

Unit 4- Memory Management

J.Premalatha

Professor/IT

Kongu Engineering College

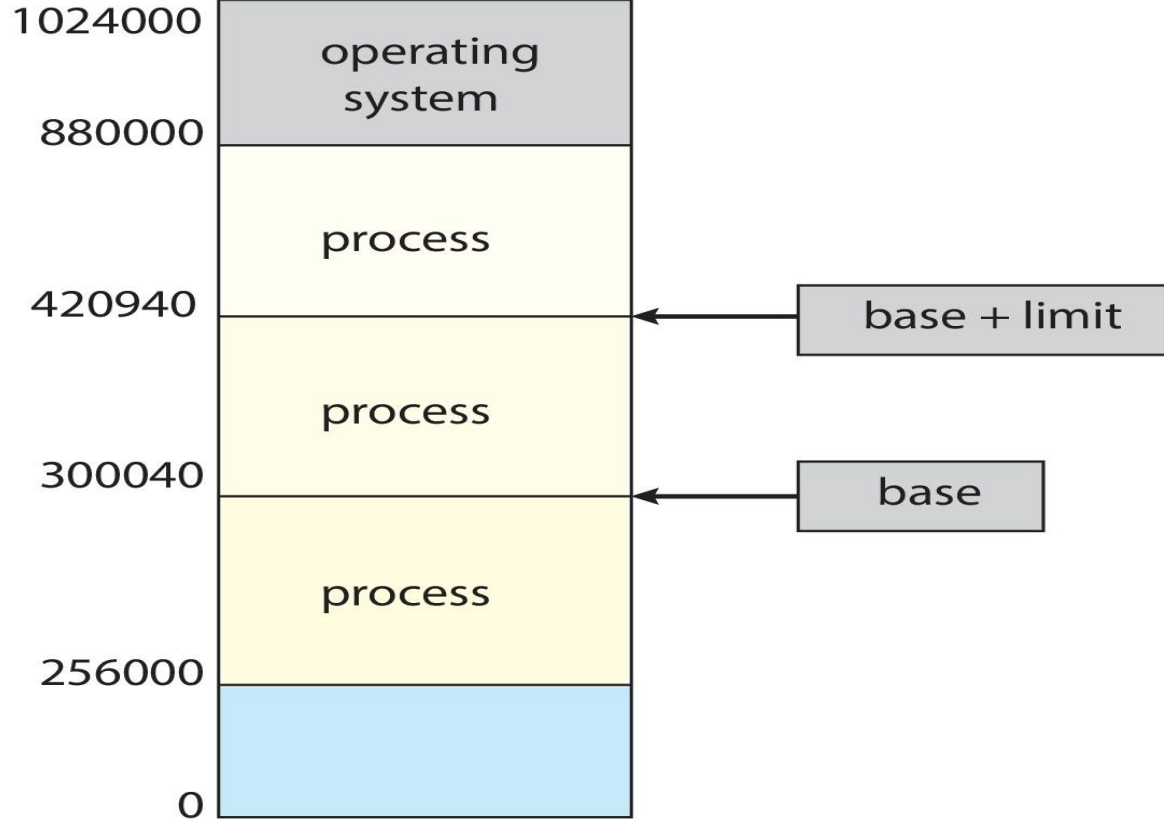
Perundurai

Background

- Program must be brought (from disk) into memory and placed within a process for it to be run
- Main memory and the registers built into each processing core are the only general-purpose storage that the CPU can access directly.
- Memory unit only sees a stream of:
 - addresses + read requests, or
 - address + data and write requests
- Register access is done in one CPU clock (or less)
- Main memory can take many cycles, causing a **stall**
- **Cache** sits between main memory and CPU registers
- For proper system operation, to protect the operating system from access by user processes, as well as protect user processes from one another. This protection must be provided by the hardware, because the operating system doesn't intervene between the CPU and its memory accesses

Protection

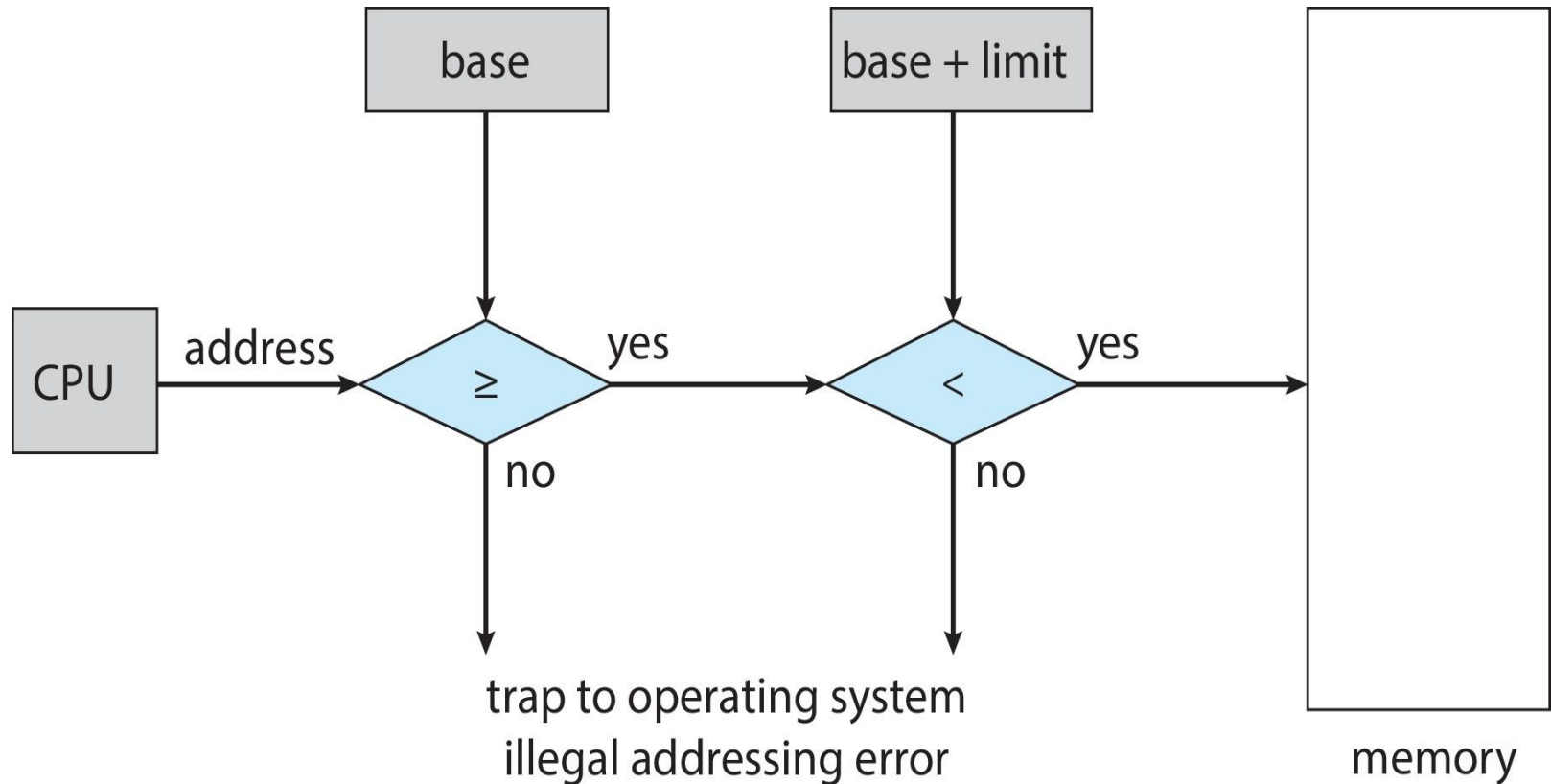
- **Each process has a separate memory space. Separate per-process memory space protects the processes from each other**
- To separate memory spaces, to determine the range of legal addresses that the process may access and to ensure that the process can access only these legal addresses.
- To provide this protection by using two registers, **a base and a limit**
- 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



Protection of memory space is accomplished by having the CPU 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.** This scheme prevents a user program from (accidentally or deliberately) modifying the code or data structures of either the operating system or other users.

Hardware Address Protection

- CPU must check every memory access generated in user mode to be sure it is between base and limit for that user



- The **base and limit registers can be loaded only by the operating system**, which uses a special privileged instruction.

Address Binding

Addresses may be represented in different ways during these steps

Source program : Addresses in the are generally symbolic (such as the variable count, MAX, MIN).

Compiler : Typically **binds these symbolic addresses to relocatable addresses** (such as “14 bytes from the beginning of this module”).

Linker or loader : Binds the relocatable addresses to absolute addresses (such as 74014).

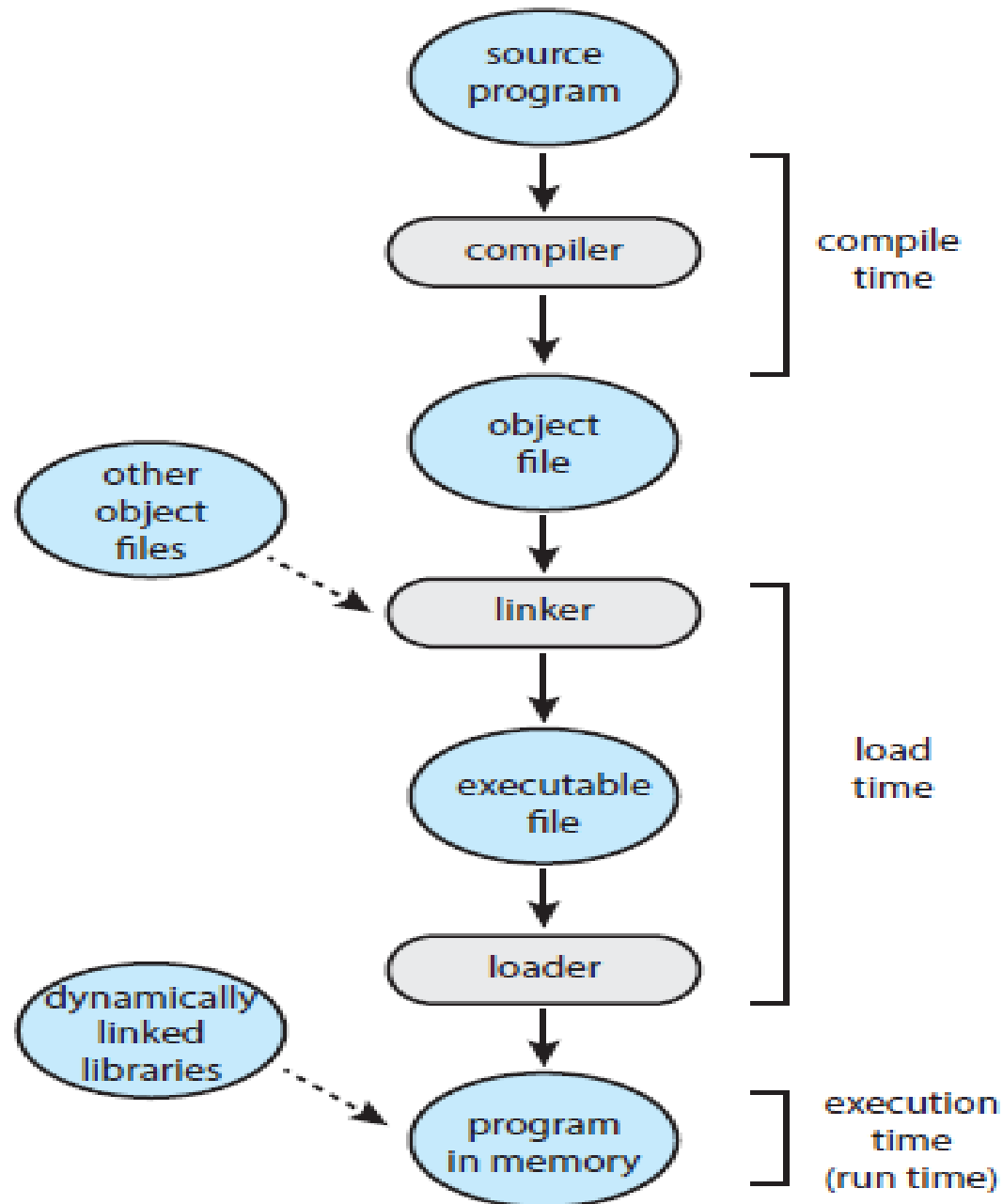
Each binding is a mapping from one address space to another.

- Programs on disk, ready to be brought into memory to execute from an **input queue**
- **Input Queue → Select a process → Loads it into memory → Executes and accesses instructions and data from memory → Process terminates → memory space is declared available → memory is reclaimed for use by other processes**

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 we know at compile time where the process will reside in memory, then absolute code can be generated. For example, if we know that a user process will reside starting at location R , *then the generated* compiler code will start at that location and extend up from there. If, at some later time, the starting location changes, then it will be necessary to recompile this code.
- **Load time:** If it is not known at compile time where the process will reside in memory, then the compiler must generate relocatable code. In this case, final binding is delayed until load time. If the starting address changes, we need only reload the user code to incorporate this changed value.
- **Execution time:** If the process can be moved during its execution from one memory segment to another, then binding must be delayed until run time. Special hardware is required for this scheme to work.

Multistep Processing of a User Program

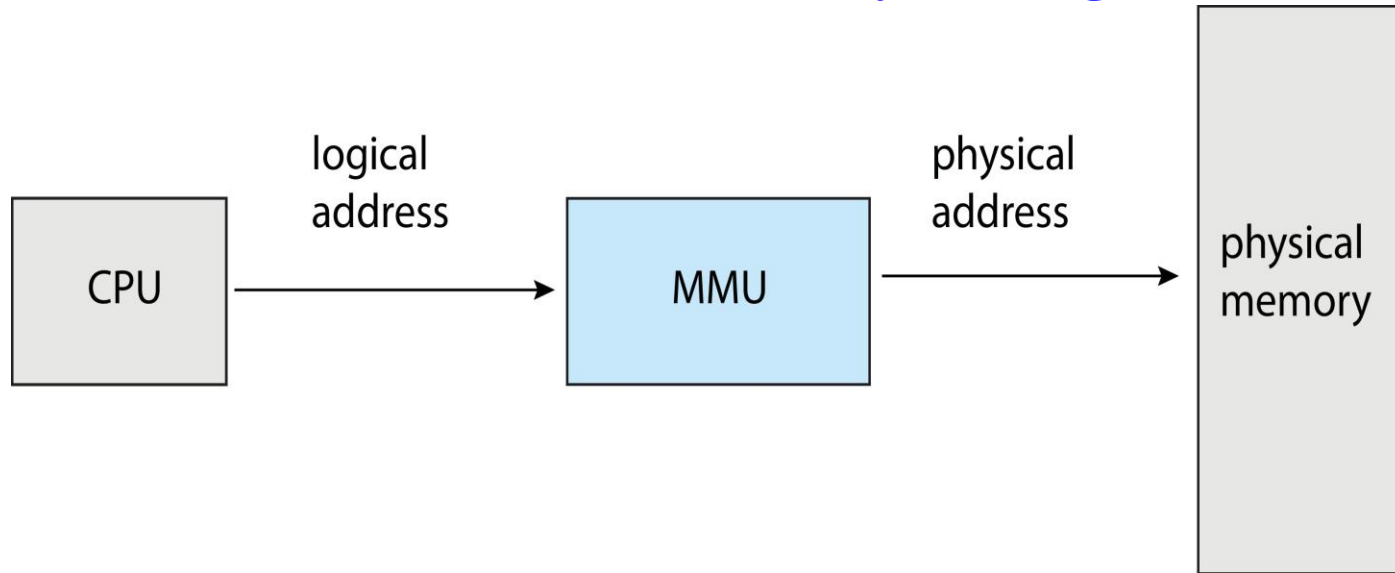


Logical vs. Physical Address Space

- **Logical address** : An address generated by the CPU
- **Physical address** : An address seen by the memory unit—that is, the one loaded into Memory Address Register [MAR] of the memory
- Compile or Load time address binding generates identical logical and physical addresses [or] Logical and physical addresses are the same in compile-time and load-time address-binding schemes;
- Execution-time address-binding scheme results in differing logical and physical addresses. In that case, logical address is referred as virtual address
- **Logical address space** is the set of all logical addresses generated by a program
- **Physical address space** is the set of all physical addresses corresponding to those logical addresses

Memory-Management Unit (MMU)

- The run-time mapping from virtual to physical addresses is done by a hardware device called the **memory-management unit (MMU)**

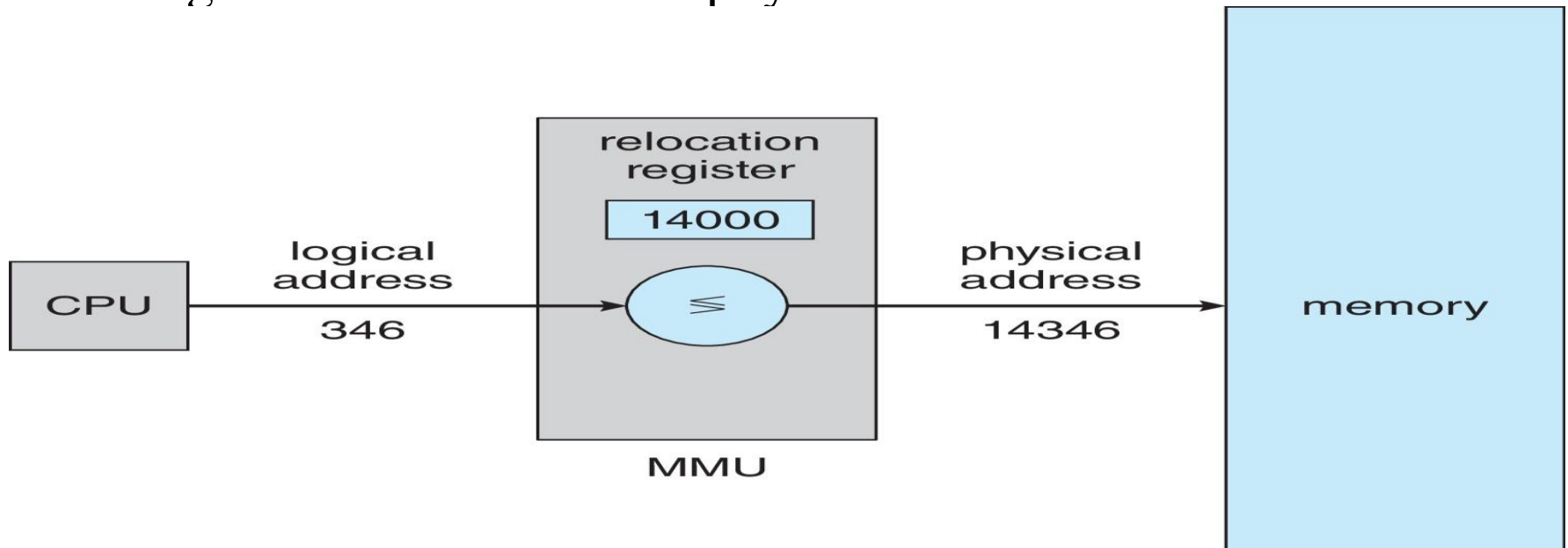


Many different methods to accomplish such mapping. One technique is mapping with a simple MMU scheme that is a generalization of the base register scheme.

Logical addresses (in the range 0 to max) and physical addresses (in the range $R + 0$ to $R + max$ for a base value R). The user program generates only logical addresses and thinks that the process runs in memory locations from 0 to max . However, these logical addresses must be mapped to physical addresses before they are used.

Memory-Management Unit (Cont.)

- The base register now called relocation register
- The value in the relocation 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
 - Execution-time binding occurs when reference is made to location in memory
 - Logical address bound to physical addresses



Dynamic Loading

- Entire program and all data of a process to be in physical memory for the process to execute. The size of a process has thus been limited to the size of physical memory. To obtain better memory-space utilization, use **dynamic loading**.
- In 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. Then control is passed to the newly loaded routine.
- Routine is not loaded until it is called is the advantage of dynamic loading; 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 - OS can help by providing libraries to implement dynamic loading

Dynamic Linking and Shared Libraries

- **Dynamically linked libraries (DLLs)** are system libraries that are linked to user programs when the programs are run
- **Static linking** – system libraries and program code combined by the loader into the binary program
- **Dynamic linking** –linking postponed until execution time.

Advantages of DLL:

1. If **DLL is not in the system**, each program on a system must include a copy of its language library in the executable format. This requirement **not only increases the size of an executable format but also waste main memory.**
2. 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, DLLs are also known as **shared libraries, and are used extensively in Windows and Linux** systems.

Small piece of code, stub, used to locate the appropriate memory-resident library routine or how to load the library if the routine is not already present.

- When the stub is executed, it checks to see whether the needed routine is already in memory
- If not, the program loads the routine into memory
- Either way, the Stub replaces itself with the address of the routine, and executes the routine
- Thus, the next time that particular code segment is called, the library routine executed directly, incurring no cost for dynamic linking

Dynamically linked libraries can be extended to library updates (such as bug fixes). In addition, a library may be replaced by a new version, and all programs that reference the library will automatically use the new version. Versioning may be needed

Memory Partition

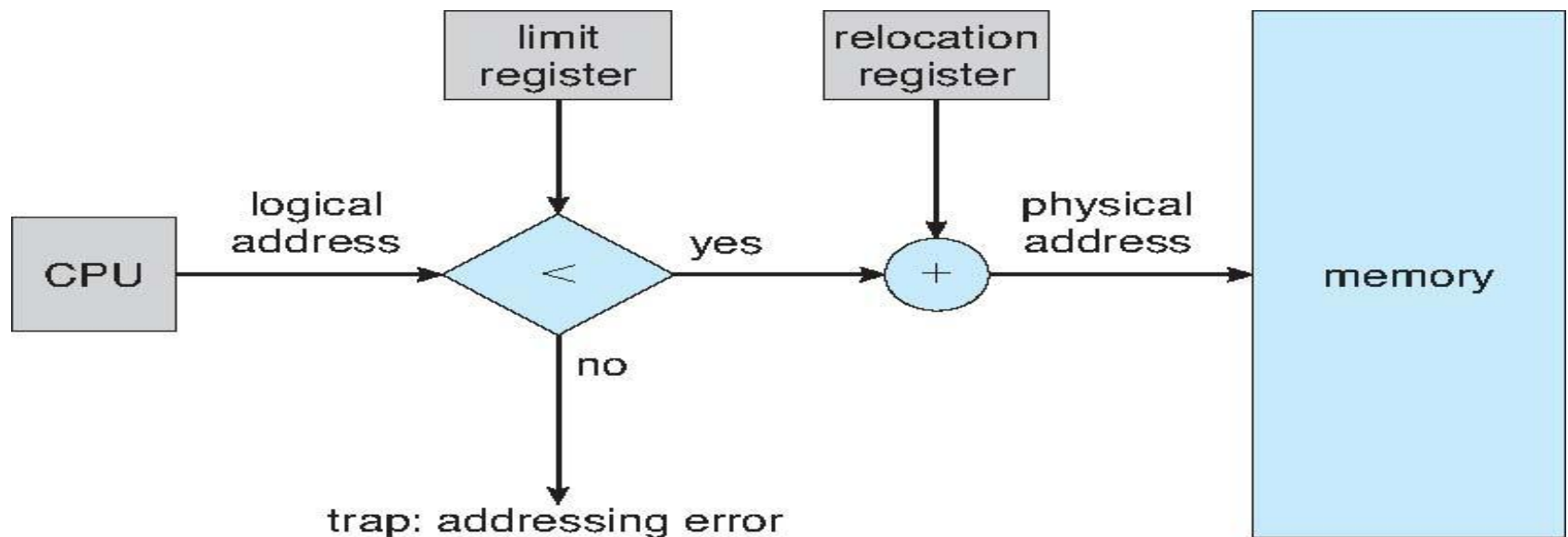
Contiguous Partition - Variable Partition - First Fit, Best Fit, Worst Fit algorithms

Fixed same Size Partition - Paging

Contiguous Allocation

- The main memory must accommodate both the operating system and the various 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 (or) high memory**
 - **User processes then held in high (or) low memory**
 - Each process contained in single contiguous section of memory

- ## Memory Protection- Hardware Support for Relocation and Limit Registers
- Prevent a process from accessing memory that it does not own. For that, a system with a relocation register together with a limit register is used.
 - The relocation register contains the value of the smallest physical address; the limit register contains the range of logical addresses (for example, relocation = 100040 and limit = 74600). Each logical address must fall within the range specified by the limit register.
 - The MMU maps the logical address dynamically by adding the value in the relocation register

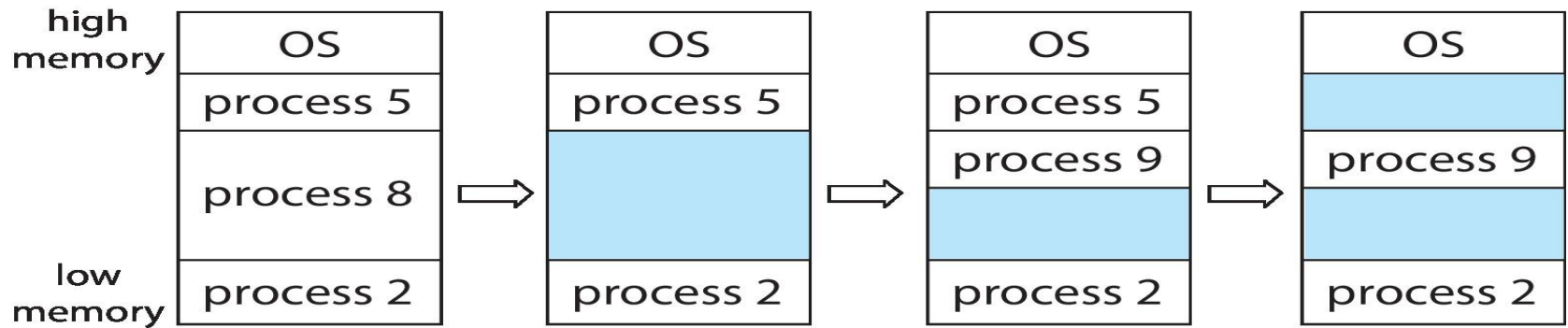


When the CPU scheduler selects a process for execution, the dispatcher loads the relocation and limit registers with the correct values as part of the context switch.

Because every address generated by a CPU is checked against these registers, it protect both the operating system and the other users' programs and data from being modified by this running process.

Memory Allocation -Variable Partition

- The simplest methods of **allocating memory is to assign processes to variably sized partitions in memory**, where **each partition may contain exactly one process**.
- In this variable partition scheme, **the operating system keeps a table indicating which parts of memory are available and which are occupied**.
- **Initially, all memory is available for user processes and is considered one large block of available memory, a hole**.
- 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)**



Initially, the memory is fully utilized, containing processes 5, 8, and 2. After process 8 leaves, there is one contiguous hole. Later on, process 9 arrives and is allocated memory. Then process 5 departs, resulting in two noncontiguous holes.

What happens when there isn't sufficient memory to satisfy the demands of an arriving process?

Two solutions

1. To simply reject the process and provide an appropriate error message.
2. Place such processes into a wait queue. When memory is later released, the operating system checks the wait queue to determine if it will satisfy the memory demands of a waiting process.

Dynamic Storage-Allocation Problem

- In general, the memory blocks available comprise a *set of holes of various sizes scattered throughout memory*.
- When a process arrives and needs memory, the system searches the set for a hole that is large enough for this process.
- If the hole is too large, it is split into two parts. One part is allocated to the arriving process; the other is returned to the set of holes.
- When a process terminates, it releases its block of memory, which is then placed back in the set of holes. If the new hole is adjacent to other holes, these adjacent holes are merged to form one larger hole.

This procedure is a particular instance of the general **dynamic storage allocation problem**, which concerns how to satisfy a request of size n *from a* list of free holes. There are many solutions to this problem. The **first-fit, best-fit, and worst-fit strategies are the ones most commonly used to select a free hole** from the set of available holes.

Dynamic Storage-Allocation Problem (continued)

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

First fit: Allocate the first hole that is big enough. Searching can start either at the beginning of the set of holes or at the location where the previous first-fit search ended. Stop searching as soon as find a free hole that is large enough.

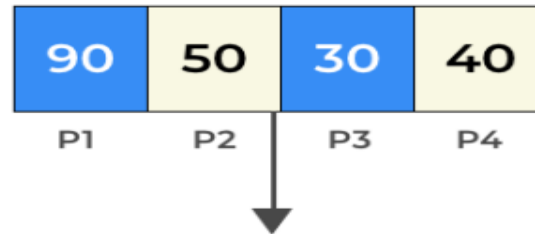
Best fit: Allocate the smallest hole that is big enough. Search the entire list, unless the list is ordered by size. This strategy produces the smallest leftover hole.

Worst fit : Allocate the largest hole. Again, search the entire list, unless it is sorted by size. This strategy produces the largest leftover hole, which may be more useful than the smaller leftover hole from a best-fit approach.

