Operating System

UNIT 2

Part 1 - Memory Management

Address Binding

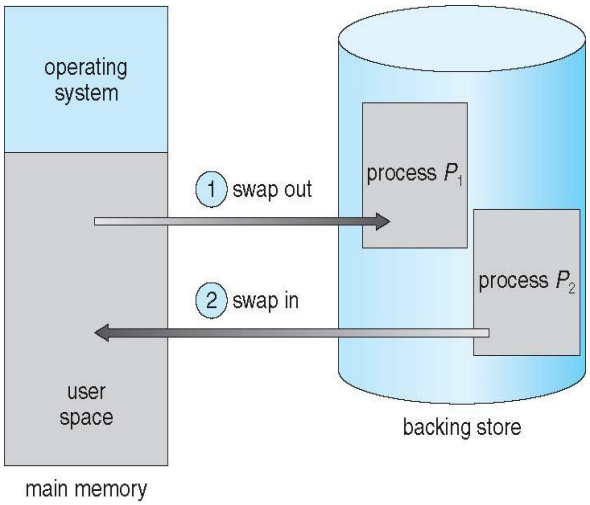
- A program resides on a disk as a **binary executable file**. To be executed, the program must be brought into memory and placed within a process.
- Depending on the memory management in use, the process may be moved between disk and memory during its execution.
- The processes on the disk that are waiting to be brought into memory for execution form the **input queue**. A user program goes through several steps before being executed.
- Addresses may be represented in different ways during these steps. Addresses in the source program are generally symbolic.
- A compiler typically binds these **symbolic addresses** to **relocatable addresses**. The linkage editor or loader in turn binds the relocatable addresses to absolute addresses.
- The binding of instructions and data to memory addresses can be done at any step along the way:
    a. **Compile time**: If you know at compile time where the process will reside in memory, then absolute code can be generated. If, at some later time, the starting location changes, then it will be necessary to recompile this code.
    b. **Load time**: If it is not known at compile time where the process will reside in memory, then the compiler must generate relocatable code. In this case, final binding is delayed until load time.
    c. **Execution time**: If the process can be moved during its execution from one memory segment to another, then binding must be delayed until run time.

Logical Versus Physical Address Space

- An address generated by the CPU is commonly referred to as a **logical address**, whereas an address seen by the memory unit—that is, the one loaded into the memory-address register of the memory—is commonly referred to as a **physical address.**
- The compile-time and load-time address-binding methods generate identical logical and physical addresses. However, the execution-time address binding scheme results in differing logical and physical addresses. In this case, we usually refer to the logical address as a **virtual address**.
- The set of all logical addresses generated by a program is a **logical address space**.
- The set of all physical addresses corresponding to these logical addresses is a physical address space. Thus, in the execution-time address-binding scheme, the logical and physical address spaces differ.
- The run-time mapping from virtual to physical addresses is done by a hardware device called the **memory-management unit** (MMU).

Swapping

- A process must be in memory to be executed. A process, however, can be swapped temporarily out of memory to a backing store and then brought back into memory for continued execution
- Standard swapping involves moving processes between main memory and a backing store.
- The backing store is commonly a fast disk. It must be large enough to accommodate copies of all memory images for all users, and it must provide direct access to these memory images.
- The system maintains a **ready queue** consisting of all processes whose memory images are on the backing store or in memory and are ready to run.
- Whenever the CPU scheduler decides to execute a process, it calls the **dispatcher**. The dispatcher checks to see whether the next process in the queue is in memory.
- If it is not, and if there is no free memory region, the dispatcher swaps out a process currently in memory and swaps in the desired process.
- It then reloads registers and transfers control to the selected process.



Contiguous Memory Allocation

- The main memory must accommodate both the operating system and the various user processes.
- The memory is usually divided into two partitions: one for the resident operating system and one for the user processes. We can place the operating system in either **low memory** or **high memory**. The major factor affecting this decision is the location of the interrupt vector. Since the interrupt vector is often in low memory, programmers usually place the operating system in low memory as well.
- We usually want several user processes to reside in memory at the same time. We therefore need to consider how to allocate available memory to the processes that are in the input queue waiting to be brought into memory.

- In contiguous memory allocation, each process is contained in a single section of memory that is contiguous to the section containing the next process.

