Main Memory

Dr. SUBHASIS DASH

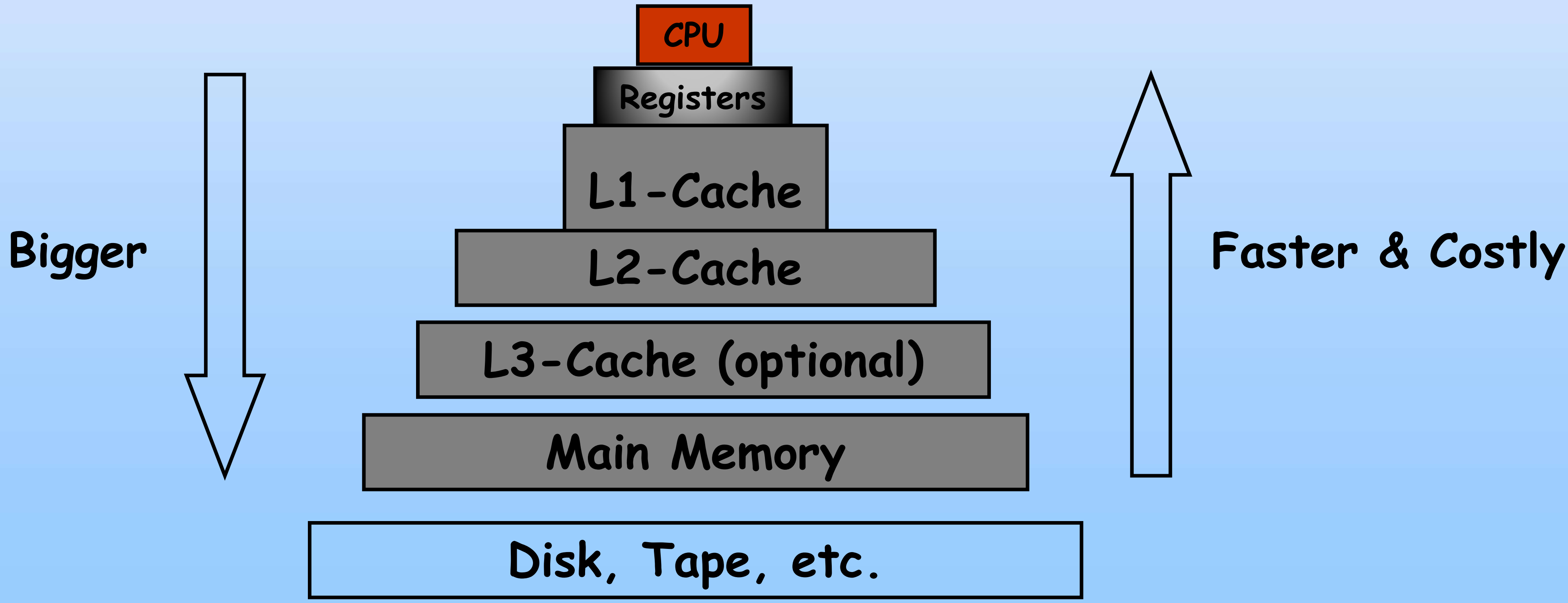
SCHOLE OF COMPUTER ENGINEERING.

KIIT UNIVERSITY

BHUBANESWAR

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
- Register access in one CPU clock (or less)
- Main memory can take many cycles
- **Cache** sits between main memory and CPU registers
- Protection of memory required to ensure correct operation



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
  - **Load time**: Must generate **relocatable code** if memory location is not known at compile time
  - **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)

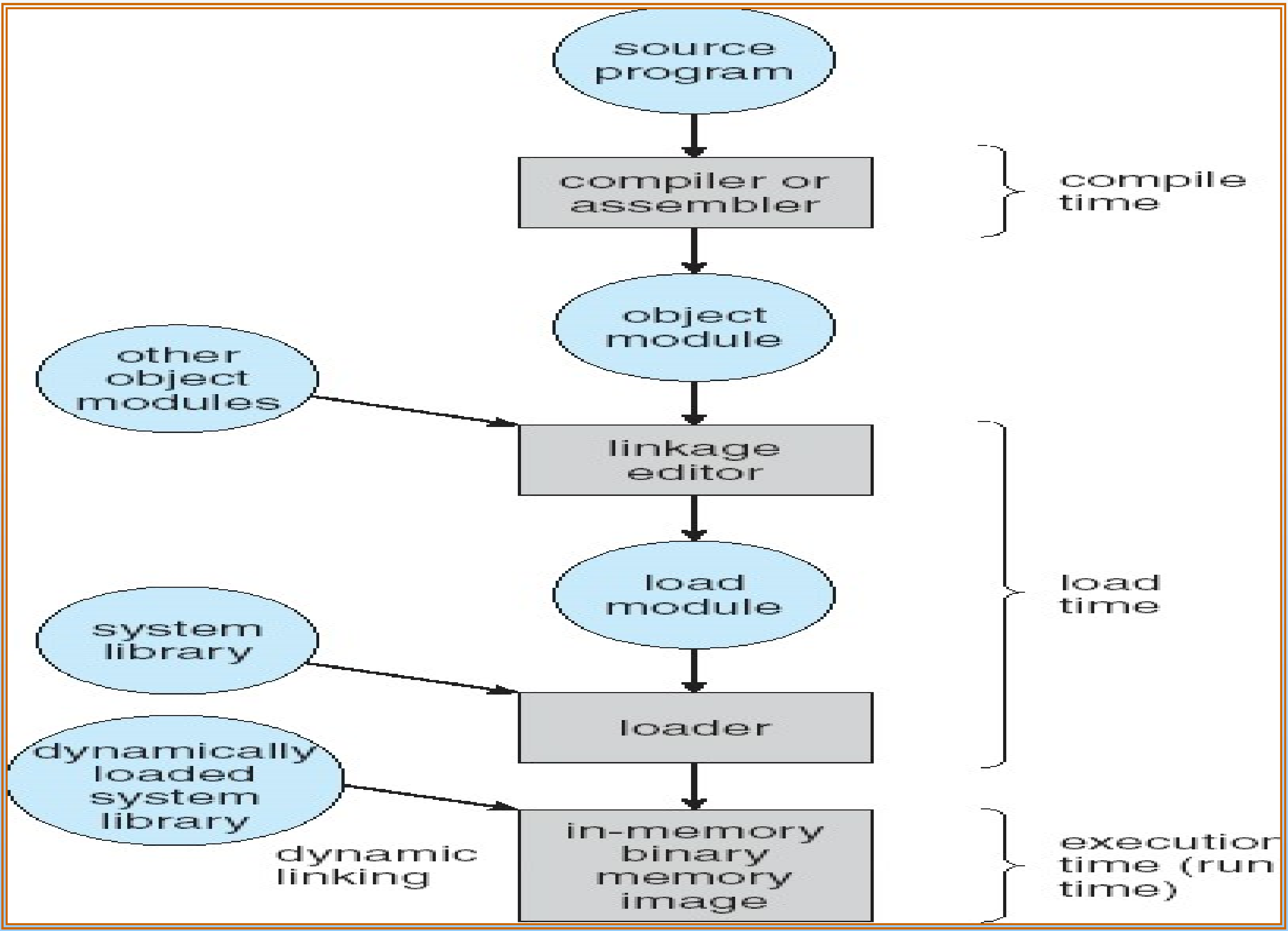
Dynamic Loading

- Routine is not loaded until it is called
- Routines are kept in a relocatable load format.
- Better memory-space utilization; unused routine is never loaded
- Useful when large amounts of code are needed to handle infrequently occurring cases
- No special support from the operating system is required implemented through program design

Dynamic Linking

- System always needs a copy of language library included in executable image →Dynamically linked libraries.
- Static linking (System language libraries in binary program image).
- Concept of dynamic linking == dynamic loading.(Rather held up loading until execution we can held up the 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 needed to check if routine is in processes' memory address
- Dynamic linking is particularly useful for libraries
- System also known as **shared libraries**

Multistep Processing of a User Program



overlays

- Program & data must be in physical memory to start execution.
- If (Program +Data) size <= size of physical memory

```
    then
            start execution
    else
            use a technique called overlays.
```

- Example:-

Given data:-

Process	Memory Size	
Pass-1	70kb	$\left.\begin{array}{l} \\ \\ \\ \end{array}\right\}$ Total memory required 200kb.
Pass-2	80kb	
Symbol table	20kb	
Common Routine	30kb	

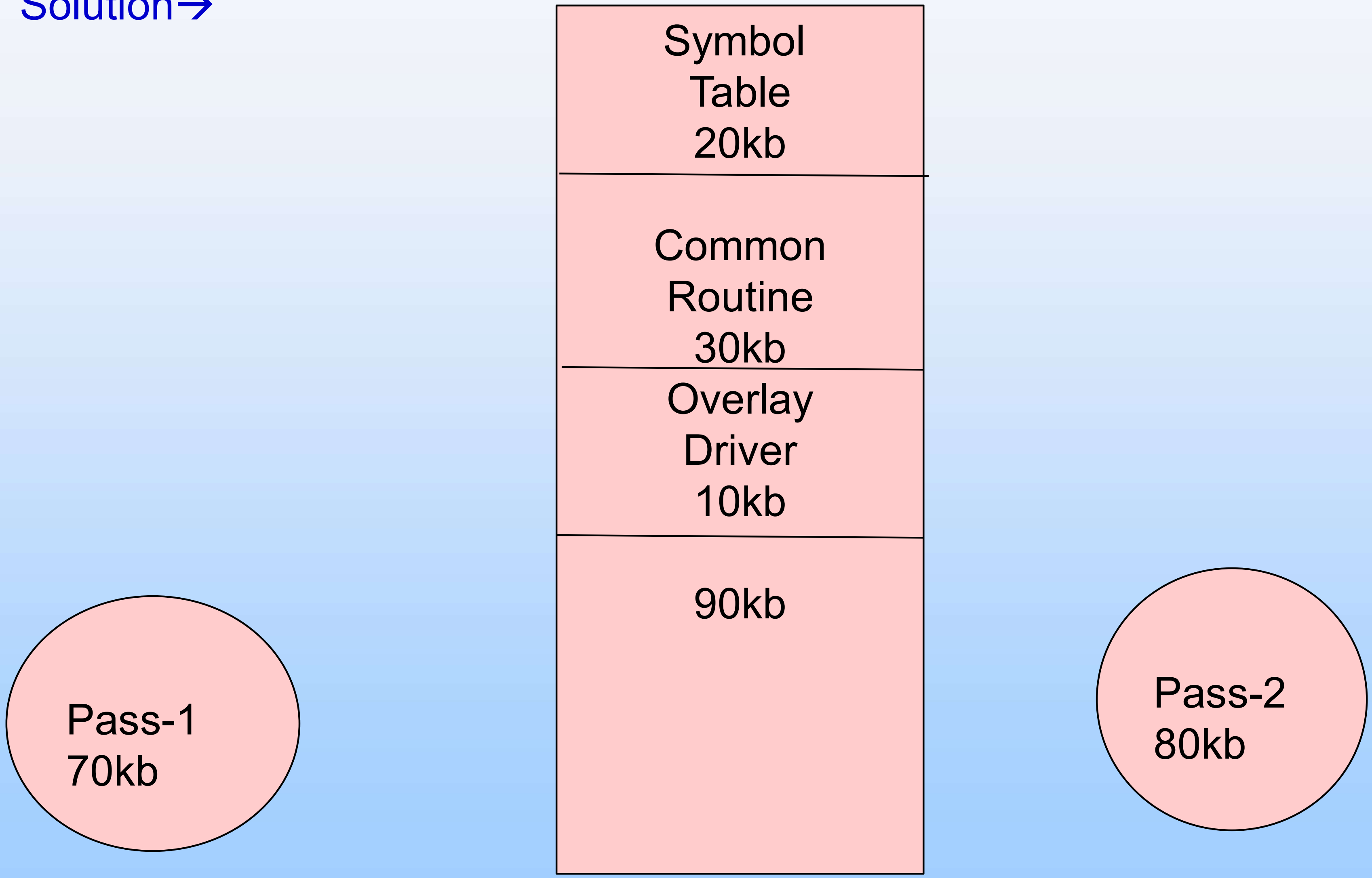
Main memory size available is 150kb.

Hence 200kb >150kb

Overlays

Continued….

Solution→

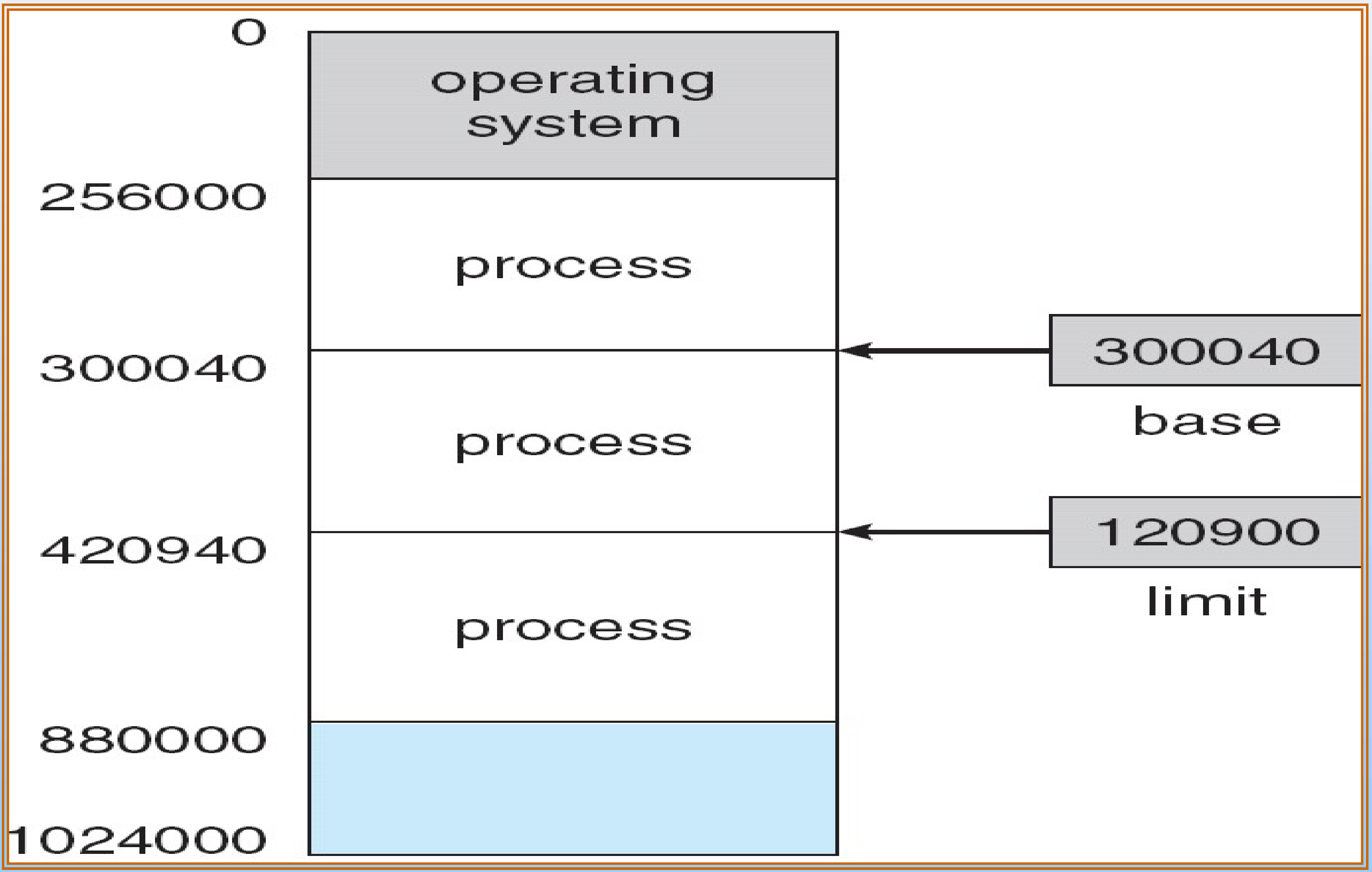


Logical vs. Physical Address Space

- User's view→ User program see primary memory as continuous space
- Actual view→ User program may be scattered according to the instruction address location.
- 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

