

Operating Systems

Prepared by Dr. Alaa Eldeen Sayed Ahmed

IT263 / CYB510
Memory Management- part2

Lecture 8



- Objectives:
 - Non-Contiguous allocation (Paging)
 - Translation Lookaside Buffer (TLB) in Paging
 - Inverted Page Table
 - Demand Paging
 - Page Replacement

- Topics To Be Covered:
 - Paging
 - Define virtual memory and describe its benefits.
 - Illustrate how pages are loaded into memory using demand paging.
 - Apply the FIFO, optimal, and LRU page-replacement algorithms.

- Physical address space of a process can be noncontiguous;
 - 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. There is one page table per process.
- Backing store likewise split into pages
- Still have Internal fragmentation

- Operating system maintains a page table for each process
 - Contains the frame location for each page in the process
 - Memory address consist of a page number and offset within the page

page number	page offset
p	d
m - n	n

Lecture 08

Paging

Frame number	Main memory
0	A.0
1	A.1
2	A.2
3	A.3
4	D.0
5	D.1
6	D.2
7	C.0
8	C.1
9	C.2
10	C.3
11	D.3
12	D.4
13	
14	

Lecture 08

Paging Model of Logical and Physical Memory

0	0
1	1
2	2
3	3

Process A
page table

0	—
1	—
2	—

Process B
page table

0	7
1	8
2	9
3	10

Process C
page table

0	4
1	5
2	6
3	11
4	12

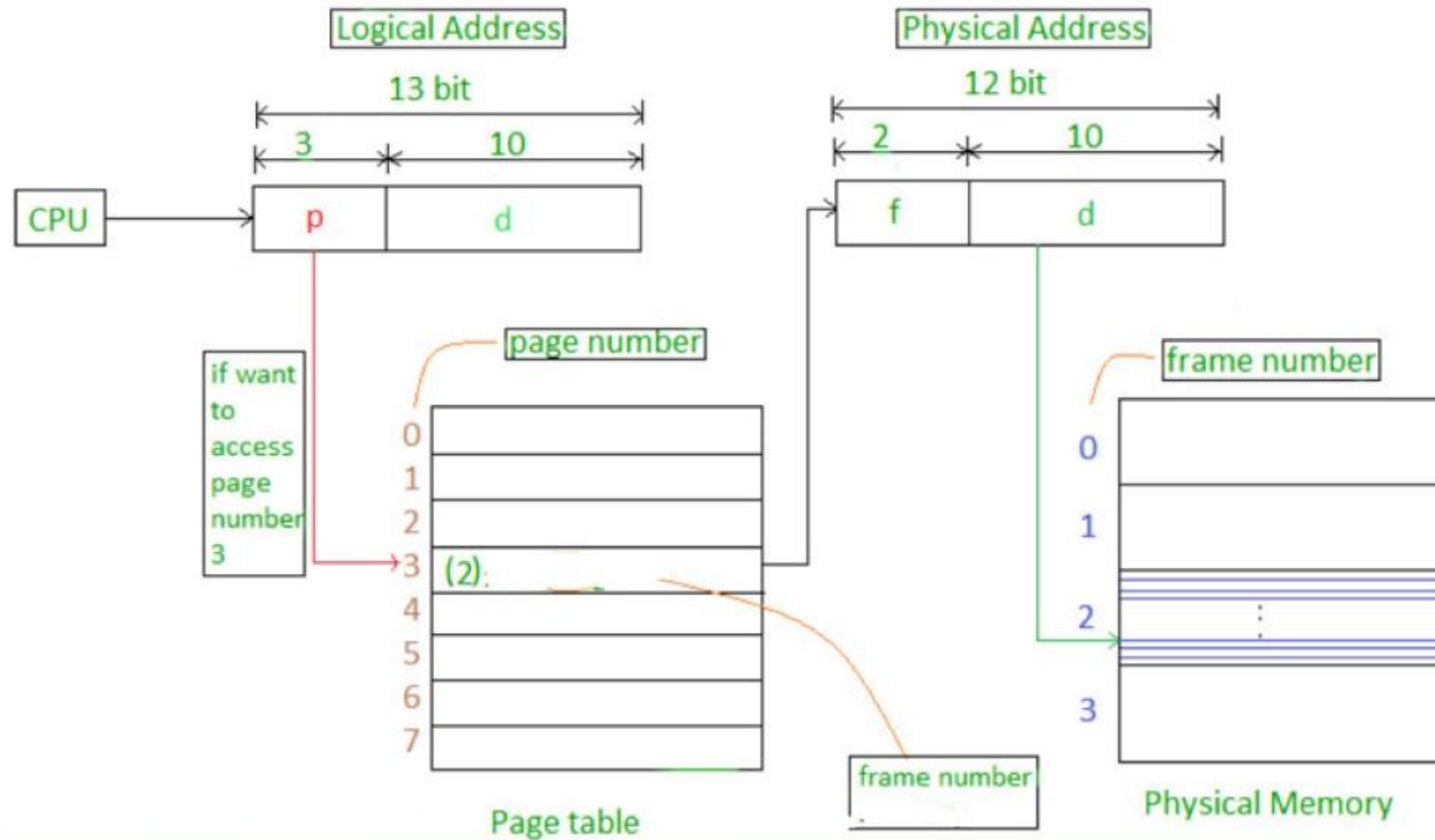
Process D
page table

13
14

Free frame
list

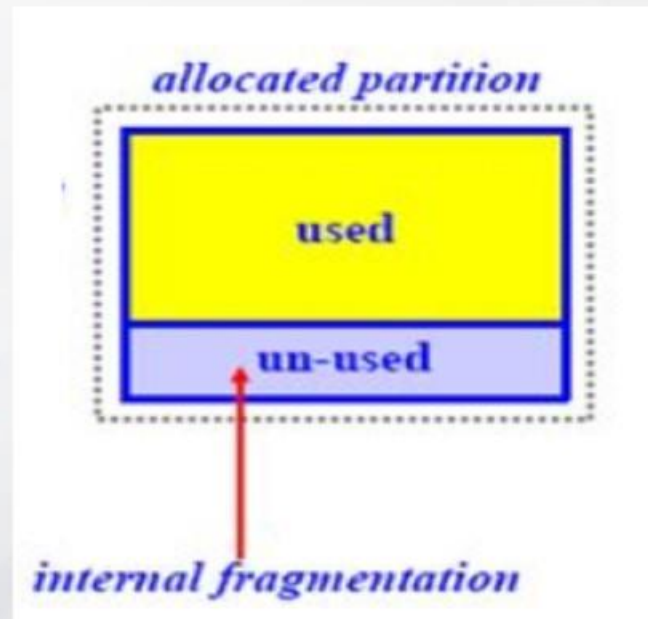
Lecture 08

Paging Hardware



Internal fragmentation

- **Internal Fragmentation** – allocated memory may be slightly larger than requested memory; this size difference is memory internal to a partition, but not being used