Memory Protection

- The relocation register contains the value of the smallest physical address; the limit register contains the range of logical addresses .Each logical address must fall within the range specified by the **limit register**.
- The MMU maps the logical address dynamically by adding the value in the relocation register. This **mapped address** is sent to memory.
- When the CPU scheduler selects a process for execution, the dispatcher loads the relocation and limit registers with the correct values as part of the context switch.
- Because every address generated by a CPU is checked against these registers, we can protect both the operating system and the other users' programs and data from being modified by this running process.

Memory Allocation

- One of the simplest methods for allocating memory is to divide memory into several fixed-sized partitions. Each partition may contain exactly one process.
- In this multiple partition method, when a partition is free, a process is selected from the input queue and is loaded into the free partition. When the process terminates, the partition becomes available for another process.
- In the variable-partition scheme, the operating system keeps a table indicating which parts of memory are available and which are occupied.
- Initially, all memory is available for user processes and is considered one large block of available memory, a hole.
- The operating system takes into account the memory requirements of each process and the amount of available memory space in determining which processes are allocated memory.
- When a process is allocated space, it is loaded into memory, and it can then compete for CPU time. When a process terminates, it releases its memory, which the operating system may then fill with another process from the input queue.
- At any given time, then, we have a list of available block sizes and an input queue.
- When a process arrives and needs memory, the system searches the set for a hole that is large enough for this process. If the hole is too large, it is split into two parts. One part is allocated to the arriving process; the other is returned to the set of holes.
- When a process terminates, it releases its block of memory, which is then placed back in the set of holes. If the new hole is adjacent to other holes, these adjacent holes are merged to form one larger hole.
- This procedure is a particular instance of the general dynamic storage allocation problem, which concerns how to satisfy a request of size *n* from a list of free holes.

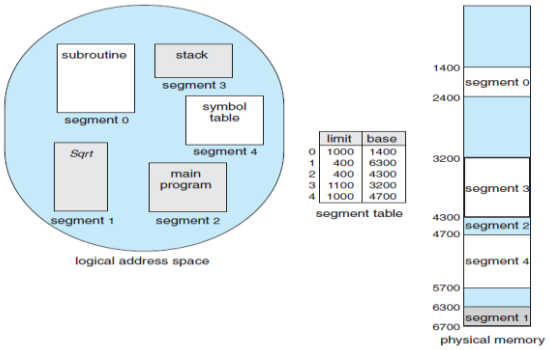
- The first-fit, best-fit, and worst-fit strategies are the ones most commonly used to select a free hole from the set of available holes.
    - **First fit:** Allocate the first hole that is big enough. Searching can start either at the beginning of the set of holes or at the location where the previous first-fit search ended. We can stop searching as soon as we find a free hole that is large enough.
    - **Best fit:** Allocate the smallest hole that is big enough. We must search the entire list, unless the list is ordered by size. This strategy produces the smallest leftover hole.
    - **Worst fit:** Allocate the largest hole. Again, we must search the entire list, unless it is sorted by size. This strategy produces the largest leftover hole, which may be more useful than the smaller leftover hole from a best-fit approach.

Fragmentation

- Both the first-fit and best-fit strategies for memory allocation suffer from external 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.
- If all these small pieces of memory were in one big free block instead, we might be able to run several more processes.
- Even if we use the first-fit or best-fit strategy it can affect the amount of fragmentation.
- Memory fragmentation can be **internal** as well as external. The memory allocated to a process may be slightly larger than the requested memory. The difference between these two numbers is **internal fragmentation**—unused memory that is internal to a partition.
- Another possible solution to the external-fragmentation problem is to permit the logical address space of the processes to be **noncontiguous**, thus allowing a process to be allocated physical memory wherever such memory is available.