Base and Limit Registers

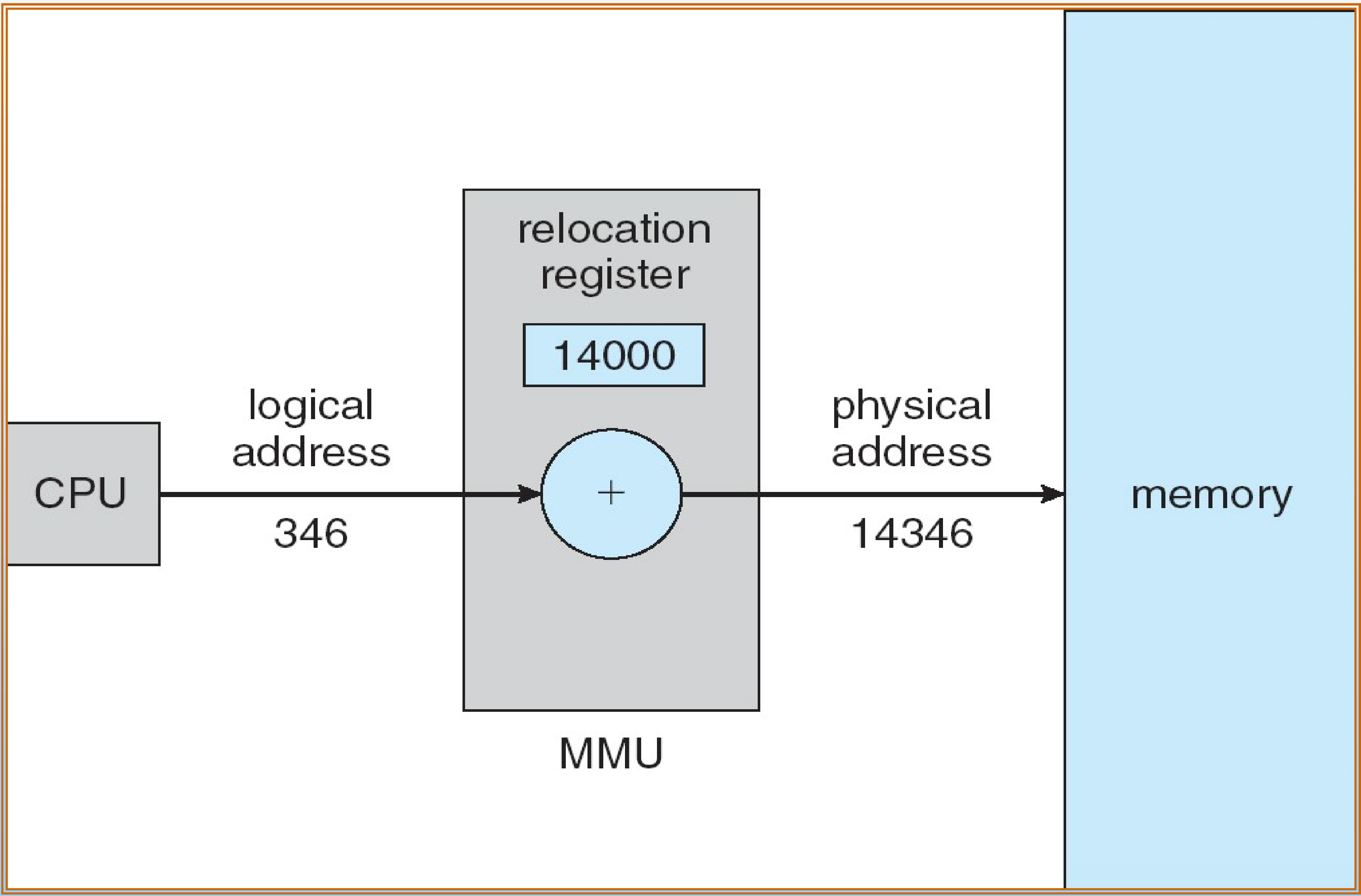
- A pair of **base** and **limit** registers define the logical address space



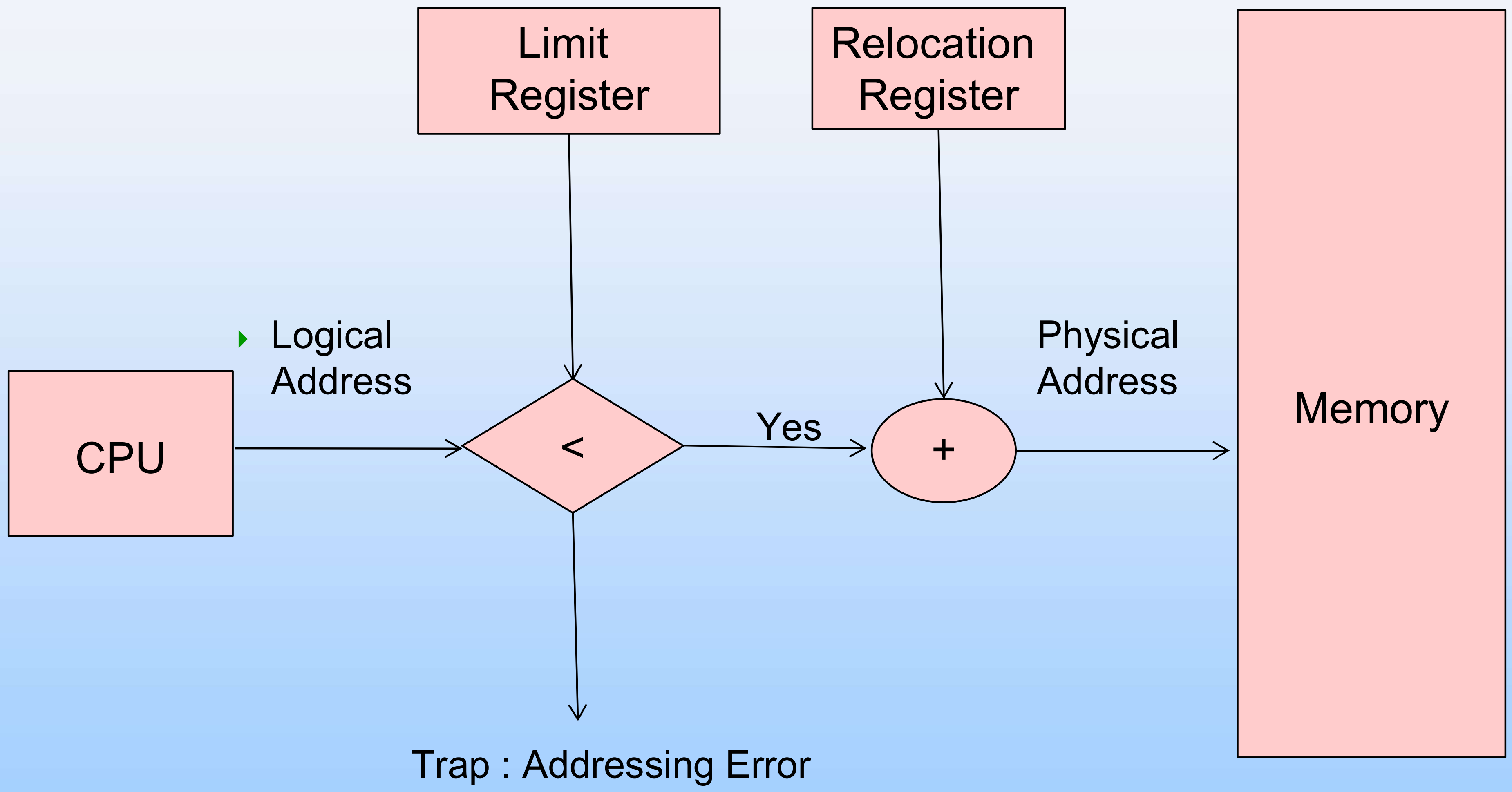
Memory-Management Unit (MMU)

- Hardware device that maps logical to physical address
- In MMU scheme, the value in the basrelocation register is added to every address generated by a user process at the time it is sent to memory
- The user program deals with *logical* addresses; it never sees the *real* physical addresses

Dynamic relocation using a relocation register



HW address protection with base and limit registers

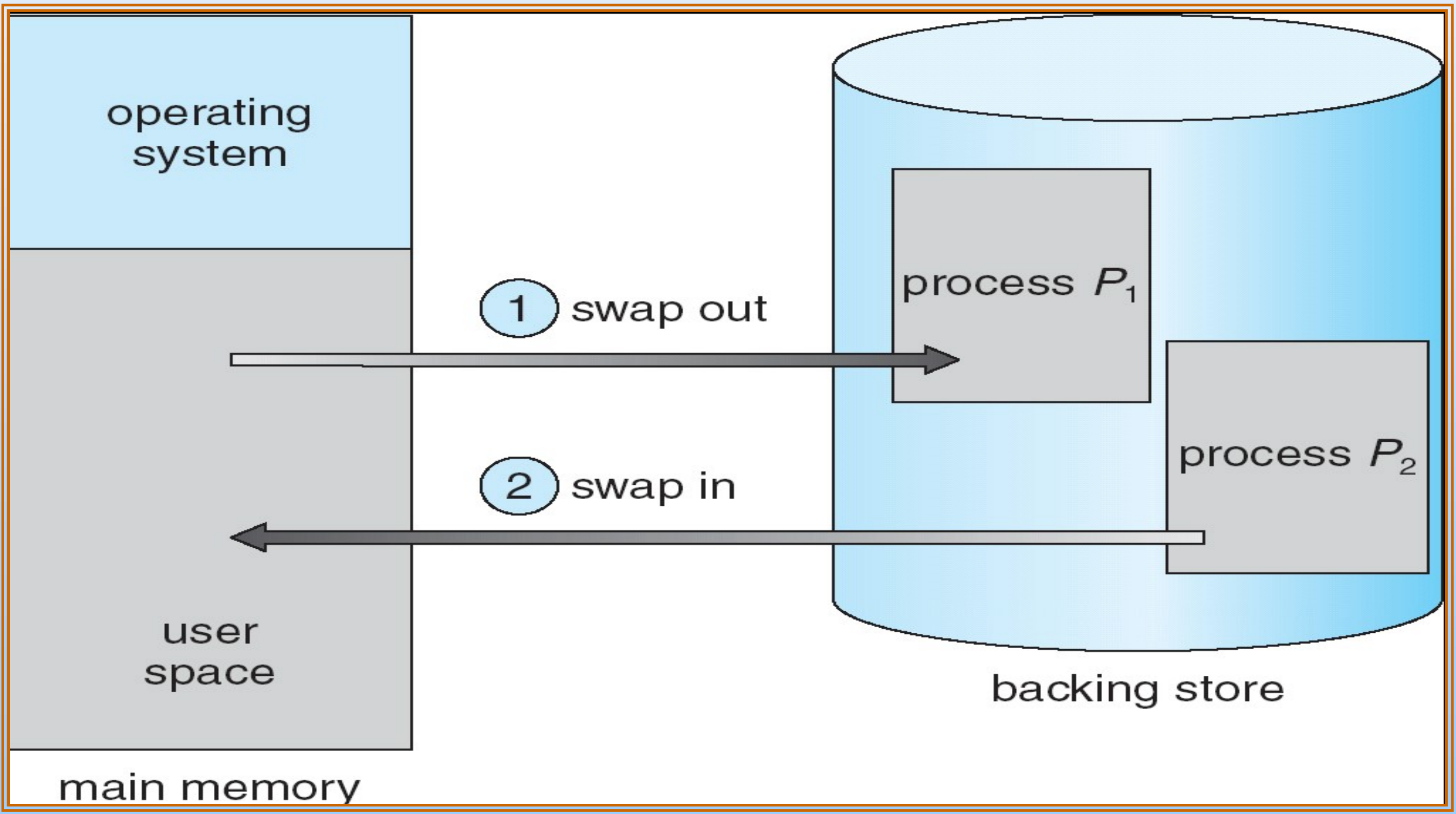


Swapping

- 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
- Modified versions of swapping are found on many systems (i.e., UNIX, Linux, and Windows)
- System maintains a **ready queue** of ready-to-run processes which have memory images on disk

Schematic View of Swapping

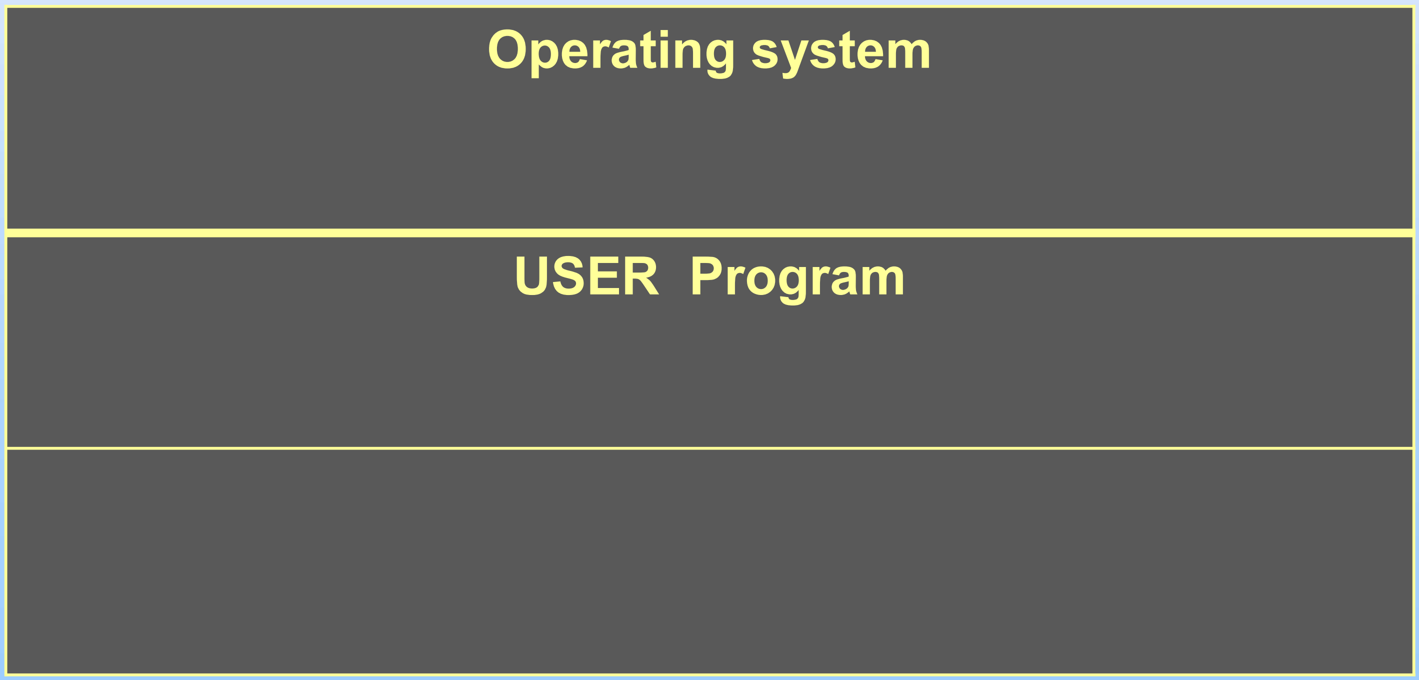
- If a system wants to swaps a process ,it must take care of some important points like:-
  - Is the process waits for its I/O operation?
  - System wants to create free space in memory



Contiguous Allocation

- Main memory usually into two partitions:
  - Resident operating system, usually held in low memory with interrupt vector
  - User processes then held in high memory.

Memory Partition



Contiguous Allocation (Cont.)

1. SINGLE PARTITION ALLOCATION→

- Relocation registers used to protect user processes from each other, and 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*

Contiguous Allocation (Cont.)