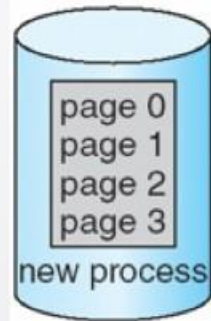
- Page size = 2,048 bytes
- Process size = 72,766 bytes
- 35 pages + 1,086 bytes
- Internal fragmentation of $2,048 - 1,086 = 962$ bytes
- Worst case fragmentation = 1 frame - 1 byte
- On average fragmentation = $1 / 2$ frame size
- So small frame sizes desirable
- But each page table entry takes memory to track

Lecture 08

Free Frames

free-frame list

14
13
18
20
15

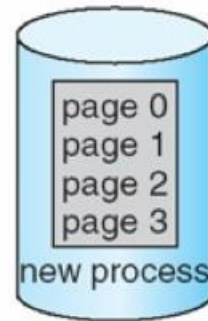


13
14
15
16
17
18
19
20
21

(a)

free-frame list

15



0 14
1 13
2 18
3 20

new-process page table

13
14
15
16
17
18
19
20
21

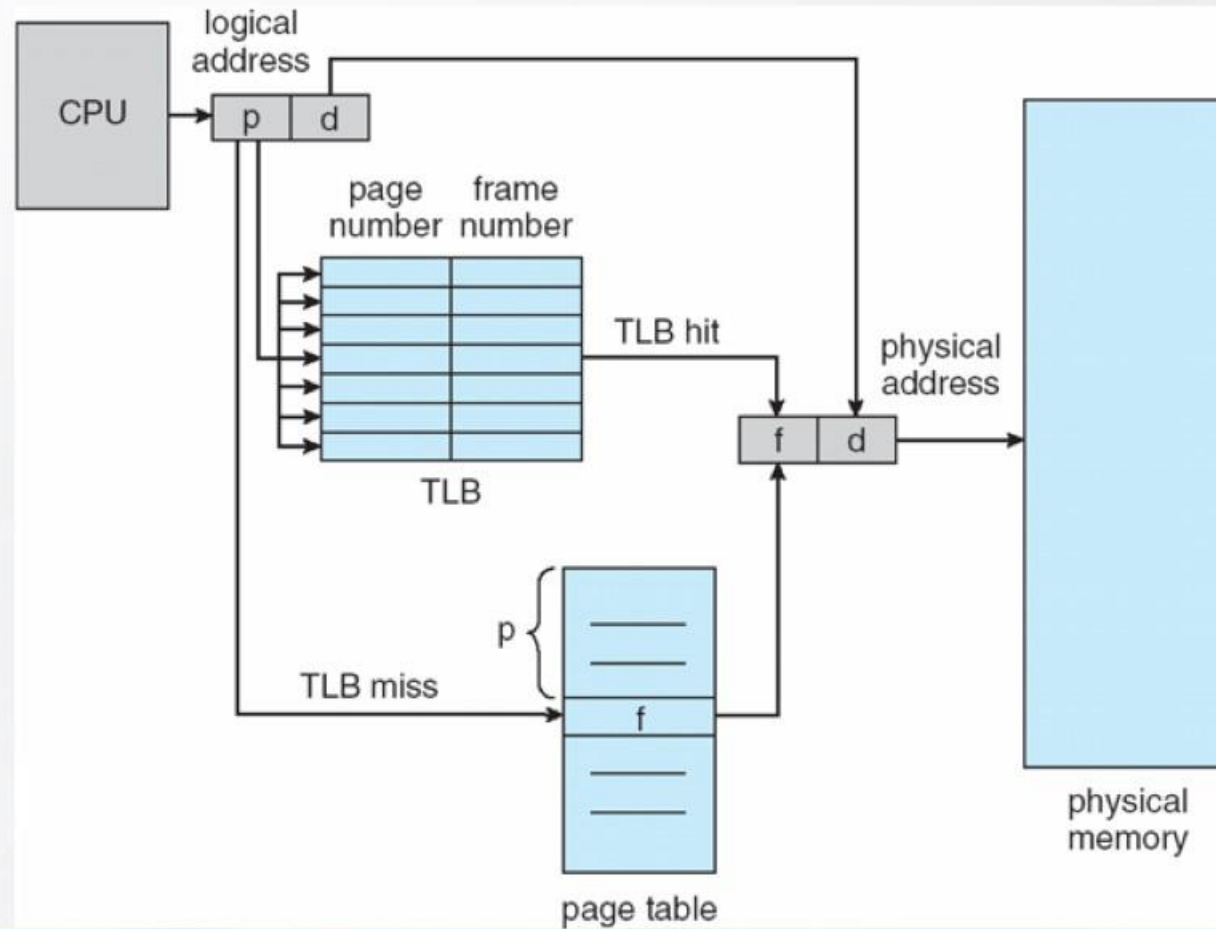
page 1
page 0
page 2
page 3

(b)

- Each virtual memory reference (translation from logical to physical address) can cause two physical memory accesses
 - The first is to fetch the page table
 - The second is to fetch the data
- Two memory Access cause a delay.
- To overcome this delay problem a high-speed special cache is set up for page table entries
 - called the TLB - Translation Lookaside Buffer
- TLB acts the same way as main memory cache. Contains those entries that have been recently accessed.

- Given a virtual (logical) address, processor examines the TLB
- If page table entry is present (a hit), the frame number is retrieved, and the real (physical) address is formed
- If page table entry is not found in the TLB (a miss), the page number is used to index the process page table

