

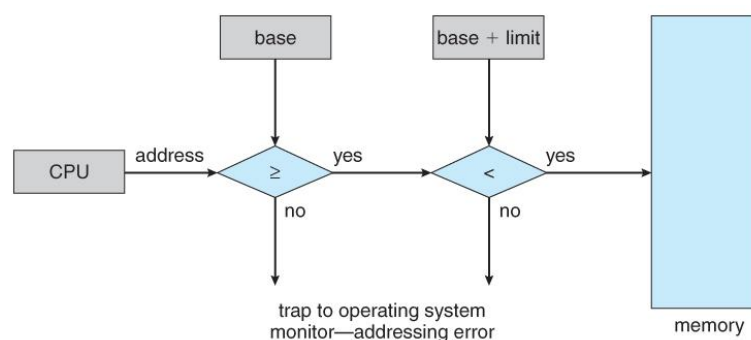
Unit IV – Memory Management

The main purpose of a computer system is to execute programs. These programs, together with the data they access, must be at least partially in main memory during execution. To improve both the utilization of the CPU and the speed of its response to users, a general-purpose computer must keep several processes in memory. Many memory-management schemes exist, reflecting various approaches, and the effectiveness of each algorithm depends on the situation. Selection of a memory-management scheme for a system depends on many factors, especially on the *hardware* design of the system. Most algorithms require hardware support.

Basic Hardware

Main memory and the registers built into the processor itself are the only storage that the CPU can access directly. There are machine instructions that take memory addresses as arguments, but none that take disk addresses. 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. Most CPUs can decode instructions and perform simple operations on register contents at the rate of one or more operations per clock tick. 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, which treats the attempt as a fatal error (figure below). This scheme prevents a user program from (accidentally or deliberately) modifying the code or data structures of either the operating system or other users.

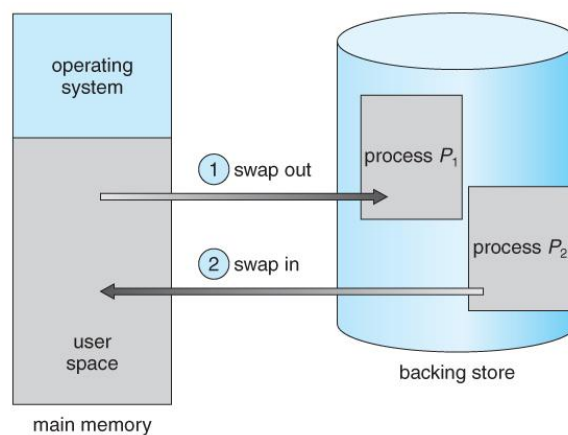


(Hardware address protection with base and limit registers)

The base and limit registers can be loaded only by the operating system, which uses a special privileged instruction. Since privileged instructions can be executed only in kernel mode, and since only the operating system executes in kernel mode, only the operating system can load the base and limit registers. This scheme allows the operating system to change the value of the registers but prevents user programs from changing the registers' contents.

Swapping

A process must be in memory to be executed. A process, however, can be temporarily out of memory to a *backing store* and then brought into memory for continued execution. For example, assume a multiprogramming environment with a round-robin CPU-scheduling algorithm. When a quantum expires, the memory manager will start to swap out the process that just finished and to swap another process into the memory space that has been freed. In the meantime, the CPU scheduler will allocate a time slice to some other process in memory. When each process finishes its quantum, it will be swapped with another process. Ideally, the memory manager can swap processes fast enough that some processes will be in memory, ready to execute, when the CPU scheduler wants to reschedule the CPU. In addition, the quantum must be large enough to allow reasonable amounts of computing to be done between swaps.



(Swapping of two processes using a disk as a backing store)

A variant of this swapping policy is used for priority-based scheduling algorithms. If a higher-priority process arrives and wants service, the memory manager can swap out the lower-priority process and then load and execute the higher-priority process. When the higher-priority process finishes, the lower-priority process can be swapped back in and continued. This variant of swapping is sometimes called *roll out*, *roll in*.

Swapping requires 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.

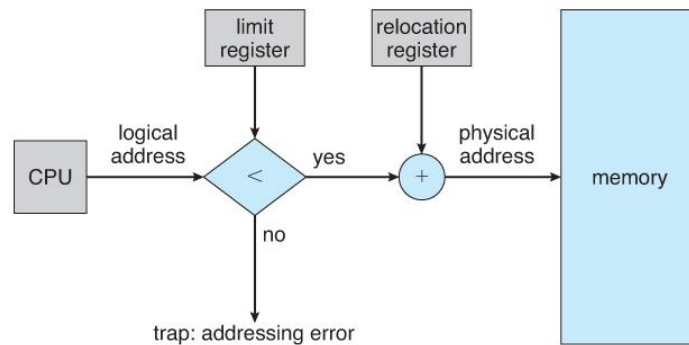
Contiguous Memory Allocation

The 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. In contiguous memory allocation, each process is contained in a single contiguous section of memory.

