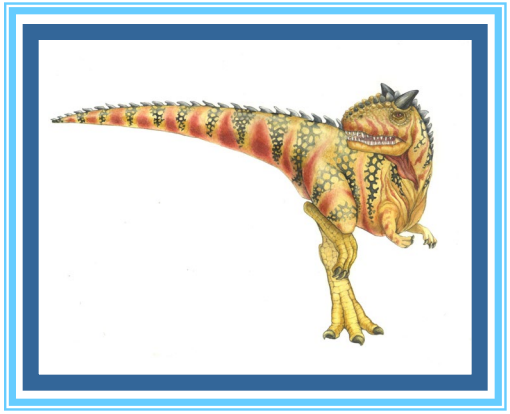
Chapter 9: Memory Management Part II



Fragmentation

- Both the **first-fit** and **best-fit** dynamic memory allocation strategies suffer from **external fragmentation**.
- **External fragmentation** exists when there is enough total memory space to satisfy an allocation request, but the available spaces are not contiguous and cannot be used:
  - *As processes are loaded and removed from memory, the free memory space (hole) is broken into further little pieces.*
  - *The memory is fragmented into a large number of small unused holes.*
- This **external fragmentation** problem can be severe.
  - *If all these small pieces of memory holes were combined into a big free hole, it might be helpful for process allocation*.

Fragmentation

- There will be an issue of **internal fragmentation** as well:
  - Consider a **hole of size18,464 bytes**. Suppose that the next process allocation request has a size of **18,462 bytes**. If the allocation exactly used the hole, there will be a **hole left with 2 bytes of size** —unused memory that is internal to a partition.
  - The *memory space allocated to a process is slightly larger than the process size, causing* ***internal fragmentation***.
- The overhead to keep track of these unused holes resulting from **internal fragmentation** will be substantially larger than the hole itself.
  - **The general approach to avoiding internal fragmentation** is *to break the physical memory into* ***fixed-sized blocks*** *and allocate memory based on the block size*.

Fragmentation

- One solution to the problem of **external fragmentation** is **compaction:**
  - *The **compaction** is to shuffle to place all free memory holes together in one large block.*
- **Compaction** is not always possible: If relocation is static and is done at assembly or load time, compaction cannot be done.
- The simplest **compaction** algorithm:
  - To move all processes toward one end of memory;
  - All holes move in the other direction, producing one large hole of available memory.
  - This scheme is expensive to implement.

Fragmentation

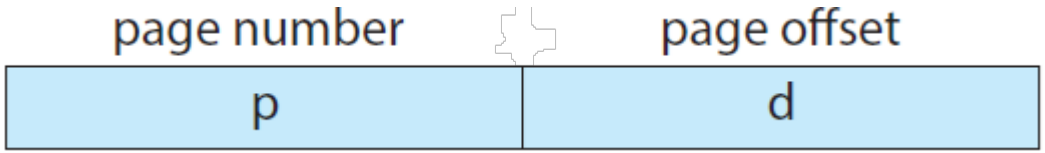
- Another possible solution to the **external-fragmentation** is to permit the ***logical address space of processes to be noncontiguous***, thus allowing a process to be allocated physical memory wherever such memory is available.
- This is the strategy used in ***paging,*** the most common memory-management technique for computer systems.
- Fragmentation is a general problem in computing that can occur wherever blocks of data are managed in memory.

Paging

- **Paging** is a memory management scheme that permits a process's physical address space to be noncontiguous.
- **Paging** avoids **external fragmentation** and is implemented through cooperation between the operating system and the computer hardware.
- The basic method for implementing **paging** involves **breaking main memory into fixed-sized blocks** called **frames** and breaking **logical memory space** (disk memory space or program's memory) into **fixed-sized blocks** called **pages.**
- An **executing program** is converted into **pages** in its logical space and is **loaded into available memory frames** dynamically.

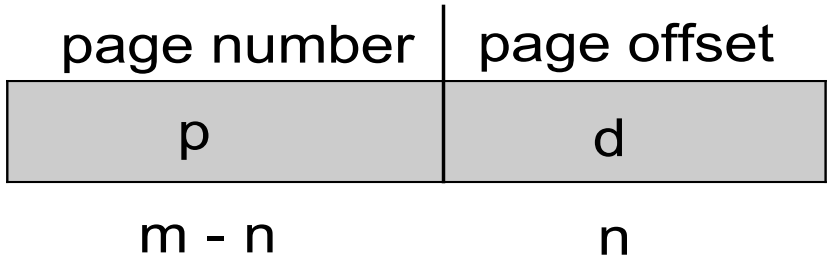
Paging

- For example, the **logical address space (disk address space)** is now totally separate from the **physical address space (main memory address space)**, so a process can have a logical **64-bit address space even though the system has less than 2^{64} bytes of physical memory space**.
- In **paging**, every address generated by the CPU (called a **page address**) is divided into two parts:
  - a **page number** (**p**) and
  - a **page offset** (**d**):



Paging

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



If PA = 32bits (*m*) and its offset size (n) is 5 bits, then its page no. size is (*m*-*n*) = 27 bits.

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

Paging

- The **page number (p)** is used as an index into a per-process **page table**. This is illustrated in **Figure 9.8**.
- The **page table** contains the **base address of each frame** in physical memory, and the **offset** (**d**) is the location in the frame being referenced.
- Thus, the **base address of the frame** is combined with the **page offset** to define the **physical memory address**.

