جامعة ألأنبار

كلية علوم الحاسوب وتكنولوجيا المعلومات

قسم أنظمة شبكات الحاسوب

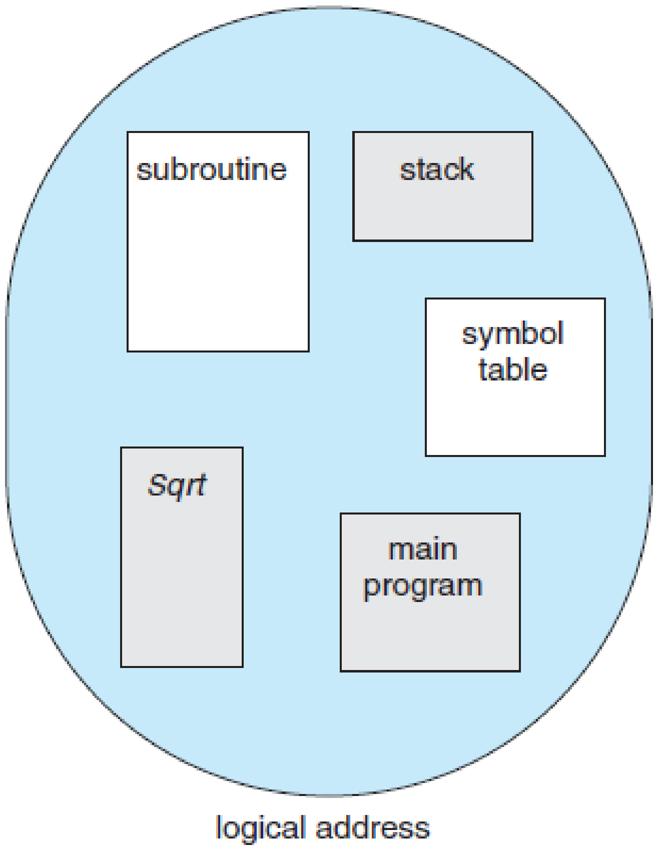
المرحلة الرابعه

Operating System

التدريسي: أ.م.د عمر منذر حسين

Segmentation

- **Segmentation** is a memory-management scheme that permits the physical address space of a process to be noncontiguous, it is a collection of segments.



- Segments vary in length, and the length of each is intrinsically defined by its purpose in the program.
- Each segment has a name (number) and a length.
- The addresses specify both the segment name and the offset within the segment.
- The programmer therefore specifies each address by two quantities: a segment name and an offset.
- Thus, a logical address consists of a two tuple:
- <segment-number, offset>

- This mapping is effected by a segment table.
- Each entry in the segment table has a segment base and a segment limit.
- The segment base contains the starting physical address where the segment resides in memory, and the segment limit specifies the length of the segment (Figure 8.8).
- A logical address consists of two parts: a segment number, s, and an offset into that segment, d.