Memory Mapping and Protection

These features can provide by using are location register, together with a limit register. The relocation register contains the value of the smallest physical address; the limit register contains the range of logical addresses. With relocation and limit registers, each logical address must be less than 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.



(Hardware support for relocation and limit registers)

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. This flexibility is desirable in many situations. For example, the operating system contains code and buffer space for device drivers. If a device driver (or other operating-system service) 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; it comes and goes as needed. Thus, using this code changes the size of the operating system during program execution.

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, 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. The operating system can order the input queue according to a scheduling algorithm.

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. Eventually memory contains a set

of holes of various sizes. Memory is allocated to processes until finally, the memory requirements of the next process cannot be satisfied -that is, no available block of memory (or hole) is large enough to hold that 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, 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. At this point, the system may need to check whether there are processes waiting for memory and whether this newly freed and recombined memory could satisfy the demands of any of these waiting processes.

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 size. There are many solutions to this problem. 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. It can stop searching as soon as it finds a free hole that is large enough.
- **Best fit.** Allocate the smallest hole that is big enough. It 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, it 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.

Simulations have shown that both first fit and best fit are better than worst fit in terms of decreasing time and storage utilization. Neither first fit nor best fit is clearly better than the other in terms of storage utilization, but first fit is generally faster.

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. This fragmentation problem can be severe. In the worst case, it 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.

Depending on the total amount of memory storage and the average process size, external fragmentation may be a minor or a major problem. Statistical analysis of first fit, for instance, reveals that, even with some optimization, given N allocated blocks, another $0.5 N$ blocks will be lost to fragmentation. That is, one-third of memory may be unusable! This property is known as the *50-percent scheme*.

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 memory that is *internal fragmentation* - used internal to a partition.

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; compaction is possible *only* if relocation is dynamic and is done at execution time. If addresses are relocated dynamically, relocation requires only moving the program and data and then changing the base register to reflect the new base address. When compaction is possible, it must determine its cost. 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 to permit the logical address space of the processes to be non-contiguous, thus allowing a process to be allocated physical memory wherever such memory is available. Two complementary techniques achieve this solution that is paging and segmentation.

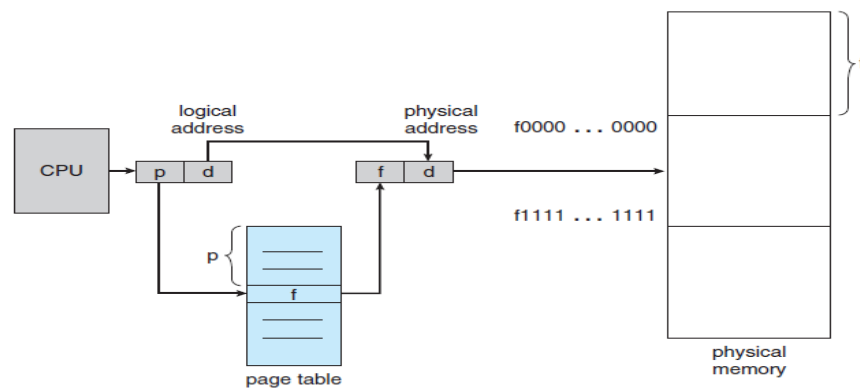
Paging

Paging is a memory-management scheme that permits the physical address space a process to be non-contiguous. Paging avoids external fragmentation and the need for compaction. It also solves the considerable problem of fitting memory chunks of varying sizes onto the backing store; most memory management schemes used before the introduction of paging suffered from this problem. The problem arises because, when some code fragments or data residing in main memory need to be swapped out, space must be found on the backing store. The backing store has the same fragmentation problems, but access is much slower, so compaction is impossible. Because of its advantages over earlier methods, paging in its various forms is used in most operating systems.

Traditionally, support for paging has been handled by hardware. However, recent designs have implemented paging by closely integrating the hardware and operating system, especially on 64-bit microprocessors.

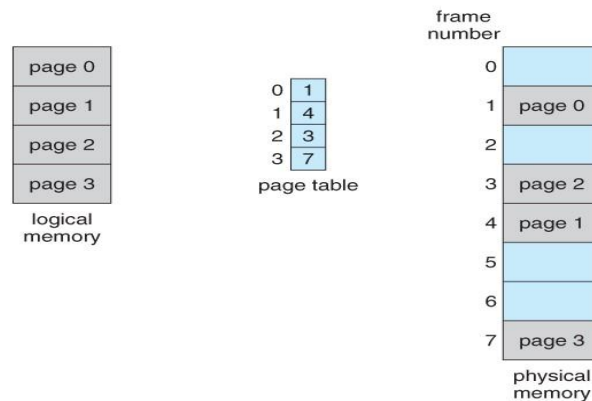
Basic Method

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 of the same size as the memory frames. The hardware support for paging is illustrated in the following figure.



