**UNIT III**

**STORAGE MANAGEMENT**

**MAIN MEMORY:**

Memory is central to the operation of a modern computer system. Memory consists of a large array of bytes, each with its own address. The CPU fetches instructions from memory according to the value of the program counter. These instructions may cause additional loading from and storing to specific memory addresses.

A typical instruction-execution cycle, for example, first fetches an instruction from memory. The instruction is then decoded and may cause operands to be fetched from memory. After the instruction has been executed on the operands, results may be stored back in memory. The memory unit sees only a stream of memory addresses; it does not know how they are generated or what they are for.

**Basic Hardware:**

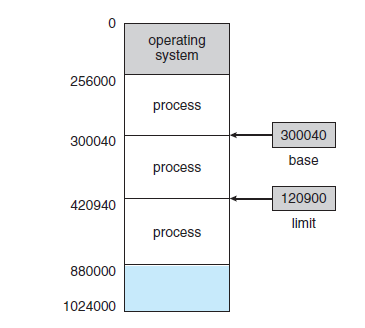
CPU can access directly from Main memory and the registers built into the processor. Therefore, any instructions in execution, and any data being used by the instructions, must be in one of these direct-access storage devices. If the data are not in memory, they must be moved there before the CPU can operate on them.

Registers that are built into the CPU are generally accessible within one cycle of the CPU clock. But Main memory, which is accessed via memory bus may take many cycles of the CPU clock. In such cases, the processor normally needs to stall, since it does not have the data required to complete the instruction that it is executing. This situation is intolerable because of the frequency of memory accesses. The remedy is to add fast memory between the CPU and main memory called cache.

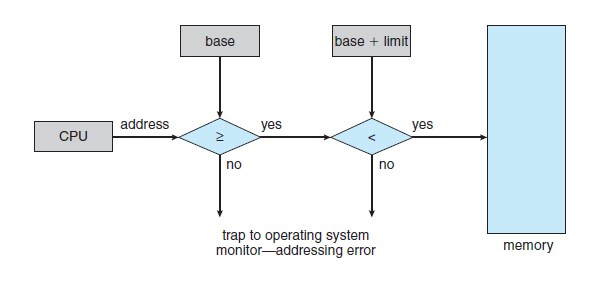
For proper system operation we must protect the operating system from access by user processes.

On multiuser systems, we must additionally protect user processes from one another. This protection must be provided by the hardware. Each process has a separate memory space and separate per-process memory space protects the processes from each other and is fundamental to having multiple processes loaded in memory for concurrent execution. To separate memory spaces, we need to determine the range of legal addresses that the process may access and to ensure that the process can access only these legal addresses. This protection is provided by using two registers, a **base** and a **limit registers.**

The **base register** holds the smallest legal physical memory address; the **limit register** specifies the size of the range. For example, if the base register holds 300040 and the limit register is 120900, then the program can legally access all addresses from 300040 through 420939



Protection of memory space is accomplished by having the CPU hardware compare every address generated in user mode with the registers. Any attempt by a program executing in user mode to access operating-system memory or other users’ memory results in a trap to the operating system.



**Hardware address protection with base and limit registers.**

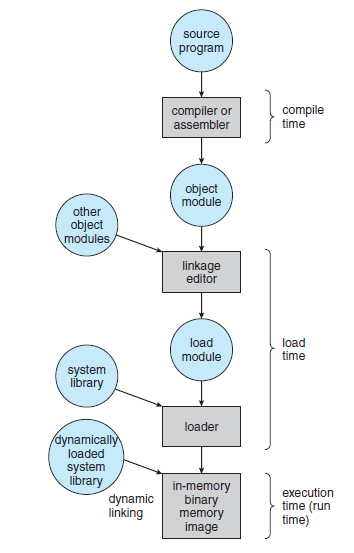
**Address Binding:**

Usually, 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. Most systems allow a user process to reside in any part of the physical memory. Thus, although the address space of the computer may start at 00000, the first address of the user process need not be 00000.

Addresses may be represented in different ways. Addresses in the source program are generally symbolic (such as the variable count). 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:

* **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. The MS-DOS .COM-format programs are bound at compile time.
* **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. If the starting address changes, we need only reload the user code to incorporate this changed value.
* **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.

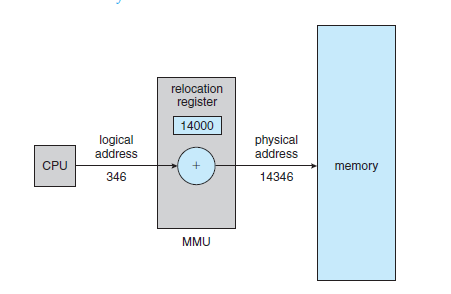


**Logical Versus Physical Address Space:**

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. 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).



The base register is now called a relocation register. The value in the relocation register is added to every address generated by a user process at the time the address is sent to memory (see Figure). For example, if the base is at 14000, then an attempt by the user to address location 0 is dynamically relocated to location 14000; an access to location 346 is mapped to location 14346.

**Dynamic Loading:**

To obtain better memory-space utilization, we can use dynamic loading. With dynamic loading, a routine is not loaded until it is called. All routines are kept on disk in a relocatable load format. The main program is loaded into memory and is executed. When a routine needs to call another routine, the calling routine first checks to see whether the other routine has been loaded. If it has not, the relocatable linking loader is called to load the desired routine into memory and to update the program’s address tables to reflect this change. Then control is passed to the newly loaded routine.

The advantage of dynamic loading is that a routine is loaded only when it is needed.