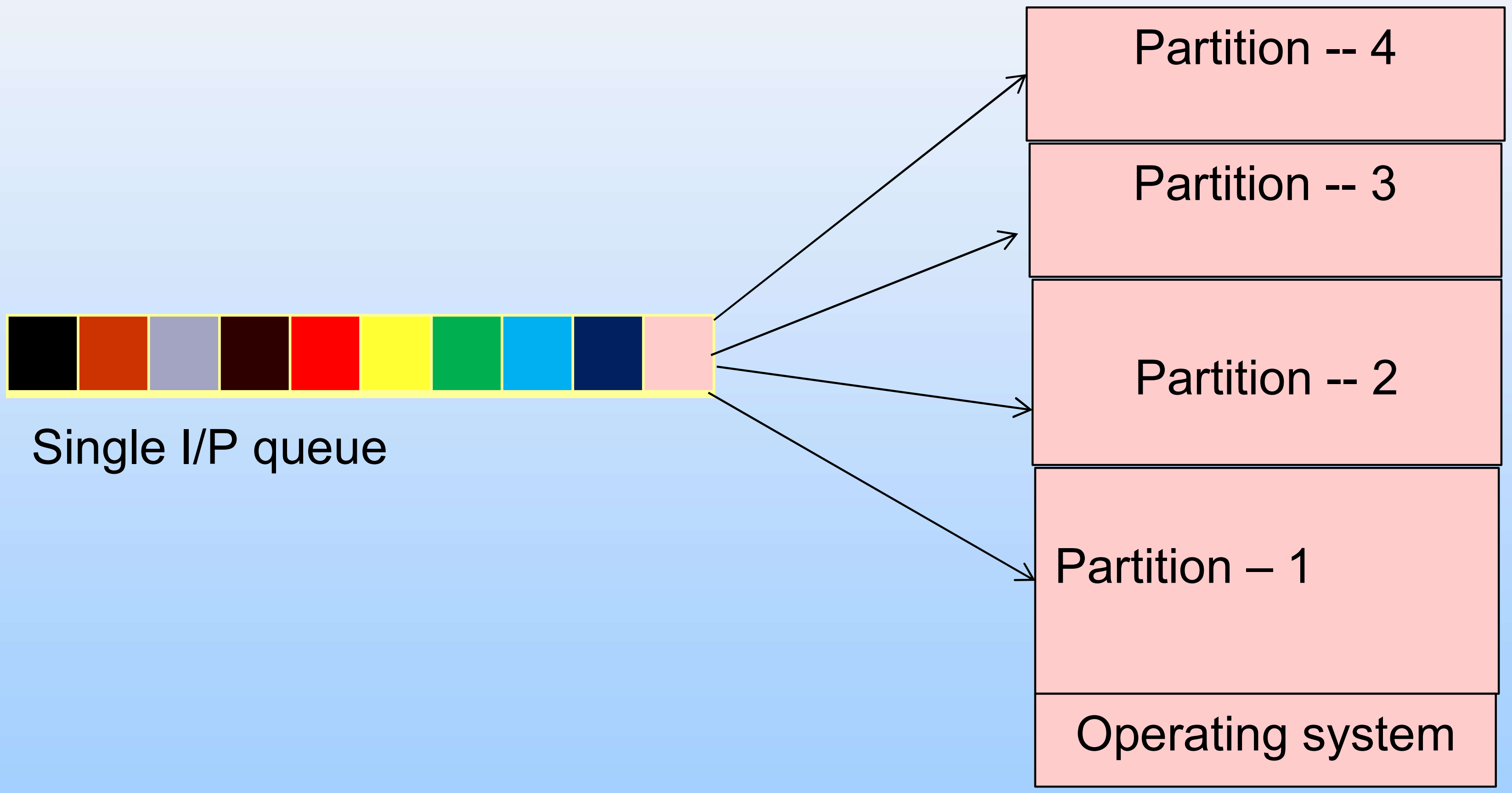
2. MULTIPLE-PARTITION ALLOCATION→

A. Multiprogramming With Fixed Partition (MFT) :-

- Objective:- To have more then one process in memory at once.
- Best way→ Divide memory into N Possible unequal partitions.
- But no single partition can accommodate more than one process
- When process arrives , it can put into the input queue for the smallest partition but large enough to accommodate the process.
- Though the technique moving towards best size fit ,still there are some free spaces in partition which can't be used further.
- Unused memory space leads to INTERNAL FRAGMENTATION.
- To avoid undesirable loss of memory space in the partition, search the whole input queue when a partition becomes free,& pick up the largest object that fits.

Contiguous Allocation (Cont.)

- Multiprogramming With Fixed Partition (MFT) :-



Contiguous Allocation (Cont.)

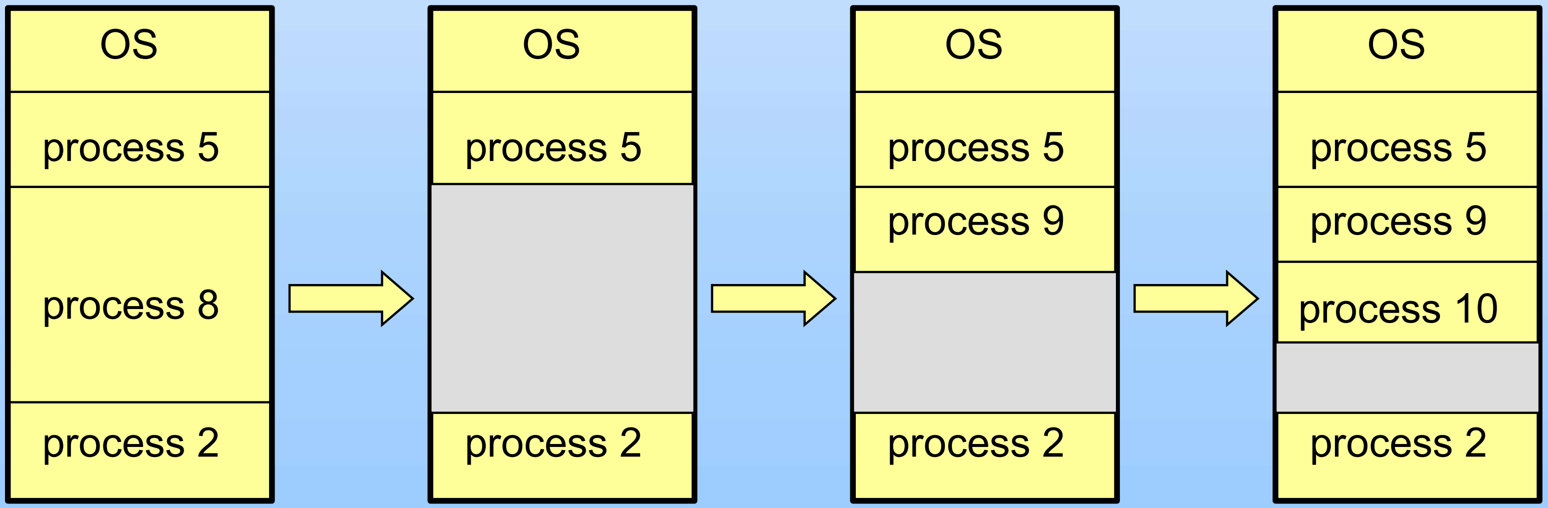
B. Multiprogramming With variable Partition (MVT) :-

- Number ,size of the process & memory partition vary dynamically .
- 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
- Operating system maintains information about:
  a) allocated partitions   b) free partitions (hole)
- Process growth due to data segment can be adjusted with the dynamic free space . If it is not sufficient then process have to swap out otherwise kill the process.

Contiguous Allocation (Cont.)

B. Multiprogramming With variable Partition (MVT) :-

- ✓ We can conclude that, When process appears → Kept into the input queue according to the scheduling algorithm → allocate the memory when it space is available → When memory is not satisfy to the next process the operating system have to wait until the required memory space is available otherwise it simply skips that process and switch for the next smaller process.
- ✓ The unused spaces between different process leads to EXTERNAL FRAGMENTATION.

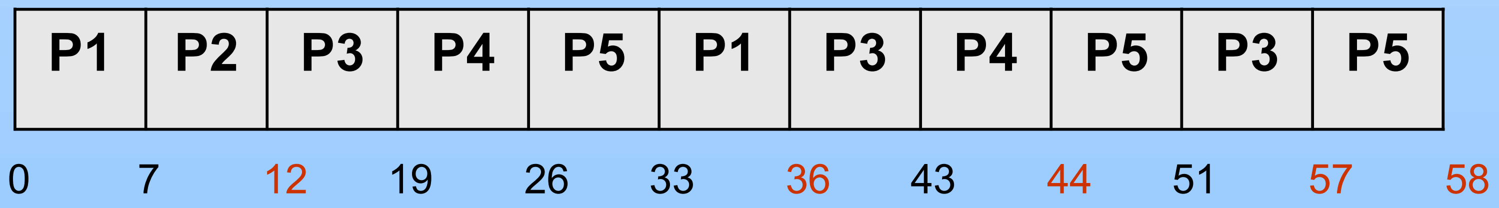


Contiguous Allocation (Cont.)

- Example:-
  - Memory available = 2560kb
  - OS takes a memory space of = 400kb
  - Memory space for user process = 2560 – 400 = 2160kb
  - Use Round robin scheduling algorithm with quantum time = 07 unit of time.
  - Given data:-

Process	Memory	Time
P1	600kb	10
P2	1000kb	05
P3	300kb	20
P4	700kb	08
P5	500kb	15

Solution:-



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
- **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

A multiprogramming system is available with a variable partition scheme. The system uses a set of free memory list to track different available memory spaces. The current list contains the entries of 270KB, 100KB, 450KB, 350KB, 175KB, and 600KB as free. The system receives requests for multiple processes like 585KB, 270KB, 346KB, 140KB, 435KB, and 90KB in order. Draw and explain the final allocation of the process by using the following dynamic memory allocation strategies like the best fit, first fit, and worst fit.

Fragmentation

- **External Fragmentation** – total memory space exists to satisfy a request, but it is not contiguous
- **Internal Fragmentation** – allocated memory may be slightly larger than requested memory; this size difference is memory internal to a partition, but not being used
- Reduce external fragmentation by **compaction**
  - Shuffle memory contents to place all free memory together in one large block
  - Compaction is possible *only* if relocation is dynamic, and is done at execution time
  - I/O problem
    - Latch job in memory while it is involved in I/O
    - Do I/O only into OS buffers

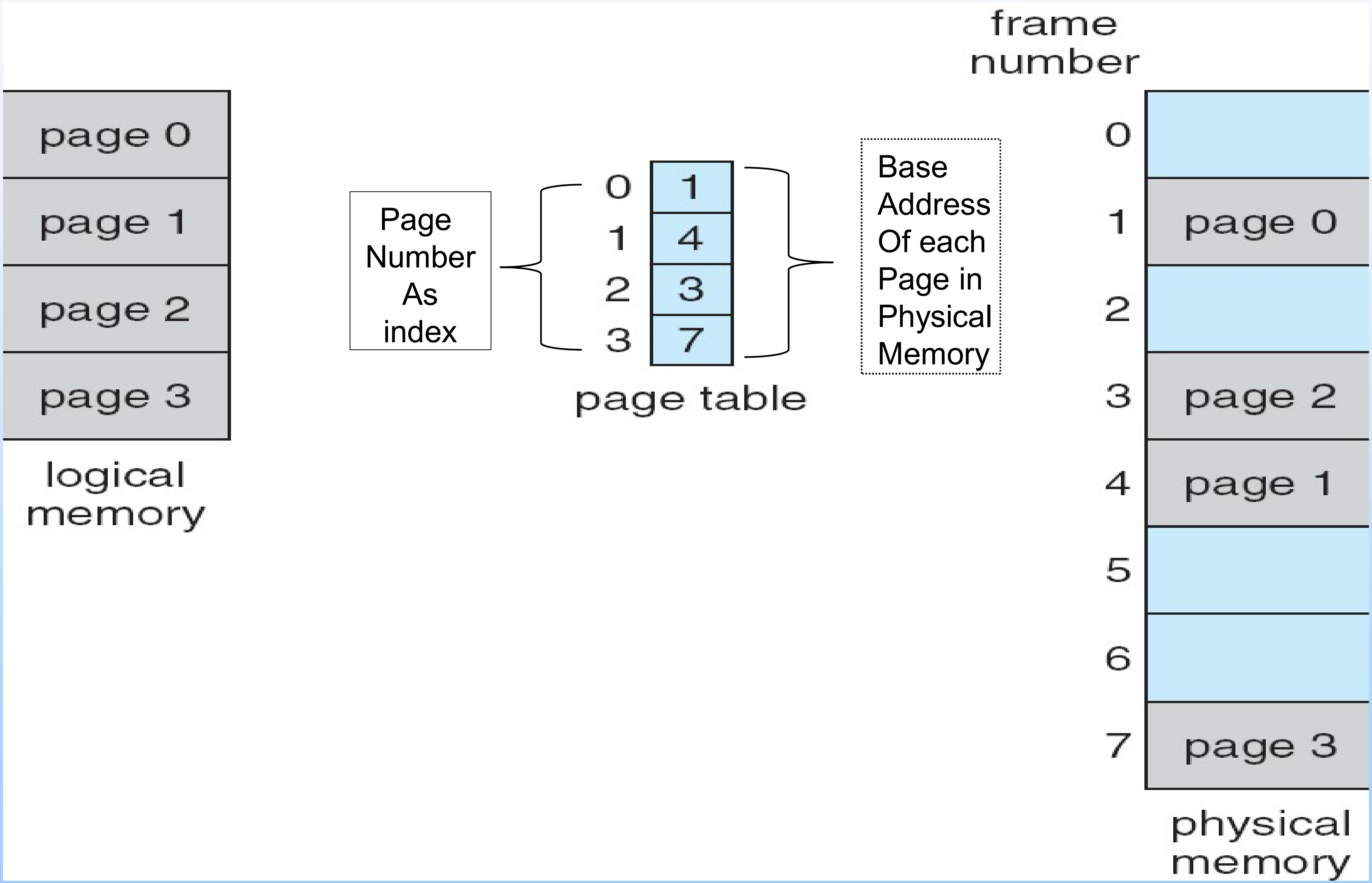
Paging

- External fragmentation problem can be treated by PAGING.
- Logical address space of a process can be noncontiguous; process is allocated physical memory whenever space is available
- Address mapping in paging scheme→
  - Divide physical memory into fixed-sized blocks called **frames** (size is power of 2, between 512 bytes and 8,192 bytes)
  - Divide logical memory into blocks of same size called **pages**
  - Keep track of all free frames
  - 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
  - Internal fragmentation

