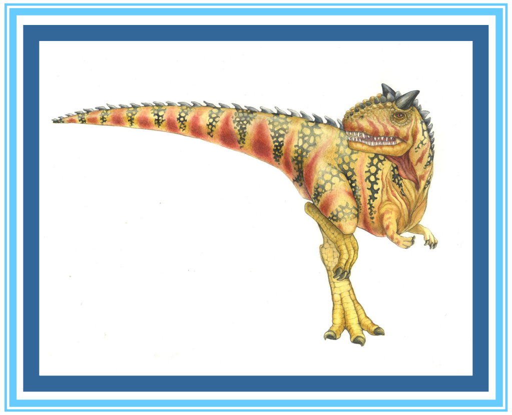
UNIT-4(CH-1): Main Memory



Chapter 8: Memory Management

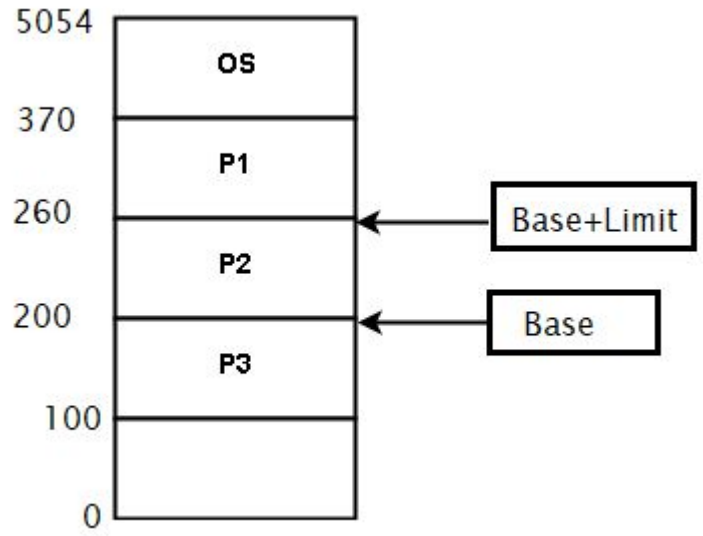
- ~~Background~~
- ~~Contiguous Memory Allocation~~
- ~~Paging~~
- ~~Structure of the Page Table~~
- Swapping

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
- by using two registers, usually a base and a limit, as illustrated in Figure 9.1. 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 holds 300040 and the limit register is 120900, then the program can legally access all addresses from 300040 through 420939 (inclusive).



Simplified example (for ur reference)

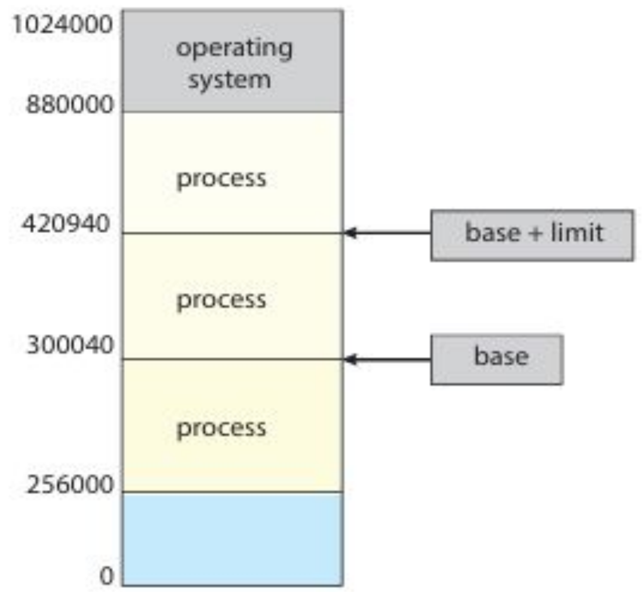


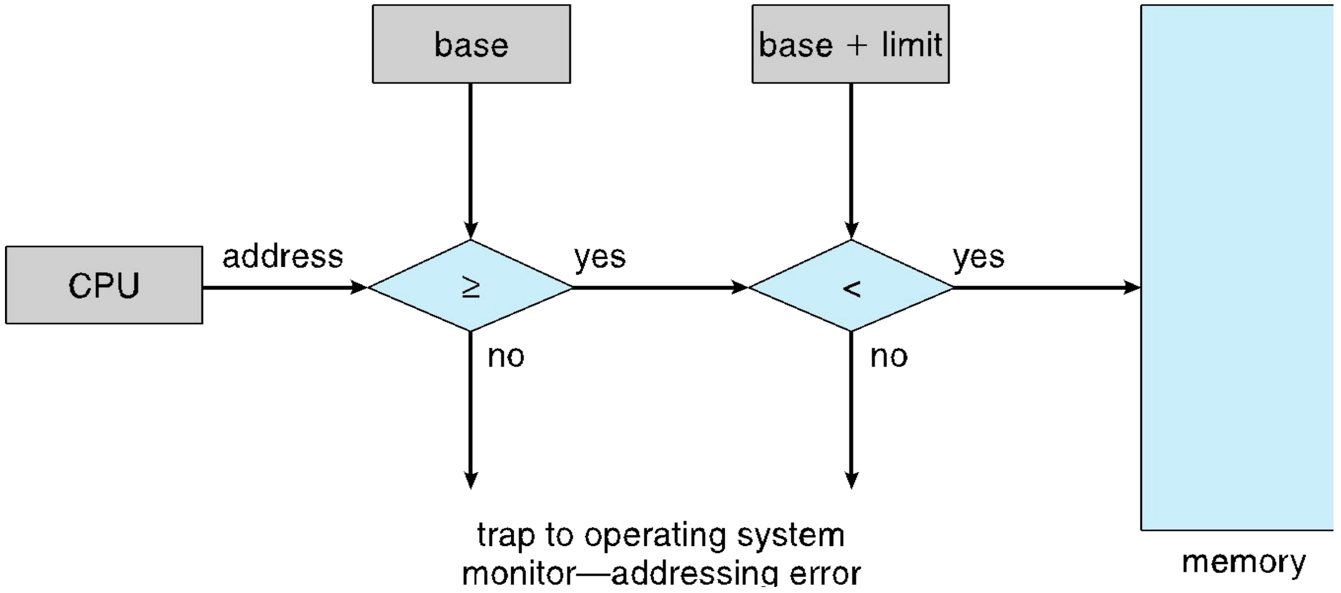
Figure 9.1 A base and a limit register define a logical address space.

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 (Figure 9.2).
This scheme prevents a user program from (accidentally or deliberately) modifying the code or data structures of either the operating system or other users.

Figure 9.2 Hardware address protection with base and limit registers.



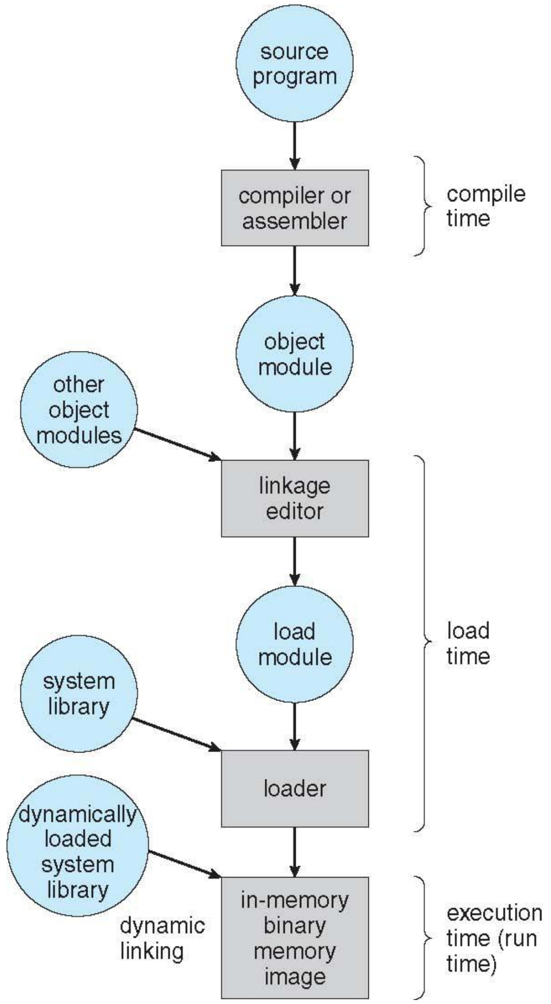
Address Binding

- Programs on disk, ready to be brought into memory to execute form an **input queue**
  - Without support, must be loaded into address 0000
- Further, addresses represented in different ways at different stages of a program's life
  - Source code addresses usually symbolic
  - Compiled code addresses **bind** 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

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

Multistep Processing of a User Program



Logical vs. Physical Address Space

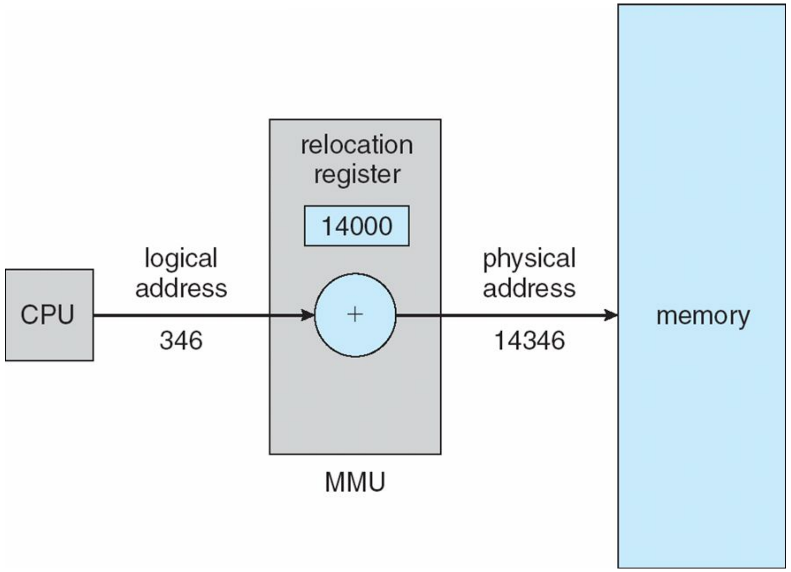
- The concept of a logical address space that is bound to a separate **physical address space** is central to proper memory management
  - **Logical address** – generated by the CPU; also referred to as **virtual address**
  - **Physical address** – address seen by the memory unit
- Logical and physical 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)

- Hardware device that at run time maps virtual to physical address
- 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**
  - MS-DOS on Intel 80x86 used 4 relocation registers
- The user program deals with *logical* addresses; it never sees the *real* physical addresses
  - Execution-time binding occurs when reference is made to location in memory

Dynamic relocation using a relocation register

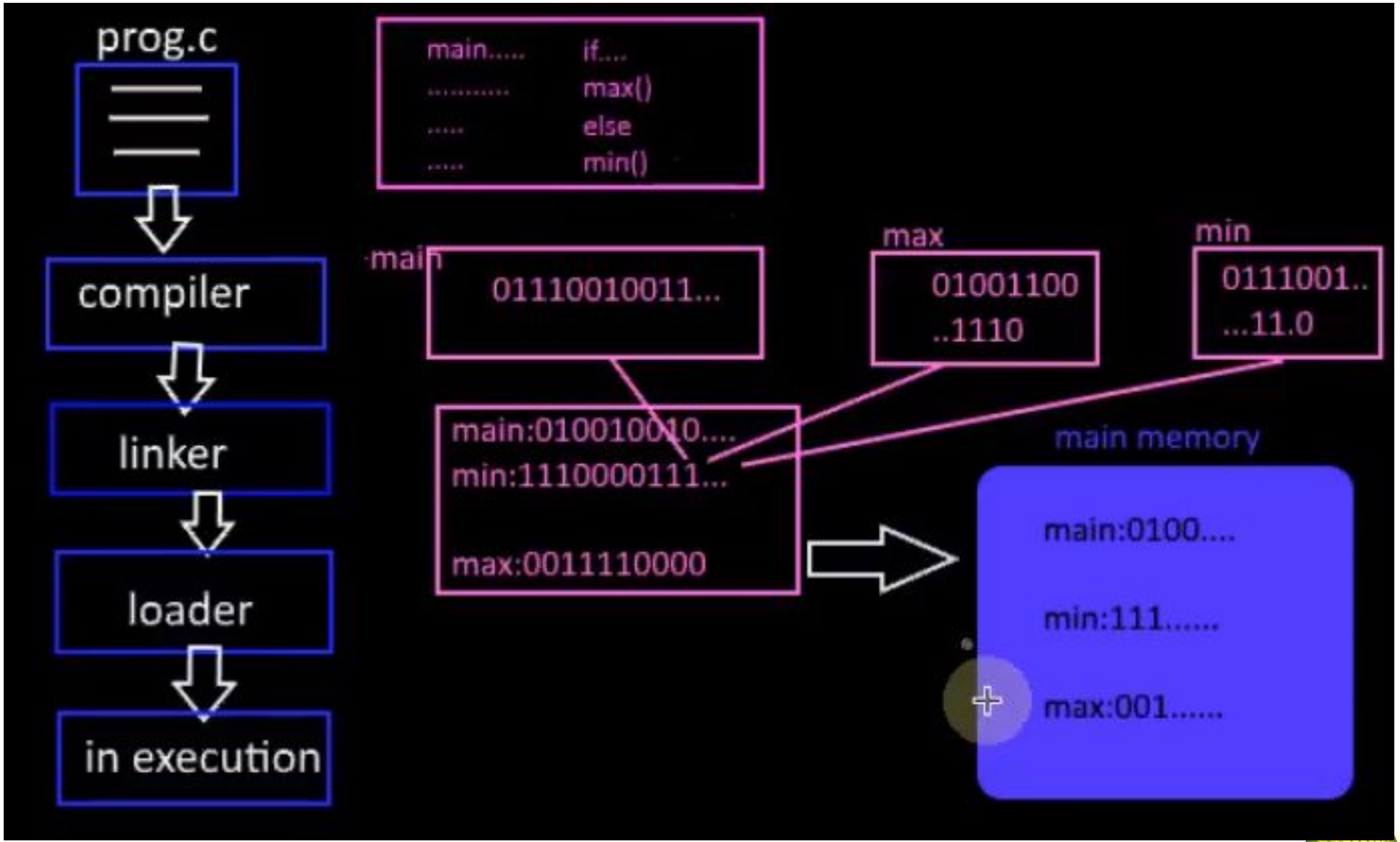
- Routine is not loaded until it is called
- **Better memory-space utilization;** unused routine is never loaded
- All routines kept on disk in **relocatable load format**
- **Useful when large amounts of code** are needed to handle infrequently occurring cases
- **No special support** from the operating system is required
  - Implemented through program design
  - OS can help by providing libraries to implement dynamic loading



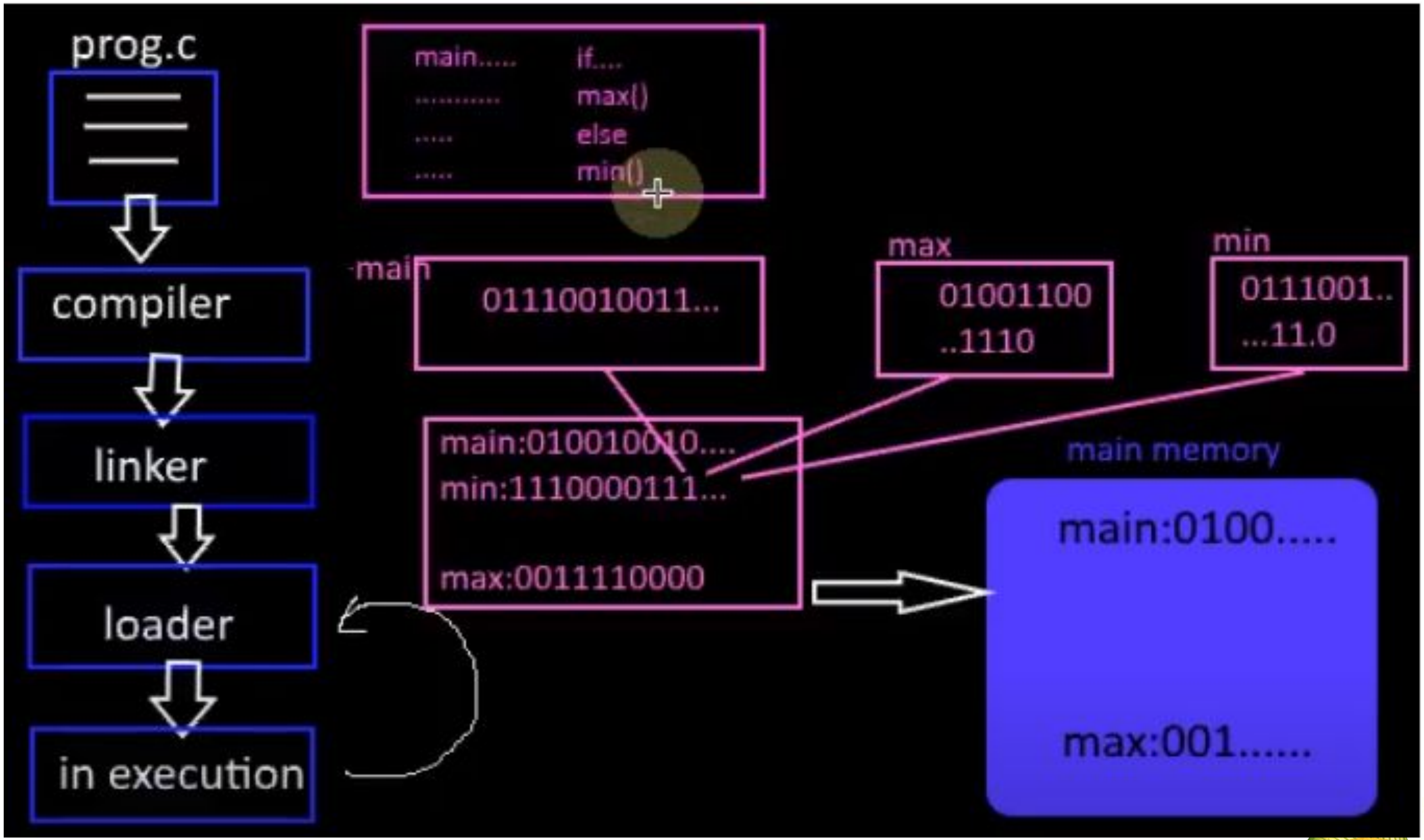
Dynamic loading

- To obtain better memory-space utilization, we can use dynamic loading. With dynamic loading, a routine is not loaded until it is called.
- All routines are kept on disk in a relocatable load format. The main program is loaded into memory and is executed.
- When a routine needs to call another routine, the calling routine first checks to see whether the other routine has been loaded.
- If it has not, the relocatable linking loader is called to load the desired routine into memory and to update the program's address tables to reflect this change. Then control is passed to the newly loaded routine.
- The advantage of dynamic loading is that a routine is loaded only when it is needed.
- This method is particularly useful when large amounts of code are needed to handle infrequently occurring cases, such as error routines. In such a situation, although the total program size may be large, the portion that is used (and hence loaded) may be much smaller.