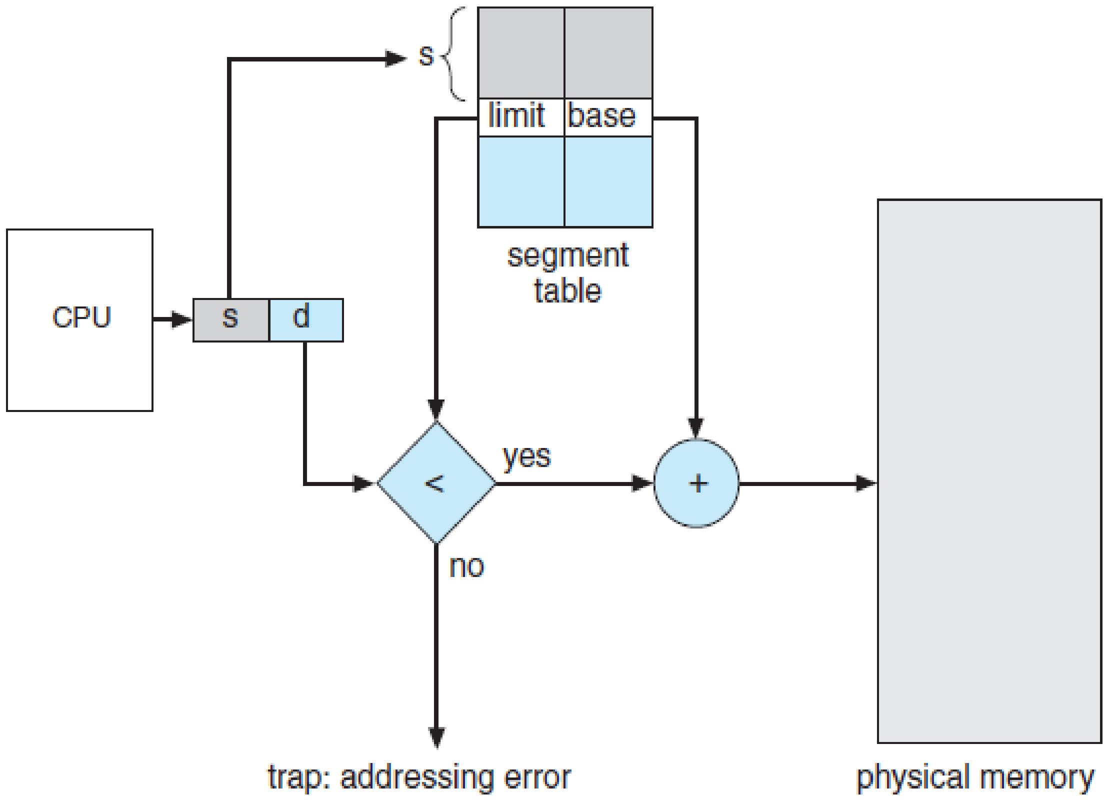
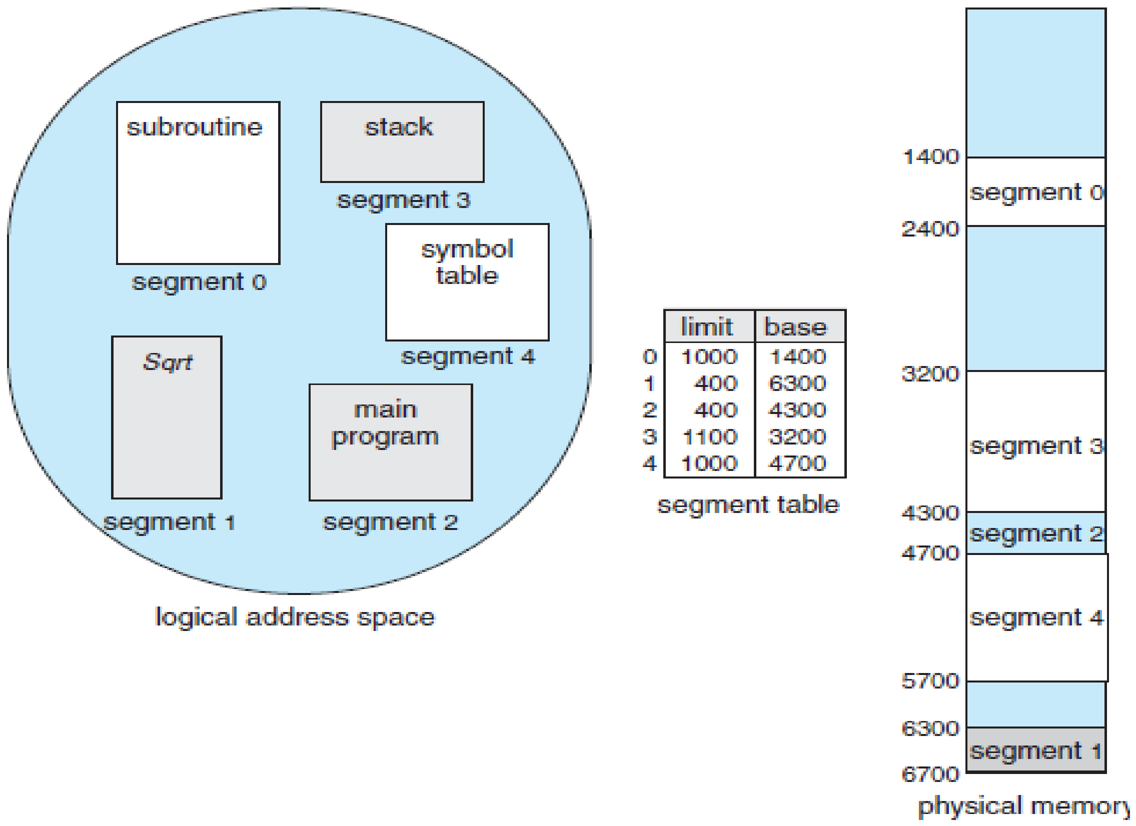


Figure 8.8 Segmentation hardware.

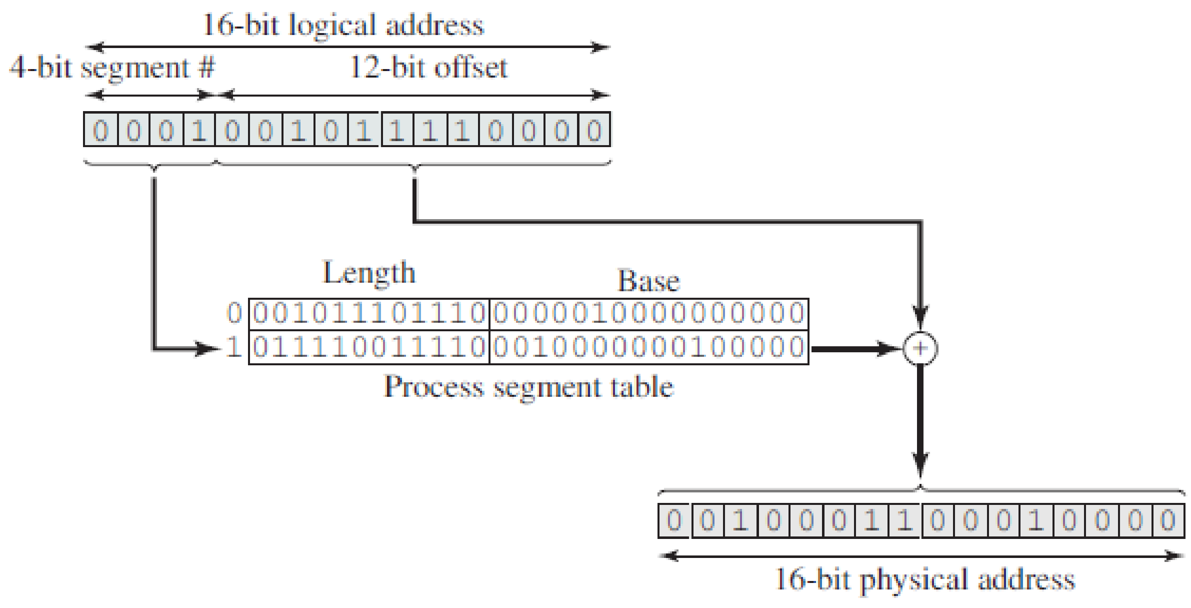


	limit	base
0	1000	1400
1	400	6300
2	400	4300
3	1100	3200
4	1000	4700

Figure 8.9 Example of segmentation.