**Dynamic Linking and Shared Libraries:**

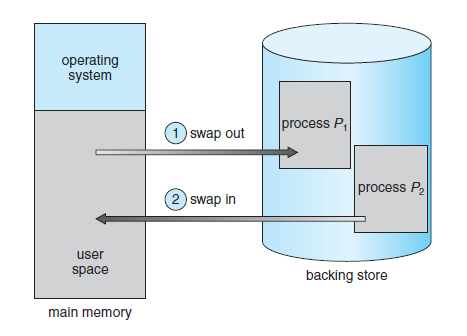
Dynamically linked libraries are system libraries that are linked to user programs when the programs are run. With dynamic linking, a stub is included in the image for each library routine reference. The stub is a small piece of code that indicates how to locate the appropriate memory-resident library routine or how to load the library if the routine is not already present. When the stub is executed, it checks to see whether the needed routine is already in memory. If it is not, the program loads the routine into memory. Either way, the stub replaces itself with the address of the routine and executes the routine.

**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. Swapping makes it possible for 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.

**Standard Swapping:**

Standard swapping involves moving processes between main memory and a backing store. The backing store is commonly a fast disk.



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.

The context-switch time in such a swapping system is fairly high. The actual transfer of the 100-MB process to or from main memory takes

100 MB/50 MB per second = 2 seconds

The swap time is 200 milliseconds. Since we must swap both out and in, the total swap time is about 4,000 milliseconds. Notice that the major part of the swap time is transfer time. The total transfer time is directly proportional to the amount of memory swapped.

Clearly, it would be useful to know exactly how much memory a user process is using, not simply how much it might be using. Then we would need to swap only what is actually used, reducing swap time. For this method to be effective, the user must keep the system informed of any changes in memory requirements.

Swapping is constrained by other factors as well. A process may be waiting for an I/O operation when we want to swap that process to free up memory. However, if the I/O is asynchronously accessing the user memory for I/O buffers, then the process cannot be swapped. Assume that the I/O operation is queued because the device is busy. If we were to swap out process *P*1 and swap in process *P*2, the I/O operation might then attempt to use memory that now belongs to process *P*2. There are two main solutions to this problem: never swap a process with pending I/O, or execute I/O operations only into operating-system buffers.

**Swapping on Mobile Systems:**

Mobile devices generally use flash memory rather than more spacious hard disks as their persistent storage. The resulting space constraint is one reason why mobile operating-system designers avoid swapping. Other reasons include the limited number of writes that flash memory can tolerate before it becomes unreliable and the poor throughput between main memory and flash memory in these devices.

Instead of using swapping, when free memory falls below a certain threshold, Apple’s iOS asks applications to voluntarily relinquish allocated memory.

Android does not support swapping and adopts a strategy similar to that used by iOS. It may terminate a process if insufficient free memory is available. However, before terminating a process, Android writes its application state to flash memory so that it can be quickly restarted.

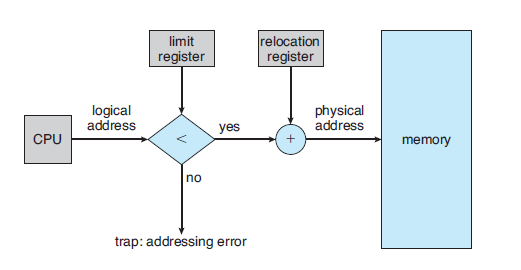
**Contiguous Memory Allocation:**

The main memory must accommodate both the operating system and the various user processes.We therefore need to allocate main memory in the most efficient way possible.

**Memory Protection:**

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.

The relocation-register scheme provides an effective way to allow the operating system’s size to change dynamically. For example, the operating system contains code and buffer space for device drivers. If a device driver is not commonly used, we do not want to keep the code and data in memory, as we might be able to use that space for other purposes. Such code is sometimes called transient operating-system code.



**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. Thus, the degree of multiprogramming is bound by the number of partitions. 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. This method was originally used by the IBM OS/360.

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** and memory contains a set of holes of various sizes.

As processes enter the system, they are put into an input queue. 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.

When no available block of memory (or hole) is large enough to hold a process, the operating system can then wait until a large enough block is available, or it can skip down the input queue to see whether the smaller memory requirements of some other process can be met.

In general, as mentioned, the memory blocks available comprise a set of holes of various sizes scattered throughout memory. 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.

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.
* **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.

**Fragmentation:**

Both the first-fit and best-fit strategies for memory allocation suffer from **external fragmentation**. 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.

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 instead, 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. The memory allocated to a process may be slightly larger than the requested memory. The difference between these two numbers is **internal fragmentation.**

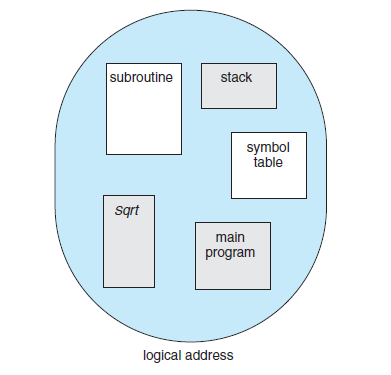
One solution to the problem of external fragmentation is **compaction**. The goal is to shuffle the memory contents so as to place all free memory together in one large block. Compaction is not always possible, however. If relocation is static and is done at assembly or load time, compaction cannot be done. It is possible only if relocation is dynamic and is done at execution time.

------------------------------------------------------------------------------------------------------------------------------

**SEGMENTATION:**

Segmentation is a memory-management scheme that supports programmer view of memory. When writing a program, a programmer thinks of it as a main program with a set of methods, procedures, or functions. It may also include various data structures: objects, arrays, stacks, variables, and so on. Each of these modules or data elements is referred to by name.

Programmers are not concerned with whether the stack is stored before or after the Sqrt() function. Segments vary in length, and the length of each is intrinsically defined by its purpose in the program. Elements within a segment are identified by their offset from the beginning of the segment.



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.