From Simulations , **both first fit and best fit are better than worst fit in terms of decreasing time and storage utilization.** Neither first fit nor best fit is clearly better than the other in terms of storage utilization, but **first fit is generally faster**

First Fit Allocation in OS

Process Sizes



First FIT Allocation



	Size	Allocated to	Memory Wastage After Process Occupies	P4 remains Unallocated
Process 1	90	Block 2	$100 - 90 = 10$	
Process 2	50	Block 4	$200 - 50 = 150$	
Process 3	30	Block 3	$40 - 30 = 10$	
Process 4	40	Unallocated	-	

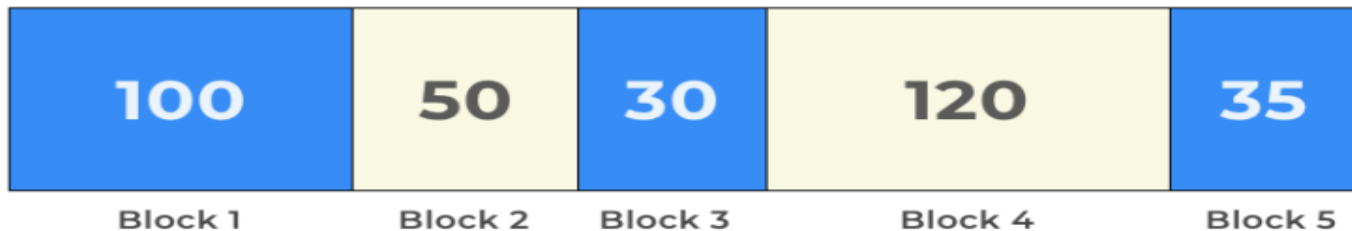
Best Fit Allocation in OS

Process Size

40



Best FIT Allocation



	Size	Can Occupy?	Memory Wastage After Process Occupies	
Block 1	100	Yes	$100 - 40 = 60$	
Block 2	50	Yes	$50 - 40 = 10$	Best
Block 3	30	No	-	
Block 4	120	Yes	$120 - 40 = 80$	
Block 5	35	No	-	

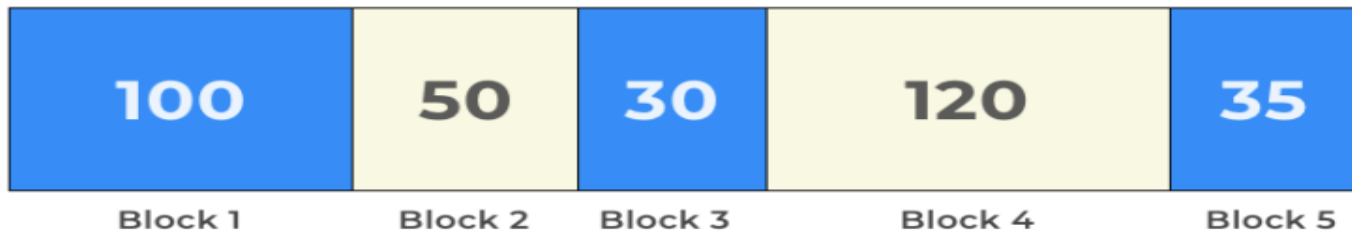
Worst Fit Allocation in OS

Process Size

40



Worst FIT Allocation



	Size	Can Occupy?	Memory Wastage After Process Occupies	
Block 1	100	Yes	$100 - 40 = 60$	
Block 2	50	Yes	$50 - 40 = 10$	
Block 3	30	No	-	
Block 4	120	Yes	$120 - 40 = 80$	Worst
Block 5	35	No	-	

EX1: Given six memory partitions of 300 KB, 600 KB, 350 KB, 200 KB, 750 KB, and 125 KB (in order), how would the First-fit, Best-Fit, and worst-Fit algorithms place processes of size 115 KB, 500 KB, 358 KB, 200 KB, and 375 KB (in order)?

First -Fit:

- a. 115 KB is put in 300-KB partition, leaving 185 KB, 600 KB, 350 KB, 200 KB, 750 KB, 125 KB
- b. 500 KB is put in 600-KB partition, leaving 185 KB, 100 KB, 350 KB, 200 KB, 750 KB, 125 KB
- c. 358 KB is put in 750-KB partition, leaving 185 KB, 100 KB, 350 KB, 200 KB, 392 KB, 125 KB
- d. 200 KB is put in 350-KB partition, leaving 185 KB, 100 KB, 150 KB, 200 KB, 392 KB, 125 KB
- e. 375 KB is put in 392-KB partition, leaving 185 KB, 100 KB, 150 KB, 200 KB, 17 KB, 125 KB

Best-Fit

- a. 115 KB is put in 125-KB partition, leaving 300 KB, 600 KB, 350 KB, 200 KB, 750 KB, 10 KB
- b. 500 KB is put in 600-KB partition, leaving 300 KB, 100 KB, 350 KB, 200 KB, 750 KB, 10 KB
- c. 358 KB is put in 750-KB partition, leaving 300 KB, 100 KB, 350 KB, 200 KB, 392 KB, 10 KB
- d. 200 KB is put in 200-KB partition, leaving 300 KB, 100 KB, 350 KB, 0KB, 392 KB, 10 KB
- e. 375 KB is put in 392-KB partition, leaving 300 KB, 100 KB, 350 KB, 0KB, 17 KB, 10 KB

Worst Fit:

- a. 115 KB is put in 750-KB partition, leaving 300 KB, 600 KB, 350 KB, 200KB, 635 KB, 125 KB
- b. 500 KB is put in 635-KB partition, leaving 300 KB, 600 KB, 350 KB, 200 KB, 135 KB, 125 KB
- c. 358 KB is put in 600-KB partition, leaving 300 KB, 242 KB, 350 KB, 200 KB, 135 KB, 125 KB
- d. 200 KB is put in 350-KB partition, leaving 300 KB, 242 KB, 150 KB, 200 KB, 135 KB, 125 KB
- e. 375 KB must wait

Problem: Given memory partitions of 100K, 500K, 200K, 300K, and 600K (in order), the First-fit memory management algorithm (search always starts from the at the beginning) places the following processes of size 212K, 417K, 112K, and 426K (in order). What is the amount of external fragmentation and internal fragmentation?

- (A) 288K, 400K (B) 200K, 600 (C) **600K, 359K** (D) 359K, 600K
(E) 500K , 200K

External Fragmentation

1. 212KB is put in 500KB partition.
2. 417KB is put in 600KB partition.
3. 112KB is put in 288KB partition (new partition $288K = 500K - 212K$).
4. 426K must wait. **Total External fragmentation** is $(100+200+300) = 600KB$

Internal Fragmentation

1. 176 KB in 500KB partition($500-212-112$) .
 2. 183 KB in 600KB partition($600-417$) .
- Total Internal fragmentation** is $(176+183) = 359KB$

Fragmentation

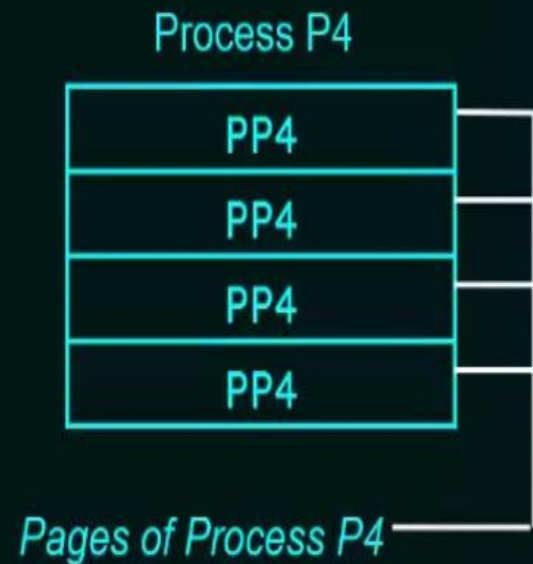
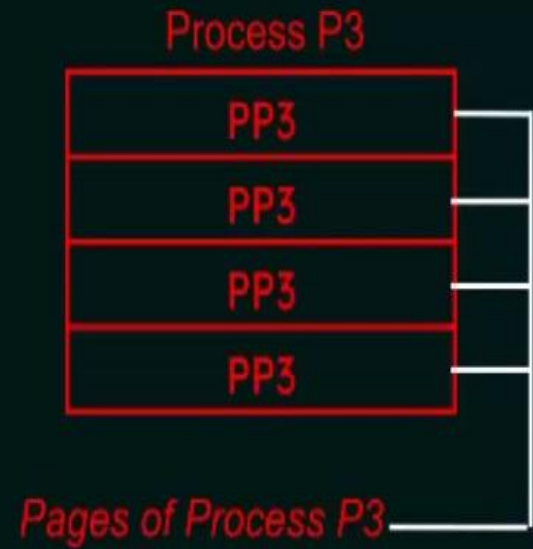
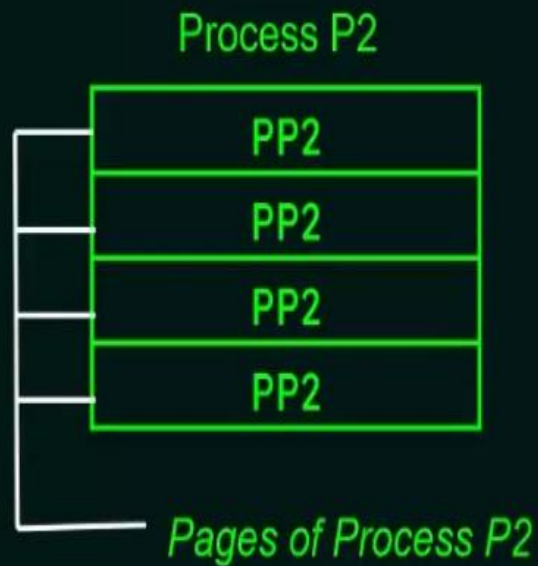
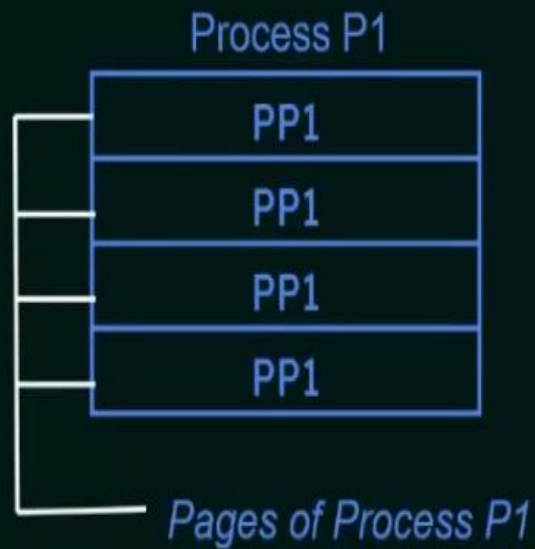
- **External Fragmentation** – total memory space exists to satisfy a request, but it is not contiguous
- **Internal Fragmentation** – allocated memory may be slightly larger than requested memory; this size difference is memory internal to a partition, but not being used
- **First fit analysis reveals that given N blocks allocated, $0.5 N$ blocks lost to fragmentation**
 - $1/3$ may be unusable -> **50-percent rule**

Solution to Fragmentation – 1.Compaction 2. Paging

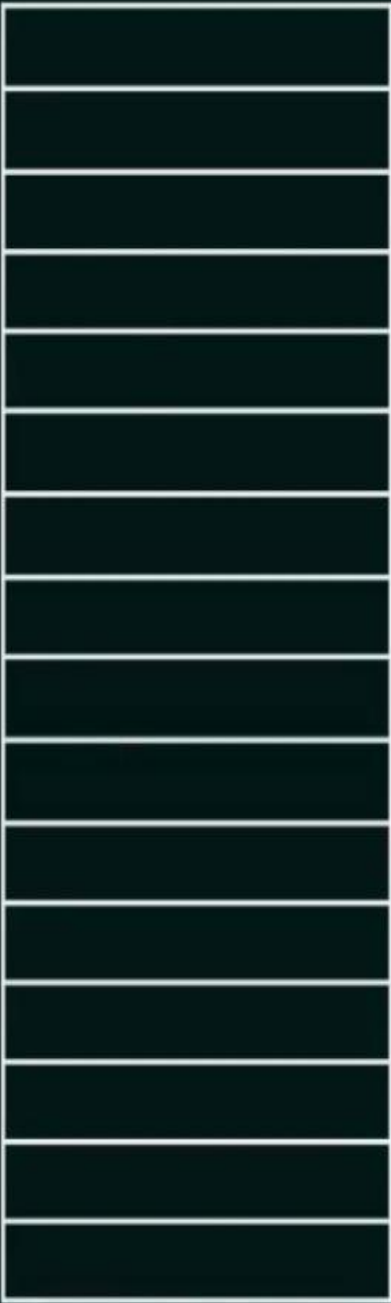
- Reduce external fragmentation by **compaction**
 - Shuffle memory contents to place all free memory together in one large block
 - Compaction is possible *only* if relocation is dynamic, and is done at execution time

Paging

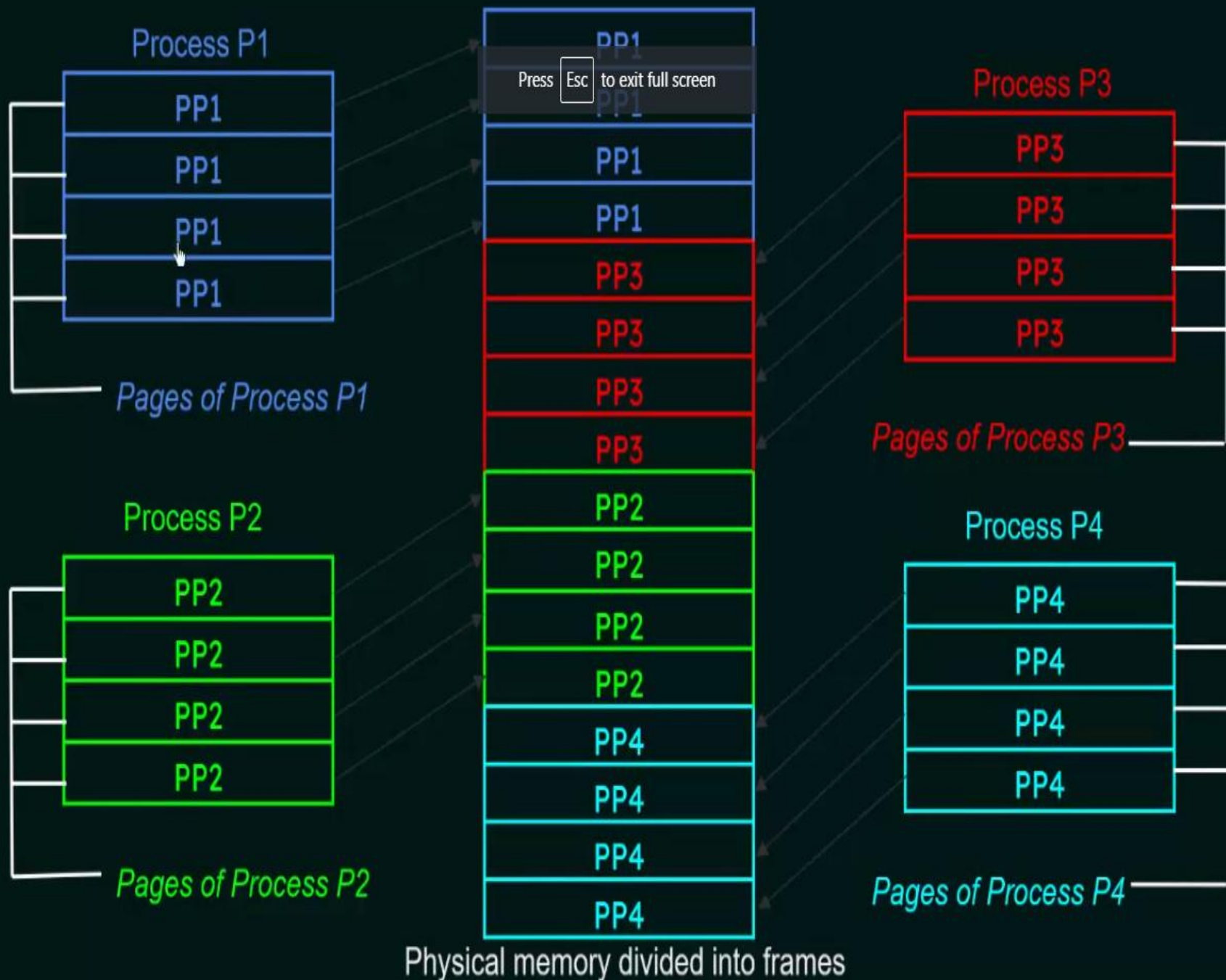
- Physical address space of a process can be noncontiguous; process is allocated physical memory whenever the latter is available
 - Avoids external fragmentation
 - Avoids problem of varying sized memory chunks
- **Divide physical memory into fixed-sized blocks called frames**
 - Size is power of 2, between 512 bytes and 16 Mbytes
- **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

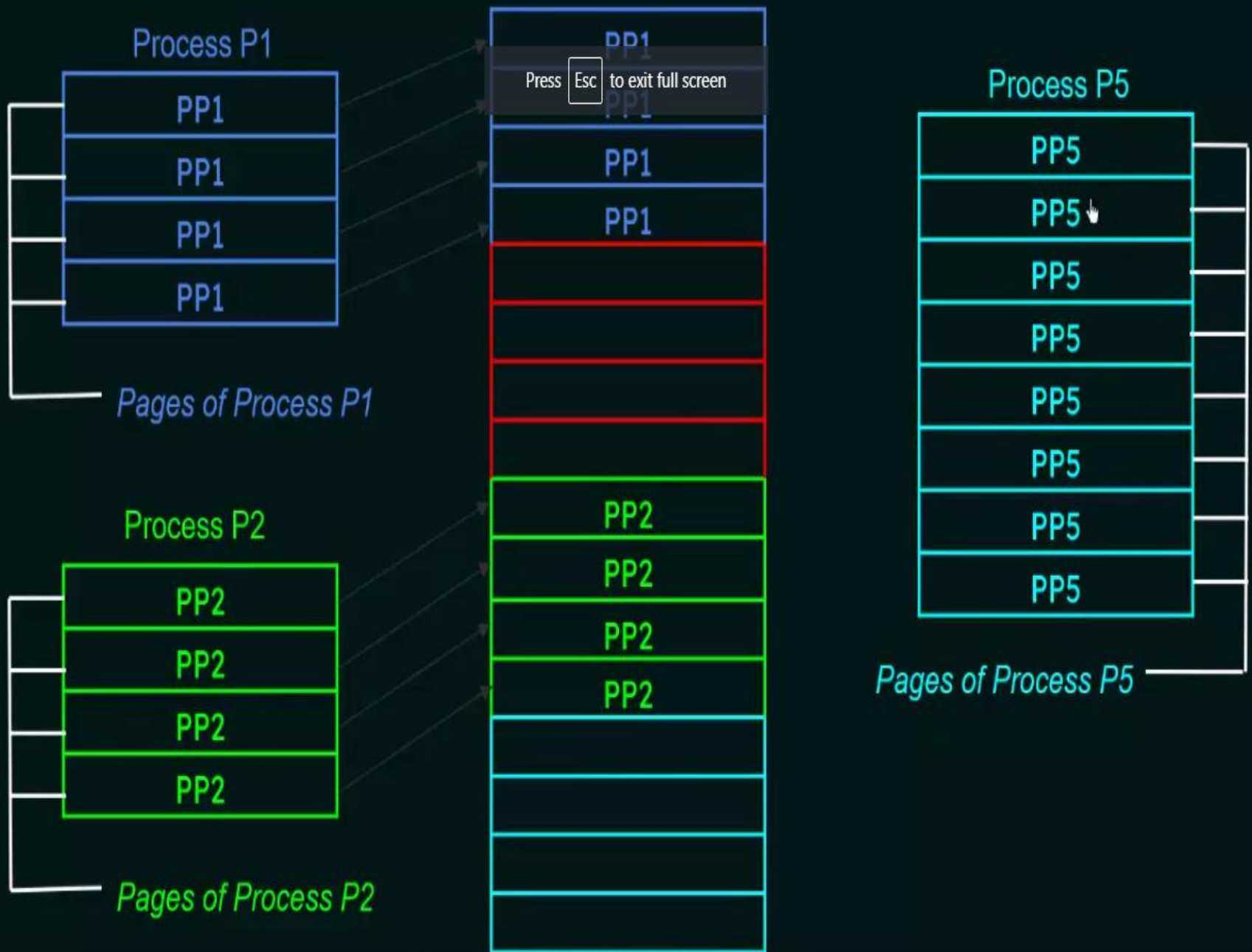


Physical memory divided into frames

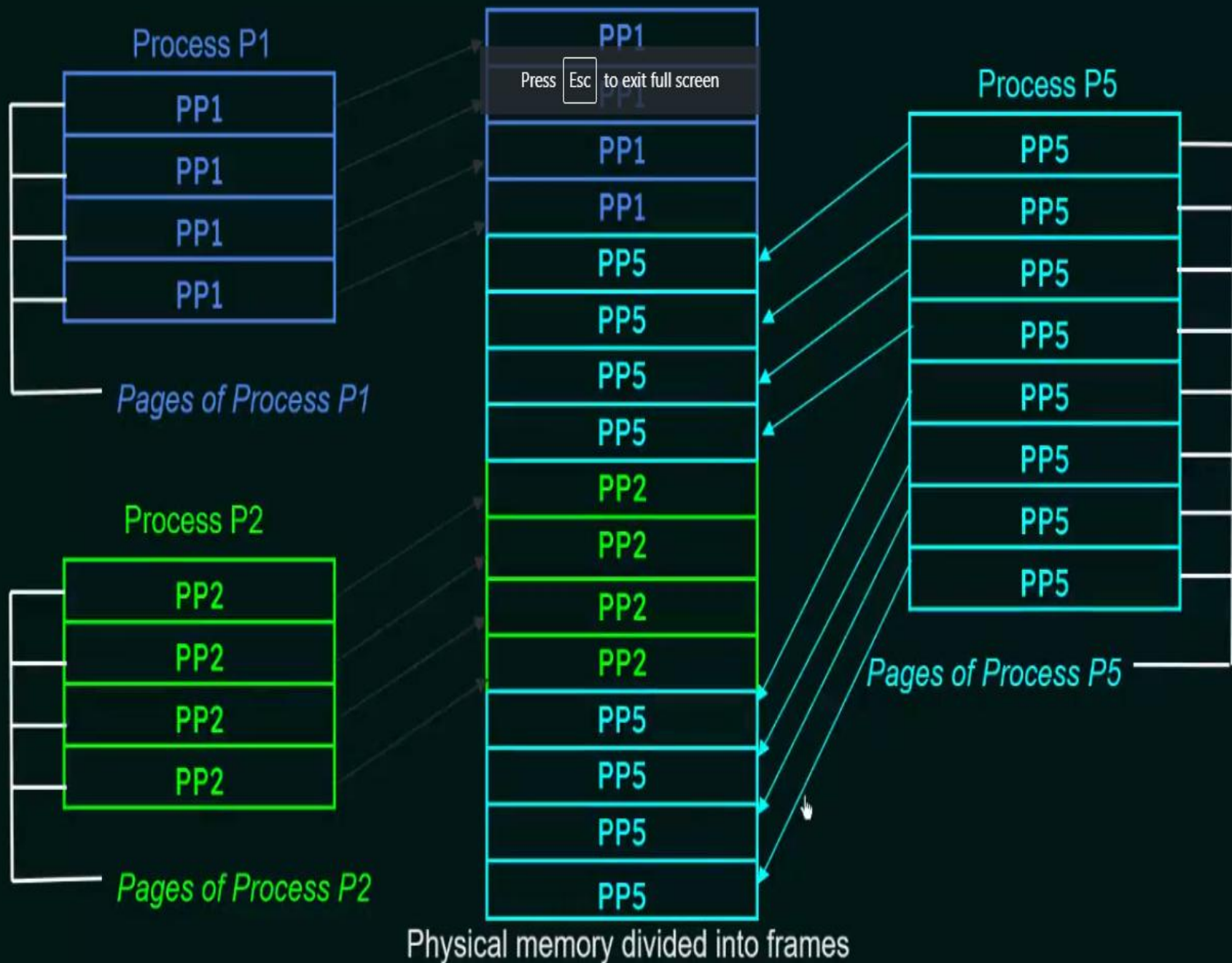


A vertical stack of 16 empty rectangular frames representing physical memory.





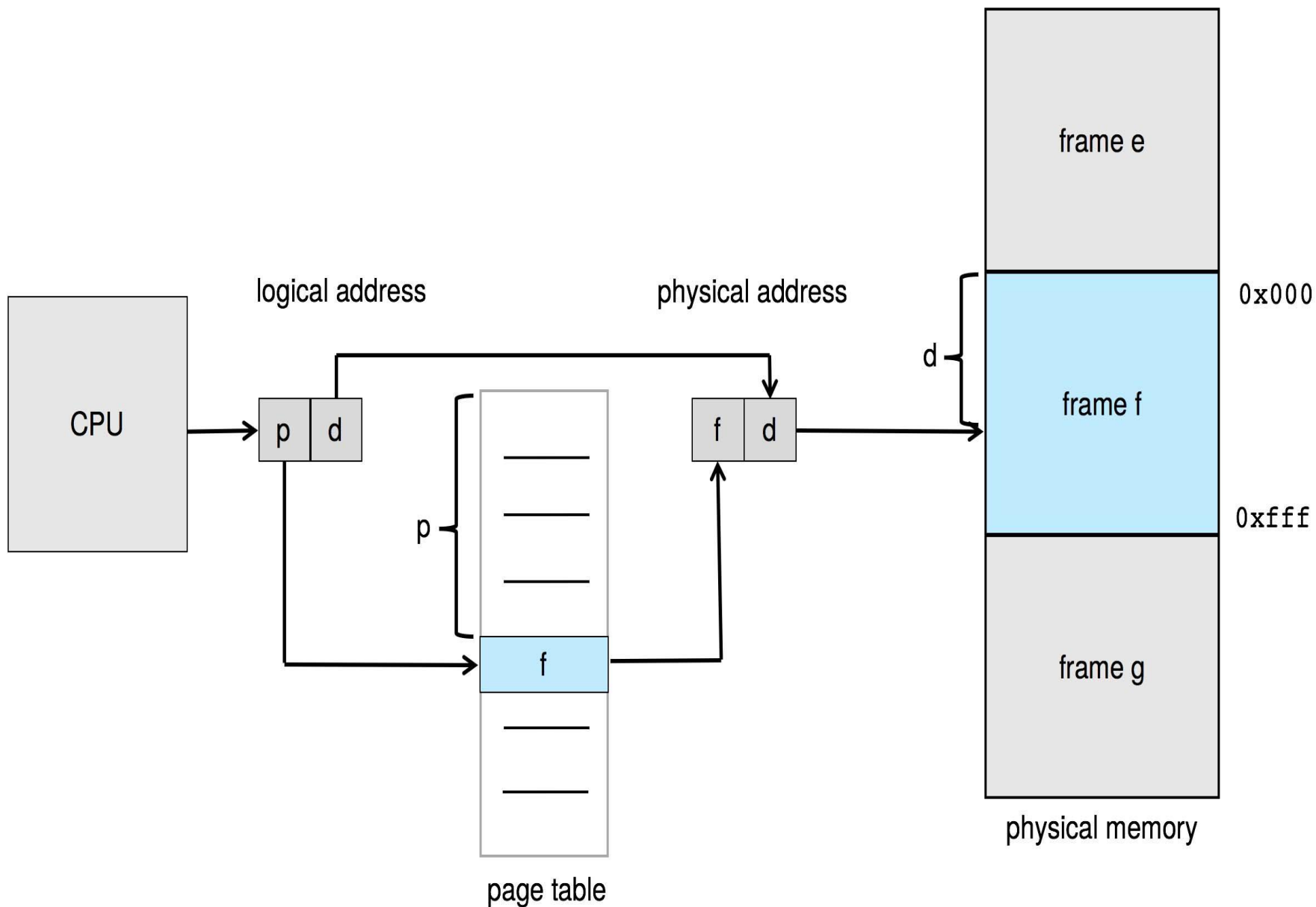
Physical memory divided into frames



Address Translation Scheme

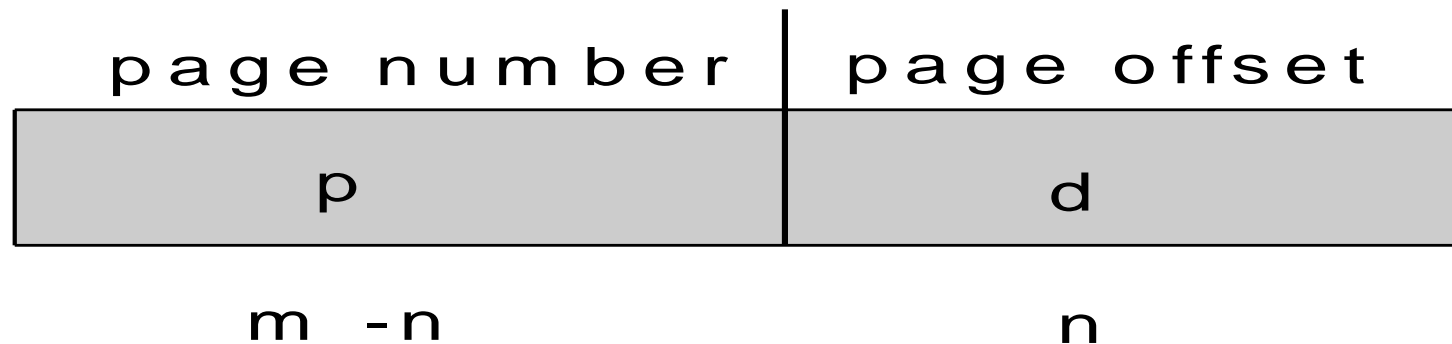
- Address generated by CPU is divided into:
 - **Page number (p)** – used as an index into a **page table** which contains base address of each page in physical memory
 - **Page offset (d)** – combined with base address to define the physical memory address that is sent to the memory unit

Paging Hardware



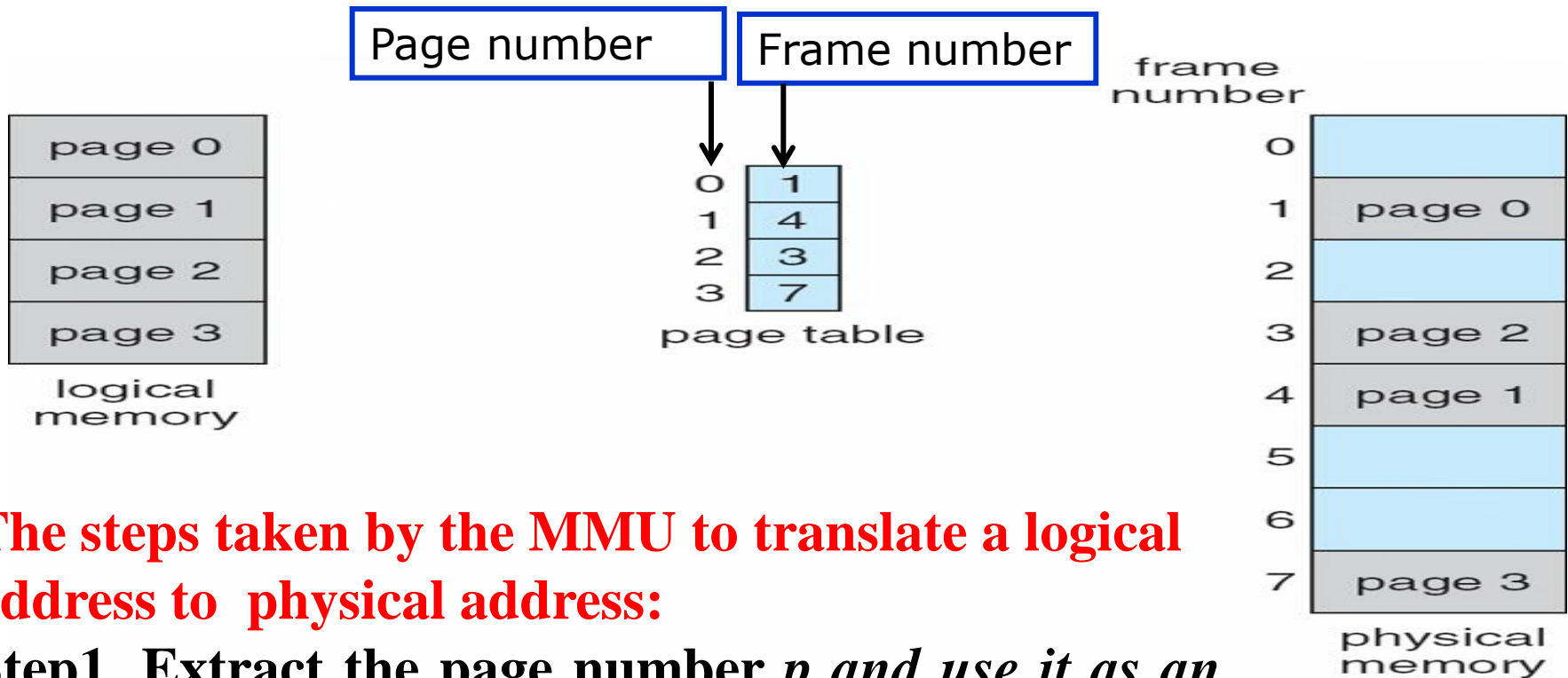
The page size is defined by the hardware. The size of a page is a power of 2, typically varying between 4 KB and 1 GB per page, depending on the computer architecture. The selection of a power of 2 as a page size makes the translation of a logical address into a page number and page offset particularly easy.

