Amirkabir University of Technology
(Tehran Polytechnic)
Department of Computer Engineering

Main Memory Management

Hamid R. Zarandi
h_zarandi@aut.ac.ir

Why memory management?

- CPU utilization and increase of performance
- → need of scheduler to schedule multiple processes
- → they should be resident in RAM (physical memory)

- Why need of memory management?
  - Having multiple processes
  - Protection (processes from one another, processes from OS)

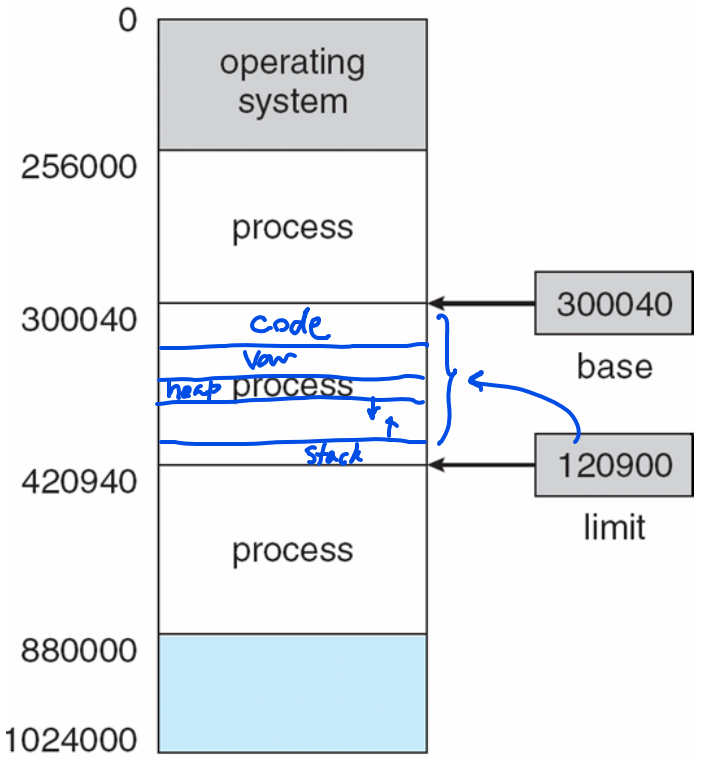
- Protection should be done by Hardware than OS (why?) ⇒
  - Performance penalty and latency

ins. fetch
result writeback
← توی حافظه مربوط به برنامه
خود OS یه برنامس!

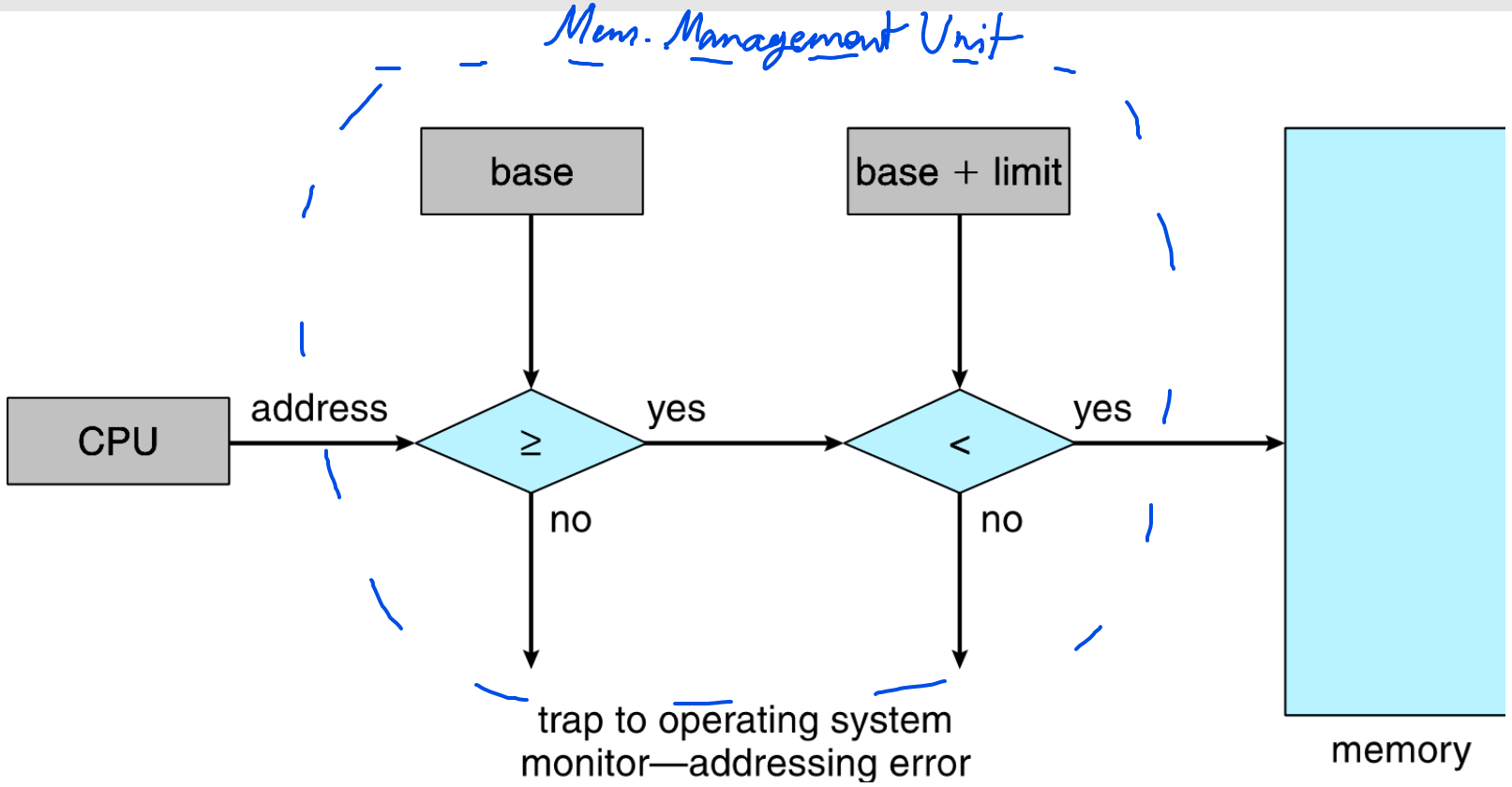
How to protect process memory space?

- Each process has a separate memory space

- To protect processes' spaces
  - Determining legal address
    - Base register (پایه): smallest legal physical memory address
    - Limit register (حد): size of the range
  - Example:
    - Base register = 300040
    - Limit register = 120900
    - Legal address space:
      Base <= (any address) < Base+Limit
    - Can easily be checked by hardware



HW address protection (base & limit regs.)



Any illegal address generates a trap exception which is known as fatal error

Who loads Base and Limit registers? → OS

Address binding

Address binding

physical → MM تری
logical → (CPU) ثبت‌های فرآیند → input queue (∈ Ready queue)

- **Input Queue**
  - Processes on disk that are waiting to be brought into memory during execution (Part of ready queue which is on disk)

- **How to put a process in a physical address?**
  - Addresses in source program are symbolic
    - Example: *count* variable
  - A compiler binds them to relocatable addresses
    - Example: 14 bytes from beginning of this module
  - Linker and loader bind them next to absolute addresses
    - Example: 74014

Source
int x
char y
compile
x_int
y_char
Linker / loader
0X...
0X...

How to bind inst./data to mem. address?

➢Compile time

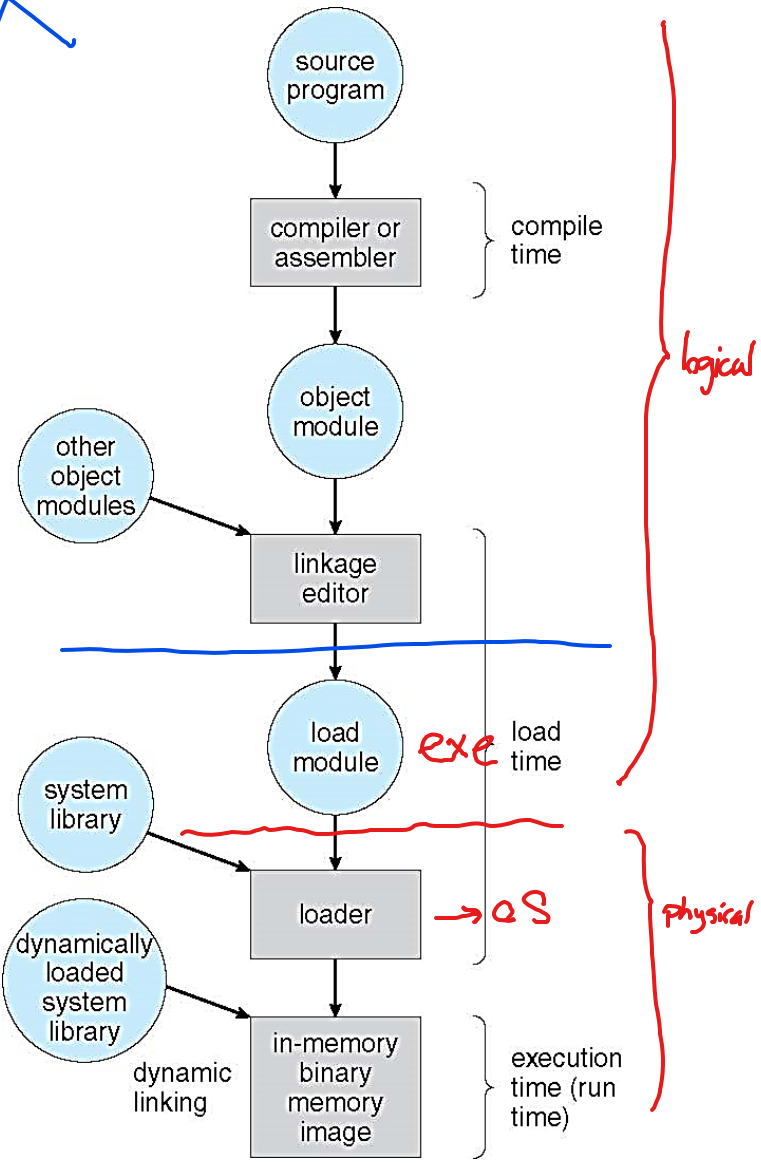
- o If memory location known a priori, absolute code can be generated; must recompile code if starting location changes
  - Example: COM files in MS-DOS

➢Load time

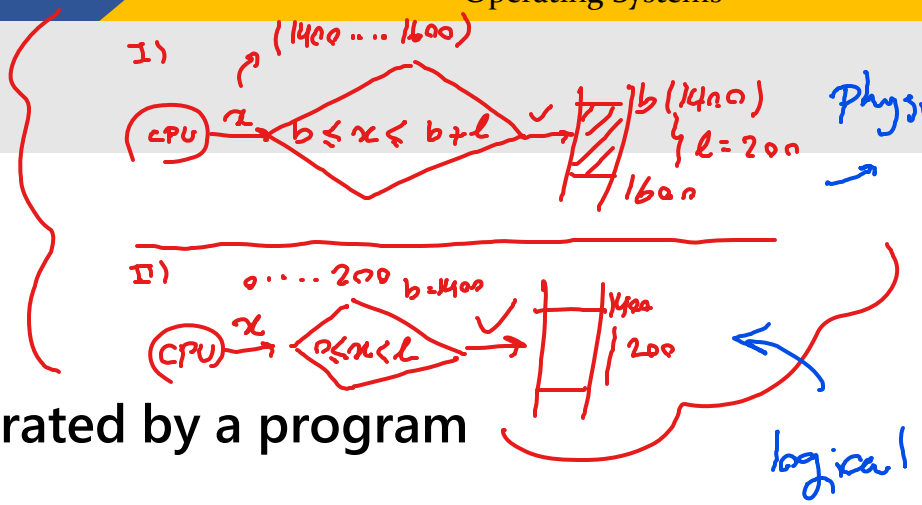
- o Must generate relocatable code if memory location is not known at compile time

➢Execution time

- o Binding delayed until run time if the process can be moved during its execution from one memory segment to another
  - Need hardware support for address maps
    - ✓ Example: Base-Limit registers



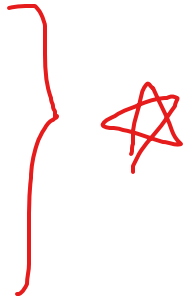
Logical vs. physical address space



- **Logical address (CPU address)**
  - CPU logically sees addresses
  - Logical address space: set of all logical addresses generated by a program

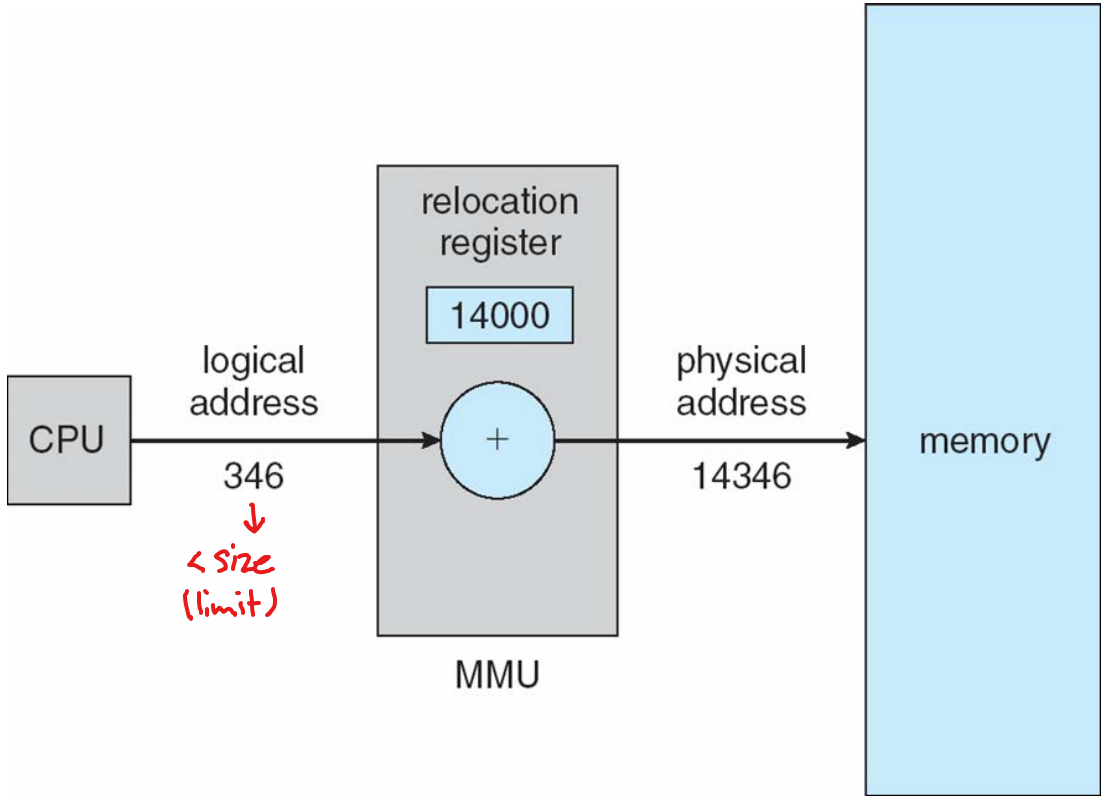
- **Physical address (Memory address)**
  - Address of memory line
  - Physical address space: set of all physical addresses generated by a program

- **May be equal or not**
  - Equal: compile-time and load-time address bindings
  - Not equal: execution-time address binding
    - In this case, logical address is said virtual address



What is MMU (Memory Management Unit)?