Dynamic loading with an example



dynamic loading(contd.,)



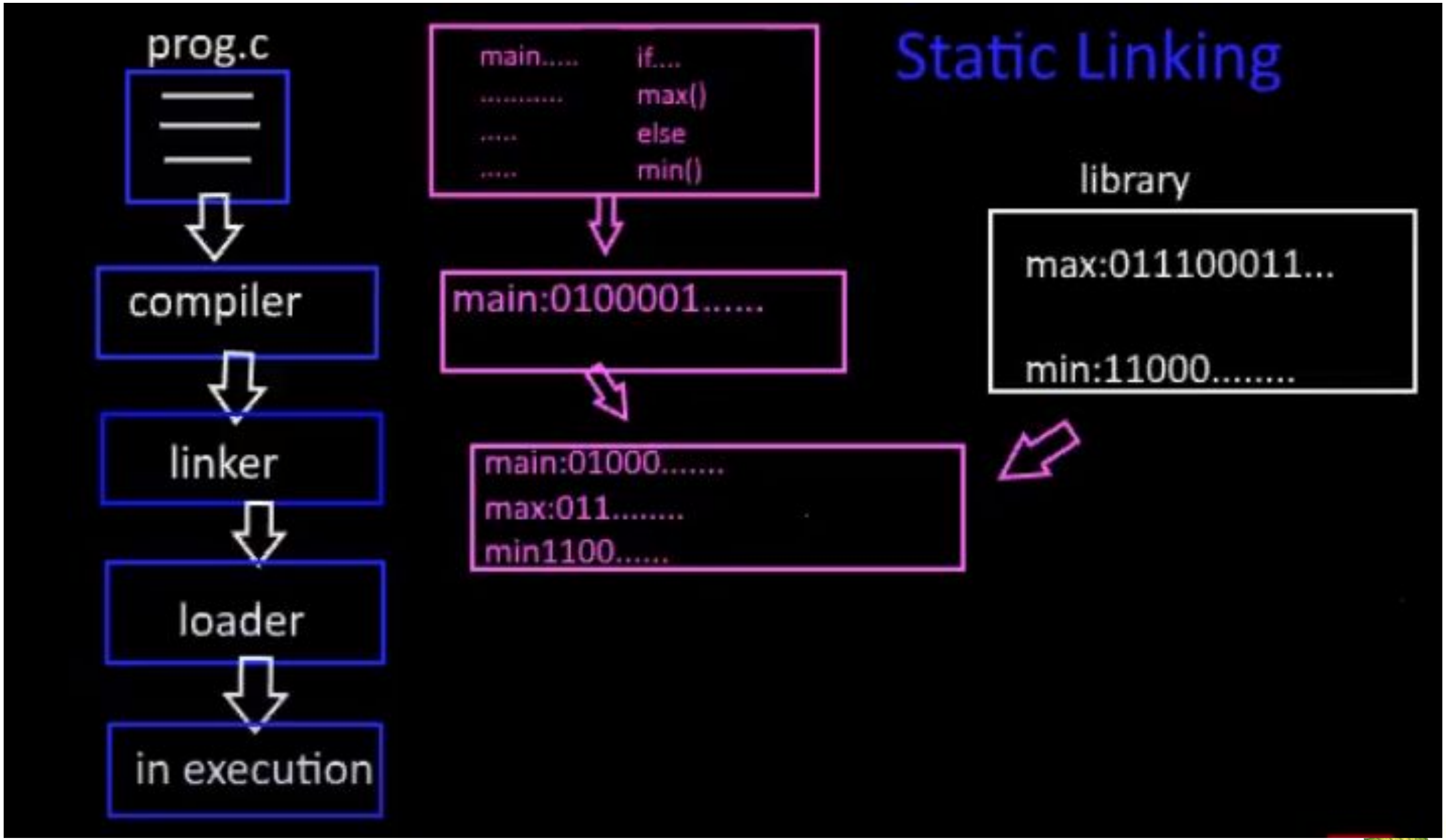
Dynamic linking

- Dynamically linked libraries(DLLs) are system libraries that are linked to user programs when the programs are run.
- Some operating systems support only static linking, in which system libraries are treated like any other object module and are combined by the loader into the binary program image.
- Dynamic linking, in contrast, is similar to dynamic loading. Here, though, linking, rather than loading, is postponed until execution time. This feature is usually used with system libraries, such as the standard C language library.
- Without this facility, each program on a system must include a copy of its language library (or at least the routines referenced by the program) in the executable image.
- This requirement not only increases the size of an executable image but also may waste main memory.
- A second advantage of DLLs is that these libraries can be shared among multiple processes, so that only one instance of the DLL in main memory. For this reason, DLL are also known as shared libraries, and are used extensively in Windows and Linux systems

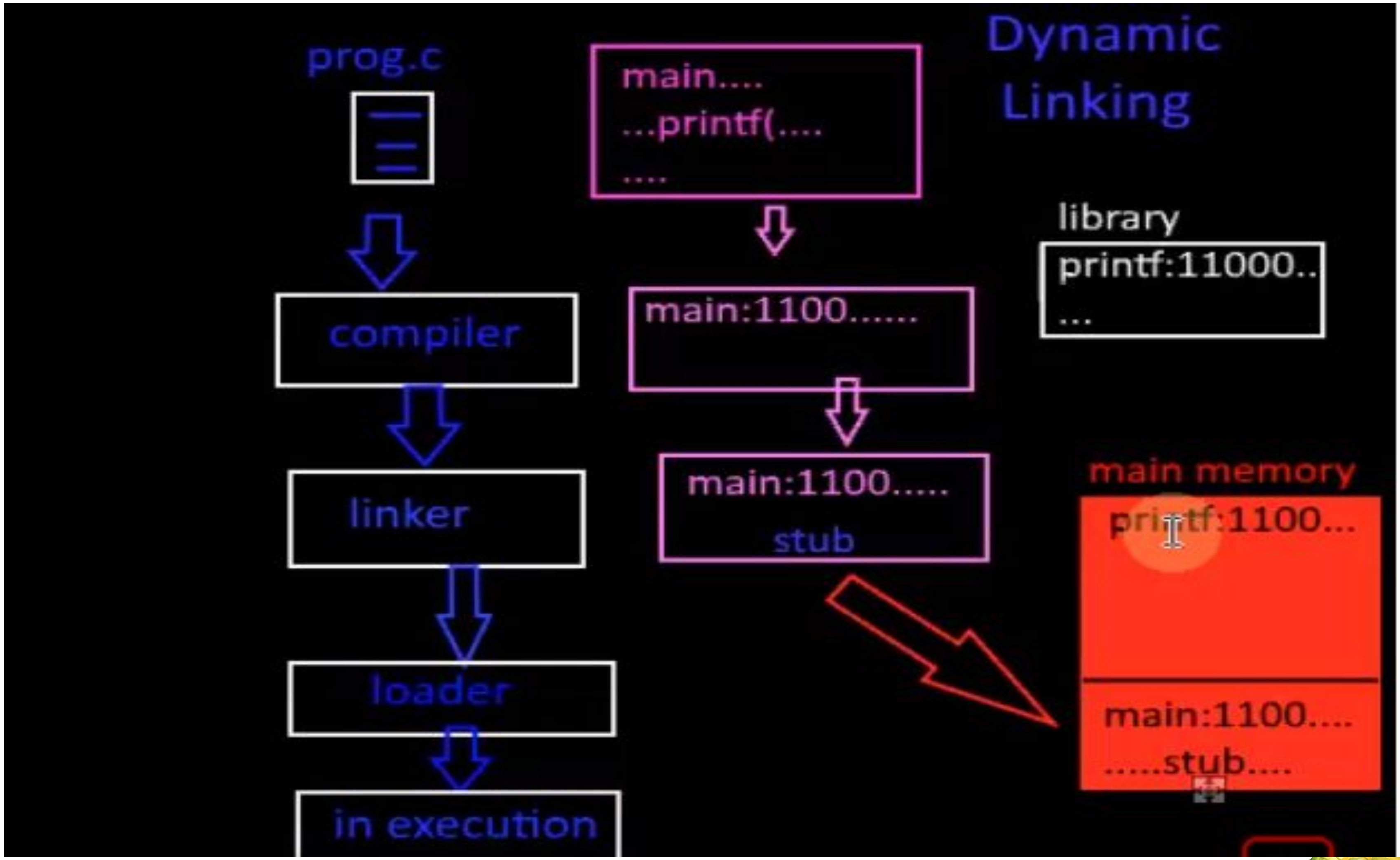
Dynamic Linking (contd..,)

- **Static linking** – system libraries and program code combined by the loader into the binary program image
- 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
- System also known as **shared libraries**

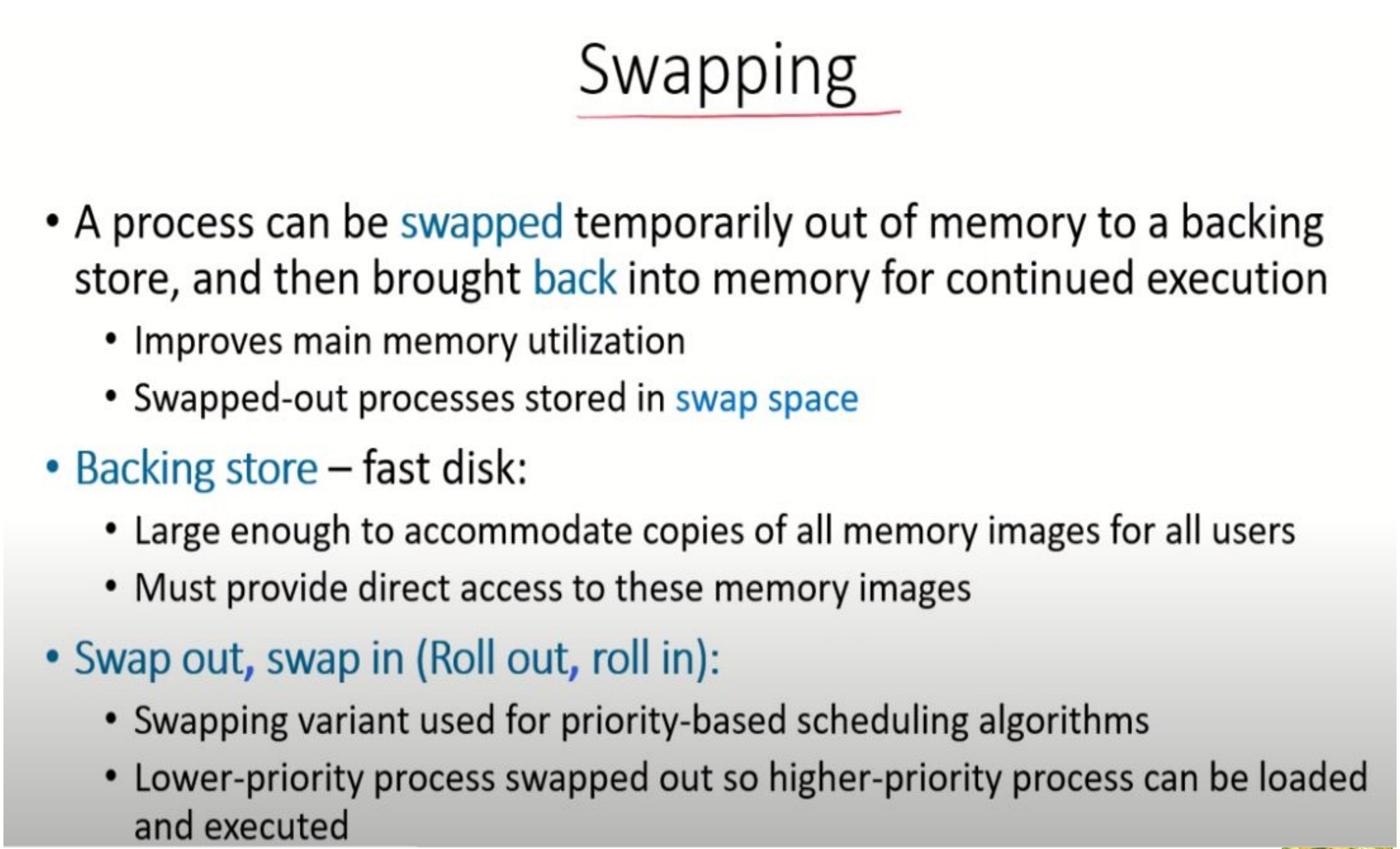
static linking



dynamic linking



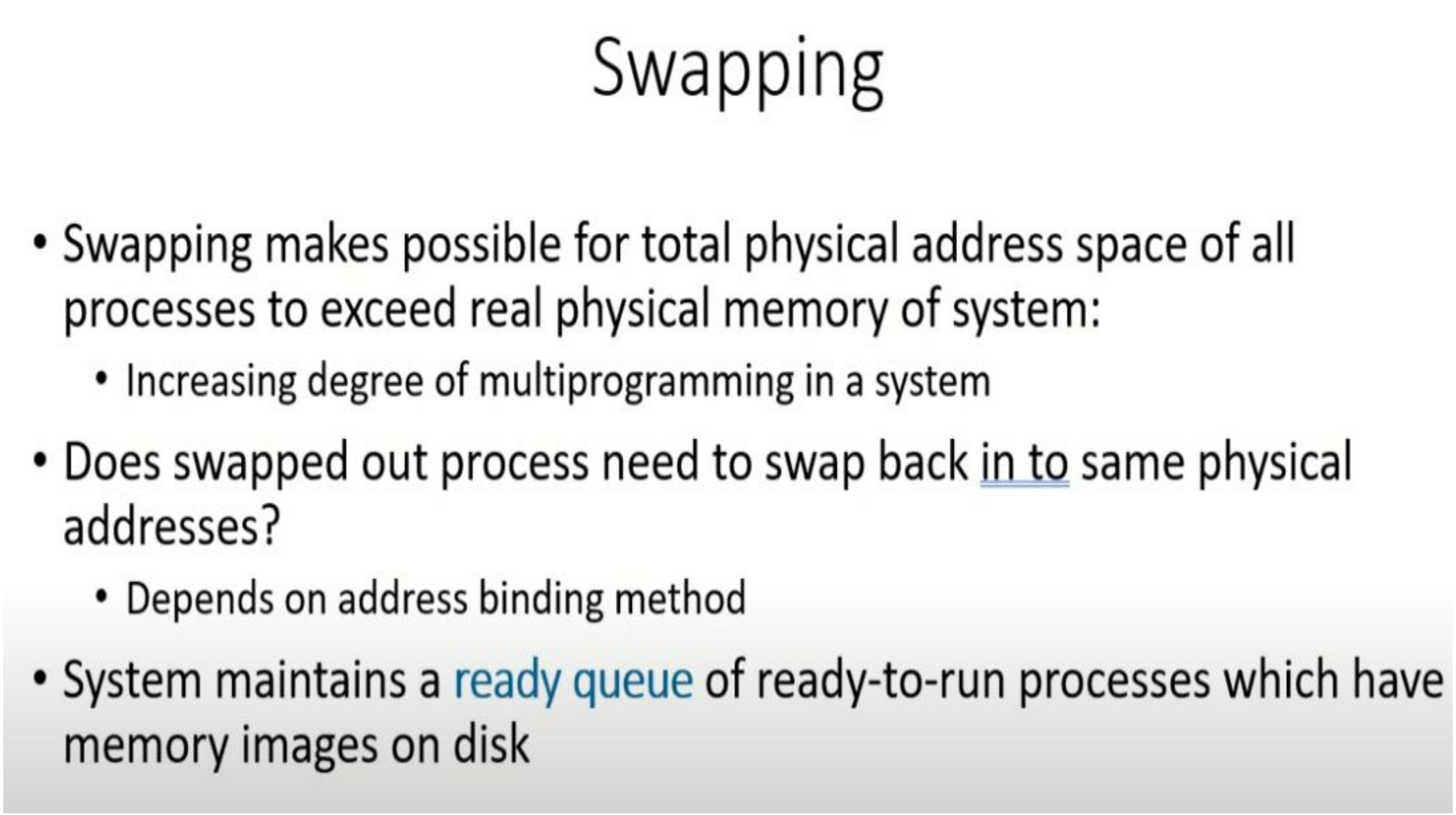
1.1)Standard Swapping:



Swapping

- A process can be swapped temporarily out of memory to a backing store, and then brought back into memory for continued execution
  - Improves main memory utilization
  - Swapped-out processes stored in swap space
- Backing store – fast disk:
  - Large enough to accommodate copies of all memory images for all users
  - Must provide direct access to these memory images
- Swap out, swap in (Roll out, roll in):
  - Swapping variant used for priority-based scheduling algorithms
  - Lower-priority process swapped out so higher-priority process can be loaded and executed

Standard swapping



Swapping

- Swapping makes possible for total physical address space of all processes to exceed real physical memory of system:
  - Increasing degree of multiprogramming in a system
- Does swapped out process need to swap back in to same physical addresses?
  - Depends on address binding method
- System maintains a ready queue of ready-to-run processes which have memory images on disk

Schematic View of standard Swapping

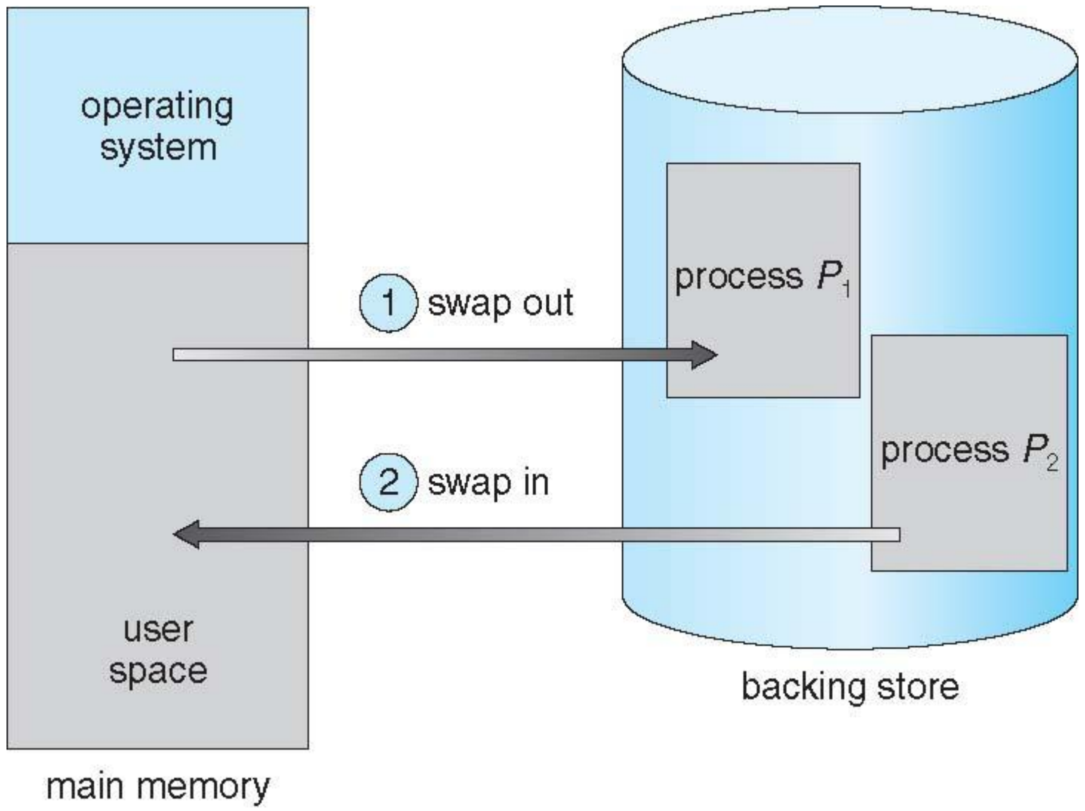


Figure 9.19 Standard swapping of two processes using a disk as a backing store.

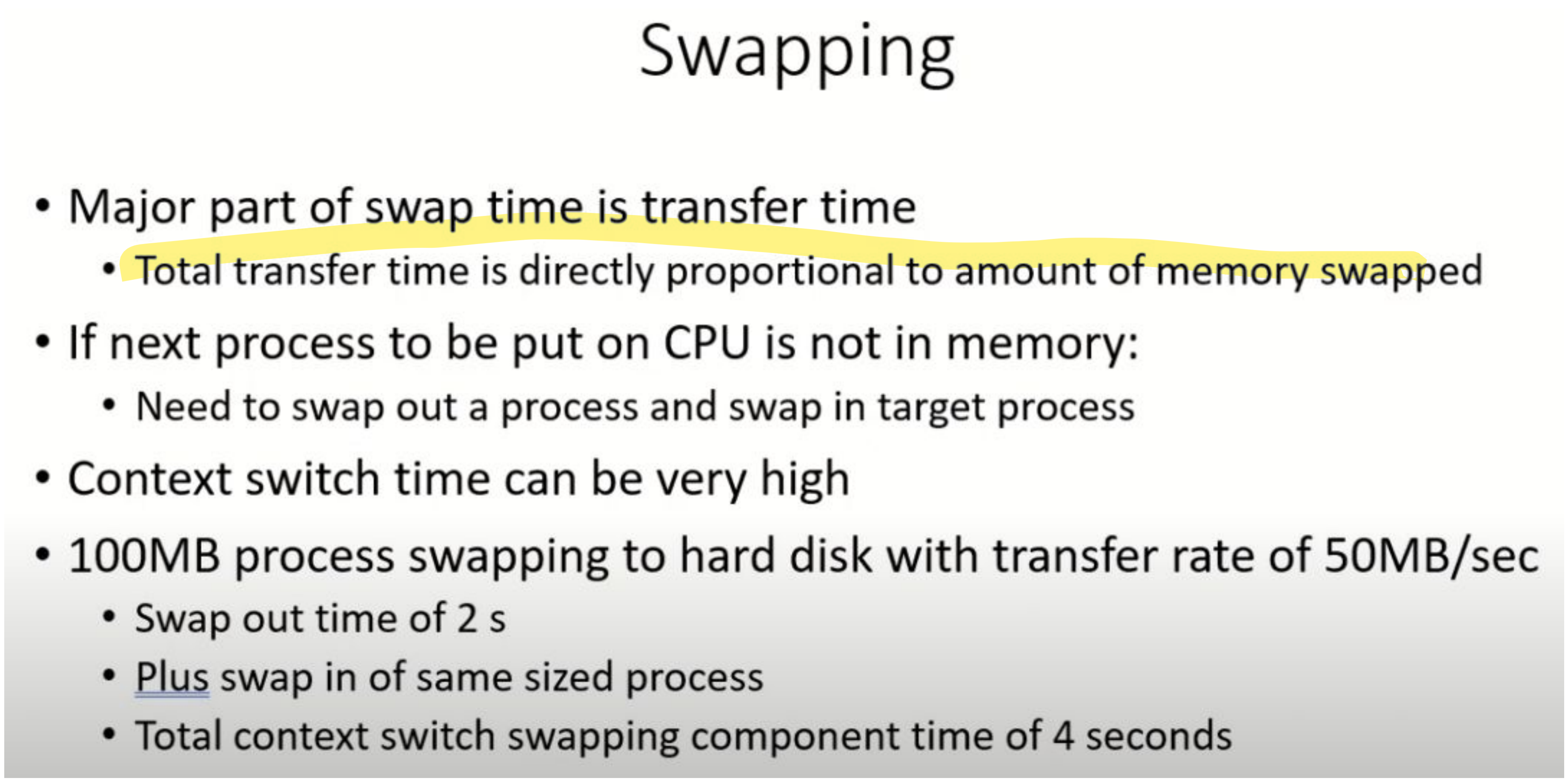
Standard Swapping (Cont.)

- Swapping is constrained by other factors as well.
- If we want to swap a process, we must be sure that it is **completely idle.**

- Assume that **the I/O operation for P1** is queued because the device is busy.

