

Operating Systems

Prepared by Dr. Alaa Mohamed

Module 5- Part2

جامعة الجميع الذكية

تسعى لتكون الكيان الأول الذي يتبنى مبدأ المجتمع التعليمي
التعاوني ويقدم العلم مقابل العمل لتنمية وبناء المجتمعات.

Memory Management Part-2

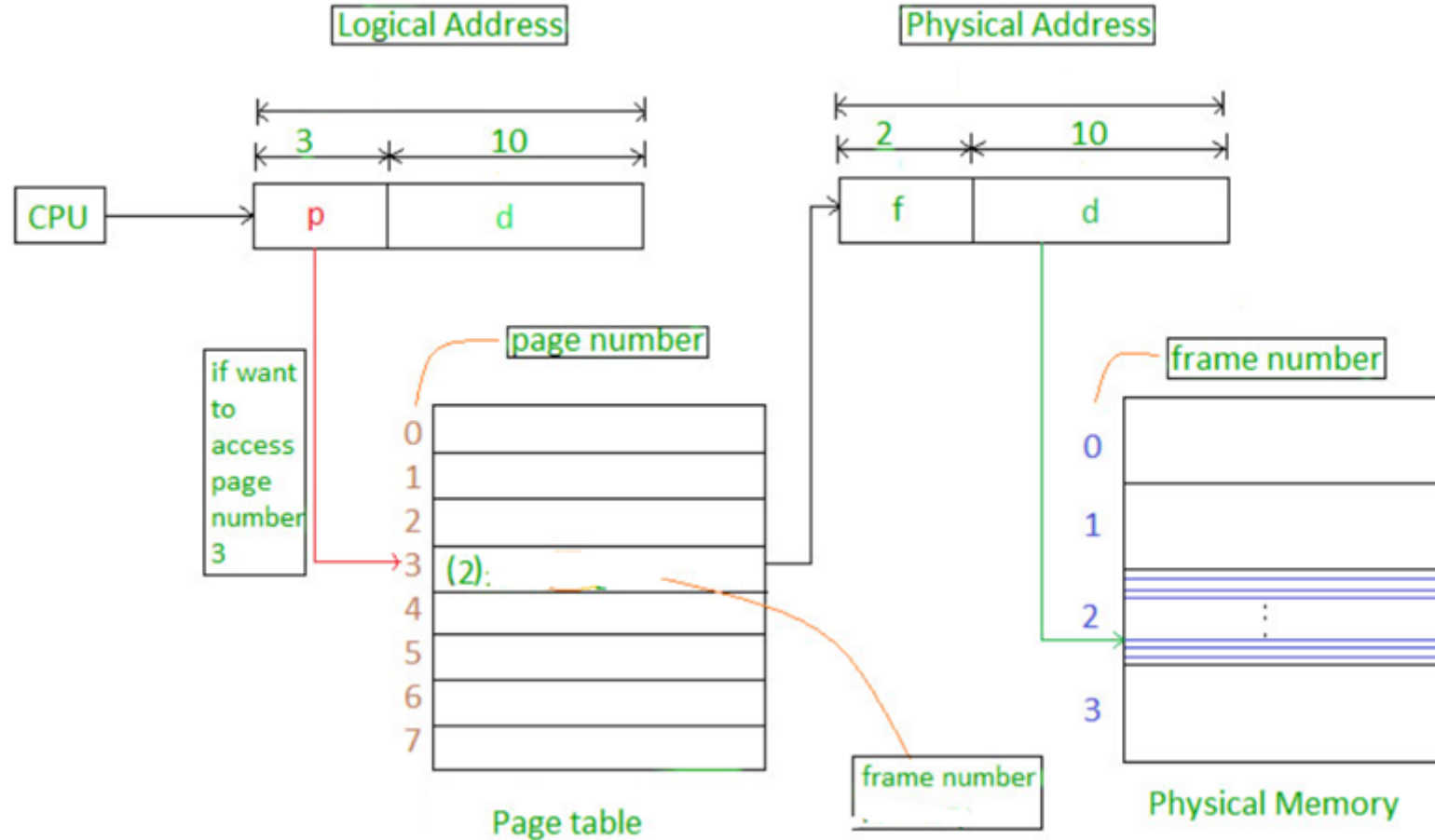


جامعة الجميع الذكية
EVERYONE'S SMART UNIVERSITY

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

- Topics To Be Covered:
 - 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.