Logical to Physical address translation

- Address of n + m bits.
- The leftmost bits are the segment number and the rightmost bits are the offset.
- 1- Extract the segment number as the leftmost bits of the address.
- 2- Use the segment number as the index to segmentation table to find the starting physical address of the segment.
- 3- compare the offset (rightmost m bits) to the length of the segment. If the offset is greater than or equal the length then the address is invalid.
- 4 the desired physical address is the sum of starting physical address of the segment plus the offset.



(b) Segmentation

Paging

- **Paging** is a non contiguous memory-management scheme.
- Avoids external fragmentation and the need for compaction.
- It also solves the considerable problem of fitting memory chunks of varying sizes onto the backing store.
- The basic method for implementing paging involves breaking physical memory into fixed-sized blocks called frames and breaking logical memory into blocks of the same size called pages.
- When a process is to be executed, its pages are loaded into any available memory frames from their source.
- Figure illustrate the use of page and frame.

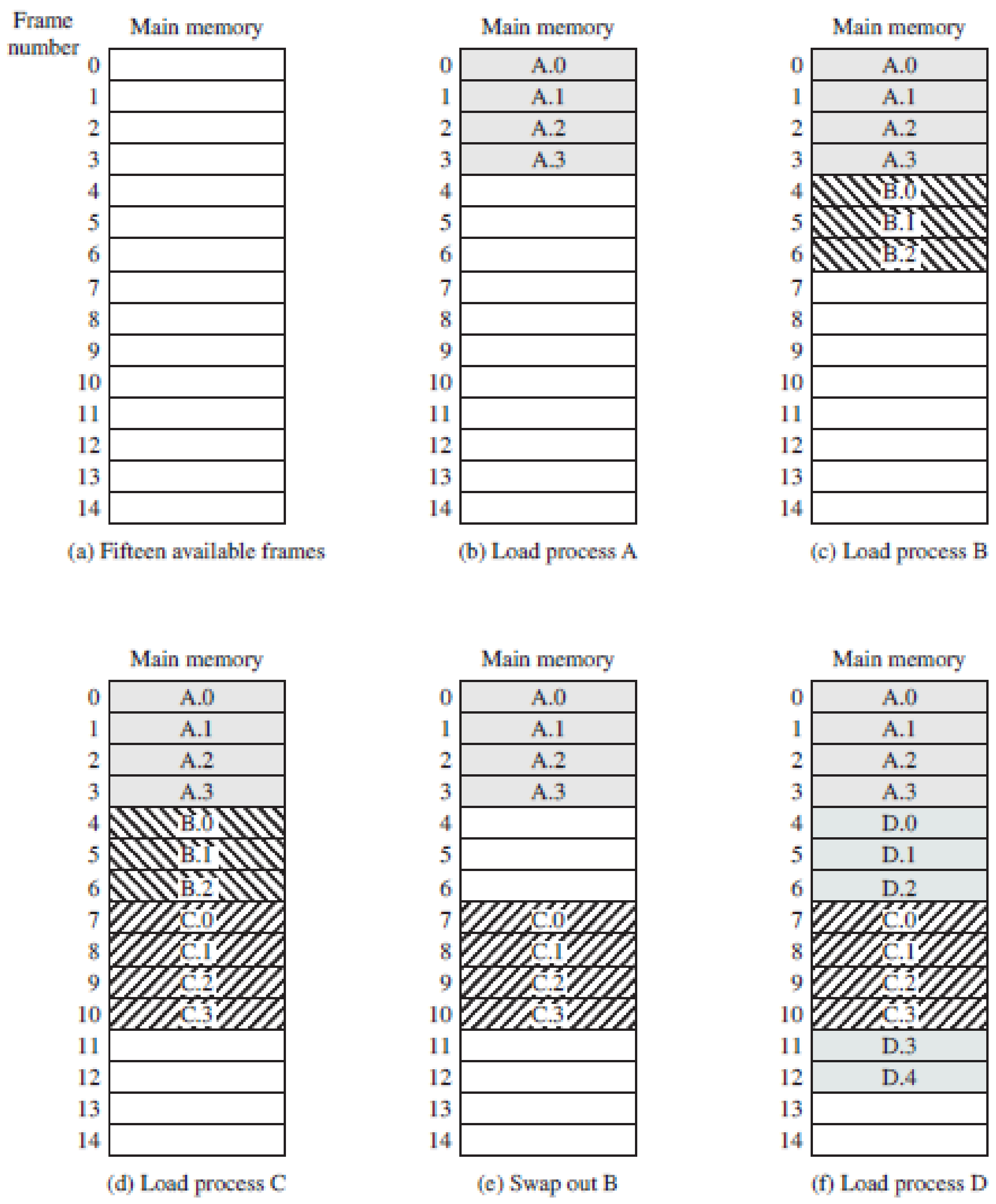


Figure 7.9 Assignment of Process to Free Frames

- Some of frames in memory are in use and some are free.
- Process **A** stored on disk consist (4 pages), **B** (3 pages), **C** (4 pages) and **D** is (5 pages).
- The backing store is divided into fixed-size blocks that are the same size as the memory frames or cluster of multiple frames.
- The hardware support for paging is in Figure 8.10.