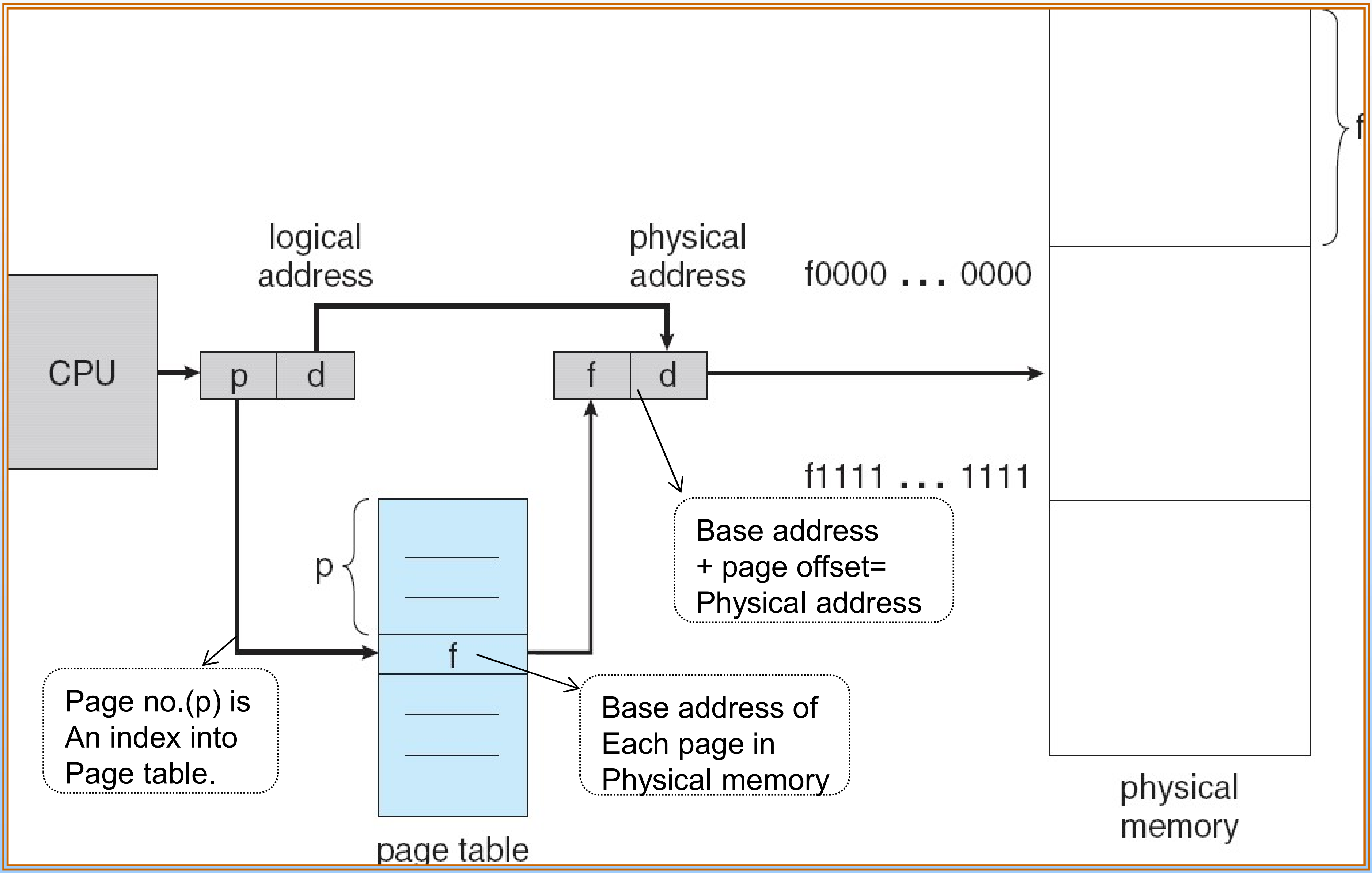
Address Translation Scheme

- Address generated by CPU (Logical Address) 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
  - Base address + page offset = physical address

Paging Model of Logical and Physical Memory



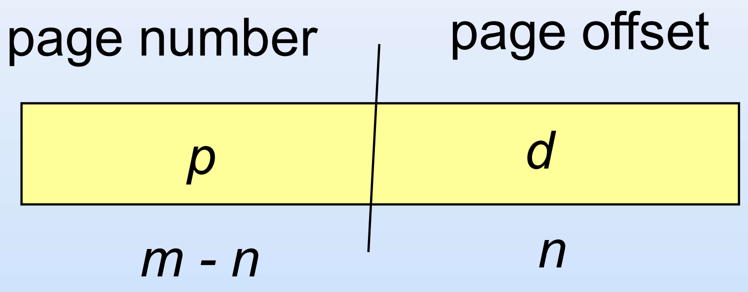
Paging Hardware



logical address
physical address
CPU
p
d
f
d
f0000 ... 0000
f1111 ... 1111
f
p
f
page table
physical memory
Base address + page offset= Physical address
Page no.(p) is An index into Page table.
Base address of Each page in Physical memory

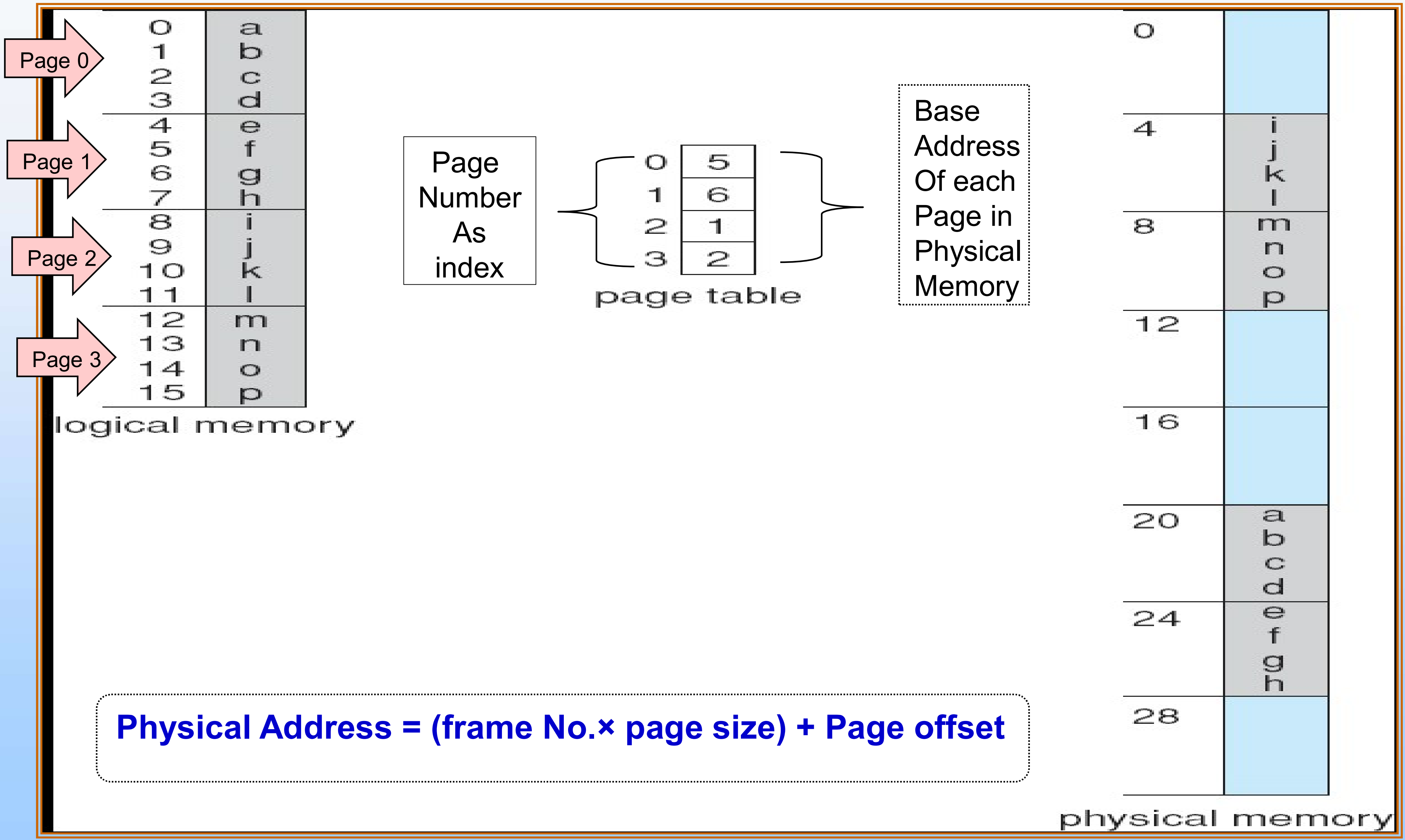
LA $\longrightarrow$ Address Mapping $\longrightarrow$ PA

- If size of logical address space 2^m and page size 2^n then
  - Low order n bits of logical address represents the page offset.
  - High order remaining bits (m-n) represents the page number.



- Page number P is an index into page table.
- page offset d is the displacement within pages.

Paging Example



Physical Address = (frame No.× page size) + Page offset

32-byte memory and 4-byte pages

LA $\longrightarrow$ Address Mapping $\longrightarrow$ PA

- **Physical Address = (frame No.× page size) + Page offset**
- Example:-
  - page size is 04 bytes.
  - Physical memory size is of 32bytes.
  - Hence total number of pages 32 bytes / 4 bytes = 08.
  - If logical address is 0 then, what is its corresponding physical address?
    - Page offset (d) = displacement within pages = 0 – 0 = 0
    - Hence Page number is = 0, which is in frame 5 given as in page table.
    - So physical address = (5 x 4) + 0 = 20
  - If logical address is 03 then, what is its corresponding physical address?
    - Page offset (d) = displacement within pages = 03 – 0 = 3
    - Hence Page number is = 0, which is in frame 5 given as in page table.
    - So physical address = (5 x 4) + 3 = 23

Paging Example

Continue…

- If logical address is 04 then, what is its corresponding physical address?
  - Page offset (d) = displacement within pages = 04 – 4 = 0
  - Hence Page number is = 1, which is in frame 6 given as in page table.
  - So physical address = (6 x 4) + 0 = 24
- If logical address is 10 then, what is its corresponding physical address?
  - Page offset (d) = displacement within pages = 10 – 8 = 2
  - Hence Page number is = 2, which is in frame 1 given as in page table.
  - So physical address = (1 x 4) + 2 = 06
- If logical address is 13 then, what is its corresponding physical address?
  - Page offset (d) = displacement within pages = 13 – 12 = 01
  - Hence Page number is = 3, which is in frame 2 given as in page table.
  - So physical address = (2 x 4) + 1 = 09
- If logical address is 15 then, what is its corresponding physical address?
  - Page offset (d) = displacement within pages = 15 – 12 = 3
  - Hence Page number is = 3, which is in frame 2 given as in page table.
  - So physical address = (2x 4) + 3 = 11

Problems with paging

- During process scheduling pages are loaded into the available frames & page table is updated.
- Paging scheme don’t suffers with external fragmentation.
- But Paging scheme suffers with internal fragmentation.
  - Because when process don’t comes under page boundaries, the last frame allocated may not be full.
  - Example:→ Page size = 2048 bytes

    Process size = 72,766 bytes

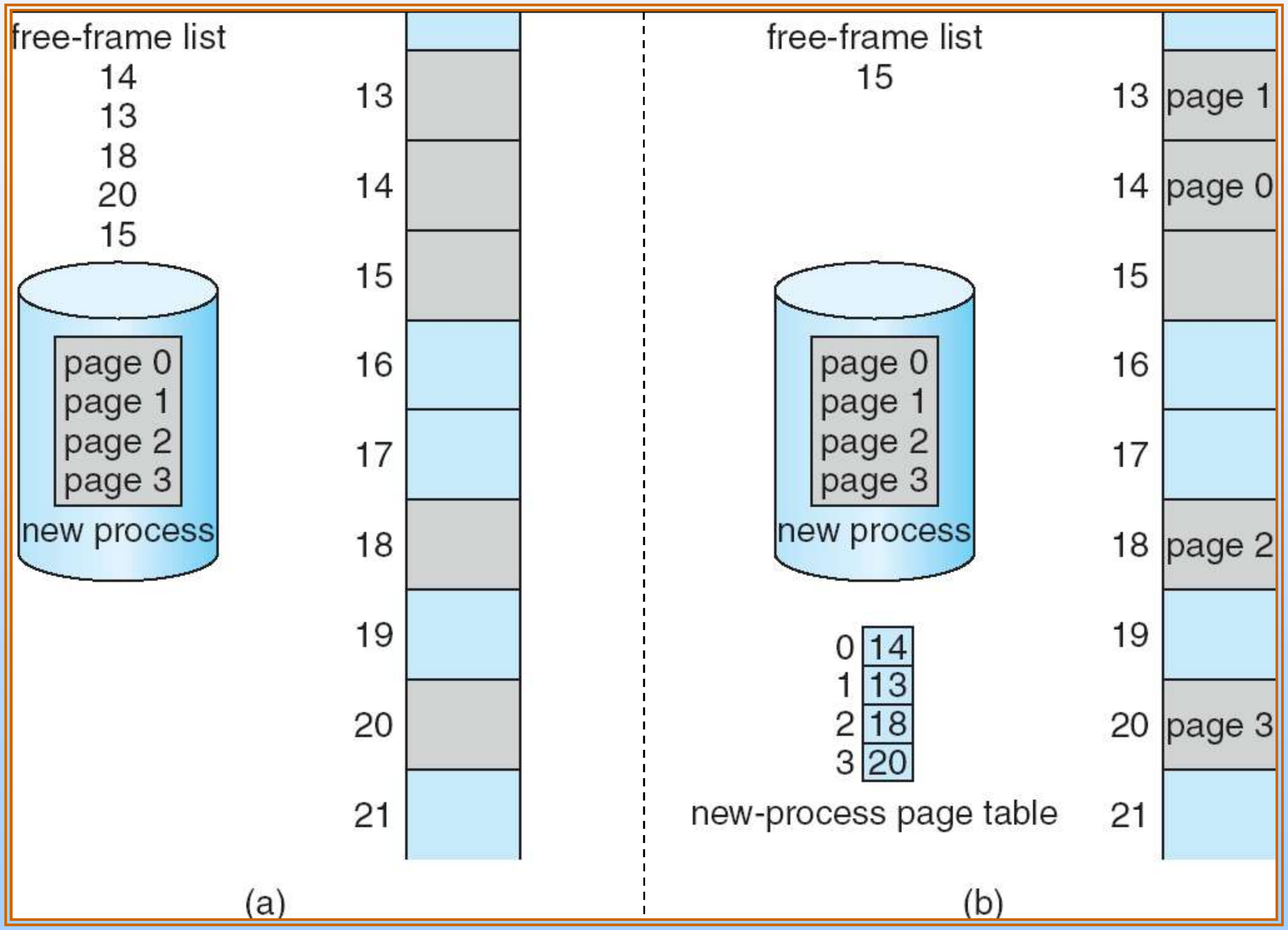
    so it needs 72,766 / 2048 = 35 (1035 bytes)

    So internal fragmentation is = 2048 – 1035 = 962 bytes free
- Remedy
  - As small as the page size to reduce internal fragmentation
  - Increase the page table size.
  - Make disk I/O more efficient that large data can be transferred.

User program execution with paging scheme

- Program size must be expressed in pages.
- Each page needs a frame.
- If n pages then there must be at least n frames available.
- Frame number is loaded on to the page table.
- Page table is saved to PCB.
- During page loading, correct page table may be updated.
- Frames of physical memory is kept a data structure called frame table.

Free Frames



Before allocation

After allocation

Implementation of Page Table

- Page table consist of dedicated registers with very high speed logic to make the address translation efficient.
- For large page table ,it must 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
- Example:→
  - If we want to access location i then
    - First page number index into the page table for i location.
    - Frame number & page offset can found from page table.
- In this scheme every data/instruction access requires two memory accesses.
  - One for the page table and one for the data/instruction.