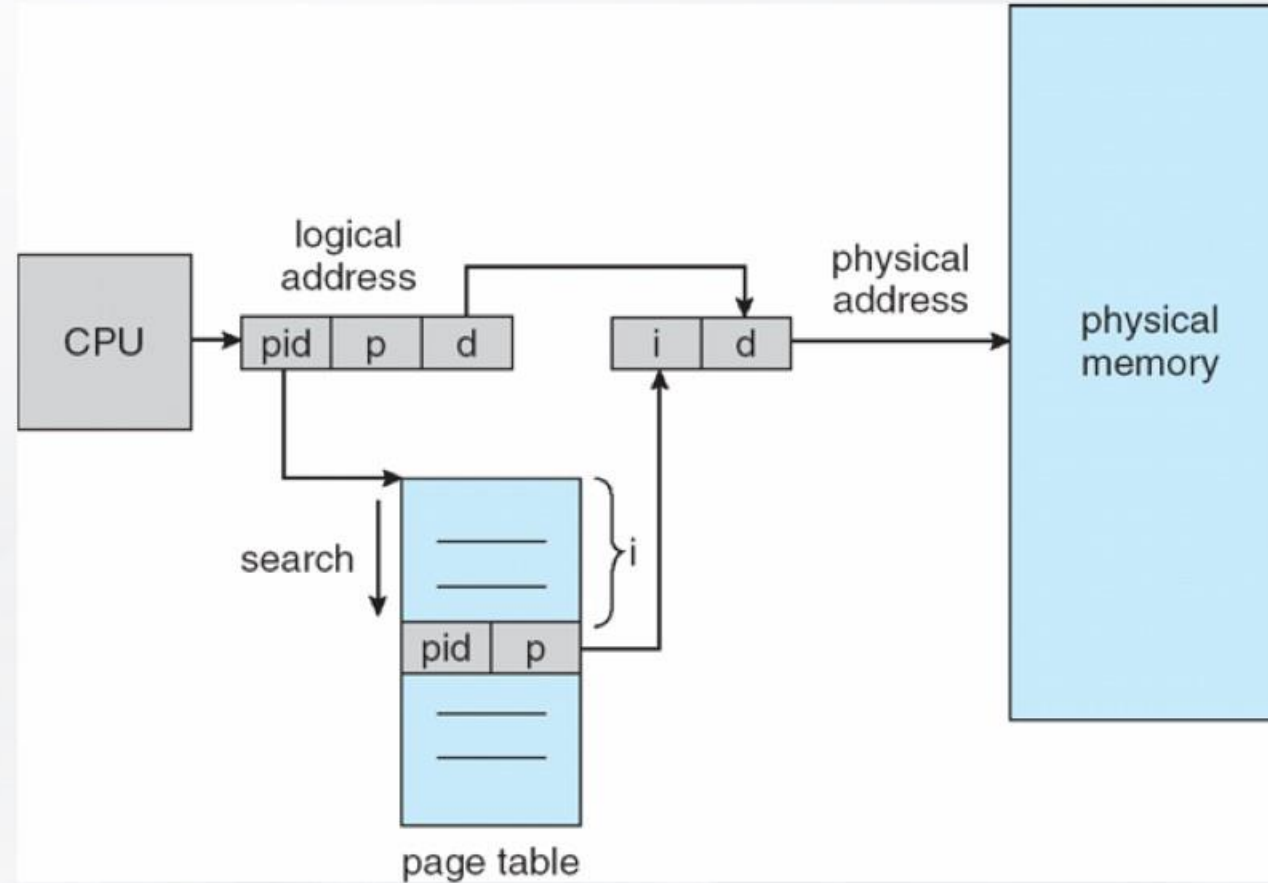
Lecture 08



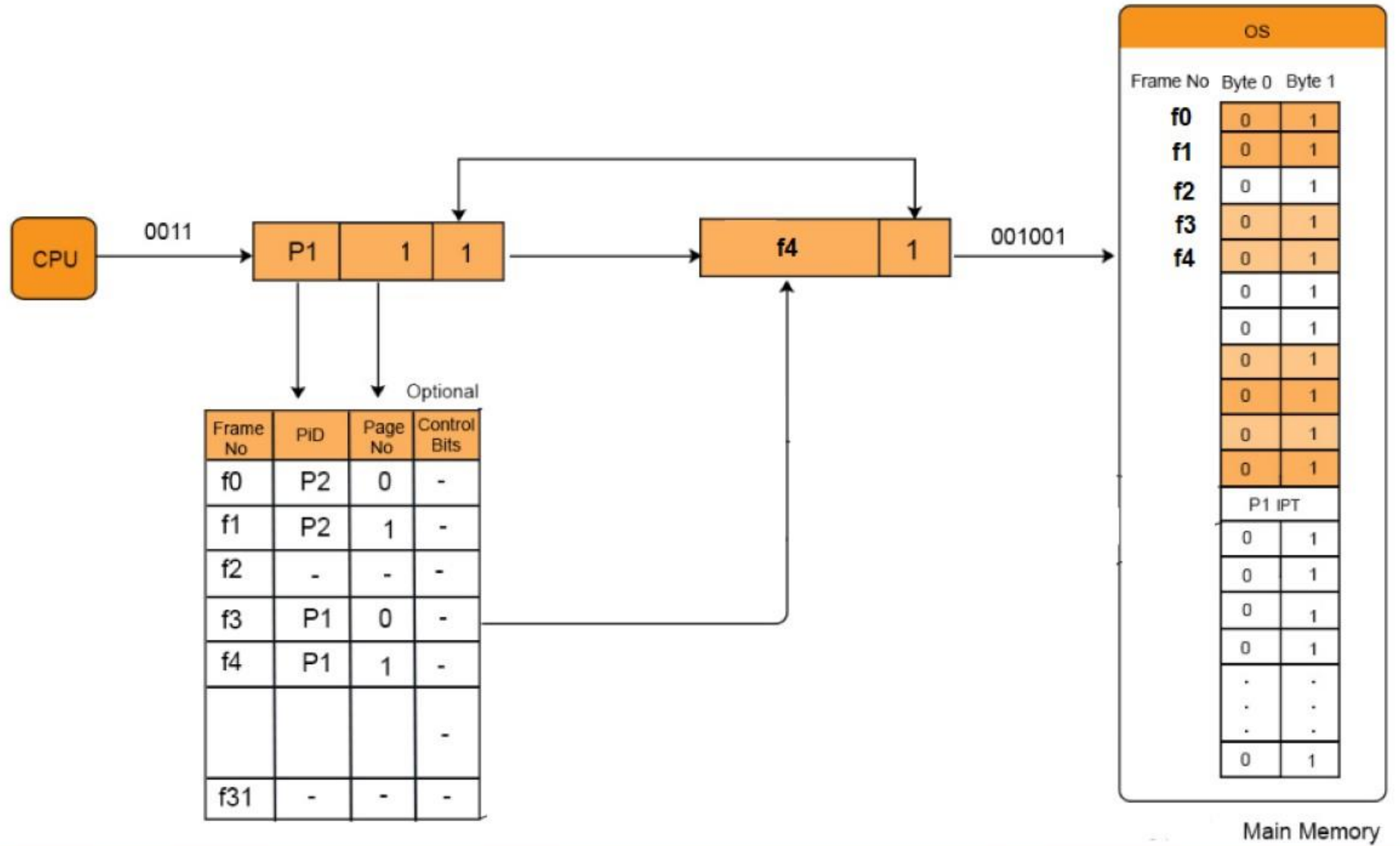
- Rather than each process having a page table and keeping track of all possible logical pages, track all physical pages
- One entry for each real page of memory
- Entry consists of the virtual address of the page stored in that real memory location, with information about the process that owns that page
- Decreases memory needed to store each page table, but increases time needed to search the table when a page reference occurs