- If we were to swap out process P1 and swap in process P2, the I/**O operation might then attempt to use memory that now belongs to process P2.**

Standard swapping



Swapping

- Major part of swap time is transfer time
  - Total transfer time is directly proportional to amount of memory swapped
- If next process to be put on CPU is not in memory:
  - Need to swap out a process and swap in target process
- Context switch time can be very high
- 100MB process swapping to hard disk with transfer rate of 50MB/sec
  - Swap out time of 2 s
  - Plus swap in of same sized process
  - Total context switch swapping component time of 4 seconds

1.2)Swapping with Paging

- **Standard swapping was used in traditional UNIX systems,**but it is generally **no longer used i**n contemporary operating systems,

- because the amount of time required to move entire processes between memory and the backing store is prohibitive.
- Most systems,including Linux and Windows, now use a variation of swap ping in which pages of a process—rather than an entire process—can be swapped.
- the term swapping now generally refers to standard swapping, and paging refers to swapping with paging. **A page out** operation moves a page from memory to the backing store; the reverse process is known as **a page in.**
- Swapping with paging is illustrated in Figure 9.20 where a subset of pages for processes A and B are being paged-out and paged-in respectively.

Swapping with paging

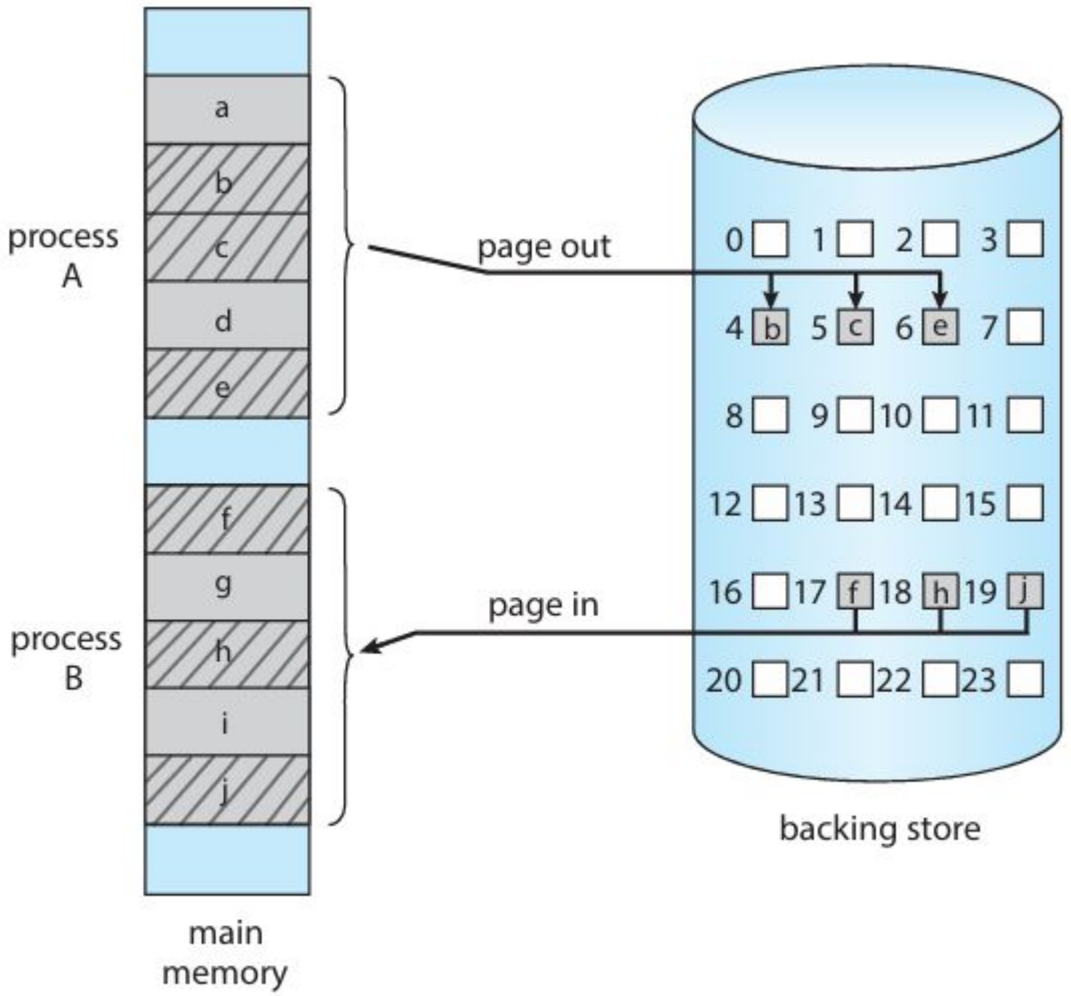


Figure 9.20 Swapping with paging.

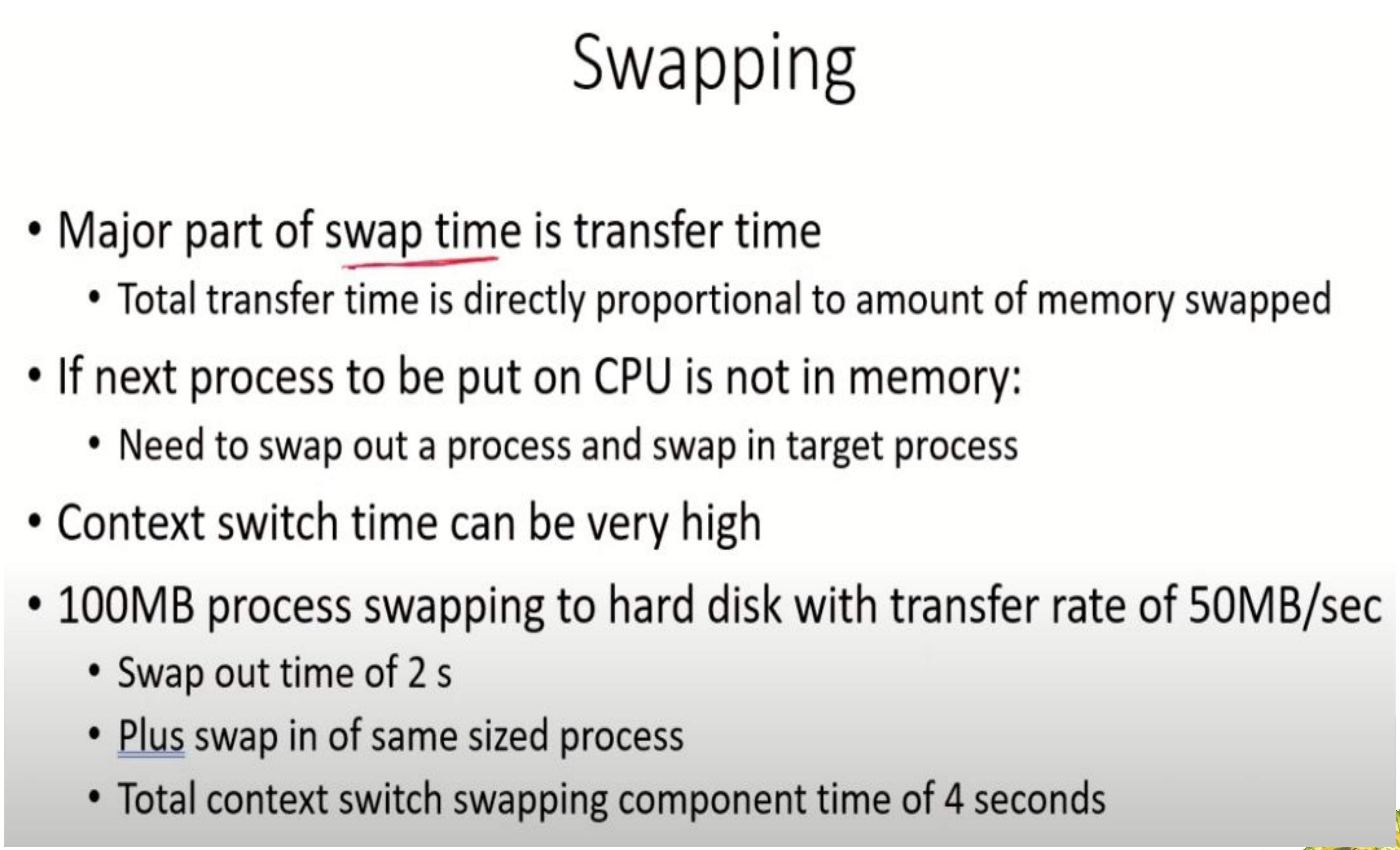
1.3) SWAPPING ON MOBILE SYSTEMS

Mobile devices generally use **flash memory** rather than more spacious hard disks as their persistent storage.

Instead of using swapping, when **free memory falls** below a certain threshold, **Apple's iOS a**sks applications to voluntarily relinquish allocated memory.

Android does not support swapping and adopts a **strategy similar to that used by iOS**. It may terminate a process if insufficient free memory is available.

Context Switch Time including Swapping



Swapping

- Major part of swap time is transfer time
  - Total transfer time is directly proportional to amount of memory swapped
- If next process to be put on CPU is not in memory:
  - Need to swap out a process and swap in target process
- Context switch time can be very high
- 100MB process swapping to hard disk with transfer rate of 50MB/sec
  - Swap out time of 2 s
  - Plus swap in of same sized process
  - Total context switch swapping component time of 4 seconds

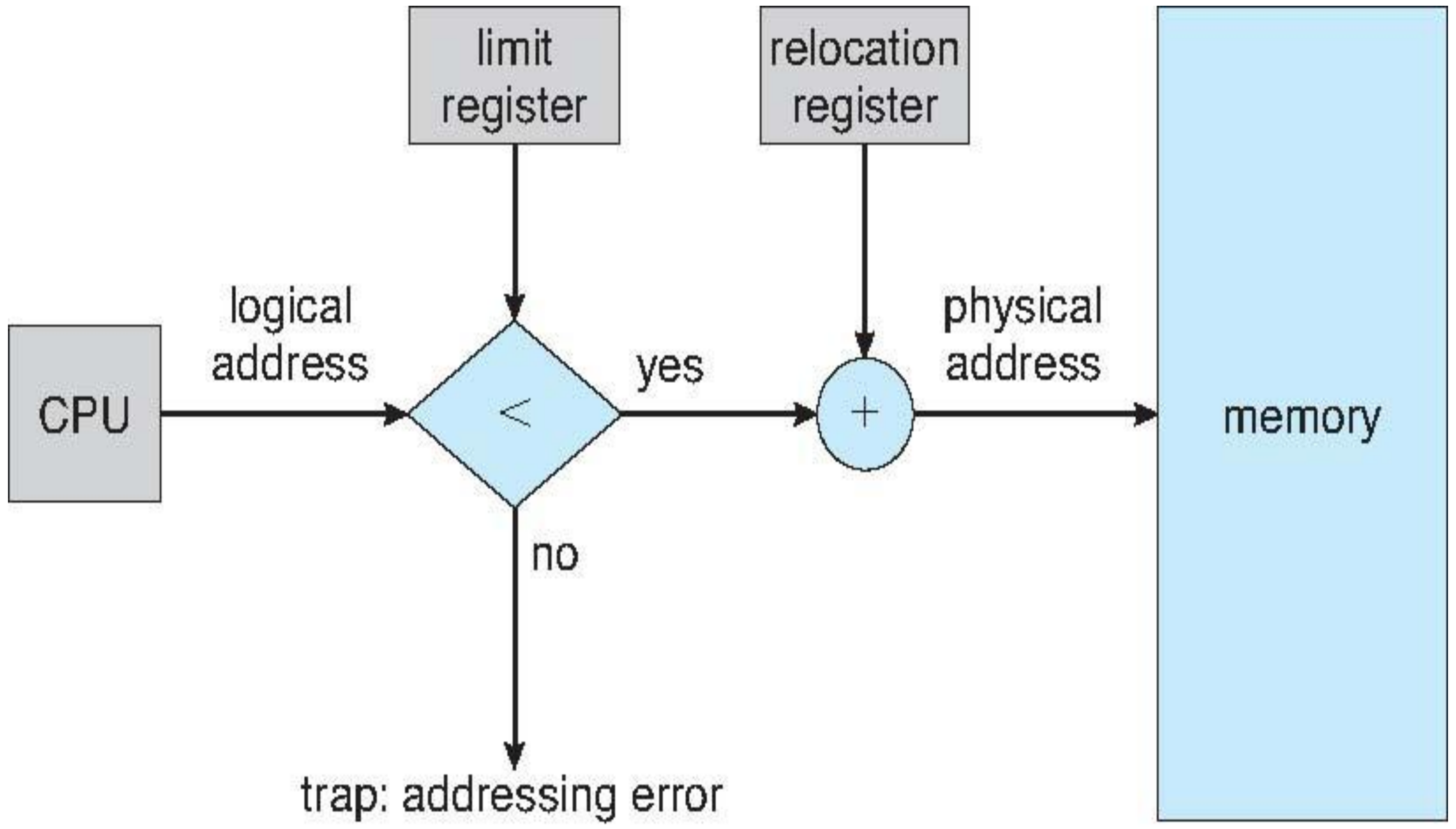
Contiguous Allocation

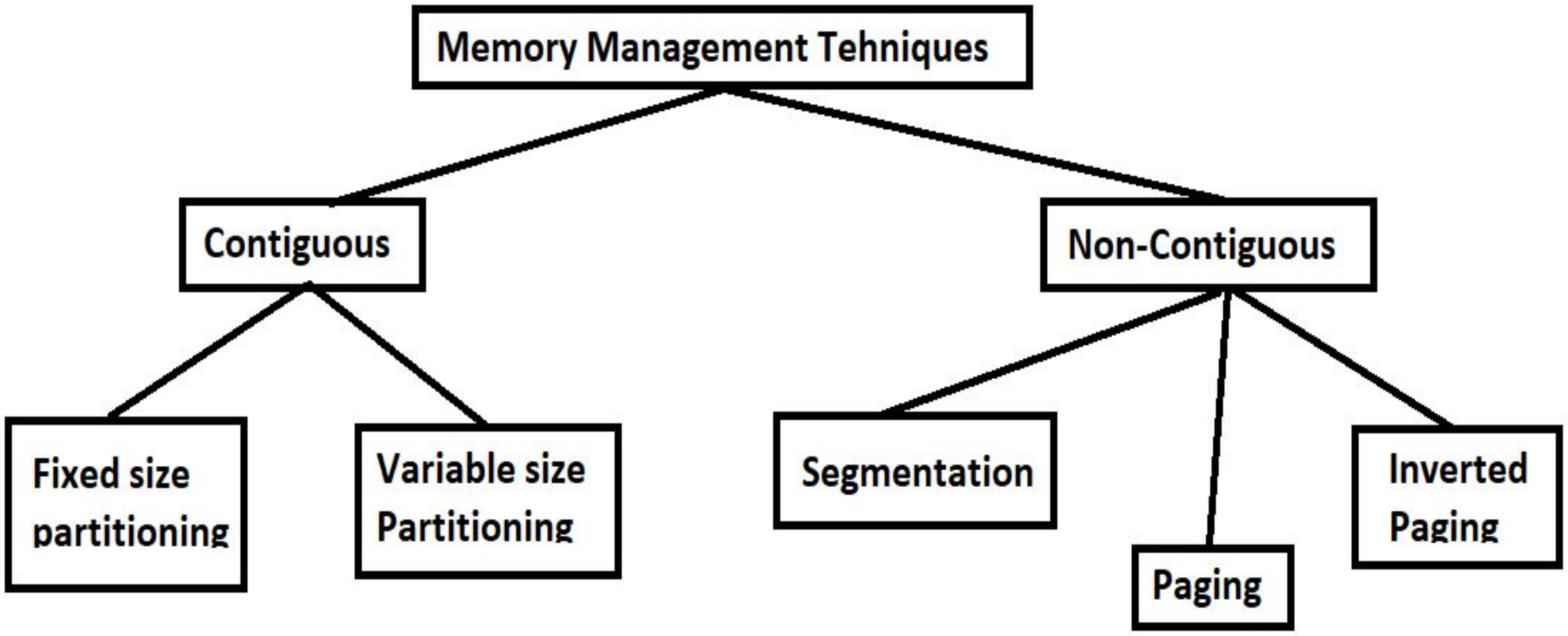
- Main memory must support both OS and user processes
- Limited resource, must allocate efficiently
- Contiguous allocation is one early method
- Main memory usually into two **partitions**:
  - Resident operating system, usually held in low memory
  - User processes then held in high memory
  - Each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

Contiguous Alloc.,(memory protection)

- 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*

Hardware Support for Relocation and Limit Registers

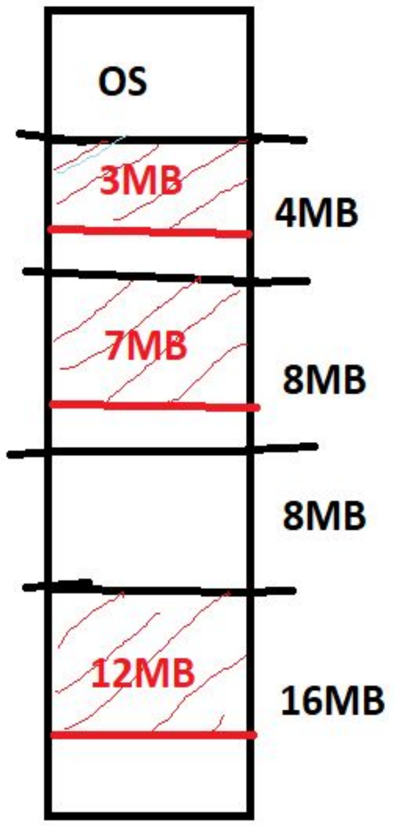



Memory Management Tehniques
Contiguous
Non-Contiguous
Fixed size partitioning
Variable size Partitioning
Segmentation
Paging
Inverted Paging

Fixed size partitioning

FIXED SIZE PARTITIONING

1) NUMBER OF PARTIONS IS FIXED
2) SIZE OF EACH PARTTION MAY/MAY NT BE SAME
3) INTERNAL FRAGMENTATION
4)NUMBER OF PROCESSES ARE FIXED.

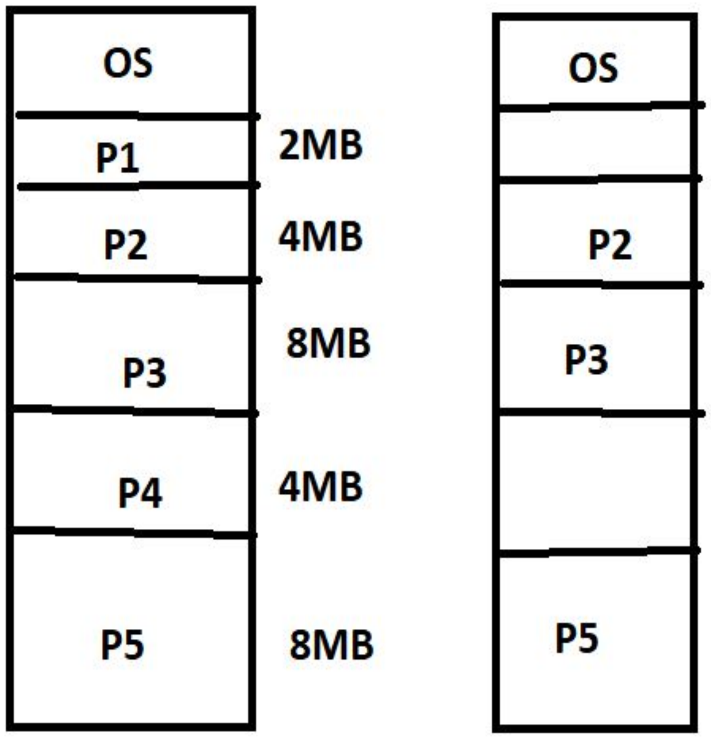


VARIABLE SIZE PARTITION

variable/dynamic partitioning

1) No internal fragmentation
2) No limitation on no. of processes
3) No limitations on process size
4) suffers from exteral fragmentaion because of insufficient size of holes to fit a whole process.

cosider, processes p1=2mb, p2=4mb, p3=8mb,p4=4mb, p5=8mb



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)

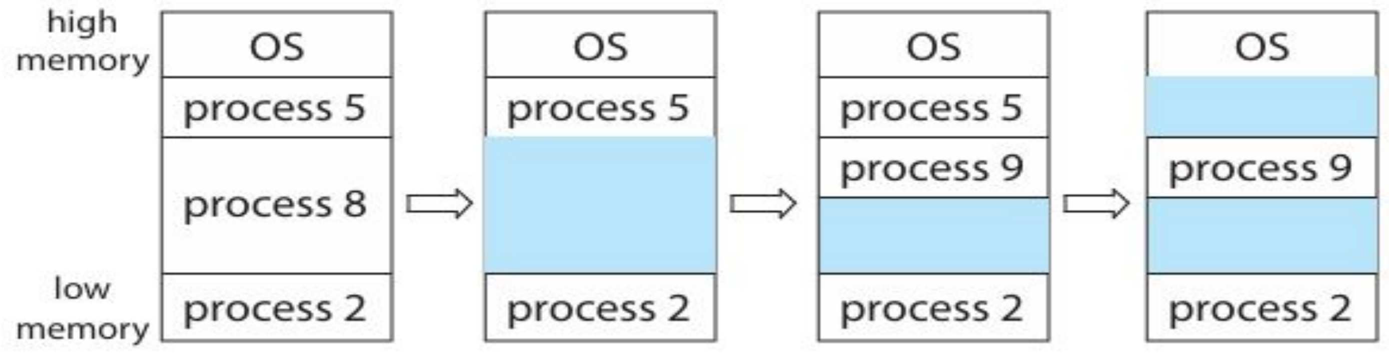


Figure 9.7 Variable partition.

