Fragmentation

- As processes are loaded and removed from memory, the free memory space is broken into little pieces.
- **External fragmentation** exists when there is enough total memory space to satisfy a request, but the available spaces are not contiguous;
- Storage is fragmented into a large number of small holes.
- This fragmentation problem can be severe.

- In the worst case, we could have a block of free (or wasted) memory between every two processes.
- If all these small pieces of memory were in one big free block ,we might be able to run several more processes.

- Memory fragmentation can be internal as well as external.
- Consider a multiple-partition allocation scheme with a hole of 18,464 bytes.
- Suppose that the next process requests 18,462 bytes.
- If we allocate exactly the requested block, we are left with a hole of 2 bytes.
- The overhead to keep track of this hole will be substantially larger than the hole itself.

- The general approach to avoiding this problem is to break the physical memory into fixed-sized blocks and allocate memory in units based on block size.
- With this approach, the memory allocated to a process may be slightly larger than the requested memory.
- The difference between these two numbers is **internal fragmentation**
- — memory that is internal to a partition but is not being used.

- One solution to the problem of external fragmentation is compaction.
- The goal is to shuffle the memory contents so as to place all free memory together in one large block.
- Compaction is possible *only if relocation is dynamic and is done at execution* time.

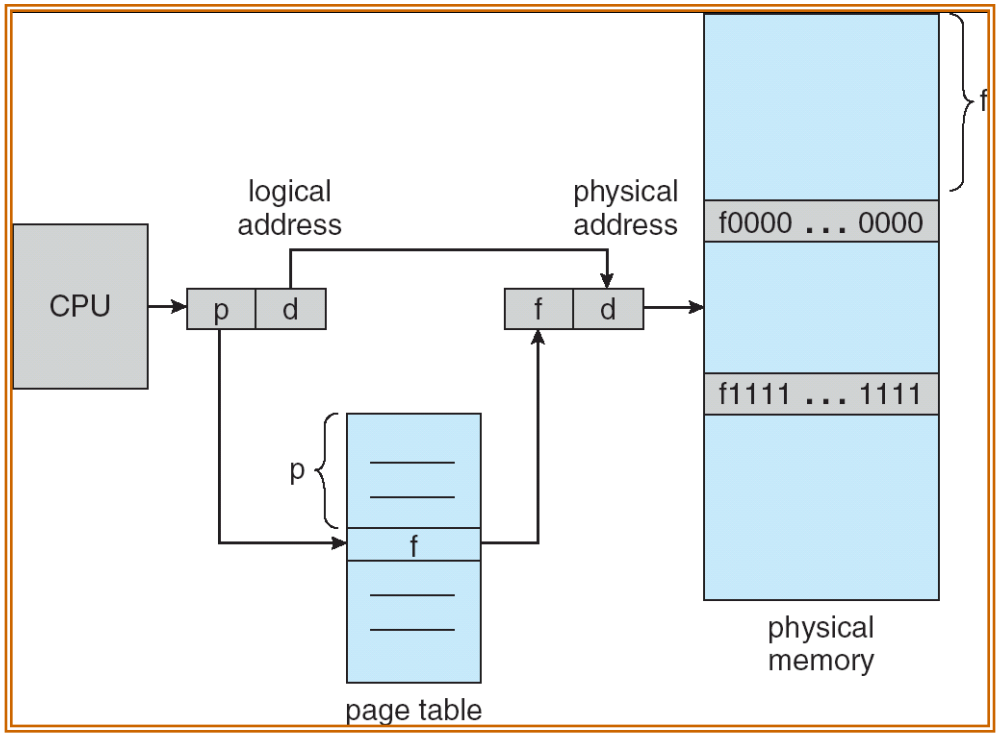
- The simplest compaction algorithm is to move all processes toward one end of memory.
- All holes move in the other direction, producing one large hole of available memory.
- This scheme can be expensive.
- Another possible solution to the external-fragmentation problem is
  - Paging
  - Segmentation