Thus, a logical address consists of a two tuple:

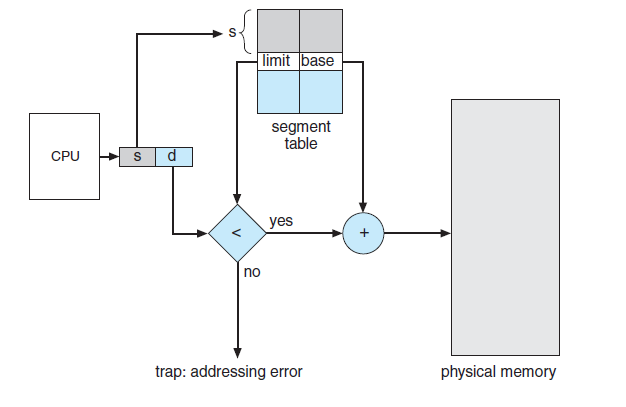
***<*segment-number, offset*>***

When a program is compiled, the compiler automatically constructs segments reflecting the input program. A C compiler might create separate segments for the following:

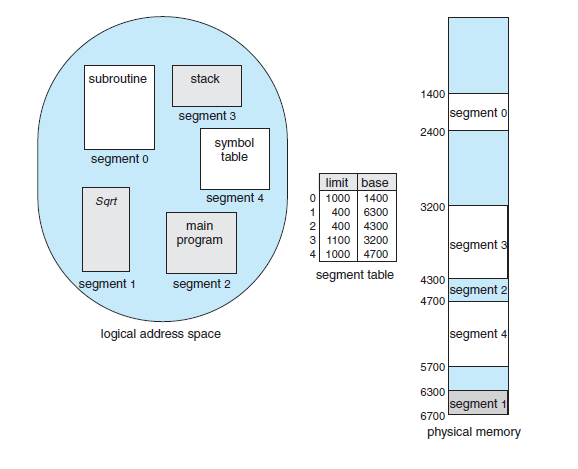
1. The code
2. Global variables
3. The heap, from which memory is allocated
4. The stacks used by each thread
5. The standard C library

**Segmentation Hardware:**

We must define an implementation to map two-dimensional user-defined addresses into one-dimensional physical addresses. 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.



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. When an offset is legal, it is added to the segment base to produce the address in physical memory of the desired byte.

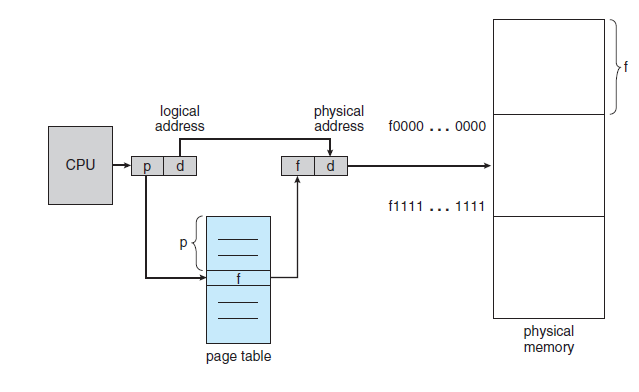


We have five segments numbered from 0 through 4. The segments are stored in physical memory. The segment table has a separate entry for each segment, giving the beginning address of the segment in physical memory (or base) and the length of that segment (or limit). For example, segment 2 is 400 bytes long and begins at location 4300. Thus, a reference to byte 53 of segment 2 is mapped onto location 4300 + 53 = 4353. A reference to segment 3, byte 852, is mapped to 3200 (the base of segment 3) + 852 = 4052. A reference to byte 1222 of segment 0 would result in a trap to the operating system, as this segment is only 1,000 bytes long.

**Paging:**

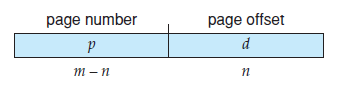
Segmentation permits the physical address space of a process to be non contiguous. **Paging** is another memory-management scheme that offers this advantage. However, paging avoids external fragmentation and the need for compaction, whereas segmentation does not.

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 ages are loaded into any available memory frames from their source. The backing store is divided into fixed-sized blocks that are the same size as the memory frames or clusters of multiple frames.

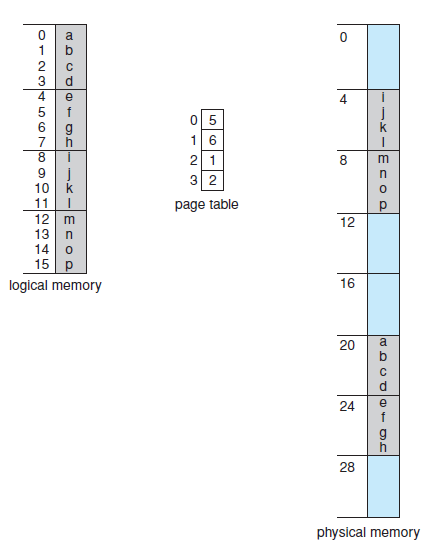


Every address generated by the CPU is divided into two parts: a **page number (p)** and a **page offset (d)**. The page number is used as an index into a **page table**. The page table contains the base address of each page in physical memory. This base address is combined with the page offset to define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



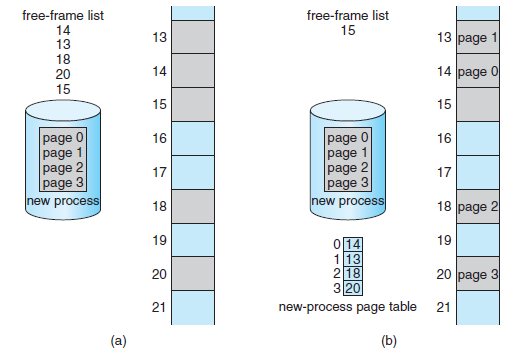
p is an index into the page table and d is the displacement within the page.



Consider the memory in the above figure were *n* = 2 and *m* = 4 with a page size of 4 bytes and a physical memory of 32 bytes. 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 × 4) + 0]. Logical address 3 (page 0, offset 3) maps to physical address 23 [= (5 × 4) + 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 [= (6 × 4) + 0]. Logical address 13 maps to physical address 9.