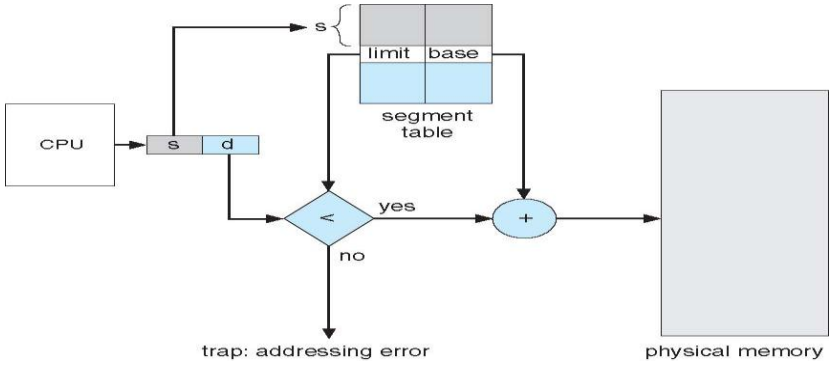
Segmentation

- Segmentation is a memory-management scheme that supports the programmer view of memory. A **logical address** space is a collection of segments.
- Each segment has a name 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.
- Segments are numbered and are referred to by a **segment number**, rather than by a **segment name.**
- Thus, a logical address consists of a *two tuple:* <segment-number, offset>.



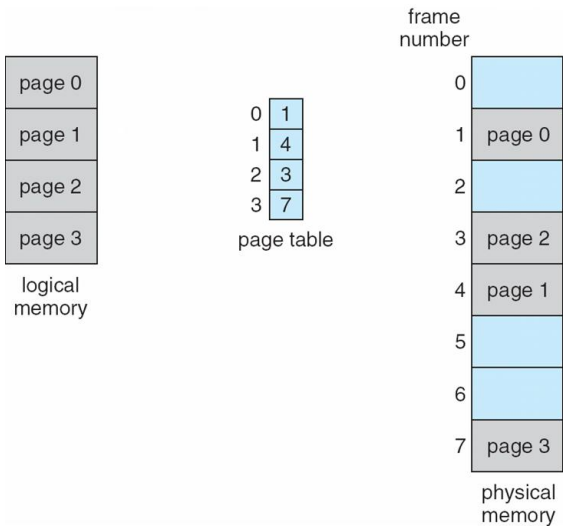
Segmentation Architecture

- We must define an implementation to map two-dimensional user-defined addresses into one-dimensional physical addresses. This mapping is done 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.
- A logical address consists of two parts: a segment number, *s*, and an offset into that segment, *d*. The segment number is used as an index to the segment table. The offset *d* of the logical address must be between 0 and the segment limit.
- If it is not, we trap to the operating system (logical addressing attempt beyond end of segment). When an offset is legal, it is added to the segment base to produce the address in physical memory of the desired byte.
- The segment table is thus essentially an array of base–limit register pairs.



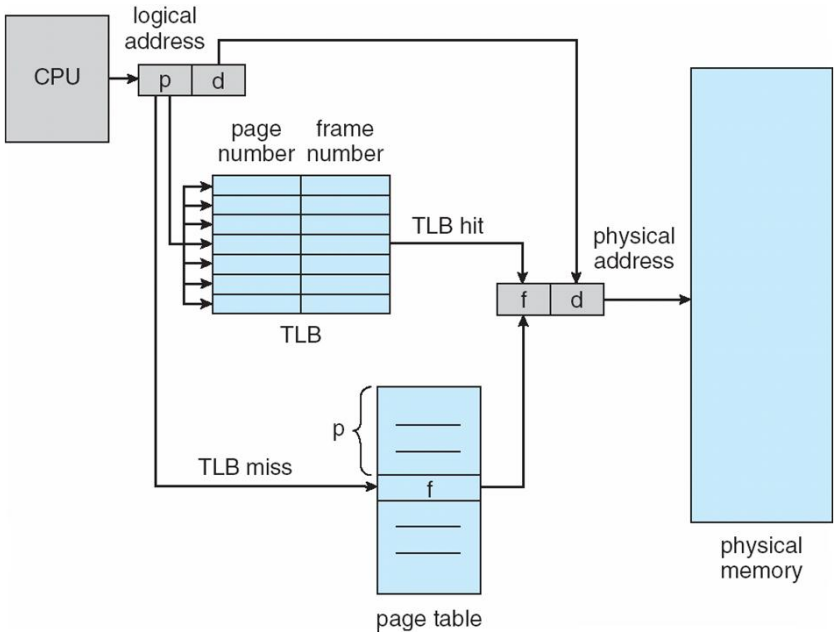
Paging

- Segmentation permits the physical address space of a process to be noncontiguous. Paging is another memory-management scheme that offers this advantage.
- 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 (a file system or the backing store). The backing store is divided into fixed-sized blocks that are the same size as the memory frames or clusters of multiple frames.
- 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 page size (like the frame size) is defined by the hardware. The size of a page is a power of 2, varying between 512 bytes and 1 GB per page, depending on the computer architecture.
- 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; its frame number is put into the page table, and so on.
- 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.