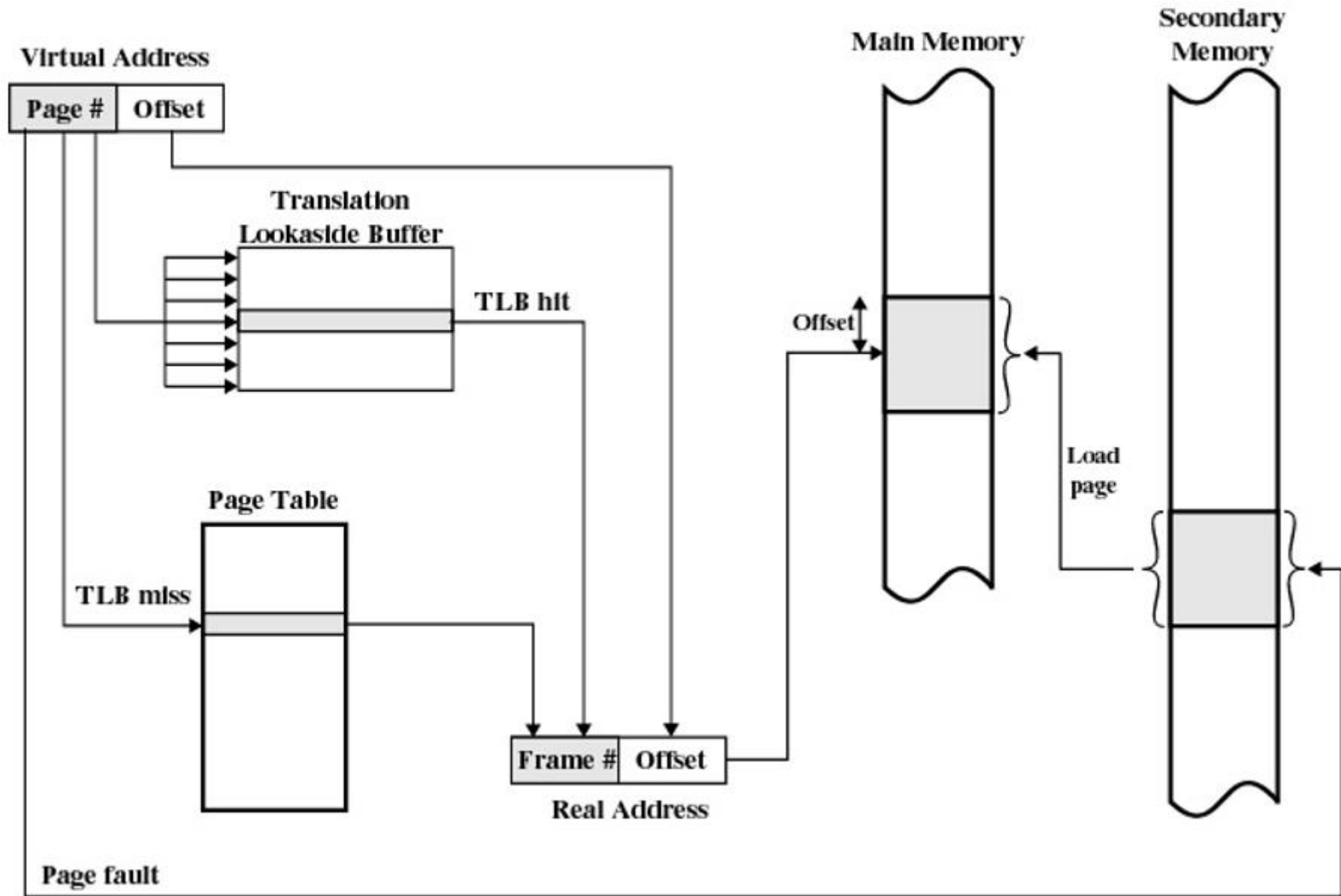


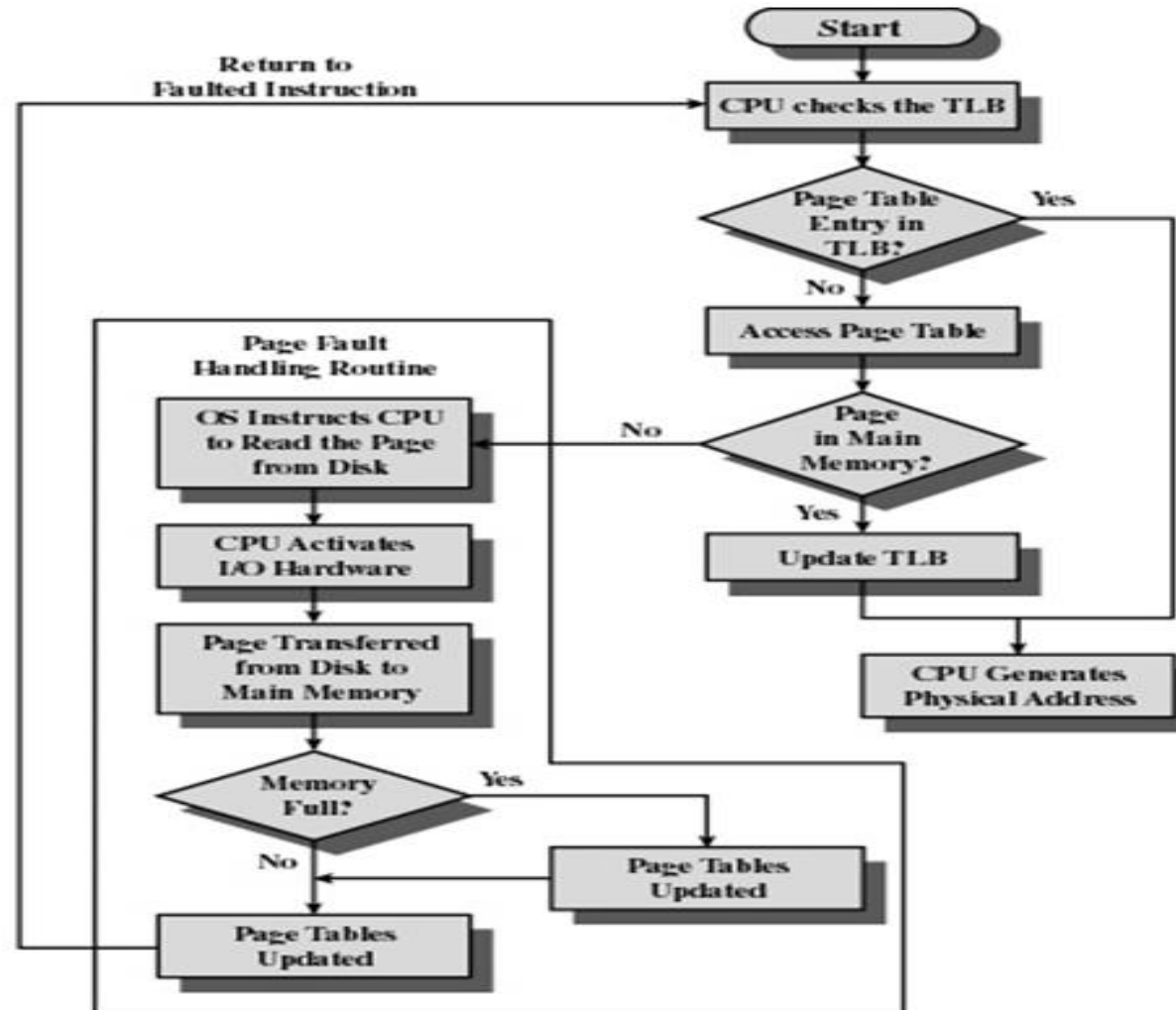
- Each virtual memory reference (translation from logical to physical address) can cause two physical memory accesses
 - The first is to fetch the page table
 - The second is to fetch the data
- Two memory Access cause a delay.
- To overcome this delay problem a high-speed special cache is set up for 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 address, processor examines the TLB
- If page table entry is present (a hit), the frame number is retrieved, and the real 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