- Every address generated by the CPU is divided into two parts: a **page number (p) and a page offset (d).**
- The page number is used as an index into a page table.
- The page table contains the base address of each page in physical memory.
- This base address is combined with the page offset to define the physical memory address that is sent to the memory unit.
- The paging model of memory is shown in Figure 8.11.

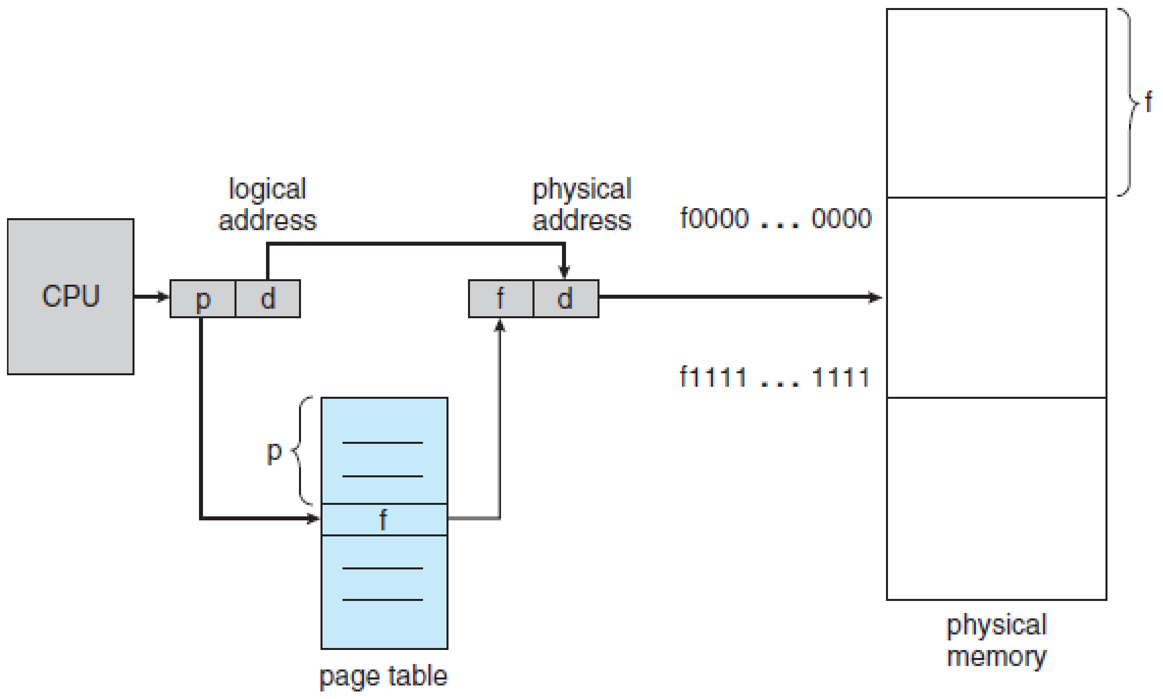


Figure 8.10 Paging hardware.

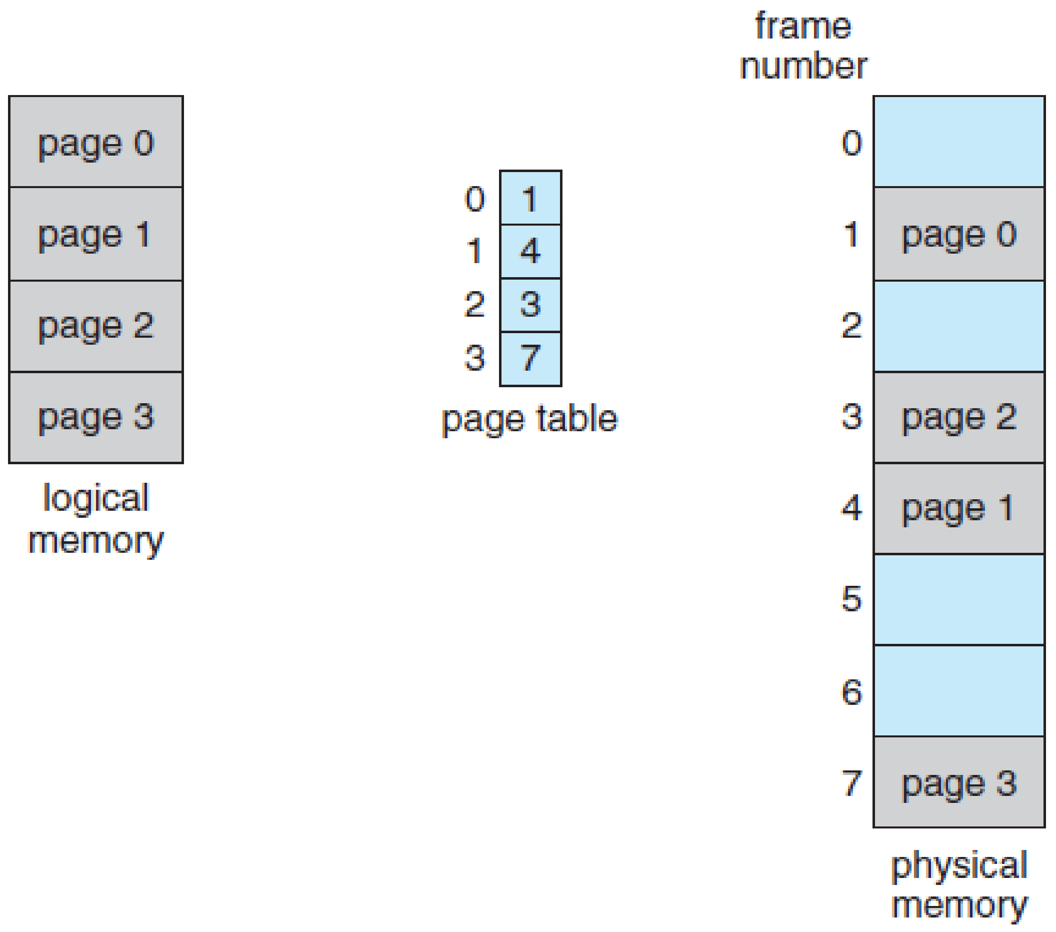
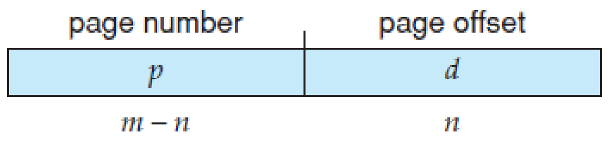