We have no external fragmentation: any free frame can be allocated to a process that needs it. However, we may have some internal fragmentation. For example, if page size is 2,048 bytes, a process of 72,766 bytes will need 35 pages plus 1,086 bytes. It will be allocated 36 frames, resulting in internal fragmentation of 2,048 − 1,086 = 962 bytes.

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 in the process control block.

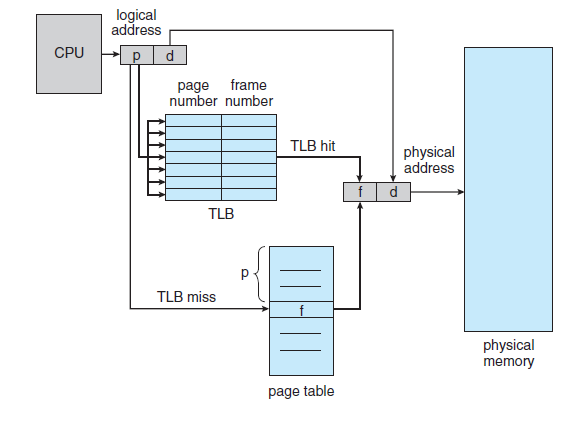
The hardware implementation of the page table can be done as a set of dedicated **registers**. These registers should be built with very high-speed logic to make the paging-address translation efficient. The use of registers for the page table is satisfactory if the page table is reasonably small (for example, 256 entries). Most contemporary computers, however, allow the page table to be very large. For these machines, the page table is kept in main memory, and a **page-table base register (PTBR)** points to the page table.

The problem with this approach is the time required to access a user memory location. If we want to access location i, we must first index into the page table, using the value in the PTBR offset by the page number for i. This task requires a memory access. 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. Thus, memory access is slowed by a factor of 2.

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 search is fast; a TLB lookup in modern hardware is part of the instruction pipeline, essentially adding no performance penalty.

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. As just mentioned, these steps are executed as part of the instruction pipeline within the CPU, adding no performance penalty compared with a system that does not implement paging.

If the page number is not in the TLB (known as a **TLB miss**), a memory reference to the page table must be made. 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 TLBs store **address-space identifiers (ASIDs)** in each TLB entry. An ASID uniquely identifies each process and is used to provide address-space protection for that process. When the TLB attempts to resolve virtual page numbers, it ensures that the ASID for the currently running process matches the ASID associated with the virtual page. If the ASIDs do not match, the attempt is treated as a TLB miss.

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 percent 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.

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

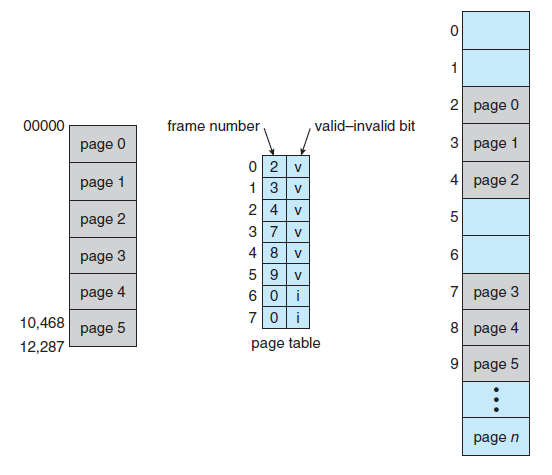
effective access time = 0.99 × 100 + 0.01 × 200 = 101 nanoseconds

**Protection:**

Memory protection in a paged environment is accomplished by protection bits associated with each frame. 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.

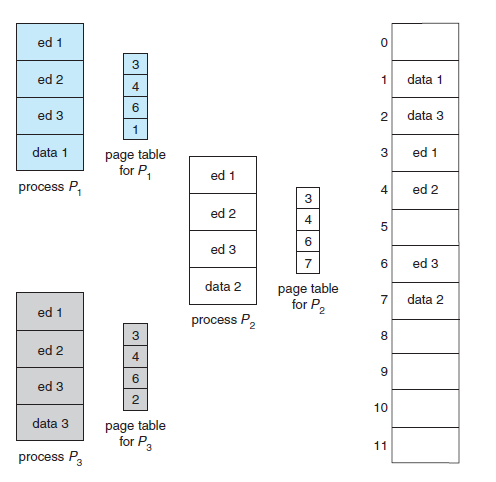
One additional bit is generally attached to each entry in the page table: a **valid–invalid** bit. When this bit is set to valid, the associated page is in the process’s logical address space and is thus a legal (or valid) page. When the bit is set to invalid, the page is not in the process’s logical address space. Illegal addresses are trapped by use of the valid–invalid bit.

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

****

**Shared Pages:**

An advantage of paging is the possibility of ***sharing*** common code. This consideration is particularly important in a time-sharing environment. Consider a system that supports 40 users, each of whom executes a text editor. If the text editor consists of 150 KB of code and 50 KB of data space, we need 8,000 KB to support the 40 users. If the code is **re-entrant code** (or **pure code**), however, it can be shared.

****

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.

**Intel 32 and 64-bit Architectures:**

The architecture of Intel chips has dominated the personal computer landscape for several years. The 16-bit Intel 8086 appeared in the late 1970s and was soon followed by another 16-bit chip—the Intel 8088—which was notable for being the chip used in the original IBM PC. Both the 8086 chip and the 8088 chip were

based on a segmented architecture. The IA-32 architecture supported both paging and segmentation. More recently, Intel has produced a series of 64-bit chips based on the x86-64 architecture.

**IA-32 Architecture:**

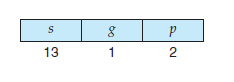
Memory management in IA-32 systems is divided into two components: segmentation and paging. The CPU generates logical addresses, which are given to the segmentation unit. The segmentation unit produces a linear address for each logical address. The linear address is then given to the paging unit, which in turn generates the physical address in main memory.