Implementation of Page Table

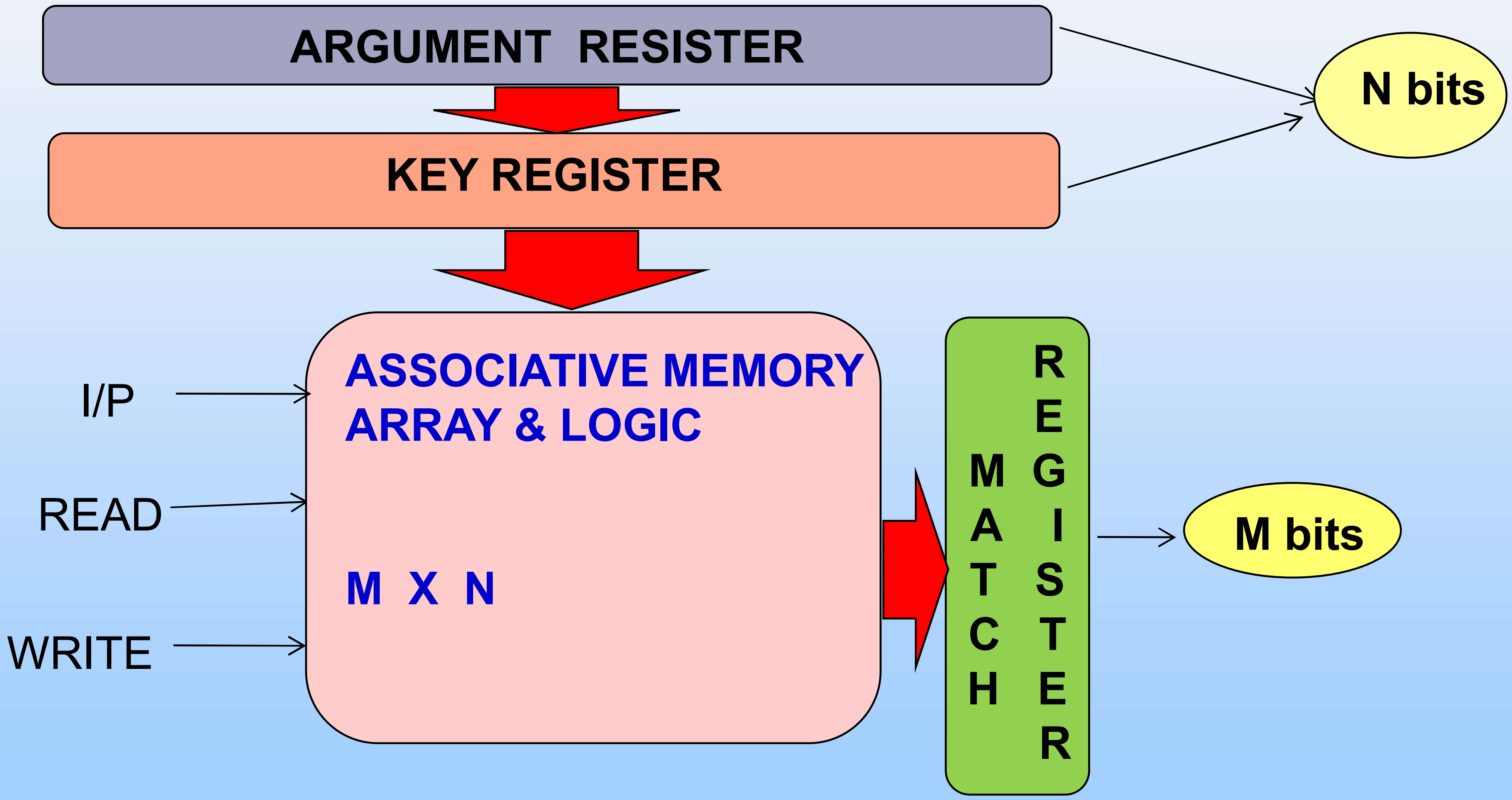
Continue….

- Solution:→
  - 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)**
  - **TLB register consist of two parts :- key & value**
  - Some TLBs store **address-space identifiers (ASIDs)** in each TLB entry – uniquely identifies each process to provide address-space protection for that process

Associative Memory

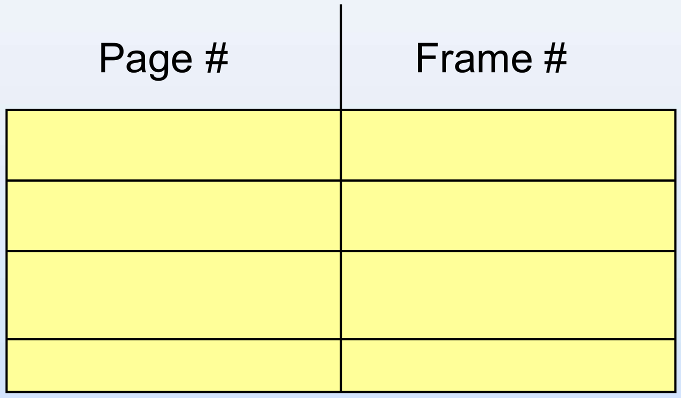
- Reduces the search time efficiently
- Address is replaced by content of data
- Called as Content based data.
- Hardwired Requirement :→
  - It contains memory array & logic for m words with n bits per each word.
  - Argument register (A) & Key register (k) each have n bits.
  - Match register (M) has m bits, one for each word in memory.
  - **Each word in memory is compared in parallel with the content of argument register and key register.**
  - **If a match found for a word which matches with the bits of argument register & its corresponding bits in the match register then a search for a data word is over.**

Associative Memory



TLB as Associative Memory

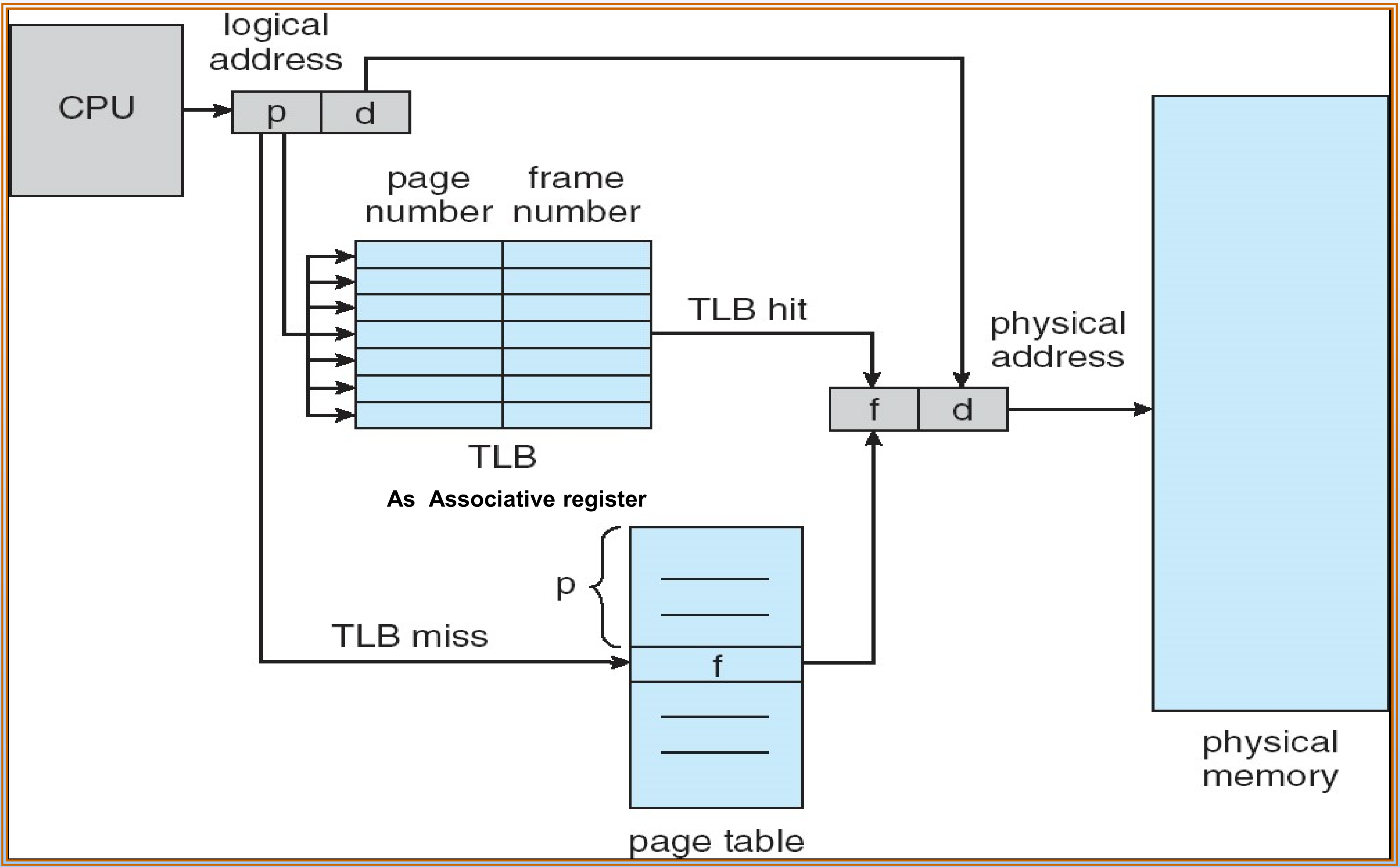
- Associative memory – parallel search



Address translation (p, d)

- If p is in associative register, get frame # out → **TLB HIT**
- Otherwise get frame # from page table in memory → **TLB MISS**

Paging Hardware With TLB



Effective Access Time

- EAT = $P_H (T_A + T_M) + (1 - P_H) (T_A + 2T_M)$
- Where
  - $P_H \rightarrow$ Hit ratio – percentage of times that a page number is found in the associative registers; ratio related to number of associative registers
  - $T_A \rightarrow$ Time to search associative memory.
  - $T_M \rightarrow$ Memory access time.
- Example:→
  - Hit ratio for associative memory = 80 %
  - Time to search associative memory (T_A) = 20 ns
  - Memory access time (T_M) = 100 ns
  - EAT = 0.80 (20 + 100) + (1 – 0.80) (20 + [2 x 100])

    = (0.8 x 120) + (0.2 x 220) = 96 + 44 = 140 ns
  - Hence we suffers with 40 % slowdown in memory access time.

Advantages of Paging

- Possibility of sharing common code in time sharing environment.
- Each process has its own copy of register s and data storage .
- Each user page table maps on to the same physical copy of the shared code but data pages are mapped on to different frames.
- Read only nature of shared code should not be left to the correctness of the code

Memory Protection in paging

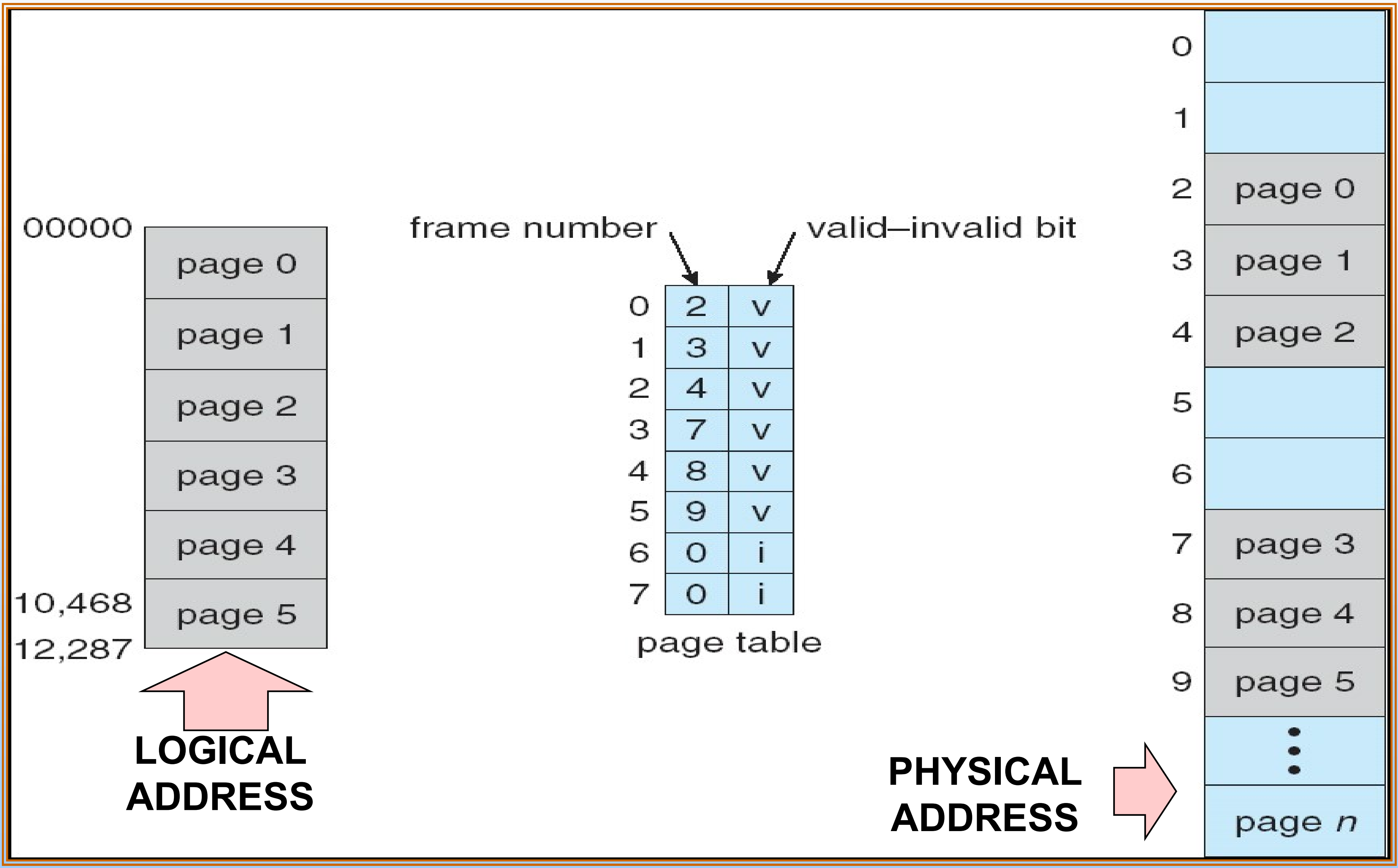
- Memory protection implemented by associating protection bit with each frame.
- One protection bit can define a page is read only or read /write.
- An attempt to write to a read only page causes a hardwire trap (memory protection violation) to the operating system.
- **Valid-invalid** bit attached to 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 and is thus an illegal page.

Memory Protection in paging

Continue……

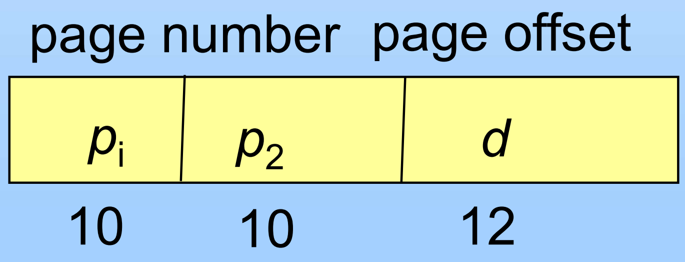
- Example:-
  - Total memory space is 16383 bytes.
  - Used memory space is 10468 bytes.
  - Page size 2000 bytes.
  - Total number of page required = 10468 / 2000 = 5 (468 / 2000)
    = 6 pages
  - So the 0,1,2,3,4,5 are valid pages.
  - But any attempt to find page 6 or 7 or other pages (Invalid pages) will be trapped by operating system.