Lecture 08

Inverted Page Table in Paging



Lecture 08

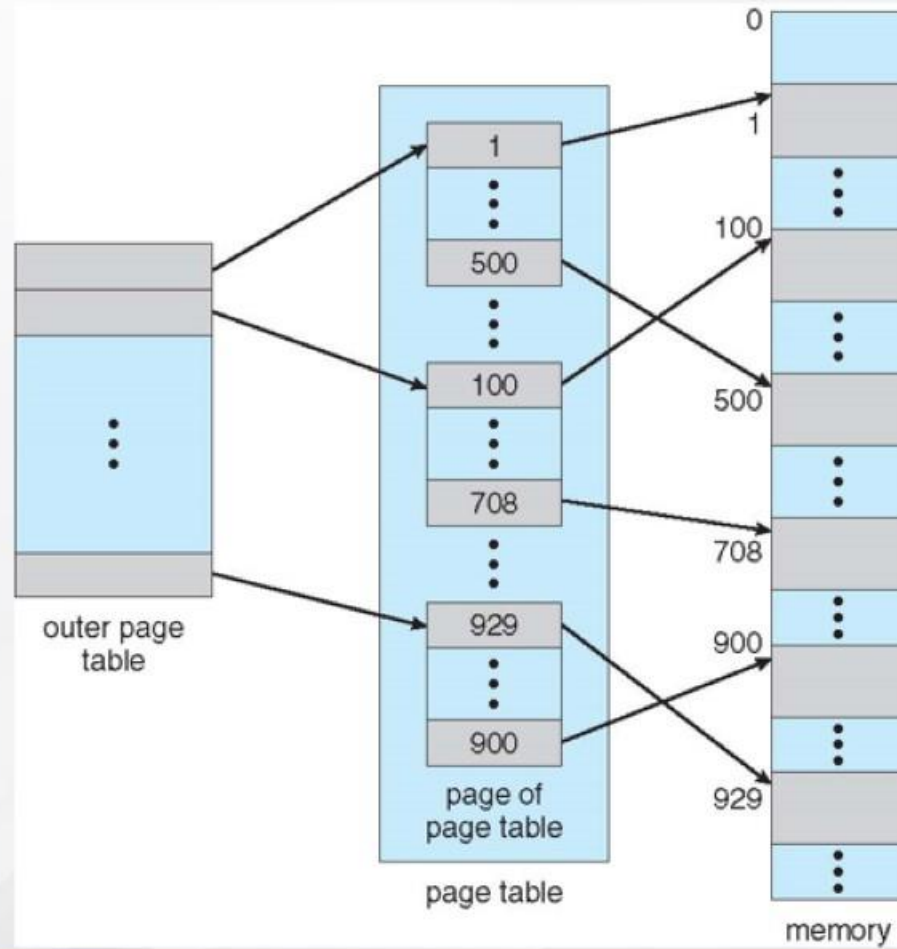


- Smaller page size, less amount of internal fragmentation
- But Smaller page size, more pages required per process
 - More pages per process means larger page tables
- Larger page tables means large portion of page tables in virtual memory