- MMU is responsible of mapping virtual (logical) address to physical address
- Simple version: Base-Limit registers
- Here Base register is called relocation register



Dynamic linking and loading

Dynamic loading

➢ Dynamic loading

- A routine is not loaded until it is called
- All routines except *main()* are kept on disk
- Better space utilization
- Some routines are infrequently needed: *error functions*
- In the case of call, if it is not loaded, linking loader first loads it to the memory and update program's address table to reflect this change

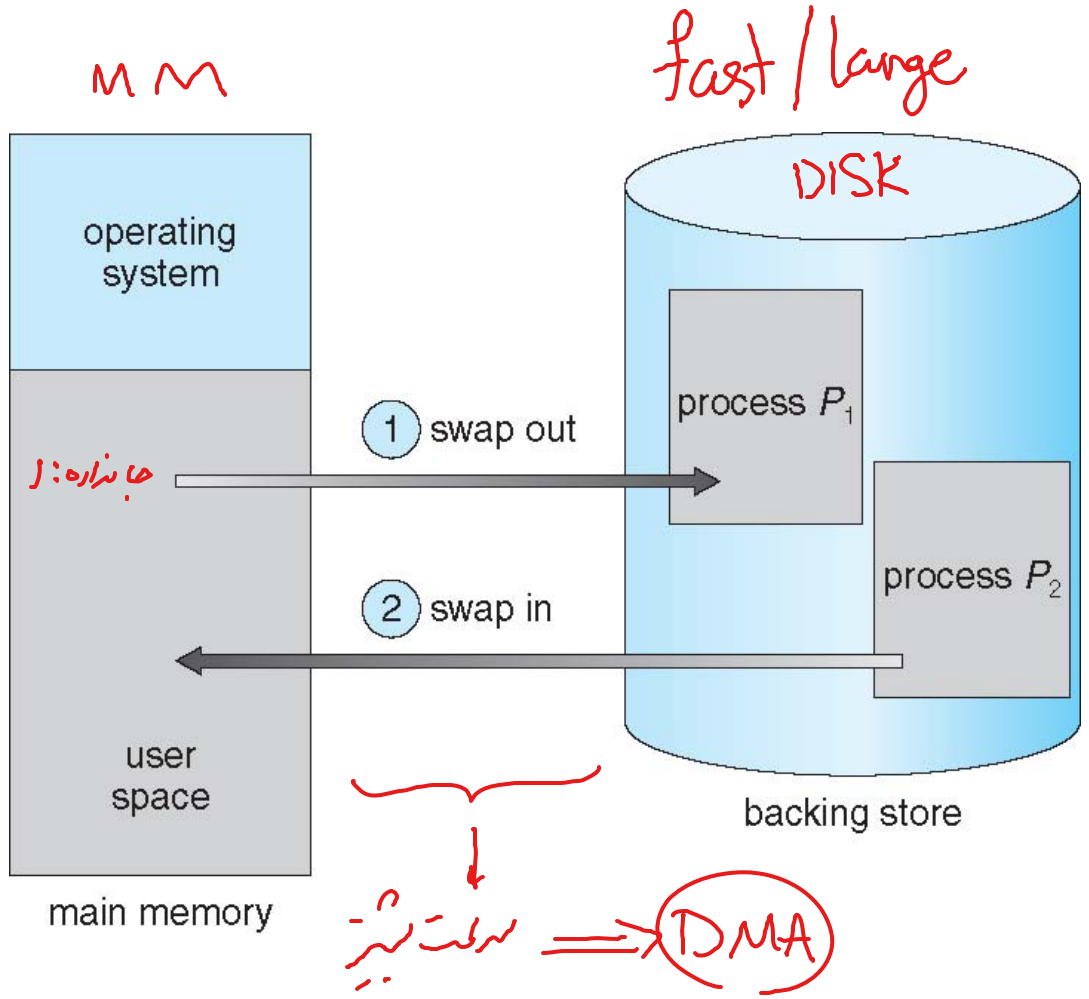
Dynamic linking

- **Dynamic linking (Dynamically linked libraries)**
  - System libraries that are linked to user programs when they are run
  - No waste of memory and disk space
  - Example: Language subroutines
  - Stub: small piece of code used to locate dynamic linked libraries
    - Replace itself with the address of the routine & executes the routine
  - Also known as Shared Libraries
- **Static linking**
  - System libraries are linked to user programs during compile time

Swapping

Swapping

- **Swapping** is performed temporarily between memory and backing store
- **Backing store** – fast disk large enough to accommodate copies of all memory images for all users; must provide direct access to these memory images
- Possible of total process memory spaces exceeds real physical memory
  - → Use of Backing store



Swapping issues

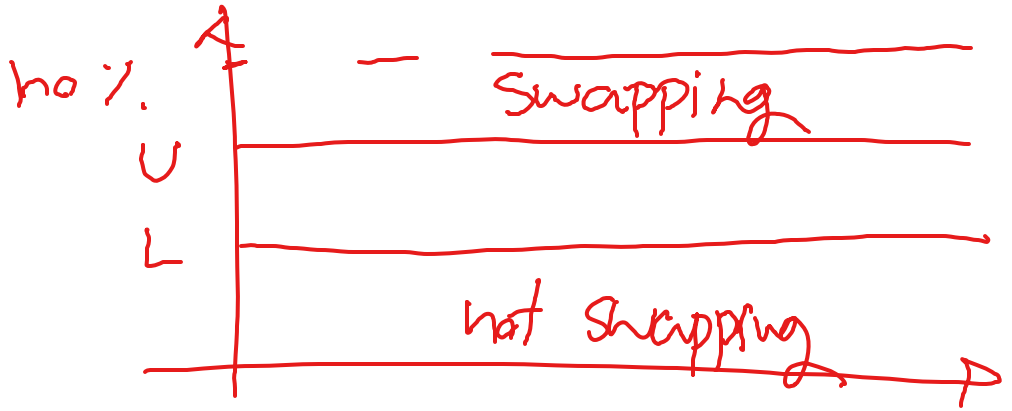
- **Does the swapped out process need to swap back in to same physical addresses?**
  - Depends on address binding method
  - Plus consider pending I/O to/from process memory space

- **Pending IO: cannot swap out as IO would occur to wrong swapped in process!**
  - Or always transfer IO to kernel space, then to IO device; this is known as double buffering; adds overheads

I/O باید برگرده سرجاش اگه میره بیرون (ج)

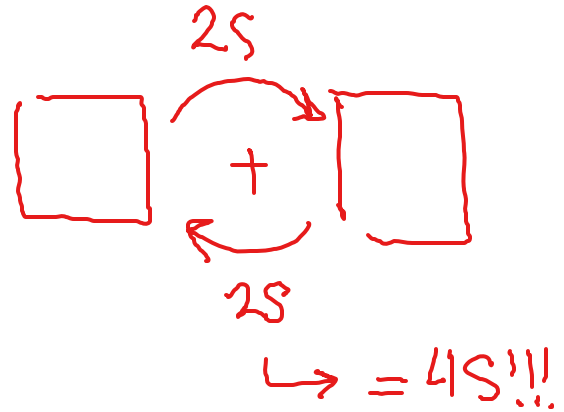
Swapping and modern OSs

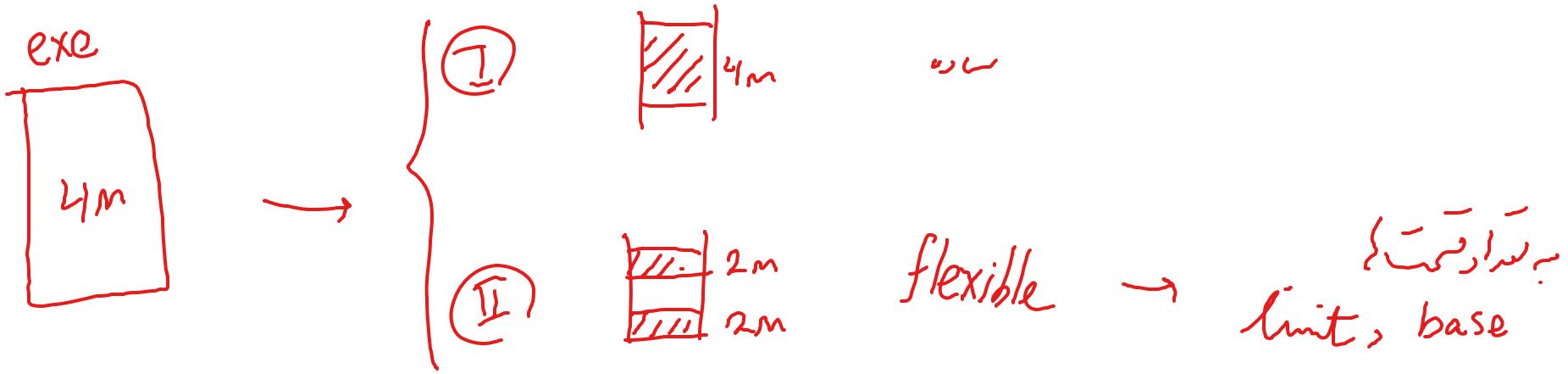
- **Standard swapping is not used in Linux & Windows!**
  - **Modified version is used:**
    - **Swap only when free memory is extremely low (less than threshold)**
    - **Disable swapping when free memory is more than threshold**



Swapping cost

- Major part of swap time is transfer time
- Total transfer time is proportional to the amount of memory swapped.
- Example:
  - 100 MB process swapping to hard disk with transfer rate of 50 MB/sec
  - Swap out time of 2(sec)+swap in of same size process
  - Total context switch swapping component time of 4 sec.





Memory allocation

Memory allocation

- **Different types**
  1. **Contiguous**
     - Each process is in a single section of memory that is contiguous to sections of others
  2. **Segmentation**
     - Each process is divided into different segments; each one is located in different part
  3. **Paging**
     - Each process is divided into same–small–size pages; some of them are swapped in/out

Criteria and problems

CPU Utilization

- Utilization = Percent of time a CPU is busy $= \frac{CPU\ time\ used}{Total\ time\ system\ is\ busy}$
  - Some times is $\frac{CPU\ time\ used}{CPU\ time\ used + 2 * swapping\ time}$
  - Some times is $\frac{CPU\ time\ used}{\max(CPU\ time\ used,\ 2 * swapping\ time)}$

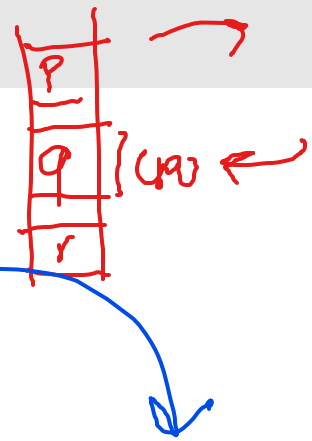
Fragmentation

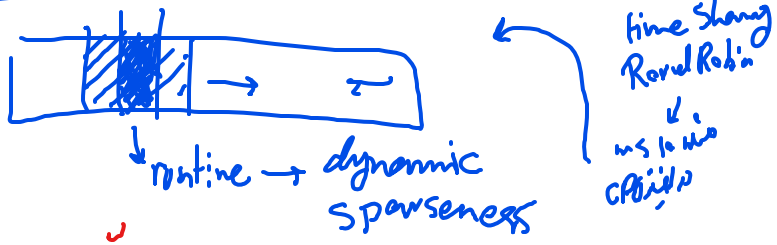
- External
- Internal

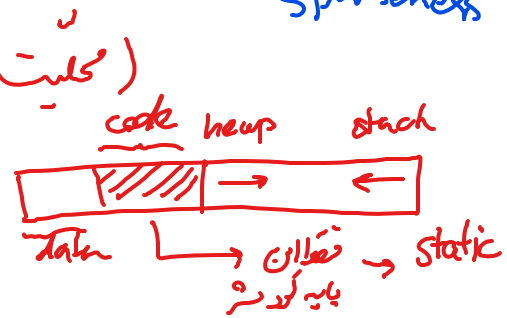
Process locality problem

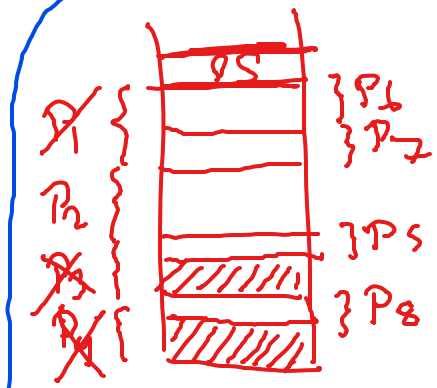
- Static sparseness
- Dynamic sparseness