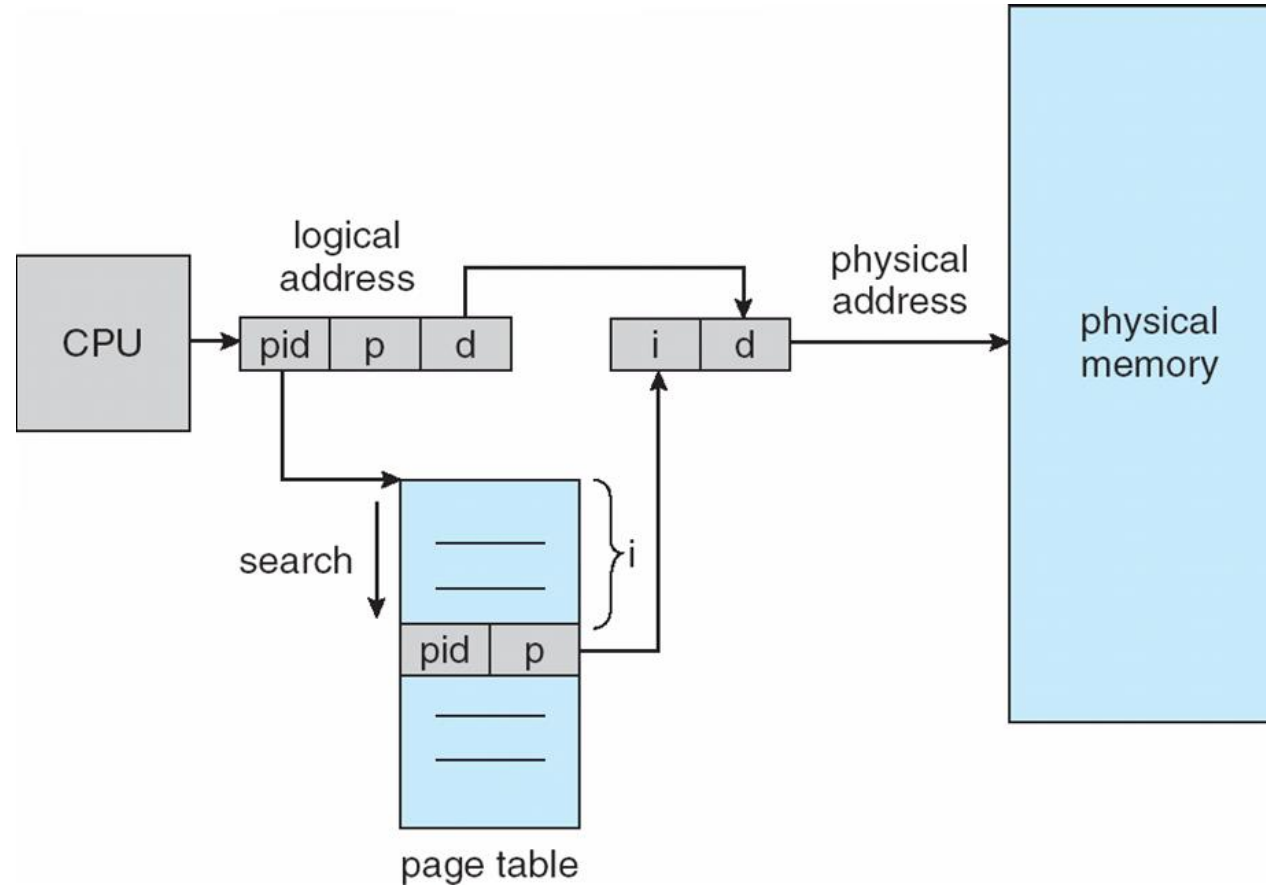
- First checks if page is already in main memory
 - if not in main memory a page fault is issued
- The TLB is updated to include the new page entry