Valid (v) or Invalid (i) Bit In A Page Table



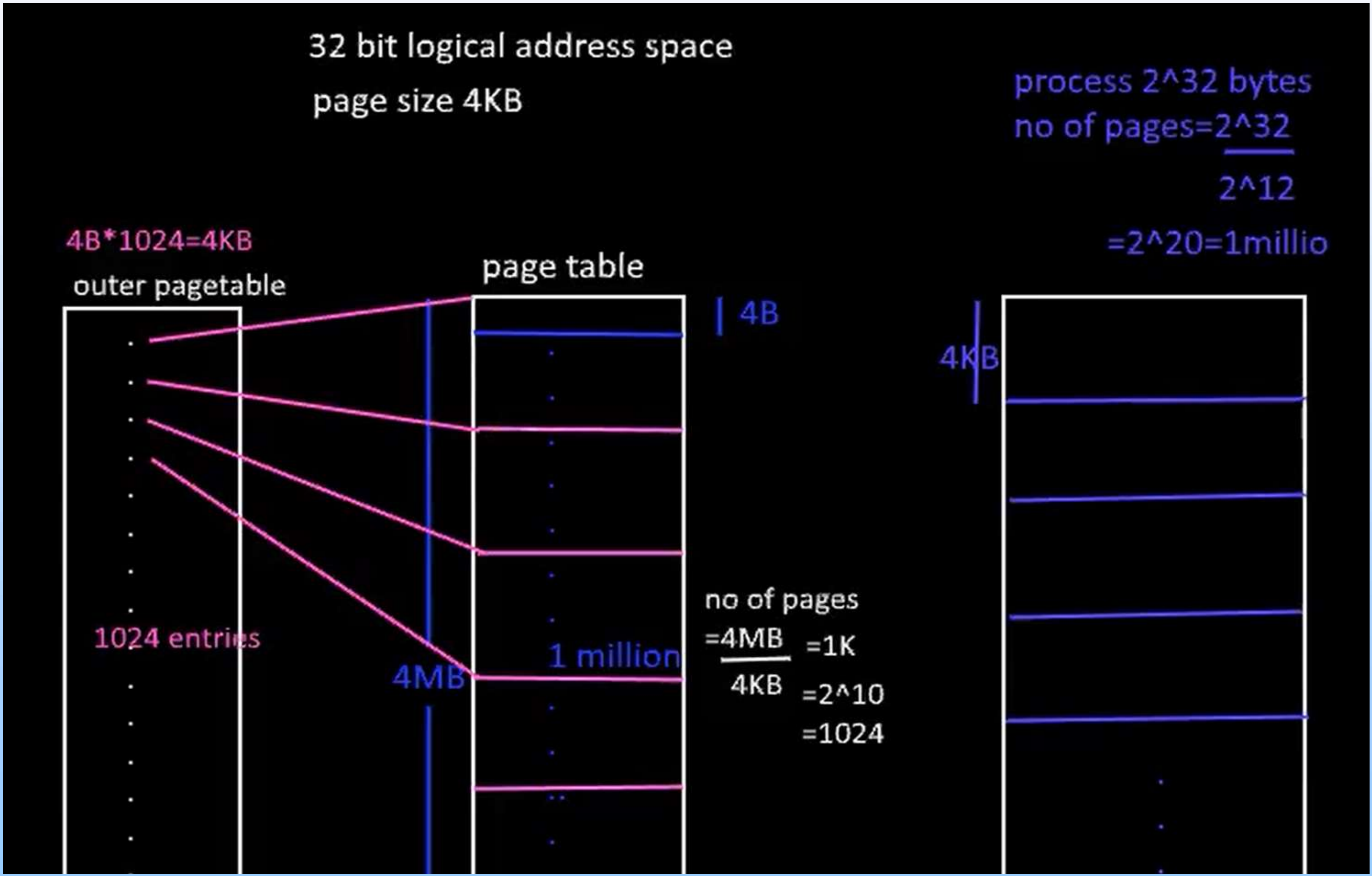
Two-Level Paging Example

- A logical address (on 32-bit machine with 4K page size) is divided into:
  - a page number consisting of 20 bits.
  - a page offset consisting of 12 bits.
- Since the page table is paged, the page number is further divided into:
  - a 10-bit page number.
  - a 10-bit page offset.
- Thus, a logical address is as follows:

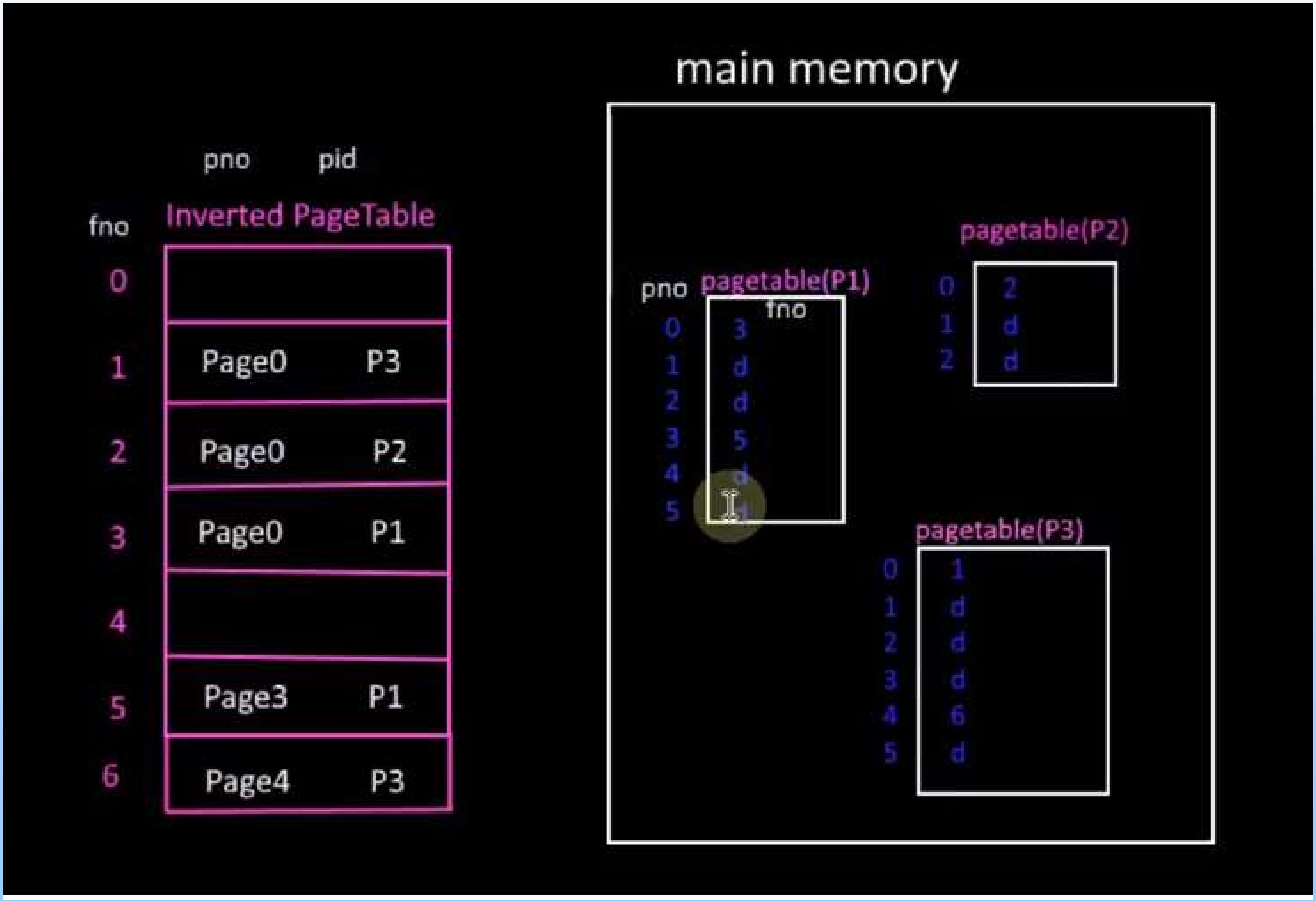


where p_i is an index into the outer page table, and p_2 is the displacement within the page of the outer page table.

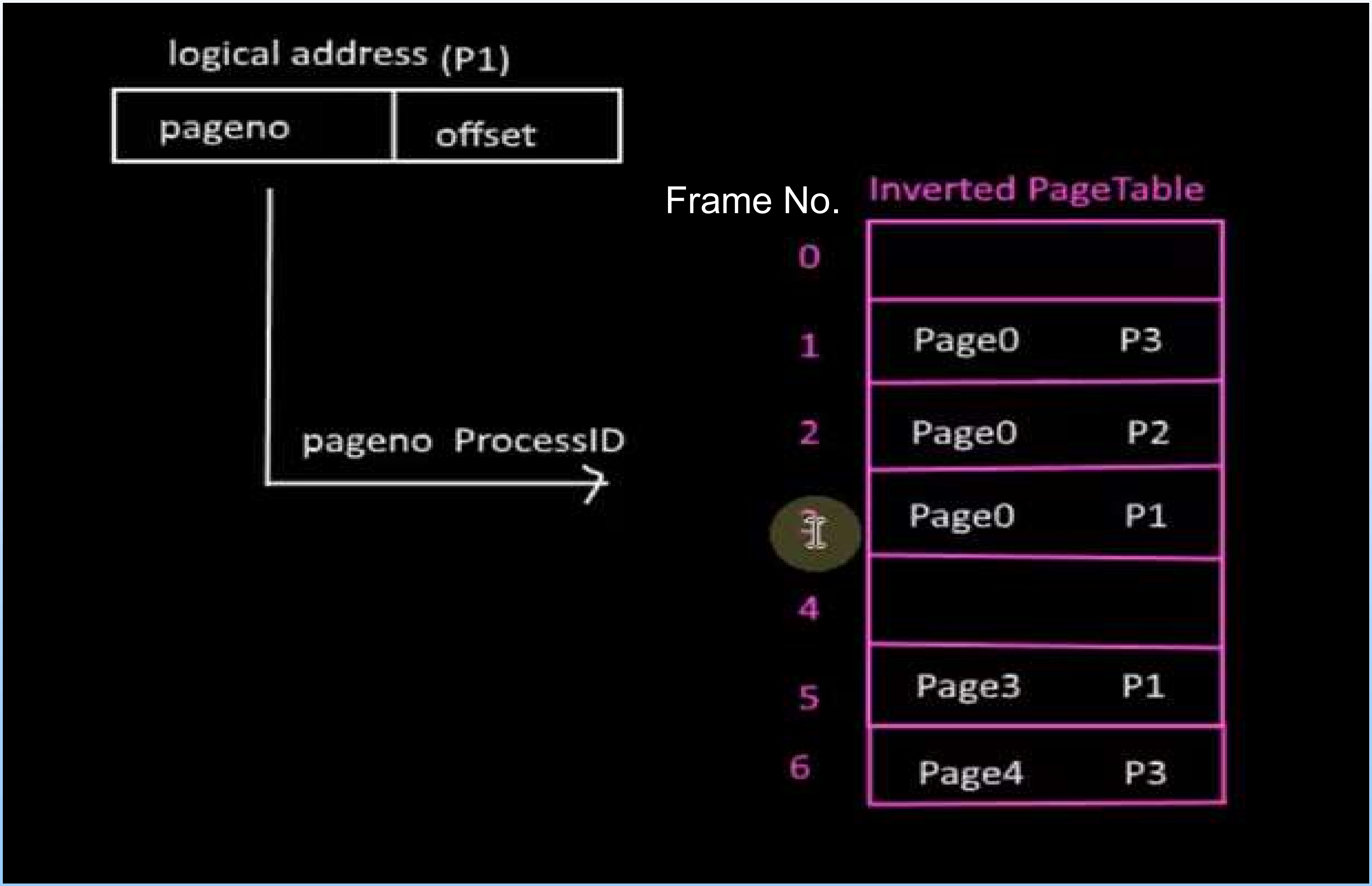
Two-Level Paging Example



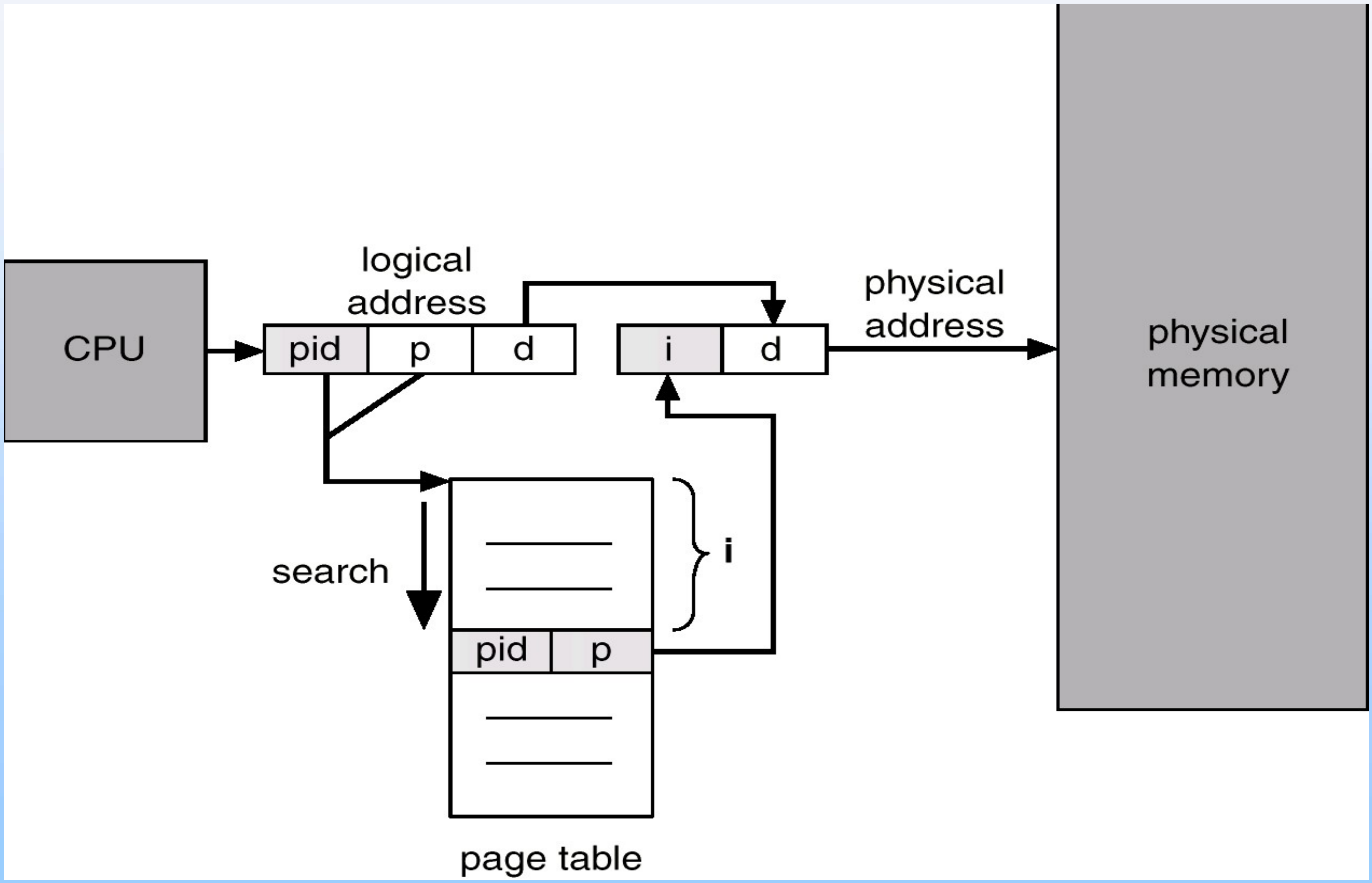
Inverted Page Table Architecture



Inverted Page Table Architecture



Inverted Page Table Architecture



Shared Pages

- **Shared code**
  - One copy of read-only (reentrant) code shared among processes (i.e., text editors, compilers, window systems).
  - Shared code must appear in same location in the logical address space of all processes