Figure 8.11 Paging model of logical and physical memory.

- Simple paging is similar to fixed partitioning with some differences.
- The partitions are smaller.
- A process may occupy more than one partition and those partitions needn't be contiguous.
- The size of page is power of 2 ranging from 512 bytes to 1 GB.
- Power of 2 page size makes translation of logical address into a page number and a 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.

- In the logical address n = 2 and m = 4.
- Page size 4 bytes and physical memory 32 bytes (8 pages).

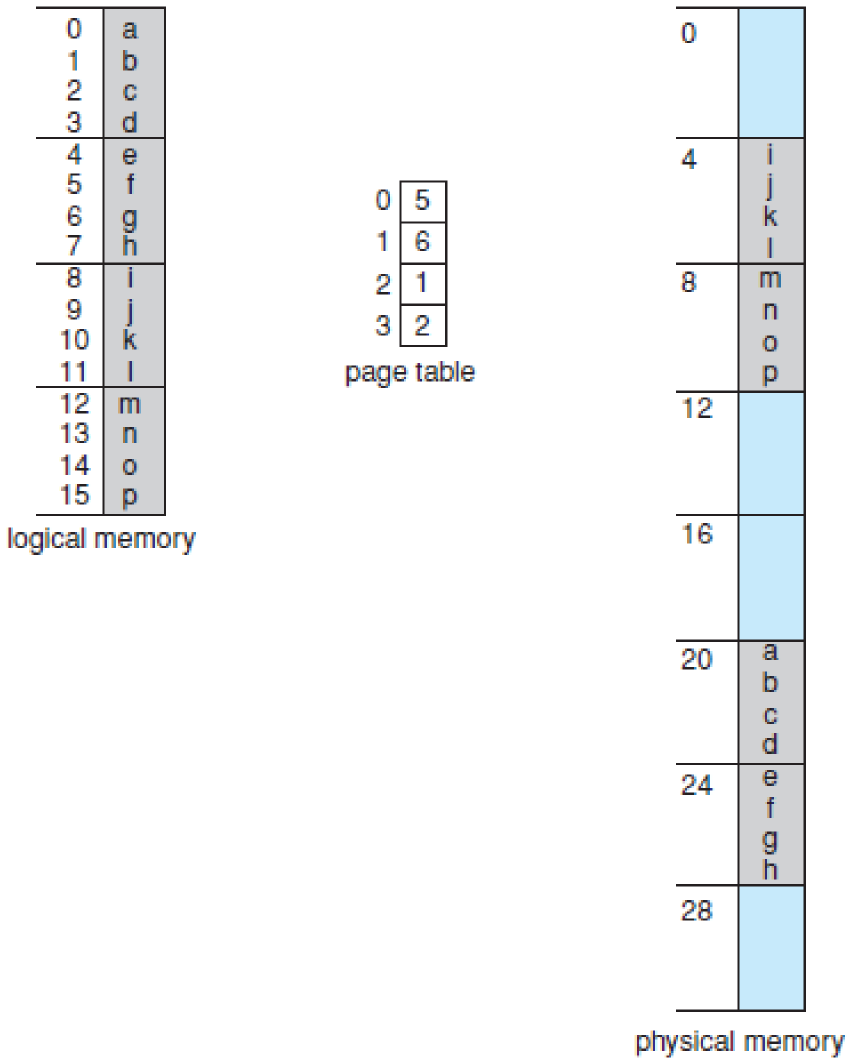


Figure 8.12 Paging example for a 32-byte memory with 4-byte pages.

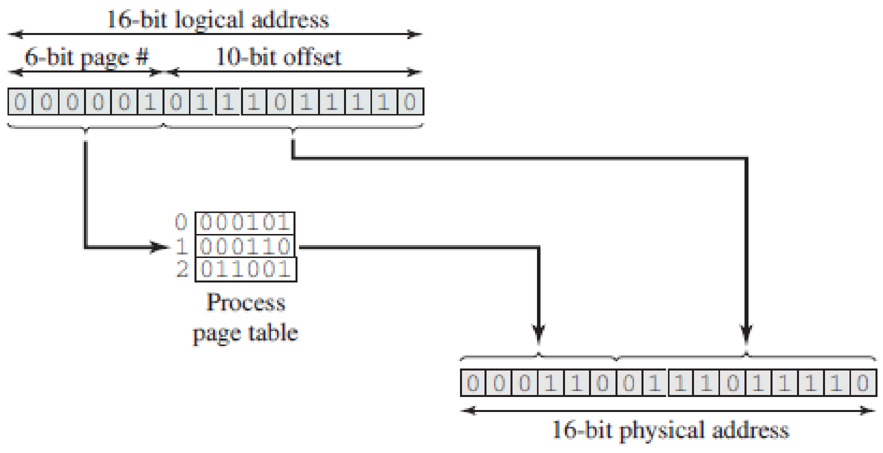
- Logical address 0 is page 0 and offset 0.