Module 05-part2

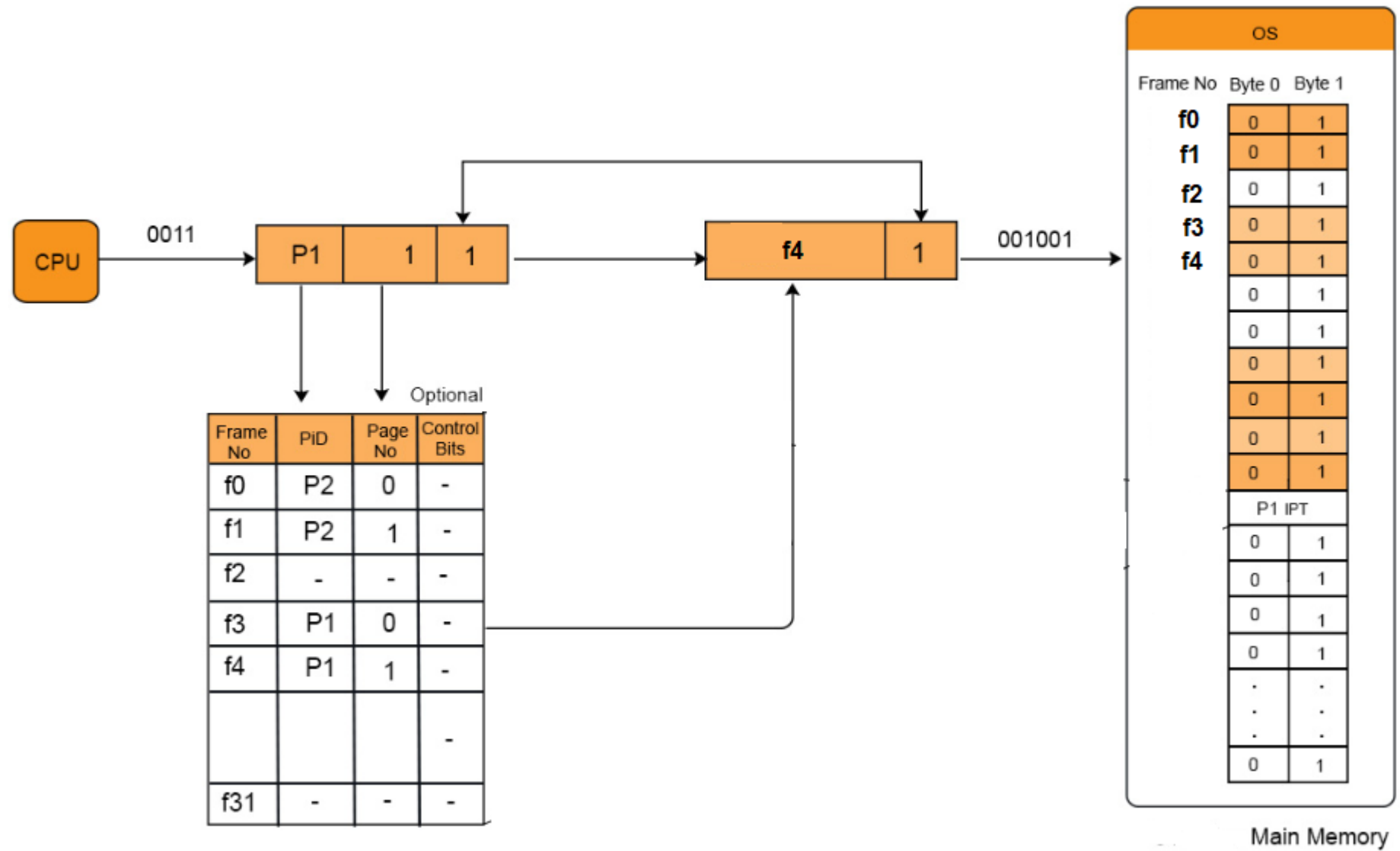




- Rather than each process having a page table and keeping track of all possible logical pages, track all physical pages
- One entry for each real page of memory
- Entry consists of the virtual address of the page stored in that real memory location, with information about the process that owns that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