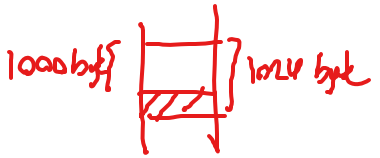
Code/data sharing problem and protection problem





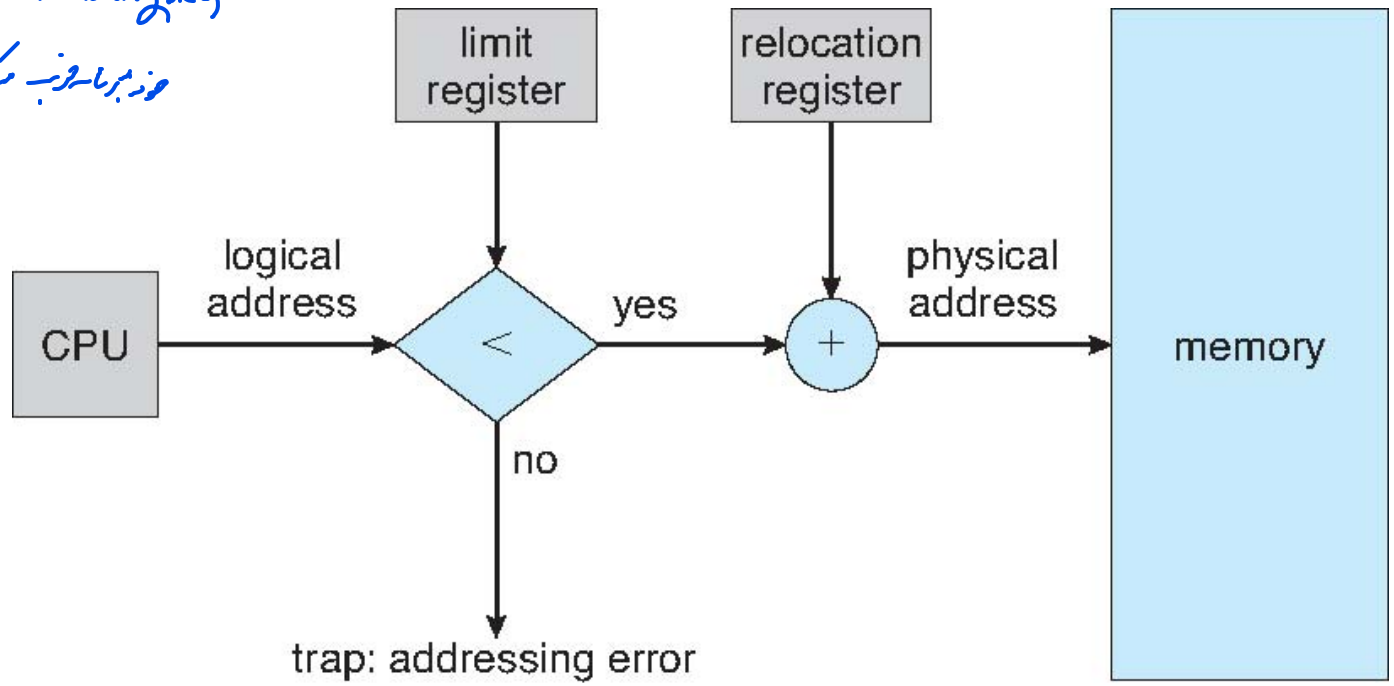






1) Contiguous allocation

- **Relocation registers** used to protect user processes from each other, and from changing operating-system code and data
  - **Base register** contains value of **smallest physical address**
  - **Limit register** contains **range of logical addresses**
    - Each logical address must be less than the limit register

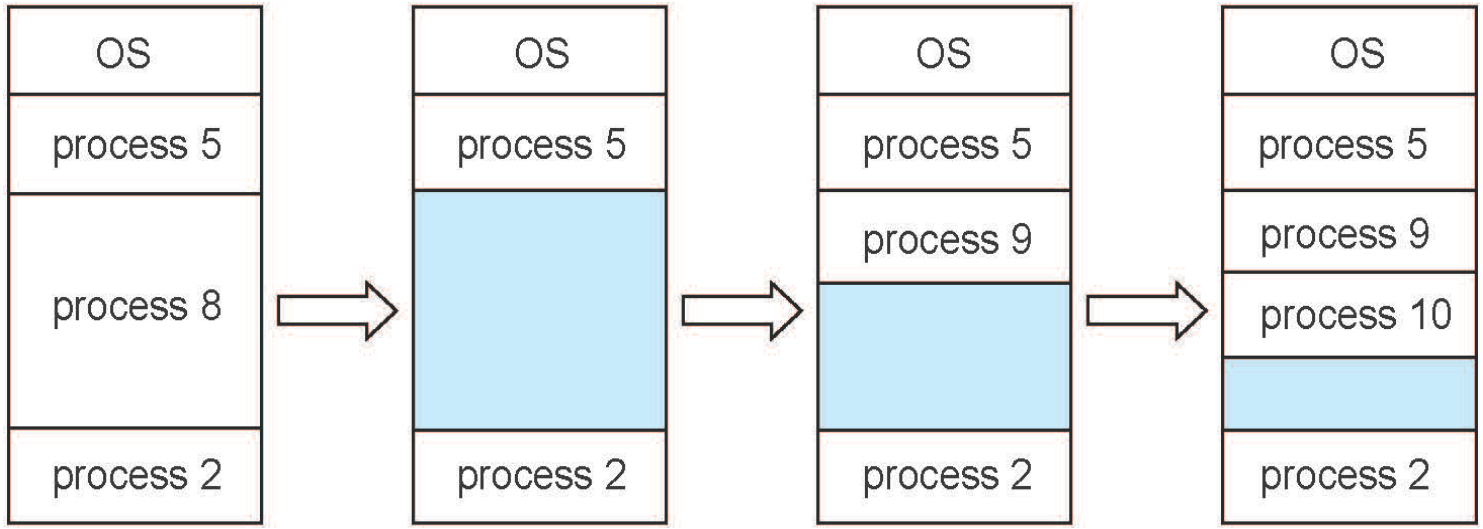


Multiple-partition memory allocation

- Degree of multiprogramming limited by number of partitions
- Variable-partition sizes for efficiency (sized to a given process' needs)

- Hole – block of available memory; holes of various size are scattered throughout memory
- When a process arrives, it is allocated memory from a hole large enough to accommodate it

- Operating system maintains information about:
  a) allocated partitions b) free partitions (hole)



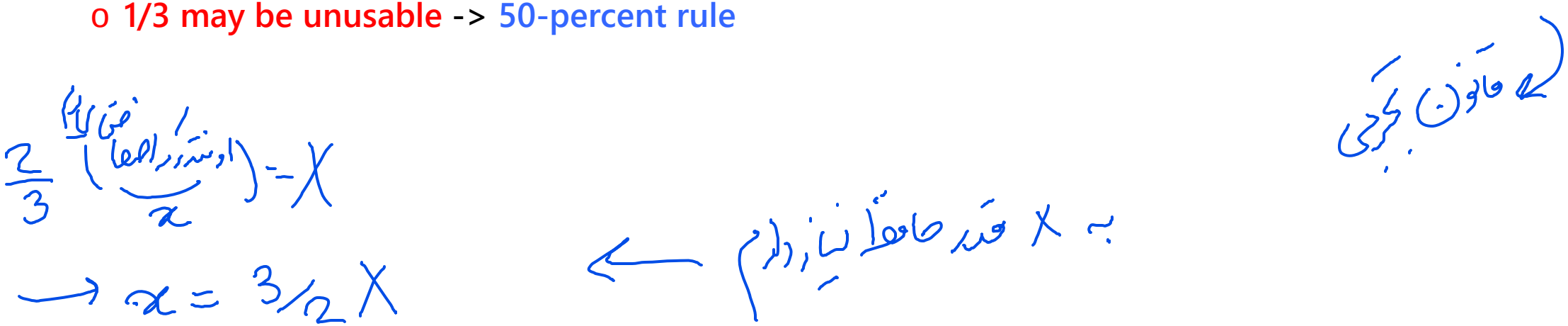
Dynamic storage-allocation problem

- How to satisfy a request of size ***n*** from a list of free holes?
  - **First-fit**: Allocate the ***first*** hole that is big enough
  - **Best-fit**: Allocate the ***smallest*** hole that is big enough; must search entire list, unless ordered by size
    - Produces the smallest leftover hole
  - **Worst-fit**: Allocate the ***largest*** hole; must also search entire list
    - Produces the largest leftover hole

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

Fragmentation problem

- **External Fragmentation** ✓
  - Total memory space exists to satisfy a request, but it is not contiguous
- **Internal Fragmentation** ✗
  - Not whole of allocated memory is filled by the process memory
- **First fit** analysis reveals that given *N* blocks allocated, **0.5** *N* blocks lost to fragmentation
  - **1/3 may be unusable** -> **50-percent rule**

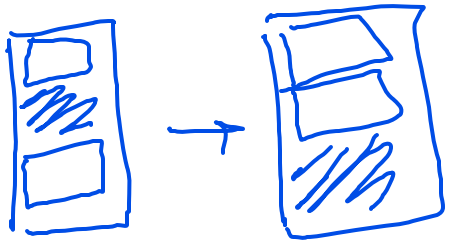


Fragmentation solution

➢Compaction

- Shuffle memory contents to place all free memory together in one large block
- Compaction is possible *only* if relocation is dynamic, and is done at execution time
- I/O problem
  - Latch job in memory while it is involved in I/O
  - Do I/O only into OS buffers

- Now consider that backing store has same fragmentation problems

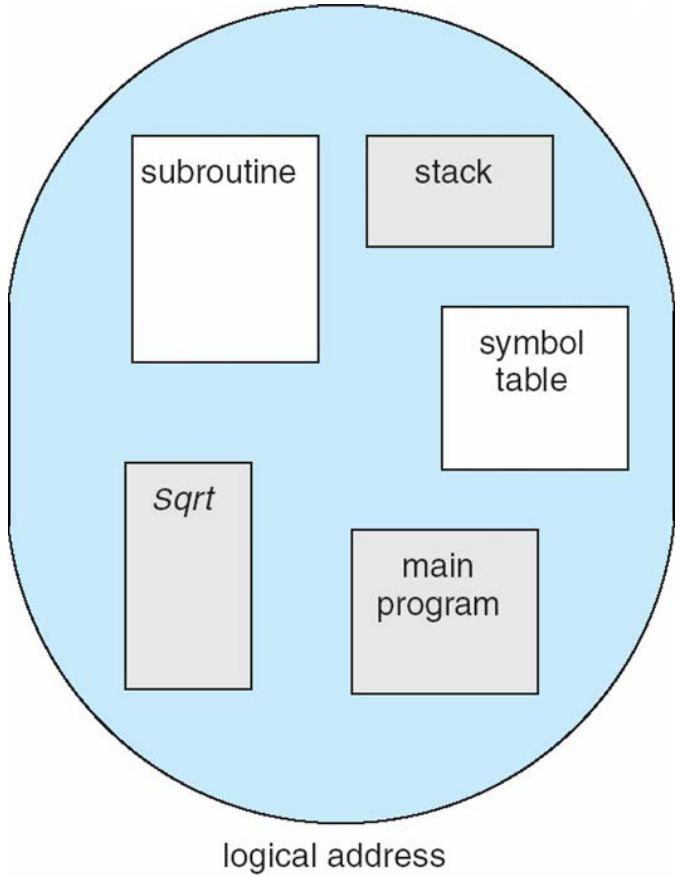


2) Segmentation

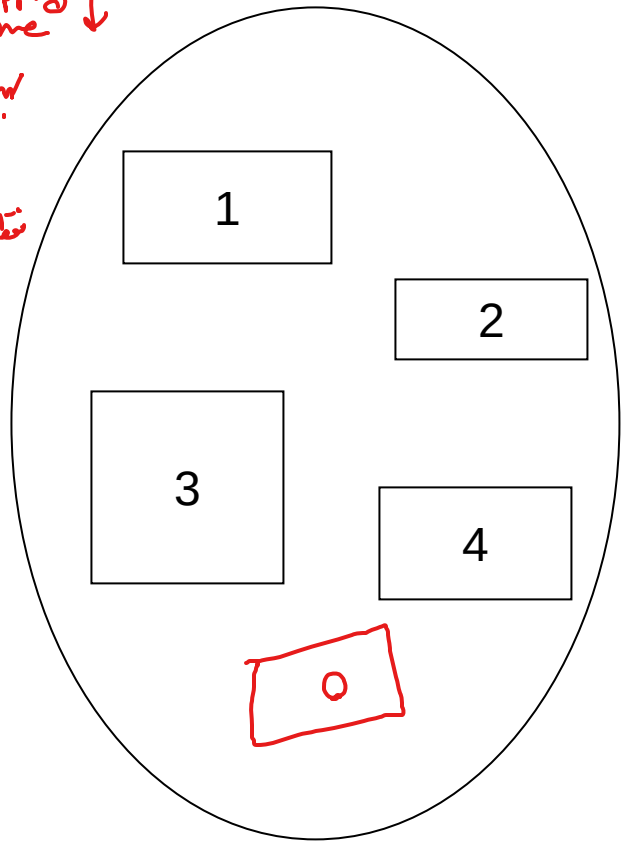
بررسی ـــ صفحه بعد

- Memory-management scheme that supports user view of memory

- A program is a collection of 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