Paging Hardware

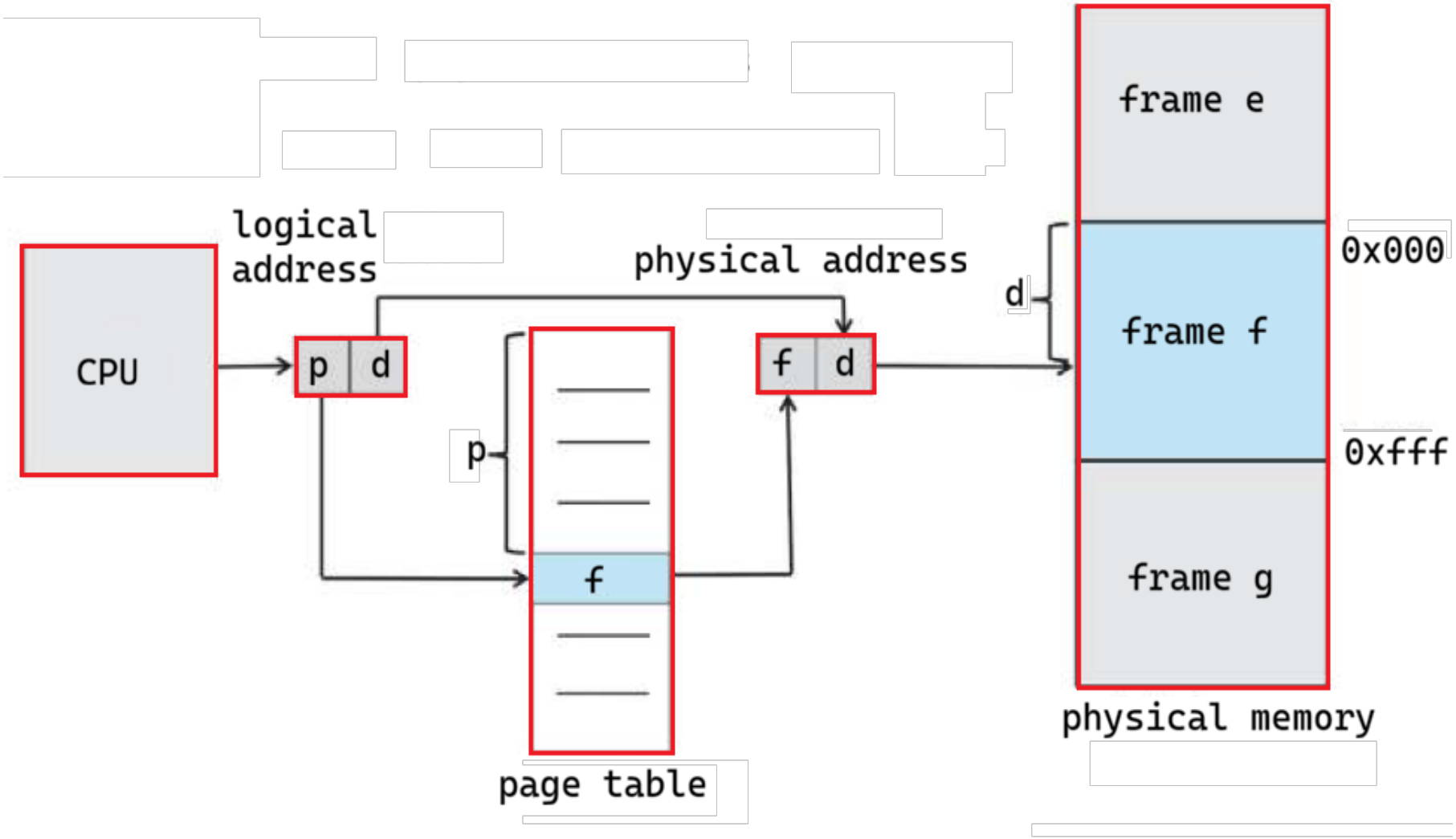


Figure 9.8 Paging hardware.

Paging

- The following outlines the steps taken by the **MMU** to translate a **logical address** generated by the CPU to a **physical address**:
  1. Extract the **page number *p*** and use it as an **index into the page table.**
  2. Extract the corresponding **frame number *f* from the page table.**
  3. Replace the **page number *p* in the logical address with the frame number *f*** .
- As the **offset *d*** does not change, it is not replaced, and the **frame number** and **offset** now comprise the **physical address**.
- The paging model of memory is shown in **Figure 9.9**.

Paging

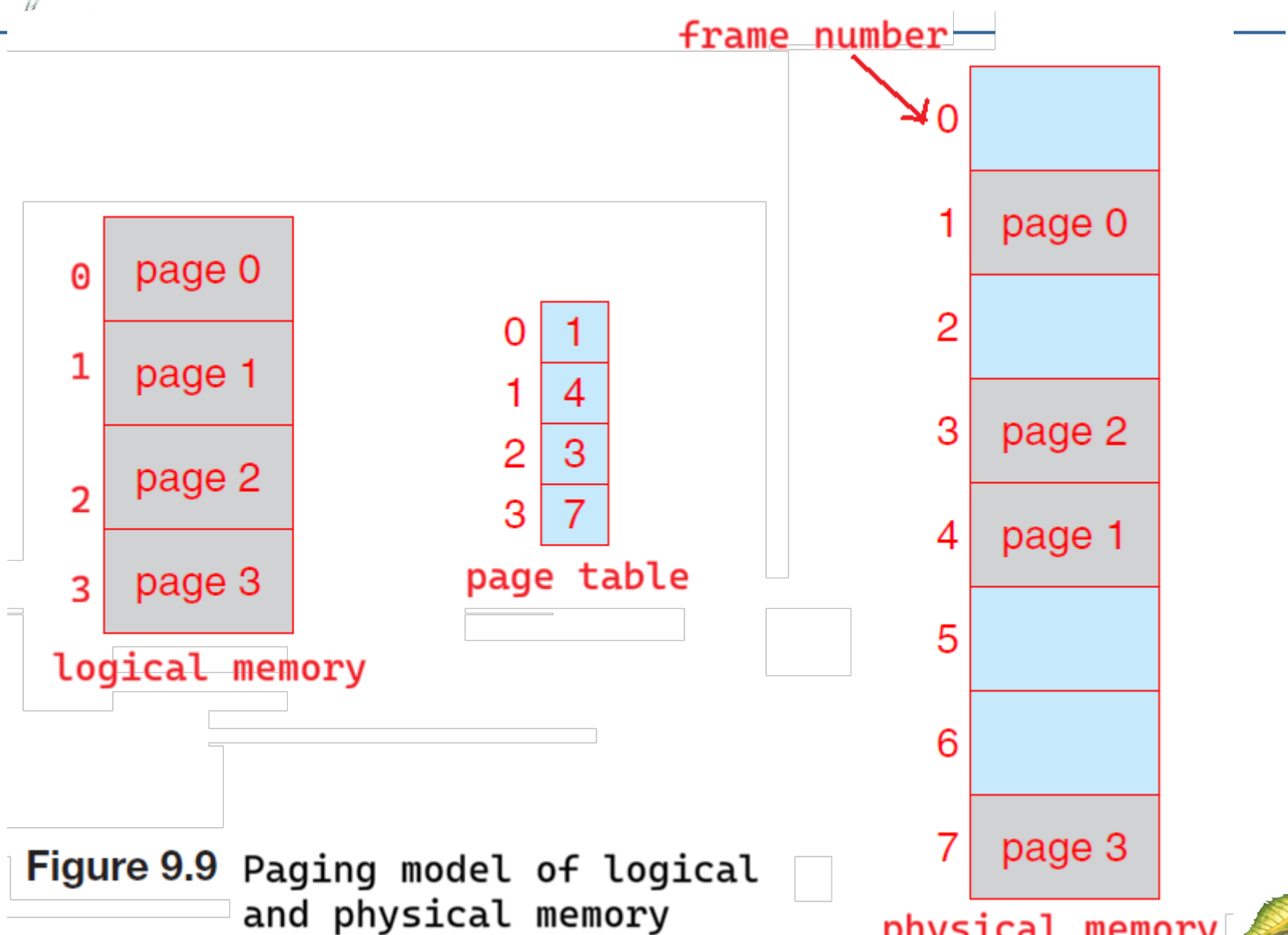
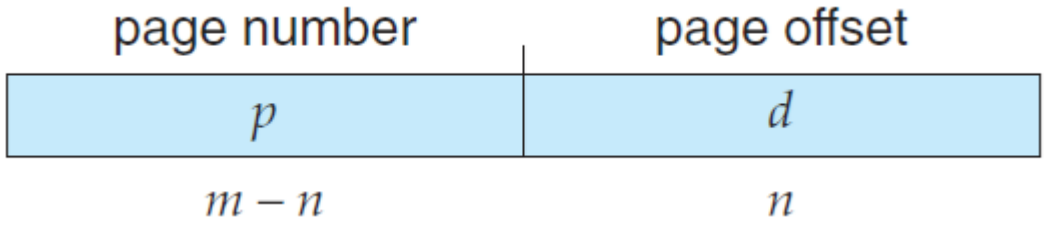


Figure 9.9 Paging model of logical and physical memory

Paging

- The **page size** (same as the **frame 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 of (***m*–*n***) **bits** of a logical address designate the **page number**, and the ***n*** low-order bits designate the **page offset**.
  - where ***p* is an index** into the page table, and ***d* is the displacement** within the page.