- From page table, page 0 is in frame 5.
- The physical address is 20 (5 * 4 + 0).
- Logical address 3 = physical address 23.
- Logical address 4 = physical address 24.
- Logical address 13 = physical address 9 (frame 2 * 4 (size of frame) + 1 (offset)).

Logical to Physical address translation

- Address of n + m bits.
- The leftmost bits are the segment number and the rightmost bits are the offset.
- 1- Extract the page number as the leftmost n bits of the address.
- 2- Use the page number as the index to page table to find the frame number k.
- 3- The starting physical address of the frame is k * 2^m and the physical address of the referenced byte is that number plus the offset.
- This physical address need not be calculated, it is easily constructed by appending the frame number to the offset.

In our example, we have the logical address 0000010111011110, which is page number 1, offset 478. Suppose that this page is residing in main memory frame 6 = binary 000110. Then the physical address is frame number 6, offset 478 = 0001100111011110 (Figure 7.12a).



(a) Paging

- With paging scheme no external fragmentation, any free frame can be allocated to a process that needs it.
- We may have some internal fragmentation.
- Example: page size 2048 bytes, a process of 72766 bytes will need 35 pages plus 1086 bytes.
- It will be allocated 36 frames resulting in internal fragmentation of 2048 – 1086 = 962 bytes.
- In 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.
- Average amount of internal fragmentation: **one-half of one page per process**.
- This suggests: **a small page size**.
- However, the **more pages** we have, the **more management** is needed.
- Today, most pages sizes lie between 8K and 4MB.
- For the page table, each entry is usually **4 bytes long**.

- 32-bit entry can point to one of 2^{32} physical **page frames**.
- If each page frame is 4KB, then a system with 4-byte entries can address 2^{44} bytes (or 16TB) of physical memory.
- For new processes of ‘n’ pages, there must be ‘n’ frames available.
- If available, they are allocated to the process.

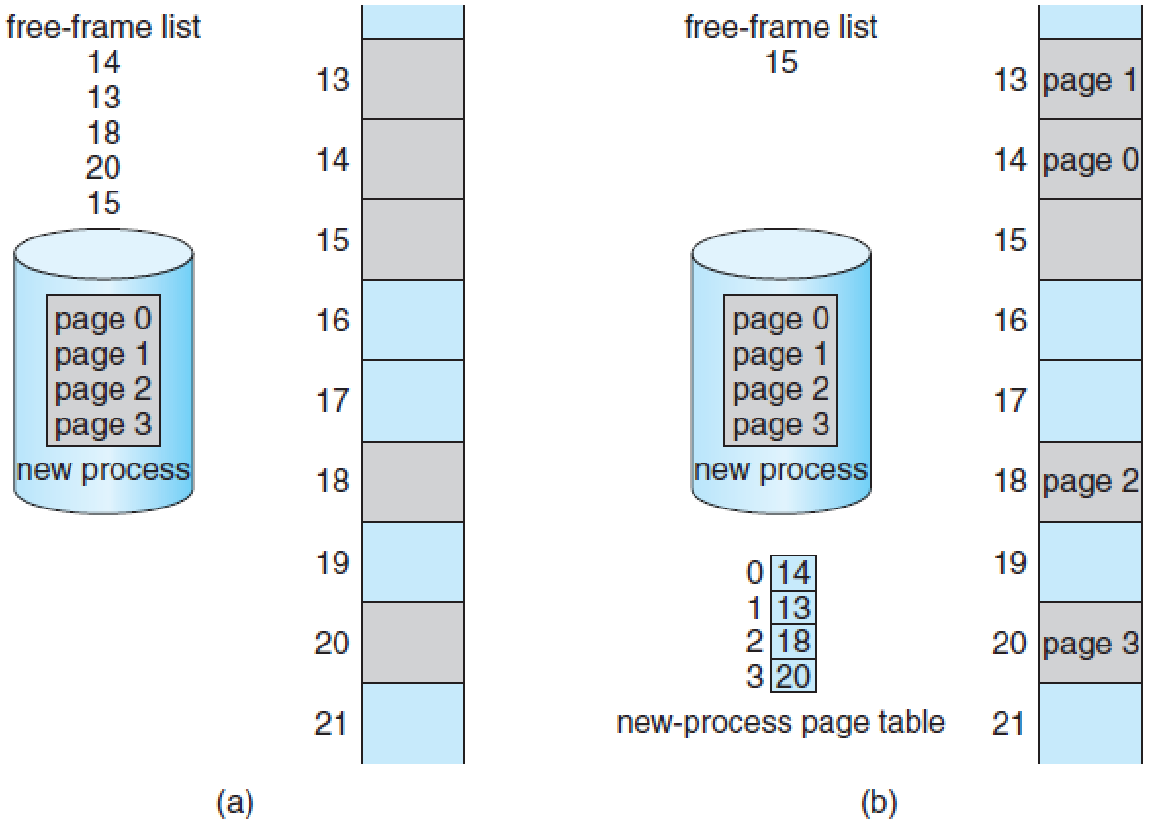


Figure 8.13 Free frames (a) before allocation and (b) after allocation.