Paging

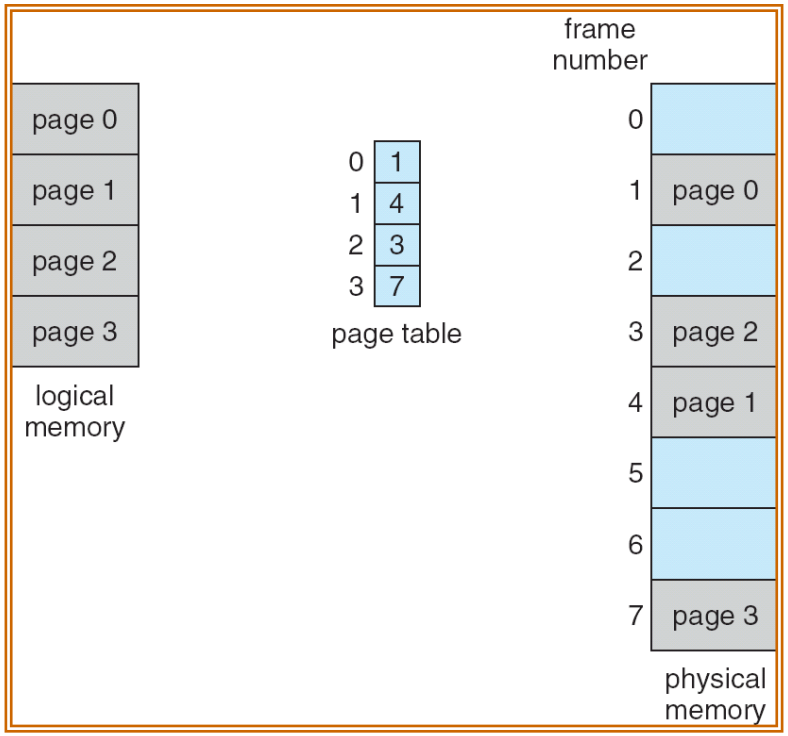
- Paging is a memory-management scheme that permits the physical address space of a process to be noncontiguous.
- The basic method for implementing paging involves breaking physical memory into fixed-sized blocks called **frames** and breaking logical memory into blocks of the same size called **pages**.
- When a process is to be executed, its pages are loaded into any available memory frames from the backing store.

- The backing store is divided into fixed-sized blocks that are of the same size as the memory frames.
- Support for paging has been handled by hardware.
- 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.

Paging Hardware



Paging Model of logical and physical memory



- The size of a page is typically a power of 2, varying between 512 bytes and 16 MB 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.
- If the size of logical address space is 2^m and a page size is 2^n addressing units (bytes or words), 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

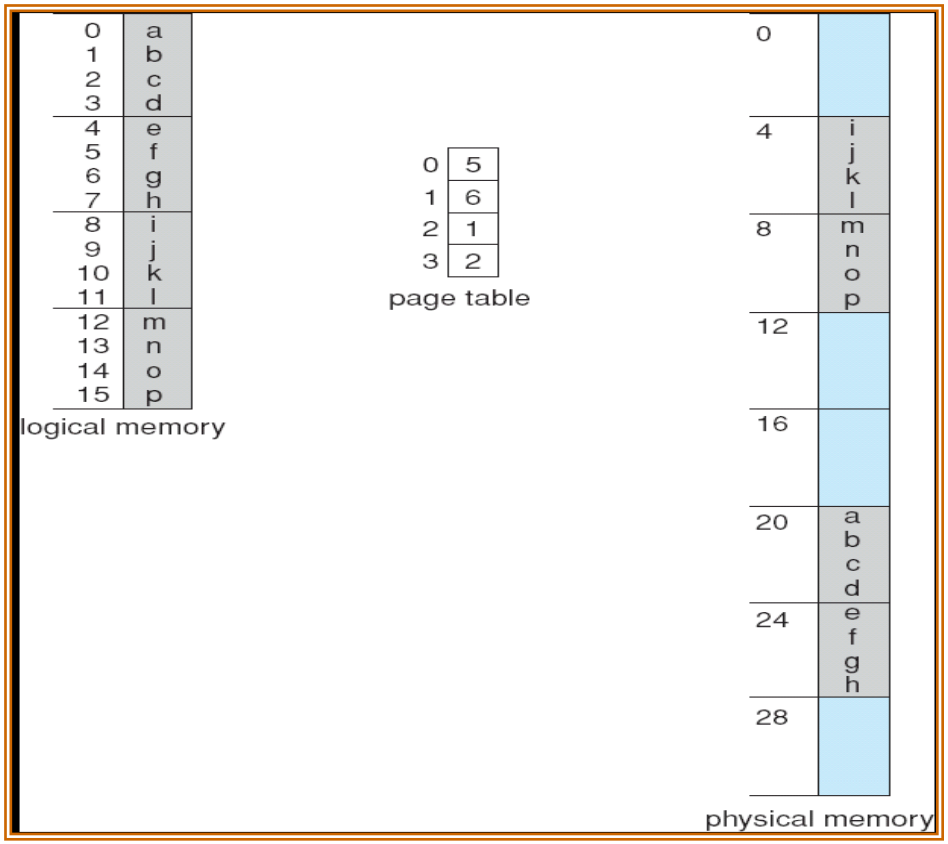
m-n n

Where p is an index into page table and d is the displacement within the page.