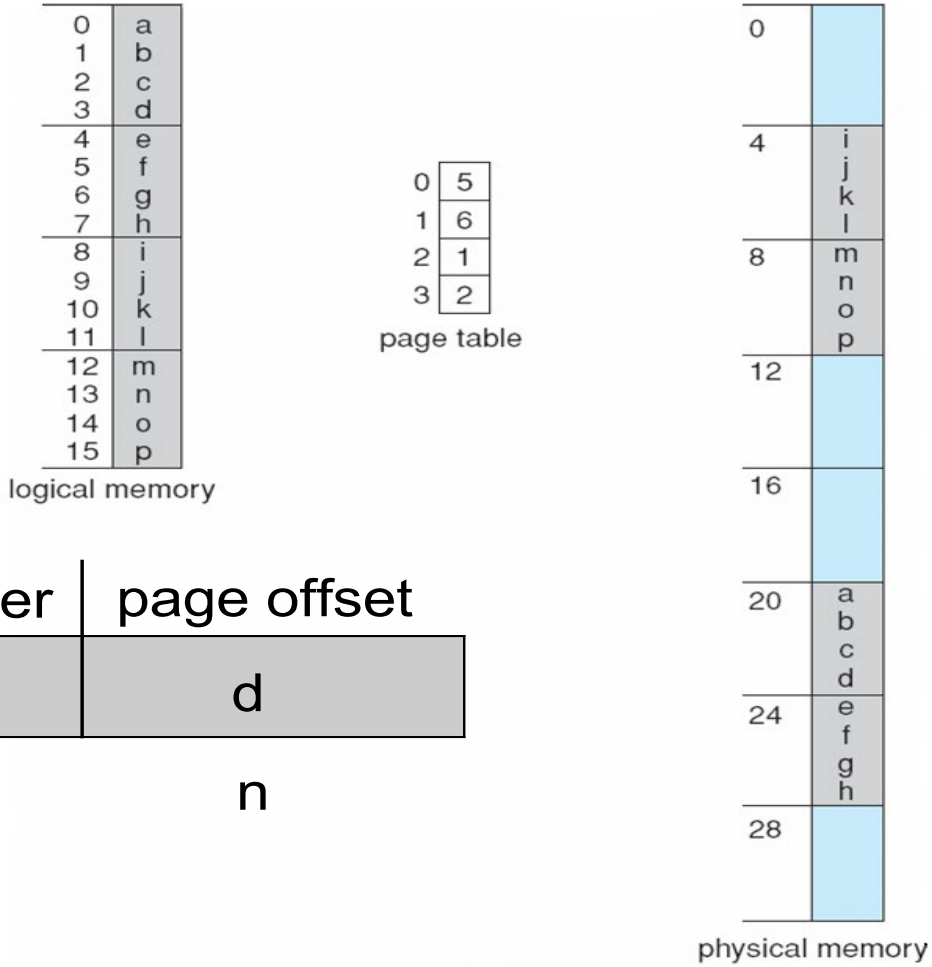


Paging -Address Translation Scheme

[Physical address = (frame no. × logical address-bits) + offset]

- Consider the memory in **Figure 9.10**. Here, ***n* = 2** and ***m* = 4**. Using a **page size** of **4 bytes** and a physical memory of **32 bytes** (32 × 8) yields **8 pages**. **We show how the programmer's view of memory can be mapped into physical memory.**
- **Logical address 0** is (2-bit **page0**, 2-bit **offset 0 = 0000).** Indexing into the **page table**, notice that **page 0 is in frame 5**. Thus, **logical address 0** maps to **physical address 20** [= (5 × 4) + 0].
- **Logical address 3** (is 2-bit **page0**, 2-bit **offset 3 = 0011**) maps to physical address **23** [= (5 × 4) + 3].
- **Logical address 4** is (2-bit **page1**, 2-bit **offset 0** = **0100**); according to the page table, **page1** is mapped to **frame 6**. **Logical address 4** maps to physical address **24** [= (6 × 4) + 0].
- **Logical address 13 is (**2-bit **page3,** 2-bit **offset 1 = 1101**) maps to physical address **9** [= (2 ×4) + 1. ***What about logical address* 15?**

Paging



n =2 and *m* =4 32-byte memory and 4-byte pages

Figure 9.10 Paging example for a **32-byte** memory with **4-byte** pages.

Paging

Consider a virtual memory system with a page size of **128-bit** and a physical memory of **256 bytes**. Each logical address has a **4-bit** offset value. The **page table** is shown below:

Page no.	Frame address
0	11
1	6
2	10
⋮	⋮
14	7
15	2

Based on the **page table**, calculate the ***physical addresses*** (in decimal) of the following ***logical addresses***:

13.1. [1 point] 01_{16}
13.2. [1 point] $2B_{16}$
13.3. [1 point] $1A_{16}$
13.4. [1 point] FC_{16}

Paging

- Frequently, on a **32-bit CPU**, each **page-table entry is 4 bytes** long, but that size can vary as well.
- A **32-bit entry** can point to one of **232 physical page frames**. If the frame size is **4 KB** (212), then a system with **4-byte** entries can address 244 bytes (or 16 TB) of physical memory.

Paging

- Consider a computer with a 32-bit logical address (VA) and a **4 Kbits** page size. How many entries are required in a page table? If each page table entry requires **4 bytes**, what is the total size of the page table?
- Solution:

  number of pages = Logical memory (VM)size/page size = $2^{32}/2^{12} = 2^{20}$

  As there is one page table entry for each page, there are 2^{20} entries. If each entry is **4 bytes**, the page table requires **2^{22} bytes or 4M bytes** of memory.

  (that is, 2^{20} X 4 byte = 4M bytes).

Paging

- When a process arrives in the system to be executed, its ***size***, expressed in pages, is examined.
- Each **page** of the process needs one memory **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 recorded in the page table, and so on (**Figure 9.11**).