logical view of segmentation



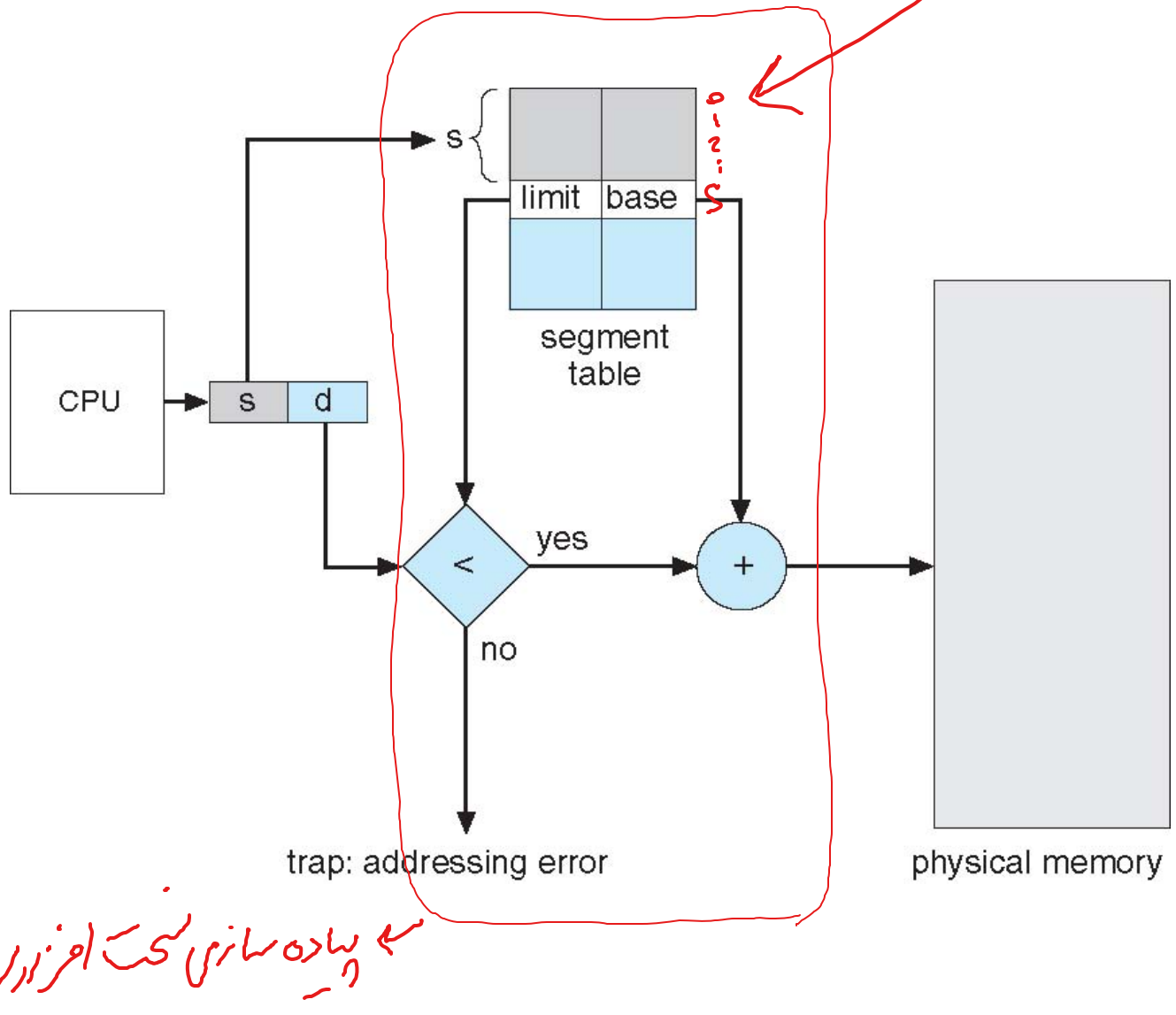
user space

physical memory space

Segmentation implementation

- **Logical address** consists of a two tuple: **<segment-number, offset>**

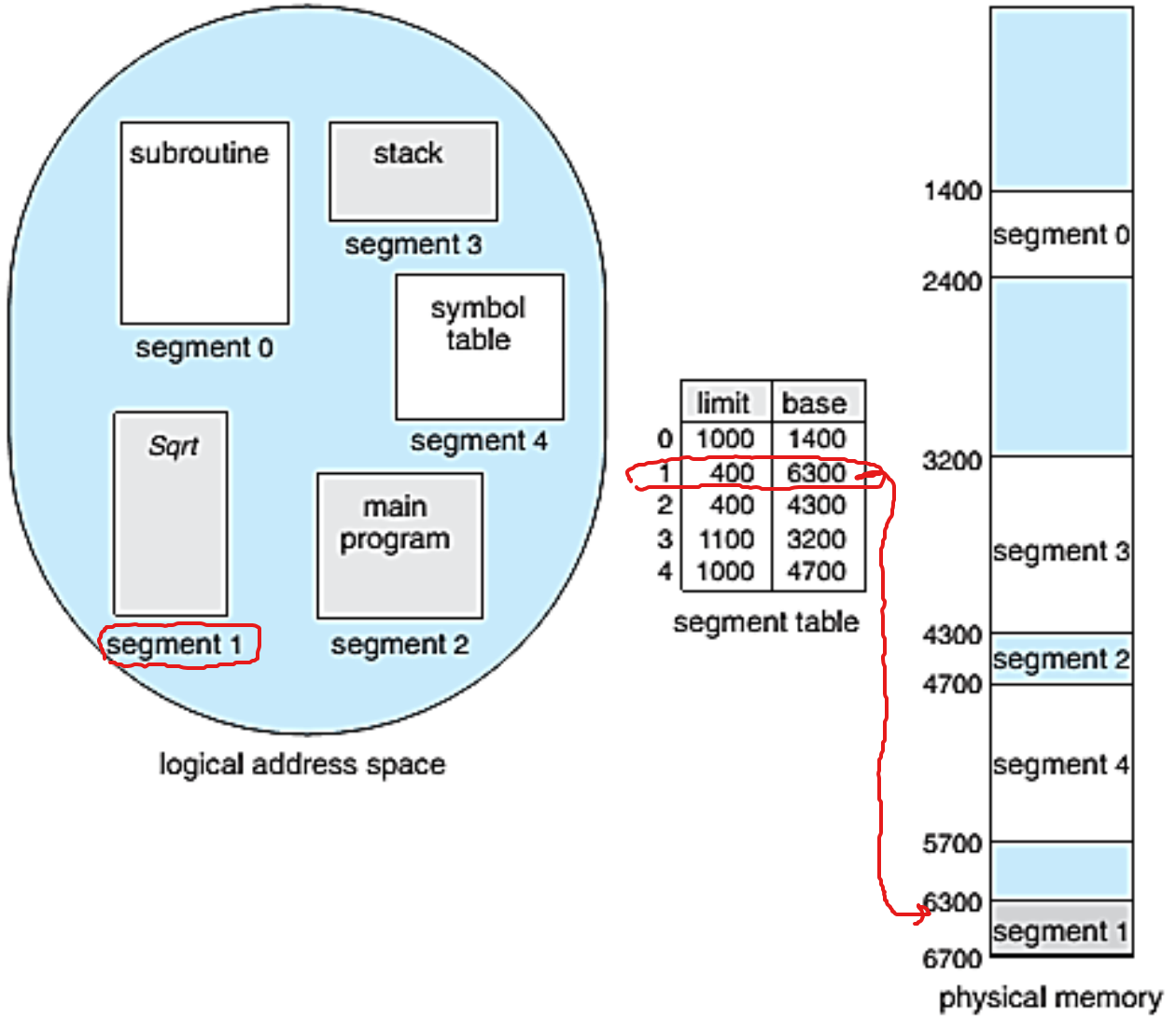
- **Segment table**
  - Maps two-dimensional physical addresses; each table entry has:
    - **base** – contains the starting physical address where the segments reside in memory
    - **limit** – specifies the length of the segment



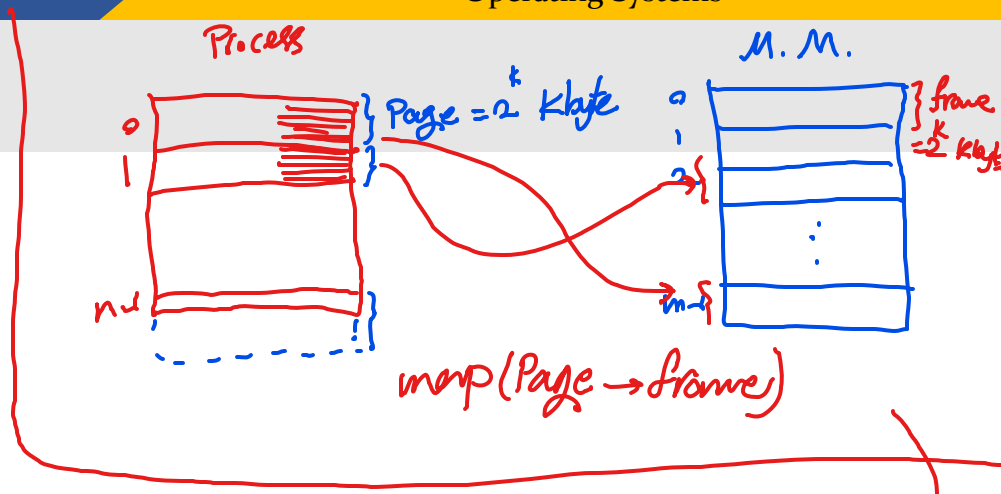
پیاده سازی سخت افزاری

Example of segmentation

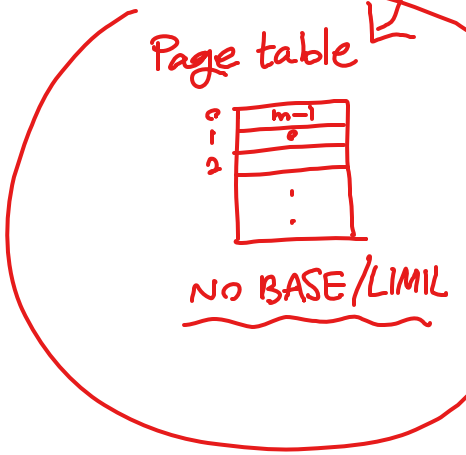
- A reference to byte 53 of segment 2:
  4300+53=4353
- A reference to byte 852 of segment 3:
  3200+852=4052
- A reference to byte 1222 of segment 0:
  Trap to OS!



3) Paging

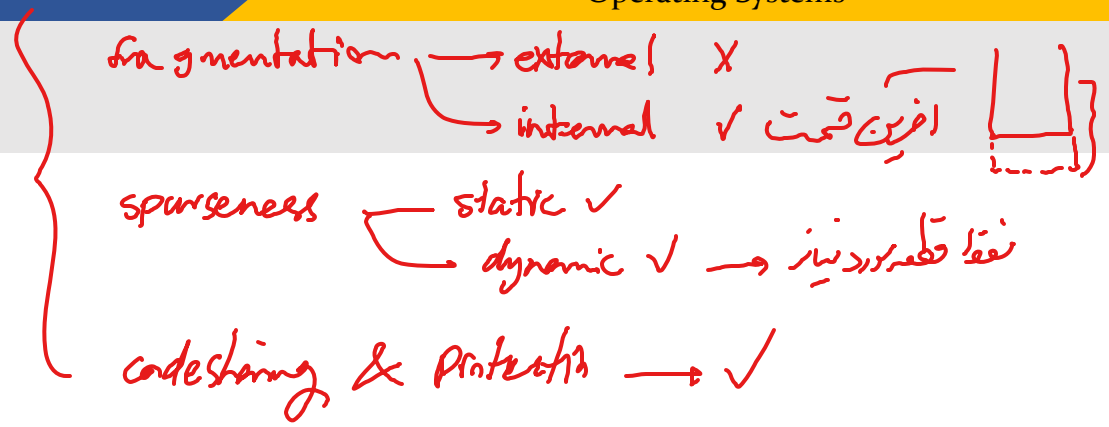


- Noncontiguous memory allocations:
  - Segmentation
  - Paging
- Paging avoids external fragmentation, and need of compaction, whereas segmentation does not.
- Process is allocated physical memory whenever physical memory is available
  - Avoid external fragmentation
  - Avoid problem of varying sized memory chunks
- Divide physical memory into fixed-sized blocks called frames.
  - Size is power of 2, between 512 bytes to 16 Mbytes
- Divide logical memory into blocks of same size called pages.



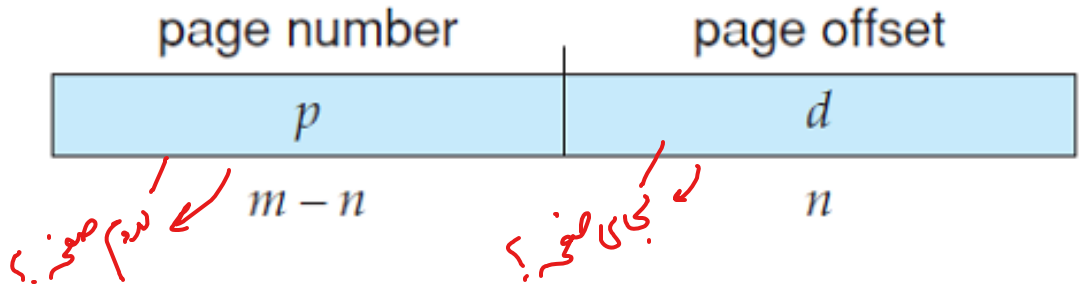
Paging

- Keep tracks of all free frames
- To run a program with N pages, need to find N different free frames and load program.
- Setup a page table to translate logical to physical address
- Still have internal fragmentation