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

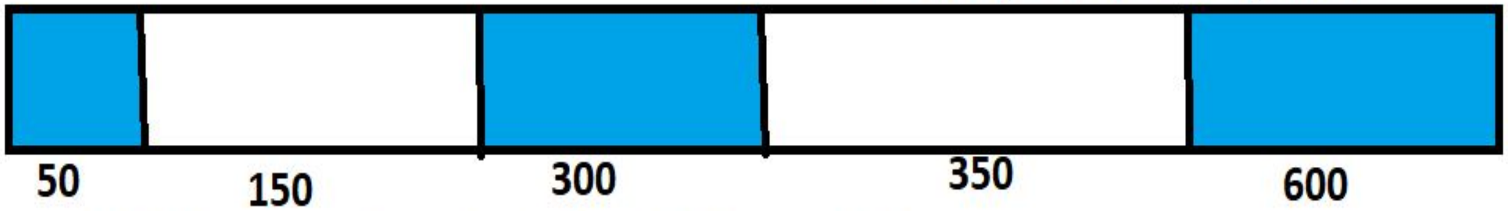
First-fit and best-fit better than worst-fit in terms of speed and storage utilization

EXAMPLE:assignment (white being free spaces)

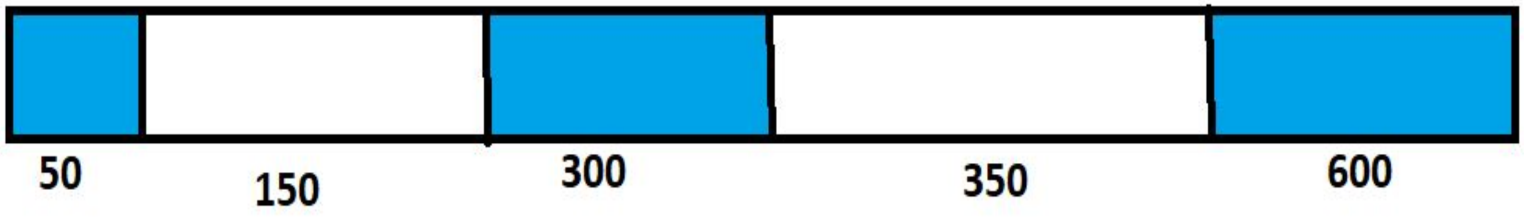
first fit(allocate the first hole thats big enough)

consider,

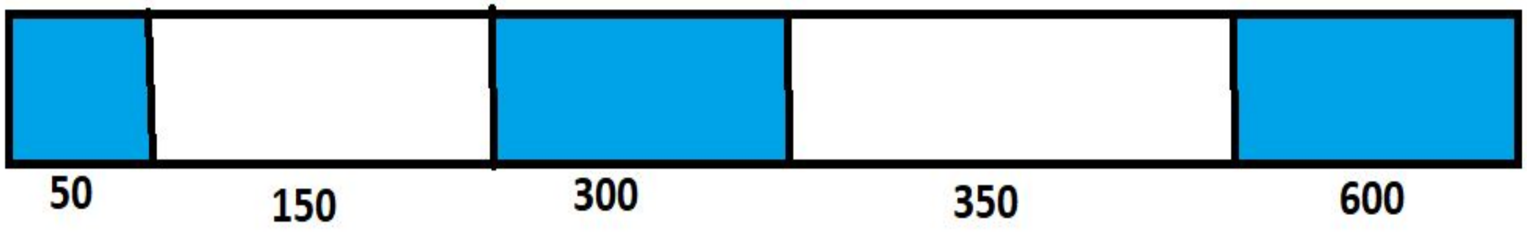
P1-300,P2-25,P3-125,P4-50



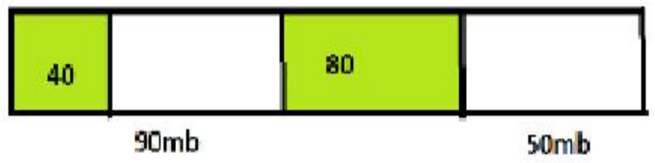
Best fit(allocate smallest hole that is big enough)



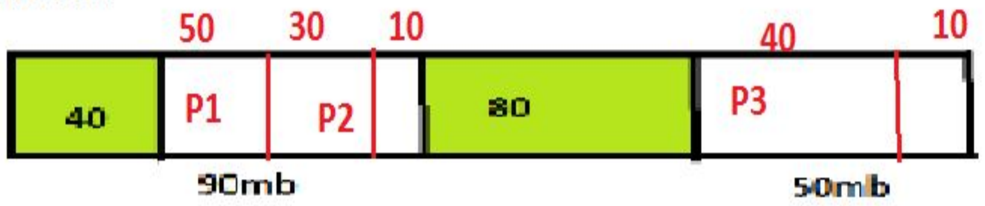
worst fit(allocate the largest hole)



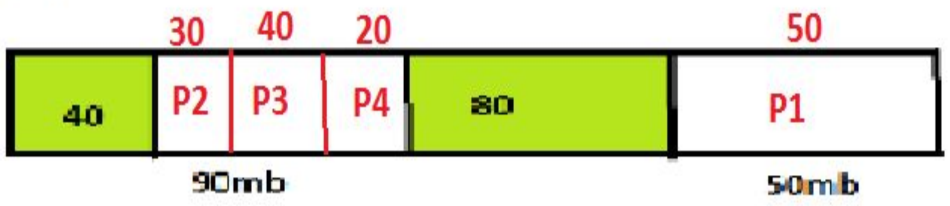
2]Consider processes x1=50mb, x2=30mb, x3=40mb,x4=20mb. Allocate these processes into the holes available using First fit, best fit and worst fit.



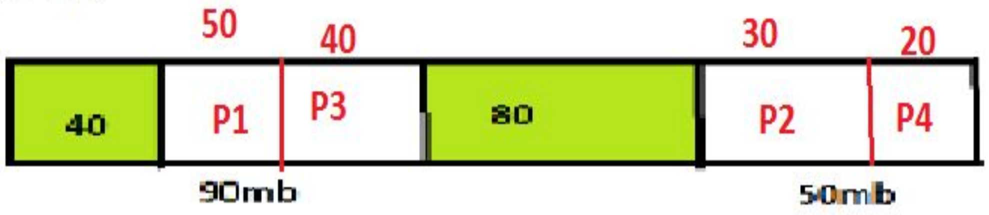
FIRST FIT:



BEST FIT:



WORST FIT:



First fit:
p1(50mb) is put in 90 mb partition
p2(30mb) is put in (90-50=40mb) partition
p3(40mb) is put in 50 mb partition
p4(20mb) must wait

Best fit:
p1(50mb) is put in 50mb partition
p2(30mb) is put in 90mb partition
p3(40mb) is put in (90-30=60mb) partition
p4(20mb) is put in (90-70=20mb) partition

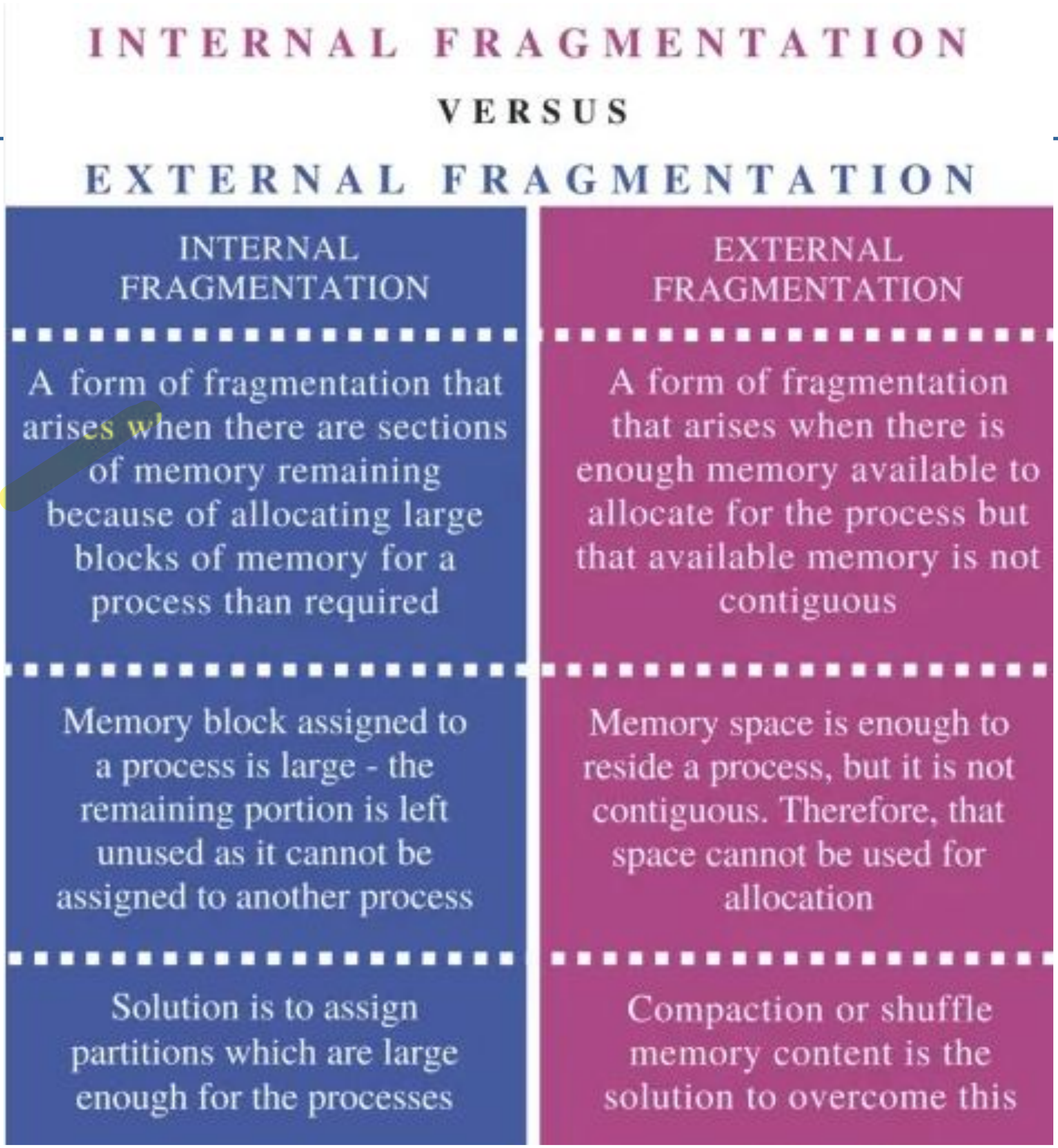
Worst fit:
p1(50mb) is put in 90mb partition
p2(30mb) is put in 50mb partition
p3(40mb) is put in (90-50=40mb) partition
p4(20mb) is put in (50-30=20mb) partition

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
- First fit analysis reveals that given *N* blocks allocated, 0.5 *N* blocks lost to fragmentation
  - 1/3 may be unusable -> **50-percent rule**

Fragmentation (Cont.)

- 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**
  - Another possible solution to the external-fragmentation problem is to permit the **logical address space of the processes to be noncontiguous**, thus allowing a process to be allocated physical memory wherever such memory is available.
  - Two complementary techniques achieve this solution: **segmentation and paging** which are discussed next.



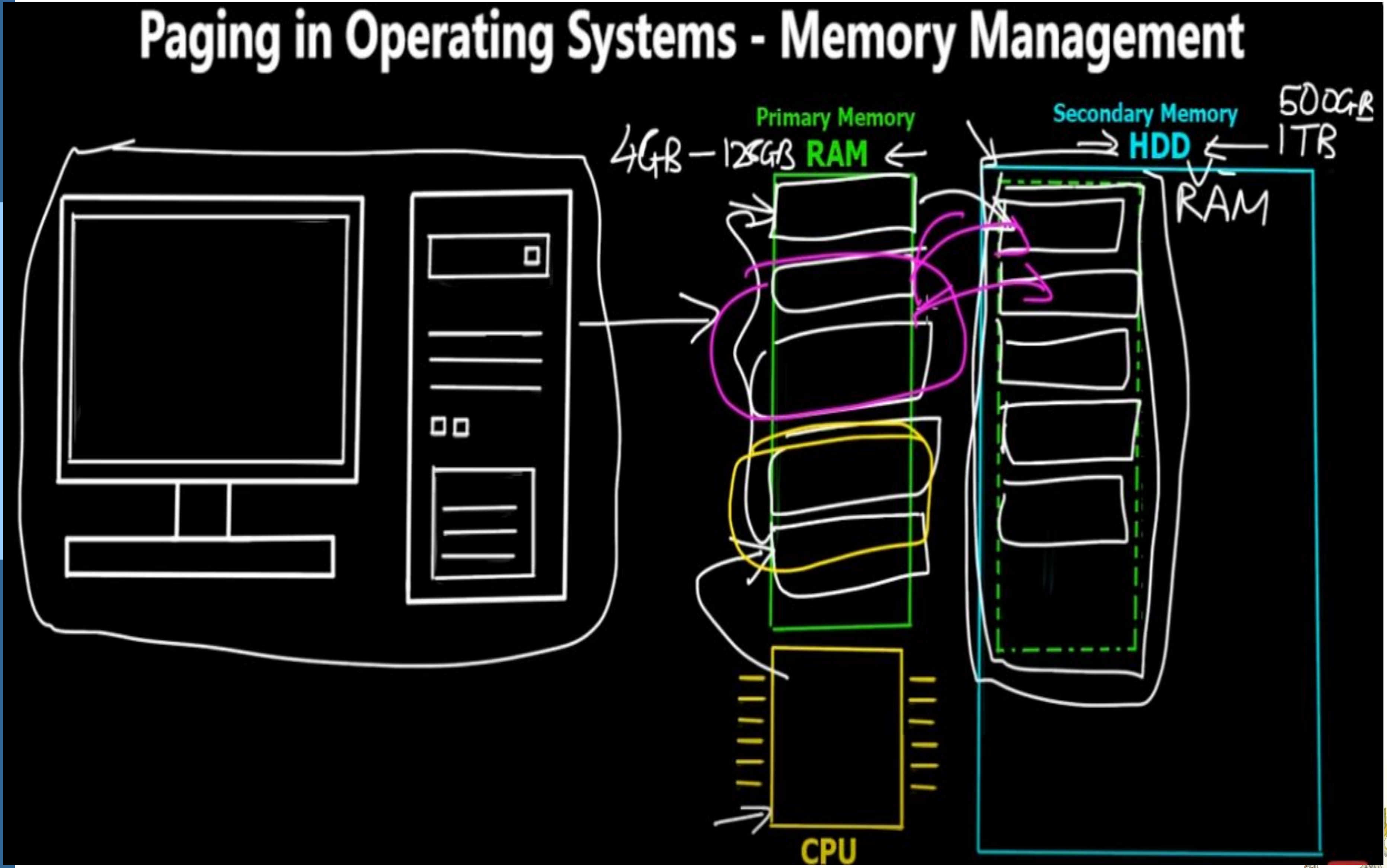
INTERNAL FRAGMENTATION VERSUS EXTERNAL FRAGMENTATION

INTERNAL FRAGMENTATION	EXTERNAL FRAGMENTATION
A form of fragmentation that arises when there are sections of memory remaining because of allocating large blocks of memory for a process than required	A form of fragmentation that arises when there is enough memory available to allocate for the process but that available memory is not contiguous
Memory block assigned to a process is large - the remaining portion is left unused as it cannot be assigned to another process	Memory space is enough to reside a process, but it is not contiguous. Therefore, that space cannot be used for allocation
Solution is to assign partitions which are large enough for the processes	Compaction or shuffle memory content is the solution to overcome this

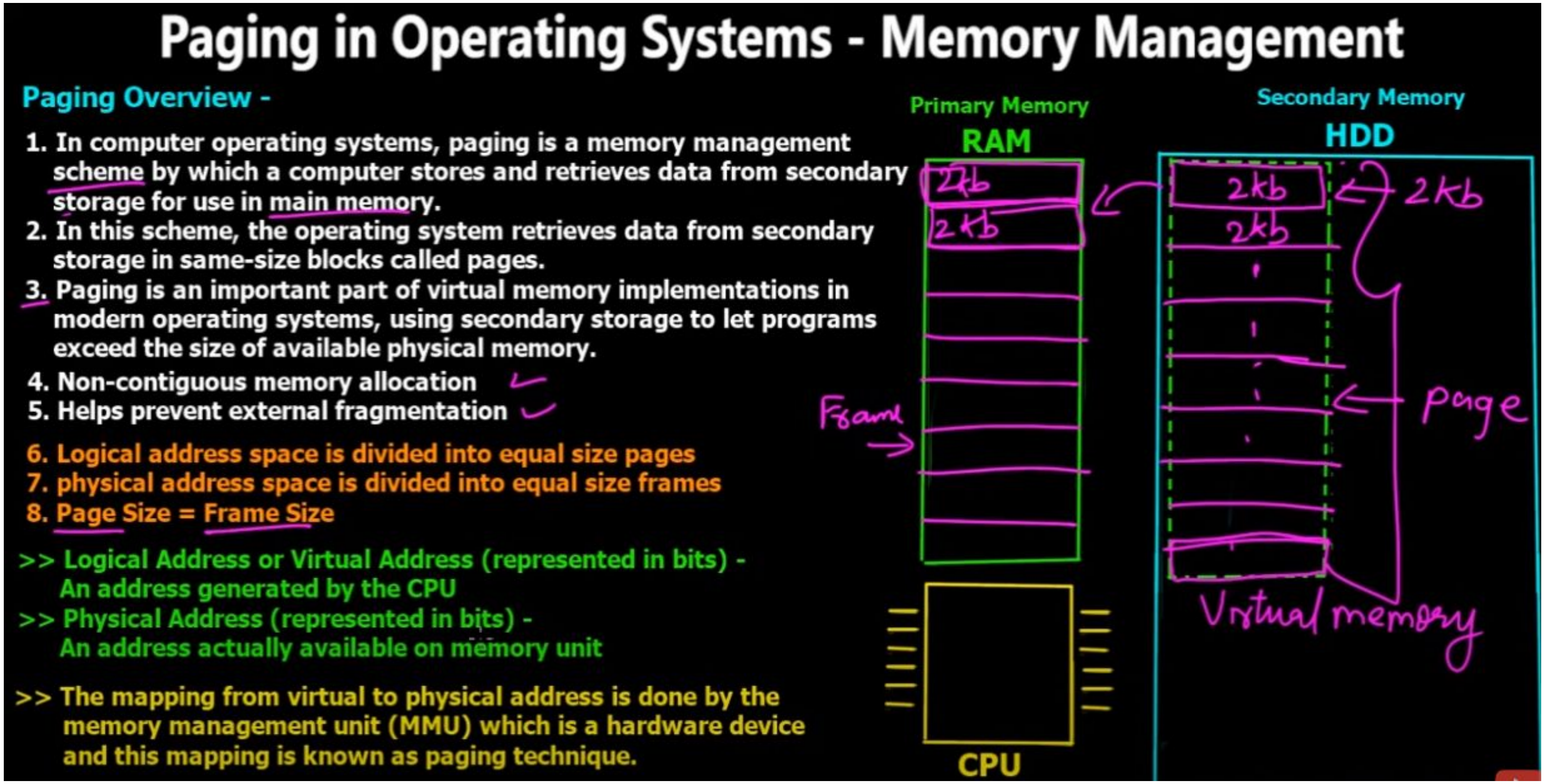
Paging

- Physical address space of a process can be noncontiguous; process is allocated physical memory whenever the latter is available
  - Avoids external fragmentation
  - Avoids problem of varying sized memory chunks
- Divide physical memory into fixed-sized blocks called **frames**
- 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
- Backing store likewise split into pages
- Still have Internal fragmentation

Paging (contd..,)



Paging (contd..,)



Paging in Operating Systems - Memory Management

Paging Overview -

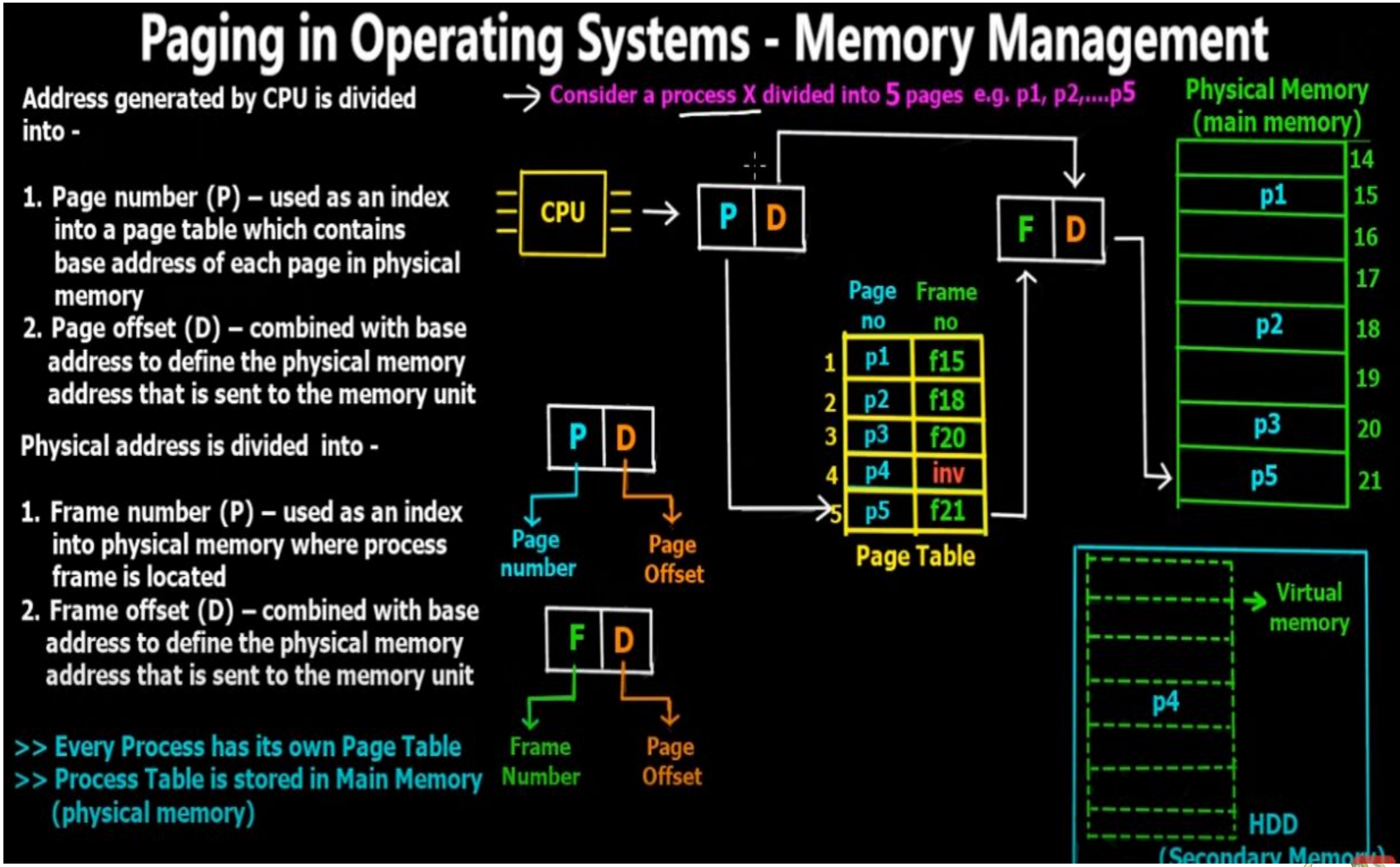
1. In computer operating systems, paging is a memory management scheme by which a computer stores and retrieves data from secondary storage for use in main memory.
2. In this scheme, the operating system retrieves data from secondary storage in same-size blocks called pages.
3. Paging is an important part of virtual memory implementations in modern operating systems, using secondary storage to let programs exceed the size of available physical memory.
4. Non-contiguous memory allocation
5. Helps prevent external fragmentation
6. Logical address space is divided into equal size pages
7. physical address space is divided into equal size frames
8. Page Size = Frame Size

>> Logical Address or Virtual Address (represented in bits) - An address generated by the CPU

>> Physical Address (represented in bits) - An address actually available on memory unit

>> The mapping from virtual to physical address is done by the memory management unit (MMU) which is a hardware device and this mapping is known as paging technique.