If the size of the logical address space is 2^m , and a page size is 2^n bytes, 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**. Thus, the logical address is as follows: where p is an index into the page table and d is the displacement within the page.



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

Paging Model of Logical and Physical Memory



The steps taken by the MMU to translate a logical address to physical address:

Step1. Extract the page number p and use it as an index into the page table.

Step2. Extract the corresponding frame number f from the page table.

Step3. Replace the page number p in the logical address with the frame number f .

Translation of Logical address into Physical address using Page Table

- Logical address: $n = 2$ (page size of 4 bytes) and $m = 4$ (16 addresses).
Using a page size of 4 bytes and a physical memory of 32 bytes (8 pages)

Page 0	0	a
	1	b
	2	c
	3	d
Page 1	4	e
	5	f
	6	g
	7	h
Page 2	8	i
	9	j
	10	k
	11	l
Page 3	12	m
	13	n
	14	o
	15	p

logical memory

0	5
1	6
2	1
3	2

page table

Frame 0	0	
Frame 1	4	i j k l
Frame 2	8	m n o p
Frame 3	12	
Frame 4	16	
Frame 5	20	a b c d
Frame 6	24	e f g
Frame 7	28	

physical memory

The programmer's view of memory can be mapped into physical memory as, Logical address 0 is page 0, offset 0.

Logical address into Physical address is calculated as ,

Physical address= [(frame \times page size)+ Offset]

Indexing into the page table, find that page 0 is in frame 5. So, logical address 0 maps to physical address 20 [= (5 \times 4) +0].

Logical address 3 (page 0, offset 3) maps to physical address 23 [= (5 \times 4) + 3].

Logical address 4 is page 1, offset 0; according to the page table, page 1 is mapped to frame 6. Thus, logical address 4 maps to physical address 24 [= (6 \times 4) + 0].

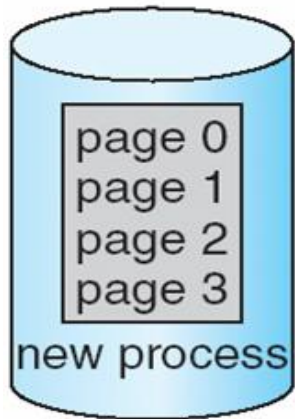
Paging -- Calculating internal fragmentation

- Page size = 2,048 bytes ; suppose Process size = 72,766 bytes
- 35 pages + 1,086 bytes
- Internal fragmentation of $2,048 - 1,086 = 962$ bytes
- In **the worst case, a process would need n pages plus 1 byte.**
- It would be allocated $n + 1$ frames, resulting in internal fragmentation of almost an entire frame.
- But each page table entry takes memory to track
- If process size is independent of page size, expect internal fragmentation to average one-half page per process. This consideration suggests that small page sizes are desirable. However, overhead is involved in each page table entry, and this overhead is reduced as the size of the pages increases.
- Page sizes growing over time; Solaris supports two page sizes – 8 KB and 4 MB- **larger page size called huge pages.**

Free Frames

free-frame list

14
13
18
20
15

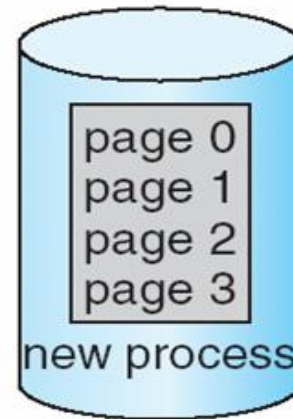


(a)

Before allocation

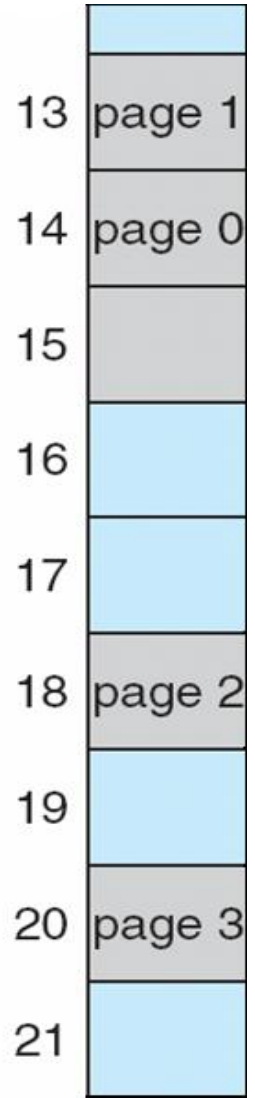
free-frame list

15



0	14
1	13
2	18
3	20

new-process page table



(b)

After allocation

- When a process arrives in the system to be executed, its size, expressed in pages, is examined.
- 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.
- The next page is loaded into another frame, its frame number is put into the page table, and so on.
- The operating system is managing physical memory, it must be aware of the allocation details of physical memory—which frames are allocated, which frames are available, how many total frames there are, and so on.
- This information is generally kept in a single, system-wide data structure called a **frame table**. **The frame table has one entry for each physical page frame**, indicating whether the latter is free or allocated and, if it is allocated, to which page of which process (or processes).

Hardware Support

Page tables are per-process data structures, a pointer to the page table is stored in the register (like the instruction pointer) in the process control block of each process.

When the CPU scheduler selects a process for execution, it must reload the user registers and the appropriate hardware page-table values from the stored user page table.

Implementation of Page Table

Method 1: The page table is implemented as a set of dedicated high-speed hardware registers, which makes the page-address translation very efficient.

Drawback: The use of registers for the page table is satisfactory if the page table is reasonably small (for example, 256 entries). Most contemporary CPUs, support much larger page tables (for example, 2^{20} entries). For these machines, the use of fast registers to implement the page table is not feasible.

Method 2 : keep the Page table in main memory, **Page-table base register (PTBR)** points to the page table and **Page-table length register (PTLR)** indicates size of the page table

Drawback: In this scheme every data/instruction access requires two memory accesses - One for the page table and one for the data / instruction

Method 3 : The two-memory access problem can be solved by the use of a special **fast-lookup hardware cache called translation look-aside buffers (TLBs)** (also called associative memory).

Translation Look-Aside Buffer

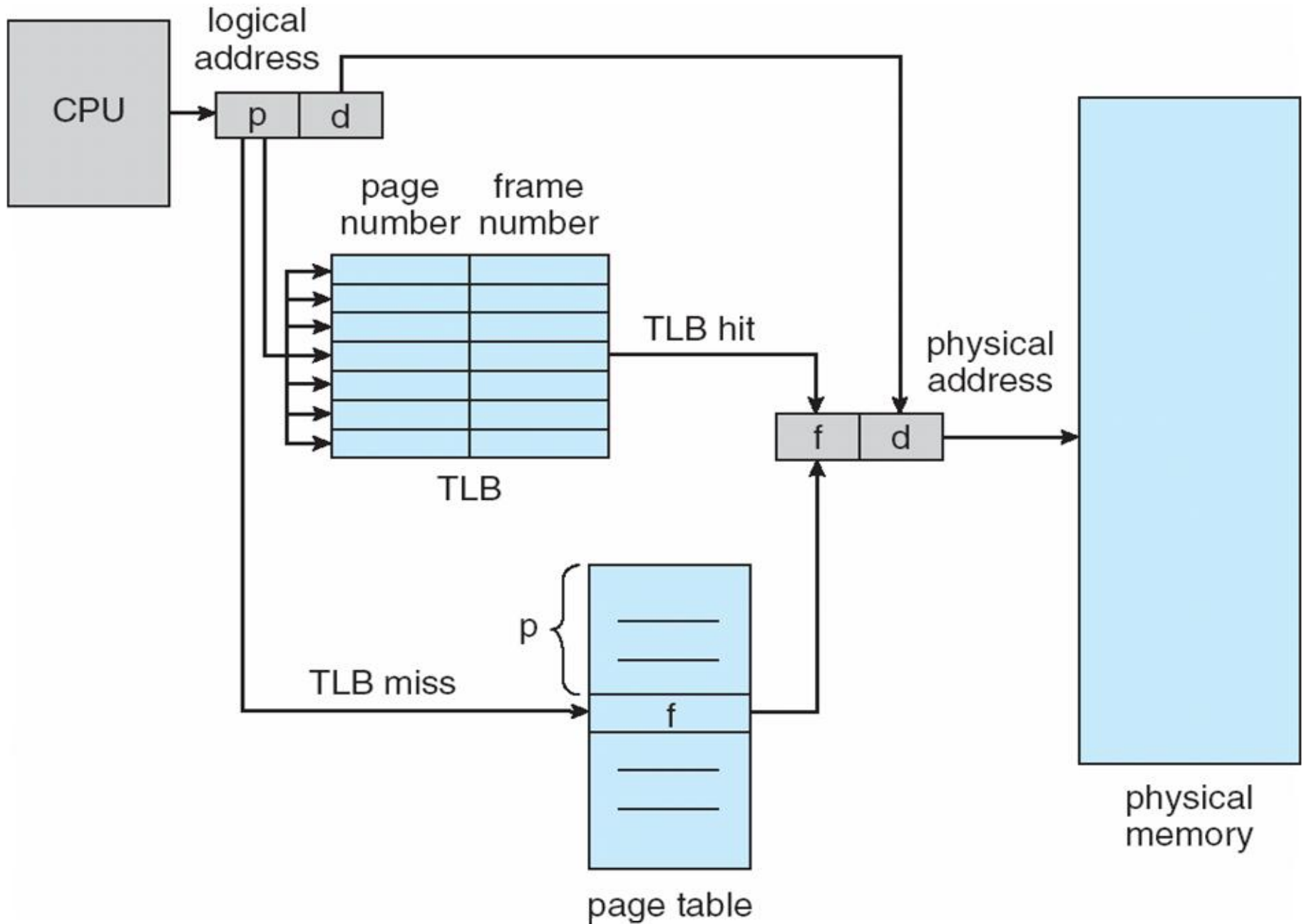
- TLB is associative, high speed memory
- It consists of two parts- Key and value
- Some TLBs store **address-space identifiers (ASIDs)** in each TLB entry – uniquely identifies each process
- When the associative memory is presented with an item, the item is compared with all keys simultaneously
- If the item is found, the corresponding value field is returned and the search is fast
- TLBs typically small (64 to 1,024 entries)
- When a logical address is generated by the CPU, its page number is presented to the TLB
- If the page number is found, its frame number is immediately available and is used to access memory → **TLB Hit**

Translation Look-Aside Buffer(Continued)

- If the page number is not in the TLB → **TLB Miss** ; A memory reference to the page table must be made. When the frame number is obtained, use it to access memory. Add the page and frame number to TLB, so that it will be found quickly on the next reference.
- Replacement policies must be considered
- Some entries can be **wired down** for permanent fast access
- Address translation (p, d)
 - If p is in associative register, get frame # out
 - Otherwise get frame # from page table in memory

Page #	Frame #

Paging Hardware With TLB



Effective Access Time

- Hit ratio – percentage of times that a page number is found in the TLB
- An 80% hit ratio means that , the desired page number in the TLB is 80% of the time.
- Suppose that 10 nanoseconds to access memory.
 - If the desired page in TLB then a mapped-memory access take 10 ns
 - Otherwise , need two memory access so it is 20 ns

- **Effective Access Time (EAT)**

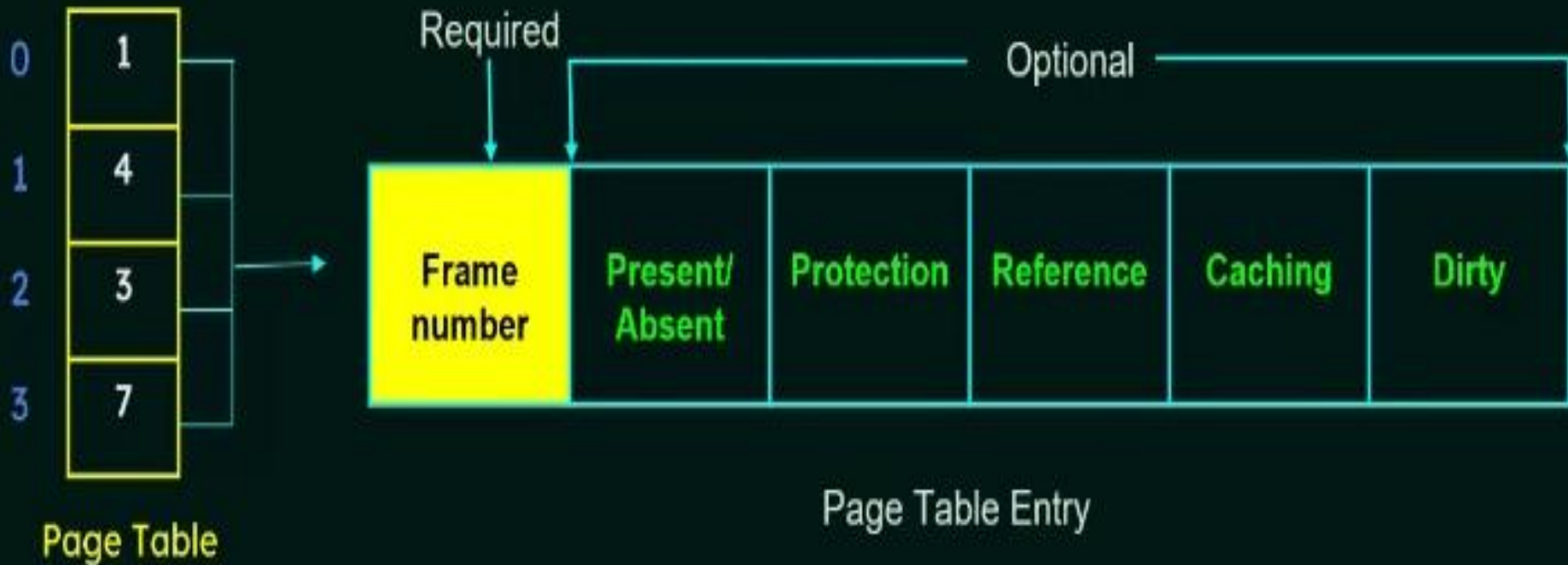
$EAT = 0.80 \times 10 + 0.20 \times 20 = 12 \text{ ns}$ implying 20% slowdown in access time

- Consider a more realistic hit ratio of 99%,

$EAT = 0.99 \times 10 + 0.01 \times 20 = 10.1\text{ns}$ implying only 1% slowdown in access time.

Page Table Entry

- Page table entries contains several information about pages which vary from OS to OS
- The most important information in a page table entry is the frame number.
- The remaining information are optional



➤ Frame number denotes the frame where the page is present in the main memory. The number of bits required for this depends on the number of frames in the main memory

Memory Protection

❑ Present/Absent bit specifies whether the page is present in the main memory or not. It is also called Valid/Invalid bit.

❑ If the page we are looking for is not present in main memory, it is called PAGE FAULT. Also the present/Absent bit is set to '0').

❖ The protection bit known as Read/Write bit is used for page protection.

❖ It specifies the permissions for read or write operations on the page

❖ The bit is set to '0' if only read operation is allowed and set to '1' if both read and write operations are allowed

✓ The Reference bit specifies whether the page has been referenced in the last clock cycle or not.

✓ It is set to '1' when the page is accessed

- The **caching bit is used for enabling or disabling caching of the page.**
- When always fresh data is needed disable caching so as to avoid fetching of old data from the cache
- When caching has to be disabled , this bit is set to '1'. Otherwise it is set to '0'.

- **Dirty bit is also known as Modified bit. It specifies whether the page has been modified or not.**
- If the page has been modified, the this bit is set to '1' , otherwise it is '0'
- This bit helps in avoiding unnecessary writes to the secondary memory when a page is replaced by another page.

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

00000	page 0
	page 1
	page 2
	page 3
	page 4
10,468	page 5
12,287	

frame number		valid-invalid bit
0	2	v
1	3	v
2	4	v
3	7	v
4	8	v
5	9	v
6	0	i
7	0	i

page table

0	
1	
2	page 0
3	page 1
4	page 2
5	
6	
7	page 3
8	page 4
9	page 5
	⋮
	page n

Shared Pages

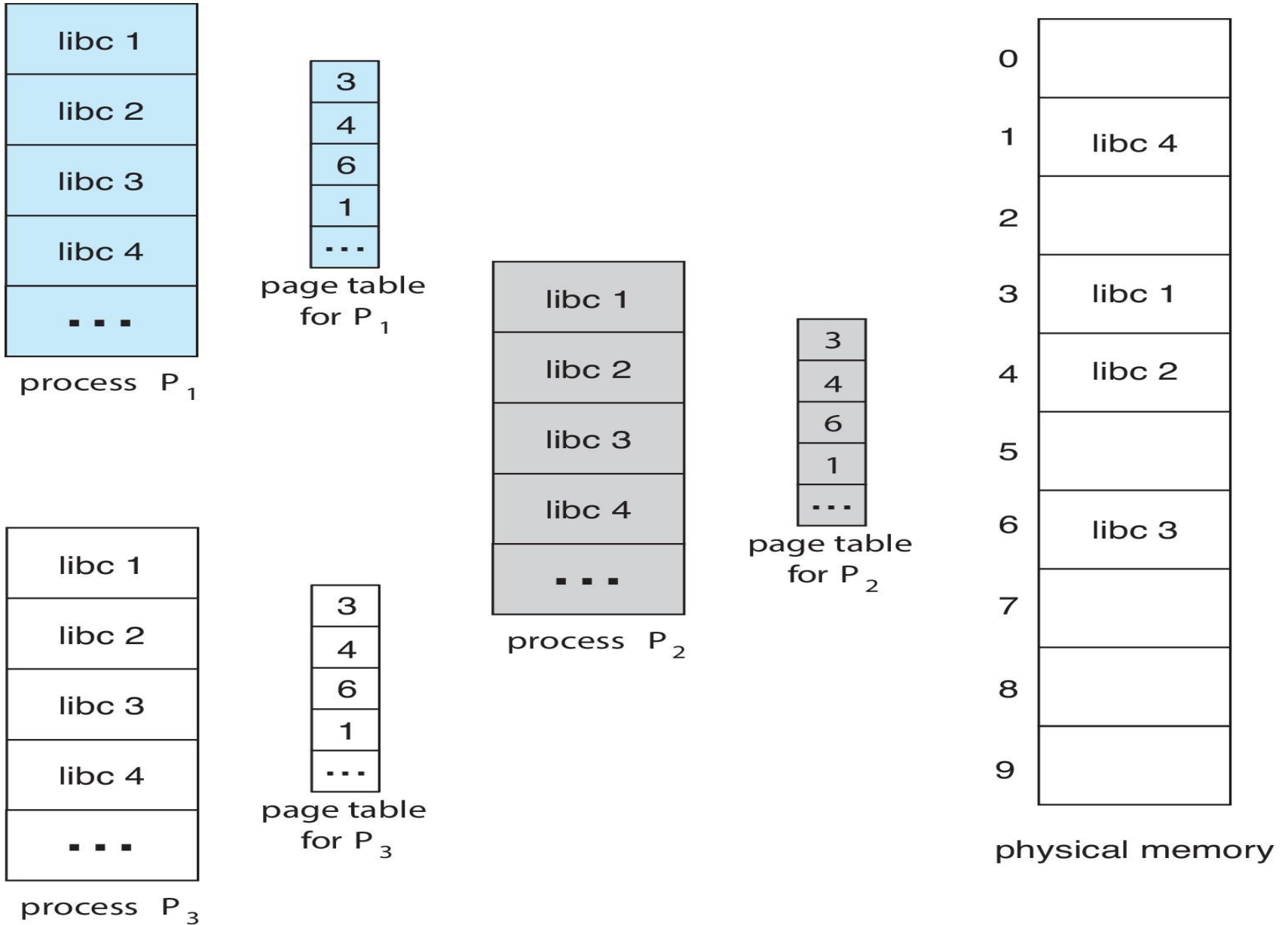
■ Shared code

- One copy of read-only (**reentrant** code or pure code) shared among processes (i.e., text editors, compilers, window systems)
- 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.
- 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



Advantage of using shared Pages

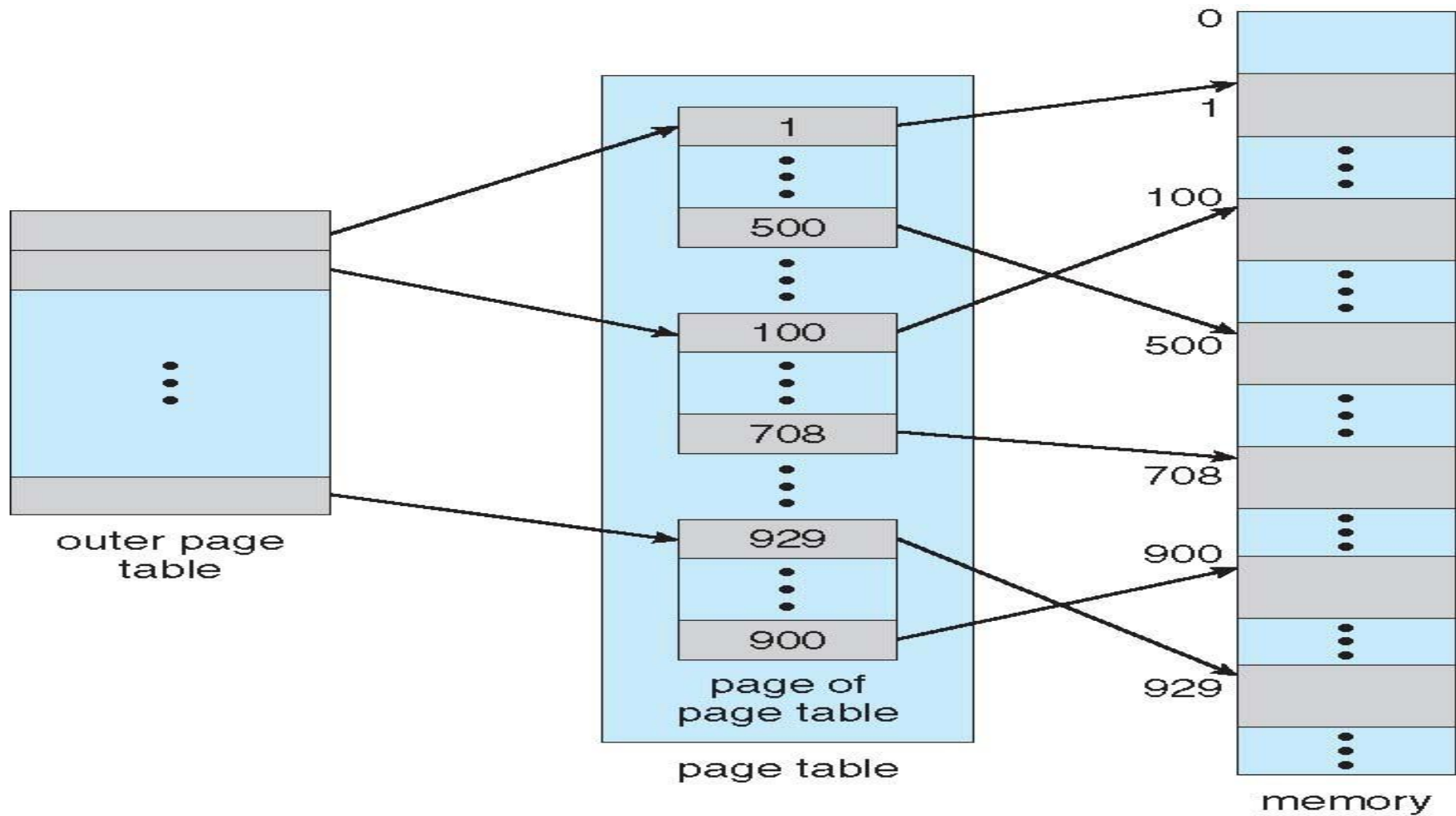
- Only one copy of the standard C library need be kept in physical memory, and the page table for **each user process maps onto the same physical copy of libc.**
- Thus, to support **40 processes, it need only one copy of the library, and the total space now required is 2 MB instead of 80 MB—a significant saving in physical memory**
- In addition to run-time libraries such as libc, other heavily used programs can also be shared—compilers, window systems, database systems

Structure of the Page Table

- Memory structures for paging can get huge.
 - 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 ($2^{32} / 2^{12} = 2^{20} = 1\text{MB}$)
 - If each entry is 4 bytes → each process 4 MB of physical address space for the page table alone
 - ▶ Don't want to allocate that contiguously in main memory
 - One simple solution is to divide the page table into smaller units. This division is done in 3 ways
 - ▶ Hierarchical Paging
 - ▶ Hashed Page Tables
 - ▶ Inverted Page Tables

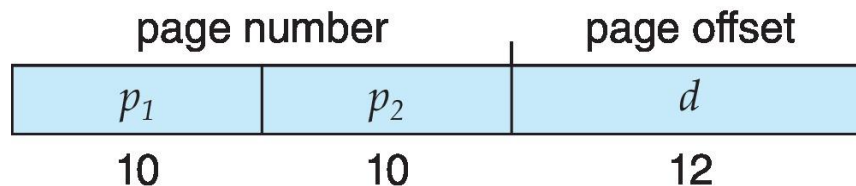
Hierarchical Page Tables

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



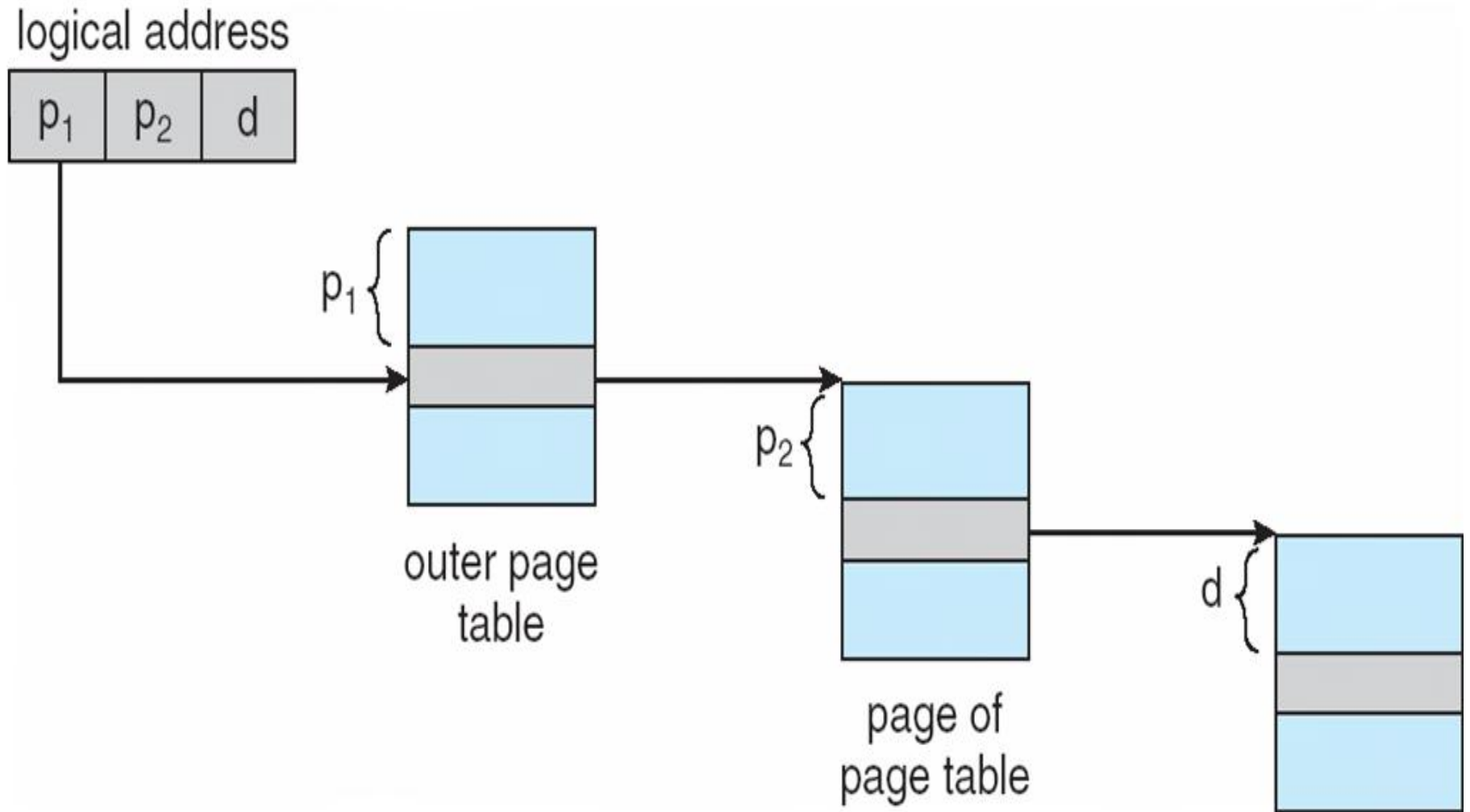
Two-Level Paging Example

- 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:



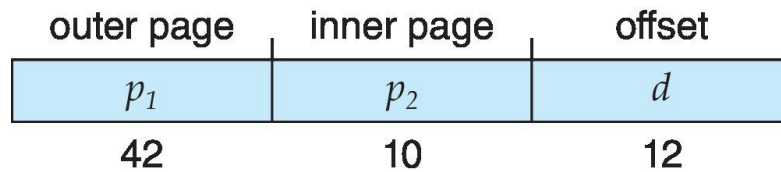
- where p_1 is an index into the outer page table, and p_2 is the displacement within the page of the inner page table
- Known as **forward-mapped page table**