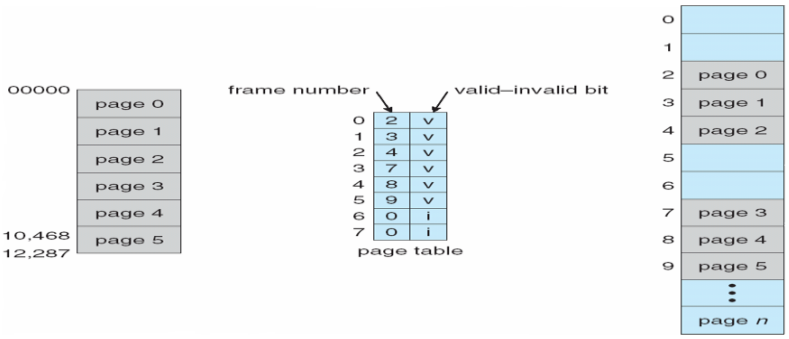
Hardware Support

- Each operating system has its own methods for storing page tables. Some allocate a **page table** for each process. A pointer to the page table is stored with the other register values (like the instruction counter) in the process control block.
- The hardware implementation of the page table can be done in several ways. In the simplest case, the page table is implemented as a set of dedicated registers. These registers should be built with very high-speed logic to make the paging-address translation efficient.
- **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. The problem with this approach is the time required to access a user memory location.
- The standard solution to this problem is to use a special, small, fast lookup hardware cache called a **translation look-aside buffer** (TLB).
- 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 is used with page tables in the following way.
- 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 **(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.



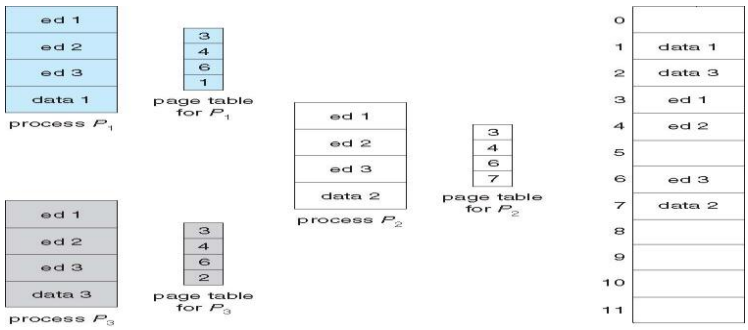
Protection

- 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 that 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. 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.



Shared Pages

- An advantage of paging is the possibility of *sharing* common code. **Reentrant code** is non-self-modifying code: it never changes during execution. Thus, two or more processes can execute the same code at the same time.
- Each process has its own copy of registers and data storage to hold the data for the process's execution. The data for two different processes will, of course, be different. Only one copy of the editor need be kept in physical memory.
- Each user's page table maps onto the same physical copy of the editor, but data pages are mapped onto different frames.

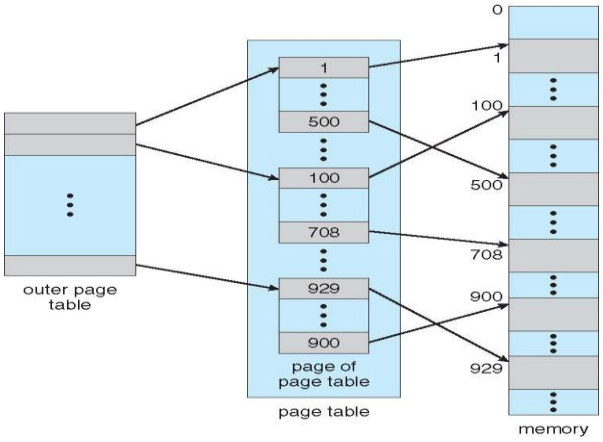


Structure of the Page Table

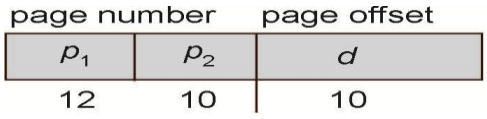
Some of the most common techniques for structuring the page table, including hierarchical paging, hashed page tables, and inverted page tables.