Paging

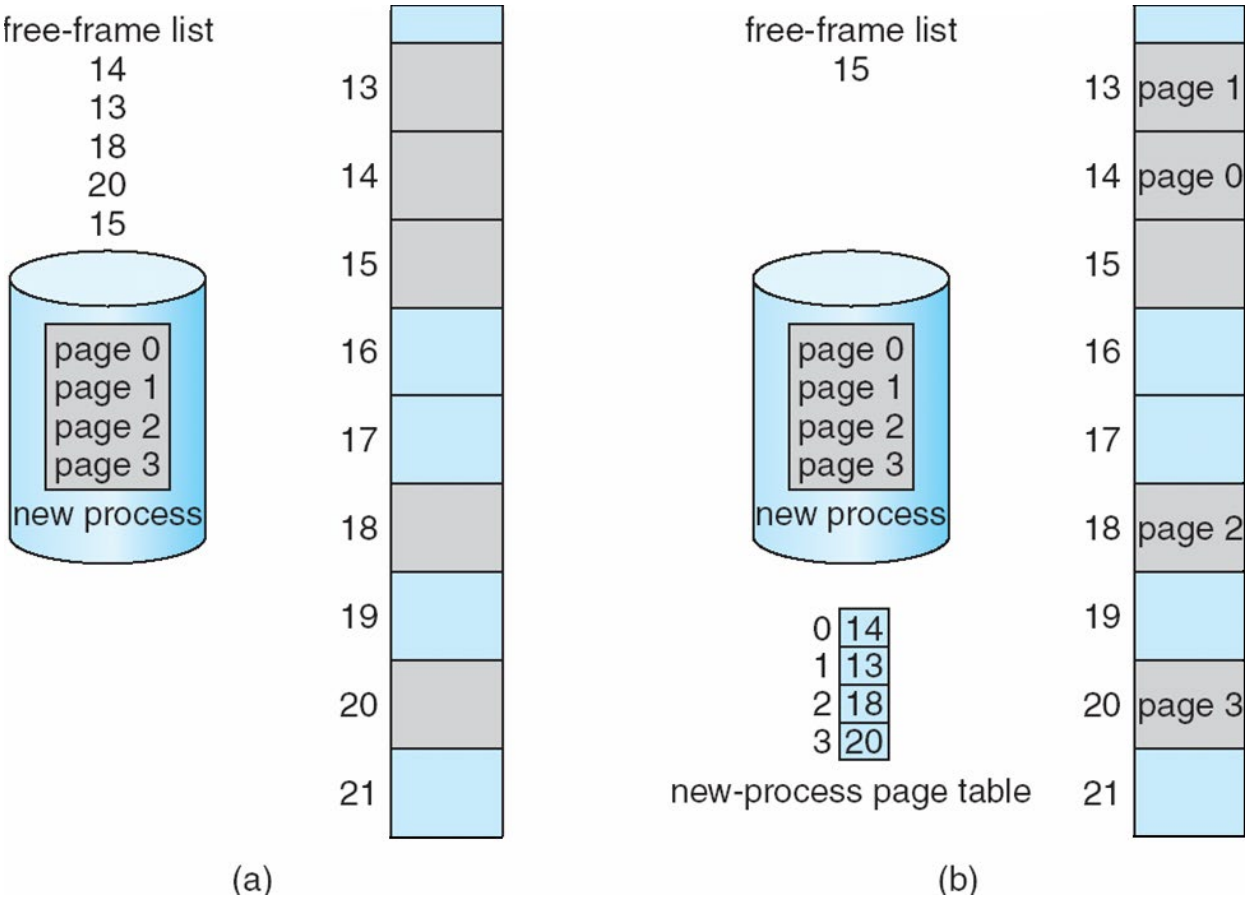


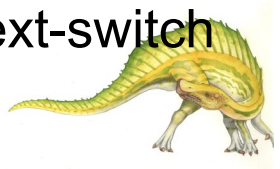
Figure 9.11 Free frames (a) before allocation and (b) after allocation.

Paging

- An important aspect of paging is the clear separation between the **programmer's view of memory** and the **actual physical memory**.
  - *The programmer views memory as one single space, containing only this one program*.
- In fact, the user program is scattered throughout physical memory, which also holds other programs.
  - The difference between the **programmer's view of memory** and the **actual physical memory** is reconciled by the ***address-translation hardware***.
  - The *logical addresses are translated into physical addresses*.
  - This mapping is hidden from the programmer and is controlled by the OS.

Paging

- Since the **OS is managing physical memory:**
  - *The **OS** determines 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 of the OS, which is called a **frame table**.
  - If a user makes a **system call** (to do I/O, for example) and provides an address as a parameter (a buffer, for instance), that address must be mapped to produce the correct **physical address**.
- The OS maintains **a copy of the page table (PT)** for each process:
  - This **PT** copy is used **to translate logical addresses to physical addresses** whenever the OS must map a logical address to a physical address manually.
  - The CPU dispatcher defines a **hardware page table** when a process is allocated the CPU. Therefore, Paging increases the context-switch time.



Implementation of Page Table

- **Page table** is kept in **main 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** and one for the **data / instruction from physical memory**
- The **two-memory access problem** can be solved using a **special fast-lookup hardware cache** called **translation look-aside buffers** (**TLBs**).

Implementation of Page Table

- If the **page number** is not in the **TLB** (known as a **TLB miss), a memory** reference to the **page table** must be made.
- Depending on the CPU, this may be done automatically in hardware or via an **interrupt to the operating system**.
- When the **frame number** is obtained, we can use it to access memory (**Figure 8.14**).
- In addition, we add the **page number** and **frame number** to the **TLB**, so that they will be found quickly on the next reference. If the TLB is already full of entries, an existing entry must be selected for **replacement**.
- **Replacement** policies range from **least recently used (LRU)** through round-robin to random.
- Some CPUs allow the operating system to participate in LRU entry replacement, while others handle the matter themselves.