Address-Translation Scheme



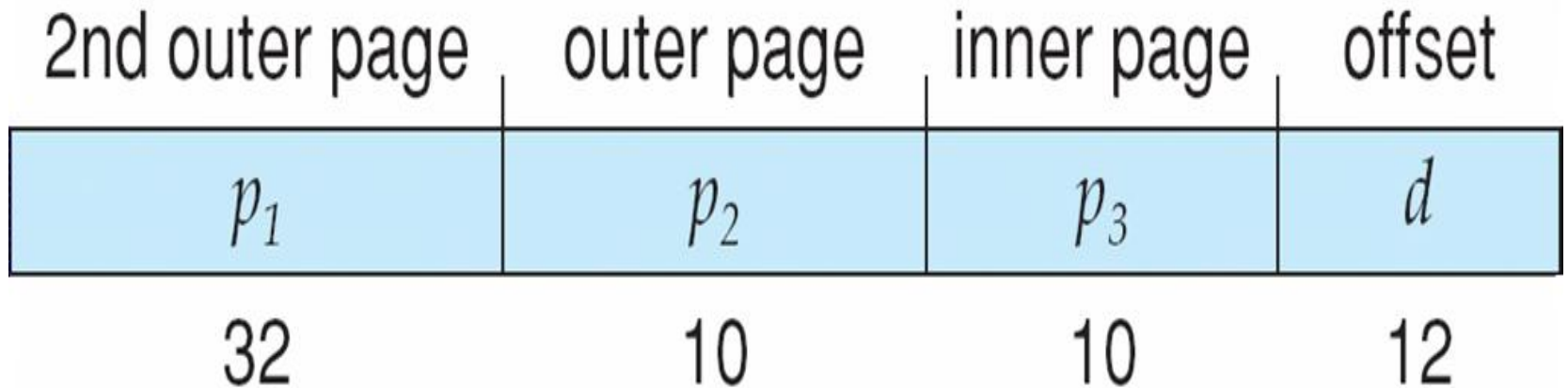
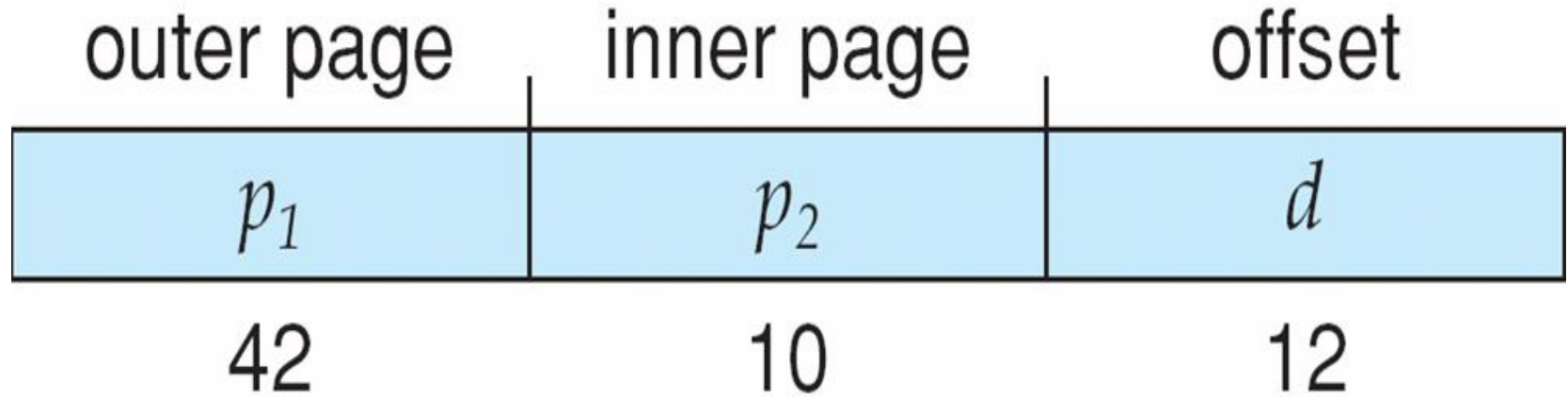
64-bit Logical Address Space

- 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} of 4byte entries
 - Address would look like



- Outer page table has 2^{42} entries – Large in size; One solution is to add a 2nd outer page table
- And possibly 4 memory access to get to one physical memory location

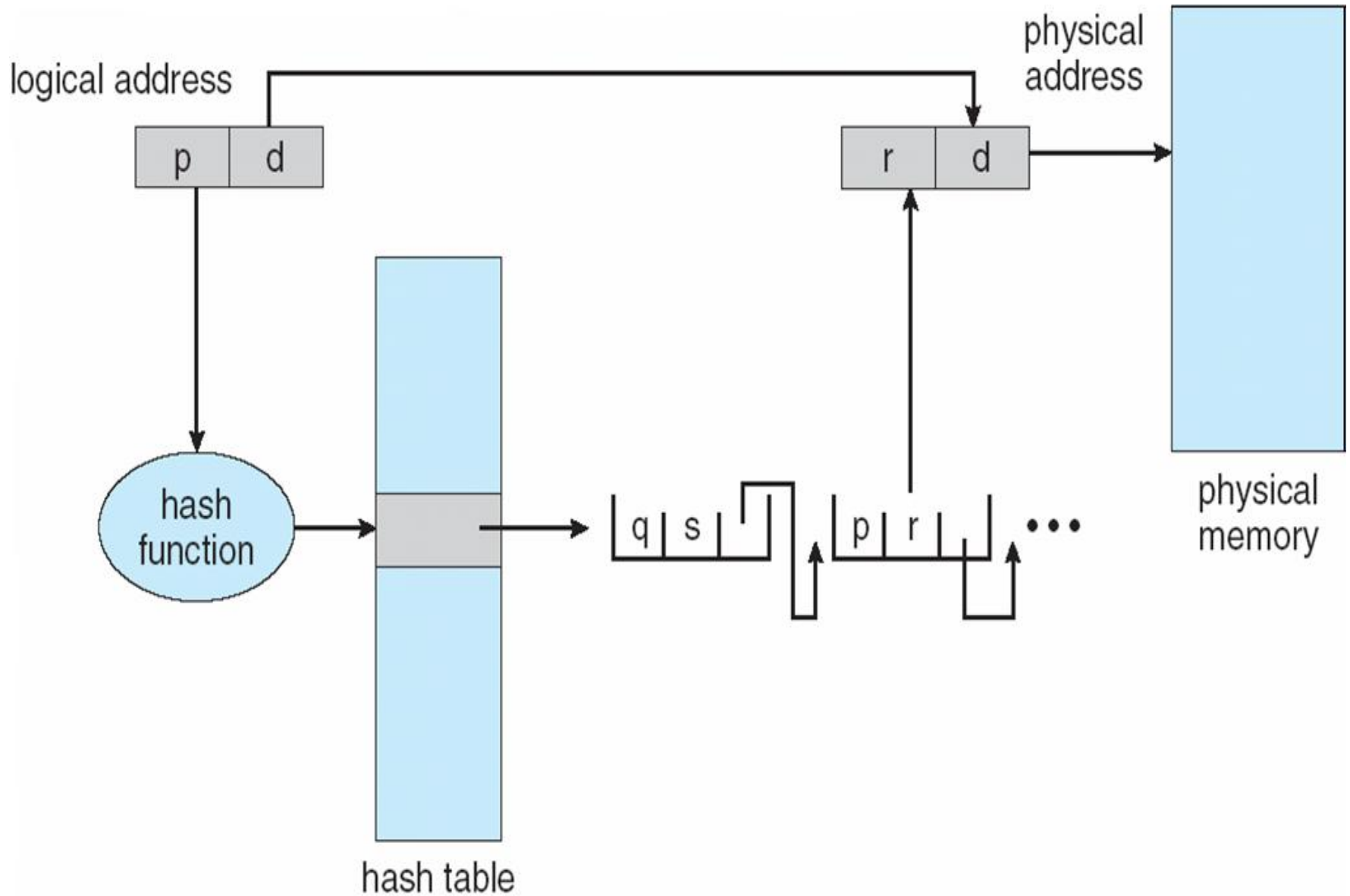
Three-level Paging Scheme



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**
- The algorithm works as follows: The virtual page number in the virtual address is hashed into the hash table. The virtual page number is compared with field 1 in the first element in the linked list. If there is a match, the corresponding page frame (field 2) is used to form the desired physical address. If there is no match, subsequent entries in the linked list are searched for a matching virtual page number.
- 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 Table



Inverted Page Table

- In most operating system, a **separate page table is maintained for each process**
- So, for '**n**' number of processes, there will be '**n**' number of page tables.
- For large processes, there would be many pages and for maintaining information about these pages, there would be too many entries in their page tables which itself would occupy a lot of the memory
- **Hence memory utilization is not efficient as a lot of memory is wasted in maintaining page tables itself**
- **Solution to memory utilization is Inverted page tables.**

Inverted Page Table(continued)

- An inverted page table has one entry for each real page(or frame) of memory
- Each entry consists of the virtual address of the page stored in that real memory location, with information about the process that owns that page
- Thus, only one page table is in the system, and it has only one entry for each page of physical memory

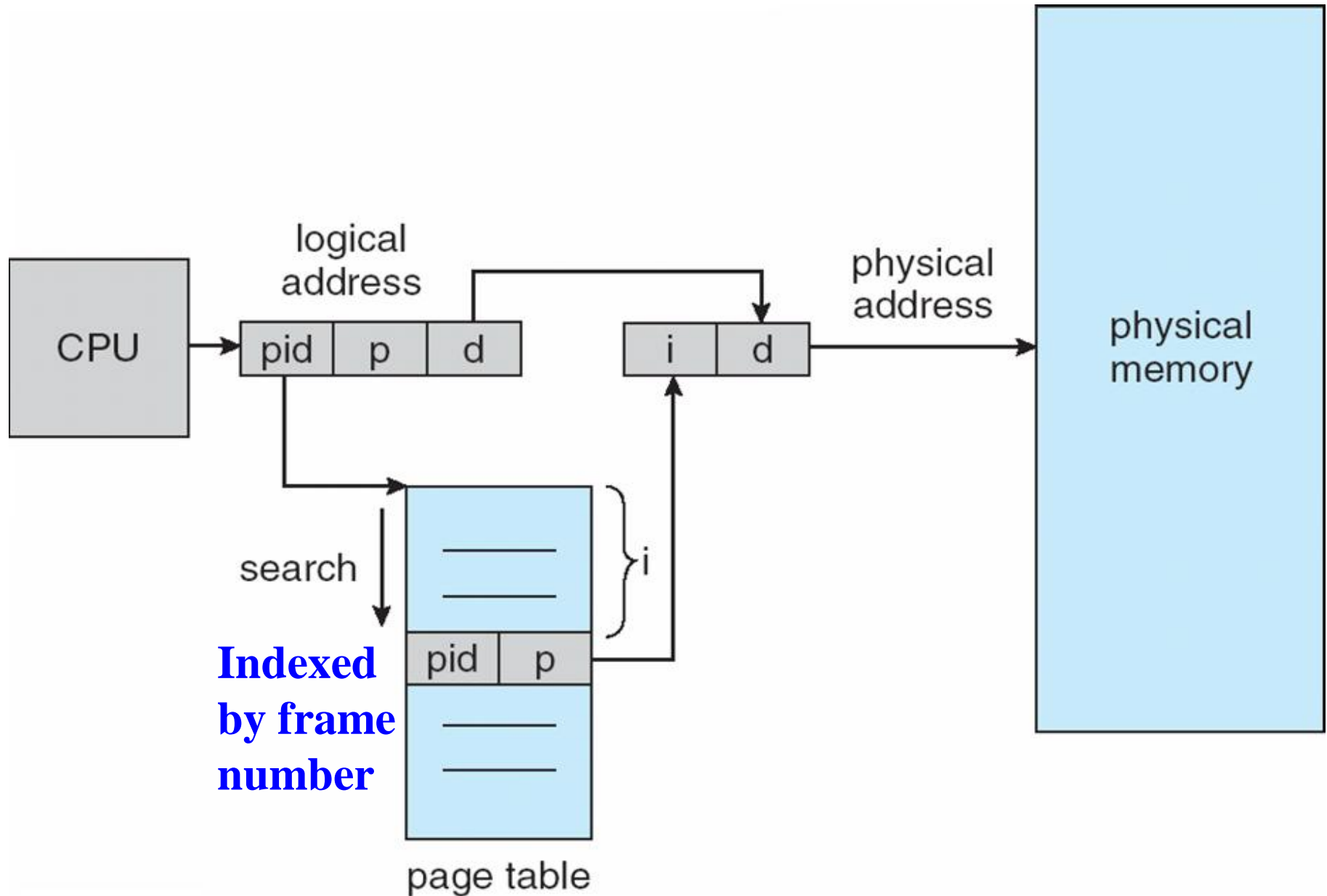
Each virtual address consists of a triple fields

Process-id	Page number	Offset
-------------------	--------------------	---------------

Each inverted Page table entry is a pair of fields

Process-id	Page number
-------------------	--------------------

Inverted Page Table Architecture



Working:

- When a memory reference occurs, part of the virtual address, consisting of $\langle \text{process-id, pagenumber} \rangle$, is presented to the memory subsystem.
- The inverted page table is then searched for a match.
- If a match is found—say, at entry i —then the physical address $\langle i, \text{offset} \rangle$ is generated.
- If no match is found, then an illegal address access has been attempted.

Advantage : Reduces memory usage

Disadvantages: 1. Increased search time as inverted page table is sorted by physical address, but lookups on virtual addresses. The entire table need to be searched for a match.

2. Difficulty in implementing shared memory

Solutions : 1. Use hash table to limit the search to one — or at most a few — page-table entries 2. TLB can accelerate access

Solution to implement shared memory is One mapping of a virtual address to the shared physical address

Problem 1: Suppose in a three-level paging scheme with 64-bit logical address space, and the page size is 4KB. The second outer page table size is 16MB, the outer page table size is 4MB, and the inner page table is 4KB. Assuming that each entry in the page table consists of 4 bytes. How many bits are required for the second outer page table, outer page table, inner page table, and page offset?

- (A) 32, 10, 10, 12 **(B) 22, 20, 10, 12** (C) 42, 05, 05, 12
(D) 32, 15, 05, 12 (E) 12, 20, 20, 12

The page size is $4\text{KB} = 2^{12}$ bytes. So, the offset is 12. The inner page table size is $4\text{KB} = 2^{10} \times 2^2$ bytes. So it needs 10 bits (every entry is of 4 Bytes = 2^2). The outer page table size is $4\text{MB} = 2^{20} \times 2^2$ bytes. So it needs 20 bits. The second outer page table size is $16\text{MB} = 2^{22} \times 2^2$ bytes. So it needs 22 bits.

Problem 2: Suppose in a three-level paging scheme with 64-bit logical address space, and the page size is 4KB. The second outer page table size is 16MB, the outer page table size is 4MB, and the inner page table is 4KB. Assuming that each entry in the page table consists of 4 bytes. If 100ns is needed for memory access, what is the Effective Access Time (EAT) if the hit ratio is 80% (The percentage of times that a particular page number is found in the TLB)?

- (A) 300ns **(B) 160ns** (C) 100ns (D) 350ns (E) 400ns

Here, a three-level paging scheme is used. Therefore, the EAT is
$$0.8 \times 100 + (1 - 0.8) \times (4 \times 100) = 160\text{ns}.$$

Problem 3: Suppose a computer system with a 256 bytes main memory runs a process with four unequal size segments. The process is loaded in the main memory using the following segment table.

Segment number	Base Address (in decimal)	Length (in bytes)
0	31	32
1	79	16
2	95	32
3	127	128

If the computer system uses the 9-bit addresses, what is the physical address corresponding to the logical address 110011001?

(A) 10011000 (B) 10011100 (C) 11100100 (D) 11110100

Total number of segments is $4 = 2^2$. Therefore, the initial 2 bits (the two most significant bits) represent the segment number. If the logical address is 110011001, then the segment number is 11, i.e., 3. The remaining 7 bits, i.e., 0011001 map to the actual physical address. The decimal equivalent of 0011001 is 25. Segment 3 has a base address of 127 (decimal). Therefore, 0011001 represents the physical address $127+25 = 152$, i.e., 10011000.

Problem 4: In the context of the memory management process, the "valid/invalid" bit is used to

- (A) signify whether the CPU is currently in a valid operational state
- (B) determine if a CPU instruction is valid or invalid
- (C) denote whether a particular page or memory segment is currently in the CPU cache
- (D) indicate whether a page or memory segment is currently in main memory or not**
- (E) represent the status of the arithmetic logic unit

Problem 5: Assume that a computer system uses hashed page tables for memory management. The system has a virtual address space of 16 bits and a physical address space of 12 bits. The hash function used is:

$$\text{hash}(\text{virtual_page_number}) = (\text{virtual_page_number} + 3) \bmod 17$$

What is the size of the hash table (number of entries), and each entry in the hash table (in bits)?

- (A) Hash table size: 15 entries, Entry size: 28 bits
- (B) Hash table size: 17 entries, Entry size: 24 bits

- (C) Hash table size: 19 entries, Entry size: 12 bits
- (D) Hash table size: 20 entries, Entry size: 20 bits
- (E) Hash table size: 17 entries, Entry size: 12 bits**

The number of entries in the hash table is 17 (0 to 16, since the mod 17 is used). Each entry in the hash page table is the address of a physical page. Since the physical address space is 12 bits, we need 12 12-bit entries in the hash page table.

Problem 6: Assume that a computer system uses an inverted page table for memory management. The physical address is the combination of

- (A) index of the page table and offset**
- (B) index of the page table and virtual page number
- (C) process id and virtual page number
- (D) index of the page table and process id
- (E) None of these

Problem 7: Assume that a computer system uses an inverted page table for memory management. The system uses a 16-bit virtual address, and each virtual address is of the form <process id, virtual page number, offset>. The size of process id, virtual page number, and offset are 4 bits, 6 bits, and 6 bits, respectively. Assume that system uses the physical address of 9 bits. The page table of the system is given as follows.

Index (3 bits)	Process id (4 bits)	Virtual page number (6 bits)
000	0000	000000
001	1001	000001
010	0010	000010
011	1011	000011
100	0100	000100
101	1101	000101
110	0110	000110
111	1111	000111

If the CPU generates the virtual address 1101000101111111, what is the corresponding physical address?

- (A) 10111111 (B) 11111111 (C) 00011111
 (D) 11111101 (E) 11110111

The virtual address is 1101000101111111. Therefore, pid is 1101. The virtual page number is 000101, and the offset is 111111. The pid will give the index in the page table. Here, the pid is 1101, so it indicates the index is 101. The physical address is <index || offset>, and so 101111111 is the physical address.

Problem 8: Assume that a computer system uses 256MB of physical memory and 34-bit virtual address space. Assuming that each entry in the page table consists of 4 bytes. If the page size is 4KB, what is the size of the page table?

- (A) 1MB (B) 2MB (C) 3MB (D) 4MB **(E) 16MB**

Page size is 4KB = 2^{12} bytes. The offset is 12 bits. The number of entries in the page table is $2^{(34-12)} = 2^{22}$. If the size of each entry is 4 bytes, then the size of the table is $2^{22} \times 2^2$ bytes = $2^{20} \times 2^4$ bytes = 16MB

Problem 9: Which of the following statements is true for a hierarchical (multi-level) paging scheme?

- (A) It utilizes physical memory efficiently.
- (B) It reduces the number of entries to be traversed during address translation.
- (C) It incurs additional complexity to the page table management process.
- (D) It involves additional memory access overhead.
- (E) All of the above**

Problem 10: What is the maximum size of a segment in the IA-32 architecture's 32-bit protected mode?

- (A) 1 MB
- (B) 1GB
- (C) 4 GB**
- (D) 8 GB
- (E) 16 GB

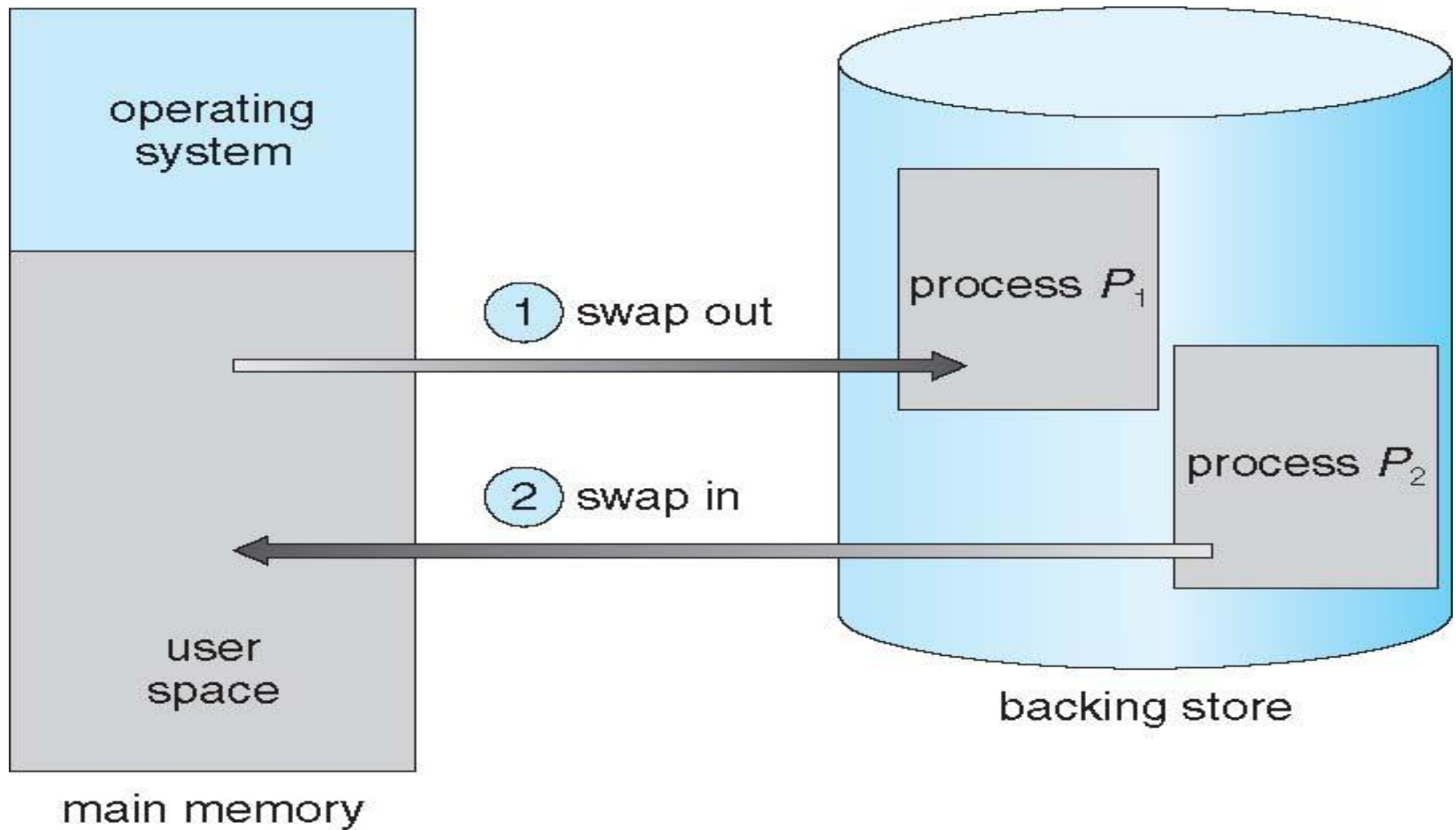
Swapping

- A process must be in memory to be executed.
- A process can be **swapped** temporarily out of memory to a backing store, and then brought **back** into memory for continued execution
 - Total physical memory space of processes can exceed physical memory

[ex]: Round robin CPU scheduling algorithm- When a quantum expires , the memory manager will start to swap out the process and to swap another process into the memory space that has been freed.

- **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

Schematic View of Swapping



Swapping (Cont.)

- Major part of swap time is transfer time; total transfer time is directly proportional to the amount of memory swapped
- System maintains a **ready queue** of ready-to-run processes which have memory images on disk
- Does the swapped out process need to swap back in to same physical addresses? – Normally a process that is swapped out will be swapped back into the same memory space it occupied previously; But it depends on address binding method
 - If **binding is done at assembly or load time** : The process is in same memory location;
 - If **binding is done at execution time**: The process is swapped into a different memory space

Modified versions of swapping are found on many systems (i.e., UNIX, Linux, and Windows)

Swapping normally disabled

Started if more than threshold amount of memory allocated

Disabled again once memory demand reduced below threshold

Context Switch Time including Swapping

- If next processes to be put on CPU is not in memory, need to swap out a process and swap in target process
- Context switch time can then be very high
- 10MB process swapping to hard disk with transfer rate of 40MB/sec
- Swap out time = process size/ Transfer rate = $10\text{MB}/40\text{MB} = 250\text{ ms}$
- For both swap out and swap in , the total swap time(context switching time) is $= 250*2 = 500\text{ms}$
 - Plus swap in of same sized process

Can reduce if reduce size of memory swapped – by knowing how much memory really being used- System calls to inform OS of memory use via `request_memory()` and `release_memory()`

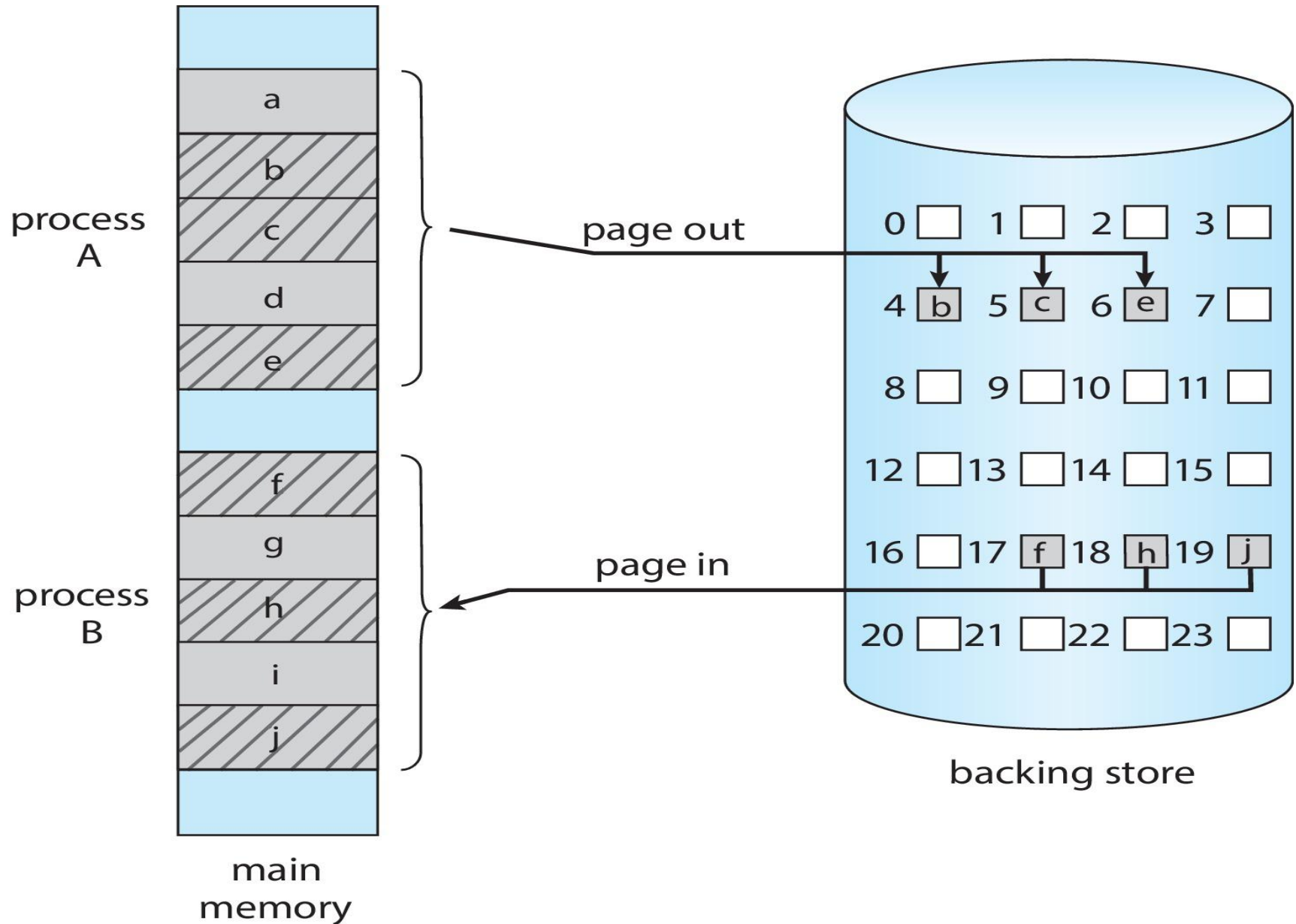
Context Switch Time and Swapping (Cont.)