**IA-32 Segmentation:**

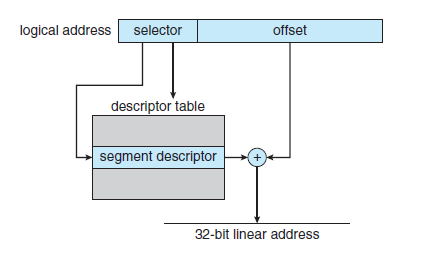
The logical address space of a process is divided into two partitions. The first partition consists of up to 8 K segments that are private to that process. The second partition consists of up to 8 K segments that are shared among all the processes. Information about the first partition is kept in the **local descriptor table (LDT)**; information about the second partition is kept in the **global descriptor table (GDT)**.

The logical address is a pair (selector, offset), where the selector is a 16-bit number:



in which *s* designates the segment number, *g* indicates whether the segment is in the GDT or LDT, and *p* deals with protection.

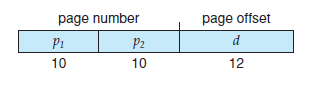
The base and limit information about the segment in question is used to generate a **linear address**. First, the limit is used to check for address validity. If the address is not valid, a memory fault is generated, resulting in a trap to the operating system. If it is valid, then the value of the offset is added to the value of the base, resulting in a 32-bit linear address.



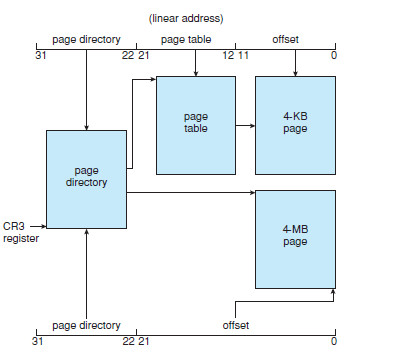
**Segmentation in IA – 32 architecture**

**IA-32 Paging:**

The IA-32 architecture allows a page size of either 4 KB or 4 MB. For 4-KB pages, IA-32 uses a two-level paging scheme in which the division of the 32-bit linear address is as follows:



The 10 high-order bits reference an entry in the outermost page table, which IA-32 terms the **page directory**. The page directory entry points to an inner page table that is indexed by the contents of the innermost 10 bits in the linear address. Finally, the low-order bits 0–11 refer to the offset in the 4-KB page pointed to in the page table.

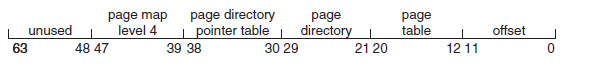


**Paging in IA – 32 architecture**

To improve the efficiency of physical memory use, IA-32 page tables can be swapped to disk. In this case, an invalid bit is used in the page directory entry to indicate whether the table to which the entry is pointing is in memory or on disk.

**x86-64:**

Its initial entry was the IA-64 (later named **Itanium**) architecture, but that architecture was not widely adopted. Meanwhile, another chip manufacturer— AMD — began developing a 64-bit architecture known as x86-64 that was based on extending the existing IA-32 instruction set. It Supports 64-bit address space which yields an astonishing 264 bytes of addressable memory - a number greater than 16 quintillion. The x86-64 architecture currently provides a 48-bit virtual address with support for page sizes of 4 KB, 2 MB, or 1 GB using four levels of paging hierarchy.



**x86-64 linear address.**

**VIRTUAL MEMORY:**

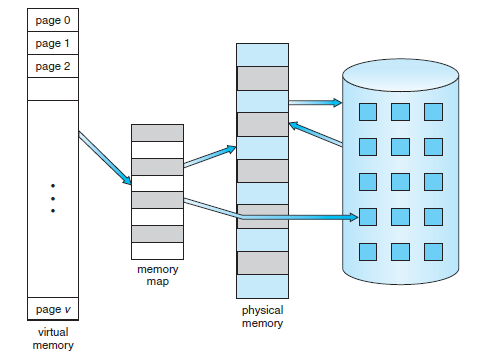
The instructions being executed must be in physical memory. The first approach to meeting this requirement is to place the entire logical address space in physical memory. An examination of real programs shows us that, in many cases, the entire program is not needed. For instance:

* Programs often have code to handle unusual error conditions. Since these errors seldom, if ever, occur in practice, this code is almost never executed.
* Arrays, lists, and tables are often allocated more memory than they actually need.

The ability to execute a program that is only partially in memory would confer many benefits:

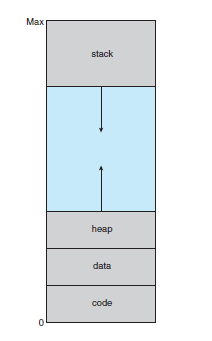
* A program would no longer be constrained by the amount of physical memory that is available.
* Because each user program could take less physical memory, more programs could be run at the same time, with a corresponding increase in CPU utilization and throughput
* Less I/O would be needed to load or swap user programs into memory, so each user program would run faster.

**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



**virtual memory that is larger than physical memory**

The **virtual address space** of a process refers to the logical (or virtual) view of how a process is stored in memory. Typically, this view is that a process begins at a certain logical address—say, address 0—and exists in contiguous memory and physical memory may be organized in page frames and that the physical page frames assigned to a process may not be contiguous.



Heap grows upward in a memory as it is used for dynamic memory allocation. Similarly stack grows downward in memory through successive function calls. The large blank space (or hole) between the heap and the stack is part of the virtual address space but will require actual physical pages only if the heap or stack grows. Virtual address spaces that include holes are known as **sparse** address spaces.