Implementation of Page Table

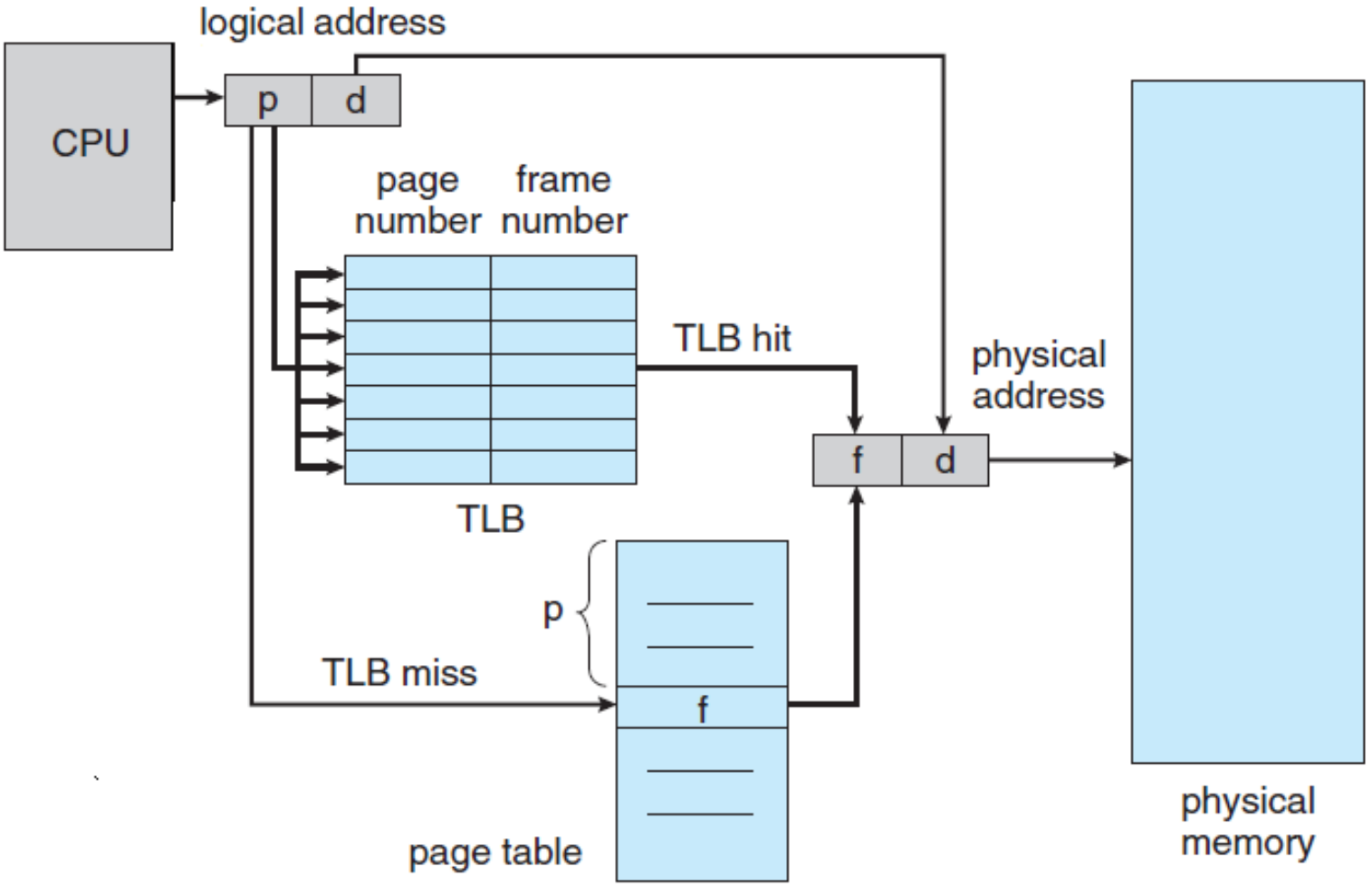
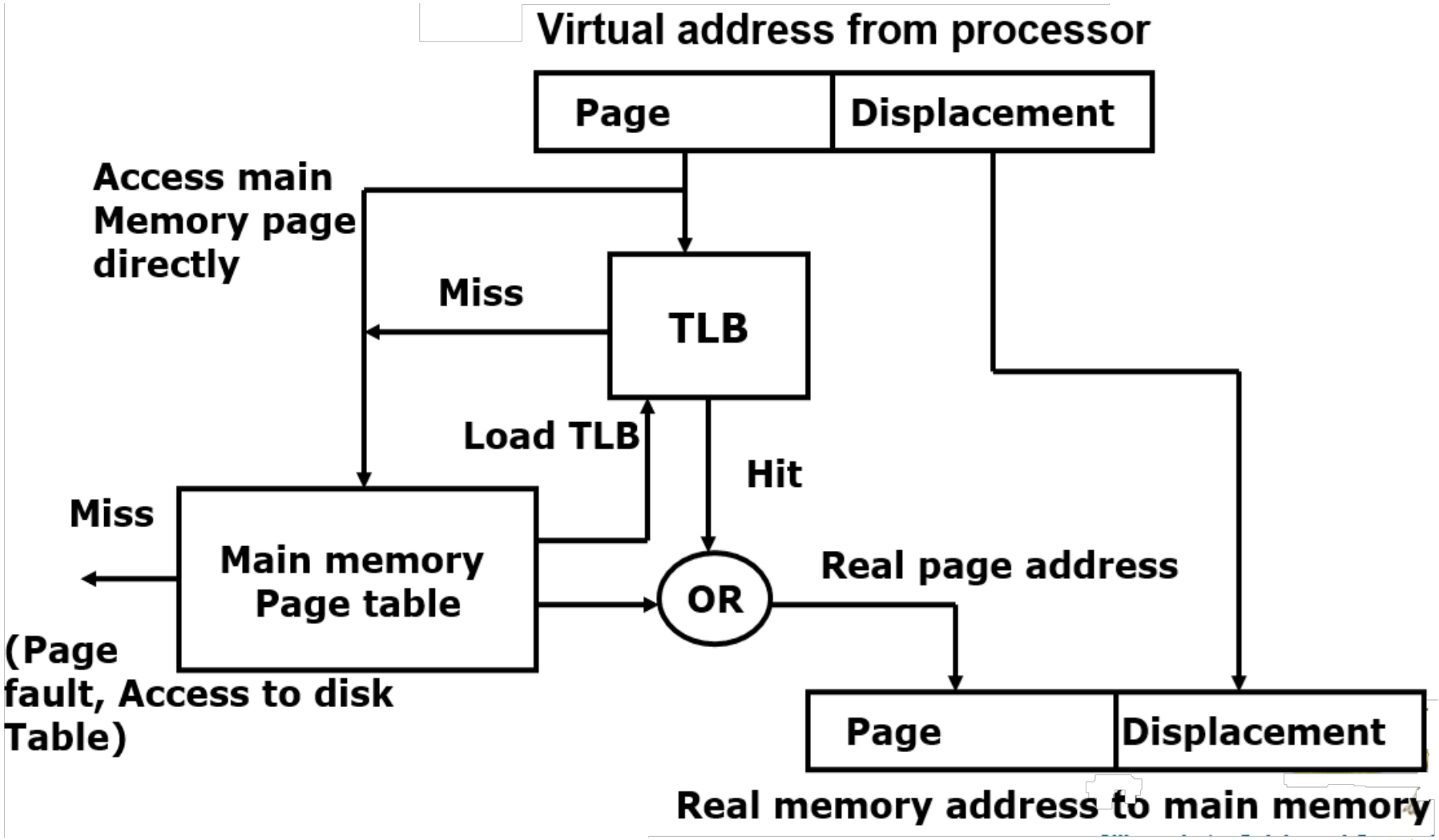
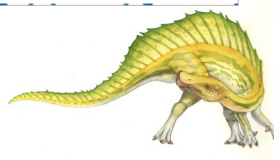


Figure 8.14 Paging hardware with TLB.