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

Swapping on Mobile Systems

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

Swapping with Paging

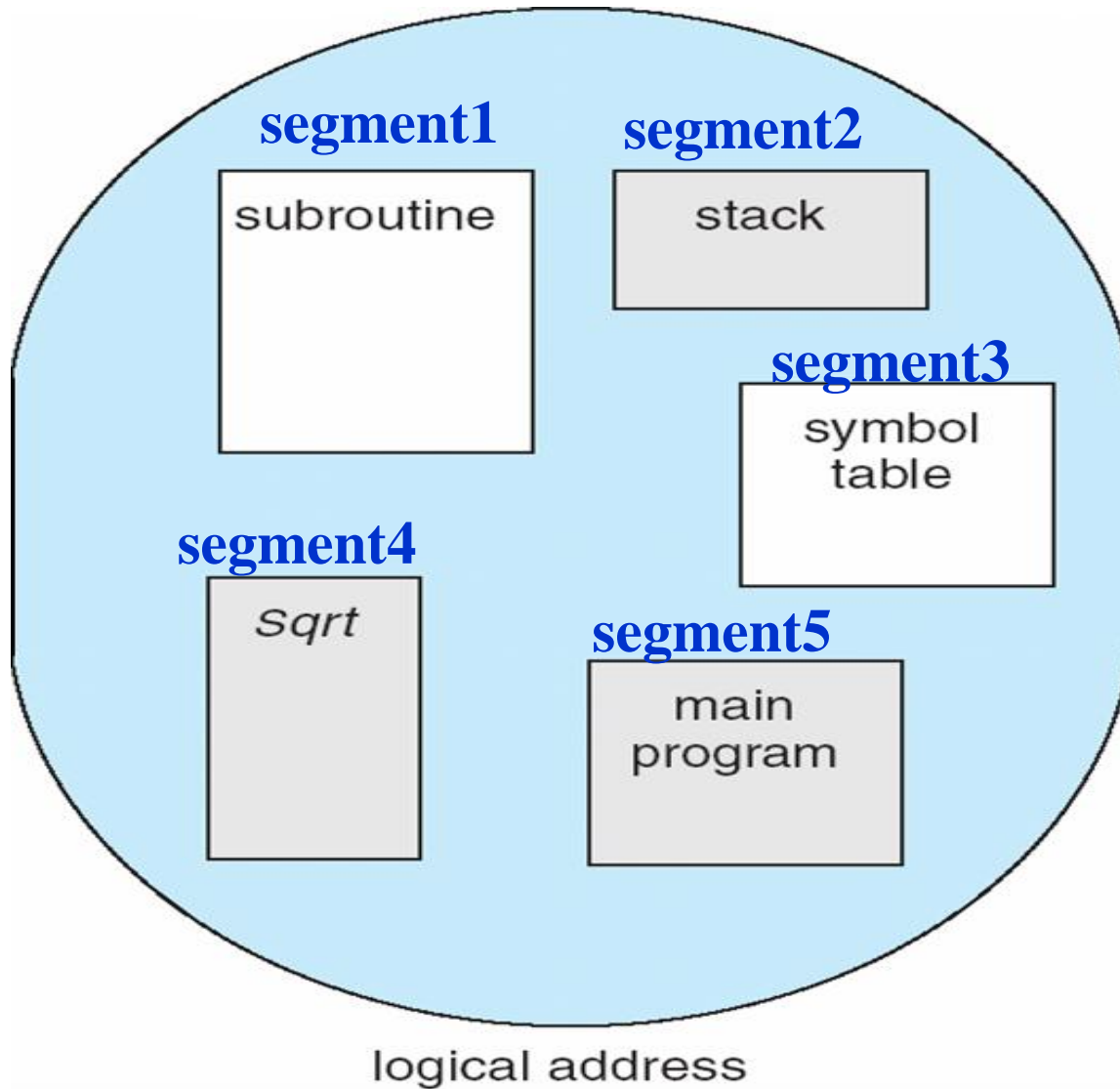


Segmentation

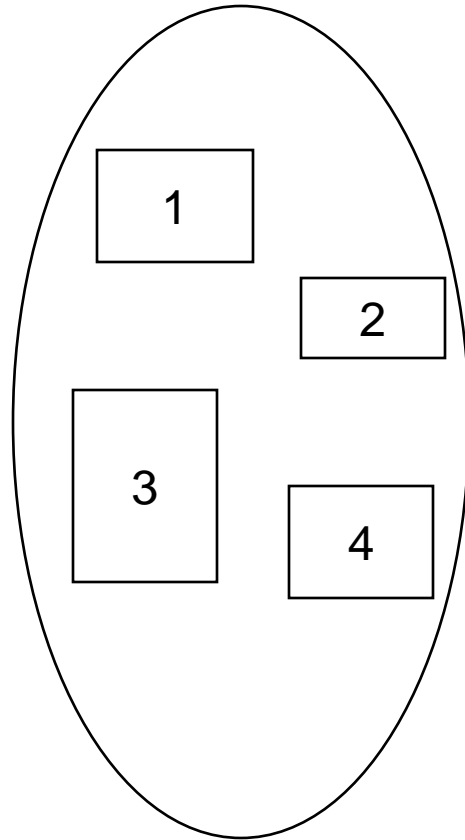
- Memory-management scheme that supports user view of memory
- Segmentation is another non-contiguous memory allocation like paging
- Unlike paging, in segmentation , the processes are not divided into fixed size pages
- Instead , the processes are divided into several modules called segments
 - A segment is a logical unit such as:
main program, procedure, function, method,
object, local variables, global variables,
common block, stack, symbol table, arrays etc

So, both secondary memory and main memory are divided into partitions of unequal sizes

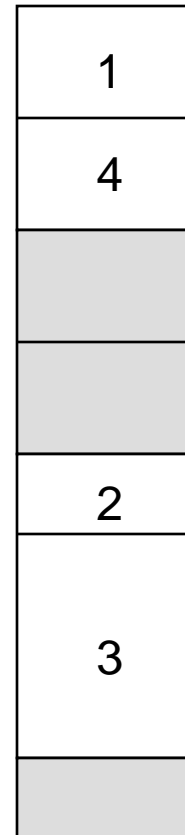
User's View of a Program



Logical View of Segmentation



user space



physical memory space

Segmentation Architecture

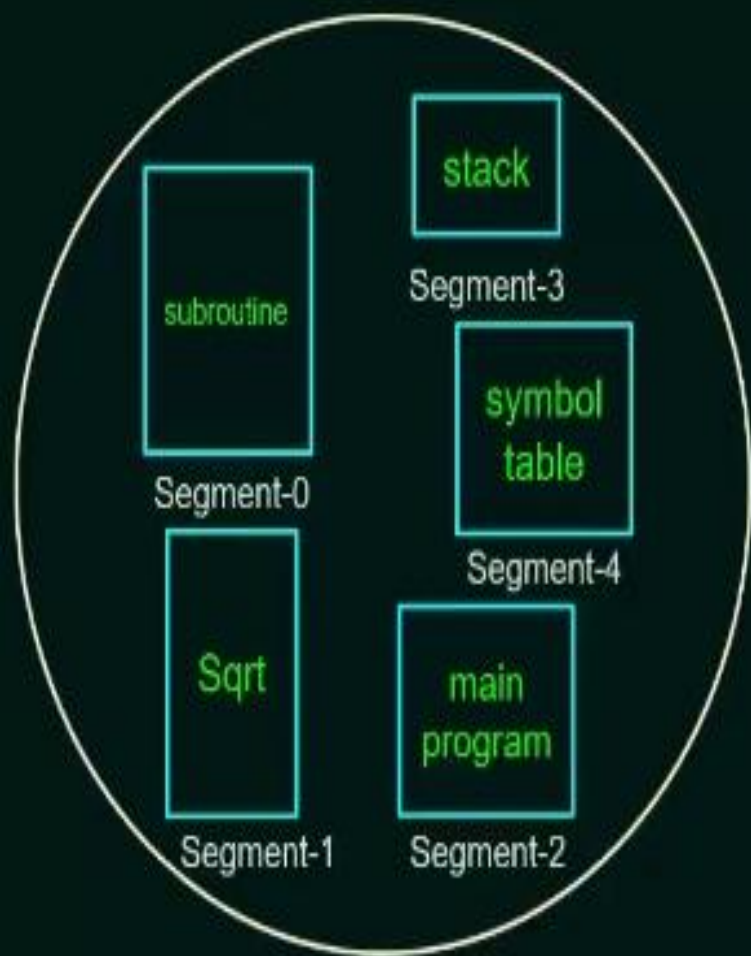
- Logical address is a collection of segments
- Logical address consists of a two tuple:
 $\langle \text{segment-number}, \text{offset} \rangle$,
- **Segment table** – maps two-dimensional user defined addresses into one dimensional physical addresses;
- each table entry has:
 - **base** – contains the 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 $s < \text{STLR}$

- Logical address consists of segment number 's' and an offset into that segment 'd'.
- The segment number is used as an index to the segment table.
- The offset 'd' of the logical address between 0 and the limit.
- When an offset is legal, it is added to the segment base to produce the address in physical memory
- When an offset is beyond the limit, trap to the operating system

	Limit	Base
0	1000	1400
1	400	6300
2	400	4300
3	1100	3200
4	1000	4700

Segment table



Logical address space

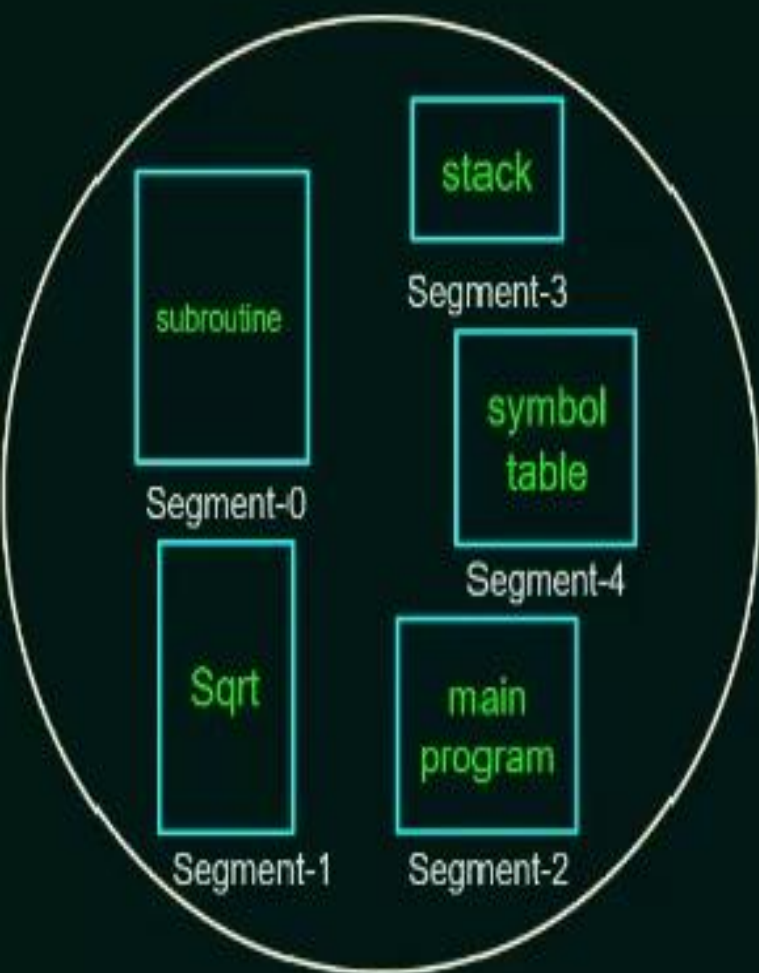
Physical Memory ---->

	Limit	Base
0	1000	1400
1	400	6300
2	400	4300
3	1100	3200
4	1000	4700

Segment table



An example of Segmentation



Logical address space

Physical Memory ---->

	Limit	Base
0	1000	1400
1	400	6300
2	400	4300
3	1100	3200
4	1000	4700

Segment table

1400

segment 0

2400

3200

segment 3

4300

segment 2

4700

segment 4

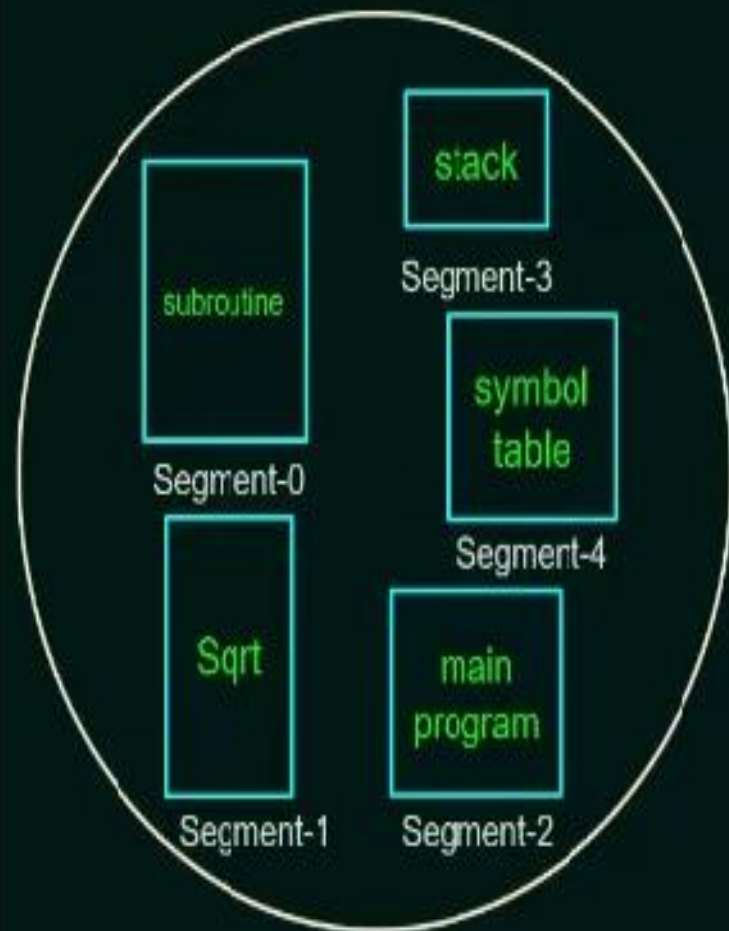
5700

6300

segment 1

6700

Example: Reference to byte 53 of segment 2 - mapped to onto location
 $4300 + 53 = 4353$

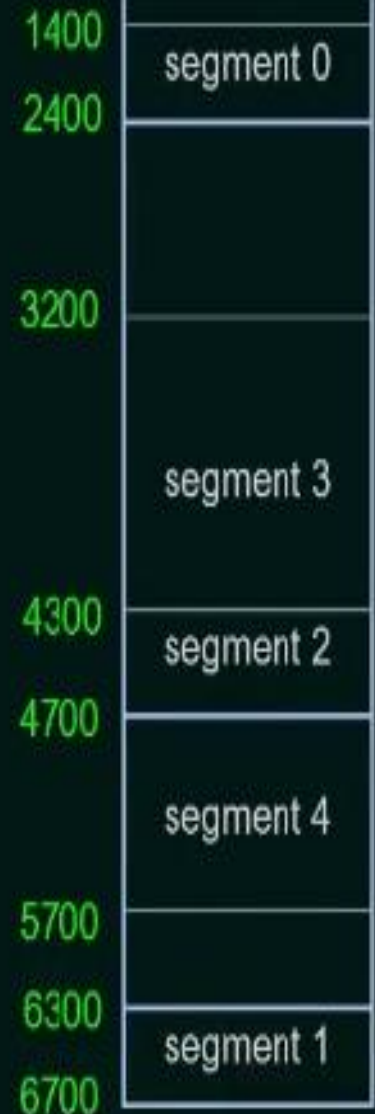


Logical address space

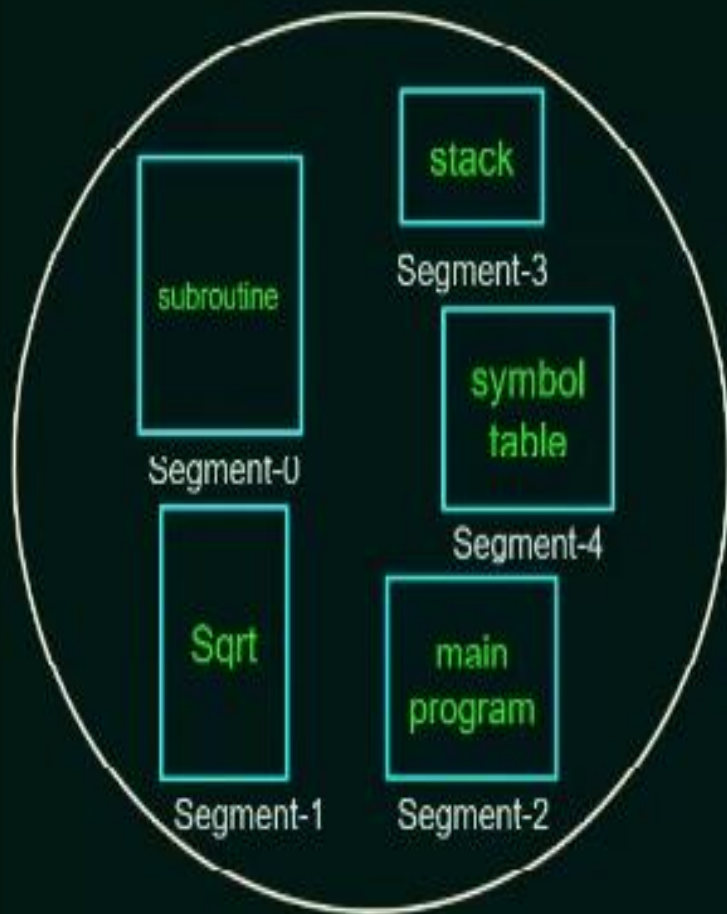
Physical Memory ---->

	Limit	Base
0	1000	1400
1	400	6300
2	400	4300
3	1100	3200
4	1000	4700

Segment table



Example: Reference to byte 852 of segment 3 - mapped to onto location
 $3200 + 852 = 4052$

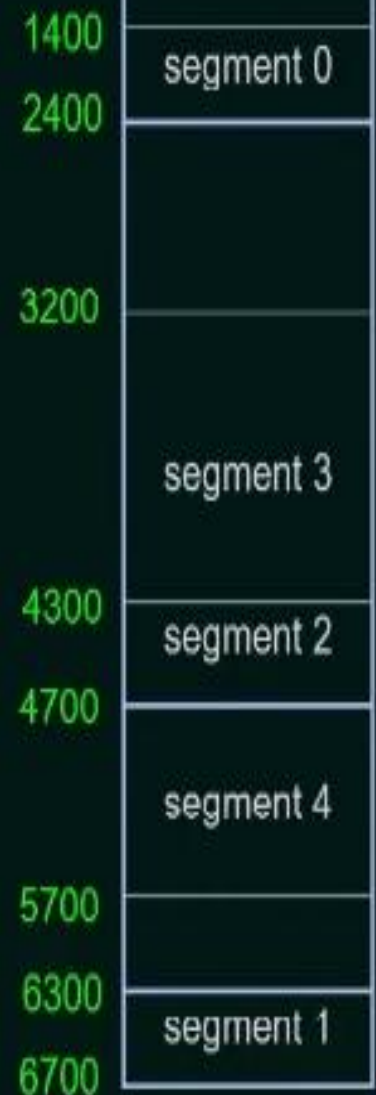


Logical address space

Physical Memory ---->

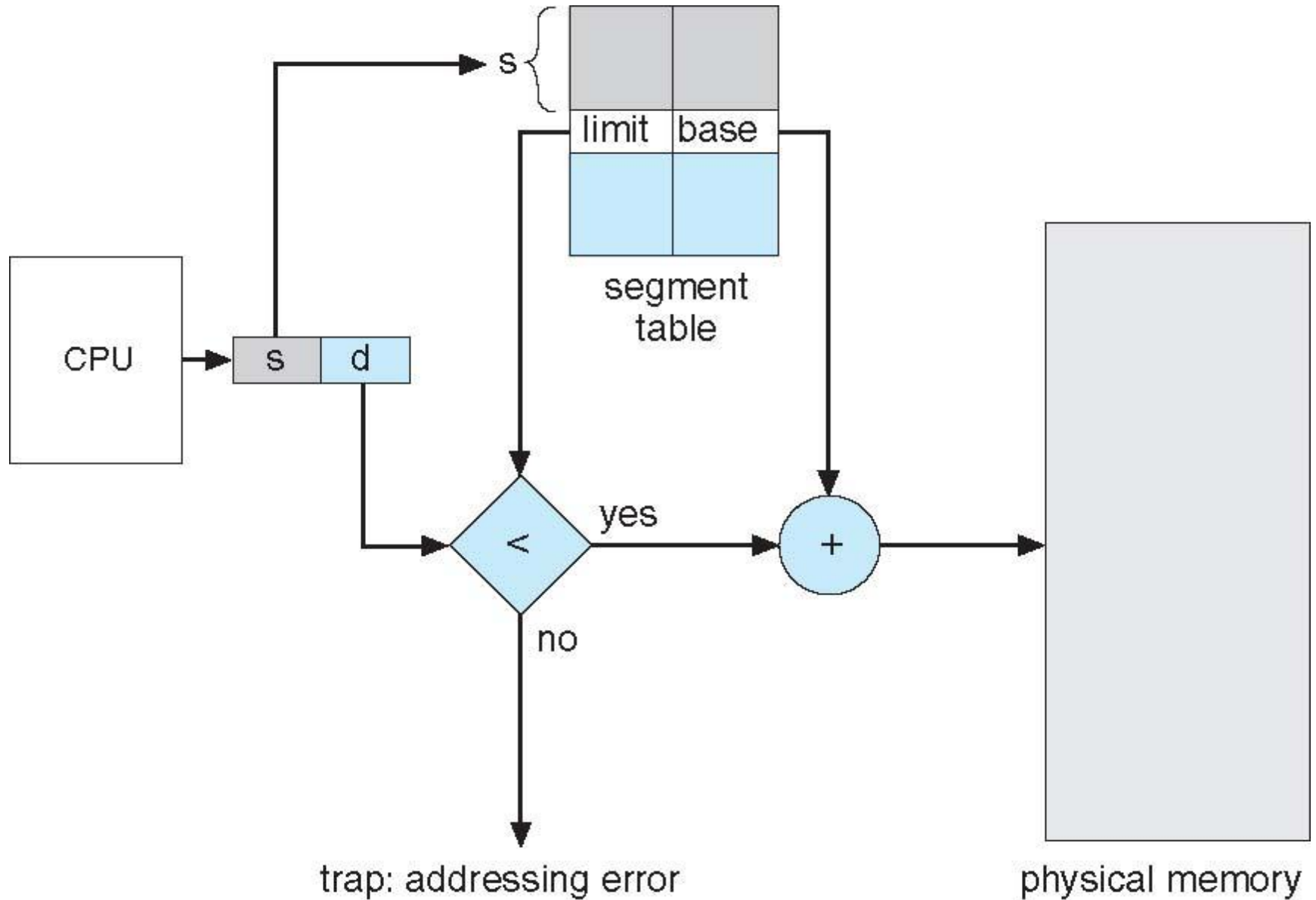
	Limit	Base
0	1000	1400
1	400	6300
2	400	4300
3	1100	3200
4	1000	4700

Segment table



Example: Reference to byte 1222 of segment 0 - would result in a trap to the operating system, as this segment is only 1,000 bytes long.

Segmentation Hardware



Segmentation Architecture (Cont.)

- Protection
 - With each entry in segment table associate:
 - ▶ validation bit = 0 \Rightarrow illegal segment
 - ▶ read/write privileges
- Protection bits associated with segments
- Since segments vary in length, memory allocation is a dynamic storage-allocation problem

Virtual Memory - Background

Code needs to be in memory to execute, but entire program rarely used. [ex]

- Programs often have code to handle unusual error conditions. Since these errors seldom, in practice, this code is almost never executed.
- Arrays, lists, and tables are often allocated more memory than they actually need. An array may be declared 100 by 100 elements, even though it is seldom larger than 10 by 10 elements.
- Certain options and features of a program may be used rarely. For instance, the routines on U.S. Government computers that balance the budget have not been used in many years.
- Symbol table have 3000 symbols , but the average program has less than 200 symbols

Entire program code not needed at same time

The ability to execute a program that is only partially in memory would confer many benefits:

- A program would no longer be constrained by the amount of physical memory that is available. Users would be able to write programs for an extremely large *virtual address space, simplifying the programming task*.
- Because each program could take less physical memory, more programs could be run at the same time, with a corresponding increase in CPU utilization and throughput but with no increase in response time or turnaround time.
- Less I/O would be needed to load or swap portions of programs into memory, so each program would run faster.

So, running a program that is not entirely in memory would benefit both the system and its users.

Virtual memory

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
- Allows for more efficient process creation
- More programs running concurrently
- Less I/O needed to load or swap processes

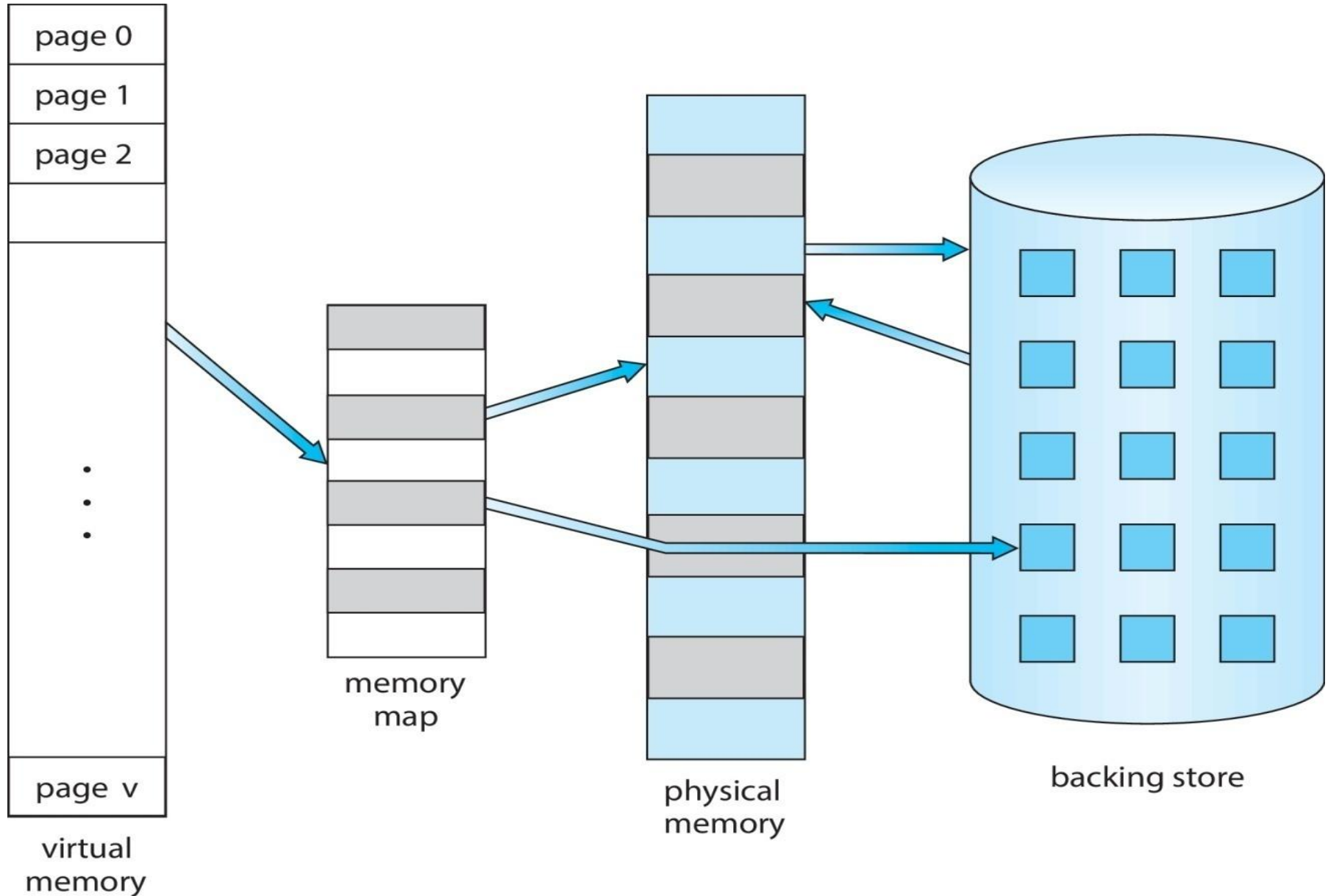
Virtual address space – logical view of how process is stored in memory

- Usually start at address 0, contiguous addresses until end of space
- Meanwhile, physical memory organized in page frames
- MMU must map logical to physical

Virtual memory can be implemented via:

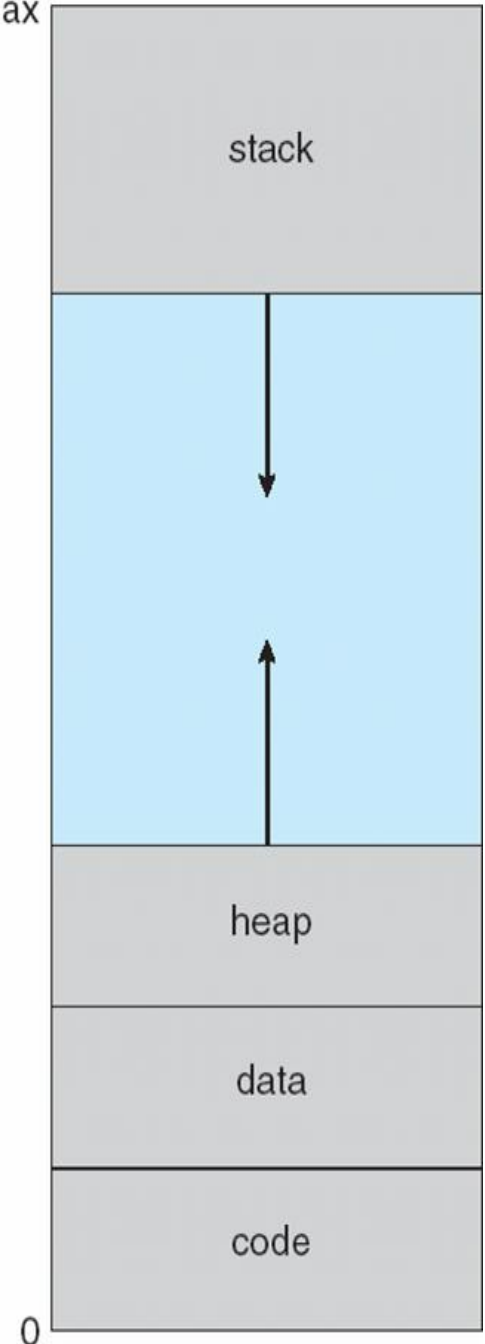
- Demand paging
- Demand segmentation

Virtual Memory That is Larger Than Physical Memory

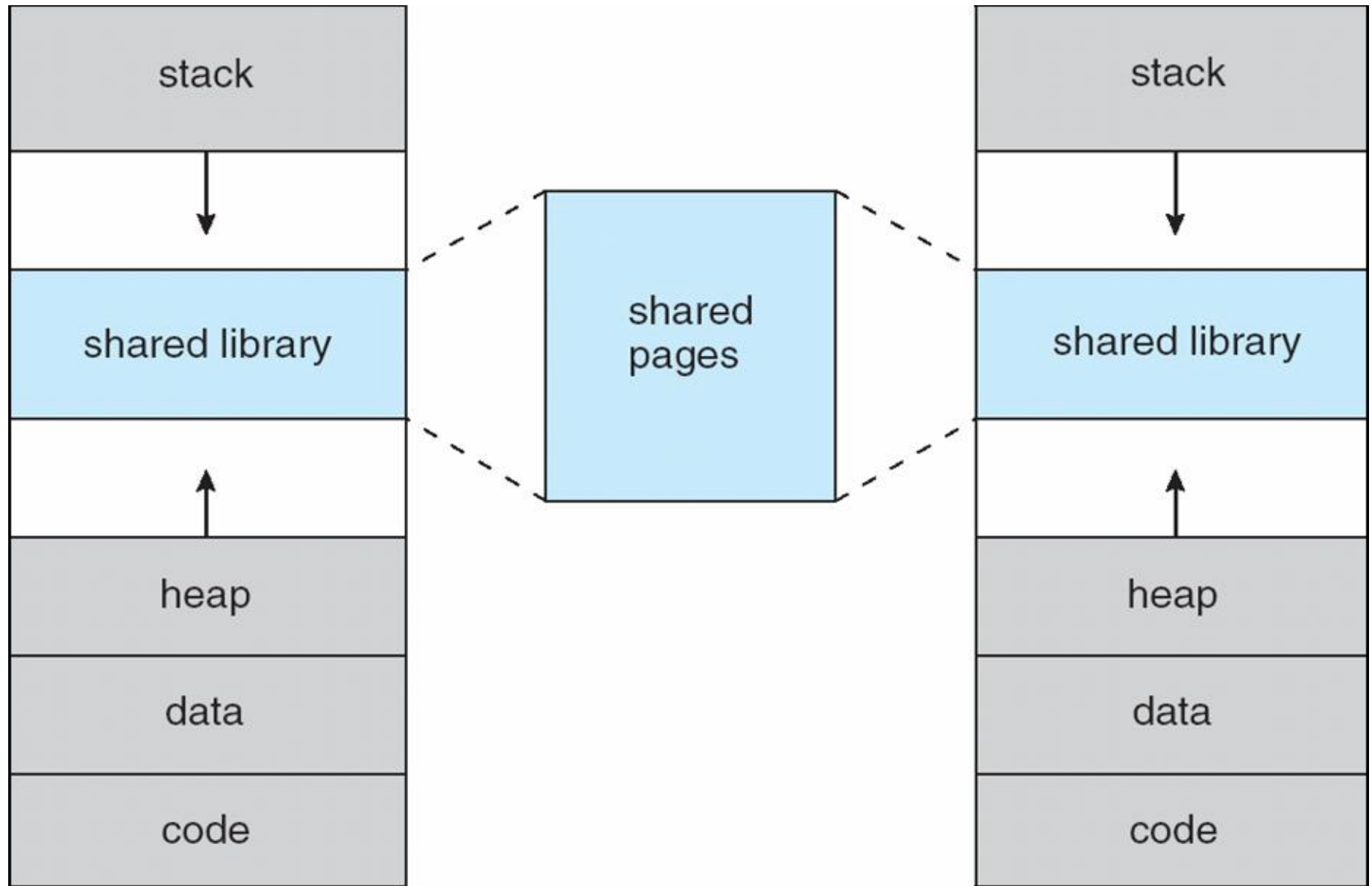


Virtual-address Space^{Max}

- Usually design logical address space for stack to start at Max logical address and grow “down” while heap grows “up”
 - Maximizes address space use
 - The large blank space (or hole) between the heap and the stack is part of the virtual address space.
- Enables **sparse** address spaces with holes left for growth, dynamically linked libraries, etc. System libraries shared via mapping into virtual address space
- Shared memory by mapping pages read-write into virtual address space