- **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

Shared Pages Example

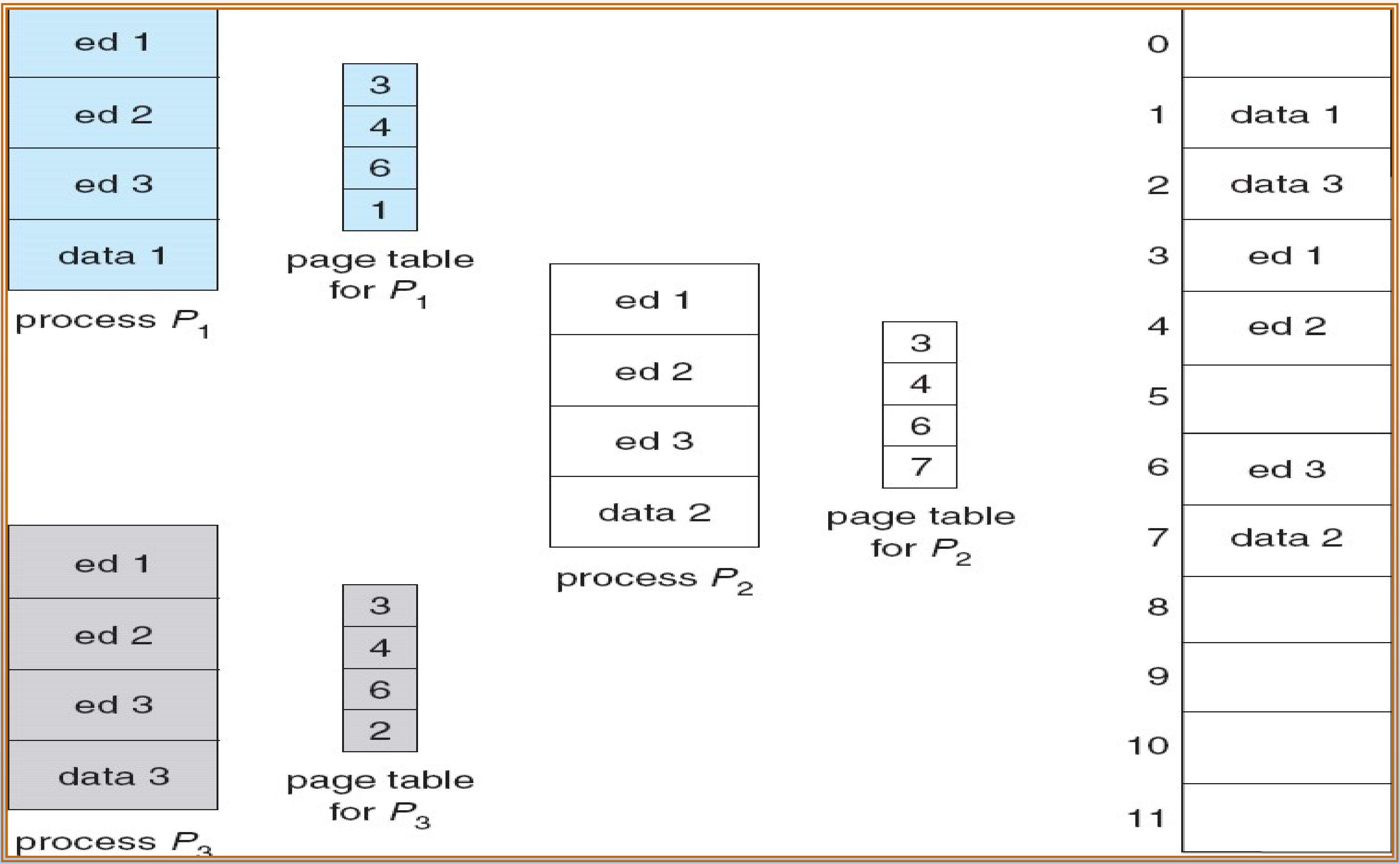
- Suppose a system supports 40 users.
- Each user executes a text editor of 150 kb of code & 50 kb of data space.
- Hence total space required = (150 + 50) x 40 = 8000 kb
- But shared page scheme can reduce the total space (8000 kb) required .
- Solution:→
  - If code is reentrant (pure code) it can be shared. **Text editor**
  - Code is reentrant means non self modifying code (CODE NEVER CHANGES DURING EXECUTION), only fetch / read can possible.
  - Non self modifying code can be used by more than one user at once.

Shared Pages Example

Continue…..

- Solution:→
  - Physical memory must contain a copy of text editor & data required for each user.
  - Hence each user can have a copy of the text editor code along with data page in their executable space, by which execution can possible.
  - Each user's page table maps on to the same physical copy of the text editor but the data pages are mapped on to different frames in the physical memory.
  - Divide text editor into 3 pages (50 kb each) & share among all 40 users.
  - But each user must have its own data page (50 kb each).
  - Hence to support 40 users total space required is
    - **150 kb + (50 kb x 40) = 2150 kb**

Shared Pages Example

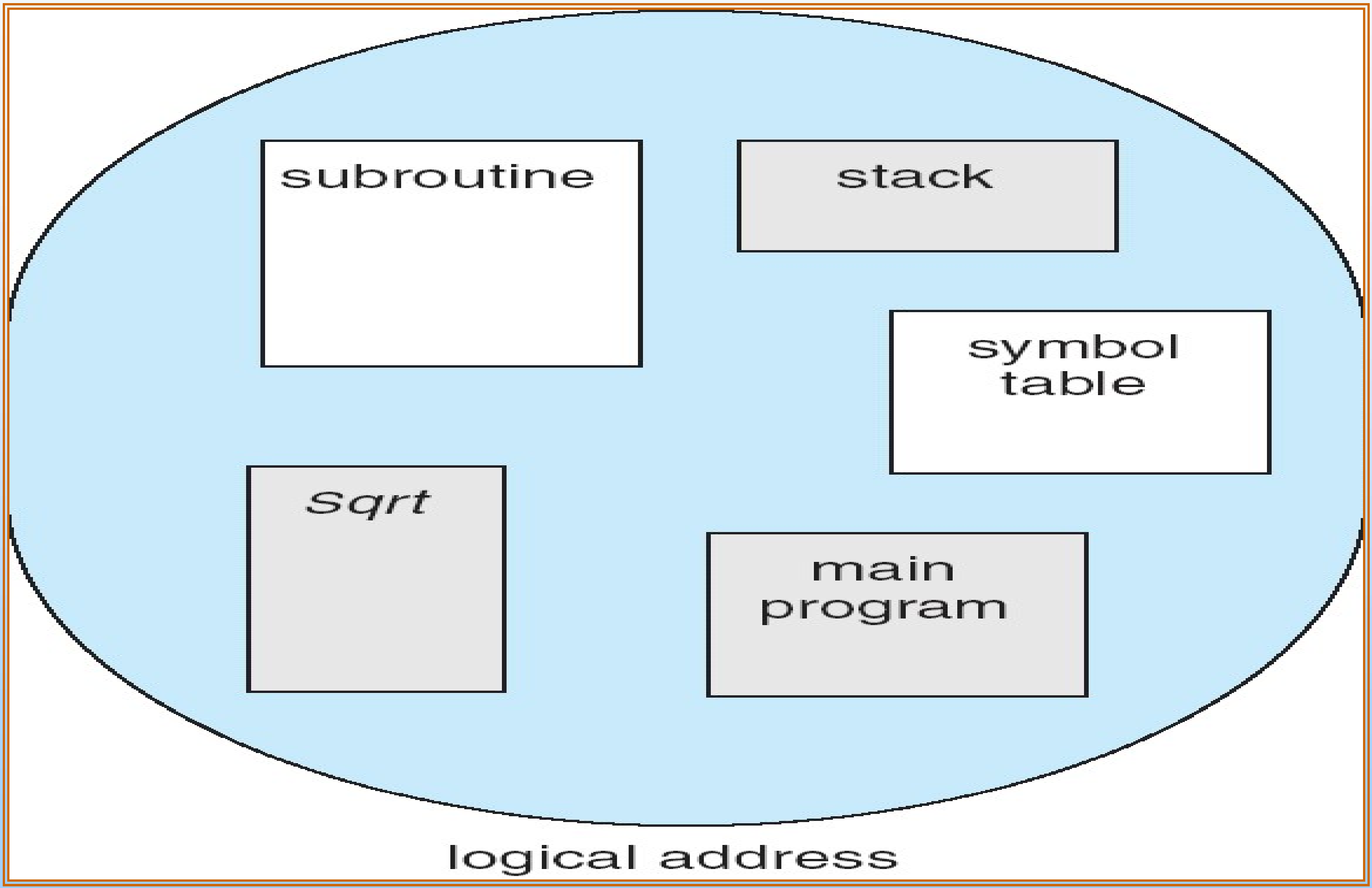


Segmentation

- Memory-management scheme that supports user view of memory
- Actually user view is not a linear array of memory space. It prefers to view memory as a collection of variable sized segments.
- Hence, A program is a collection of segments. A segment is a logical unit such as:

Main program,	**Local variables**
Procedure,	**Global variables**
Function,	**common block,**
Method,	**Stack,**
Object,	**Symbol table**
Subroutine	**Arrays**

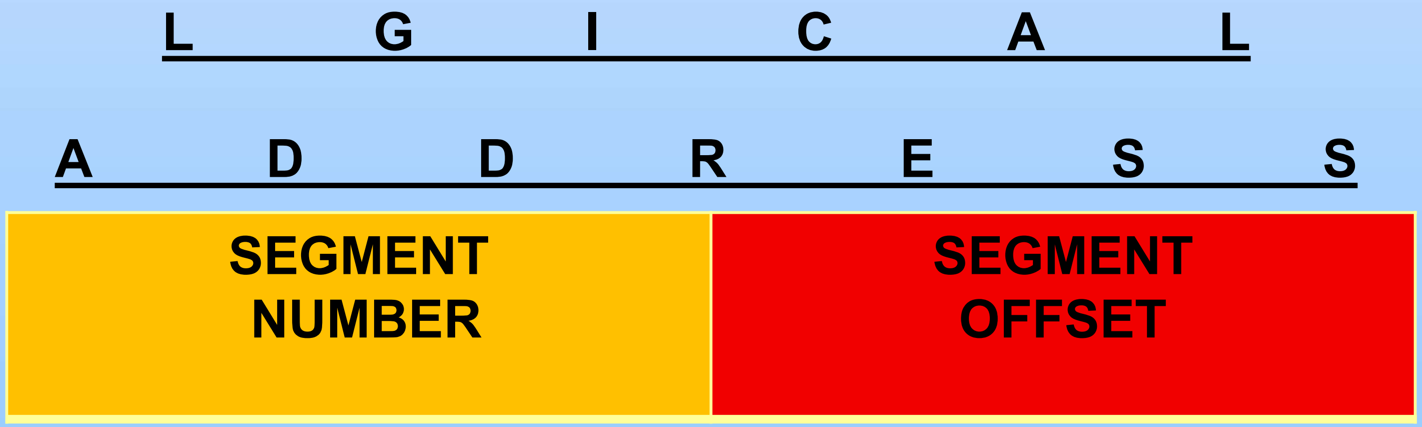
User’s View of a Program



Logical View of Segmentation

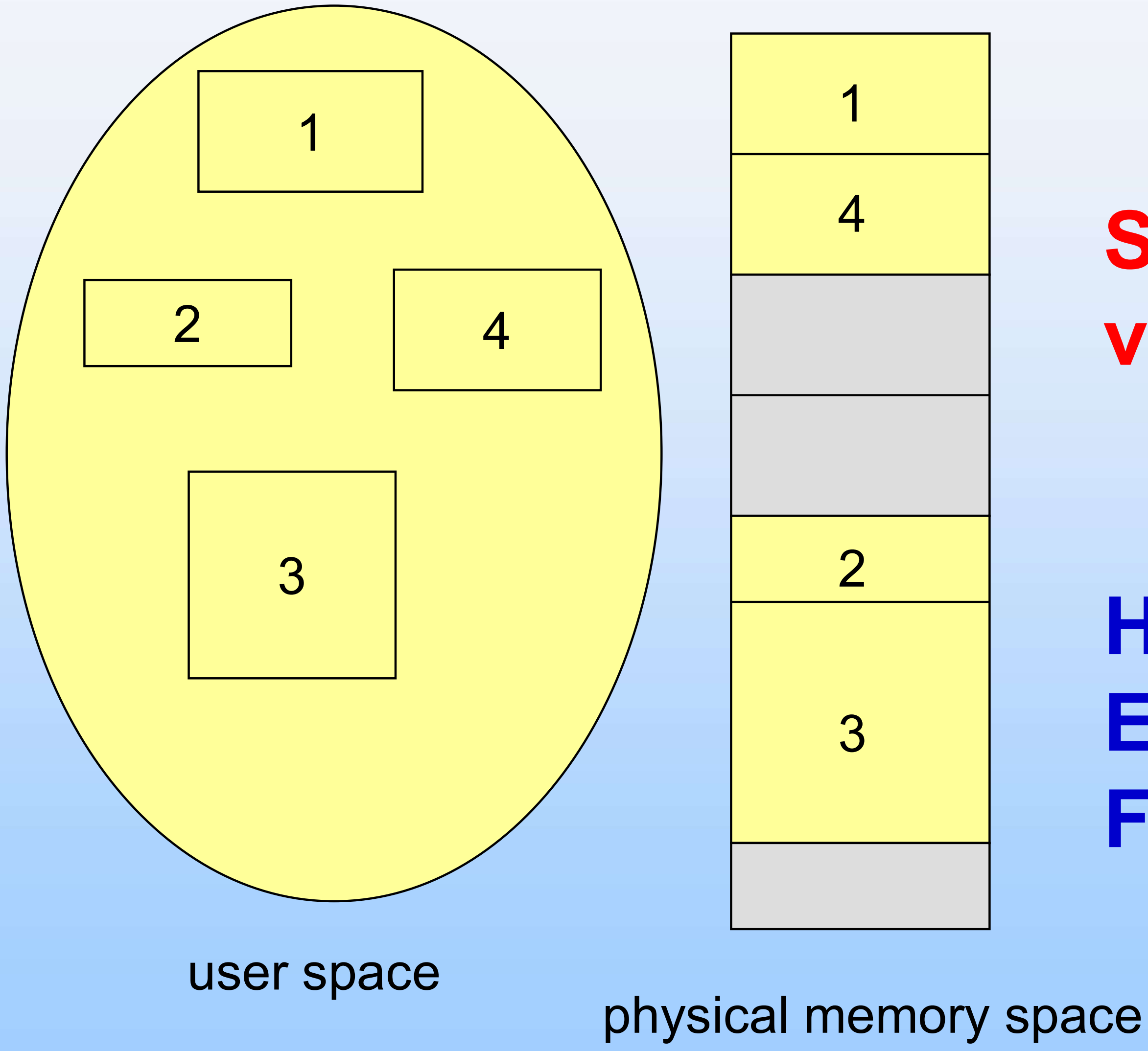
- Segmentation supports the view of memory
  - A logical address space is a collection of segments .
  - Logical Address specify both the segment number & offset within the segment.
  - Each segment has a number & length.
  - Logical address consists of a two tuple:

    < segment-number, segment offset >



Logical View of Segmentation

Continue……



Segmentation uses variable size partition.

Hence it suffers from EXTERNAL FRAGMENTATION.

Implementation of Segment Table

- Segment table can be either in buffer or in main memory.
- **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**
- Segment table entry = STBR + S(offset)
- Read the above result from memory & segment offset is checked against the segment length ,hence the physical address can be found.