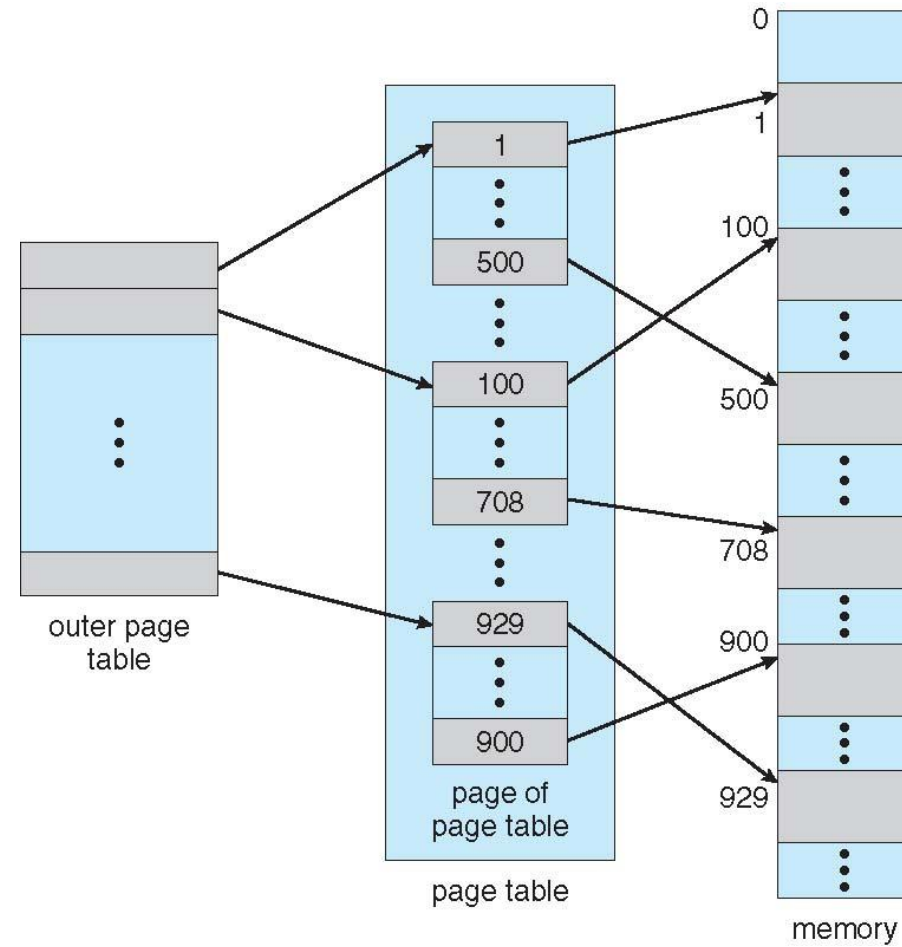


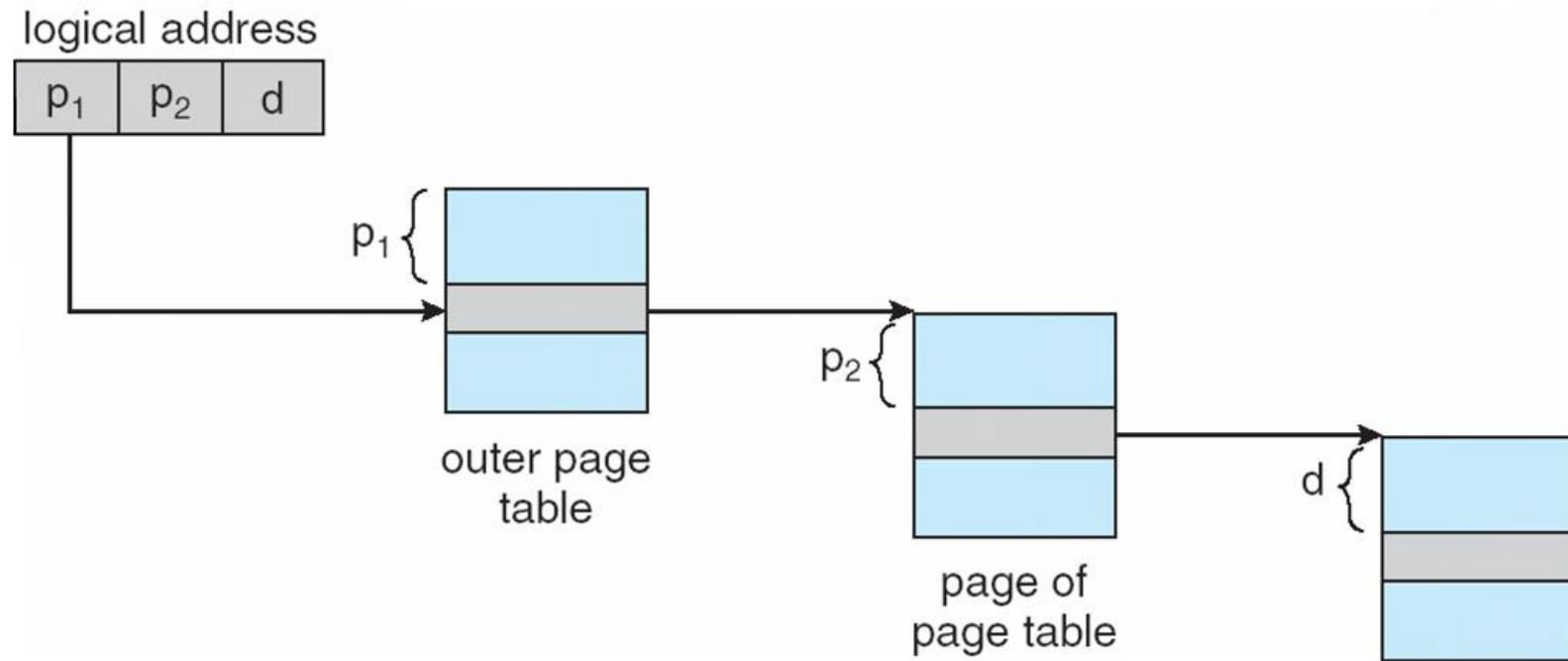
Module 05-part2



- 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





Three-level Paging Scheme

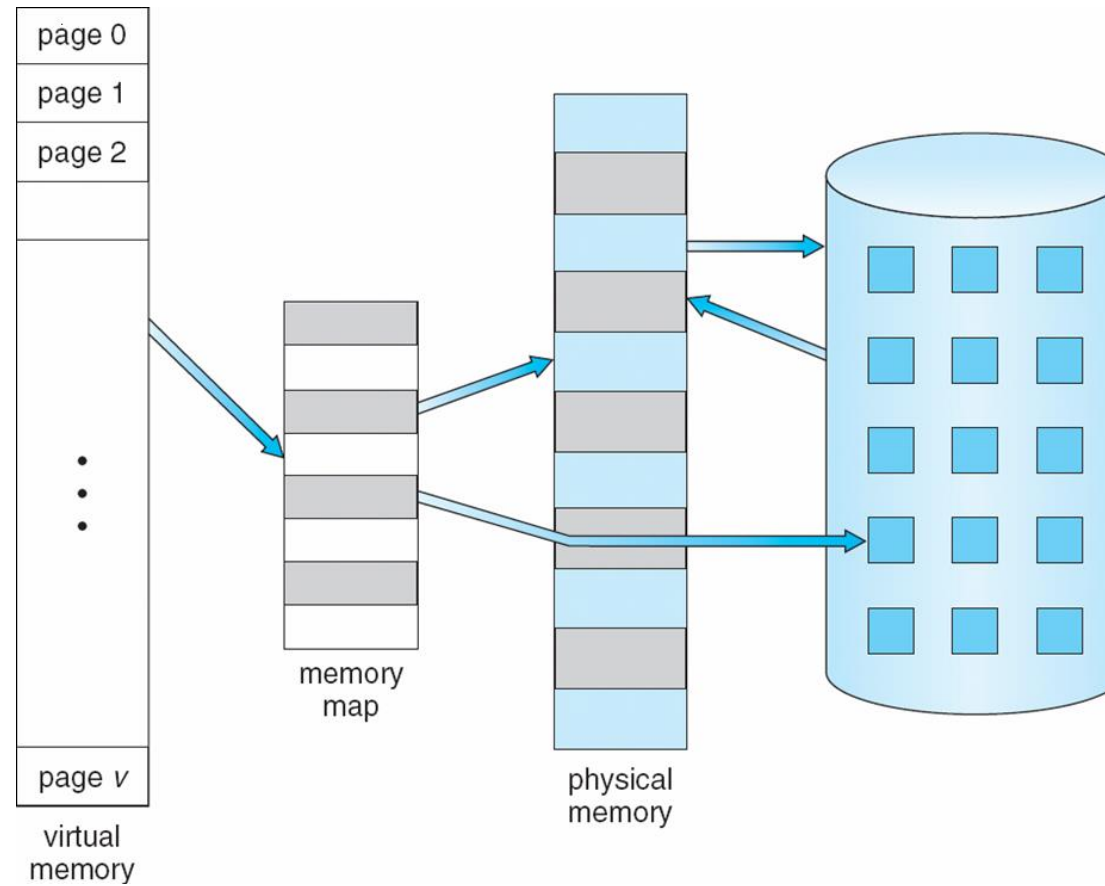
outer page	inner page	offset
p_1	p_2	d
42	10	12

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

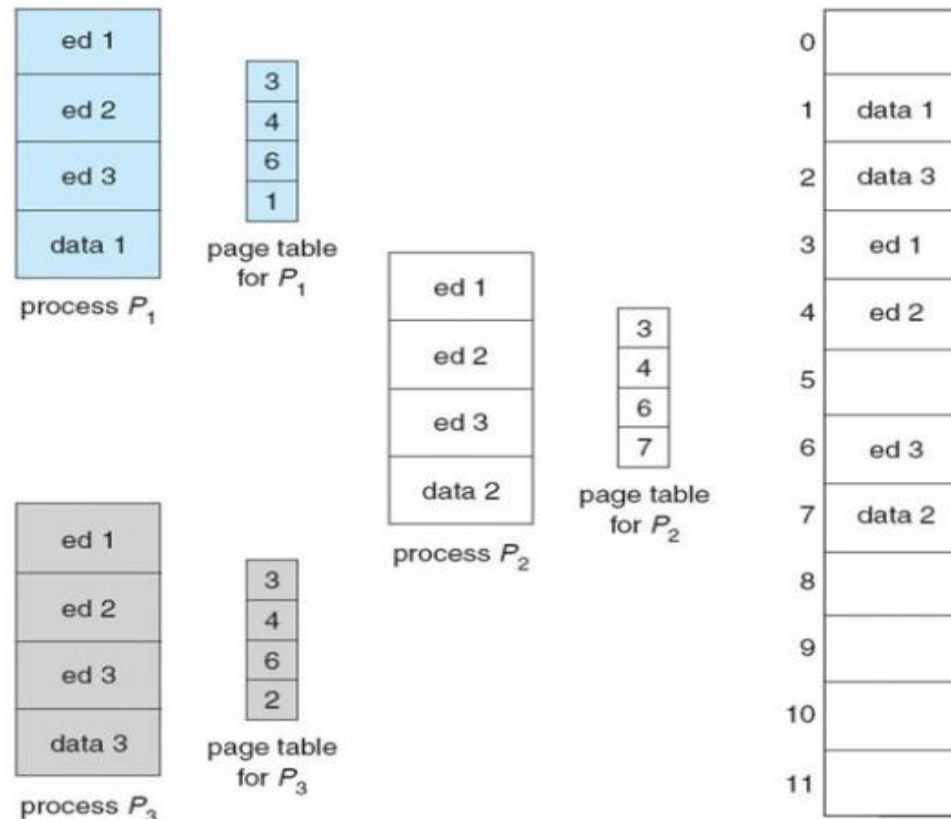
- Code needs to be in memory to execute, but entire program rarely used
 - Error code, unusual routines, large data structures
- Entire program code not needed at same time