Paging (contd..,)



Paging in Operating Systems - Memory Management

Address generated by CPU is divided into -

1. Page number (P) – used as an index into a page table which contains base address of each page in physical memory
2. Page offset (D) – combined with base address to define the physical memory address that is sent to the memory unit

Physical address is divided into -

1. Frame number (P) – used as an index into physical memory where process frame is located
2. Frame offset (D) – combined with base address to define the physical memory address that is sent to the memory unit

>> Every Process has its own Page Table
>> Process Table is stored in Main Memory (physical memory)

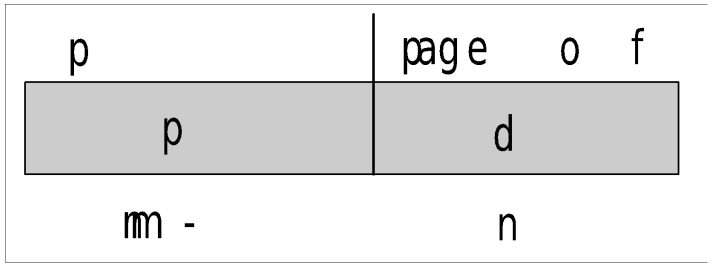
→ Consider a process X divided into 5 pages e.g. p1, p2,....p5

Page Table

	Page no	Frame no
1	p1	f15
2	p2	f18
3	p3	f20
4	p4	inv
5	p5	f21

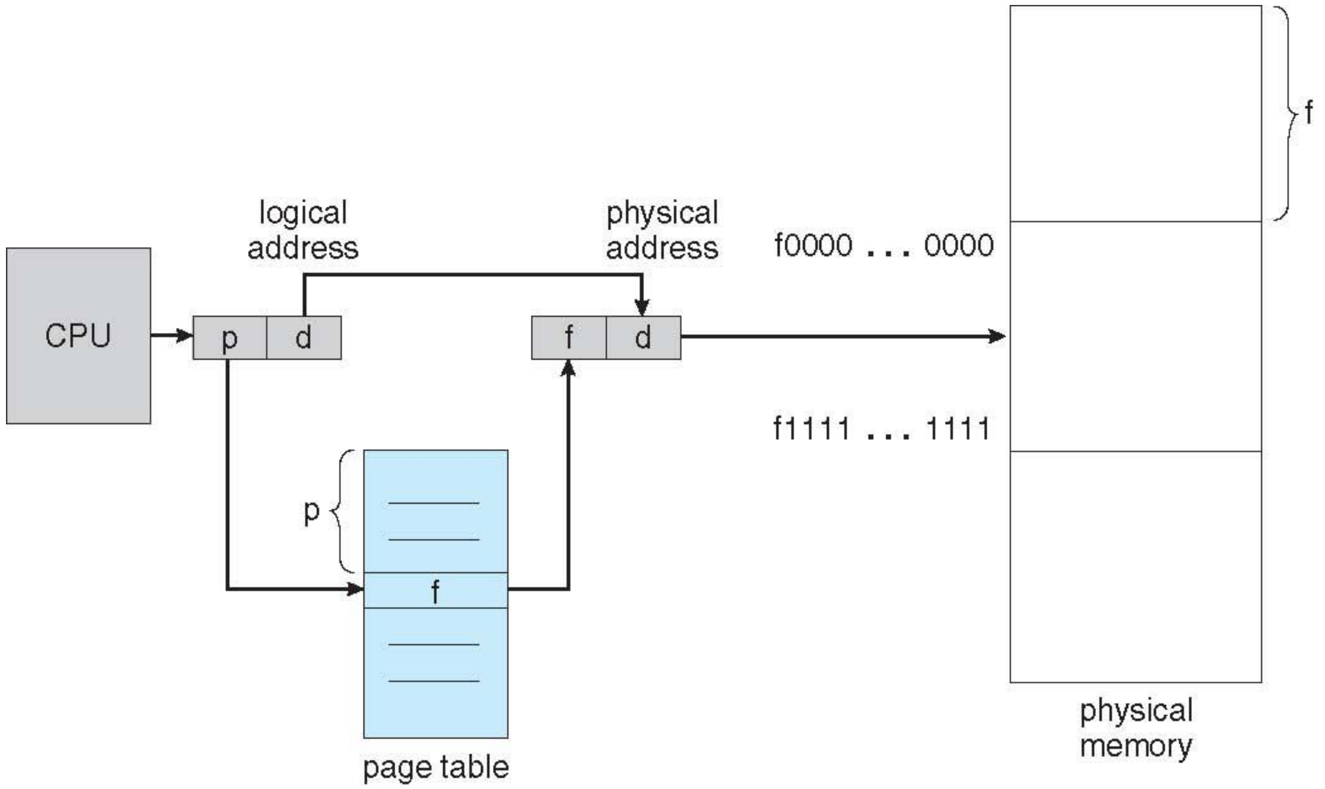
Address Translation Scheme

- 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

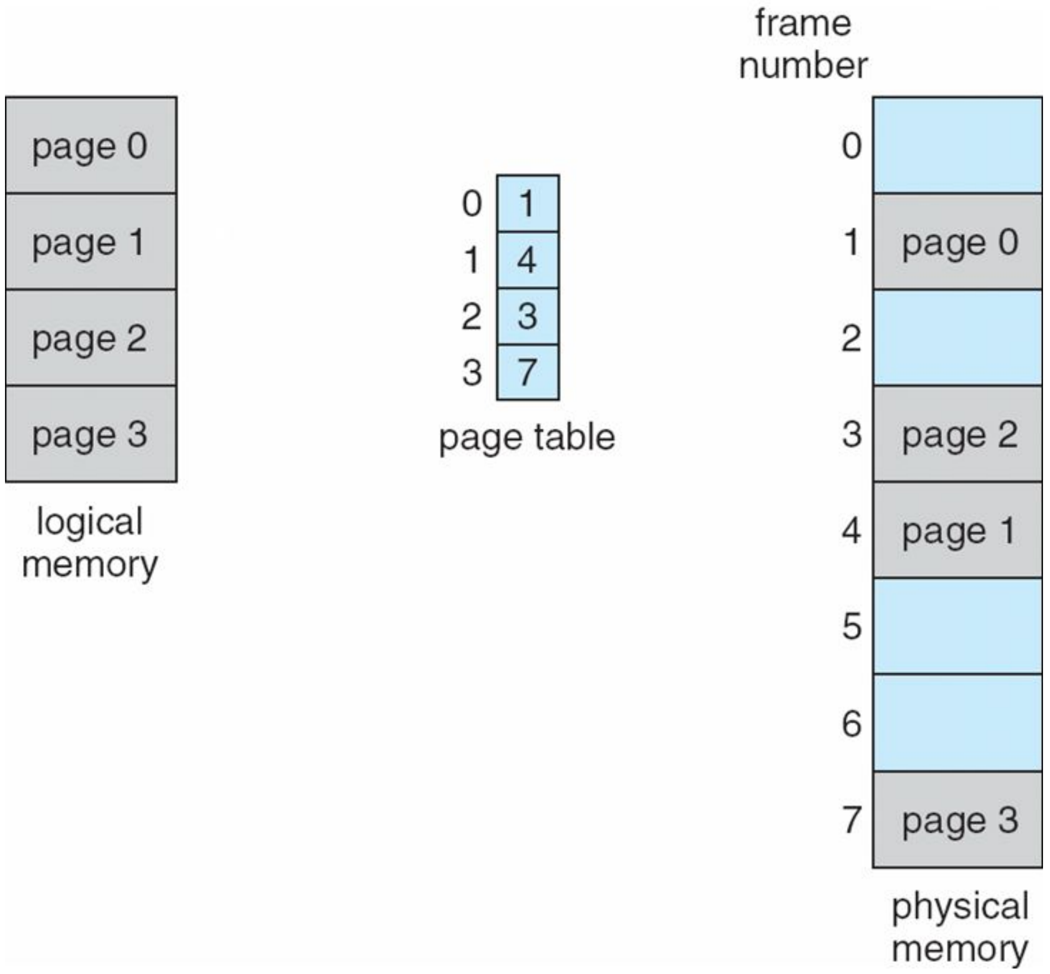


  - For given logical address space 2^m and page size 2^n

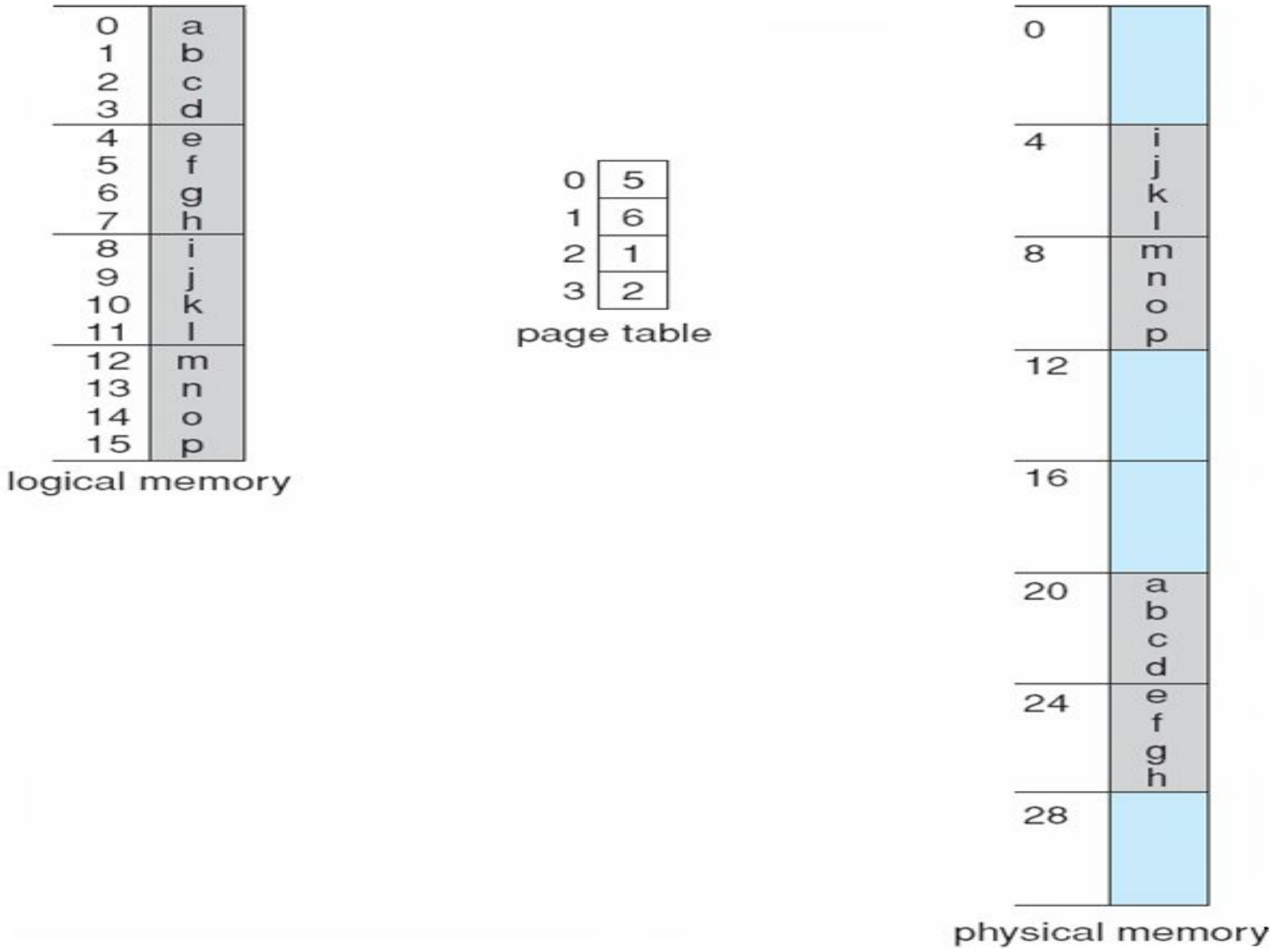
Paging Hardware



Paging Model of Logical and Physical Memory



Paging Example

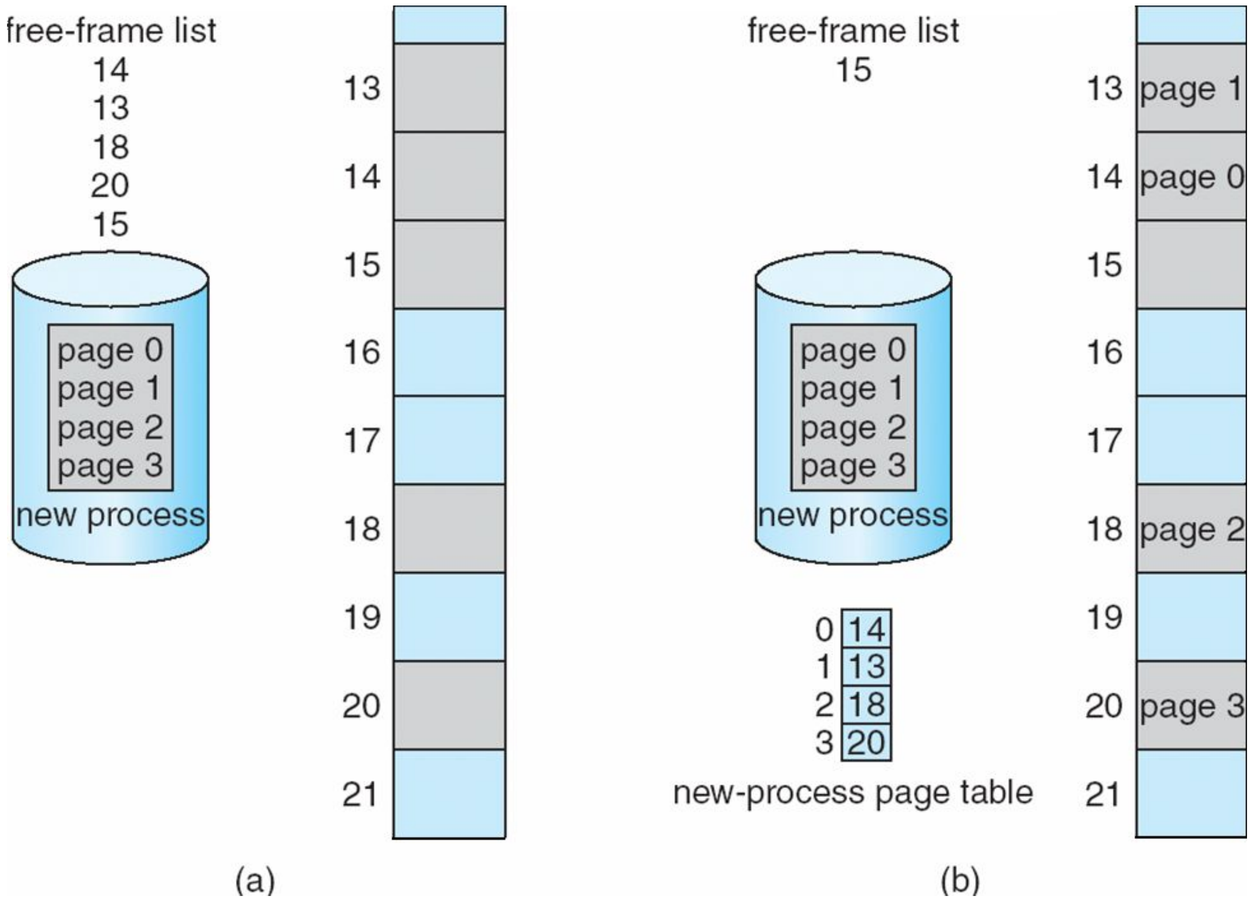


n=2 and m=4   32-byte memory and 4-byte pages

Logical address to physical adress

- Here, in the logical address, n=2 and m=4.
- Using a page size of 4 bytes and a physical memory of 32 bytes (8pages), we show how the programmer's view of memory can be mapped into physical memory. Logical address 0 is page 0, offset 0.
- **Indexing into the page table, we find that page 0 is in frame 5.**
- **Thus, logical address 0 maps to physical address 20[=(5×4)+ 0].**
- Logical address 3 (page0,offset3) maps to physical address 23[= (5×4)+3].
- Logical address 4 is page1,offset0; according to the page table, page 1 is mapped to frame 6.
- **When logical address is (page 3, offset 2) what is the physical address? (refer previous diagram)**

Free Frames(Fig: 9.11)



Before allocation

After allocation

Free frames

- When a process arrives in the system to be executed, its size, expressed in pages, is examined.
- Each page of the process need some 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.
- The next page is loaded into another frame, its frame number is put into the page table, and soon (Figure 9.11).

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
- **Although storing the page table in main memory can yield faster context switches, it may also result in slower memory access times**
- 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 (Cont.)