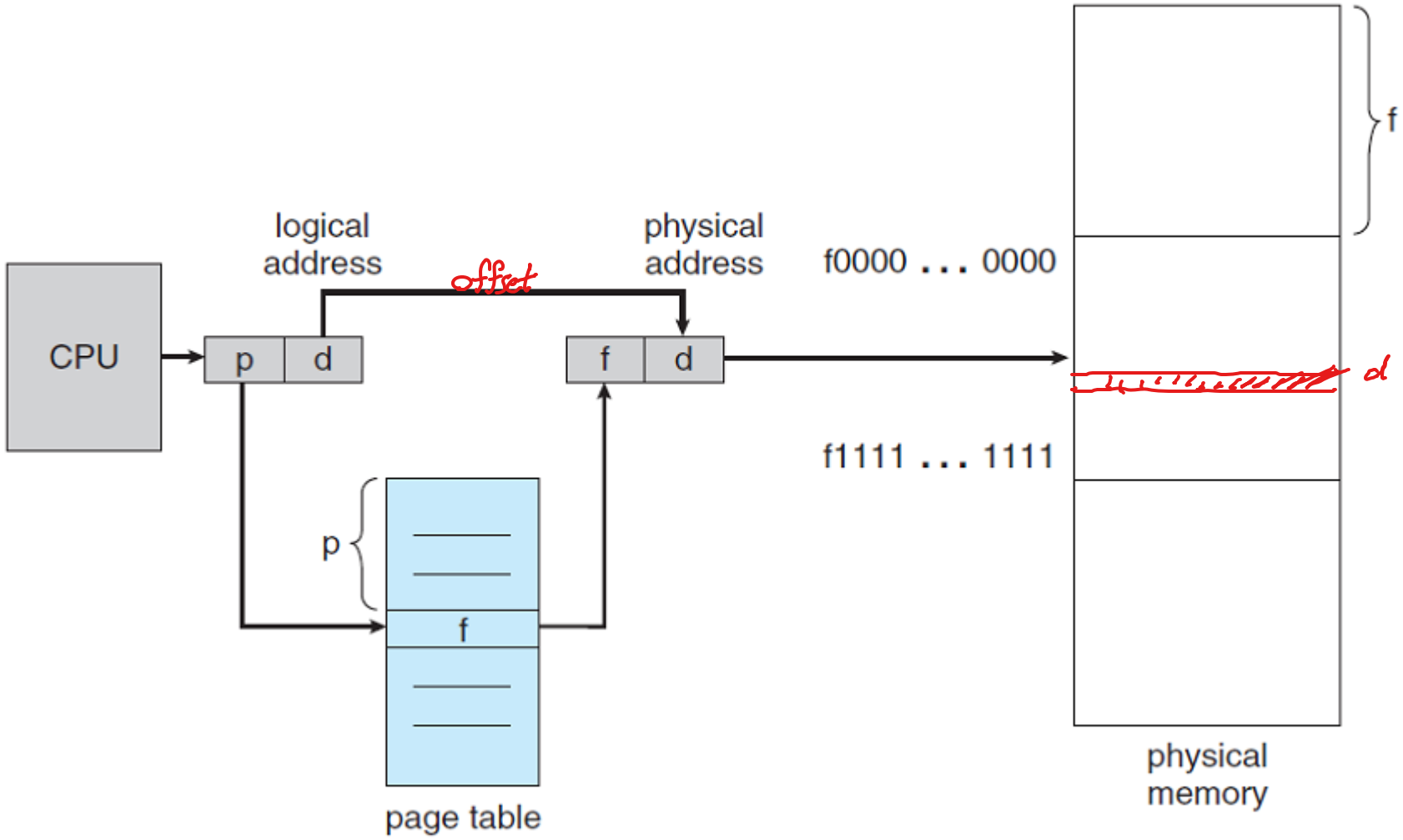
Address translation scheme

- CPU address (Logical address) is divided into two parts:
  - Page number (p): used as an index into a page table that contains base register of each page in physical memory
  - Page offset (d): combined with base address to define the physical memory address that is sent to the memory unit.

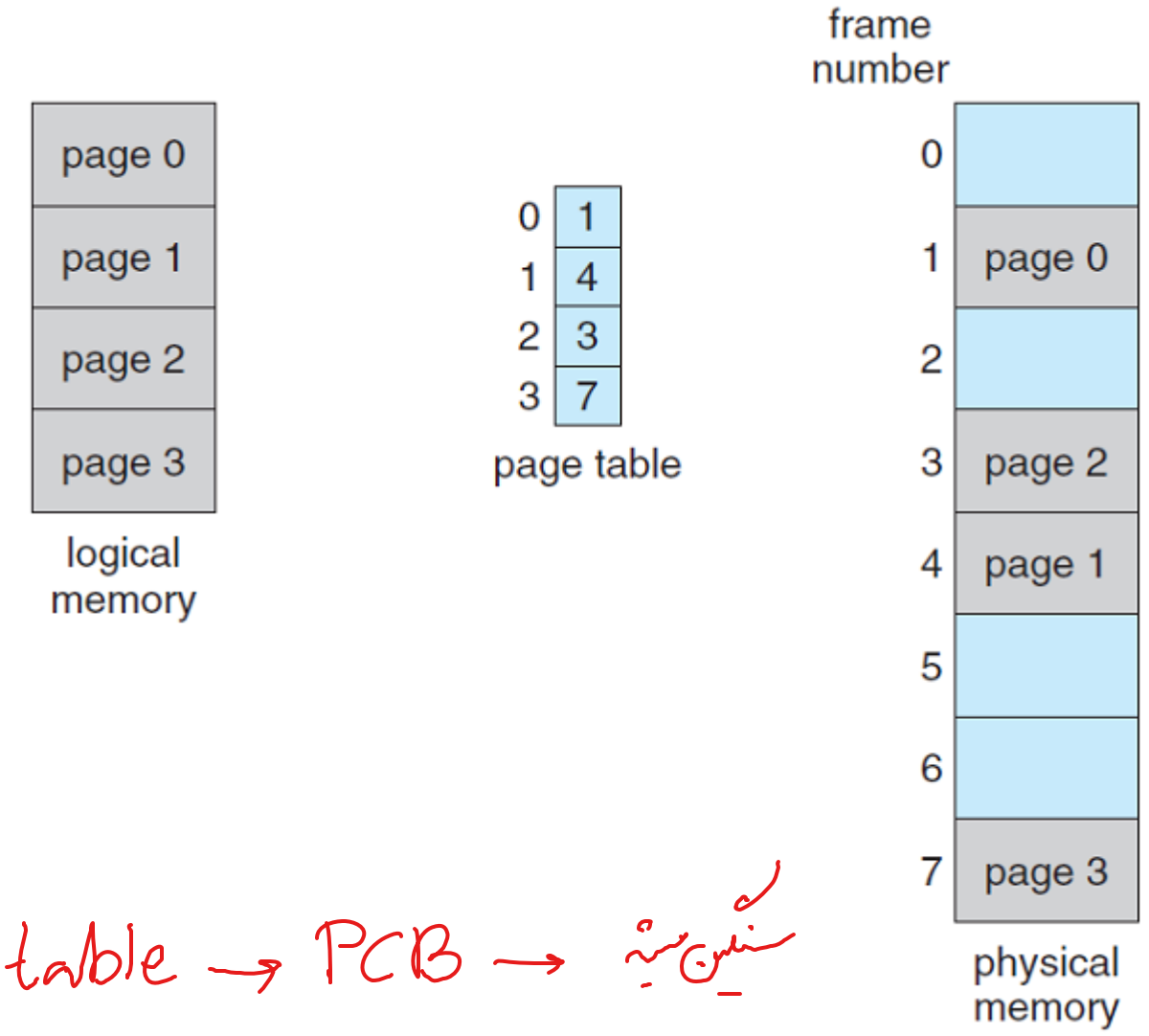


- For given logical address space 2^m and page size 2^n

Paging hardware

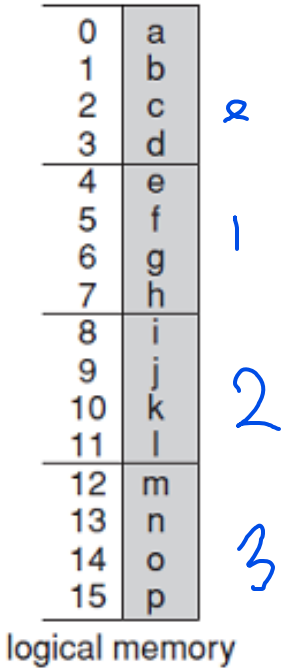


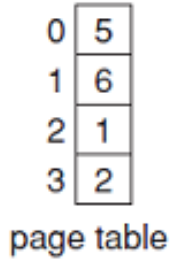
Paging model of logical and physical memory

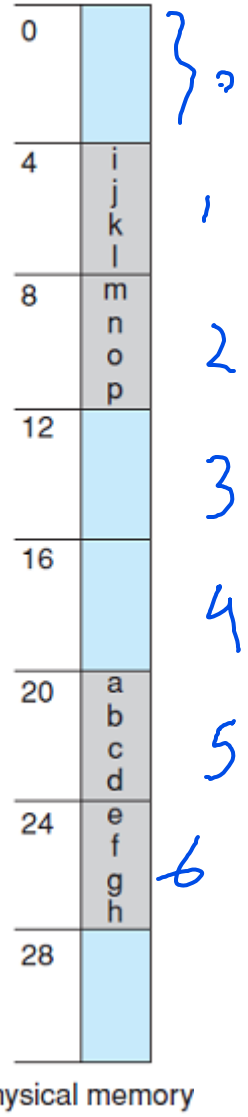


Page table → PCB → سیستم عامل

Paging example







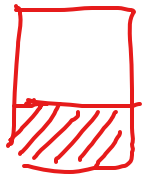
- n=2 and m=4, 32-byte memory and 4-byte pages.

Free frames


free-frame list
14
13
18
20
15
page 0
page 1
page 2
page 3
new process
13
14
15
16
17
18
19
20
21
before allocation
free-frame list
15
page 0
page 1
page 2
page 3
new process
0 14
1 13
2 18
3 20
new-process page table
13 page 1
14 page 0
15
16
17
18 page 2
19
20 page 3
21
after allocation

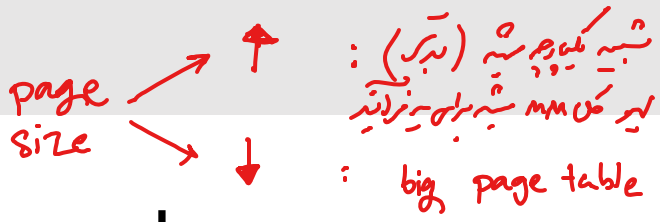
Paging example – Internal fragmentation

- **Page size** = 2048 bytes (2kB)
- **Process size** = 72766 bytes
- 72766/2048 = 35 pages +1086 bytes

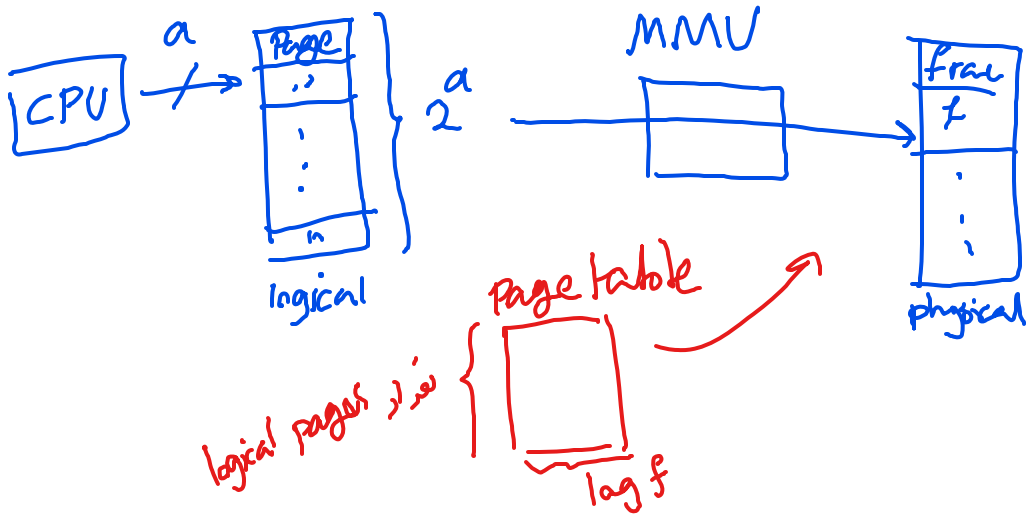


- **Internal fragmentation**: 2048 – 1086 = 962 bytes
- **Worst case** fragmentation: 1 frame – 1 byte
- **On average** fragmentation: ½ frame size

Small page size vs big page size



- On average fragmentation: ½ page size, small page size are good.
- Small page size, more overhead is in the page-table, this overhead is reduced when page size increases.
- Disk I/O is more efficient when the amount of data being transferred is larger (e.g. big pages)
- Page typically are between 4 kB and 8 kB in size.

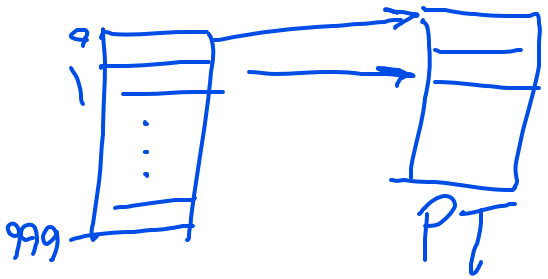


$P \downarrow \rightarrow$ Page table $\uparrow \Rightarrow \frac{\#\text{Pages} \times (\log_2 f) \times \frac{1}{8}}{\text{Page size}}$ byte

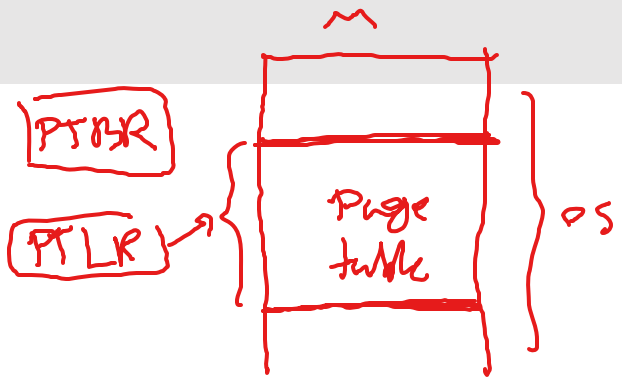
حجم جدول صفحه برای هر برنامه در حافظه اصلی PT کاهش.