Implementation of Page Table

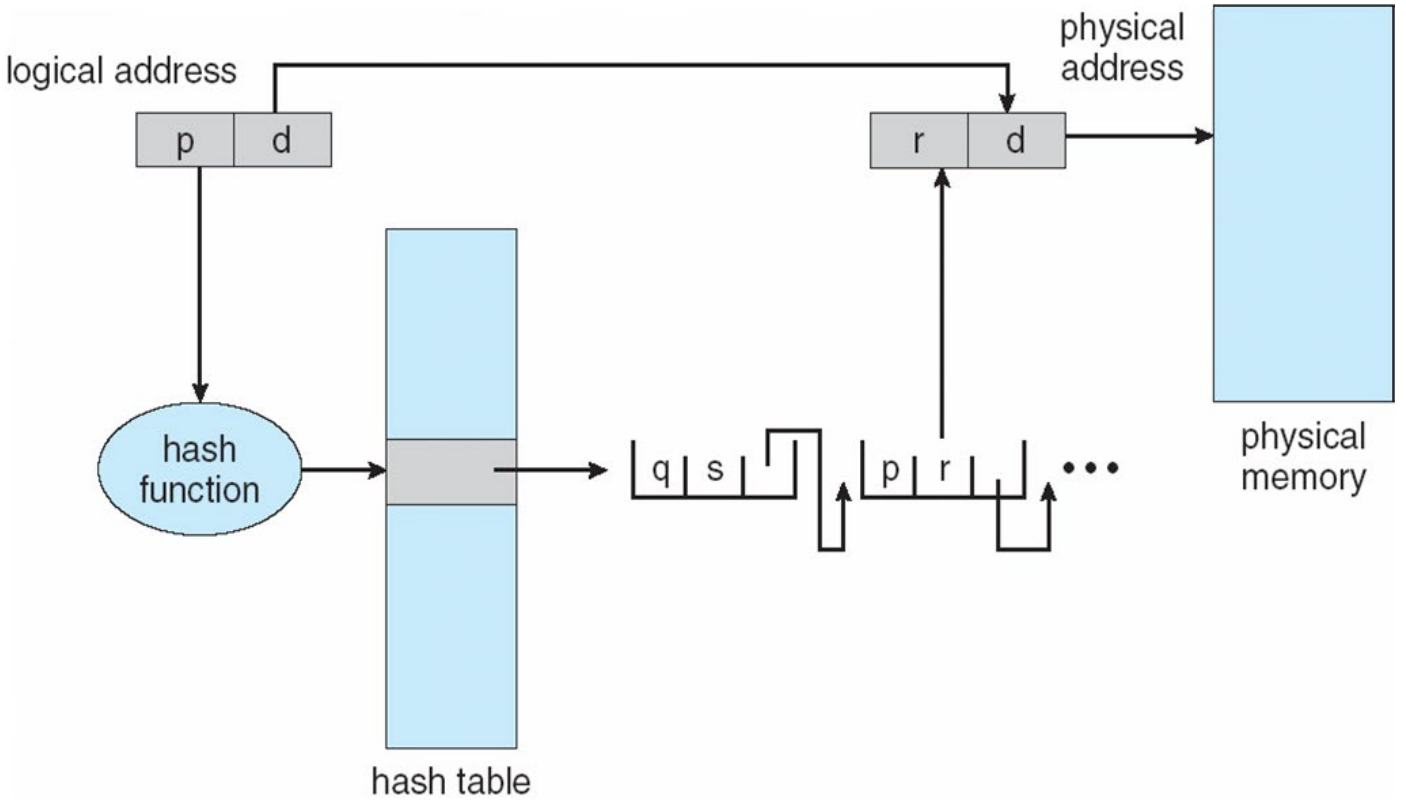


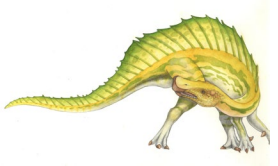


Implementation of Page Table

- 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 the desired **page number** stays in the TLB 80 percent of the time.
- If it takes **10 nanoseconds to access memory**, then a mapped-memory access takes **10 nanoseconds** when **the page number** is in the TLB.
- **If it fails to find the page number in the TLB**, then **first access memory for the page table** and **frame number** (10 nanoseconds) and then **next access the desired byte in memory** (10 nanoseconds), for a total of **20 nanoseconds**.
- **`Effective memory-access time = (hit-percent x memory_access time) + (miss-percent x page_miss_handling time)`**
- **The effective access time** = (0.80 × 10) + (0.20 × 20) = **12 ns**
- Find an effective memory access time when the hit ratio is 99%

Hashed Page Table

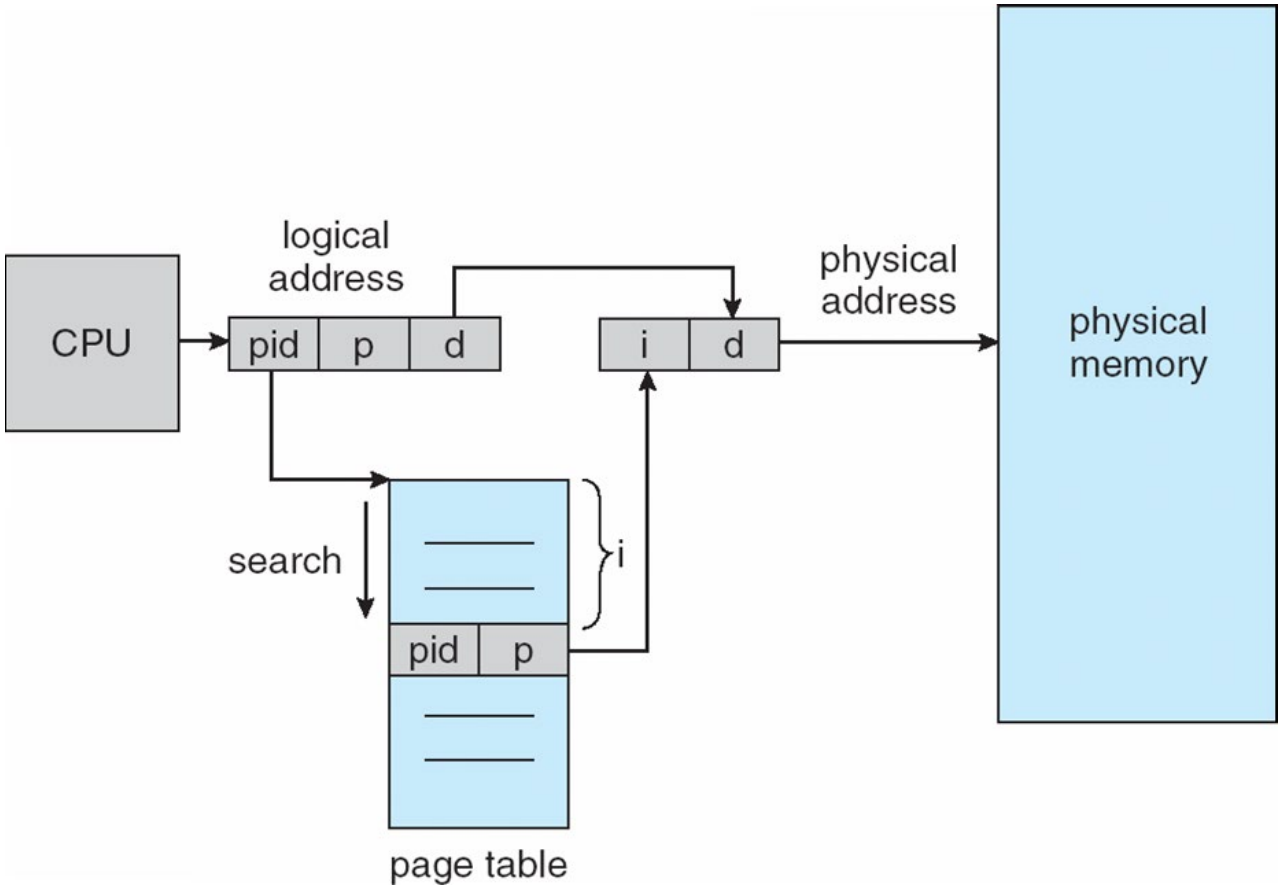




Inverted Page Table

- Rather than each process having a **page table** and keeping track of all possible **logical pages**, track all **physical pages**
- An **inverted page table** has one table entry for every page frame in real memory, resulting significant **reduction in page table size**
- 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