Virtual memory leads to the following benefits:

* System libraries can be shared by several processes through mapping of the shared object into a virtual address space.
* Two or more processes can communicate through the use of shared memory. Virtual memory allows one process to create a region of memory that it can share with another process.
* Pages can be shared during process creation with the fork() system call, thus speeding up process creation

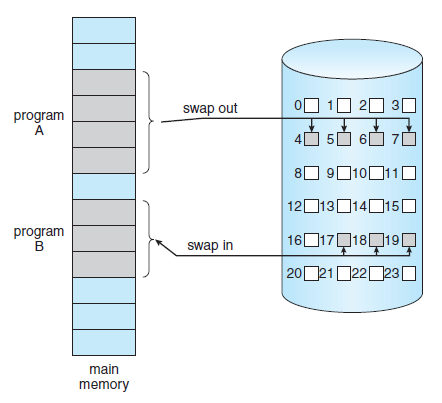
----------------------------------------------------------------------------------------------------------------------------

**Demand Paging:**

With demand-paged virtual memory, pages are loaded only when they are demanded during program execution. Pages that are never accessed are thus never loaded into physical memory.

A demand-paging system is similar to a paging system with swapping where processes reside in secondary memory (usually a disk). When we want to execute a process, we swap it into memory. Rather than swapping the entire process into memory, though, we use a **lazy swapper**. A lazy swapper never swaps a page into memory unless that page will be needed.

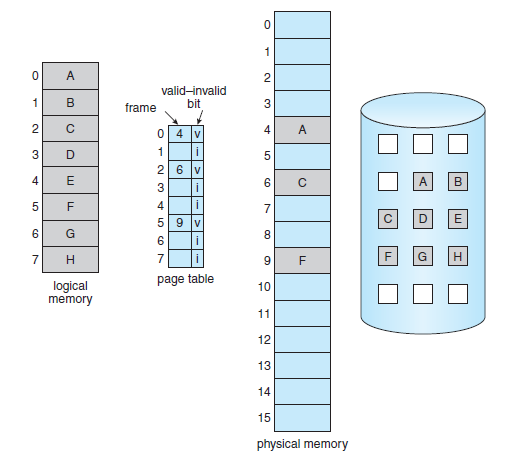
We use “pager,” rather than “swapper,” in connection with demand paging.



**Transfer of a paged memory to contiguous disk space**

**Basic Concepts:**

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 and reduces the swap time. With this scheme, 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 described can be used for this purpose.



**Page table when some pages are not in main memory.**

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 or is valid but is currently on the disk. Access to a page marked invalid causes a **page fault**.

The procedure for handling this page fault is straightforward:

1. We check an internal table for this process to determine whether the reference was a valid or an invalid memory access.
2. If the reference was invalid, we terminate the process. If it was valid but we have not yet brought in that page, we now page it in.
3. We find a free frame
4. We schedule a disk operation to read the desired page into the newly allocated frame.
5. When the disk read is complete, we modify the internal table kept with the process and the page table to indicate that the page is now in memory
6. We restart the instruction that was interrupted by the trap. The process can now access the page as though it had always been in memory.

In the extreme case, we can start executing a process with ***no*** pages in memory. When the operating system sets the instruction pointer to the first instruction of the process, which is on a non-memory-resident page, the process immediately faults for the page. After this page is brought into memory, the process continues to execute, faulting as necessary until every page that it needs is in memory. At that point, it can execute with no more faults. This scheme is **pure demand paging**: never bring a page into memory until it is required.

The hardware to support demand paging is

* Page table
* Secondary memory

A crucial requirement for demand paging is the ability to restart any instruction after a page fault. Because we save the state (registers, condition code, instruction counter) of the interrupted process when the page fault occurs, we must be able to restart the process in **exactly** the same place and state.

**Performance of Demand Paging:**

Demand paging can significantly affect the performance of a computer system. For most computer systems, the memory-access time, denoted *ma,* ranges from 10 to 200 nanoseconds. As long as we have no page faults, the effective access time is equal to the memory access time. If, however, a page fault occurs, we must first read the relevant page from disk and then access the desired word.

Let *p* be the probability of a page fault (0 ≤ *p* ≤ 1). We would expect *p* to be close to zero - that is, we would expect to have only a few page faults. The **effective access time** is then

effective access time = (1 − *p*) × *ma* + *p* × page fault time.

A page fault causes the following sequence to occur:

1. Trap to the operating system.

2. Save the user registers and process state.

3. Determine that the interrupt was a page fault.

4. Check that the page reference was legal and determine the location of the page on the disk.

5. Issue a read from the disk to a free frame:

a. Wait in a queue for this device until the read request is serviced.

b. Wait for the device seek and/or latency time.

c. Begin the transfer of the page to a free frame.

6. While waiting, allocate the CPU to some other user (CPU scheduling, optional).

7. Receive an interrupt from the disk I/O subsystem (I/O completed).

8. Save the registers and process state for the other user (if step 6 is executed).

9. Determine that the interrupt was from the disk.

10. Correct the page table and other tables to show that the desired page is now in memory.

11. Wait for the CPU to be allocated to this process again.

12. Restore the user registers, process state, and new page table, and then resume the interrupted instruction.

With an average page-fault service time of 8 milliseconds and a memory access time of 200 nanoseconds, the effective access time in nanoseconds is

effective access time = (1 − *p*) × (200) + *p* (8 milliseconds)

= (1 − *p*) × 200 + *p* × 8,000,000

= 200 + 7,999,800 × *p.*

If we want performance degradation to be less than 10 percent, we need to keep the probability of page faults at the following level:

220 *>* 200 + 7,999,800 × *p,*

20 *>* 7,999,800 × *p*,

*p <* 0.0000025.

*-------------------------------------------------------------------------------------------------------------------------------*

**Allocation of Frames:**

Consider a single-user system with 128 KB of memory composed of pages 1 KB in size. This system has 128 frames. The operating system may take 35 KB, leaving 93 frames for the user process. Under pure demand paging, all 93 frames would initially be put on the free-frame list. When a user process started execution, it would generate a sequence of page faults. The first 93 page faults would get free frames from the free-frame list. When the free-frame list was exhausted, a page-replacement algorithm would be used to select one of the 93 in-memory pages to be replaced with the 94th, and so on. When the process terminated, the 93 frames would once again be placed on the free-frame list.

**Minimum Number of Frames:**

Strategies for the allocation of frames are constrained in various ways. We cannot allocate more than the total number of available frames and we must also allocate at least a minimum number of frames for a particular process.