Shared Library Using Virtual Memory



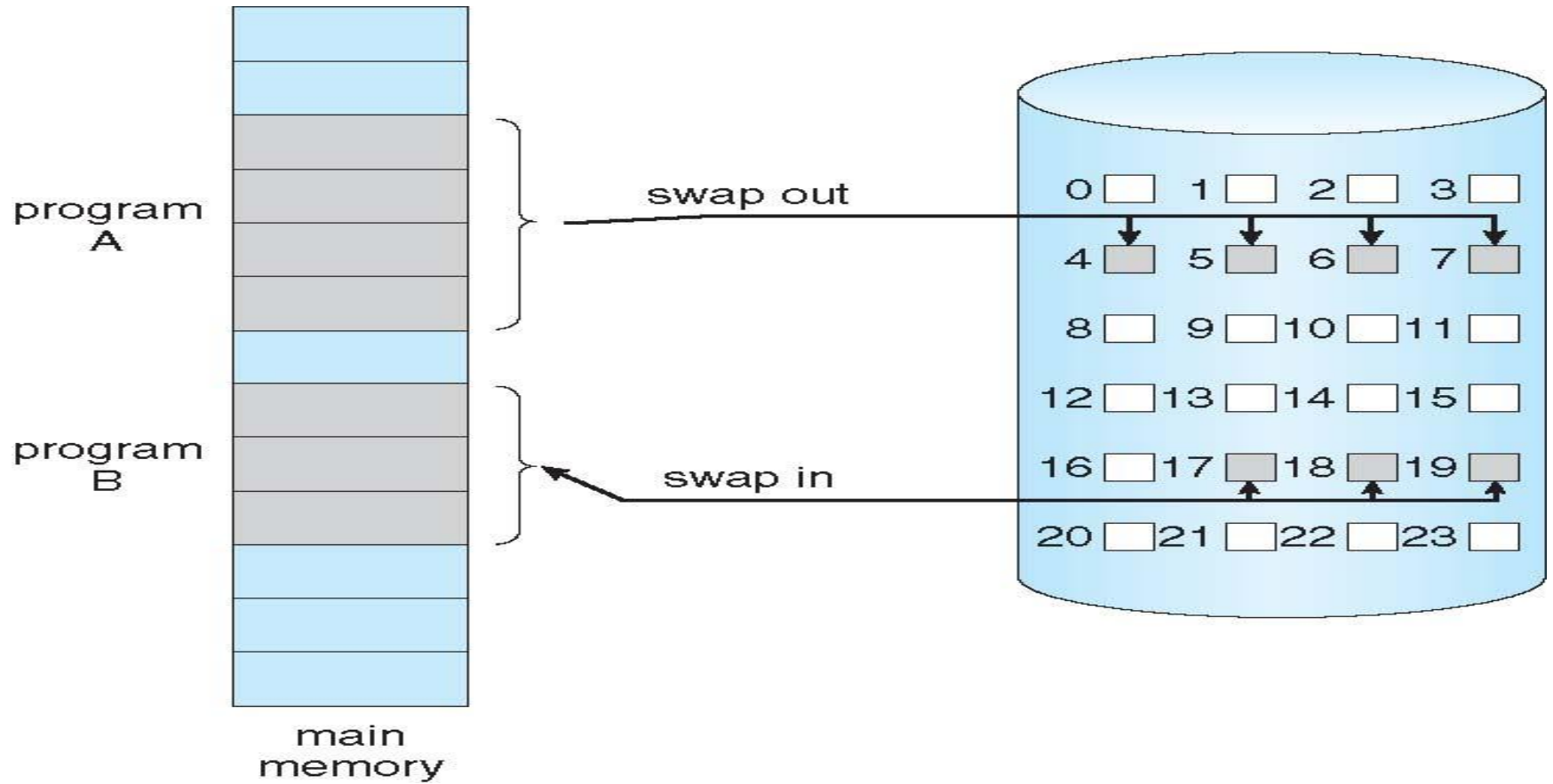
Demand Paging

- Load pages into memory only when it is needed. Pages that are never accessed are never loaded into physical memory. This is called demand paging.

Advantages of Demand paging:

- Less I/O needed, no unnecessary I/O
 - Less memory needed
 - Faster response
 - More users
- Similar to paging system with swapping; Rather than swapping the entire process into memory, use **a lazy swapper**.
 - **Lazy swapper** – never swaps a page into memory unless page will be needed
 - Swapper that deals with pages is a **pager**

Demand Paging



Basic Concepts

- With swapping, pager guesses which pages will be used before swapping out again; Instead, pager brings in only those pages into memory
- How to determine that set of pages?
 - Need new MMU functionality to implement demand paging
- If pages needed are already **memory resident**
 - No difference from non demand-paging
- If page needed and not memory resident
 - Need to detect and load the page into memory from storage
 - ▶ Without changing program behavior
 - ▶ Without programmer needing to change code

Valid-Invalid Bit

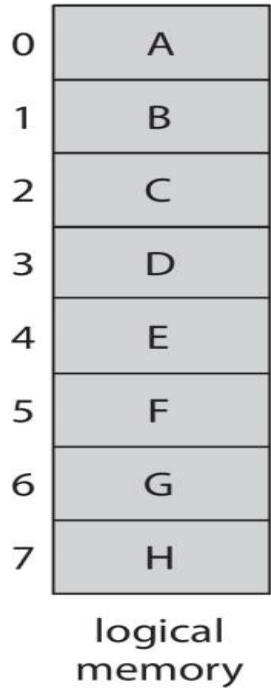
- With each page table entry a valid–invalid bit is associated (**v** \Rightarrow in-memory – **memory resident**, **i** \Rightarrow not-in-memory)
- Initially valid–invalid bit is set to **i** on all entries
- Example of a page table snapshot:

Frame #	valid-invalid bit
	v
	v
	v
	i
...	
	i
	i

page table

- During MMU address translation, if valid–invalid bit in page table entry is **i** \Rightarrow page fault

Page Table When Some Pages Are Not in Main Memory

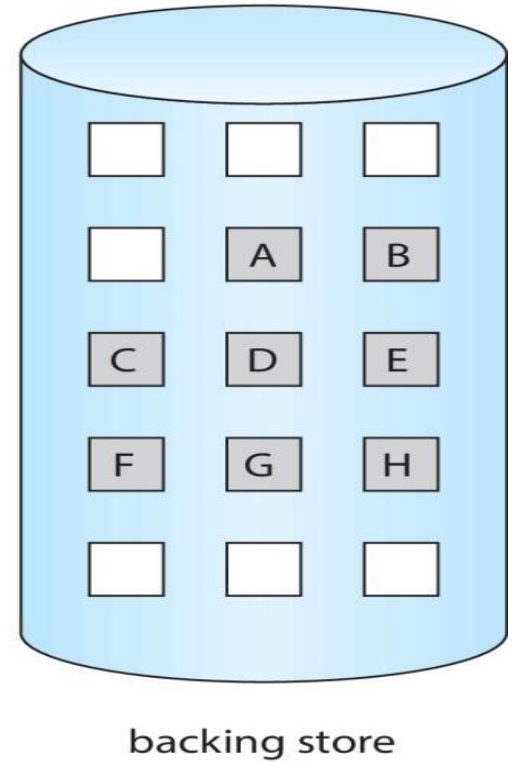
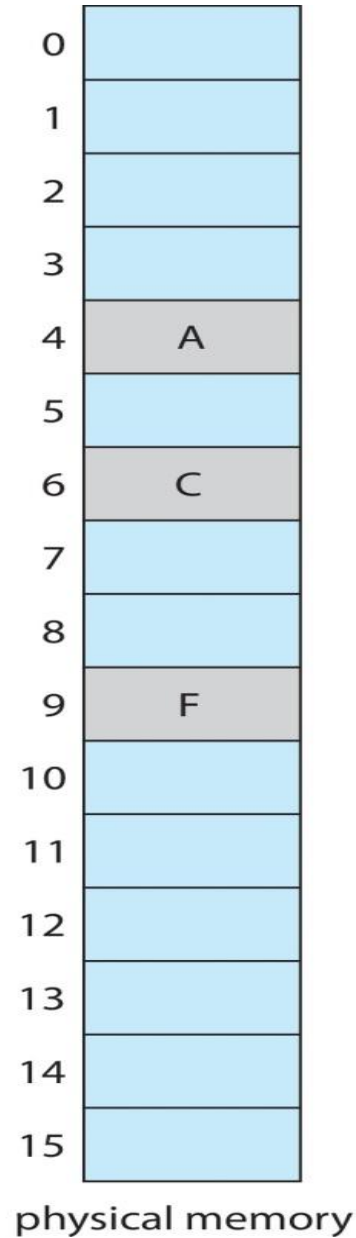


valid-invalid bit

frame

frame	valid-invalid bit
0	4 v
1	i
2	6 v
3	i
4	i
5	9 v
6	i
7	i

page table



Aspects of Demand Paging

- Access a page that is not present in memory is known as **Page fault** and it causes **page fault trap to operating system**

Page fault may occur at any memory reference:

- If the page fault occurs on the instruction fetch, restart by fetching the instruction again
- If the page fault occurs while fetching an operand, fetch and decode the instruction again and then fetch the operand.

Consider a three-address instruction such as ADD the content of A to B, placing the result in C. These are the steps to execute this instruction:

- 1. Fetch and decode the instruction (ADD).**
- 2. Fetch A.**
- 3. Fetch B.**
- 4. Add A and B.**
- 5. Store the sum in C.**

Page fault occurs when try to store in C (because C is in a page not currently in memory). So, to get the desired page, bring it in, correct the page table, and restart the instruction.

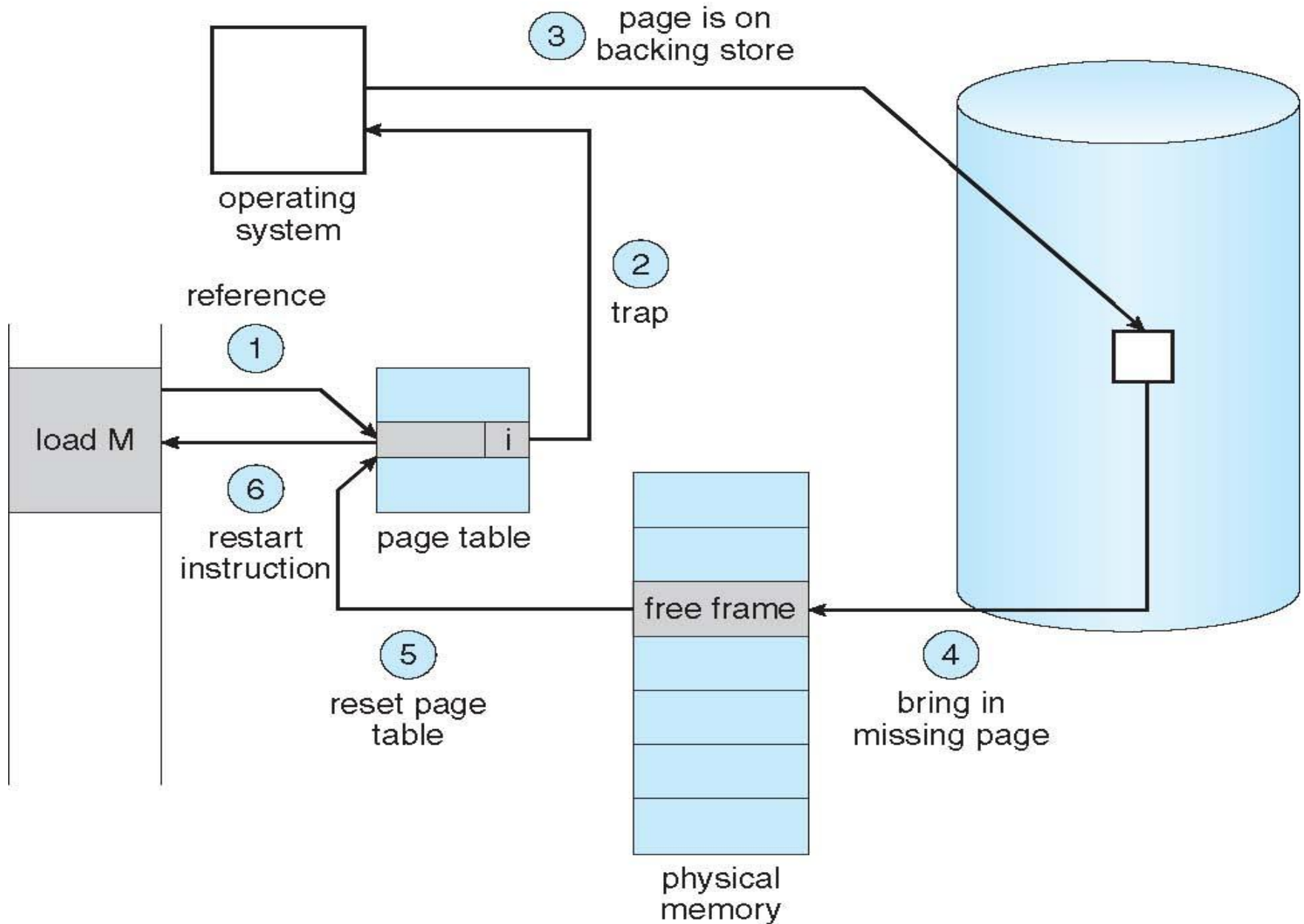
The restart will require

- fetching the instruction again
- decoding it again
- fetching the two operands again, and
- then adding again
- Store the sum in C

Steps in Handling Page Fault

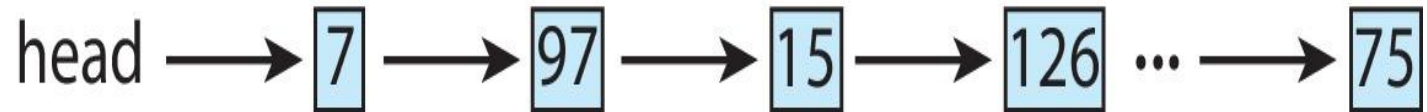
1. If there is a reference to a page, first reference to that page will trap to operating system
 - Page fault
2. Operating system looks at PCB(Process Control Block) table to decide:
 - Invalid reference \Rightarrow abort
 - Just not in memory \Rightarrow bring into memory
3. Find free frame
4. Swap page into frame via scheduled disk operation
5. Reset tables to indicate page now in memory
Set validation bit = **v**
6. Restart the instruction that caused the page fault

Steps in Handling a Page Fault (Cont.)



Free-Frame List

- When a page fault occurs, the operating system must bring the desired page from secondary storage into main memory.
- Most operating systems maintain a **free-frame list** -- a pool of free frames for satisfying such requests.



- Operating system typically allocate free frames using a technique known as **zero-fill-on-demand** -- the content of the frames zeroed-out before being allocated.
- When a system starts up, all available memory is placed on the free-frame list.

Performance of Demand Paging

- A computer system performance is affected by Demand paging
- Memory Access Time (ma) ranges from 10 to 200ns
- If there are no page faults, the Effective Access Time will be equal to the Memory Access Time
- If page fault occurs, Let 'p' be the probability of Page Fault $0 \leq p \leq 1$
 - if $p = 0$ no page faults
 - if $p = 1$, every reference is a fault
- Effective Access Time (EAT)

$$\text{EAT} = \overbrace{(1 - p) \times \text{memory access time}} + \overbrace{p \times \text{page fault time}}$$

Time spent for normal memory access

Time spent for handling page fault

Stages in Demand Paging – Worse Case

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

Not all the steps are necessary in every case. But in all cases the **three major activities for page fault**

1. Service the page fault interrupt - may take 1 to 100 μ s
2. Read the page – lots of time – can take 8ms
3. Restart the process - may take 1 to 100 μ s

So, an average page fault handling time of 8ms and memory access time of 200ns.

$EAT = (1 - p) \times \text{memory access time} + p \times \text{page fault time}$

$$= (1-p) \times 200\text{ns} + p \times 8\text{ms}$$

$$= 200\text{ns} - 200p(\text{ns}) + 8000000p(\text{ns}) = 200\text{ns} + 7999800p(\text{ns})$$

The effective access time is directly proportional to page fault time. If one access out of 1,000 causes a page fault, then

$$\text{Effective access time} = 200\text{ns} + (7999800 \times 1/1000) \text{ ns} = 199.8\text{ns} = 8.2\mu\text{s}$$

This is a slowdown by a factor of 40 percent

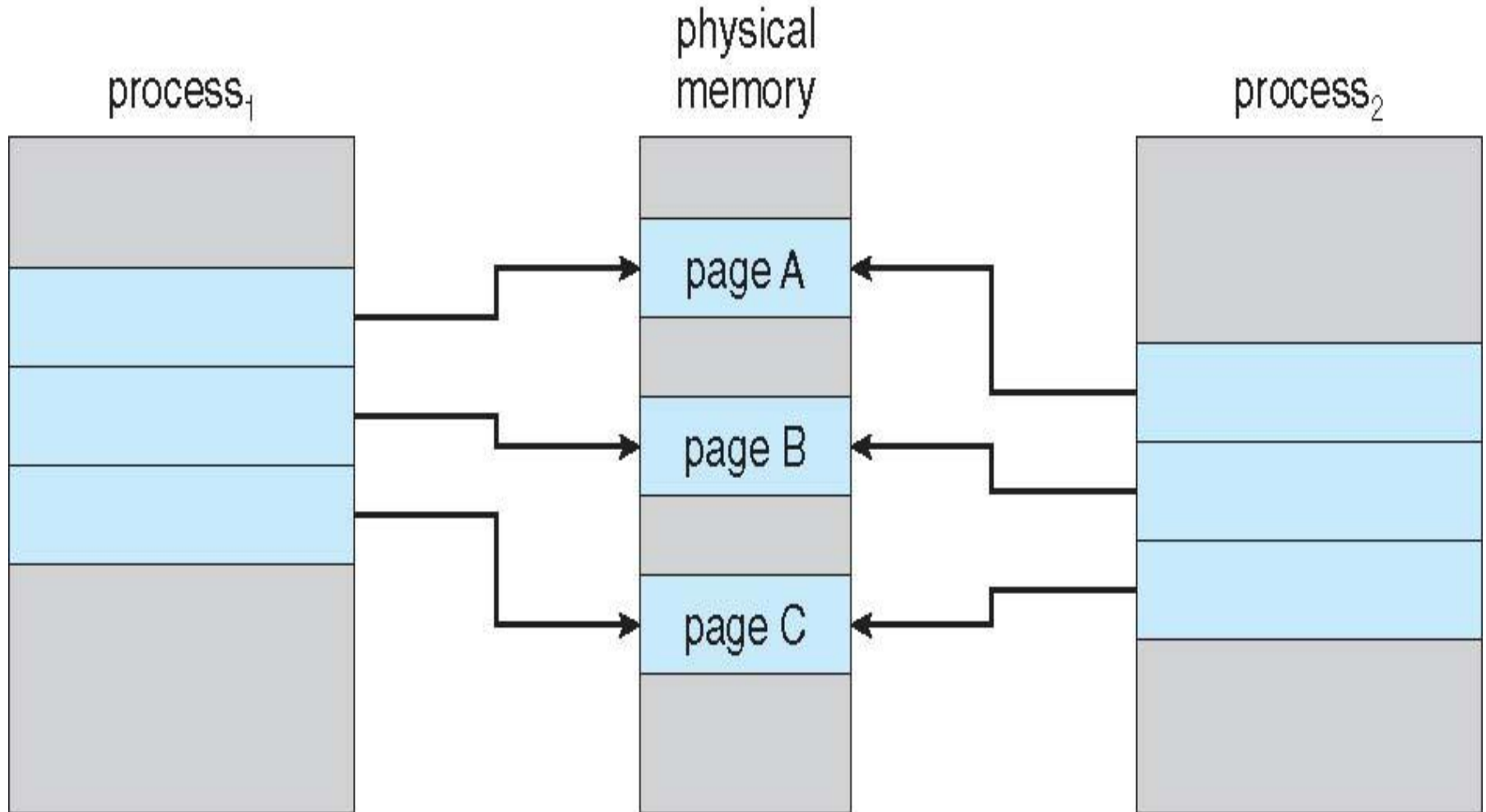
If want performance degradation < 10 percent

One page fault in every 400,000 memory accesses

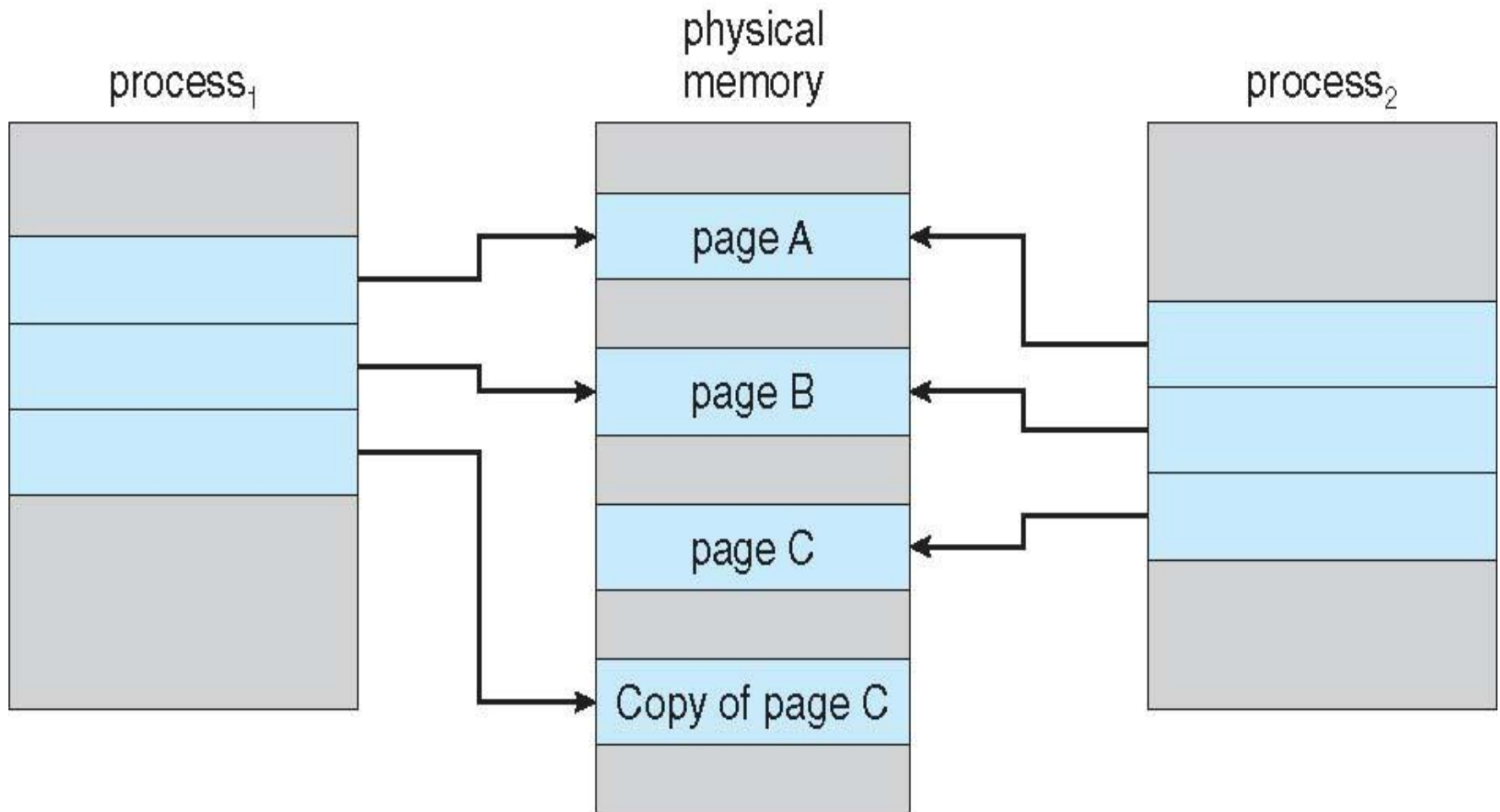
Copy-on-Write

- **Copy-on-Write (COW)** is a technique that allows both parent and child processes sharing virtual memory or pages. Fork() will create a copy of the parent's address space for the child, duplicating the pages belonging to the parent.
- Instead of creating duplicate pages, COW make the parent and child to share the common pages and if either process modifies a shared page, only that page is copied
- In general, free pages are allocated from a **pool of zero-fill-on-demand** pages
 - Pool should always have free frames for fast demand page execution
- vfork() variation on fork() system call has parent suspend and child using copy-on-write address space of parent
 - Designed to have child call exec()
 - Very efficient

Before Process 1 Modifies Page C



After Process 1 Modifies Page C



Copy-on-write is a common technique used by several operating systems, including Windows, Linux, and macOS.

Several versions of UNIX (including Linux, macOS, and BSD UNIX) provide a variation of the `fork()` system **call—`vfork()` (for virtual memory fork)**— that operates differently from `fork()` with copy-on-write. With `vfork()`, the parent process is suspended, and the child process uses the address space of the parent. **Because `vfork()` does not use copy-on-write, if the child process changes any pages of the parent's address space, the altered pages will be visible to the parent once it resumes.**

Therefore, `vfork()` must be used with caution to ensure that the child process does not modify the address space of the parent. `vfork()` is intended to be used when the child process calls `exec()` immediately after creation. Because no copying of pages takes place, `vfork()` is an extremely efficient method of process creation.

Problems of Demand paging

By using demand paging, increasing degree of multiprogramming by loading only the required pages into memory; so, that more processes are loaded into memory. **This lead to over allocation of memory.**

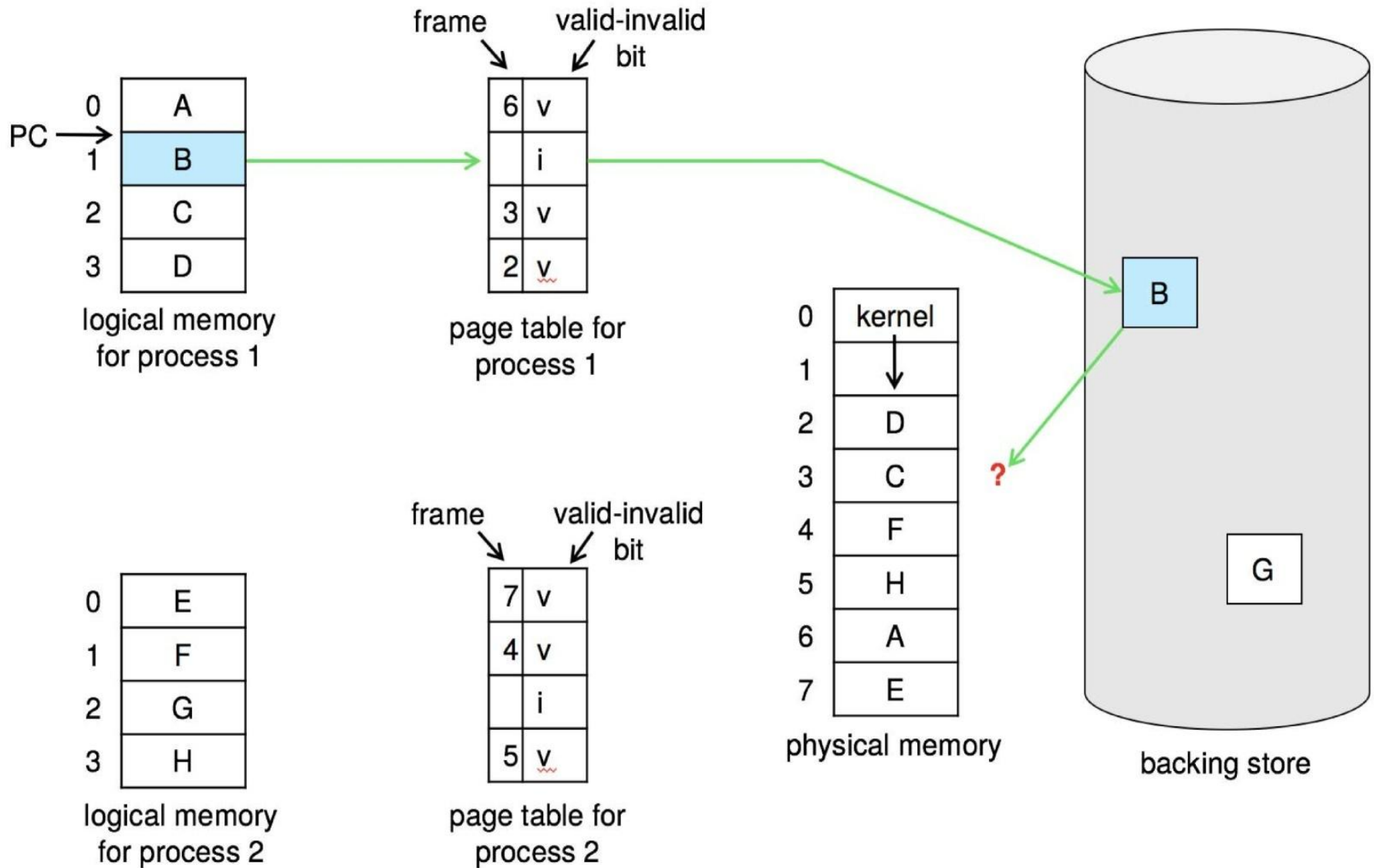
[Eg]: Suppose 40 frames in memory , 6 processes each of which has 10 pages but uses only 5 at the moment. Load these $(6*5) = 30$ pages into the memory. So, all the processes are executing simultaneously and still have 10 frames free.