- Some TLBs store **address-space identifiers** (**ASIDs**) in each TLB entry – uniquely identifies each process to provide address-space protection for that process
  - Otherwise need to flush at every context switch
- TLBs typically small (64 to 1,024 entries)
- On a TLB miss, value is loaded into the TLB for faster access next time
  - Replacement policies must be considered
  - Some entries can be **wired down** for permanent fast access

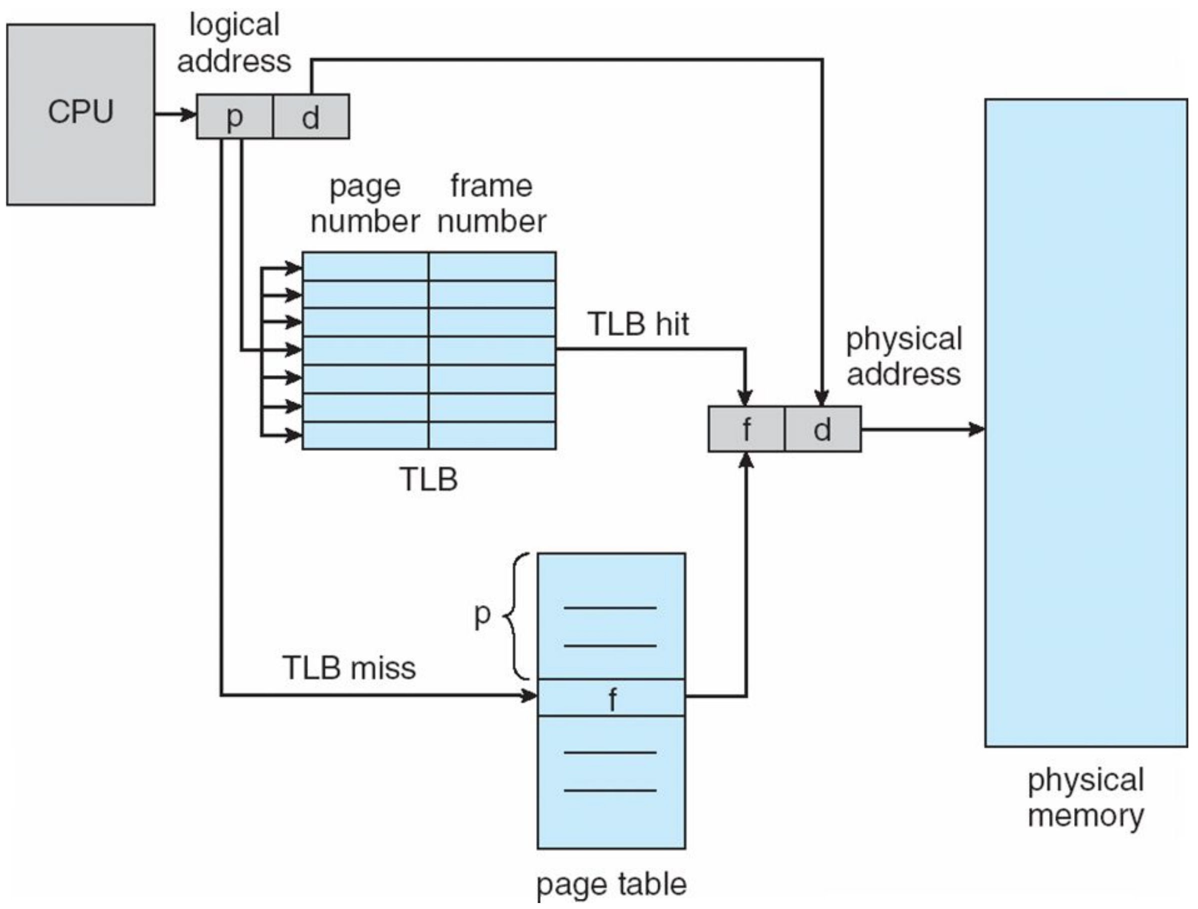
Associative Memory

- Associative memory – parallel search

Page #	Frame #

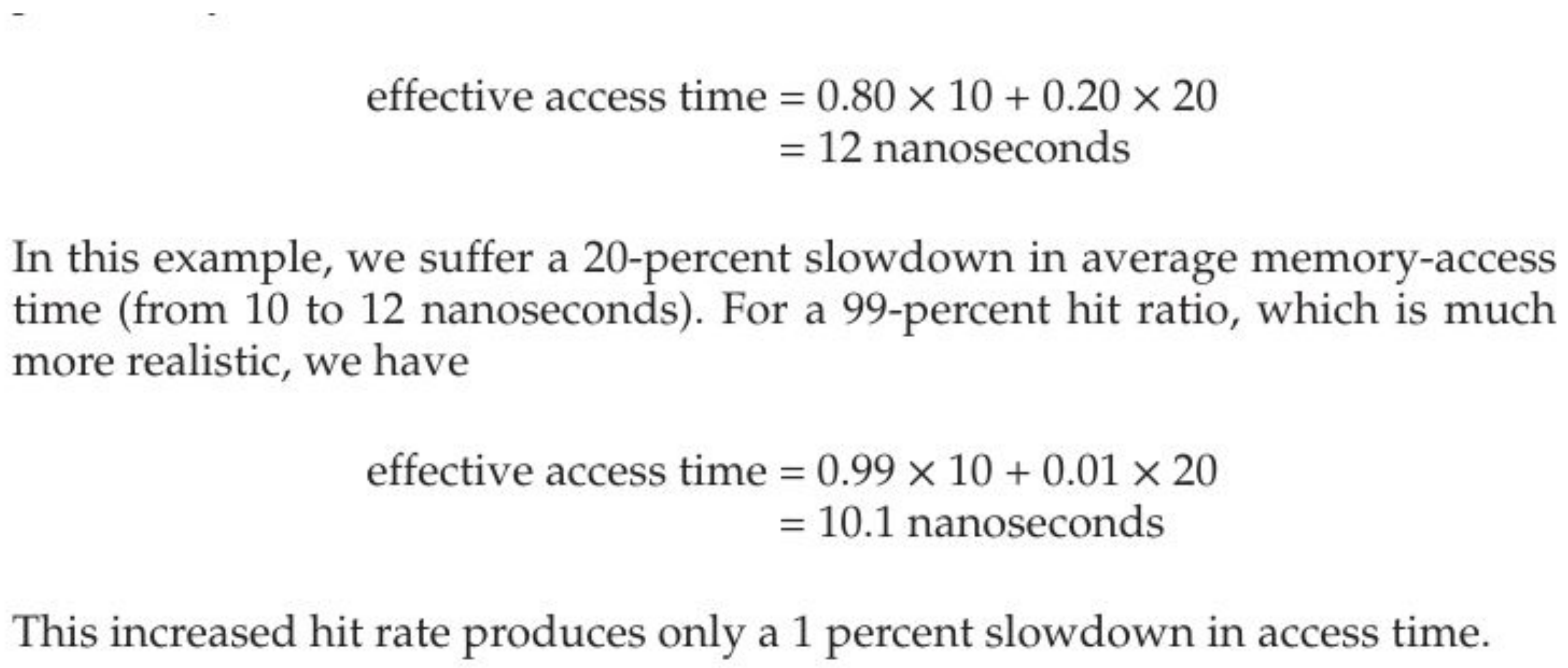
- Address translation (p, d)
  - If p is in associative register, get frame # out
  - Otherwise get frame # from page table in memory

Paging Hardware With TLB



Effective access time

- An 80-percent hit ratio, for example, means that we find the desired page number in the TLB 80 percent of the time.
- If it takes 10 nanoseconds to access memory, then a mapped-memory access takes 10 nanoseconds when the page number is in the TLB.
- If we fail to find the page number in the TLB then we must first access memory for the page table and frame number (10 nanoseconds) and then access the desired byte in memory (10 nanoseconds), for a total of 20 nano seconds.



$$\begin{aligned}\text{effective access time} &= 0.80 \times 10 + 0.20 \times 20 \\ &= 12 \text{ nanoseconds}\end{aligned}$$

In this example, we suffer a 20-percent slowdown in average memory-access time (from 10 to 12 nanoseconds). For a 99-percent hit ratio, which is much more realistic, we have

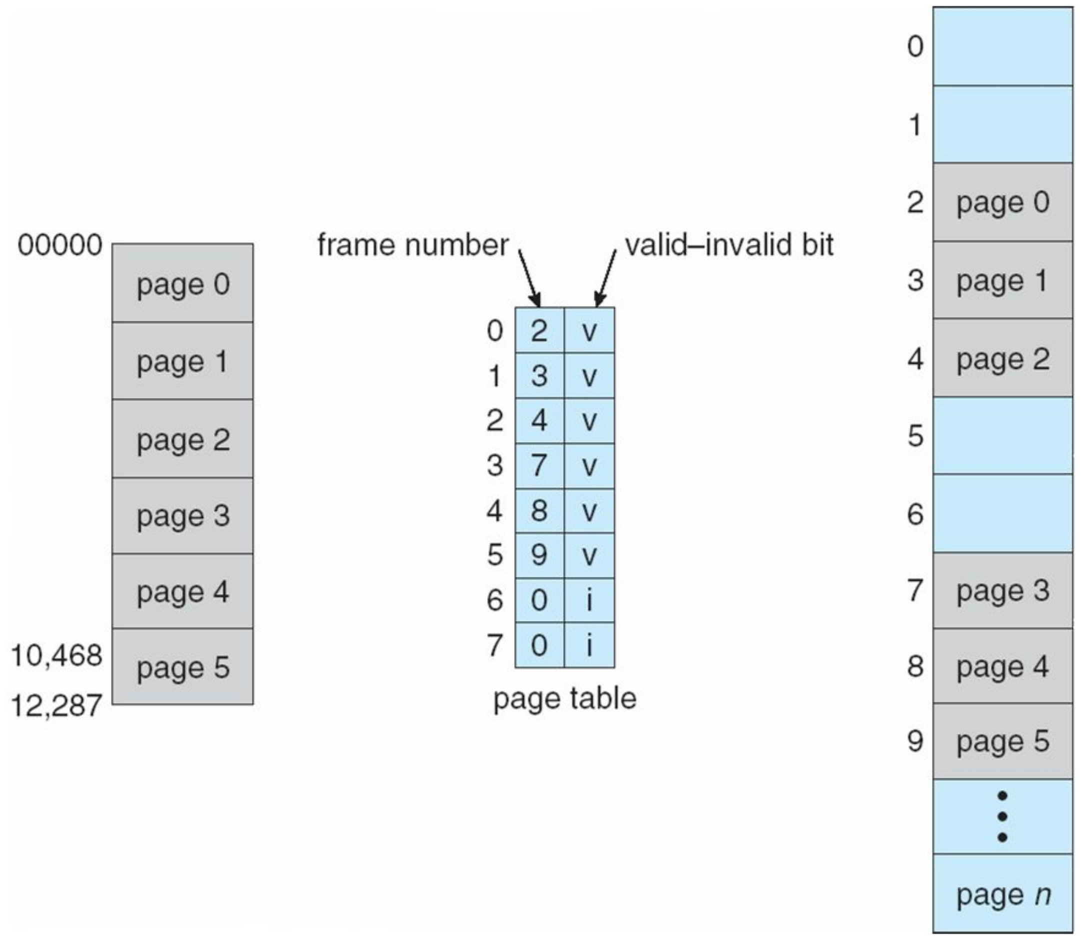
$$\begin{aligned}\text{effective access time} &= 0.99 \times 10 + 0.01 \times 20 \\ &= 10.1 \text{ nanoseconds}\end{aligned}$$

This increased hit rate produces only a 1 percent slowdown in access time.

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

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



Shared Pages

- An advantage of paging is the possibility of sharing common code, a consideration that is particularly important in an environment with multiple processes.
- Consider the standard C library, which provides a portion of the system call interface for many versions of UNIX and Linux.
- On a typical Linux system, most user processes require the standard C library libc.
- Eg:40 user processes, and the libc library is 2 MB, this would require 80 MB of memory. If the code is reentrant code, however, it can be shared, as shown in Figure 9.14.
- Here, we see three processes sharing the pages for the standard C library libc. (Although the figure shows the libc library occupying four pages, in reality, it would occupy more.)
- Reentrant code is non-self-modifying code: it never changes during execution.
- Thus, two or more processes can execute the same code at the same time. Each process has its own copy of registers and data storage to hold the data for the process's execution.

Shared pages (contd..,)

- Only one copy of the standard C library need be kept in physical memory, and the page table for each user process maps onto the same physical copy of libc.
- Thus, to support 40 processes, we need only one copy of the library, and the total space now required is 2 MB instead of 80 MB—a significant saving!

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

Shared Pages Example

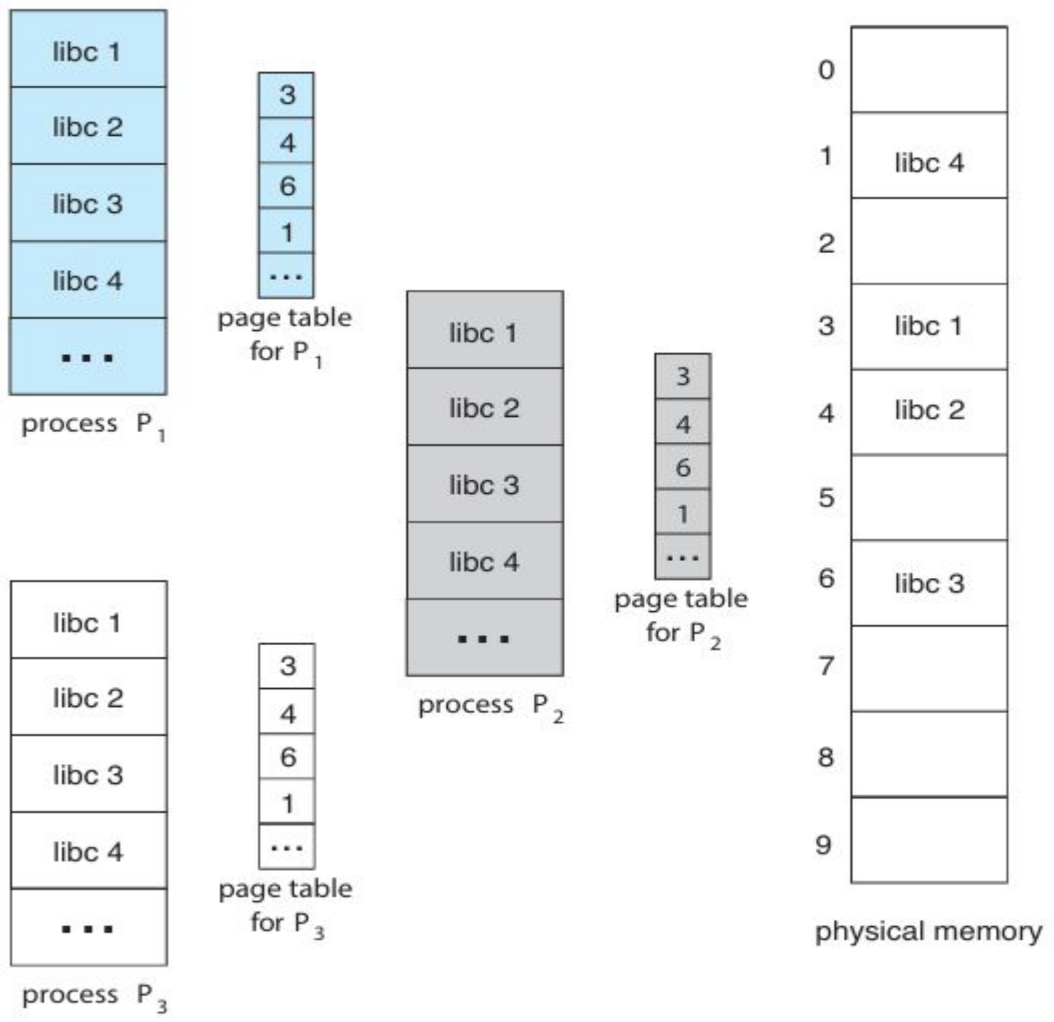
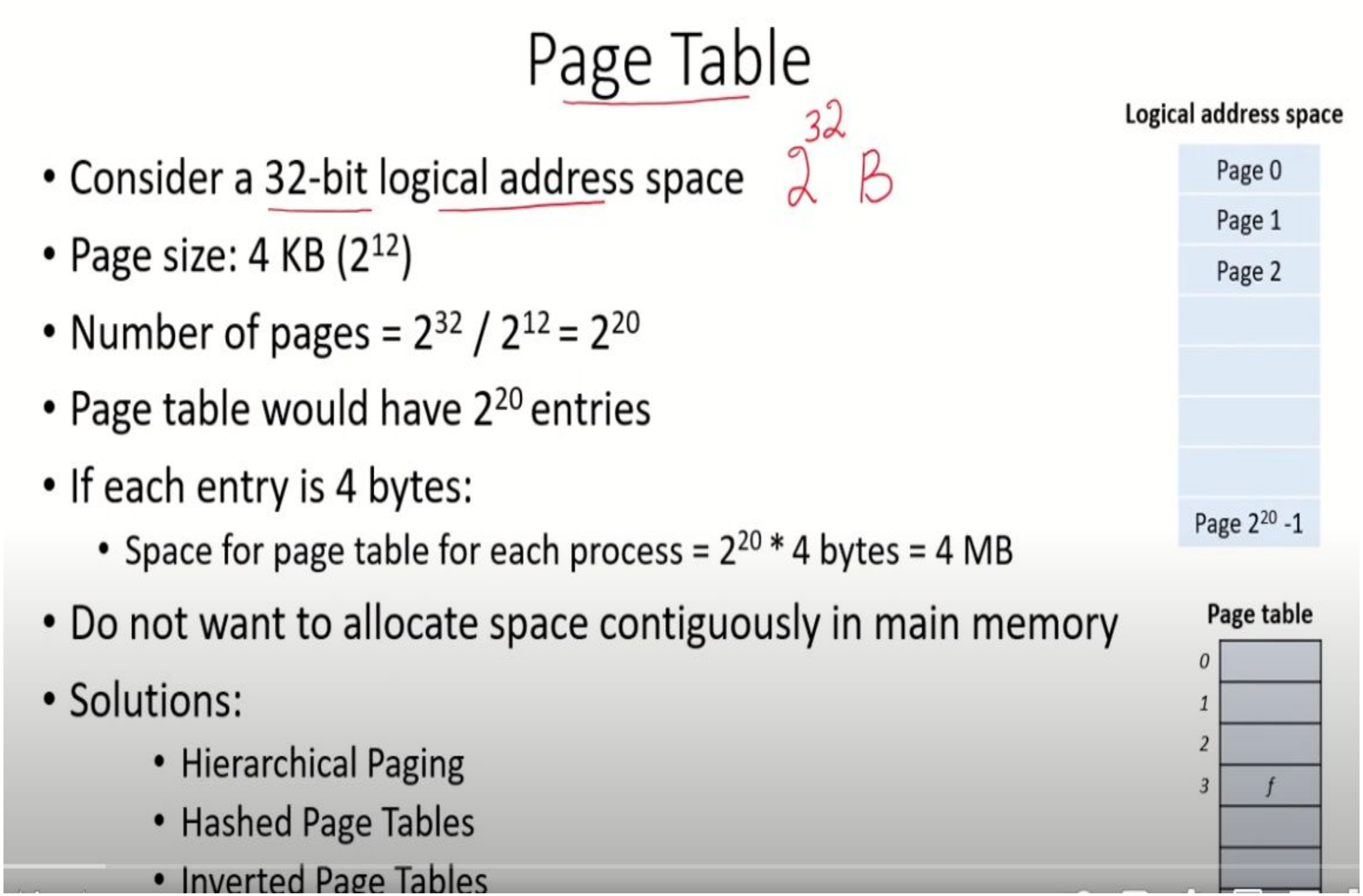


Figure 9.14 Sharing of standard C library in a paging environment.

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 ($2^{32} / 2^{12}$)
  - If each entry is 4 bytes -> 4 MB of physical address space / memory for page table alone
    - 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**



Page Table

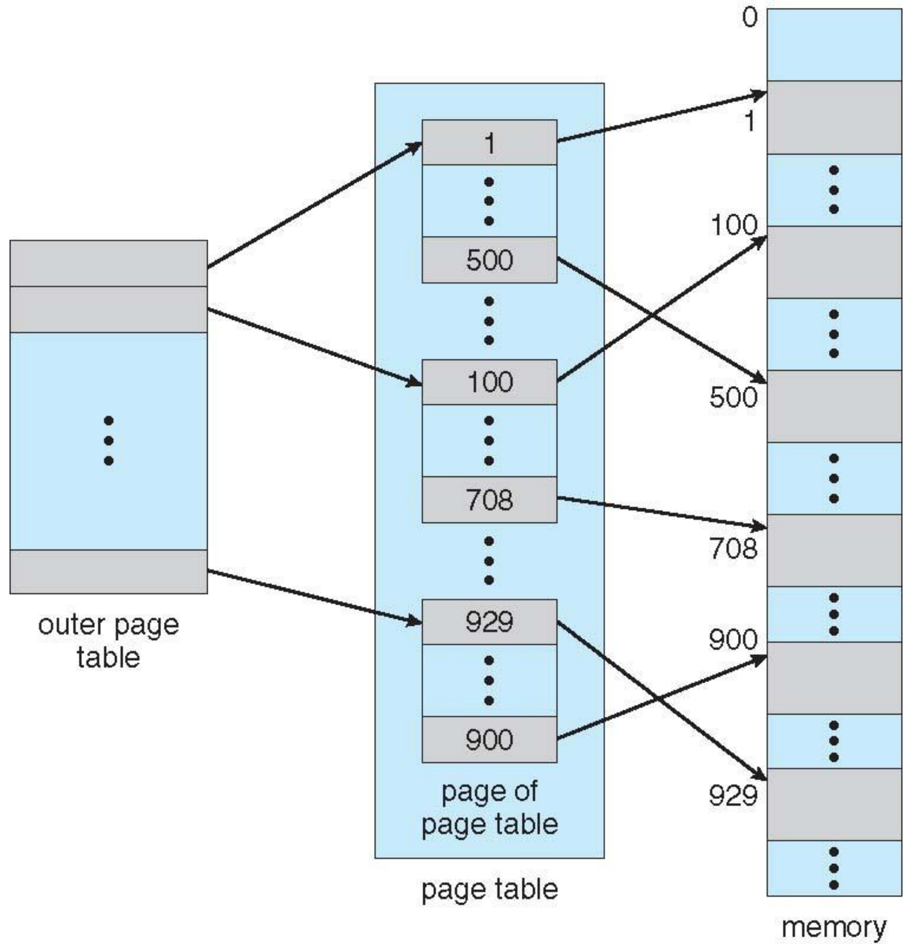
2^{32} B

- Consider a 32-bit logical address space
- Page size: 4 KB (2^{12})
- Number of pages = $2^{32} / 2^{12} = 2^{20}$
- Page table would have 2^{20} entries
- If each entry is 4 bytes:
  - Space for page table for each process = 2^{20} * 4 bytes = 4 MB
- Do not want to allocate space contiguously in main memory
- Solutions:
  - Hierarchical Paging
  - Hashed Page Tables
  - Inverted Page Tables

1)Hierarchical Page Tables

- Break up the logical address space into multiple page tables
- A simple technique is a two-level page table
- We then page the page table