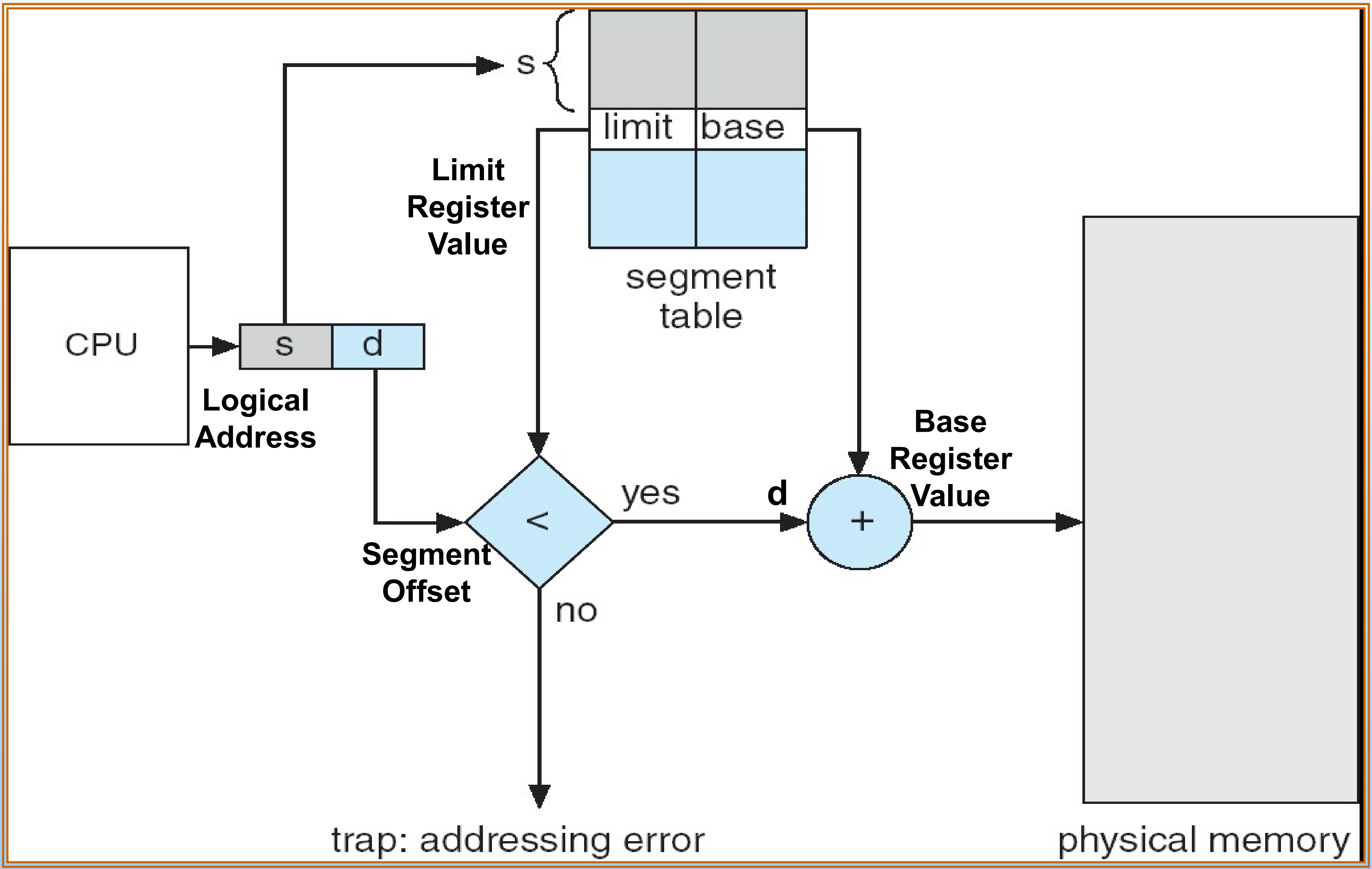
LA $\longrightarrow$ Address Mapping $\longrightarrow$ PA

- Segment table helps in mapping.
  - Logical address consists of a two tuple:

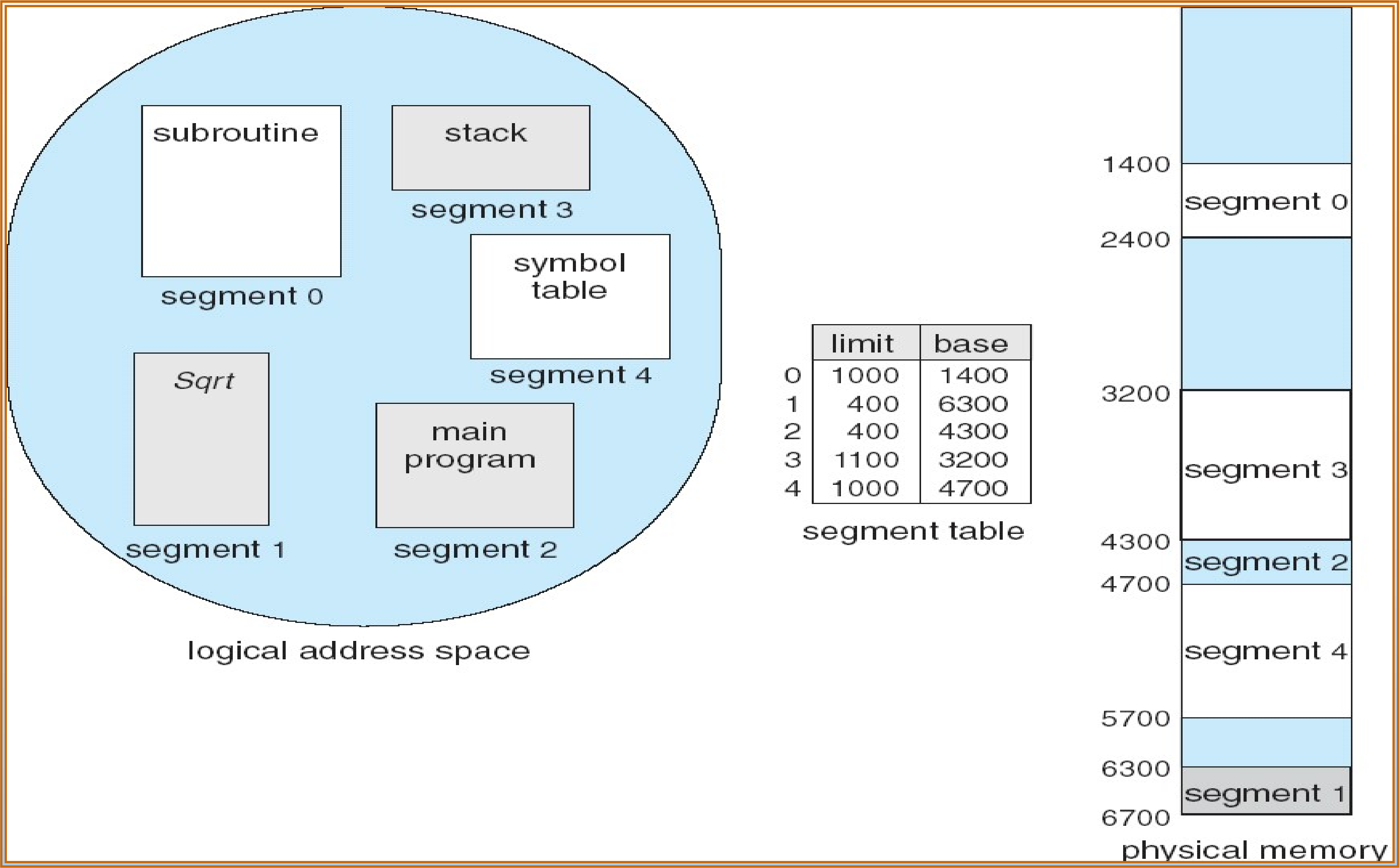
    < segment-number, segment offset >
- Segment number is used as an index to the segment table.
- Each entry to the segment table has a base (starting physical address) and limit (range of segment) register.
- The segment offset given in logical address must occur between the range given by limit register (legal offset) otherwise there is an addressing error.
- If (Segment Offset Value < Limit Register Value)

```
If  (Segment Offset  Value  <  Limit Register Value)
        then
                ADD Base Register Value with Segment Offset value
        else
                Addressing  Error
```

Segmentation Hardware



Example of Segmentation



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

segment table

LA $\rightarrow$ Address Mapping $\rightarrow$ PA

- Example :→
  - Here we have 5 segments.
- If a logical address is

02	53
S	d

what is its corresponding physical address?

then check **53 < 400**

Physical Address = 4300 + 53 = 4353

- If a logical address is

03	852
S	d

what is its corresponding physical address?

then check **852 < 1100**

Physical Address = 3200 + 852 = 4052

- If a logical address is

00	1222
S	d

what is its corresponding physical address?

then check **1222 >1000**

illegal segment offset (d).

Addressing Error (trap).

Protection in Segmentation

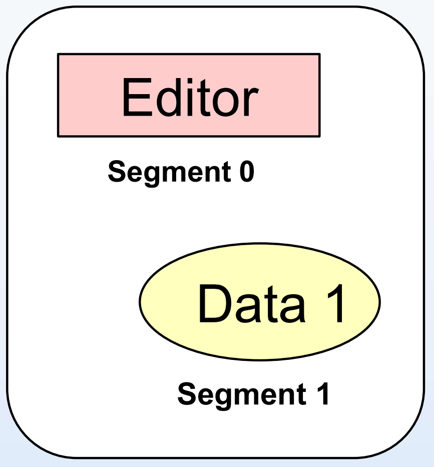
- A program consists of instruction segment, data segment, editor segment.
- Instructions are non self modifying, so read only.
- Protection bit is checked to prevent illegal access.
  - With each entry in segment table associate:
    - validation bit = 0 $\Rightarrow$ illegal segment
    - validation bit = 1 $\Rightarrow$ Legal segment
    - read/write/execute privileges
- Protection bits associated with segments; code sharing occurs at segment level
- Since segments vary in length, memory allocation is a dynamic storage-allocation problem

Sharing in Segmentation

- Instructions are non self modifying, (read only) hence the code can be sharable.
- When entries to the segment table of two different process points to same physical locations then the segments are able to share.
- Example:→
  - Consider a text editor in time sharing system.
  - If the text size is large, it is divided into multiple segments.
  - As the text editor is non self modifying, we can keep one copy on physical memory along with data for respective process.
  - The text editor is shared among all the process for their execution

Sharing in Segmentation

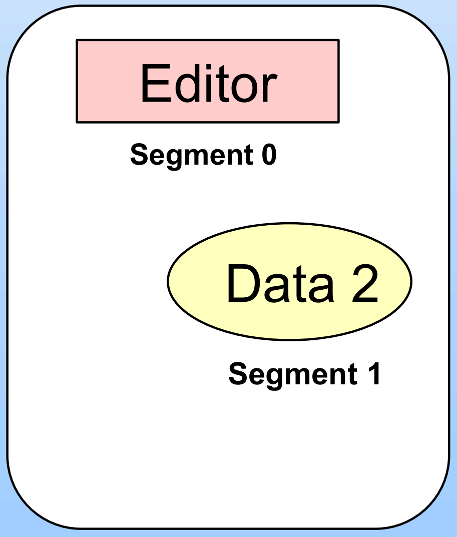
Continue….



Logical Memory
Process P1

	Limit	Base
0	25286	43062
1	4425	68348

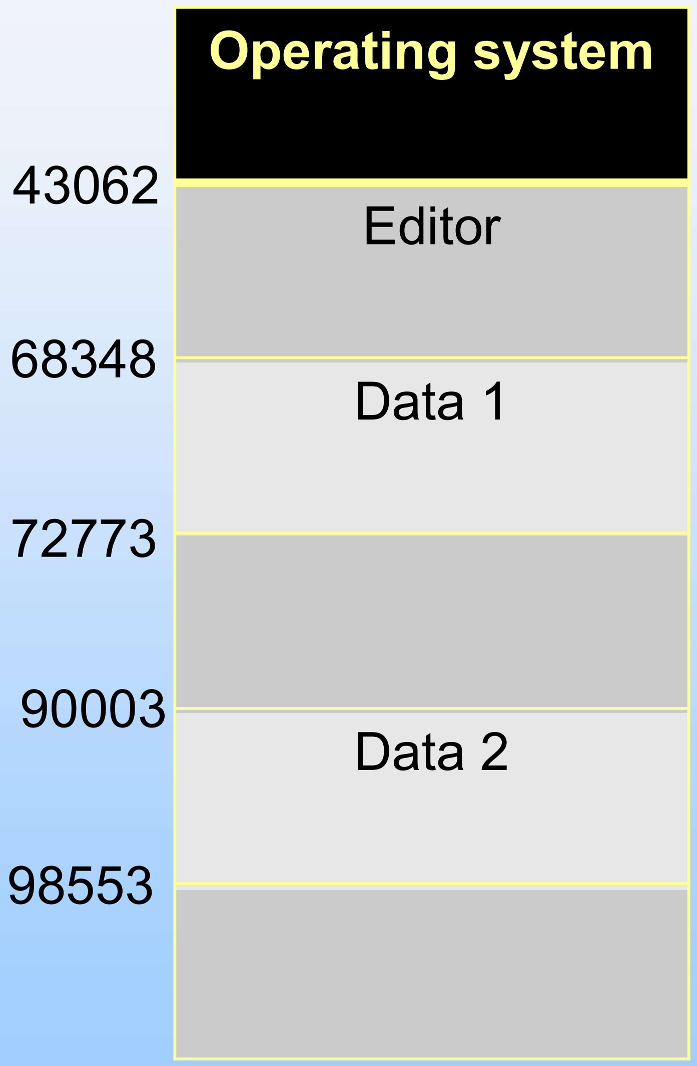
Segment Table
Process P1



Logical Memory
Process P2

	Limit	Base
0	25286	43062
1	8550	90003

Segment Table
Process P1

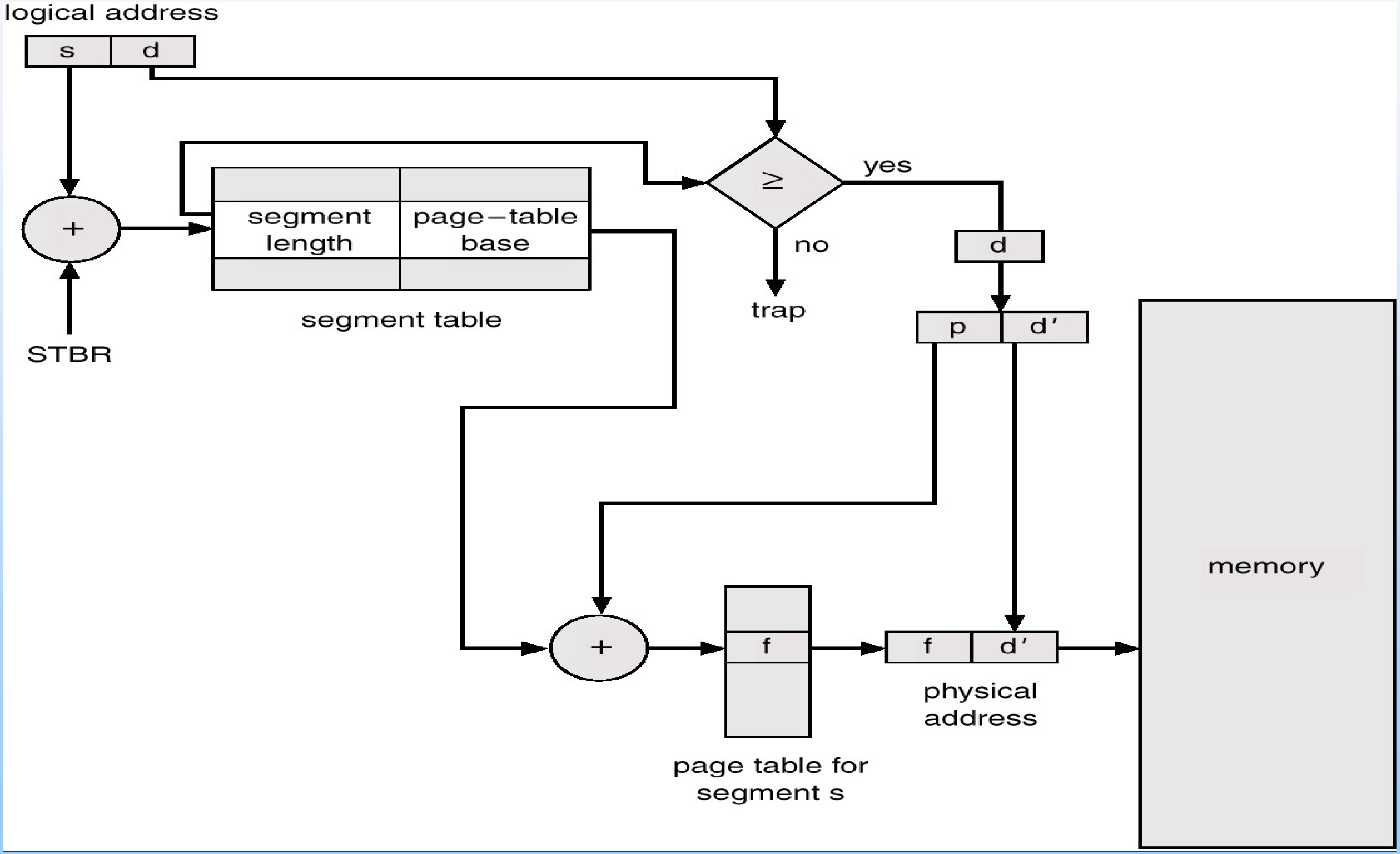


Physical Memory

Segmentation with Paging – MULTICS

1. *The MULTICS system solved problems of external fragmentation and lengthy search times by paging the segments.*
2. *Solution differs from pure segmentation in that the segment-table entry contains not the base address of the segment, but rather the base address of a page table for this segment.*

MULTICS Address Translation Scheme



End of Chapter 8