- The operating system maintains **a copy of the page table for each process**.
- It is used by the CPU dispatcher to define the hardware page table when a process is to be allocated the CPU.
- Paging therefore **increases the context-switch time**.
- The hardware implementation of the page table can be done in several ways:

- **Simple: Set of dedicated registers (built with very high-speed logic to make the paging address translation efficient).**
- Every access to memory must go through the paging map, so efficiency is a major consideration.
- This is a good solution if the page table is small (e.g. 256 entries).
- Unfortunately, many large systems allow for a page table to be in the order of one million entries.
- **Other way**: keep the paging table in the main memory and use page-table base register (PTBR) to point to the page table.
- Changing page tables requires changing only this one register, substantially reducing context-switch time.
- But there is a problem with this approach: **Every data/instruction access requires two memory accesses:**
- One for the page table and one for the data/instruction.
- The two memory access problem can be solved by the use of a special fast lookup hardware cache called **translation look-aside buffers** (TLBs).
- Each entry in the TLB consists of two parts: **a key (or tag) and a value**.
- The search in TLB is fast; a TLB lookup in modern hardware is part of the instruction pipeline, essentially adding no performance penalty. (TLB must be kept small. It is typically between 32 and 1,024 entries in size).
- The TLB is used with page tables in the following way:
- The TLB contains only a few of the page-table entries.
- Logical address is generated by the CPU.
- The 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.
- If the page number is not in the TLB (known as a **TLB miss**), a memory reference to the page table must be made.

- When the frame number is obtained, we can use it to access memory. In addition, we add the page number and frame number to the TLB, so they be found quickly on the next reference.

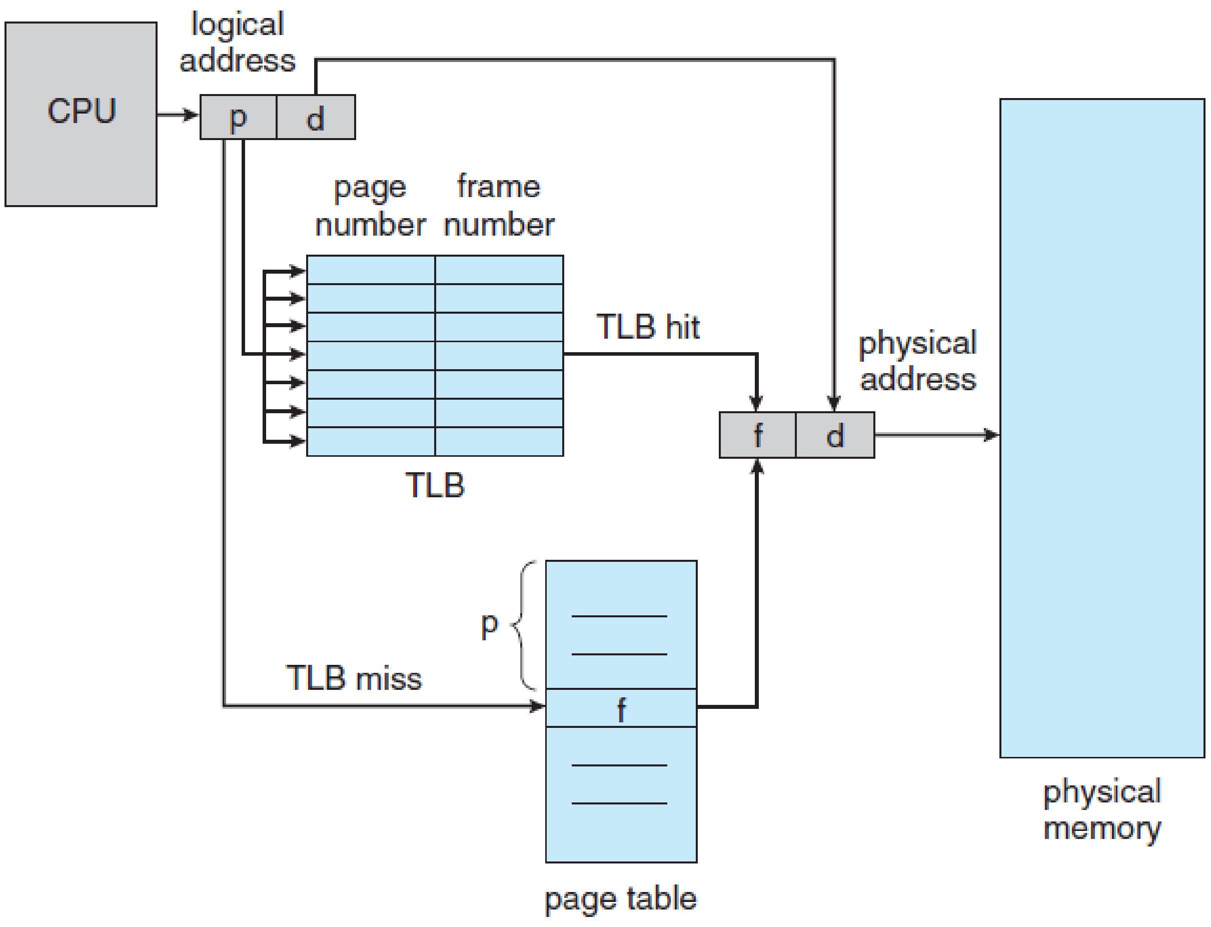


Figure 8.14 Paging hardware with TLB.