Hierarchical Paging

- Most modern computer systems support a large logical address space (232 to 264). In such an environment, the page table itself becomes excessively large.
- We use a two-level paging algorithm, in which the **page table itself is also paged**.
- For example, consider again the system with a 32-bit logical address space and a page size of 4 KB. A logical address is divided into a page number consisting of 20 bits and a page offset consisting of 12 bits.



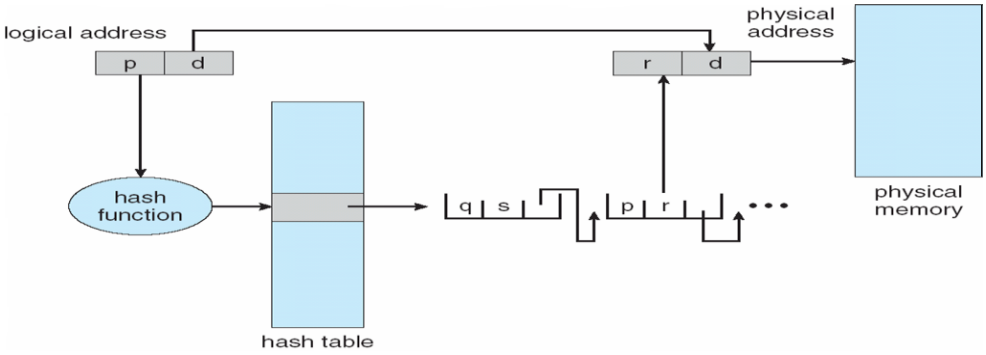
- Because we page the page table, the page number is further divided into a 10-bit page number and a 10-bit page offset. Thus, a logical address is as follows: where *p*1 is an index into the outer page table and *p*2 is the displacement within the page of the inner page table.



Because address translation works from the outer page table inward, this scheme is also known as a **forward-mapped page table**.

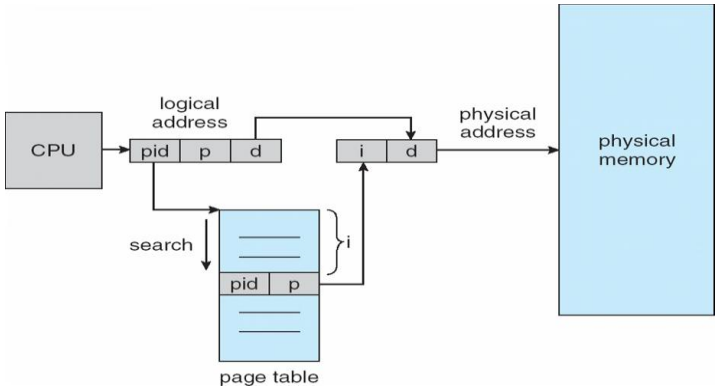
Hashed Page Tables

- A common approach for handling address spaces larger than 32 bits is to use a **hashed page table**, with the hash value being the virtual page number.
- Each entry in the hash table contains a linked list of elements that hash to the same location (to handle collisions). Each element consists of three fields: (1) the virtual page number, (2) the value of the mapped page frame, and (3) a pointer to the next element in the linked list.
- The algorithm works as follows: The virtual page number in the virtual address is hashed into the hash table. The virtual page number is compared with field 1 in the first element in the linked list.
- If there is a match, the corresponding page frame (field 2) is used to form the desired physical address. If there is no match, subsequent entries in the linked list are searched for a matching virtual page number.



Inverted Page Tables

- An inverted page table has one entry for each real page (or frame) of memory. Each entry consists of the virtual address of the page stored in that real memory location; with information about the process that owns the page.
- Thus, only one page table is in the system, and it has only one entry for each page of physical memory.
- Inverted page tables often require that an address-space identifier be stored in each entry of the page table, since the table usually contains several different address spaces mapping physical memory.
- Storing the address-space identifier ensures that a logical page for a particular process is mapped to the corresponding physical page frame.