One reason for allocating at least a minimum number of frames involves performance. Obviously, as the number of frames allocated to each process decreases, the page-fault rate increases, slowing process execution.

For example, consider a machine which needs at least one frame for the instruction and one frame for the memory reference. In addition, if one-level indirect addressing is allowed (for example, a load instruction on page 16 can refer to an address on page 0, which is an indirect reference to page 23), then paging requires at least three frames per process. The minimum number of frames is defined by the computer architecture.

For example in IBM 370 MVC instruction, since the instruction is from storage location to storage location, it takes 6 bytes and can straddle two pages. The block of characters to move and the area to which it is to be moved can each also straddle two pages. This situation would require six frames.

The minimum number of frames per process is defined by the architecture, the maximum number is defined by the amount of available physical memory.

**Allocation Algorithms:**

The easiest way to split *m* frames among *n* processes is to give everyone an equal share, *m*/*n* frames. For instance, if there are 93 frames and five processes, each process will get 18 frames. The three leftover frames can be used as a free-frame buffer pool. This scheme is called **equal allocation**.

If a small student process of 10 KB and an interactive database of 127 KB are the only two processes running in a system with 62 free frames, it does not make much sense to give each process 31 frames. The student process does not need more than 10 frames, so the other 21 are, strictly speaking, wasted.

To solve this problem, we can use **proportional allocation**, in which we allocate available memory to each process according to its size. Let the size of the virtual memory for process *pi* be *si* , and define



Then, if the total number of available frames is *m,* we allocate *ai* frames to process *pi*, where *ai* is approximately

*ai* = *si* /*S* × *m*.

With proportional allocation, we would split 62 frames between two processes, one of 10 pages and one of 127 pages, by allocating 4 frames and 57 frames, respectively, since

10/137 × 62 ≈ 4, and

127/137 × 62 ≈ 57.

**Global versus Local Allocation:**

With multiple processes competing for frames, we can classify page-replacement algorithms into two broad categories: **global replacement** and **local replacement**. Global replacement allows a process to select a replacement frame from the set of all frames, even if that frame is currently allocated to some other process; that is, one process can take a frame from another. Local replacement requires that each process select from only its own set of allocated frames.

Under local replacement, the set of pages in memory for a process is affected by the paging behavior of only that process.

**Non-Uniform Memory Access:**

In systems with multiple CPUs, a given CPU can access some sections of main memory faster than it can access others. These performance differences are caused by how CPUs and memory are interconnected in the system. Frequently, such a system is made up of several system boards, each containing multiple CPUs and some memory. The system boards are interconnected in various ways, ranging from system buses to high-speed network connections like InfiniBand. So the CPUs on a particular board can access the memory on that board with less delay than they can access memory on other boards in the system. Systems in which memory access times vary significantly are known collectively as **non-uniform memory access (NUMA).**

Under NUMA memory frames are allocated “as close as possible” to the CPU on which the process is running. The definition of “close” is “with minimum latency,” which typically means on the same system board as the CPU.

**Allocating Kernel Memory:**

Kernel memory is often allocated from a free-memory pool different from the list used to satisfy ordinary user-mode processes. There are two primary reasons for this:

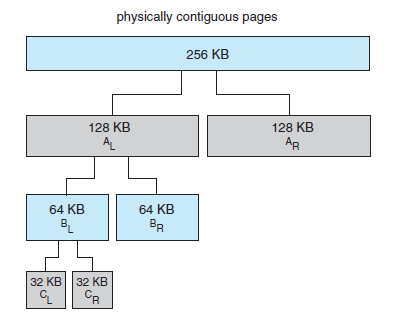
1. The kernel requests memory for data structures of varying sizes, some of which are less than a page in size. As a result, the kernel must use memory conservatively and attempt to minimize waste due to fragmentation.
2. Pages allocated to user-mode processes do not necessarily have to be in contiguous physical memory. However, certain hardware devices interact directly with physical memory without the benefit of a virtual memory interface may require memory residing in physically contiguous pages.

We examine two strategies for managing free memory that is assigned to kernel processes: the buddy system and slab allocation

**Buddy System:**

The buddy system allocates memory from a fixed-size segment consisting of physically contiguous pages. Memory is allocated from this segment using a **power-of-2 allocator**(4 KB, 8 KB, 16 KB, and so forth)**.** A request in units not appropriately sized is rounded up to the next highest power of 2. For example, a request for 11 KB is satisfied with a 16-KB segment.

Assume the size of a memory segment is initially 256 KB and the kernel requests 21 KB of memory. The segment is initially divided into two **buddies** *AL* and *AR*—each 128 KB in size. One of these buddies is further divided into two 64-KB buddies *BL* and *BR*. However, the next-highest power of 2 from 21 KB is 32 KB so either *BL* or *BR* is again divided into two 32-KB buddies, *CL* and *CR*. One of these buddies is used to satisfy the 21-KB request.



An advantage of the buddy system is how quickly adjacent buddies can be combined to form larger segments using a technique known as **coalescing.**

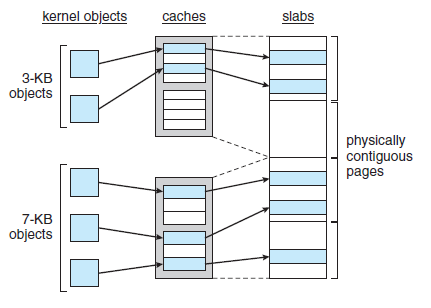
When the kernel releases the *CL* unit it was allocated, the system can coalesce *CL* and *CR* into a 64-KB segment. This segment, *BL*, can in turn be coalesced with its buddy *BR* to form a 128-KB segment.

Drawback to the buddy system is that rounding up to the next highest power of 2 is very likely to cause internal fragmentation. For example, a 33-KB request can only be satisfied with a 64- KB segment.