Now, all the 6 processes wants to use all their 10 pages; So, to load the remaining $(6*5) = 30$ pages into the memory, but only 10 free frames.

What can the OS do at this point?

1. It could terminate the user process- Not the best choice as it destroys the purpose of demand paging
2. The OS could instead swap out a process, freeing all its frames and reducing the level of multiprocessing
3. Use Page replacement technique – most common solution

Need For Page Replacement



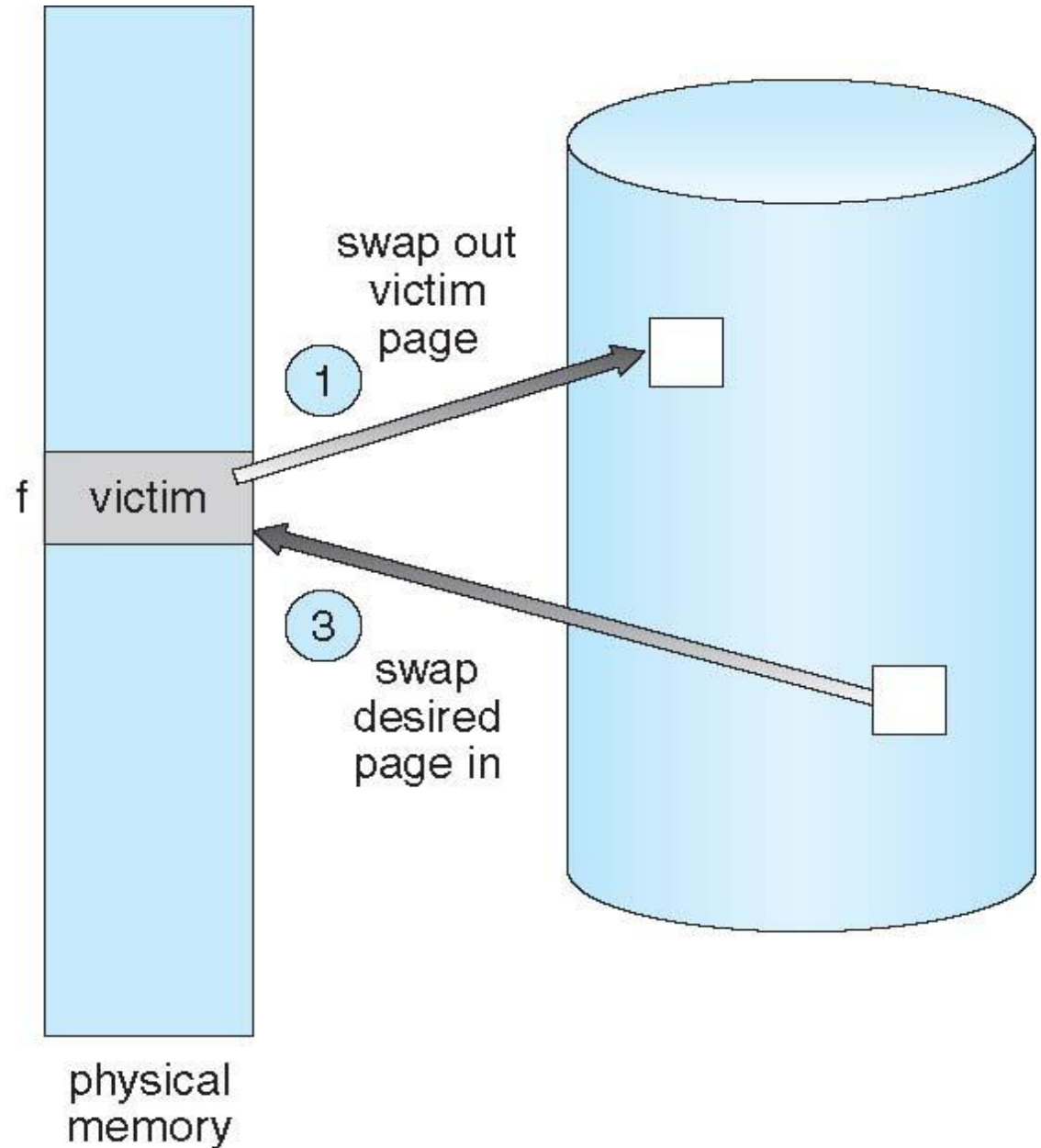
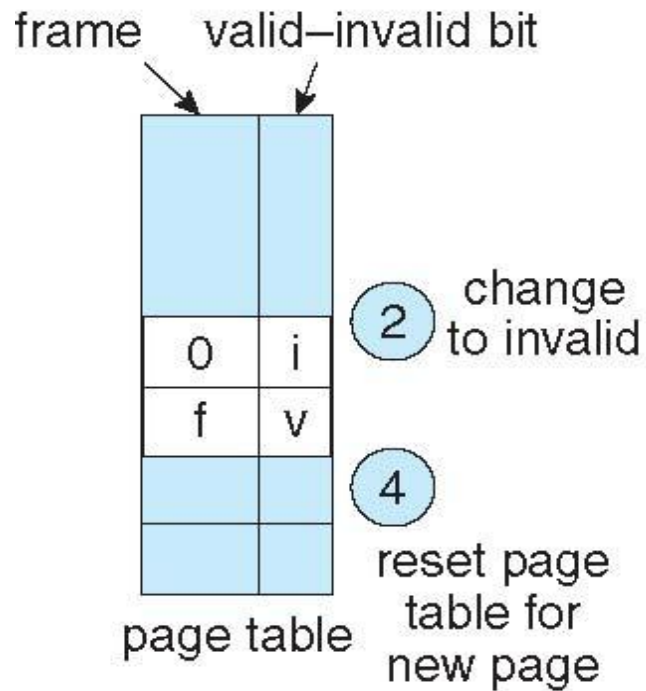
Page Replacement

- Prevent **over-allocation** of memory by modifying page-fault service routine to include page replacement
- It is the technique of swapping out pages from physical memory when there are no free frames available, in order to make room for other pages which has to be loaded to the physical memory
- If a process wants to access a page which is not in memory, it will cause a page fault. Now, page has to be loaded into memory
- Page replacement completes separation between logical memory and physical memory – large virtual memory can be provided on a smaller physical memory

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**
 - Write victim frame to disk if dirty
 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.
- If no frames are free, two page transfers(one out and one in) are required. It increase the Effective Access Time.** To reduce this time, make use of **modify (dirty) bit**. The modify bit is set when page has been modified.
- If modify bit is set** – It means that page has been modified and need to write in the disk
- If modify bit is not set** – It means that page has not been modified and no need to write in the disk. So it reduce Effective access Time.

Page Replacement

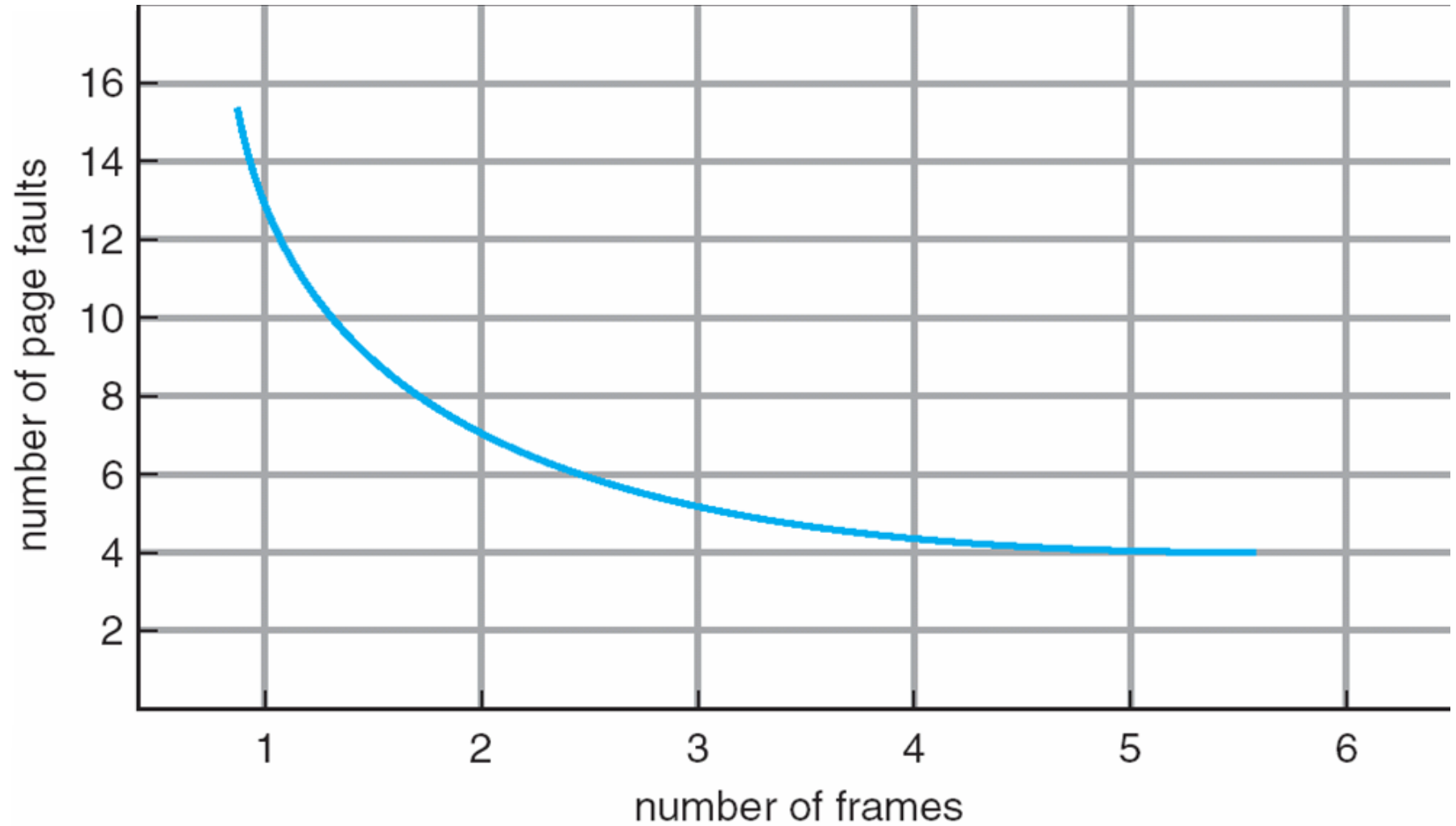


Page and Frame Replacement Algorithms

- **Frame-allocation algorithm** determines
 - How many frames to give each process
 - Which frames to replace
- **Page-replacement algorithm**
 - Want lowest page-fault rate on both first access and re-access
- Evaluate algorithm by running it on a particular string of memory references (reference string) and computing the number of page faults on that string
 - String is just page numbers, not full addresses
 - Repeated access to the same page does not cause a page fault
 - Results depend on number of frames available
- In all our examples, the **reference string** of referenced page numbers is

7,0,1,2,0,3,0,4,2,3,0,3,0,3,2,1,2,0,1,7,0,1

Graph of Page Faults Versus the Number of Frames



First-In-First-Out (FIFO) Algorithm

- Reference string: **7,0,1,2,0,3,0,4,2,3,0,3,0,3,2,1,2,0,1,7,0,1**
- 3 frames (3 pages can be in memory at a time per process)

reference string

7 0 1 2 0 3 0 4 2 3 0 3 0 3 2 1 2 0 1 7 0 1

7	7	7	2																		
	0	0	0		2	2	4	4	4	0				0	0				7	7	7
					3	3	3	2	2	2				1	1				1	0	0
		1	1		1	0	0	0	3	3				3	2				2	2	1

page frames

15 page faults

- Can vary by reference string: consider 1,2,3,4,1,2,5,1,2,3,4,5
 - Adding more frames can cause more page faults!
- **Belady's Anomaly**

Advantage : Simple

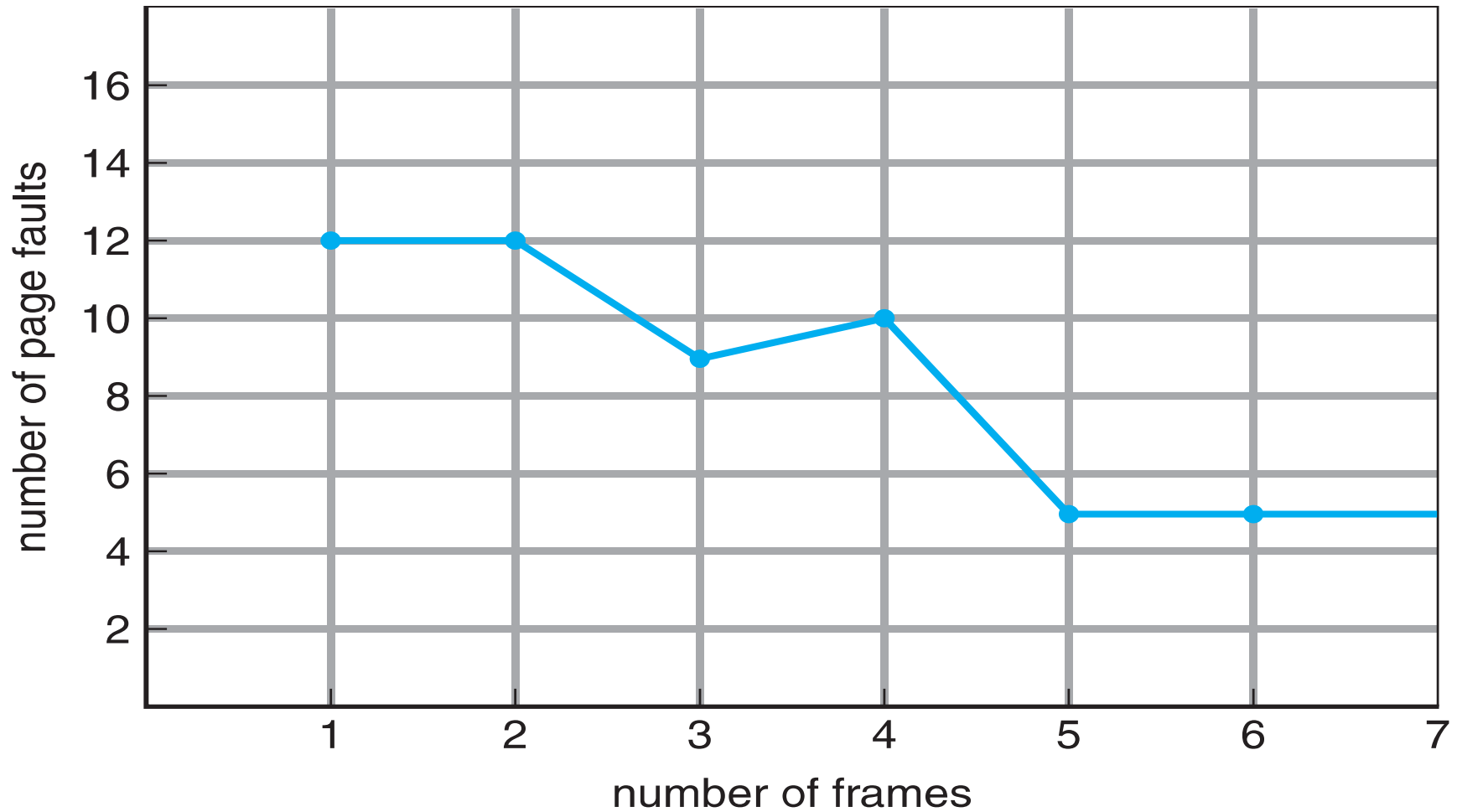
Consider 1,2,3,4,1,2,5,1,2,3,4,5 with 3 pages : 9 page faults

1	2	3	4	1	2	5	1	2	3	4	5
1	1	1	4	4	4	5	5	5	5	5	5
	2	2	2	1	1	1	1	1	3	3	3
		3	3	3	2	2	2	2	2	4	4
F	F	F	F	F	F	F			F	F	

Consider 1,2,3,4,1,2,5,1,2,3,4,5 with 4 pages : 10 page faults

1	2	3	4	1	2	5	1	2	3	4	5
1	1	1	1	1	1	5	5	5	5	4	4
	2	2	2	2	2	2	1	1	1	1	5
		3	3	3	3	3	3	2	2	2	2
			4	4	4	4	4	4	3	3	3
F	F	F	F			F	F	F	F	F	F

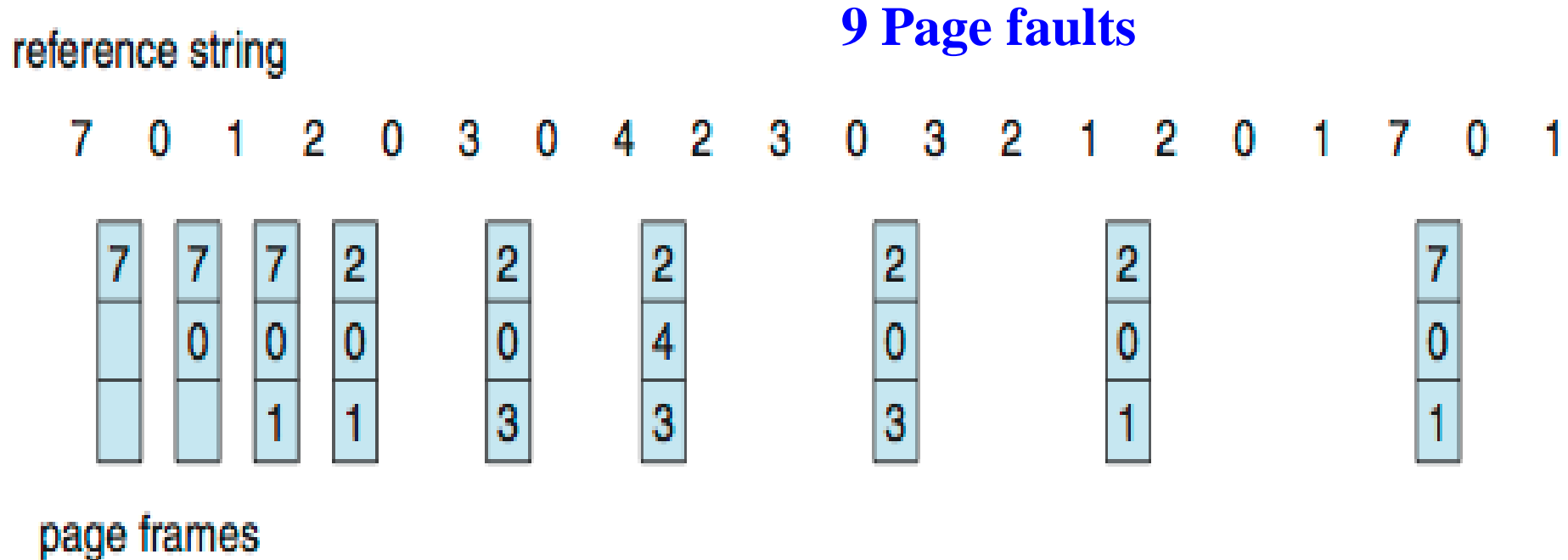
FIFO Illustrating Belady's Anomaly



Consider 1,2,3,4,1,2,5,1,2,3,4,5 with more than 4 pages: 5 page faults
Because the distinct values are 1,2,3,4,5

Optimal Algorithm

- Replace page that will not be used for longest period of time
- Never suffer from Belady's anomaly
- Guarantees the lowest page fault rate



Advantage : Page fault is low compared to all algorithms

Dis Advantages : No hardware support for implementation, Can't read the future

Least Recently Used (LRU) Algorithm

- Use past knowledge rather than future
- Replace page that has not been used in the most amount of time
- Associate time of last use with each page
- Better than FIFO but worse than OPT
- Generally good algorithm and frequently used

12 Page faults

reference string

7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 7 0 1

7	7	7	2		2		4	4	4	0			1		1		1		
	0	0	0		0		0	0	3	3			3		0		0		
		1	1		3		3	2	2	2			2		2		7		

page frames

EX1 : Consider the following page reference string:

1, 2, 3, 4, 2, 1, 5, 6, 2, 1, 2, 3, 7, 6, 3, 2, 1, 2, 3, 6.

How many page faults would occur for the following replacement algorithms, **assuming one, two, three, four, five, six, and seven frames?** Remember that all frames are initially empty, so your first unique pages will cost one fault each.

- LRU replacement
- FIFO replacement
- Optimal replacement

<u>Number of frames</u>	<u>LRU</u>	<u>FIFO</u>	<u>Optimal</u>
1	20	20	20
2	18	18	15
3	15	16	11
4	10	14	8
5	8	10	7
6	7	10	7
7	7	7	7

Ex2 : Consider the following page reference string:

7, 2, 3, 1, 2, 5, 3, 4, 6, 7, 7, 1, 0, 5, 4, 6, 2, 3, 0 , 1.

Assuming demand paging with **three frames**, how many page faults would occur for the following replacement algorithms?

- LRU replacement - 18
- FIFO replacement - 17
- Optimal replacement - 13

G1. Consider a computer system with ten physical page frames. The system is provided with an access sequence $(a_1, a_2, \dots, a_{20}, a_1, a_2, \dots, a_{20})$, where each a_i is a distinct virtual page number. The difference in the number of page faults between the last-in-first-out page replacement policy and the optimal page replacement policy is _____. (GATE 2016)

- A. 1 B.2 C.3 D.4**

Calculate the difference between the last-in-first-out page replacement policy and the optimal page replacement policy.

First, Consider LIFO (Last In First Out) $\rightarrow a_1$ to a_{10} will result in page faults = 10 page faults. Then a_{11} will replace a_{10} (last in is a_{10}), a_{12} replace a_{11} and ...till a_{20} = 10 page faults and a_{20} will be top of stack and $a_9 \dots a_1$ are remained as such. Then a_1 to a_9 are already there. So, 0 page faults from a_1 to a_9 . a_{10} will replace a_{20} , a_{11} will replace a_{10} and so on = So 11 page faults.

So total faults will be $10+10+11 = 31$.

Second Optimal Page Replacement Policy \rightarrow

a_1 to $a_{10} = 10$ page faults, a_{11} will replace a_{10} because among a_1 to a_{10} , a_{10} will be used later, a_{12} will replace a_{11} and so on = 10 page faults.

a_{20} will be top of stack and $a_9 \dots a_1$ are remained as such.

a_1 to $a_9 = 0$ page fault. a_{10} will replace a_1 because it will not be used afterwards and so on, a_{10} to a_{19} will have 10 page faults.

So no page fault for a_{20} .

Total = $10 + 10 + 10 = 30$.

So the difference between LIFO - Optimal = 1

G2. Consider a main memory with five page frames and the following sequence of page references: 3, 8, 2, 3, 9, 1, 6, 3, 8, 9, 3, 6, 2, 1, 3. Which one of the following is true with respect to page replacement policies First-In-First Out (FIFO) and Least Recently Used (LRU)? (GATE 2015)

A. Both incur the same number of page faults B. FIFO incurs 2 more page faults than LRU
C. LRU incurs 2 more page faults than FIFO
D. FIFO incurs 1 more page faults than LRU

LRU:

3	8	2	3	9	1	6	3	8	9	3	6	2	1	3
						1	1	1	1	1	1	2	2	2
				9	9	9	9	9	9	9	9	9	9	9
		2	2	2	2	2	2	8	8	8	8	8	1	1
	8	8	8	8	8	6	6	6	6	6	6	6	6	6
3	3	3	3	3	3	3	3	3	3	3	3	3	3	3
F	F	F	H	F	F	F	H	F	H	H	H	F	F	H

Both LRU and FIFO have 9 page faults

FIFO:

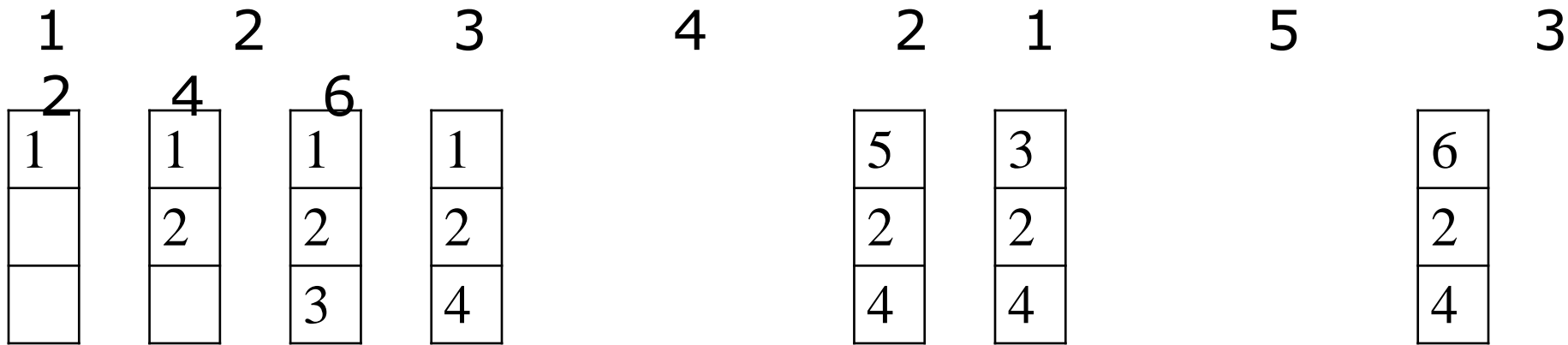
3	8	2	3	9	1	6	3	8	9	3	6	2	1	3	1
					1	1	1	1	1	1	1	1	1	1	1
				9	9	9	9	9	9	9	9	2	2	2	2
		2	2	2	2	2	2	8	8	8	8	8	8	8	8
	8	8	8	8	8	6	6	3	3	3	3	3	3	3	3
3	3	3	3	3	3	6	6	6	6	6	6	6	6	6	6
F	F	F	H	F	F	F	H	F	H	H	H	F	F	H	

A. 7

B.8

C.9

D.10



Total 7 faults

G4. Consider the virtual page reference string 1, 2, 3, 2, 4, 1, 3, 2, 4, 1 on a demand paged virtual memory system running on a computer system that main memory size of 3 pages frames which are initially empty. Let LRU, FIFO and OPTIMAL denote the number of page faults under the corresponding page replacements policy. Then (GATE 2012)

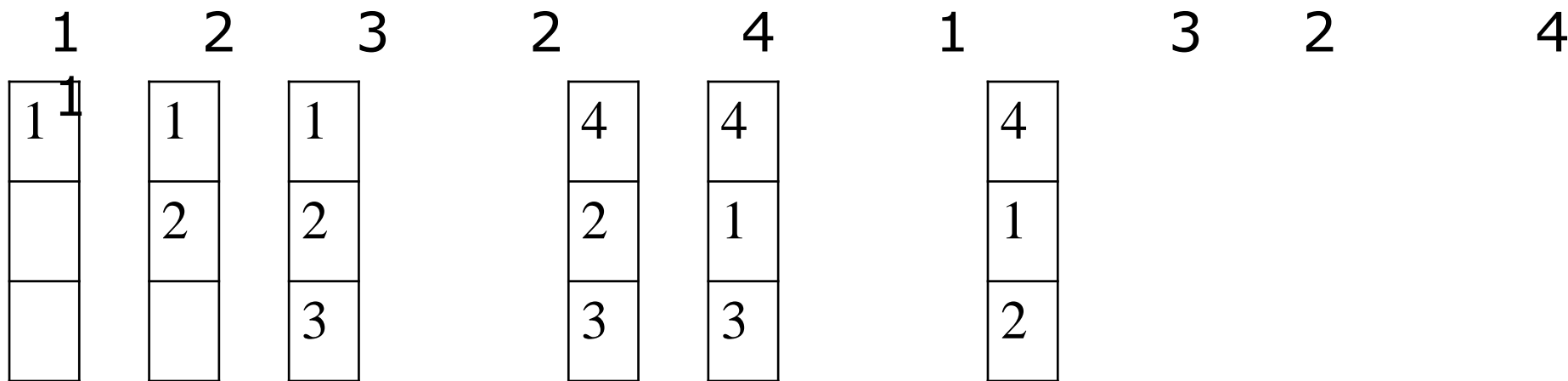
A. $\text{OPTIMAL} < \text{LRU} < \text{FIFO}$

B. $\text{OPTIMAL} < \text{FIFO} < \text{LRU}$

C. $\text{OPTIMAL} = \text{LRU}$

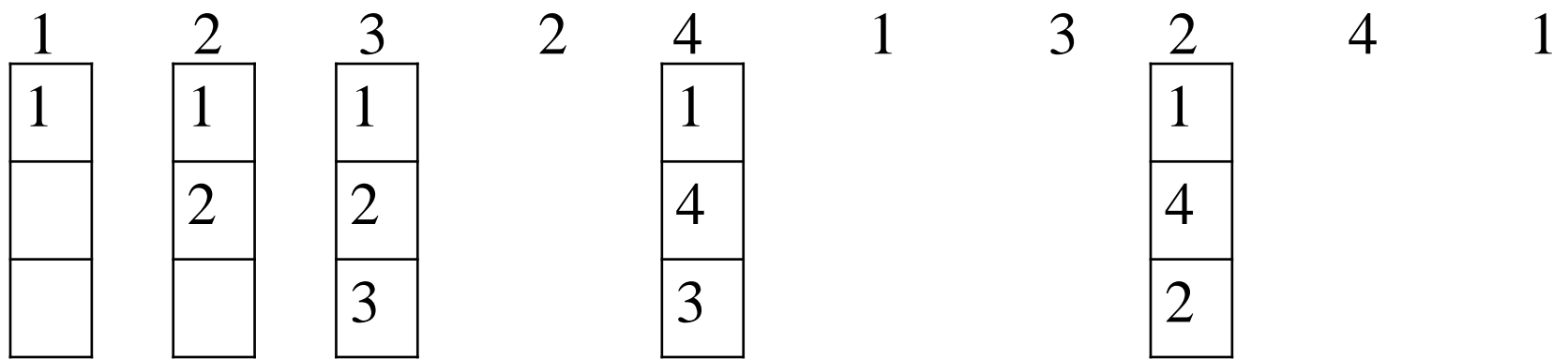
D. $\text{OPTIMAL} = \text{FIFO}$

FIFO



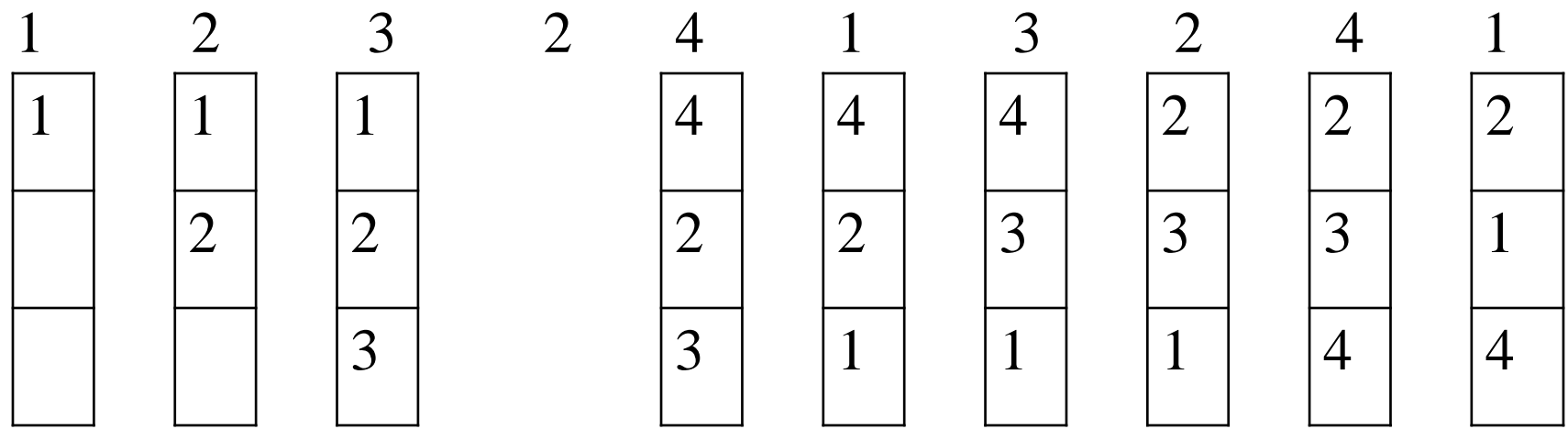
No. of page faults with FIFO = 6

OPTIMAL:



No. of page faults with OPTIMAL =5

LRU



No. of page faults with LRU = 9

∴ Optimal < FIFO < LRU

G5. A system uses FIFO policy for page replacement. It has 4 page frames with no pages loaded to begin with. The system first accesses 100 distinct pages in some order and then accesses the same 100 pages but now in the reverse order. How many page faults will occur? (GATE 2010)

- A. 196** B. 192 C. 197 D. 195

The first 100 accesses causes 100 page faults for Page1 to Page100. Now, when they are accesses in reverse order page 100, 99, 98, 97 are already present. So 96 page faults for Page0 to Page 96. So, total of 196 page faults.