Page table implementation

مثال: Page table Size = 1000



Page table



- Page table is kept in memory
- Page-table base register (PTBR) points to the page table
- Page-table length register (PTLR) indicates size of the page table
- In this scheme every data/instruction access requires two memory accesses:
  - One for the page table, another for data/instruction

Translation look-aside buffer (TLB)

- The two memory access problem can be solved by the use of a special fast-lookup hardware cache called associative memory or translation look-aside buffer (TLB).

caching ← تعمیم

- Associative memory: parallel search

Address Locality

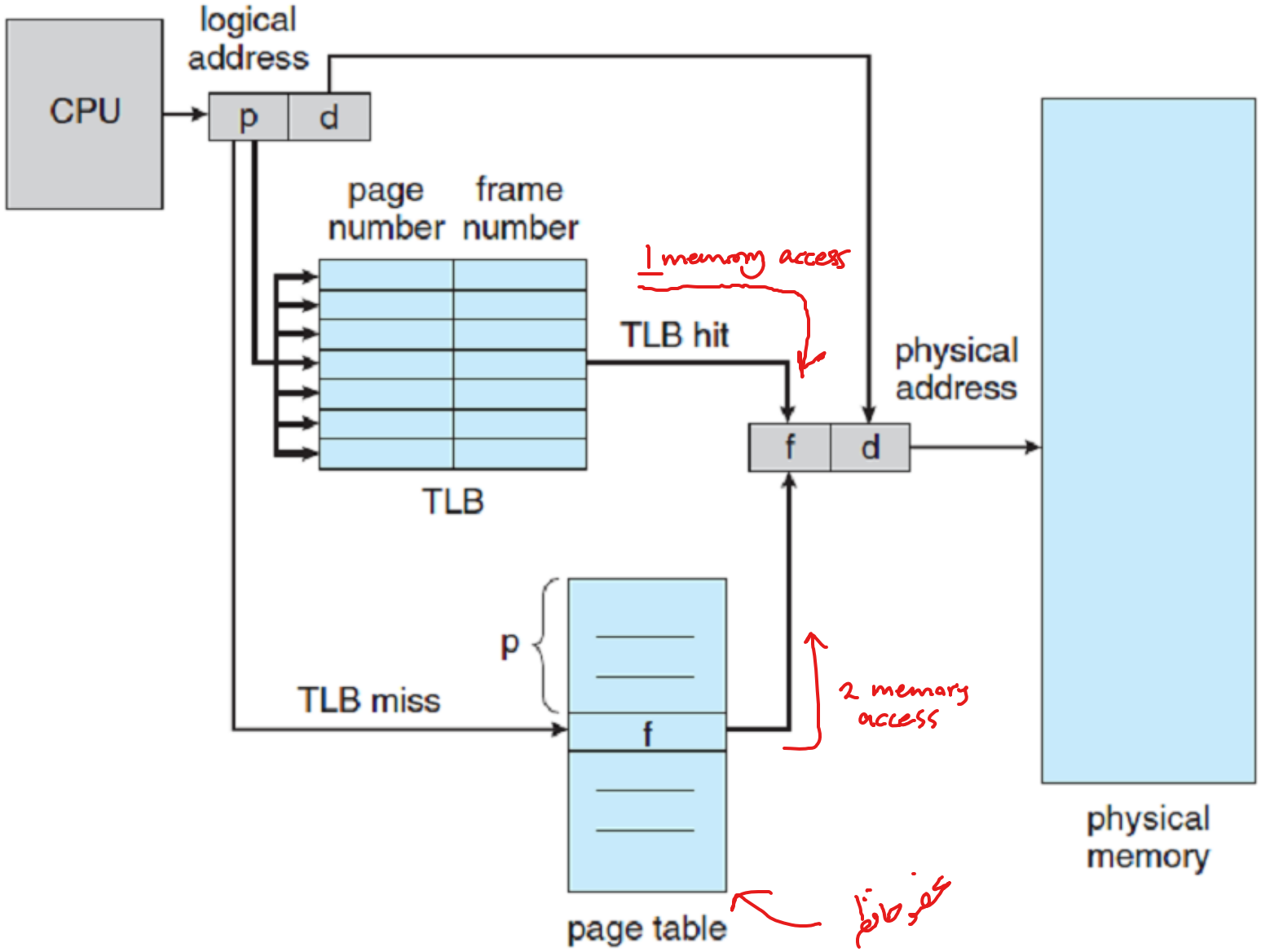
Page #	Frame #

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

frame ← اگر توی TLB بود

frame ← PT ← وگرنه

Paging hardware with TLB



Effective access time

➢**Hit ratio**: **percentage of times** that a page number is found in the **TLB**.

Effective Access Time (EAT)

- α: memory access latency
- h: hit ratio
- $EAT = h \times \alpha + (1 - h) \times 2\alpha$

a = memory access
b = TLB access

miss

$\hookrightarrow h(a+b) + (1-h)(b+2a)$

$h = 80\%, \alpha = 100ns \Rightarrow EAT = 0.80 \times 100 + 0.20 \times 200 = 120ns$

$h = 99\%, \alpha = 100ns \Rightarrow EAT = 0.99 \times 100 + 0.01 \times 200 = 101ns$

More about TLB

- Some TLBs store address-space identifier (ASID) in each TLB entry
  - Uniquely identifies each process to provide address-space protection for that process
  - Otherwise, need to flush at every context switch
- TLB is typically small (64 to 1024 entries)
- On a TLB miss, value is loaded into the TLB for faster access next time.
  - Replacement policies must be considered.

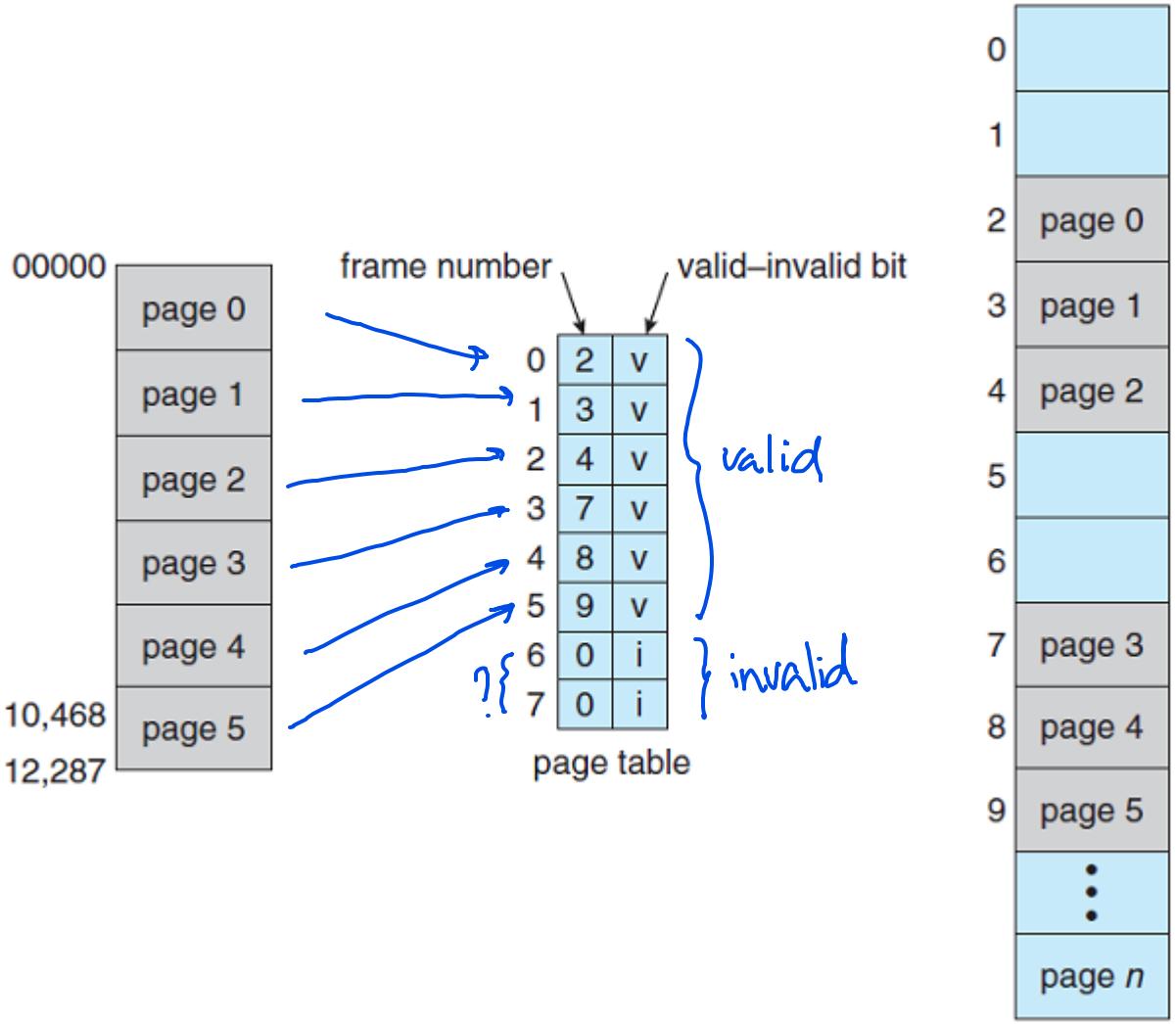
Memory protection

- Memory protection is implemented by protection bit with each frame to indicate if read-only or read-write access is allowed.

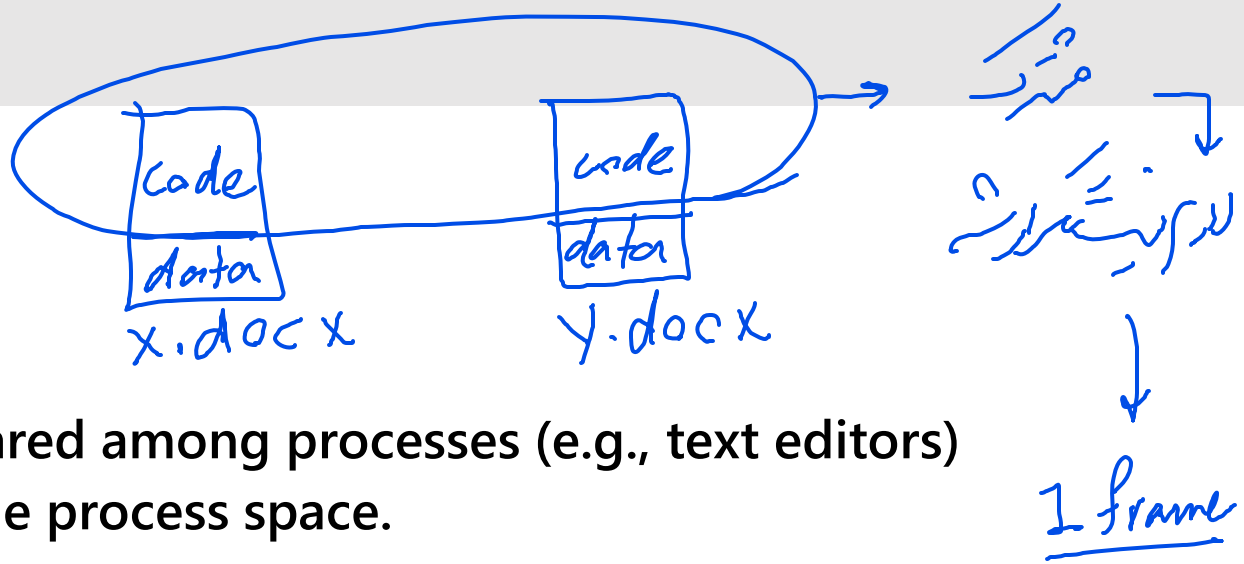
- Valid-invalid bit attached to each entry in page table:
  - Valid indicates that the page is in the process logical address space (legal page)
  - Invalid indicates that the page is not in the process logical address space (illegal page)
  - Or use page-table length register (PTLR)

- Any violation result in trap to the kernel

Valid/invalid bit in a page table



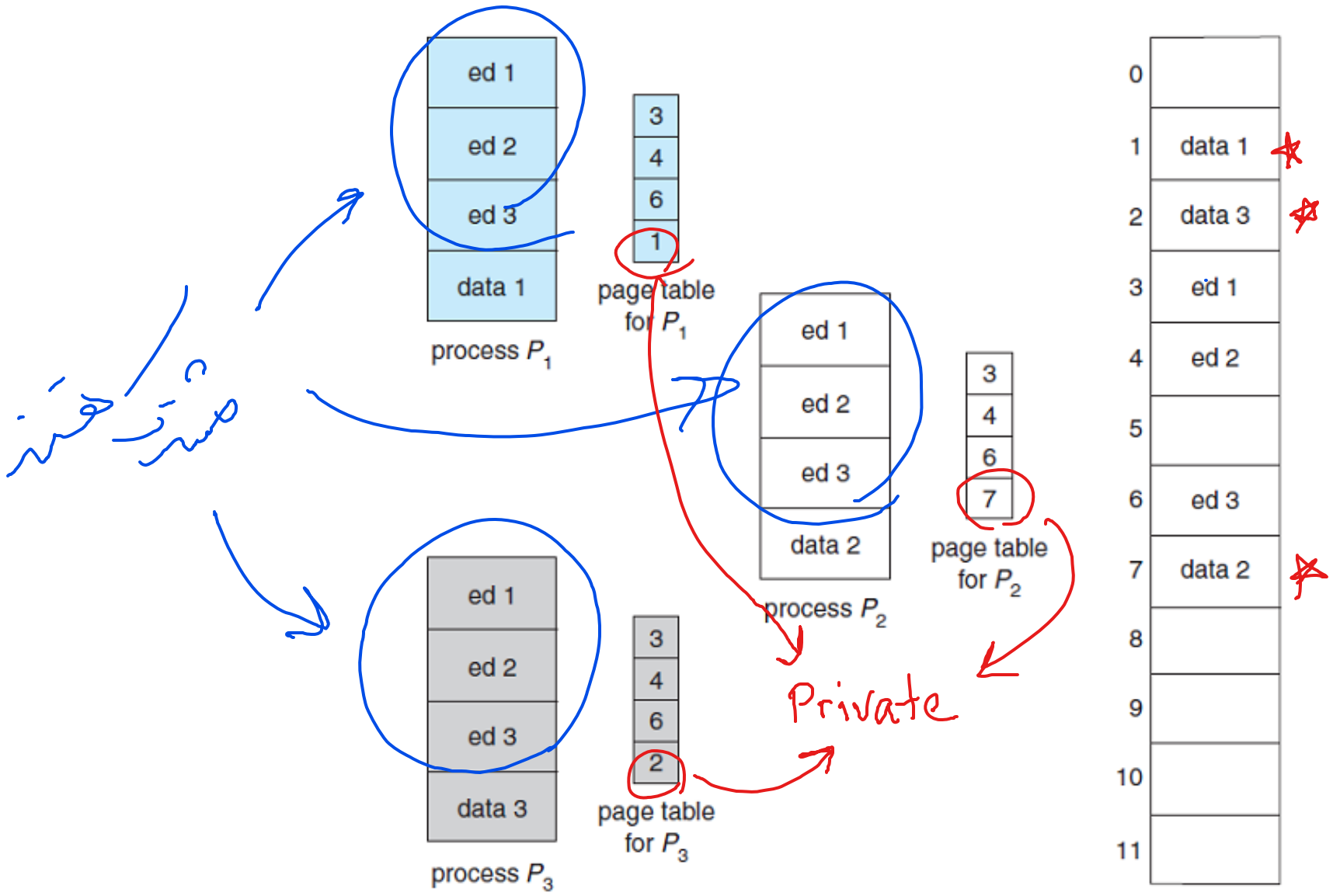
Shared pages



- **Shared code**
  - One copy of read-only (reentrant) code shared among processes (e.g., text editors)
  - Similar to multiple threads sharing the same process space.