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

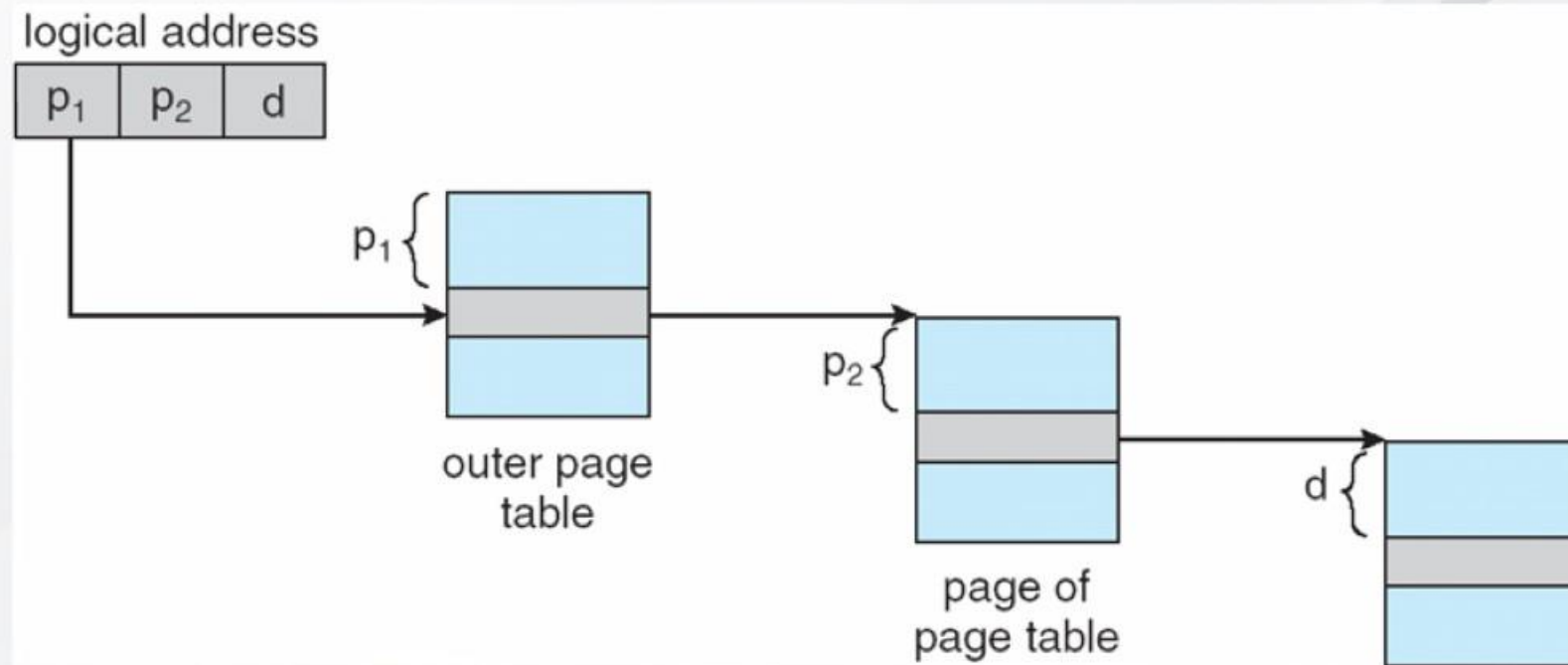
Lecture 08

Hierarchical Page Tables



Lecture 08

Two-level Paging Scheme



Lecture 08

Three-level Paging Scheme

outer page	inner page	offset
p_1	p_2	d
42	10	12

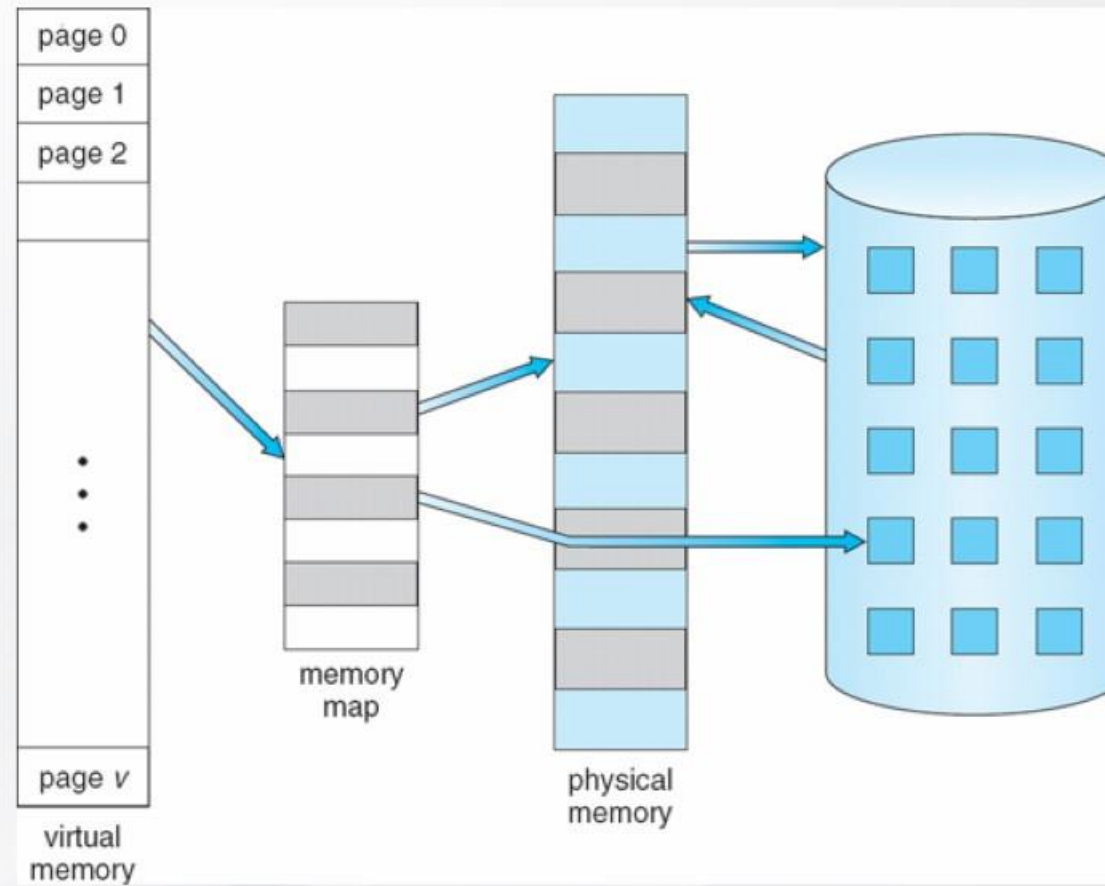
2nd outer page	outer page	inner page	offset
p_1	p_2	p_3	d
32	10	10	12

- Consider ability to execute partially-loaded program
 - Program no longer constrained by limits of physical memory
 - Each program takes less memory while running -> more programs run at the same time
 - Increased CPU utilization and throughput with no increase in response time or turnaround time
 - Less I/O needed to load or swap programs into memory
 - > each user program runs faster