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

- **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



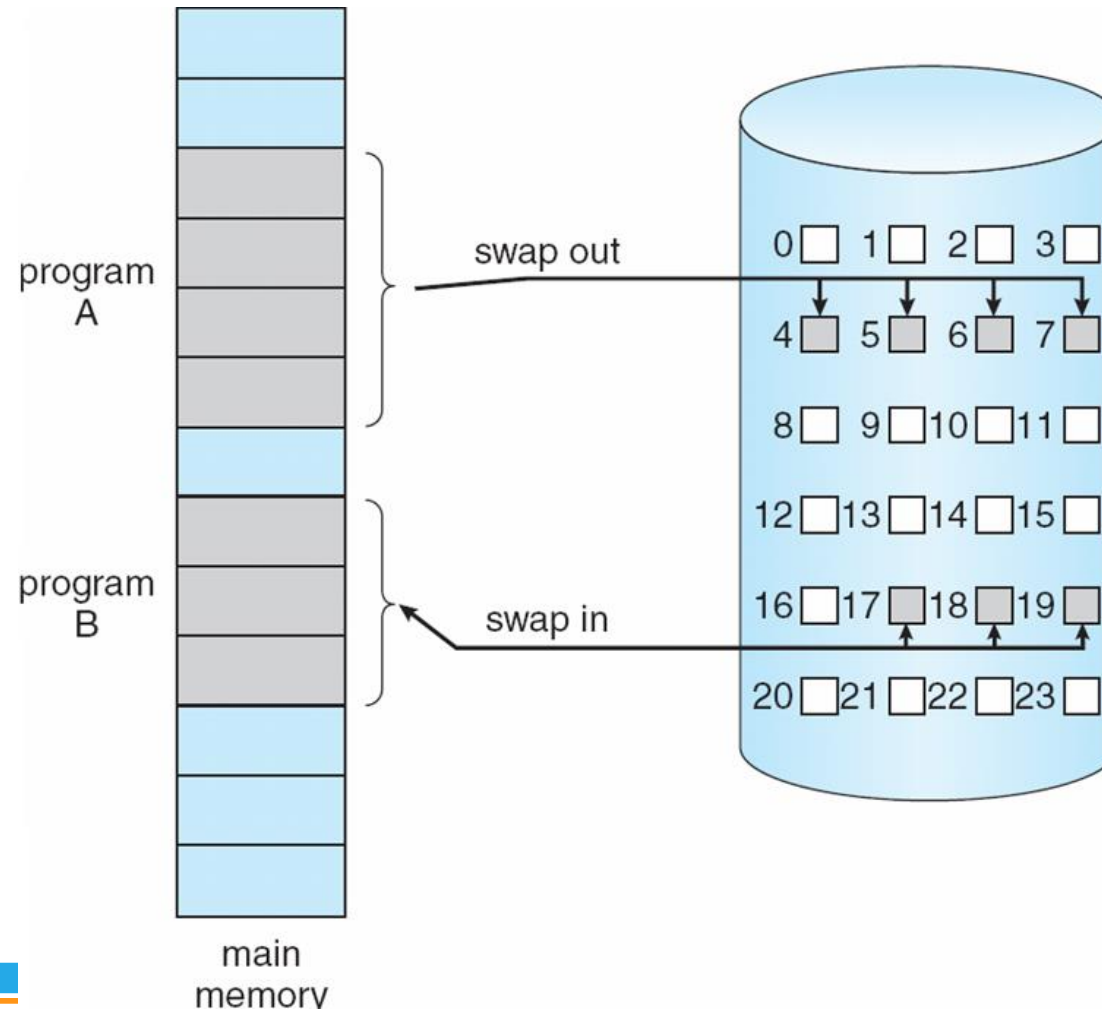
Shared pages example



Demand Paging

- 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**
 - If the requested page is not in memory, kernel brings it from hard disk by making a page fault.