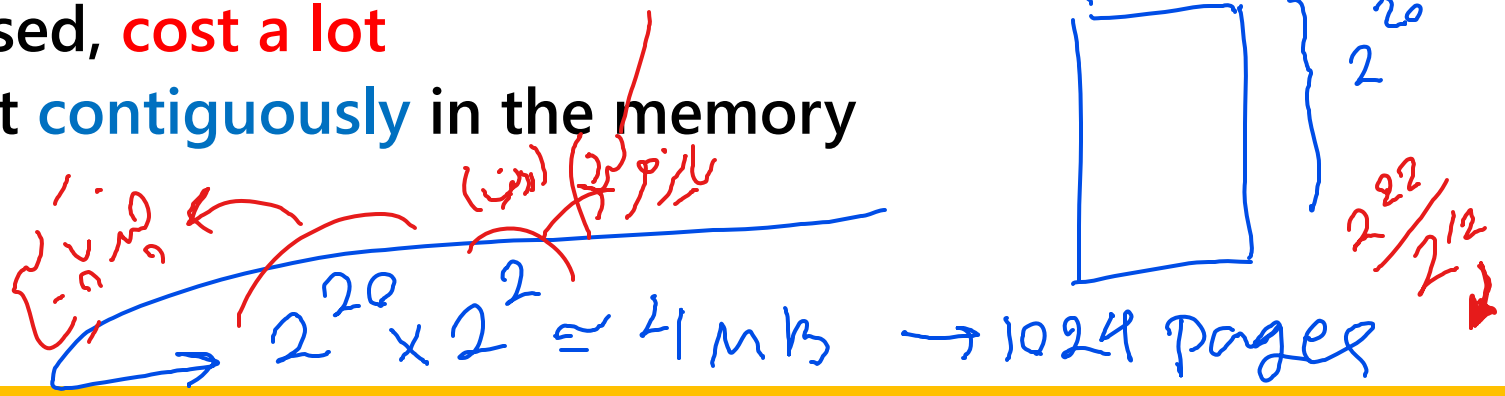
- **Private code and data**
  - Each process keeps a separate copy of the code and data
  - The page for the private code and data can appear anywhere in the logical address space

Shared pages - Example



Problem of big page tables

- **Memory structure for paging can get huge using straight-forward methods.**
- **Consider a 32-bit logical address space on a modern computers:**
  - Page size of 4 kB = 2^{12}
  - Page table would have 1 million entries ($2^{32}/2^{12}$) = 2^{20}
  - If each entry is 4 B: 4 MB of physical address space memory for page table alone.
  - The amount of memory used, cost a lot
  - Don't want to allocate that contiguously in the memory



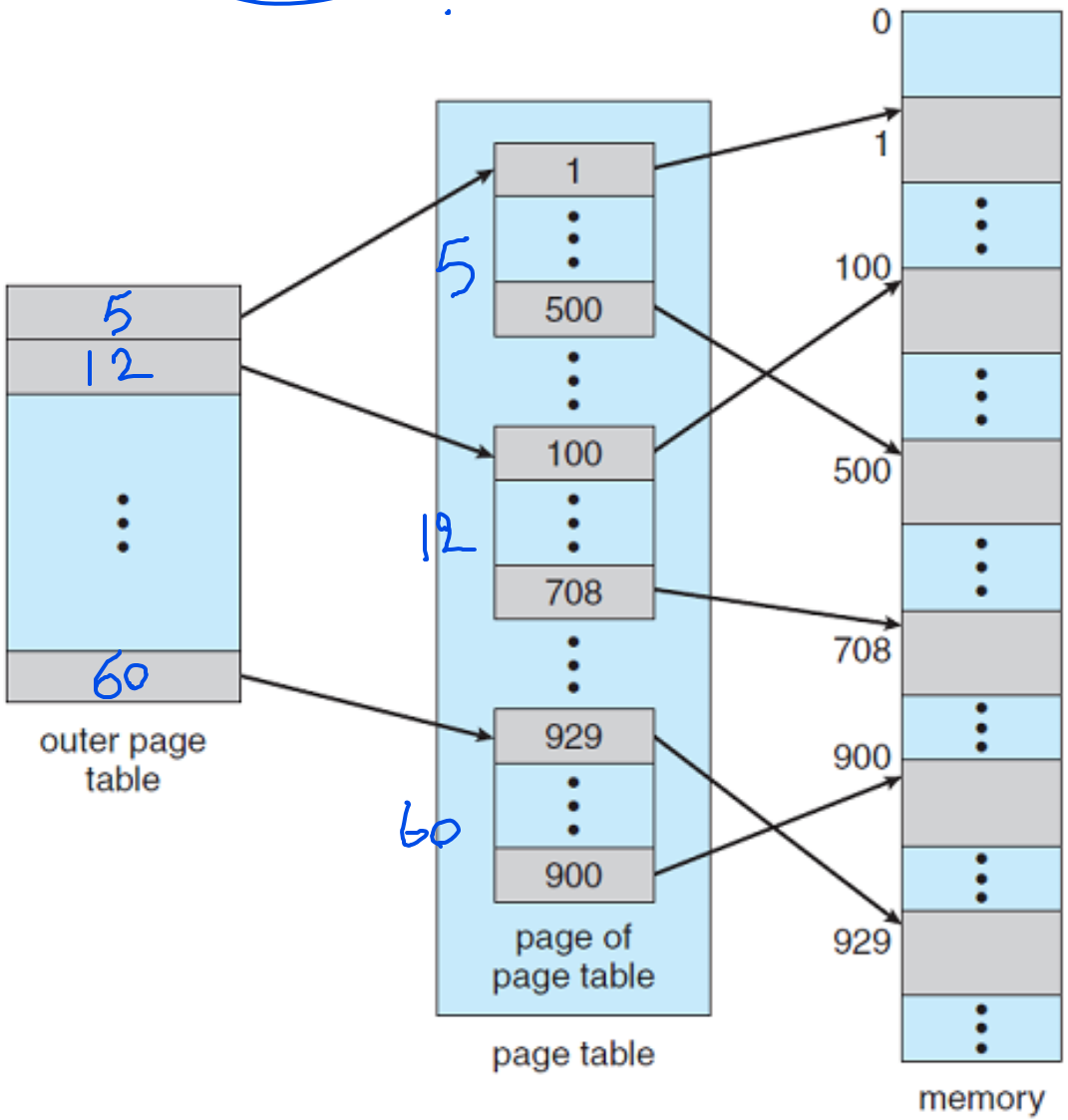
Solutions to maintain huge page tables

- A) Hierarchical paging
- B) Hashed page tables
- C) Inverted page tables

A) Hierarchical paging

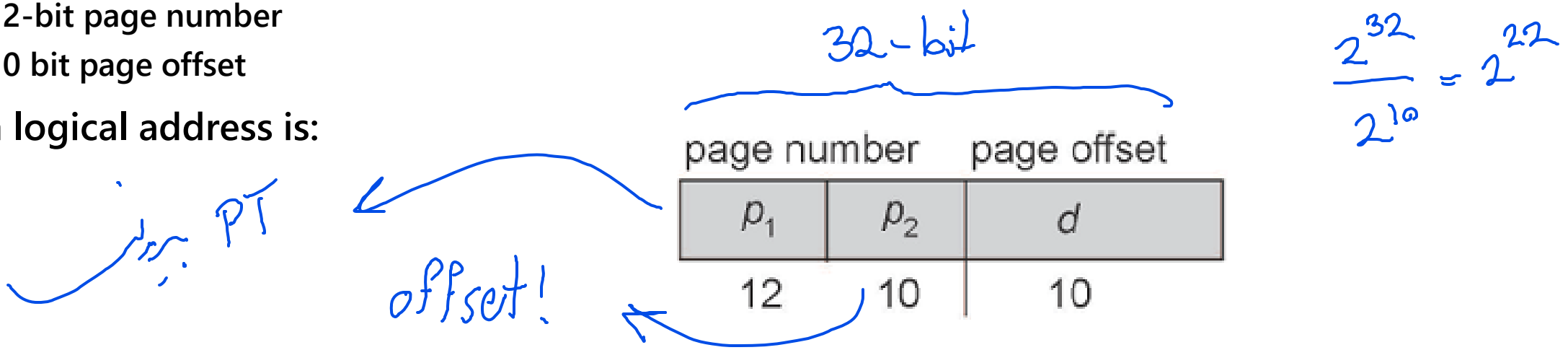
صفحه بندی سلسله مراتبی

- Use of two-level page table!
- We then page the page table!



Two-level paging - example

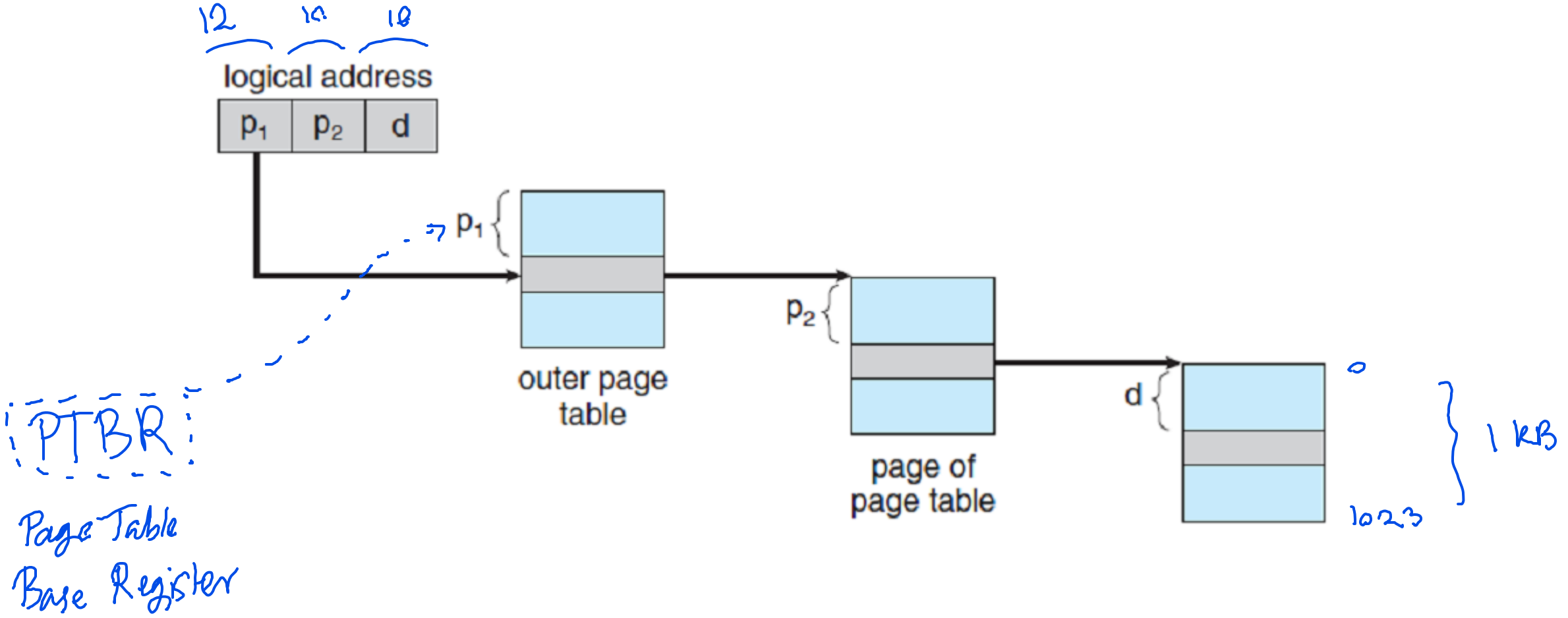
- A logical address, on 32-bit machine with 1 kB page size, is divided:
  - A page offset consisting 10 bits
  - A page number consisting of 22 bits
- Since the page table is paged, the page number is divided into:
  - A 12-bit page number
  - A 10 bit page offset
- Thus a logical address is:



where p_1 is an index into outer page table, and p_2 is the displacement within the page of inner page table