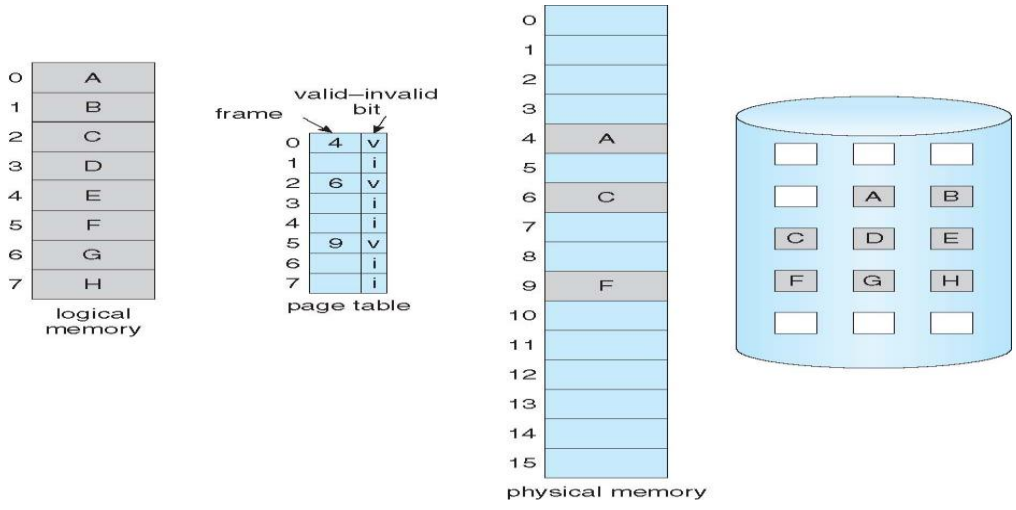


Virtual Memory

- Virtual memory is a technique that allows the execution of processes that are not completely in memory. One major advantage of this scheme is that programs can be larger than physical memory.
- **Virtual memory** involves the separation of logical memory as perceived by users from physical memory. This separation allows an extremely large virtual memory to be provided for programmers when only a smaller physical memory is available.
- The **virtual address space** of a process refers to the logical (or virtual) view of how a process is stored in memory.

Demand Paging

- When a process is to be swapped in, the pager guesses which pages will be used before the process is swapped out again. Instead of swapping in a whole process, the pager brings only those pages into memory.
- Thus, it avoids reading into memory pages that will not be used anyway, decreasing the swap time and the amount of physical memory needed. We need some form of hardware support to distinguish between the pages that are in memory and the pages that are on the disk.
- The valid–invalid bit scheme can be used for this purpose. This time, however, when this bit is set to "valid," the associated page is both legal and in memory.
- If the bit is set to "invalid," the page either is not valid (that is, not in the logical address space of the process) or is valid but is currently on the disk.
- The page-table entry for a page that is brought into memory is set as usual, but the page-table entry for a page that is not currently in memory is either simply marked invalid or contains the address of the page on disk.



Page Replacement

We modify the page-fault service routine to include page replacement:

a. Find the location of the desired page on the disk.
b. Find a free frame:
    i. If there is a free frame, use it.
    ii. If there is no free frame, use a page-replacement algorithm to select a victim frame.
    iii. Write the victim frame to the disk; change the page and frame tables accordingly.
c. Read the desired page into the newly freed frame; change the page and frame tables.
d. Continue the user process from where the page fault occurred.

If no frames are free, *two* page transfers (one out and one in) are required. This situation effectively doubles the page-fault service time and increases the effective access time accordingly.

NOTE: For Page Replacement Algorithms study all the examples during classwork.

Thrashing

- If the number of frames allocated to a low-priority process falls below the minimum number required by the computer architecture, we must suspend that process's execution.
- We should then page out its remaining pages, freeing all its allocated frames. This provision introduces a swap-in, swap-out level of intermediate CPU scheduling.
- Any process that does not have "enough" frames. If the process does not have the number of frames it needs to support pages in active use, it will quickly page-fault.
- At this point, it must replace some page. However, since all its pages are in active use, it must replace a page that will be needed again right away.
- Consequently, it quickly faults again, and again, and again, replacing pages that it must bring back in immediately.
- This high paging activity is called **thrashing**. A process is thrashing if it is spending more time paging than executing.