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

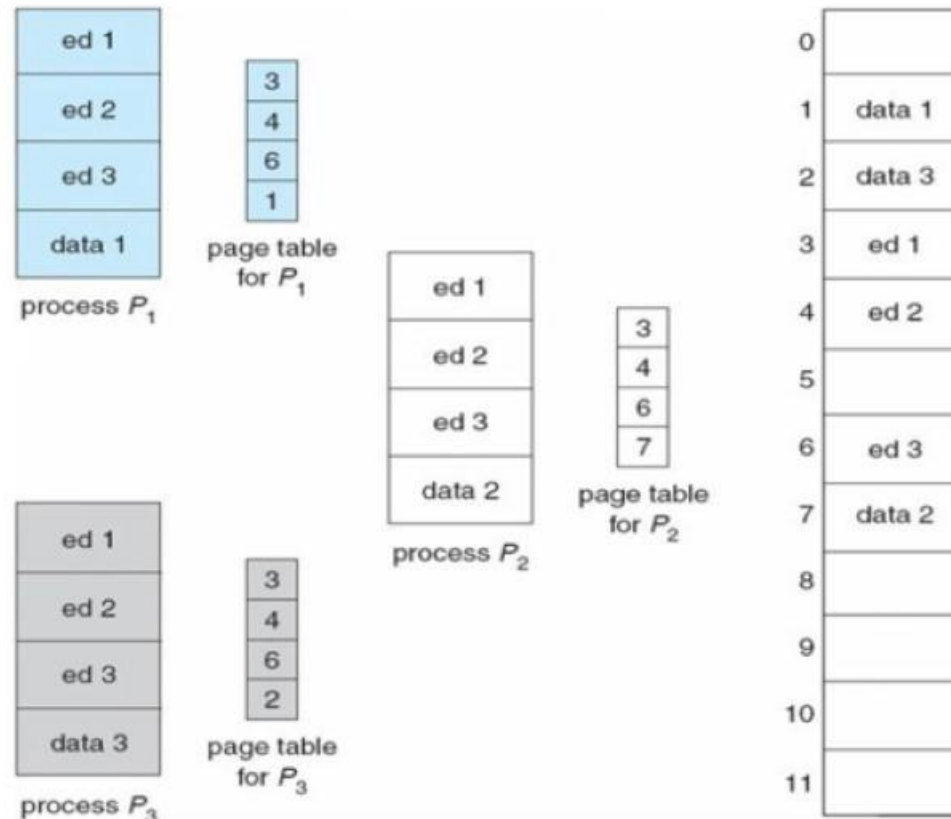
Lecture 08

Virtual Memory That is Larger Than Physical Memory



Lecture 08

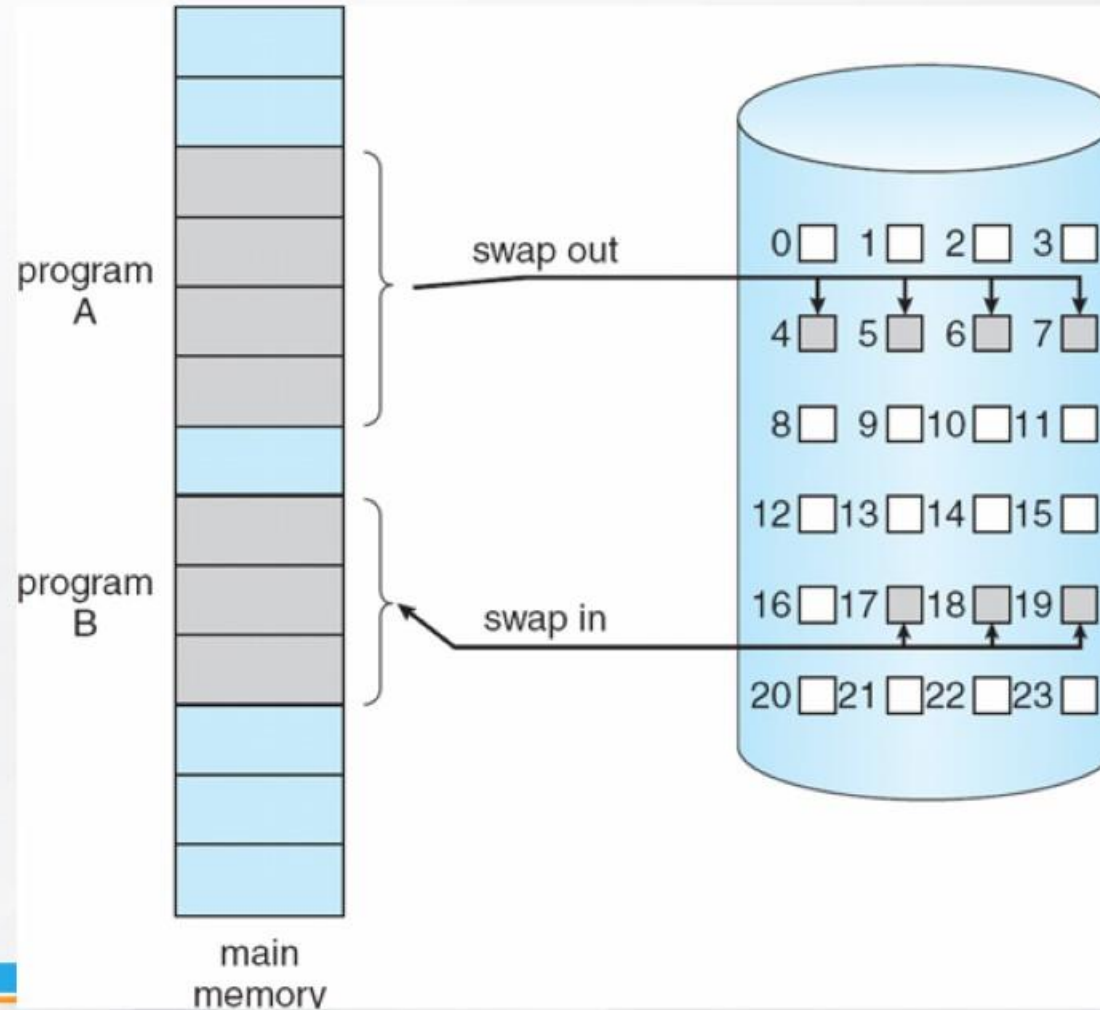
Shared pages example



- The **core enabling idea** of virtual memory systems:
 - A page is brought into memory only when needed
- Why?
 - Less I/O needed
 - Faster response, because we load only first few pages
 - Less memory needed for each process →
 - More processes can be admitted to the system
- How?
 - Process generates logical (virtual) addresses which are **mapped** to physical addresses using a **page table**. there is one page table per process.
 - If the requested page is not in memory, kernel brings it from hard disk by making a page fault.

Lecture 08

Transfer of a Paged Memory to Contiguous Disk Space



Lecture 08

Valid-Invalid Bit

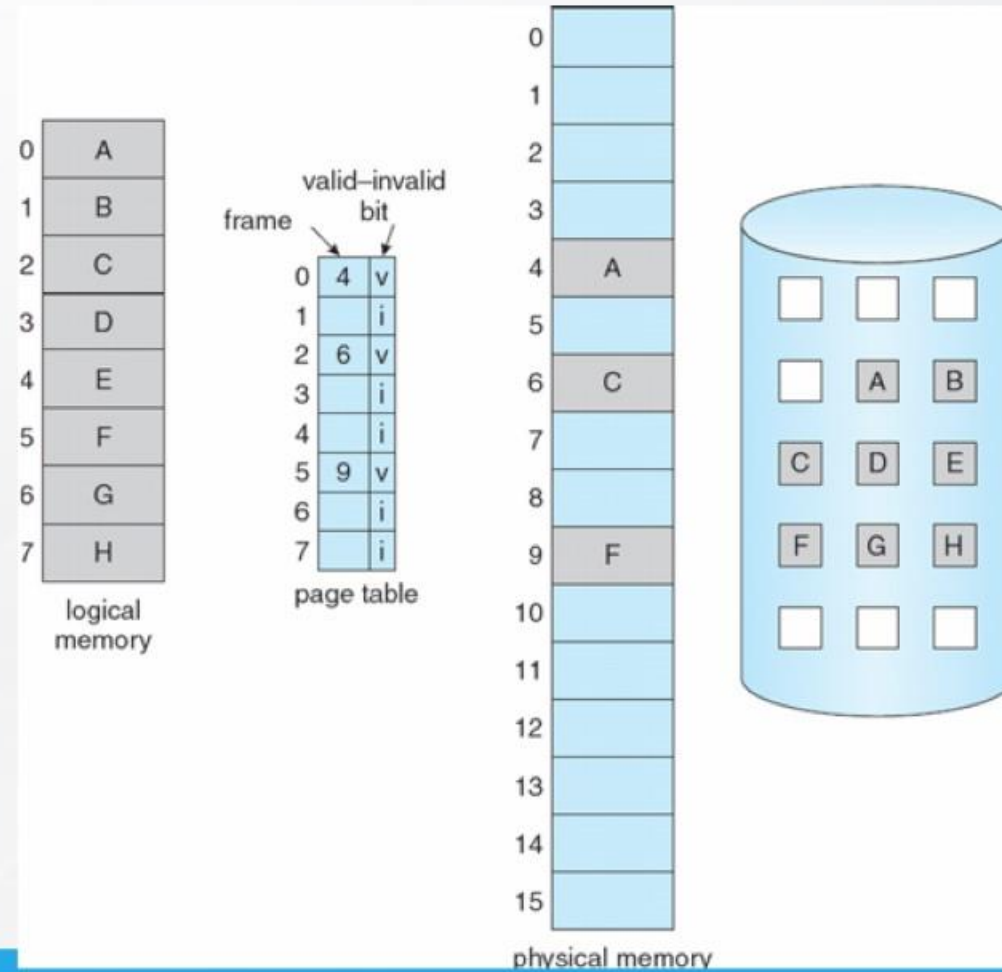
- Each page table entry has a valid-invalid bit:
 - **v** $\Rightarrow$ in-memory,
 - **i** $\Rightarrow$ not-in-memory
 - Initially, it is set to **i** for all entries
- During address translation, if valid-invalid bit is **i**, then it could be:
 - Illegal reference (outside process' address space) $\Rightarrow$ abort process
 - Legal reference but not in memory $\Rightarrow$ **page fault** (bring the page from disk)