**Slab Allocation:**

A **slab** is made up of one or more physically contiguous pages. A **cache** consists of one or more slabs. There is a single cache for each unique kernel data structure —for example, a separate cache for the data structure representing process descriptors, a separate cache for file objects, a separate cache for semaphores, and so forth. Each cache is populated with **objects** that are instantiations of the kernel data structure the cache represents. For example, the cache representing semaphores stores instances of semaphore objects.

The slab-allocation algorithm uses caches to store kernel objects. When a cache is created, a number of objects which are initially marked as free are allocated to the cache. The number of objects in the cache depends on the size of the associated slab.

****

In Linux, a slab may be in one of three possible states:

1. **Full**: All objects in the slab are marked as used.

2. **Empty**: All objects in the slab are marked as free.

3. **Partial**: The slab consists of both used and free objects.

The slab allocator first attempts to satisfy the request with a free object in a partial slab. If none exists, a free object is assigned from an empty slab. If no empty slabs are available, a new slab is allocated from contiguous physical pages and assigned to a cache.

The slab allocator provides two main benefits:

1. No memory is wasted due to fragmentation. Fragmentation is not an issue because each unique kernel data structure has an associated cache, and each cache is made up of one or more slabs that are divided into chunks the size of the objects being represented.
2. Memory requests can be satisfied quickly. The slab allocation scheme is thus particularly effective for managing memory when objects are frequently allocated and deallocated.

**Operating-System Examples:**

**WINDOWS:**

Windows implements virtual memory using demand paging with **clustering**. Clustering handles page faults by bringing in not only the faulting page but also several pages following the faulting page. When a process is first created, it is assigned a working-set minimum and maximum. The **working-set minimum** is the minimum number of pages the process is guaranteed to have in memory. If sufficient memory is available, a process may be assigned as many pages as its **working-set maximum**.

The virtual memory manager maintains a list of free page frames. Associated with this list is a threshold value that is used to indicate whether sufficient free memory is available.

When the amount of free memory falls below the threshold, the virtual memory manager uses a tactic known as **automatic working-set trimming** to restore the value above the threshold. Automatic working-set trimming works by evaluating the number of pages allocated to processes. If a process has been allocated more pages than its working-set minimum, the virtual memory manager removes pages until the process reaches its working-set minimum. A process that is at its working-set minimum may be allocated pages from the free-page-frame list once sufficient free memory is available.

**Thrashing:**

Consider 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.

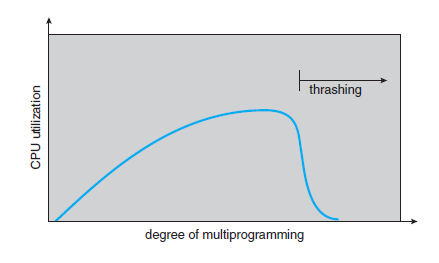
This high paging activity is called **thrashing**. A process is thrashing if it is spending more time paging than executing.

**Cause of Thrashing:**

The operating system monitors CPU utilization. If CPU utilization is too low, it increases the degree of multiprogramming by introducing a new process to the system. A global page-replacement algorithm replaces pages without regard to the process to which they belong. Now suppose that a process enters a new phase in its execution and needs more frames. It starts faulting and taking frames away from other processes. These processes need those pages, however, and so they also fault, taking frames from other processes. These

faulting processes must use the paging device to swap pages in and out. As they queue up for the paging device, the ready queue empties. As processes wait for the paging device, CPU utilization decreases.

The CPU scheduler sees the decreasing CPU utilization and **increases** the degree of multiprogramming as a result. The new process tries to get started by taking frames from running processes, causing more page faults and a longer queue for the paging device. As a result, CPU utilization drops even further.



In above graph CPU utilization is plotted against the degree of multiprogramming. As the degree of multiprogramming increases, CPU utilization also increases, although more slowly, until a maximum is reached. If the degree of multiprogramming is increased even further, thrashing sets in, and CPU utilization drops sharply. At this point, to increase CPU utilization and stop thrashing, we must **decrease** the degree of multiprogramming.

We can limit the effects of thrashing by using a **local replacement algorithm**. With local replacement, if one process starts thrashing, it cannot steal frames from another process.

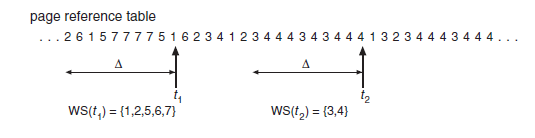
The locality model states that, as a process executes, it moves from locality to locality. A locality is a set of pages that are actively used together. A program is generally composed of several different localities, which may overlap.

There are several techniques to avoid thrashing:

**Working-Set Model:**

**Working-set model** is based on the assumption of locality. This model uses a parameter ,, to define the **working-set window**. The idea is to examine the most recent page references. The set of pages in the most

recent page references is the **working set.** If a page is in active use, it will be in the working set. If it is no longer being used, it will drop from the working set. Thus, the working set is an approximation of the program’s locality.

****

For example, given the sequence of memory references shown in Figure, if = 10 memory references, then the working set at time *t*1 is *{* 1, 2, 5, 6, 7 *}*. By time *t*2, the working set has changed to *{* 3, 4 *}*.

If we compute the working-set size, *WSSi* , for each process in the system, we can then consider that

****

where *D* is the total demand for frames. Each process is actively using the pages in its working set. Thus, process *i* needs *WSSi* frames. If the total demand is greater than the total number of available frames (*D> m*), thrashing will occur.

**Page-Fault Frequency:**

Thrashing has a high page-fault rate. Thus, we want to control the page-fault rate. When it is too high, we know that the process needs more frames. Conversely, if the page-fault rate is too low, then the process may have too many frames. We can establish upper and lower bounds on the desired page-fault rate.

If the actual page-fault rate exceeds the upper limit, we allocate the process another frame. If the page-fault rate falls below the lower limit, we remove a frame from the process. Thus, we can directly measure and control the page-fault rate to prevent thrashing.