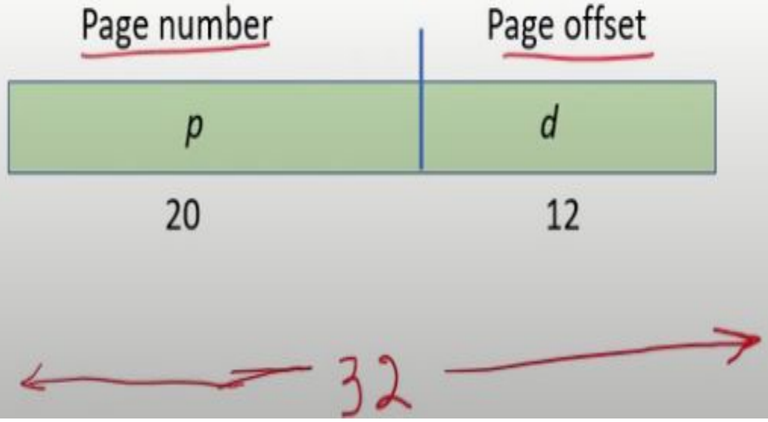
Two-Level Page-Table Scheme

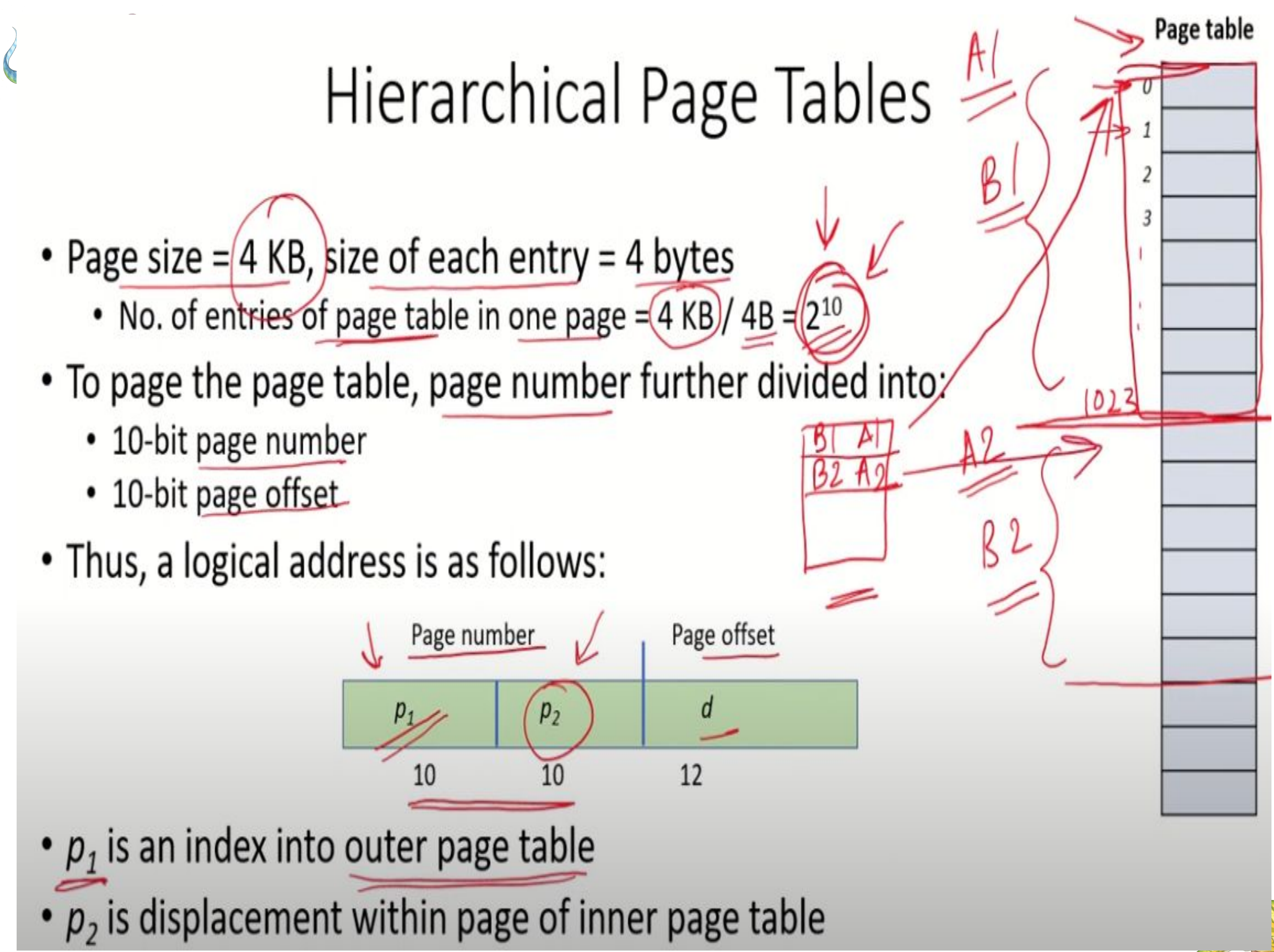


Hierarchical Page Tables

- Page the page table
  - Multi-level page table
- 32-bit logical address, 4K page size, is divided into:
  - Page number: 20 bits
  - Page offset: 12 bits

$2^{\boxed{20}}$ pages



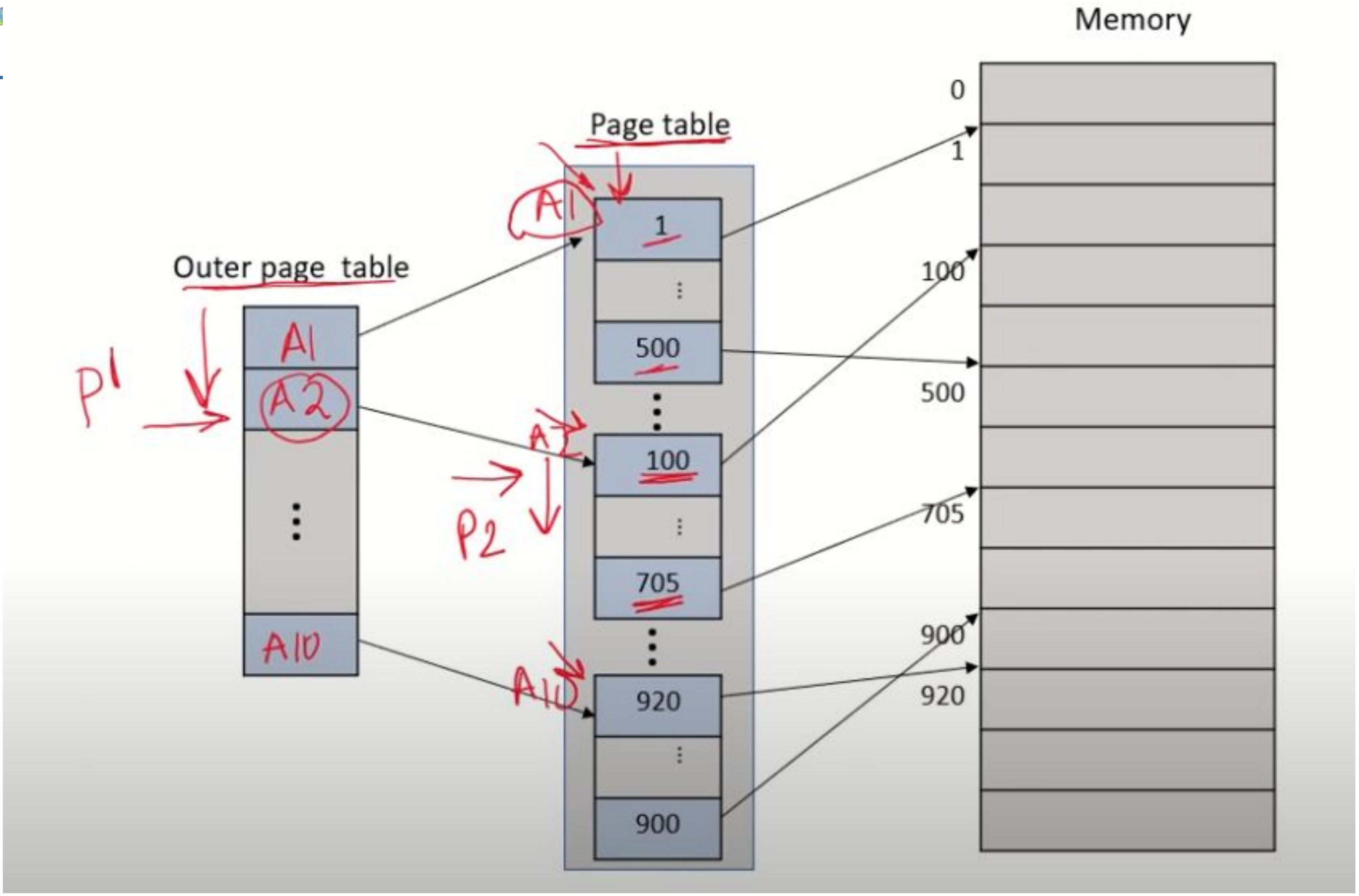


Hierarchical Page Tables

- Page size = 4 KB, size of each entry = 4 bytes
  - No. of entries of page table in one page = 4 KB / 4B = 2^{10}
- To page the page table, page number further divided into:
  - 10-bit page number
  - 10-bit page offset
- Thus, a logical address is as follows:

Page number		Page offset
p_1	p_2	d
10	10	12

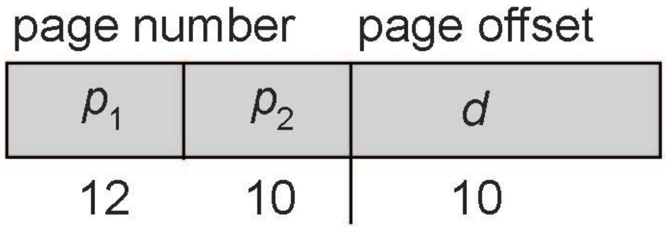
- p_1 is an index into outer page table
- p_2 is displacement within page of inner page table



Memory
Page table
Outer page table
A1
A2
A10
P1
P2
1
500
100
705
920
900
0
1
100
500
705
900
920

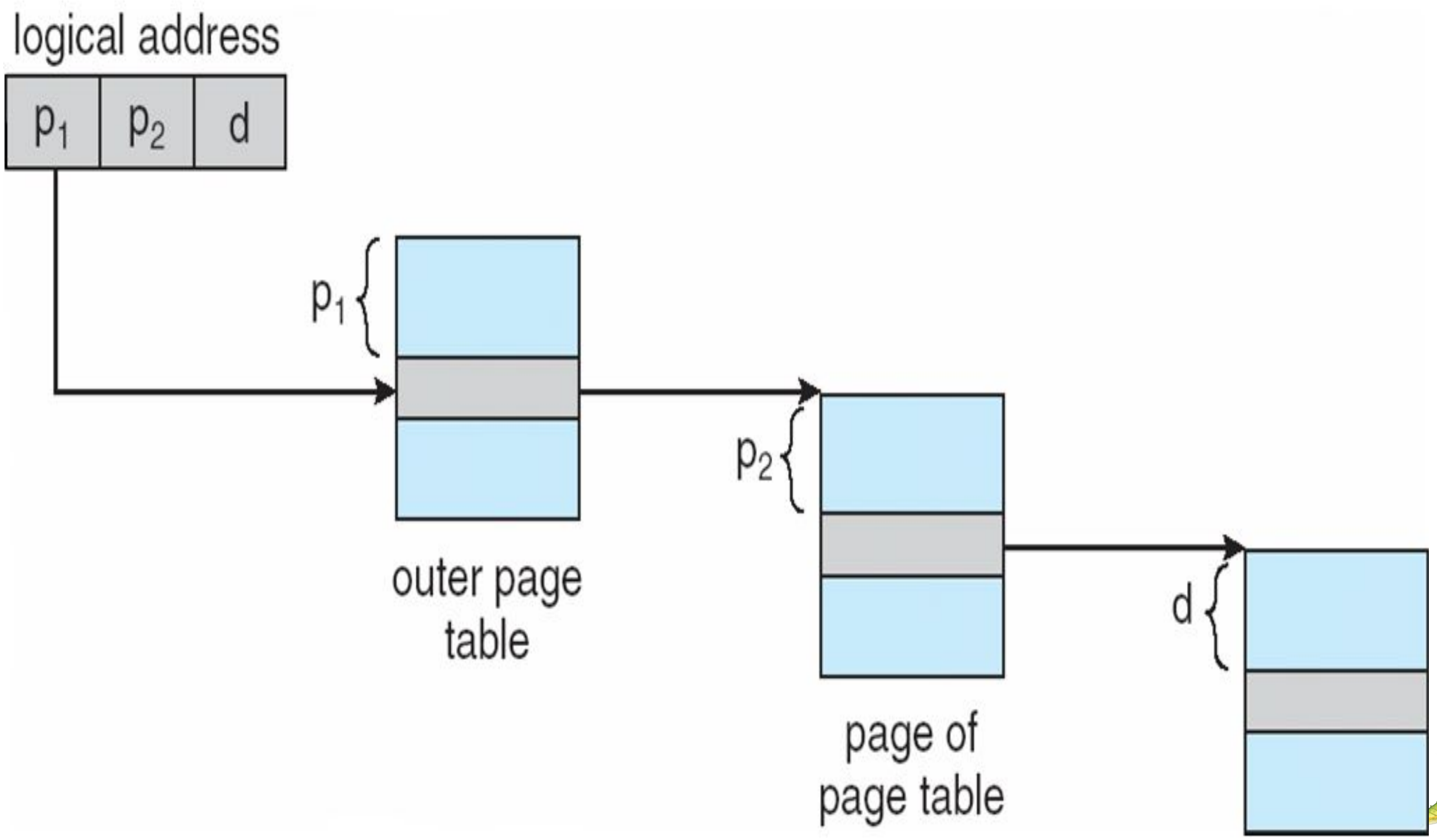
Two-Level Paging (Textbook-Example)

- 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:



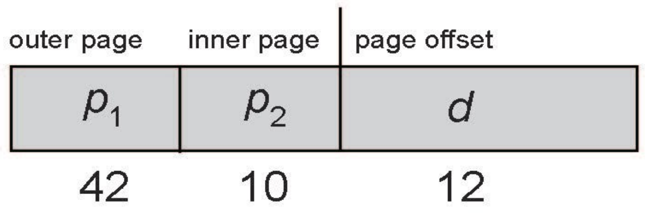
- where p_1 is an index into the outer page table, and p_2 is the displacement within the page of the inner page table
- Known as **forward-mapped page table**

Heirarchical paging:Address-Translation Scheme



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 2nd outer page table
  - But in the following example the 2nd 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

outer page	inner page	offset
p_1	p_2	d
42	10	12

2nd outer page	outer page	inner page	offset
p_1	p_2	p_3	d
32	10	10	12

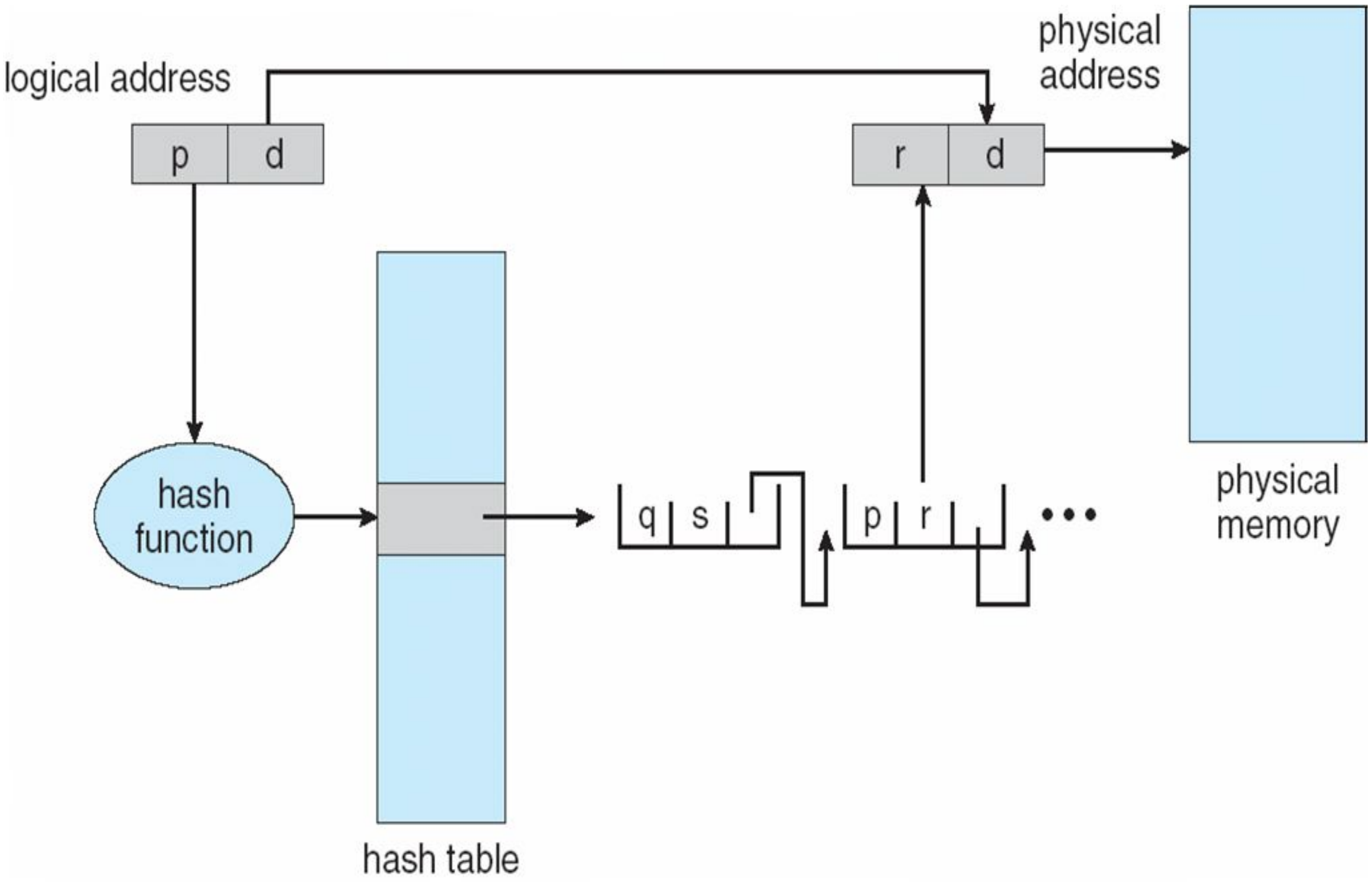
2)Hashed Page Tables

- Common in address spaces > 32 bits
- The virtual 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 virtual page number (2) the value of the mapped page frame (3) a pointer to the next element
- Virtual page numbers are compared in this chain searching for a match
  - If a match is found, the corresponding physical frame is extracted
- Variation for 64-bit addresses is **clustered page tables**
  - Similar to hashed but each entry refers to several pages (such as 16) rather than 1
  - Especially useful for **sparse** address spaces (where memory references are non-contiguous and scattered)

Hashed Page Tables

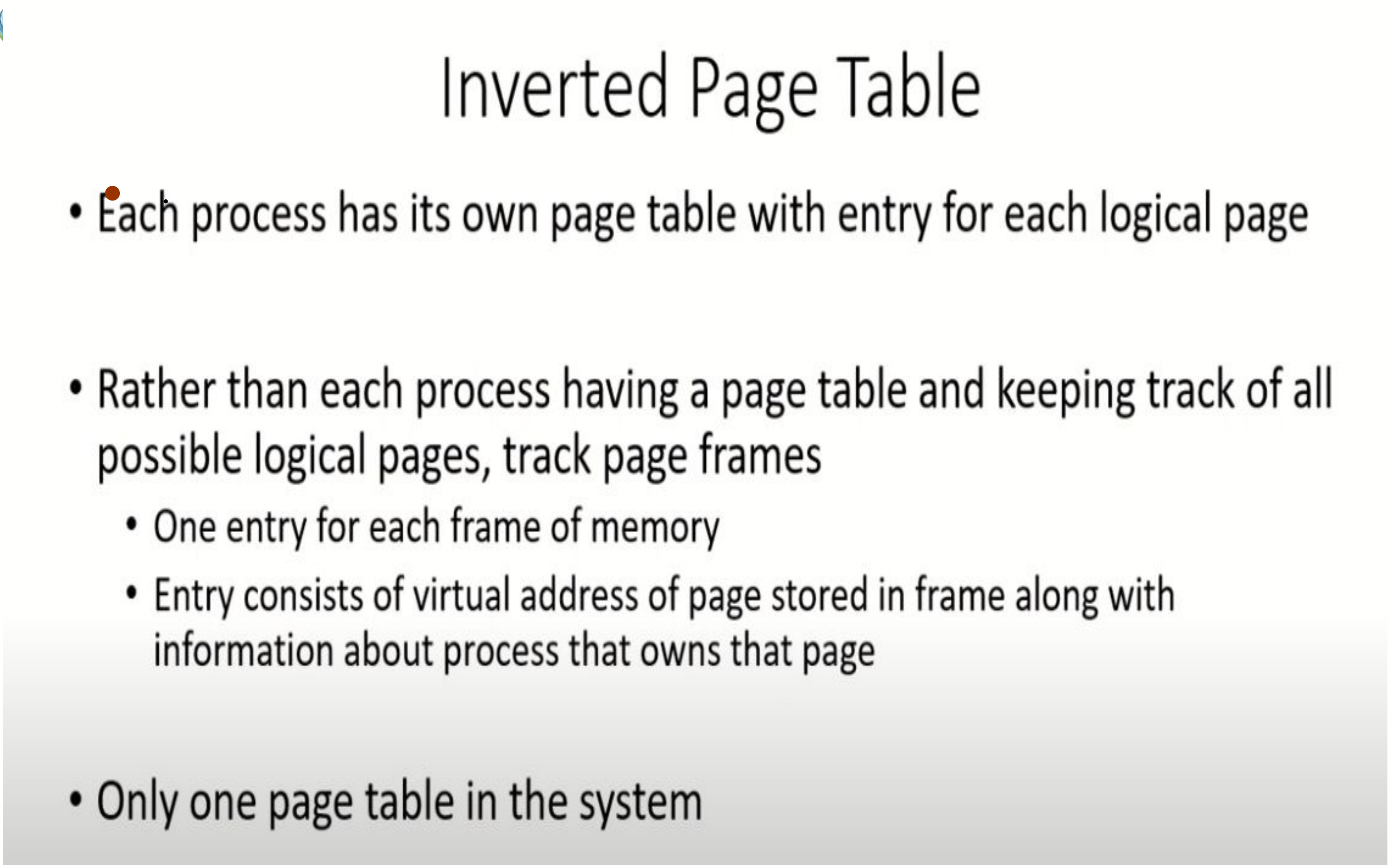
- Common in address spaces > 32 bits
- Virtual page number is hashed into a page table
  - This page table contains a chain of elements hashing to same location
- Each element contains:
  - Virtual page number
  - Value of mapped page frame
  - Pointer to next element
- Virtual page numbers are compared in this chain to look for a match
  - If a match is found, corresponding physical frame is extracted