- Known as forward-mapped page table

Address-translation scheme



64-bit logical address space

- Even two-level paging scheme is not sufficient

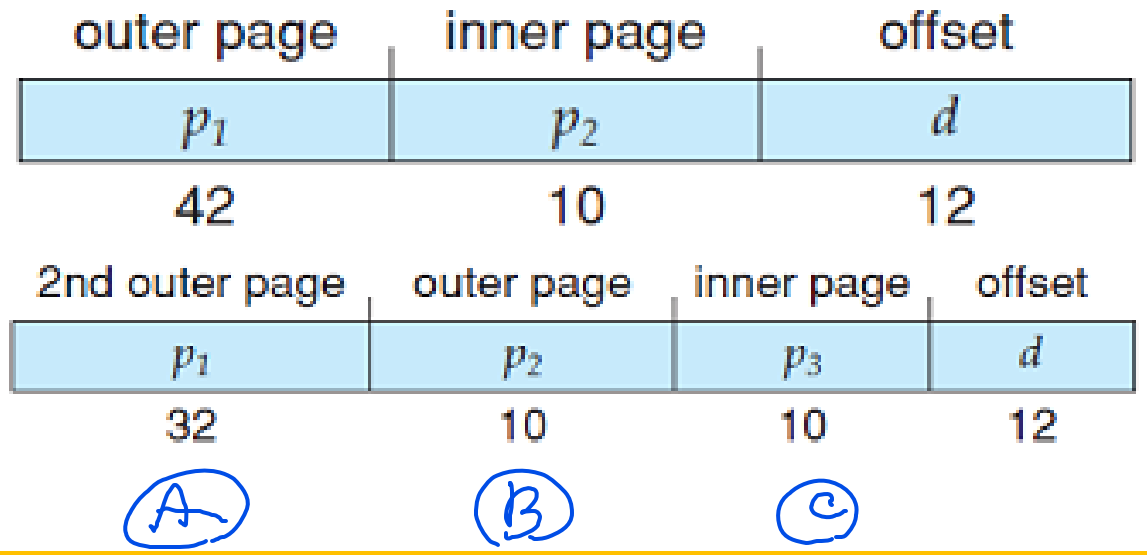
- If page size of 4 kB (2^{12})
  - Page table entries = 2^{52}
  - Assuming each page table entry size = 4 B
  - If two-level scheme, inner page table could be 2^{10}, 4 kB entries
  - Outer page table has 2^{42} entries or 2^{44} B
  - Address would look like:

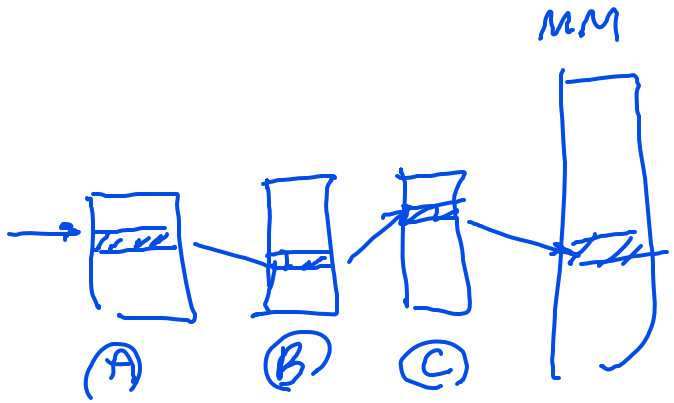
outer page	inner page	offset
p_1	p_2	d
42	10	12

64 bit

Three-level paging scheme

- One solution is to add a 2nd outer page table
- But in the following example, the 2nd outer page table is still 2^{34} bytes in size!
- And possible 4 memory access to get to one physical memory location:

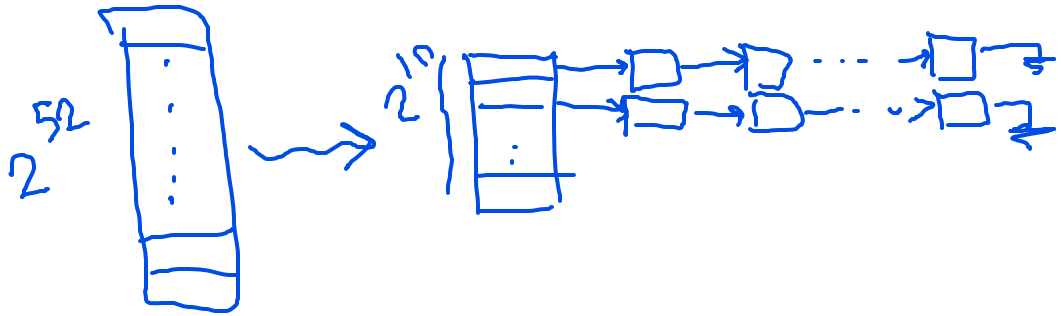




B) Hashed page tables

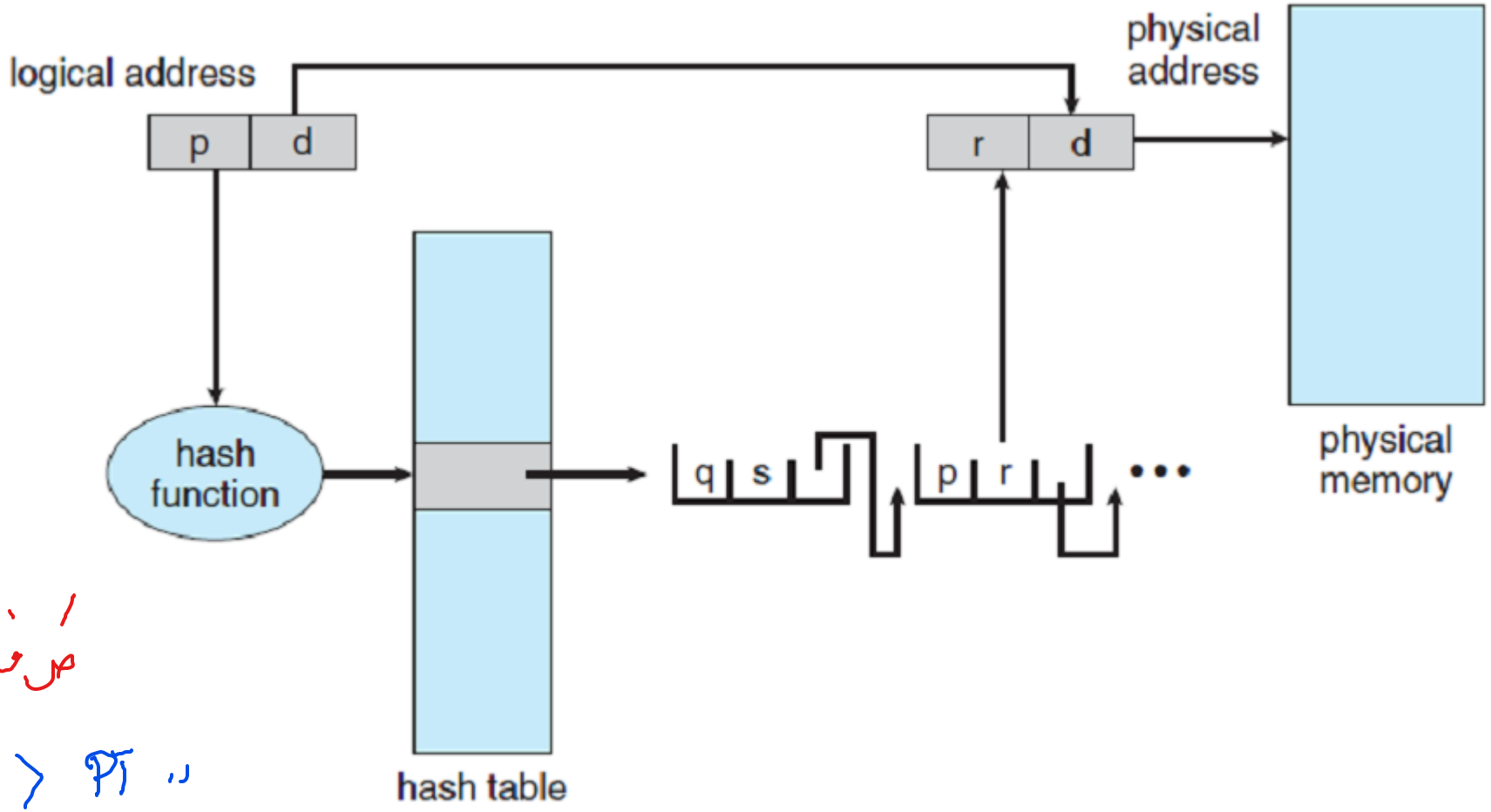
- Common in address space > 32 bits

- The logical page number is hashed into a page table
- This page table contains a chain of elements hashing to the same location



Hash page table scheme

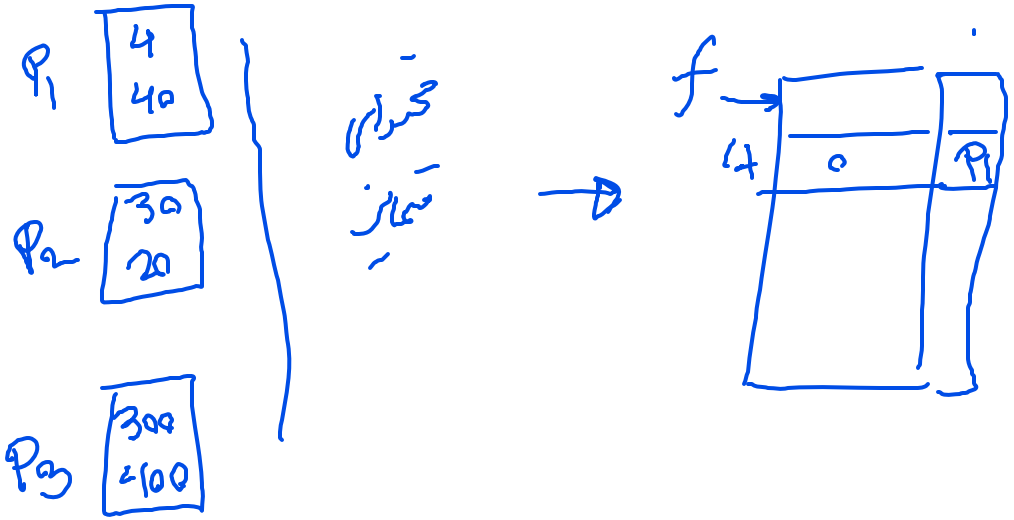
- Each element contains
  - Logical page number
  - Physical frame number
  - Pointer to the next node



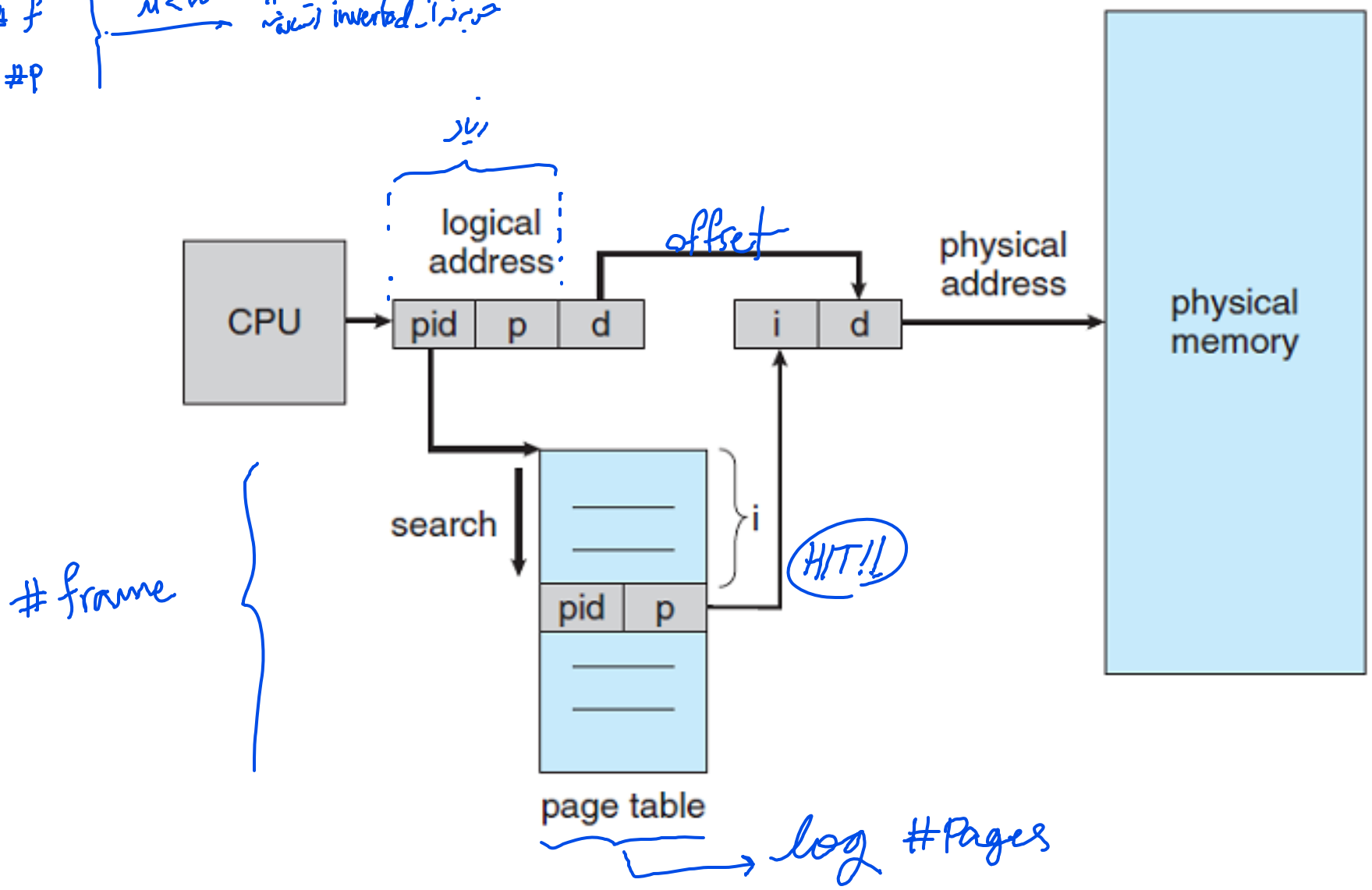
- Search is done serially in the linked list

C) Inverted page table

- Rather than keeping all possible logical page numbers, track all physical pages (frame) numbers
- One entry for each physical page of memory
- Entry consists of
  - Virtual address of the page stored in that frame
  - + Process info



Inverted page table scheme



Problem and solutions

- **Good:**
  - Decrease memory needed
- **Bad:**
  - Increase time needed to search the table

- Use hash table to limit the search to one, or at most a few, page-table entries.

- How to implement shared memory?
  - One mapping of a virtual address to the shared physical address

Questions?