- Consider the following example
- Consider the memory in the following Figure .
- Using a page size of 4 bytes and a physical memory of 32 bytes (8 pages).
- The user's view of memory can be mapped into physical memory as follows.

- Logical address 0 is page 0, offset 0.
- Indexing into the page table, we find that page 0 is in frame 5.
- Thus, logical address 0 maps to physical address 20 (= (5 x 4) + 0).
- Logical address 3 (page 0, offset 3) maps to physical address 23 *{- (5x4) + 3).*
- *Logical address 4 is page 1, offset 0; according to the page* table, page 1 is mapped to frame 6.
- Thus, logical address 4 maps to physical address 24 (= (6x4) + 0). Logical address 13 maps to physical address 9.

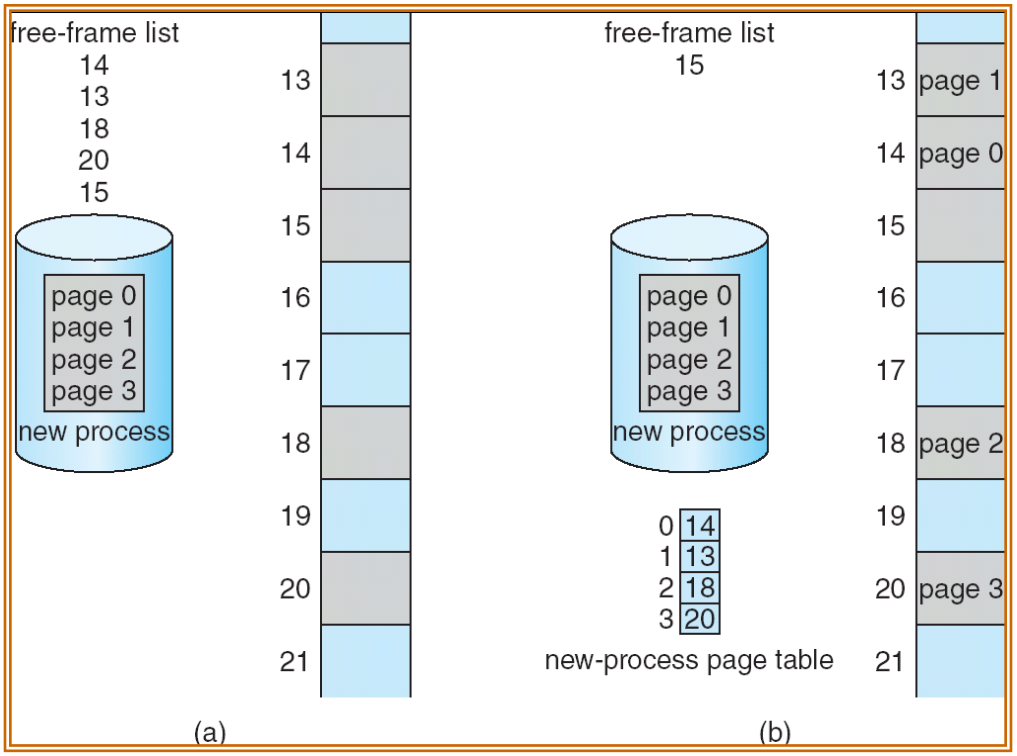
Paging Example



- When we use a paging scheme, we have no external fragmentation.
- Any free frame can be allocated to a process that needs it.
- There is a chance of internal fragmentation.

- When a process arrives in the system to be executed, its size, expressed in pages, is examined.
- Each page of the process needs one 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, and its frame number is put into the page table, and so on

Free Frames- Before and After allocation



- An important aspect of paging is the clear separation between the user's view of memory and the actual physical memory.
- The user program 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 user'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 user and is controlled by the operating system.