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

page table

Lecture 08

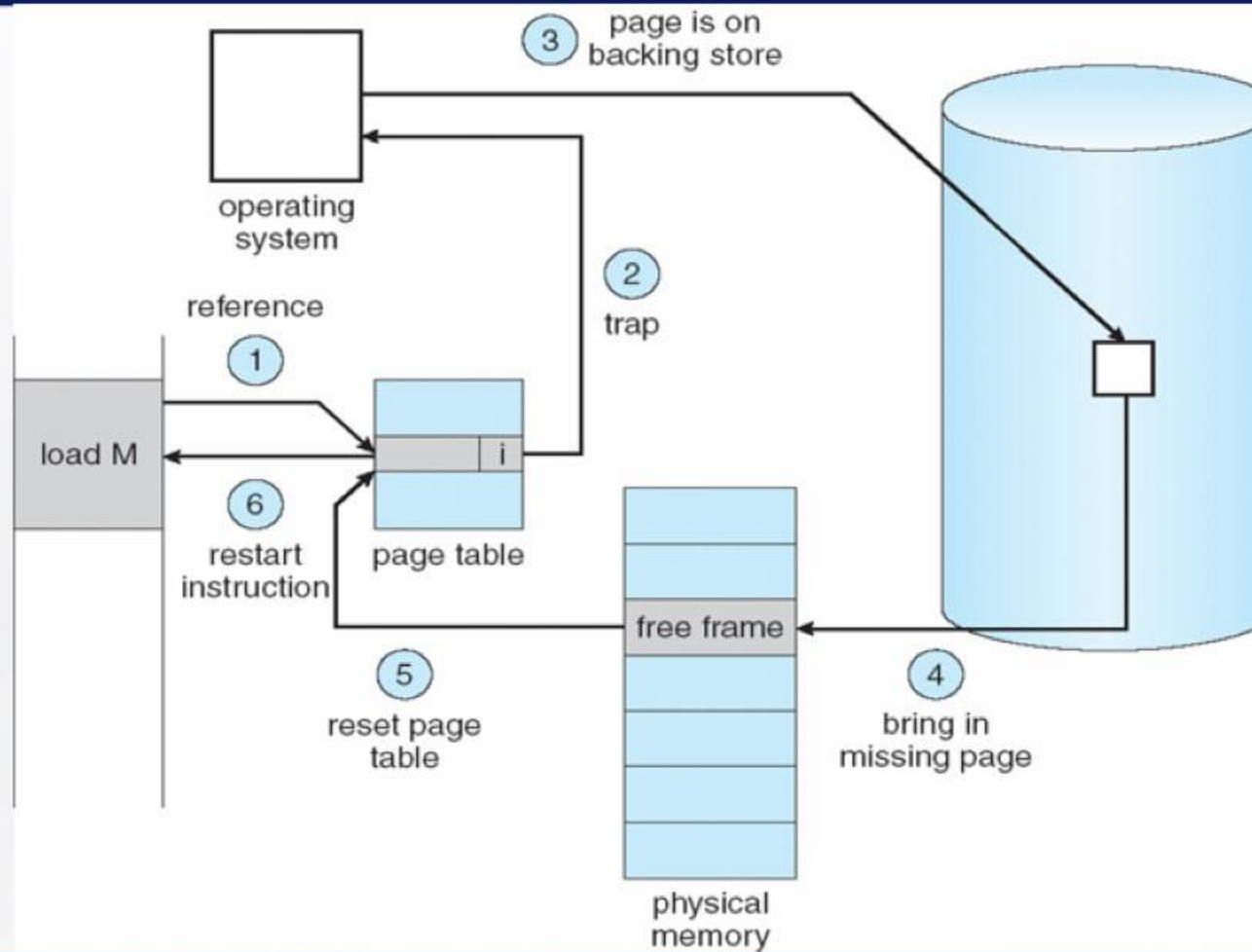
Page Table When Some Pages Are Not in Main Memory



- If there is a reference to a page, first reference to that page will trap to operating system:
 - **page fault**
- 1. Operating system looks at another table to decide:
 - Invalid reference $\Rightarrow$ abort
 - Just not in memory
- 2. Get empty frame
- 3. Swap page into frame
- 4. Reset tables
- 5. Set validation bit = **v**
- 6. Restart the instruction that caused the page fault

Lecture 08

Steps in Handling a Page Fault



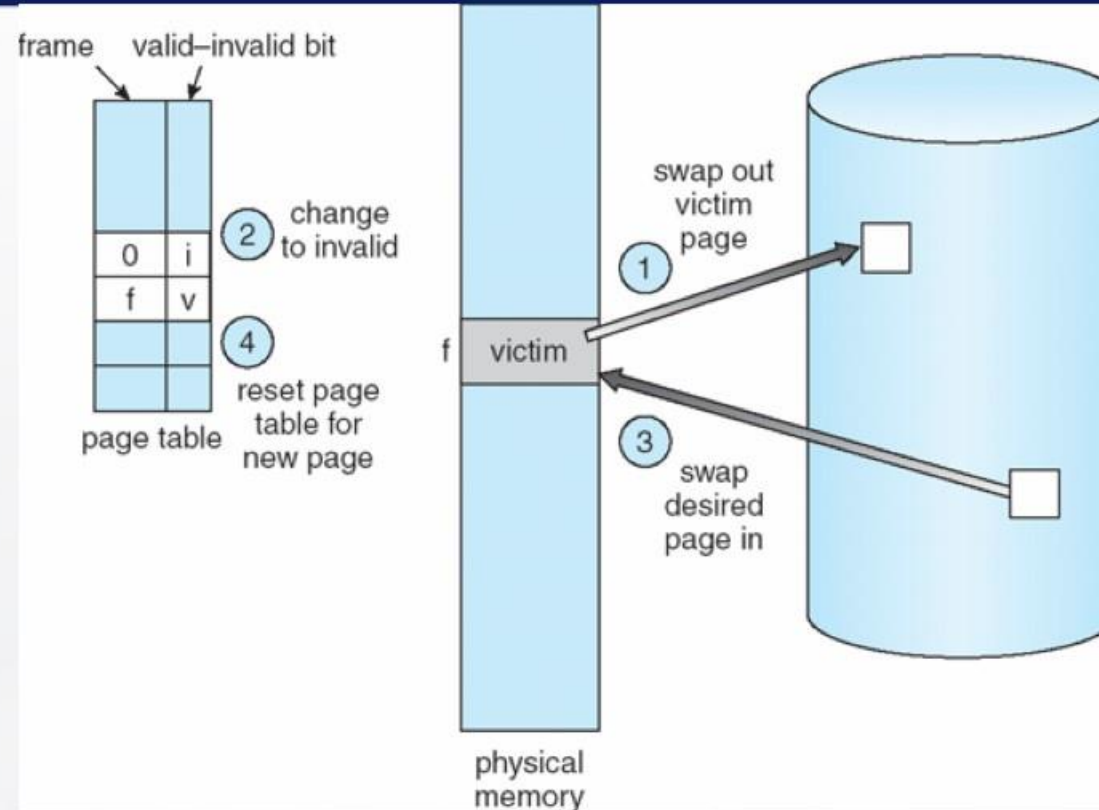
What happens if there is no free frame?

- Page replacement – is used to find some page in memory, but not really in use, then swap it out
 - algorithm
 - The key performance issue of using page replacement algorithm is to minimize number of page faults.