Inverted Page Table Architecture



Inverted Page Table Architecture

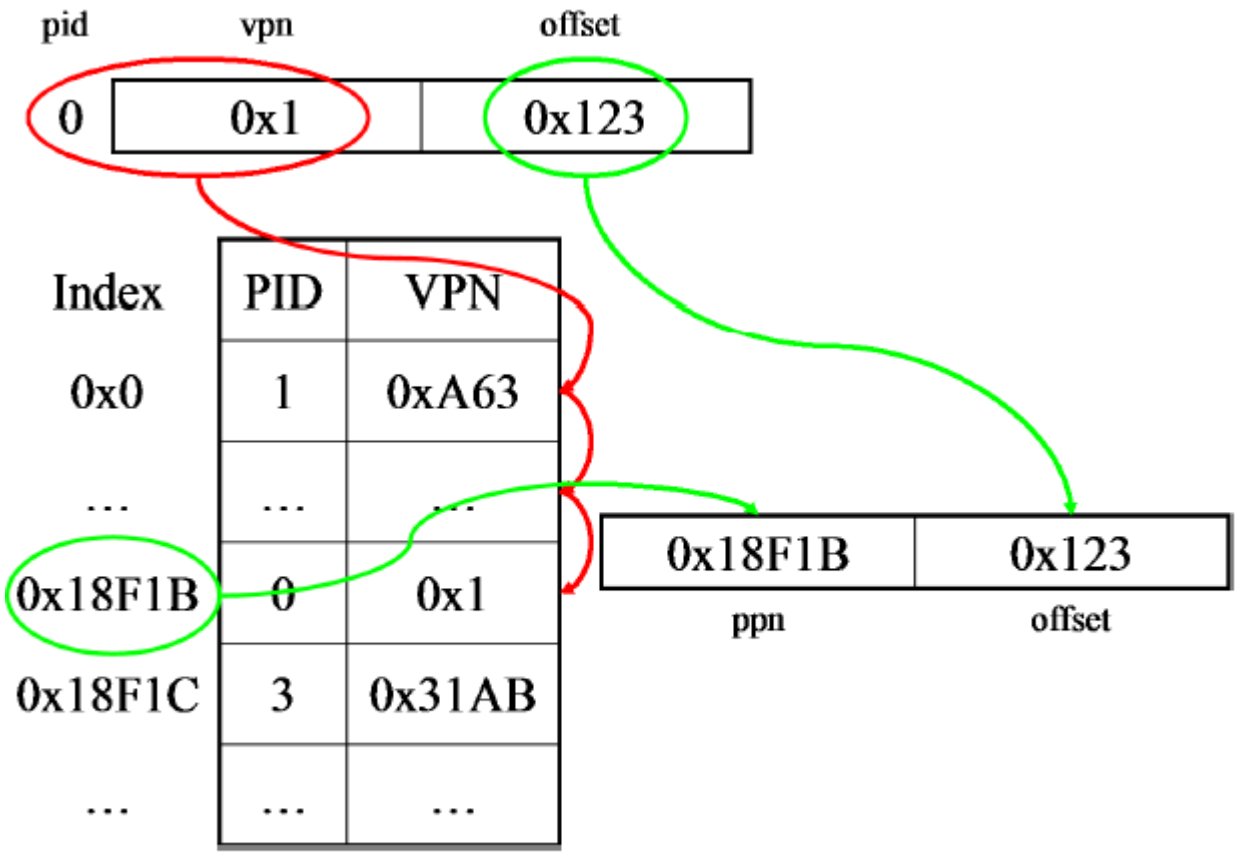


Figure: Translation procedure using a linear inverted page table. Info bits exist in each entry, though they are not shown.

Page Protection

- **Memory protection** in a paged environment is accomplished by protection bits associated with each frame.
- Normally, these bits are kept in the page table.
- **One bit** can define a page to be **read–write** or **read-only**.
- Every reference to memory goes through the **page table** to find the **correct frame number**.
- At the same time, the **physical address** is being computed, the **protection bits** can be checked to verify that no writes are being made to a read-only page.
- An attempt to write to a **read-only page** causes a **hardware trap** to the operating system (or memory-protection violation).

Page 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 using the **valid–invalid bit**.
- The OS sets this bit for each page to allow or disallow access to the page.

Page Protection

- Suppose, for example, that in 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 9.13**.
- Addresses in **pages 0, 1, 2, 3, 4, and 5** are mapped normally through the **page table**.
- Any attempt to generate an address on **pages 6 or 7**, however, will find that the **valid–invalid bit** is set to **invalid**, and the computer will **trap** to the OS (it is an **invalid page reference**).

Page Protection

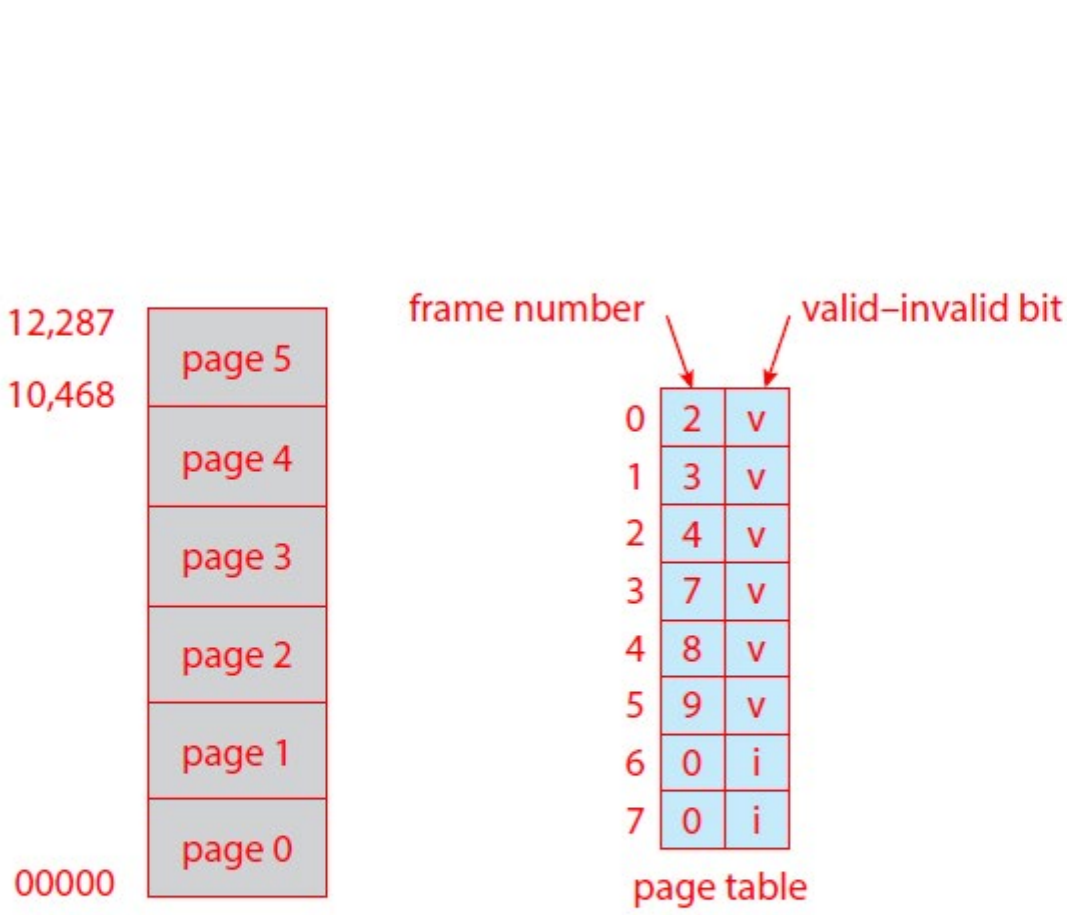


Figure 9.13 Valid (v) or invalid (i) bit in a page table.

Page Protection

- 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**.
- This problem is due to the **2-KB page size** and reflects **internal fragmentation of paging**.

Swapping

- *Instructions* and the *data* of a process must be in the main memory to be executed.
- However, a process, or a portion of a process, can be **swapped** temporarily out of memory to a **backing store** and then brought back into memory for continued execution (**Figure 9.19**).
- **Swapping** allows the total **physical address space** of all processes to exceed the real physical memory of the system, thus *increasing the degree of multiprogramming in a system*.

