inner page table = 10 bits

Problem 8: Solution

Outer Page Table Size-

Outer page table is required to keep track of the frames storing the pages of inner page table.

Outer Page table size
= Number of entries in outer page table × Page table entry size

= Number of pages the inner page table is divided × Number of bits in frame number

= $2^{10} \times 32$ bits

= $2^{10} \times 4$ bytes = 4 KB

Problem 8: Solution

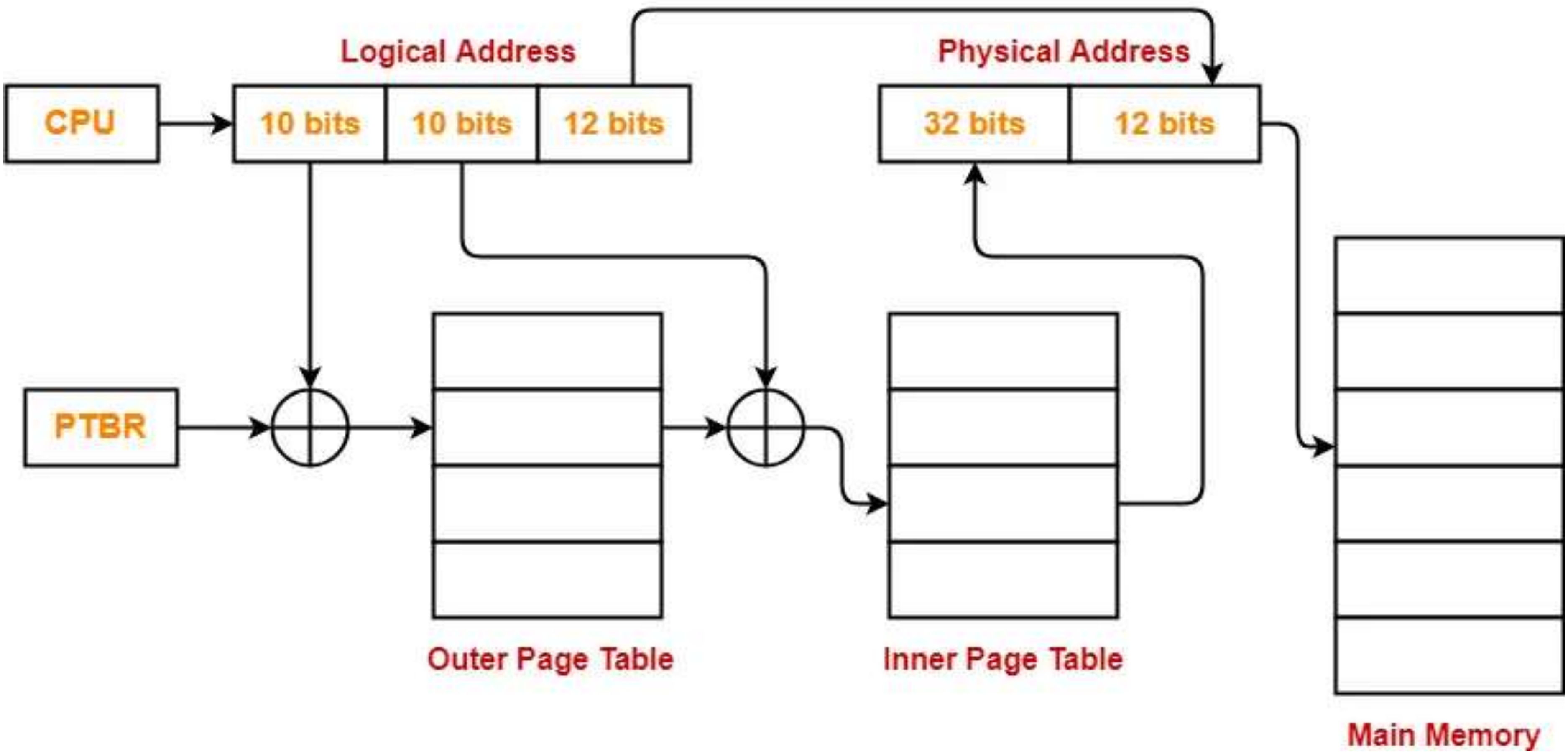
Outer Page table size = 4 KB

Now, we can observe-

- The size of outer page table is same as frame size (4 KB).
- Thus, outer page table can be stored in a single frame.
- So, for given system, we will have two levels of page table.
- Page Table Base Register (PTBR) will store the base address of the outer page table.

Problem 8: Solution

The paging system will look like as shown below-



Problem 9

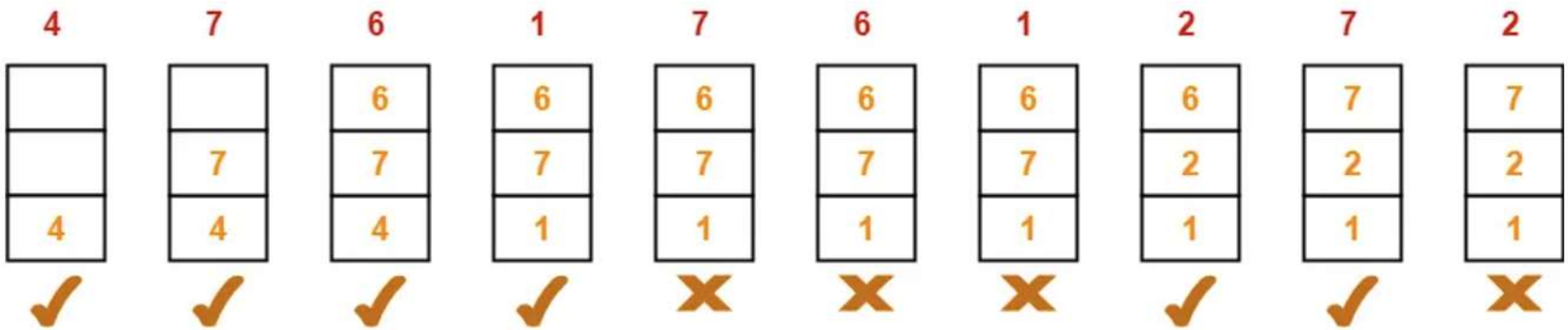
A system uses 3 page frames for storing process pages in main memory. It uses the Least Recently Used (LRU) page replacement policy. Assume that all the page frames are initially empty. What is the total number of page faults that will occur while processing the page reference string given below-

4 , 7, 6, 1, 7, 6, 1, 2, 7, 2

Also calculate the hit ratio and miss ratio.

Problem 9: Solution

Total number of references = 10



From here,

Total number of page faults occurred = 6

In the similar manner as above-

Hit ratio = 0.4 or 40%
Miss ratio = 0.6 or 60%

End of Chapter 8