1. Find the location of the desired page on disk
2. Find a free frame:
 - If there is a free frame, use it
 - If there is no free frame, use a page replacement algorithm to select a **victim** frame
3. Bring the desired page into the (newly) free frame; update the page and frame tables
4. Restart the process

Lecture 08

Page Replacement



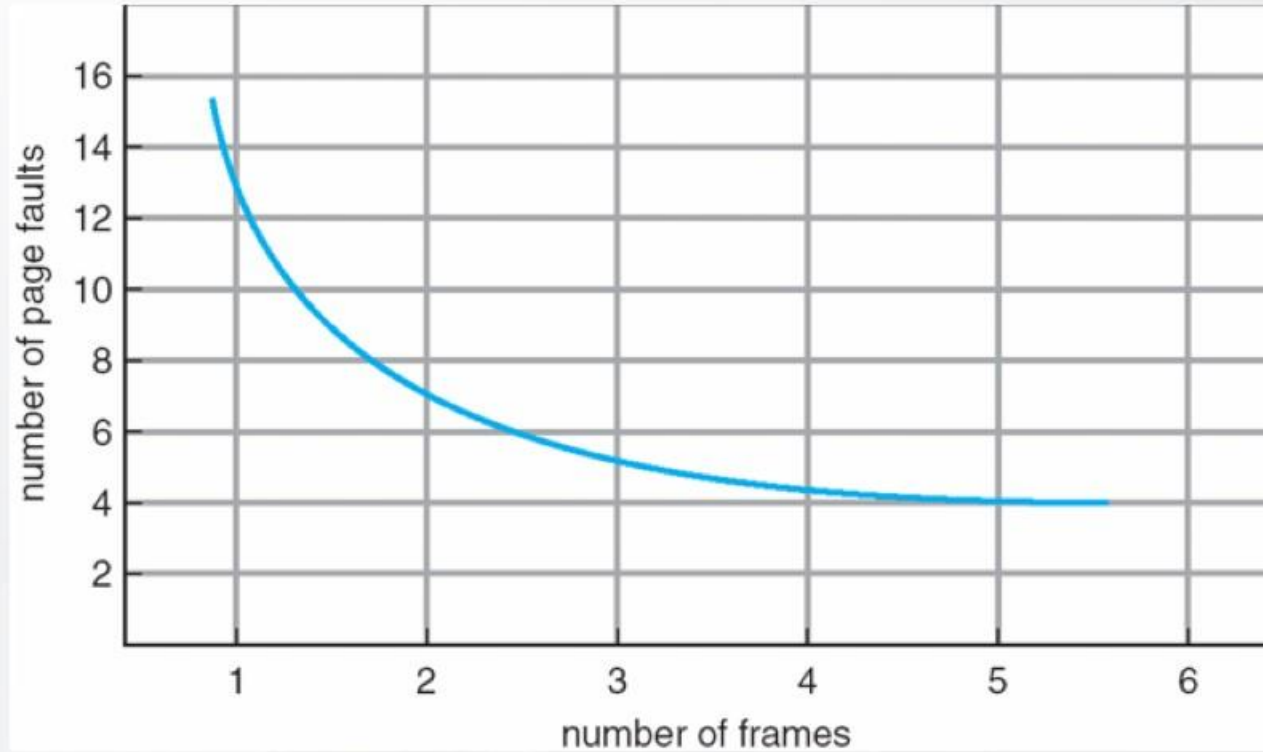
- ❑ **How do we choose the victim page? What would be the goal of this selection algorithm?**

- Want lowest page-fault rate
- Evaluate algorithm by running it on a particular string of memory references (reference string) and computing the number of page faults on that string
- In all our examples, the reference string is

1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

Lecture 08

Graph of Page Faults Versus The Number of Frames



First-In-First-Out (FIFO) Algorithm

- It Replaces page that has been in memory the longest (oldest page).
- Thus, it focuses on the length of time a page has been in memory rather than on how much the page is being use.
- FIFO's main asset is that it is simple to implement.
- Its behavior is not particularly well suited to the behavior of most programs. Few programs use it.

Lecture 08

First-In-First-Out (FIFO) Algorithm

- Reference string: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5
- 3 frames (3 pages can be in memory at a time per process)

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

9 page faults

- 4 frames

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

10 page faults

More frames are supposed to result in fewer page faults!

- Belady's Anomaly: more frames $\Rightarrow$ more page faults

Lecture 08

First-In-First-Out (FIFO) Algorithm

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

- ❑ Advantages

- ❖ Easy to understand and implement

- ❑ Disadvantages

- ❖ Performance may not always be good:
 - It suffers from **Belady's anomaly**

Optimal Page Replacement Algorithm

- An optimal page replacement algorithm has the lowest page-fault rate of all algorithms and will never suffer from Belady's anomaly.
- It replace page that **will not** be used for longest period of time
- 4-frame example: 1, 2, 3, 4, 1, 2, **5**, 1, 2, 3, **4**, 5

1	1	1	1	1	<u>4</u>
	2	2	2	2	2
		3	3	3	3
			4	<u>5</u>	5

- 6 page faults
- Used for comparing algorithms

Lecture 08

Optimal Page Replacement Algorithm

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

page frames

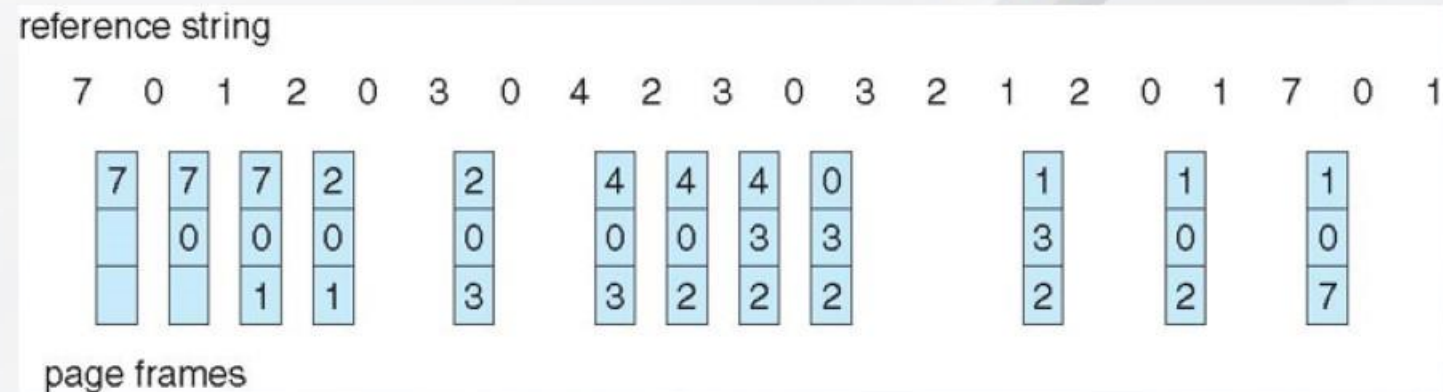
9 page faults

- Optimal Algorithm is difficult to implement, because it requires future knowledge of the reference string.
- As a result, optimal algorithms is used mainly for comparison purposes.
- For instance, it may be useful to know that, although a new page-replacement algorithm is not optimal, it is within 12.3 % of optimal at worst and withing 4.7% on average

Lecture 08

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



- 12 faults – better than FIFO but worse than OPT
- Generally good algorithm and frequently used



Thank You