Transfer of a Paged Memory to Contiguous Disk Space

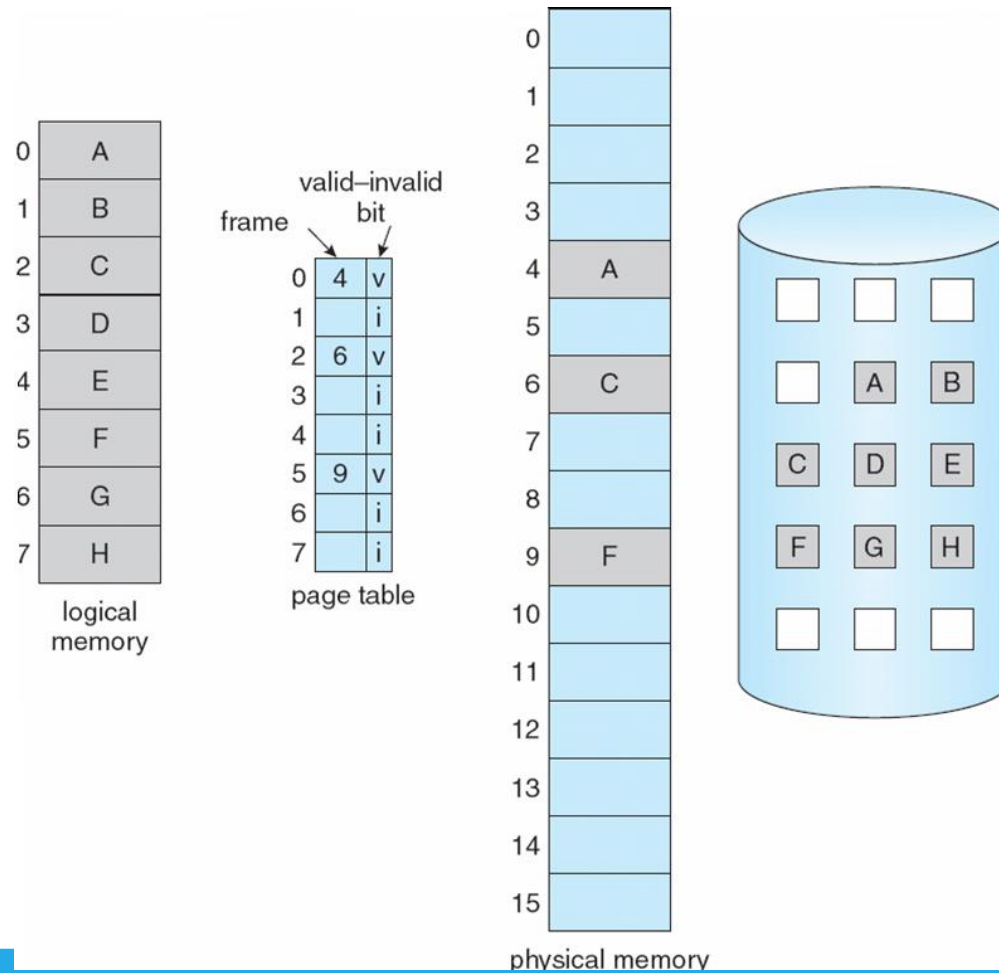


- Each page table entry has a valid–invalid bit:
 - v** $\Rightarrow$ in-memory,
 - i** $\Rightarrow$ not-in-memory
 - Initially, it is set to **i** for all entries
- During address translation, if valid–invalid bit is **i**, then it could be:
 - Illegal reference (outside process' address space) $\Rightarrow$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

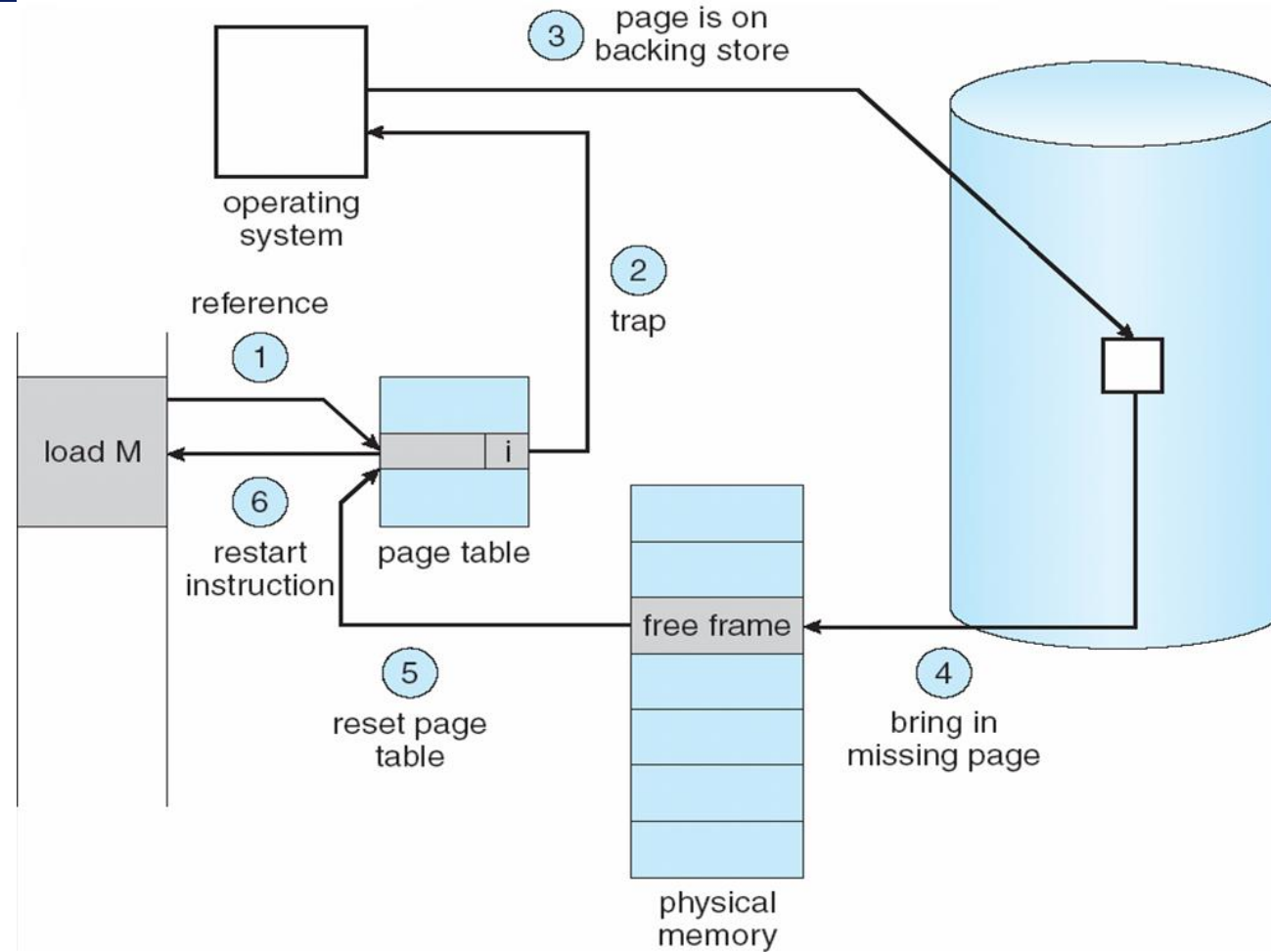
Page Table When Some Pages Are Not in Main