G6. A process has been allocated 3 page frames. Assume that none of the pages of the process are available in the memory initially. The process makes the following sequence of page references (reference string): 1, 2, 1, 3, 7, 4, 5, 6, 3, 1 If optimal page replacement policy is used, how many page faults occur for the above reference string? (GATE 2007)

- A. 7** B. 8 C. 9 D. 10

Using Optimal page replacement,

1	2	1	3	7	4	5	6	3	1
1	1		1	1	1	1	1		
	2		2	7	4	5	6		
			3	3	3	3	3		

Total 7 page faults

G7. The address sequence generated by tracing a particular program executing in a pure demand paging system with 100 bytes per page is 0100, 0200, 0430, 0499, 0510, 0530, 0560, 0120, 0220, 0240, 0260, 0320, 0410. Suppose that the memory can store only one page and if x is the address which causes a page fault then the bytes from addresses x to x + 99 are loaded on to the memory. How many page faults will occur ? (GATE 1995, 2007)

(A) 0 (B) 4 (C) 7 (D) 8

Page are fitted in frames, so first to determine the pages but given request is the record request in decimal. Assume that page1 to contain records from 0000 to 0099 and page 2 contains records from 0100 to 0199 and so on (it is given in question that each page contains 100 records) and so on. So page request string is 01, 02, 04, 04, 05, 05, 05, 01, 02, 02, 02, 03, 03. Clearly 7 page faults.

G8. Consider a demand paging system with four-page frames (initially empty) and an LRU page replacement policy. For the following page reference string 7, 2,7,3, 2,5,3, 4,6,7,7,1,5,6,1 the page fault rate, defined as the ratio of number of page faults to the number of memory accesses (rounded off to one decimal place) is_____. (GATE 2022)

A 1.5 B 0.5 C **0.6** D 0.8

String : 7,2,7,3,2,5,3,4,6,7,7,1,5,6,1

Page fault : 9

Number of Memory accesses : 15 [String contain 15 references]

thus page fault rate = $9/15=0.6$

G9. Which one of the following statements is FALSE? (GATE 2022)

- A. The TLB performs an associative search in parallel on all its valid entries using page number of incoming virtual address.
- B. If the virtual address of a word given by CPU has a TLB hit, but the subsequent search for the word results in a cache miss, then the word will always be present in the main memory.
- C. The memory access time using a given inverted page table is always same for all incoming virtual addresses.**
- D. In a system that uses hashed page tables, if two distinct virtual addresses V1 and V2 map to the same value while hashing, then the memory access time of these addresses will not be the same.

G10. Consider allocation of memory to a new process. Assume that none of the existing holes in the memory will exactly fit the process's memory requirement. Hence, a new hole of smaller size will be created if allocation is made in any of the existing holes. Which one of the following statement is TRUE ?(GATE 2020)

- (A) The hole created by first fit is always larger than the hole created by next fit.
- (B) The hole created by worst fit is always larger than the hole created by first fit.
- (C) The hole created by best fit is never larger than the hole created by first fit.**
- (D) The hole created by next fit is never larger than the hole created by best fit.

G11. In which one of the following page replacement algorithms it is possible for the page fault rate to increase even when the number of allocated frames increases?

- (A) LRU (Least Recently Used) (B) OPT (Optimal Page Replacement)
- (C) MRU (Most Recently Used) **(D) FIFO (First In First Out)**

In some situations FIFO page replacement gives more page faults when increasing the number of page frames. This situation is Belady's anomaly.

LRU Algorithm Implementation

There are two ways to implement LRU ;

1. Using Counters 2. Using Stack

■ Counter implementation

Page Table Entry

Frame No	Present /Absent	Protection	Reference	Caching	Dirty	Time of Use
----------	-----------------	------------	-----------	---------	-------	-------------

Clock/
Counter



With each page table entry, associate a time of use field and add logical clock or counter to the CPU

The clock/counter is incremented for every memory reference

Whenever a reference to a page is made, the contents of the clock/counter are copied to the time of use field in the page table entry for that page

So, always have the ‘time/count’ value of the last reference to each page

- **When a page needs to be changed, look at the counters to find smallest value**

▶ Search through table needed

LRU Algorithm Implementation

Stack implementation

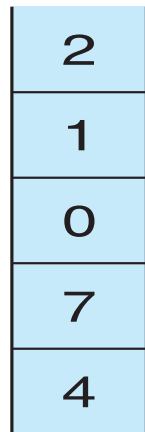
- Maintain a stack of Page Numbers
- Whenever a page is referenced, it is removed from the stack and put on the top
- So, the most recently used page is always at the top of the stack and the least recently used page is always at the bottom
- Since, entries must be removed from the middle of the stack, it is best to implement by using a doubly linked list with head and tail pointer
- But each update more expensive
- No search for replacement

LRU Algorithm (Cont.)

- LRU and OPT are cases of **stack algorithms** that don't have Belady's Anomaly

reference string

4 7 0 7 1 0 1 2 1 2 7 1 2



stack
before
a



stack
after
b



LRU Approximation Page Replacement Algorithms

- LRU needs special hardware and still slow
- Many systems provide support for LRU in the form of **Reference bit**

Steps:

- Initially, the OS sets all the reference bits for all the pages to '0'
- Whenever a page is referenced, the reference bit set as '1'
- By checking these reference bits, know about which pages have been used and which haven't been used
- But, do not determine the order of use - For this use **Additional-Reference-Bits Algorithm**

LRU approximation algorithm –Additional Reference Bits Algorithm

- Additional ordering information by recording the reference bits at regular intervals.
- Keep an 8-bit byte for each page in a table in memory.
- At regular intervals (say, every 100 milliseconds), a timer interrupt transfers control to the operating system.
- The operating system shifts the reference bit for each page into the high-order bit of its 8-bit byte, shifting the other bits right by 1 bit and discarding the low-order bit.
- These 8-bit shift registers contain the history of page use for the last eight time periods

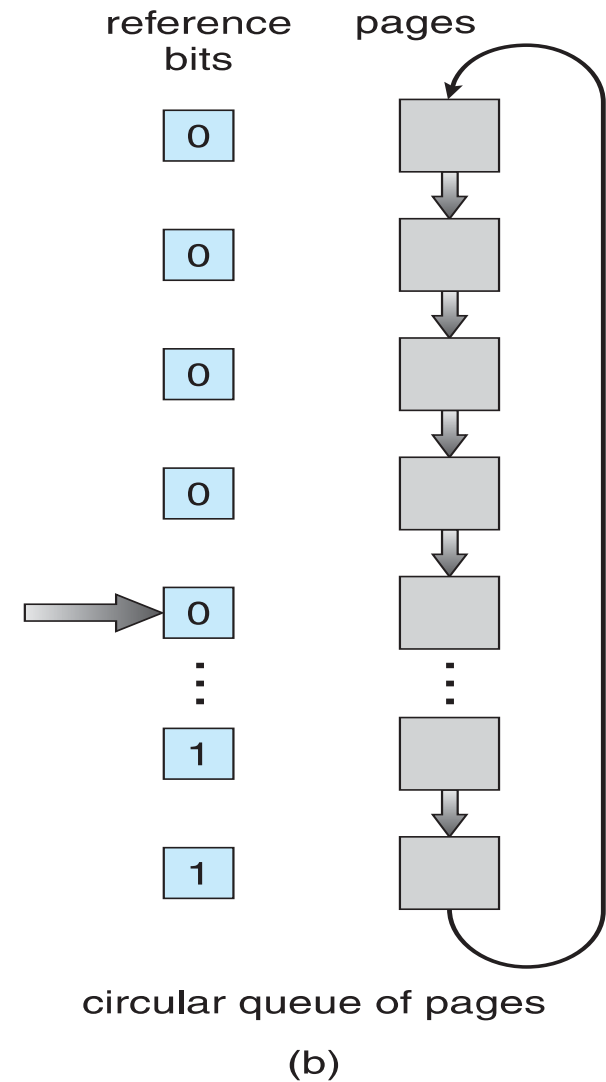
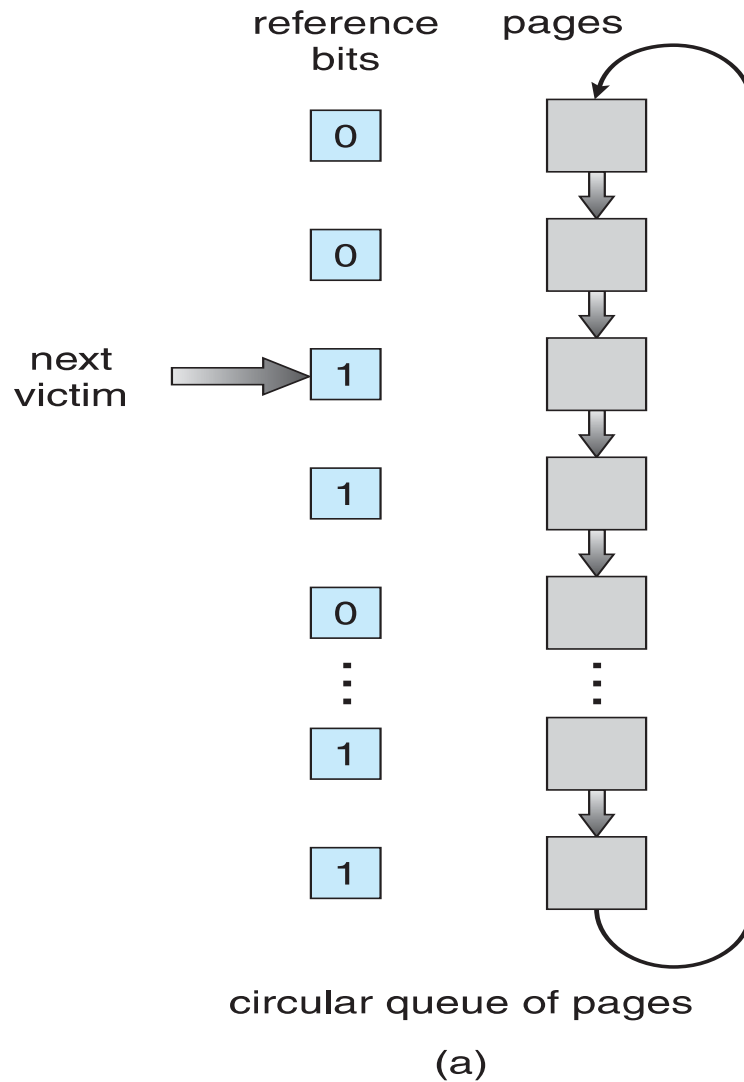
Page No	Shift Register Content	What it means
P1	00000000	This page has not been used for eight time periods
P2	11111111	The page has been used atleast once in each period
P3	11000100	The page has been used more recently than P4
P4	01110111	The page has been used less recently than P3

- Interpret these **8-bit bytes as unsigned integers**, the page with the lowest number is the LRU page, and it can be replaced.
- Note that the numbers are not guaranteed to be unique however, either replace (swap out) all pages with the smallest value or use the FIFO method to choose among them.
- The number of bits of history included in the shift register can be varied, and is selected (depending on the hardware available) to make the updating as fast as possible.
- In the extreme case, the number can be reduced to zero, leaving only the reference bit itself. This algorithm is called the **second chance page-replacement algorithm**

LRU Approximation Algorithms -Second-chance algorithm

- Basic algorithm of second-chance replacement is a FIFO replacement algorithm.
- When a page has been selected, check its reference bit. If the value is 0, proceed to replace this page;
- But if the reference bit is set to 1, give the page a second chance and move on to select the next FIFO page.
- When a page gets a second chance, its reference bit is cleared, and its arrival time is reset to the current time.
- Thus, a page that is given a second chance will not be replaced until all other pages have been replaced (or given second chances).
- In addition, if a page is used often enough to keep its reference bit set, it will never be replaced.

Second-chance Algorithm Implementation



- Implement the second-chance algorithm (sometimes referred to as the clock algorithm) is as a circular queue.
- A pointer (that is, a hand on the clock) indicates which page is to be replaced next.
- When a frame is needed, the pointer advances until it finds a page with a 0 reference bit. As it advances, it clears the reference bits .
- Once a victim page is found, the page is replaced, and the new page is inserted in the circular queue in that position.
- In the worst case, when all bits are set, the pointer cycles through the whole queue, giving each page a second chance.
- It clears all the reference bits before selecting the next page for replacement.

Dis Advantage: Second-chance replacement degenerates to FIFO replacement if all bits are set

Enhanced Second-Chance Algorithm

Consider both Reference bit and the Modify bit as an ordered pair.



Reference bit is set when a page is referenced



Modify bit is set when page is modified

Four possible cases

Order ed pair	Refere nce bit	Modi fy bit	What it means	Remarks
(0,0)	0	0	The page was neither used recently nor modified	Best page to replace
(0,1)	0	1	The page was not used recently, but it has been modified	Not the best page to be replaced, because to write this page to disk when replacement is done

(1,0)	1	0	The page was used recently but not modified	There are chances that this page will be used again soon
(1,1)	1	1	The page was used recently and also has been modified	There are chances that this page will be used again and write this page to disk when replacement is done

- Each page is in one of these four classes.
- When page replacement is called for, use the same scheme as in the clock algorithm; but instead of examining whether the page is pointing to the reference , examine the class to which that page belongs.

Counting Algorithms

- Keep a counter of the number of references that have been made to each page
- **Least Frequently Used (LFU) Algorithm:**
 - Replaces page with smallest count
- **Most Frequently Used (MFU) Algorithm:**
 - Based on the argument that the page with the smallest count was probably just brought in and has yet to be used

Page-Buffering Algorithms

Keep a pool of free frames:

- When a page fault occurs, a victim frame is chosen as before.
- The desired page is read into a free frame from the pool before the victim is written out.
- This procedure allows the process to restart as soon as possible, without waiting for the victim page to be written out.
- When the victim is later written out, its frame is added to the free-frame pool.

There are two ways of modifications in the algorithm:

First Method : To maintain a list of modified pages.

Whenever the paging device is idle, a modified page is selected and is written to secondary storage. Its modify bit is then reset.

This scheme increases the probability that a page will be clean when it is selected for replacement and will not need to be written out

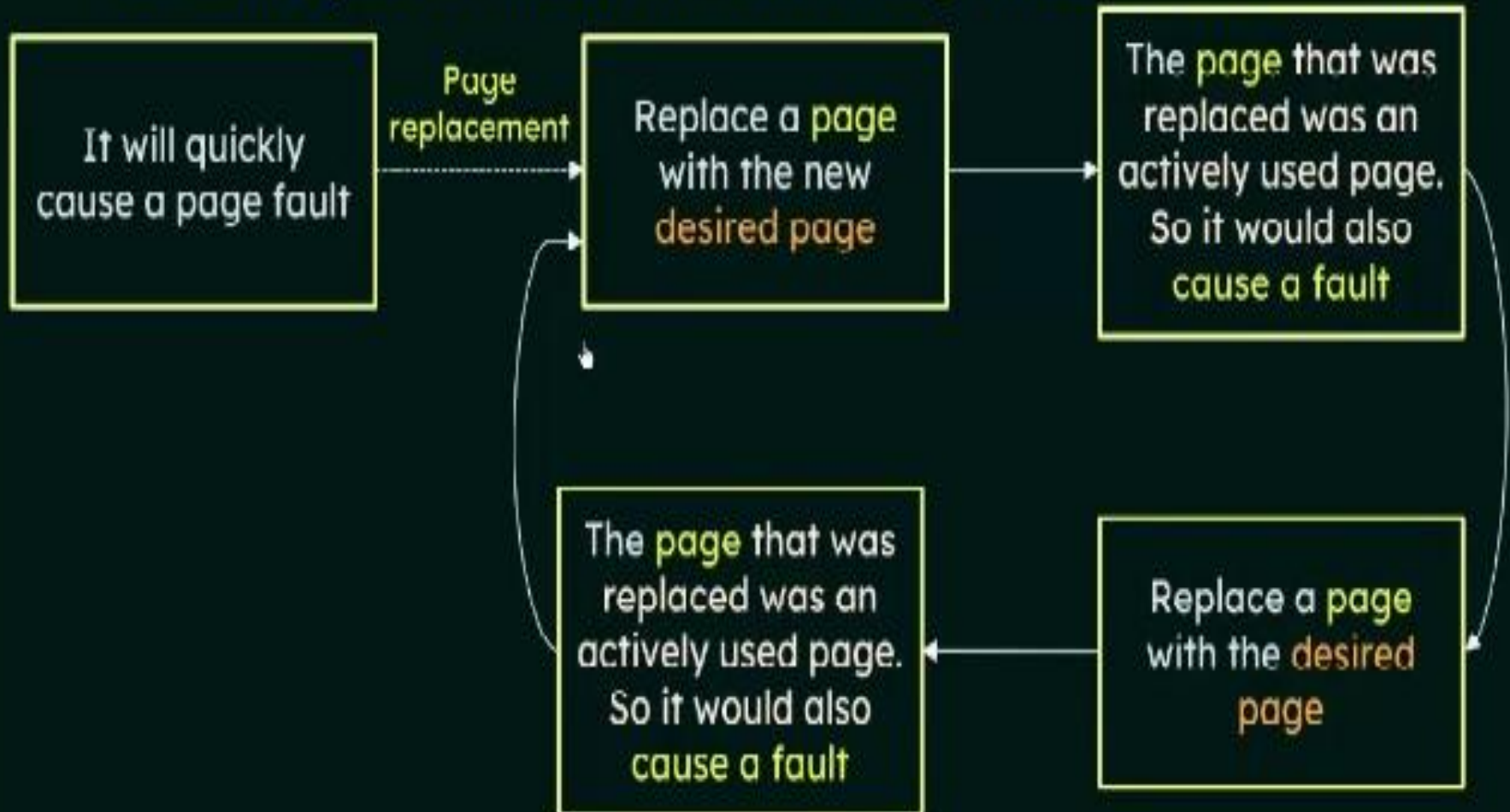
Second Method : Keep a pool of free frames but to remember which page was in each frame.

Since the frame contents are not modified when a frame is written to secondary storage, the old page can be reused directly from the free-frame pool if it is needed before that frame is reused.

No I/O is needed in this case. When a page fault occurs, first check whether the desired page is in the free-frame pool. If it is not, then select a free frame and read into it.

Thrashing

Consider a process that doesn't have enough frames for its execution.



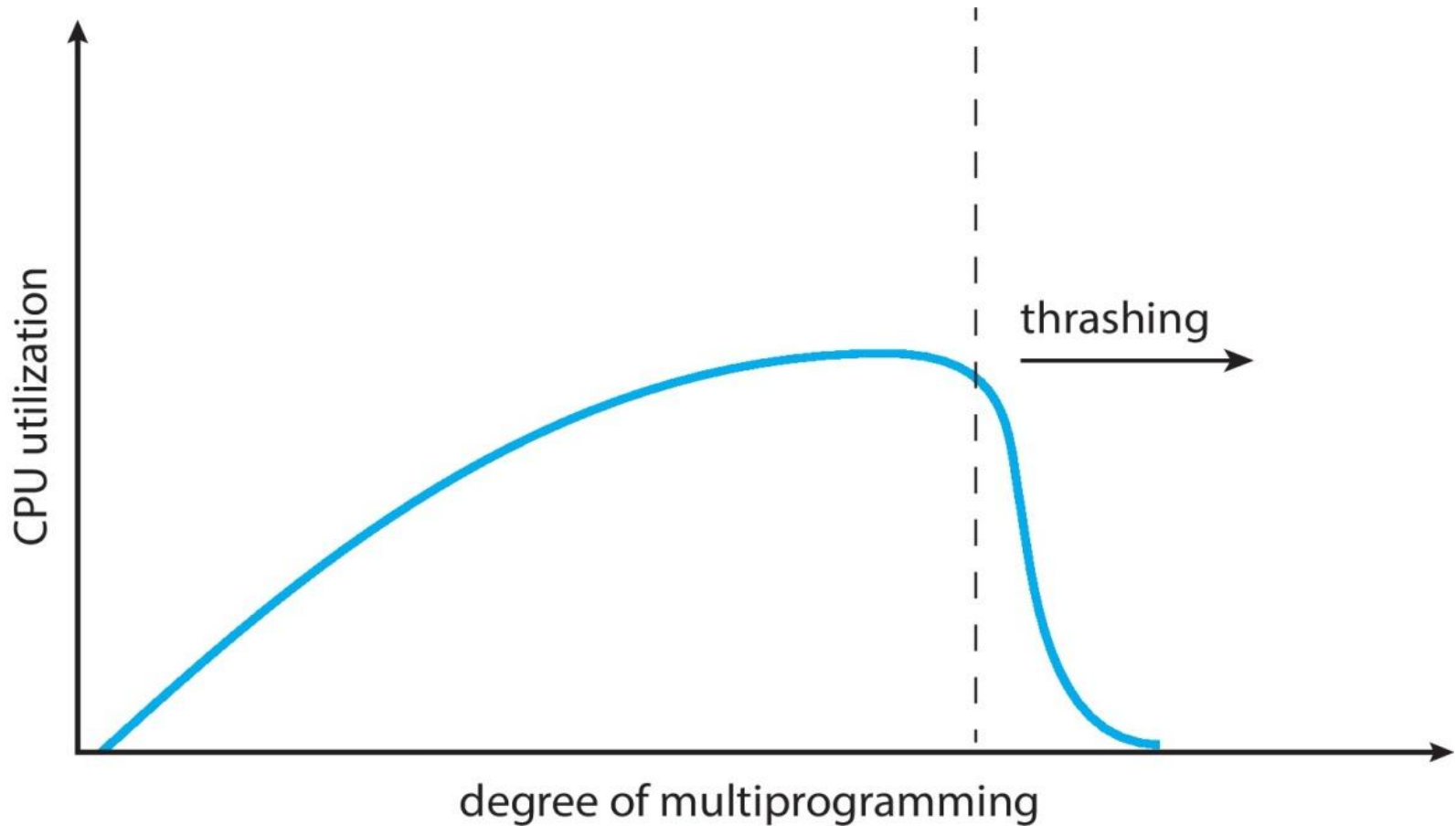
If a process is spending more time in paging rather than in executing, then the process is **Thrashing**.

Cause of Thrashing - severe performance problems

- The operating system monitors CPU utilization. If CPU utilization is too low, so increase the degree of multiprogramming by introducing a new process to the system.
- A page-replacement algorithm is used; it replaces pages for the new processes. It starts faulting and taking frames away from other processes.
- These faulting processes must use the paging device to swap pages in and out. As they queue up for the paging device, the ready queue empties.
- As processes wait for the paging device, CPU utilization decreases further.
- Thrashing occurs, the system throughput also decreasing tremendously.

Thrashing (Cont.)

Thrashing: A process is busy swapping pages in and out



Working-Set Model

To prevent Thrashing , provide a process with as many frames as it needs.

To know about the number of frames needed for each process is by using Working Set Model.

This strategy checks how many frames a process is actually using. This approach defines the **locality model of process execution**.

The locality model states that, **as a process executes, it moves from locality to locality**.

A **locality is a set of pages that are actively used together**. A running program is generally composed of several different localities, which may overlap.

For example, when a function is called, it defines a new locality .In this locality, memory references are made to the instructions of the function call, its local variables, and a subset of the global variables. When exit the function, the process leaves this locality, since the local variables and instructions of the function are no longer in active use.

Implementation : Use a parameter Δ which defines the working set window. The idea is to examine the most recent Δ page references. The set of pages in the most recent Δ page references is the **working set**.

If a page is in active use, it will be in the working set.

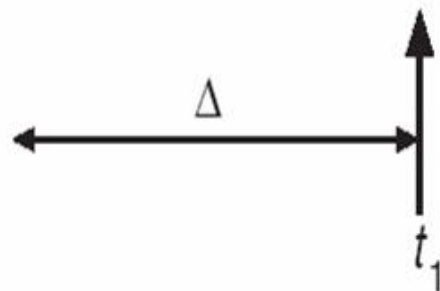
If it is no longer being used, it will drop from the working set Δ time units after its last reference.

Thus, the working set is an approximation of the program's locality.

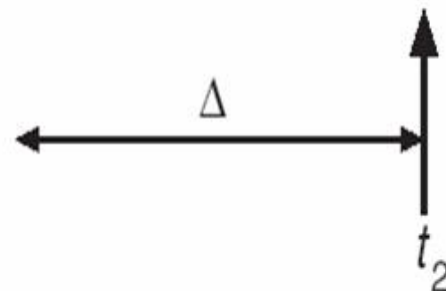
For example, given the sequence of memory references,

page reference table

... 2 6 1 5 7 7 7 7 5 1 6 2 3 4 1 2 3 4 4 4 3 4 3 4 4 4 1 3 2 3 4 4 4 3 4 4 4 ...



$$WS(t_1) = \{1, 2, 5, 6, 7\}$$



$$WS(t_2) = \{3, 4\}$$

if $\Delta = 10$ memory references, then the working set at time t_1 is $\{1, 2, 5, 6, 7\}$. By time t_2 , the working set has changed to $\{3, 4\}$.

The accuracy of the working set depends on the selection of Δ .

If Δ is too small, it will not cover the entire locality;

If Δ is too large, it may overlap several localities.

If Δ is infinite, the working set is the set of pages touched during the process execution.

The most important property of the working set is its size. If compute the working-set size, WSS_i [process 'i' needs WSS_i number of frames], for each process in the system.

So, total demand for frames in a system will be $D = \sum WSS_i$

From the example, $D = WSS_1 + WSS_2 = 5 + 2 = 7$ frames required

If the total demand is greater than the total number of available frames ($D > m$), then thrashing will occur.

In this case, the OS will select a process to suspend. The process pages are swapped out and its frames are reallocated to other processes. The suspended process can be restarted later.

The working set model prevents thrashing while keeping the degree of multiprogramming as high as possible. So, CPU utilization is improved.

Page-Fault Frequency

- The working-set model is successful, and knowledge of the working set can be useful for prepaging but it seems a clumsy way to control thrashing.
- A strategy that uses the **page-fault frequency (PFF)** takes a more direct approach.
- The specific problem is how to prevent thrashing. Thrashing has a high page-fault rate. Thus, to control the page-fault rate.
- When it is too high, that the process needs more frames.
- Conversely, if the pagefault rate is too low, then the process have too many frames.
- So, establish upper and lower bounds on the desired page-fault rate.
- If the actual page-fault rate exceeds the upper limit, allocate the process another frame.
- If the page-fault rate falls below the lower limit, remove a frame from the process. Thus, we can directly measure and control the pagefault rate to prevent thrashing.

Page-Fault Frequency