Hashed Page Table



3) Inverted Page Table

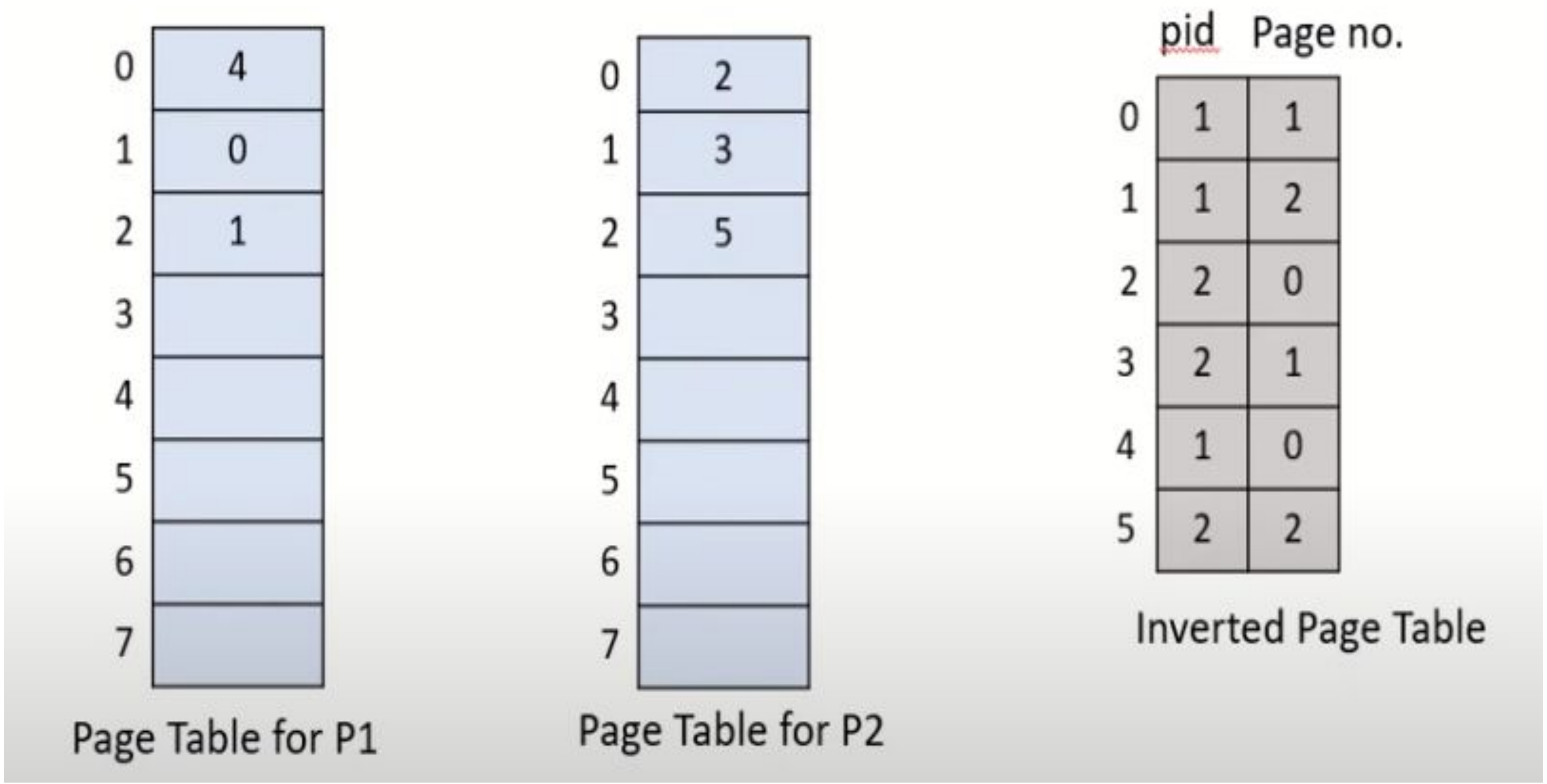
- Rather than each process having a page table and keeping track of all possible logical pages, track all physical pages
- One entry for each real page of memory
- Entry consists of the virtual address of the page stored in that real memory location, with information about the process that owns that page
- Decreases memory needed to store each page table, but increases time needed to search the table when a page reference occurs
- Use hash table to limit the search to one — or at most a few — page-table entries
  - TLB can accelerate access
- But how to implement shared memory?
  - One mapping of a virtual address to the shared physical address



Inverted Page Table

- Each process has its own page table with entry for each logical page
- Rather than each process having a page table and keeping track of all possible logical pages, track page frames
  - One entry for each frame of memory
  - Entry consists of virtual address of page stored in frame along with information about process that owns that page
- Only one page table in the system

Inverted Page Table



0	4
1	0
2	1
3	
4	
5	
6	
7	

Page Table for P1

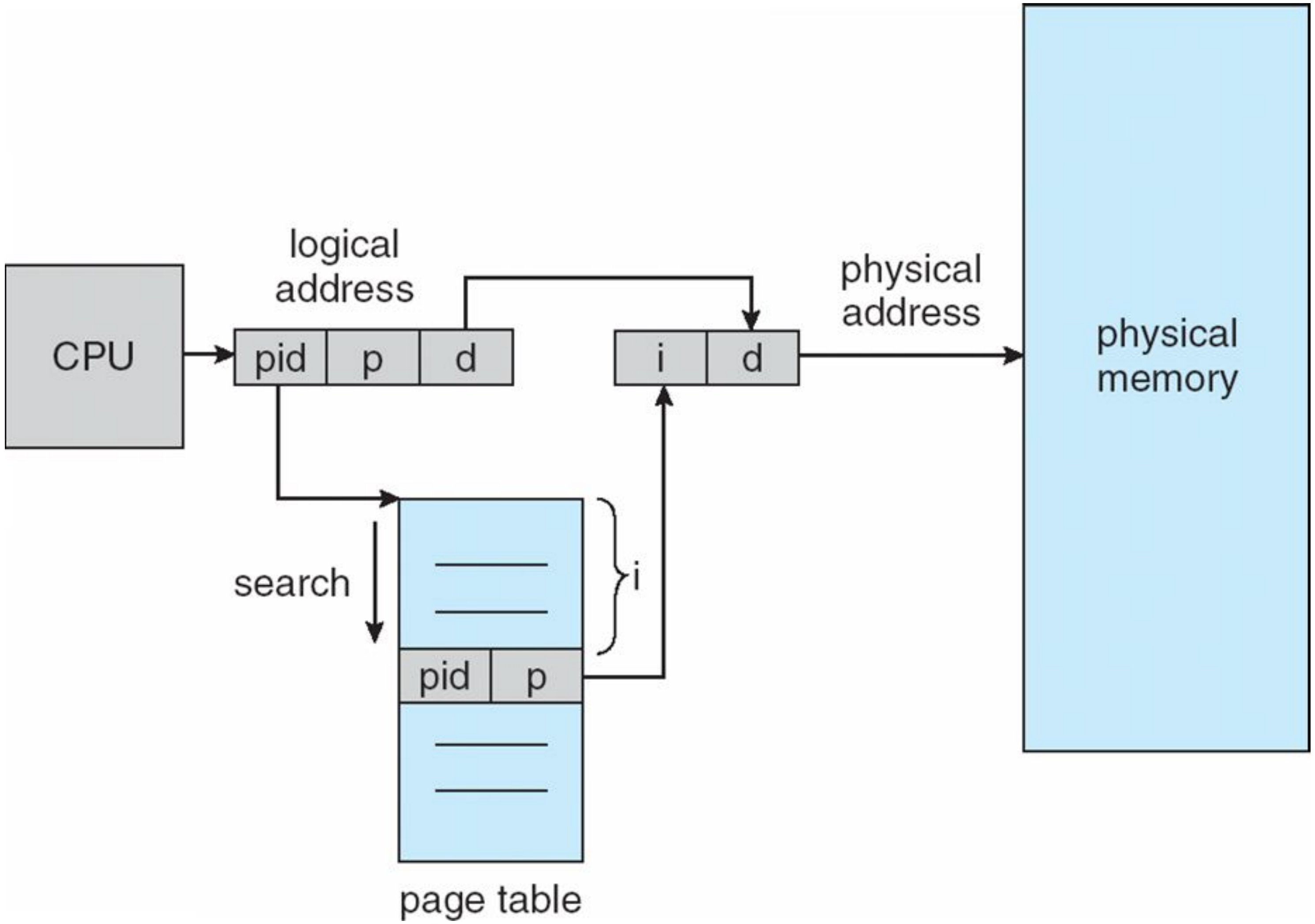
0	2
1	3
2	5
3	
4	
5	
6	
7	

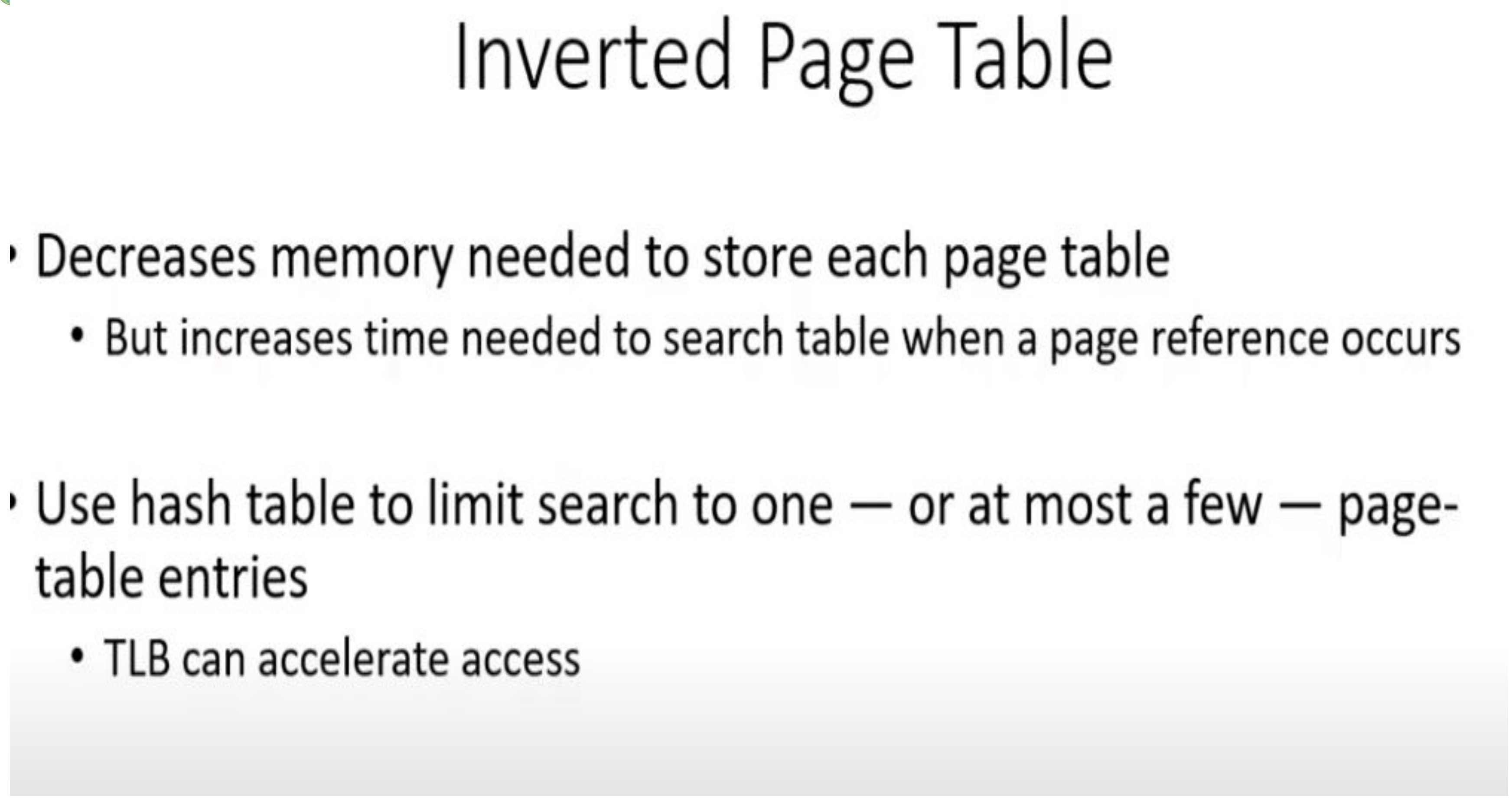
Page Table for P2

	pid	Page no.
0	1	1
1	1	2
2	2	0
3	2	1
4	1	0
5	2	2

Inverted Page Table

Inverted Page Table Architecture





Inverted Page Table

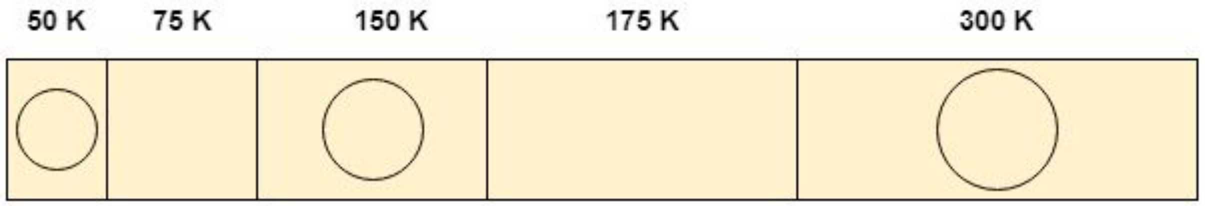
- Decreases memory needed to store each page table
  - But increases time needed to search table when a page reference occurs
- Use hash table to limit search to one — or at most a few — page-table entries
  - TLB can accelerate access

PROBLEMS ON
FIRST FIT
BEST FIT
WORST FIT

Problem 1:

Q. Process requests are given as;

25 K , 50 K , 100 K , 75 K

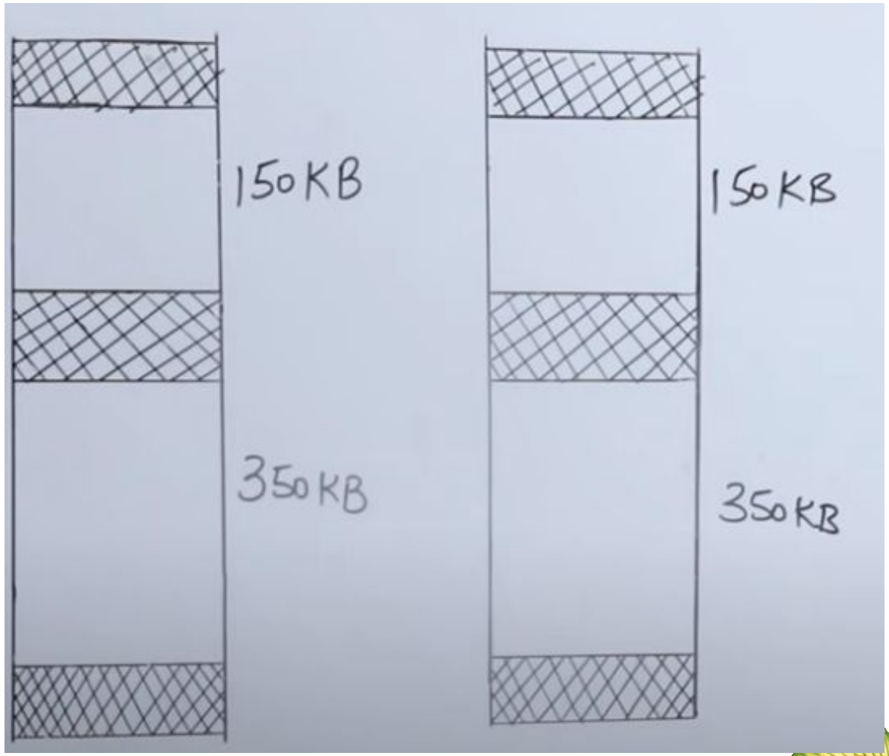


Determine the algorithm which can optimally satisfy this requirement.

Problem 2:

rEQUESTS FROM PROCESS ARE 300k, 25K, 125K, 50K

aPPLY aLL THE 3 ALGOS AND JUSTIFY WHICH IS BEST FOR THIS EXAMPLE.



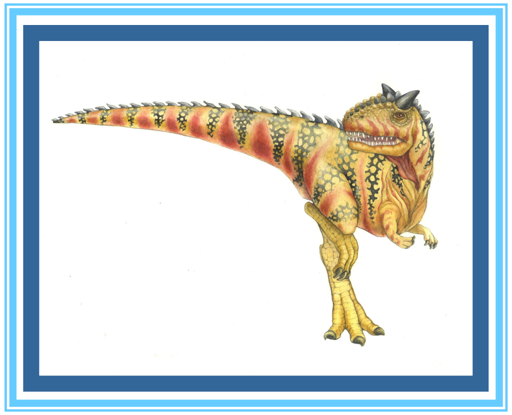
Problem 3:

GIVEN 5 MEMORY PARTITIONS OF 100K, 500K, 200K, 300k

How would These algorithms (first fit, best fit, worst fit) place each of these processes in the available partition:

212K, 417K, 112K, 426K

End of Chapter 1 (UNIT-4)



Page replacement algorithms

- Page replacement is a process of swapping out an existing page from the frame of a main memory and replacing it with the required page.
- Page replacement is required when-
- All the frames of main memory are already occupied.
- Thus, a page has to be replaced to create a room for the required page

- A page fault occurs when a page referenced by the CPU is not found in the main memory.
- The required page has to be brought from the secondary memory into the main memory.
- A page has to be replaced if all the frames of main memory are already occupied.
-

Page replacement algorithms

- Page replacement algorithms help to decide which page must be swapped out from the main memory to create a room for the incoming page.

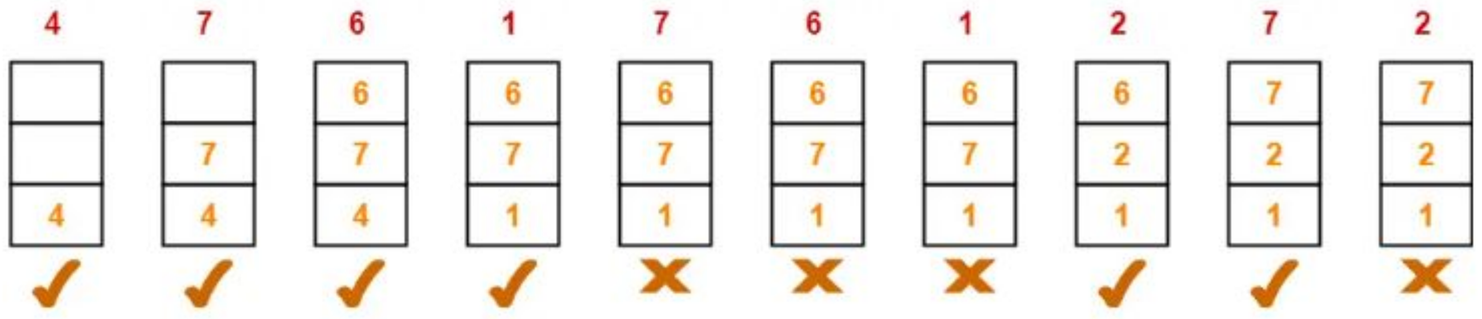
- FIFO Page Replacement Algorithm
- LRU Page Replacement Algorithm
- Optimal Page Replacement Algorithm

Page replacement algorithms

- **FIFO Page Replacement Algorithm-**
- As the name suggests, this algorithm works on the principle of "**First in First out**".
- It **replaces the oldest page that has been present** in the main memory for the longest time .It is implemented by keeping track of all the pages in a queue.
- **LRU Page Replacement Algorithm-**
- As the name suggests, this algorithm works on the principle of "**Least Recently Used**".
- It **replaces the page that has not been referred by the CPU** for the longest time.
- **Optimal Page Replacement Algorithm-**
- This algorithm replaces the page that will not be referred by the CPU in future for the longest time.
- it is the best known algorithm and gives the least number of page faults.

Page replacement algorithms

- .FIFO
- Total number of page faults occurred = 6
- **<u>Calculating Hit ratio-</u>**
- Total number of page hits
- = Total number of references – Total number of page misses or page faults
- = 10 – 6 = 4
- Thus, Hit ratio
- = Total number of page hits / Total number of references
- = 4 / 10
- = 0.4 or 40%



PROBLEM

- Consider page reference string 6, 1, 1, 2, 0, 3, 4, 6, 0, 2, 1, 2, 1, 2, 0, 3, 2, 1, 4, 0
- with 4 page frames. Find number of page faults.
- (use FIFO, LRU and optimal page replacement algorithms)
-

- Consider page reference string 0 1 2 3 0 1 4 0 1 2 3 4
- with 3 page frames. Find number of page faults.
- (use FIFO, LRU and optimal page replacement algorithms)

- Consider page reference string **Page string 2,3,4,1,7,4,2,5,7,1**
- with 3 page frames. Find number of page faults.
- (use FIFO, LRU and optimal page replacement algorithms)