Swapping

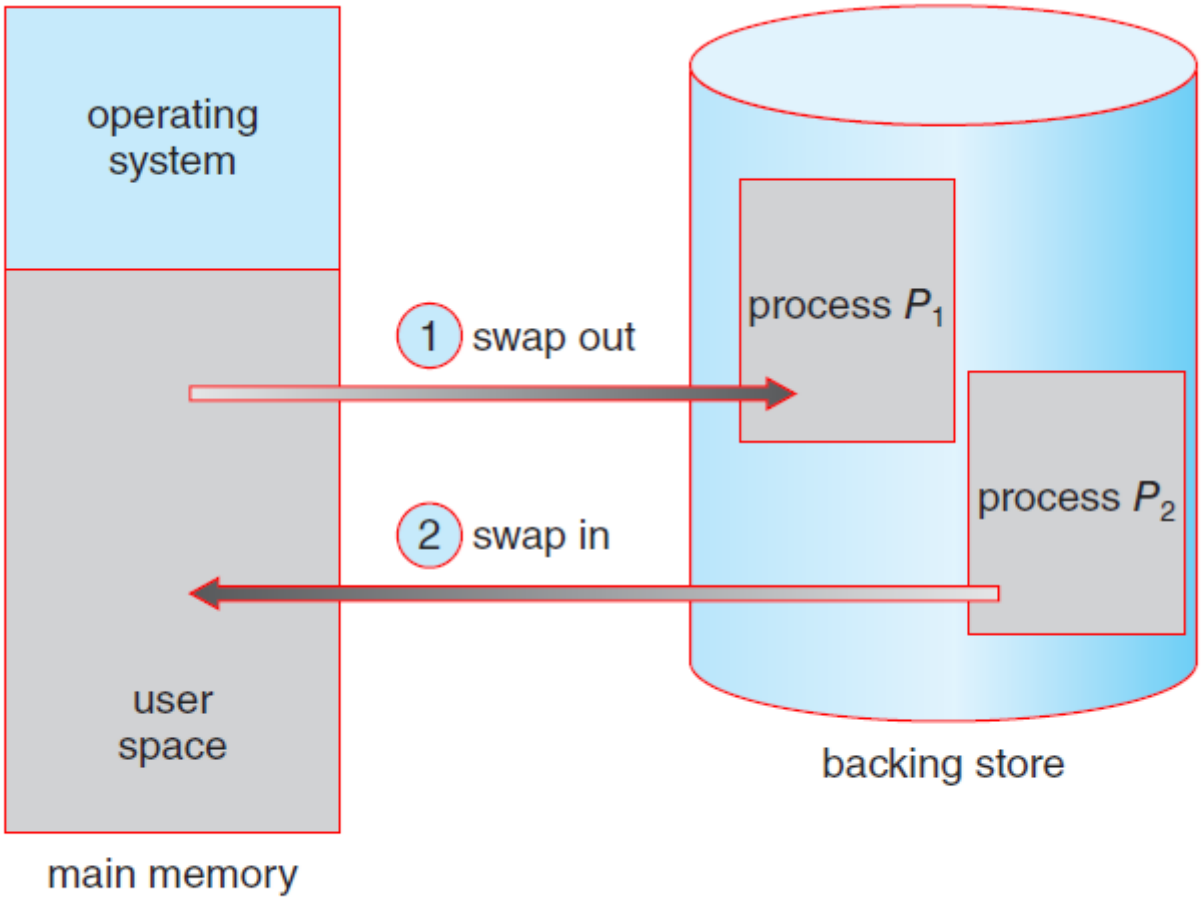


Figure 9.19 Standard swapping of two processes using a disk as a backing store.

Swapping

- **Standard swapping** was used **in traditional UNIX systems.**
  - But it is generally no longer used in contemporary operating systems, because *the amount of time required to move entire processes between memory and the backing store is prohibitive*.
- Most operating systems, including **Linux** and **Windows**, now use a variation of swapping in which **pages** of a process, rather than the entire process, are swapped.
- **In swapping with paging**, the **page-out** operation moves a page from memory to the backing store; the reverse process is known as a **page-in**.
- Swapping with paging is illustrated in **Figure 9.20,** where a subset of pages for **processes A and B** is being **paged out** and **paged in**, respectively.

Swapping

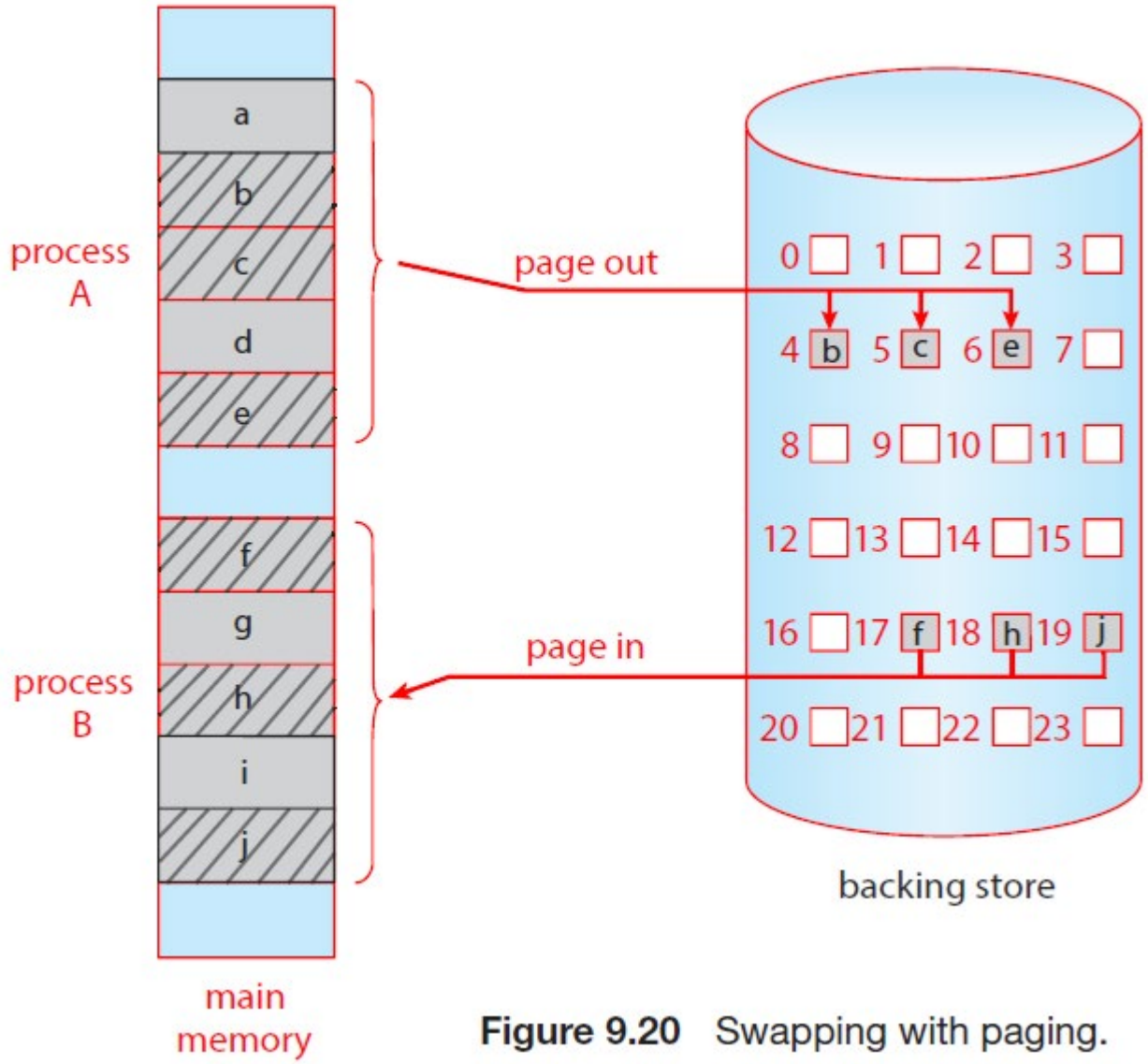


Figure 9.20 Swapping with paging.

Example: ARM Architecture

- Dominant mobile platform chip (Apple iOS and Google Android devices for example)
- Modern, energy efficient, 32-bit CPU
- 4 KB and 16 KB pages
- 1 MB and 16 MB pages (termed **sections**)
- One-level paging for sections, two-level for smaller pages
- Two levels of TLBs
  - Outer level has two micro TLBs (one data, one instruction)
  - Inner is single main TLB
  - First inner is checked, on miss outers are checked, and on miss page table walk performed by CPU