- Since the operating system is managing physical memory, it must be aware of the allocation details of physical memory—which frames are allocated, which frames are available, how many total frames there are, and so on.
- This information is generally kept in a data structure called a **frame table.**
- The frame table has one entry for each physical page frame, indicating whether the latter is free or allocated and, if it is allocated, to which page of which process or processes.

Hardware Support for Page tables

- Each operating system has its own methods for storing page tables.
- Most allocate a page table for each process.
- A pointer to the page table is stored in the process control block.
- The hardware implementation of the page table can be done as a set of dedicated **registers.**
- These registers should be built with very high-speed logic to make the paging-address translation efficient.

- The CPU dispatcher reloads these registers.
- The use of registers for the page table is satisfactory if the page table is reasonably small.(256 entries)
- Most of the computers, allow the page table to be very large (for example, 1 million entries).
- For these machines, the use of fast registers to implement the page table is not feasible.
- Rather, the page table is kept in main memory, and a page-table base register (PTBR) points to the page table.
- Changing page tables requires changing only this one register, substantially reducing context-switch time.

- It provides us with the frame number, which is combined with the page offset to produce the actual address.
- We can then access the desired place in memory.
- With this scheme, *two memory* accesses are needed to access a byte (one for the page-table entry, one for the byte).
- Thus, memory access is slowed by a factor of 2.

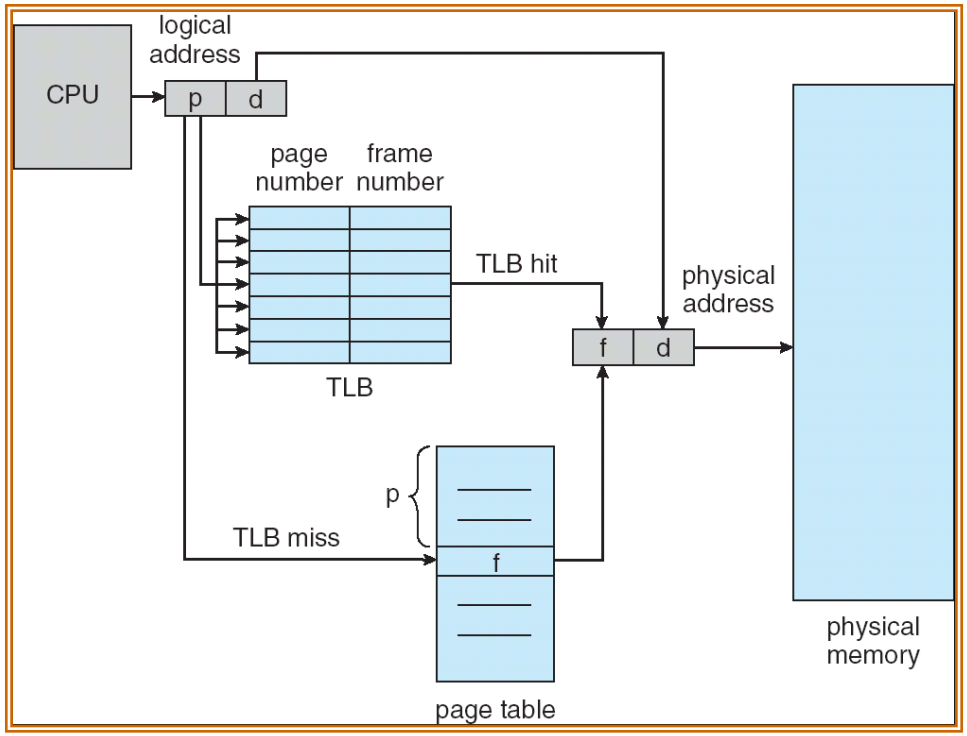
Implementation of page table

- Page table is kept in main memory
- *Page-table base register (*PTBR) points to the page table
- *Page-table length register* (PRLR) 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.
- 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 buffers (TLBs

- The TLB is associative, high-speed memory
- Each entry in the TLB consists of two parts:
- a key (or tag) and a value.
- When the associative memory is presented with an item, the item is compared with all keys simultaneously.
- If the item is found, the corresponding value field is returned.

- The TLB contains only a few of the page-table entries.
- When a logical address is generated by the CPU, its 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.


CPU
logical address
p
d
page number
frame number
TLB hit
physical address
f
d
TLB
p
TLB miss
f
physical memory
page table