- The percentage of times that the page number of interest is found in the TLB is called the **hit ratio**.
- An 80-percent hit ratio, for example, means that we find the desired page number in the TLB 80% of the time.
- If it takes 100 nanoseconds to access memory, then a mapped-memory access takes 100 nanoseconds when the page number is in the TLB.
- If we fail to find the page number in the TLB then we must first access memory for the page table and frame number (100 nanoseconds) and then access the desired byte in memory (100 nanoseconds), for a total of 200

nanoseconds. (We are assuming that a page-table lookup takes only one memory access)

- To find the effective memory-access time, we weight the case by its probability:
- **effective access time = 0.80 × 100 + 0.20 × 200 = 120 nanoseconds**
- In this example, we suffer a 20-percent slowdown in average memory-access time (from 100 to 120 nanoseconds).
- For a 99-percent hit ratio, which is much more realistic, we have
- effective access time = 0.99 × 100 + 0.01 × 200 = 101 nanoseconds.
- **Protection**
- One additional bit is generally attached to each entry in the page table: a **valid–invalid bit**.
- When this bit is set to ***valid***, the associated page is in the process's logical address space and is thus a legal (or valid) page.
- When the bit is set to ***invalid***, the page is not in the process's logical address space.
- Illegal addresses are trapped by use of the valid–invalid bit.
- The operating system sets this bit for each page to allow or disallow access to the page.
- Example ,a system with a 14-bit address space (0 to 16383), we have a program that should use only addresses 0 to 10468.
- Given a page size of 2 KB, we have the situation shown in Figure 8.15.
- Addresses in pages 0, 1, 2, 3, 4, and 5 are mapped normally through the page table.
- Any attempt to generate an address in pages 6 or 7, however, will find that the valid–invalid bit is set to invalid, and the computer will trap to the operating system (invalid page reference).
- Notice that this scheme has created a problem. Because the program extends only to address 10468, any reference beyond that address is illegal.

- However, references to page 5 are classified as valid, so accesses to addresses up to 12287 are valid. Only the addresses from 12288 to 16383 are invalid.

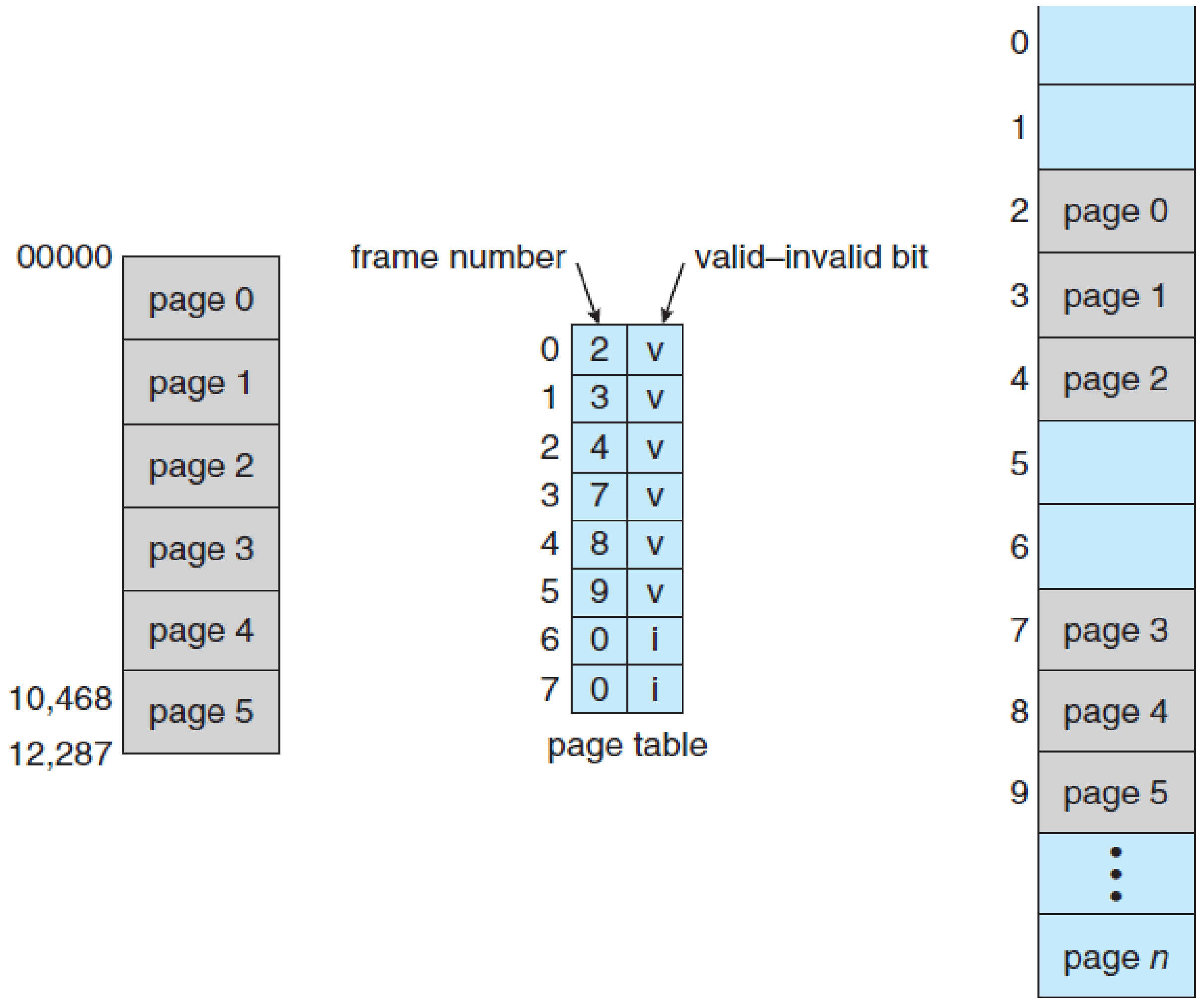


Figure 8.15 Valid (v) or invalid (i) bit in a page table.