Page Fault

- 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

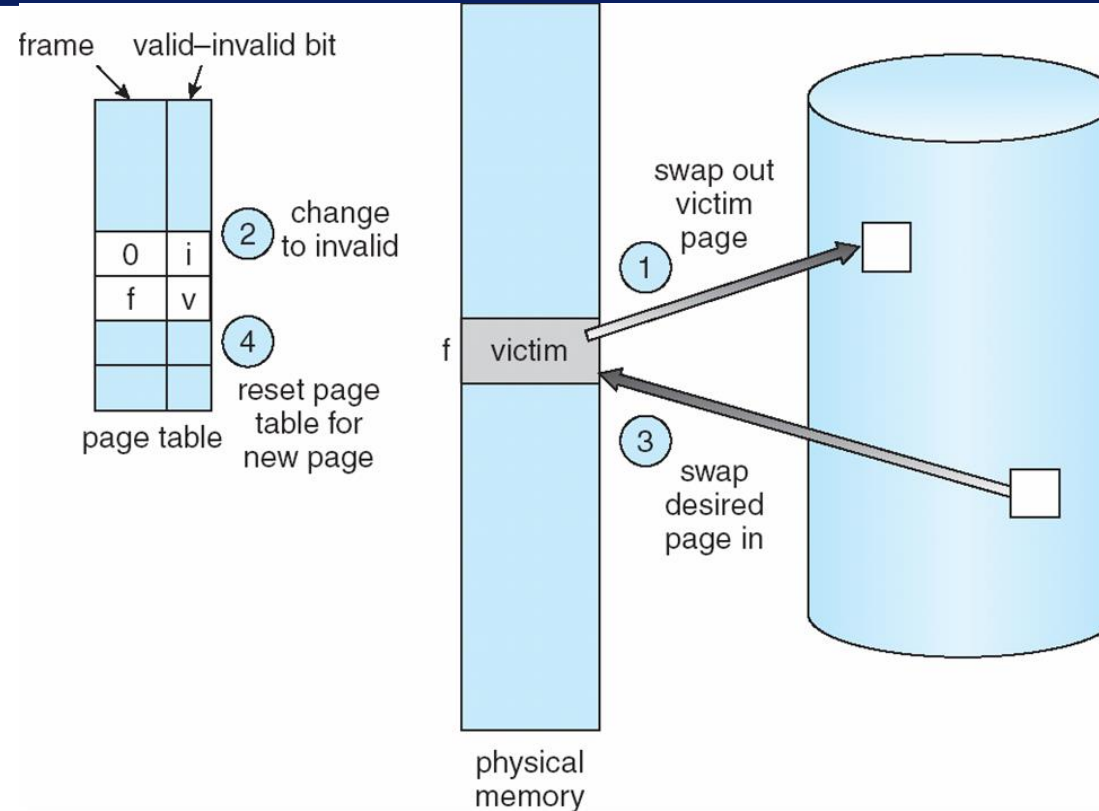
Steps in Handling a Page Fault



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

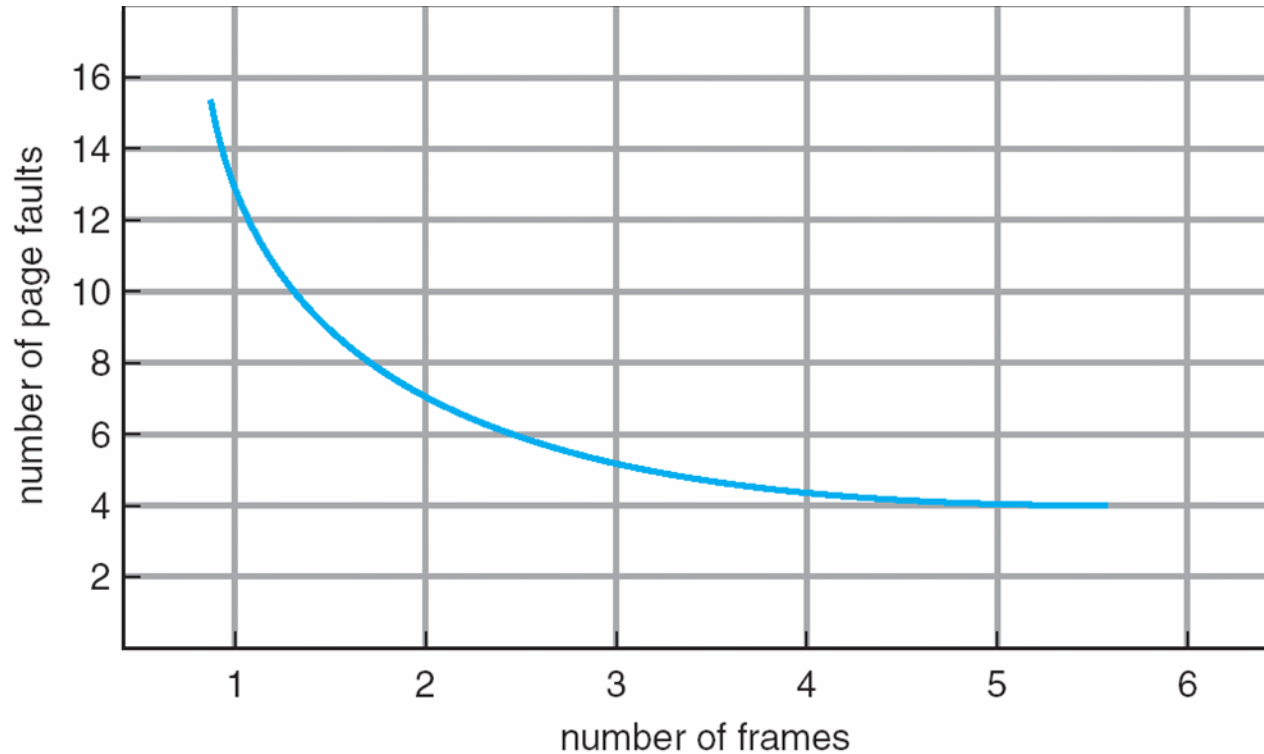


- ❑ **How do we choose the victim page? What would be the goal of this selection algorithm?**

- 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

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.

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

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																
	0	0	0																
		1	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

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		0		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



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

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

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



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

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

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



End of Module 5- Part2