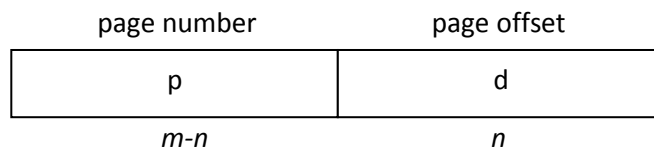
(Paging hardware)

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



(Paging model of logical and physical memory)

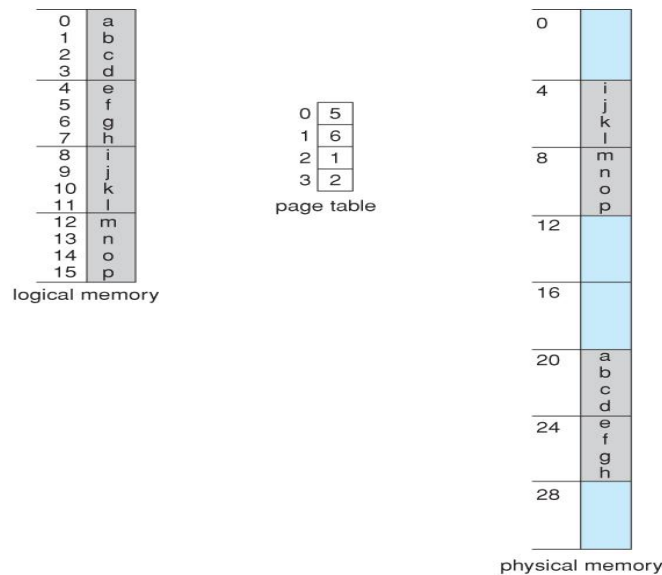
The page size (like the frame size) is defined by the hardware. The size of a page is typically a power of 2, varying between 512 bytes and 16 MB per page, depending on the computer architecture. The selection of a power of 2 as a page size makes the translation of a logical address into a page number and page offset particularly easy. If the size of the 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 as follows:



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

Consider the memory in the figure given below, here, in the logical address, $n = 2$ and $m = 4$. Using a page size of 4 bytes and a physical memory of 32 bytes (8 pages), it shows how the user's view of memory can be mapped into physical memory. 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

$[= (5 \times 4) + 0]$. Logical address 3 (page 0, offset 3) maps to physical address 23 $[= (5 \times 4) + 3]$. 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 $[= (6 \times 4) + 0]$. Logical address 13 maps to physical address 9.



(Paging example for a 32-byte memory with 4-byte pages)

Paging itself is a form of dynamic relocation. Every logical address is bound by the paging hardware to some physical address. Using paging is similar to using a table of base (or relocation) registers, one for each frame of memory.

Hardware support

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. Every access to memory must go through the paging map, so efficiency is a major consideration. The CPU dispatcher reloads these registers, just as it reloads the other registers. Instructions to load or modify the page-table registers are, of course, privileged, so that only the operating system can change the memory map.

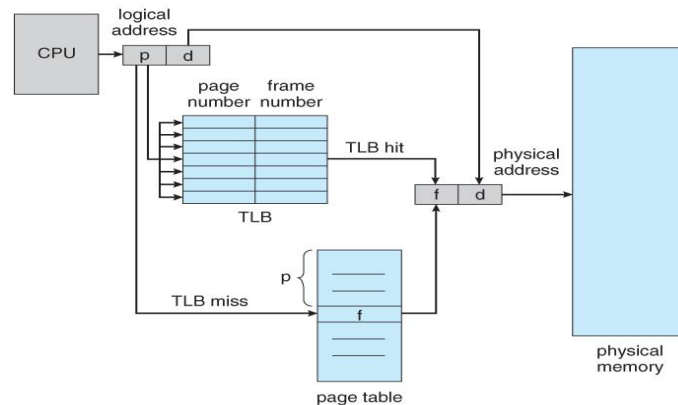
The use of registers for the page table is satisfactory if the page table is reasonably small (for example, 256 entries). Most contemporary computers, however, allow the page table to be very large (for example, 1 million entries). For these machines, the use of fast registers to implement the page table 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.

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. With this scheme, *two* memory accesses are needed to access a byte (one for the page-table entry, one for the byte).

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; the hardware, however, is expensive. Typically, the number of entries in a TLB is small, often numbering between 64 and 1,024.

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 (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.



(Paging hardware with TLB.)

In addition, we add the page number and frame number to the TLB, so that they will be found quickly on the next reference. If the TLB is already full of entries, the operating system must select one for replacement. Replacement policies range from least recently used (LRU) to random. Furthermore, some TLBs allow certain entries to be meaning that they cannot be removed from the TLB. Typically, TLB entries for kernel code are wired down.

The percentage of times that a particular page number 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 20 nanoseconds to search the TLB and 100 nanoseconds to access memory, then a mapped-memory access takes 120 nanoseconds when the page number is in the TLB. If we fail to find the page number in the TLB (20 nanoseconds), 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 220 nanoseconds. To find the *effective memory access time*, we weight the case by its probability:

$$\begin{aligned}\text{effective access time} &= 0.80 \times 120 + 0.20 \times 220 \\ &= 140 \text{ nanoseconds.}\end{aligned}$$

In this example, we suffer a 40-percent slowdown in memory-access time (from 100 to 140 nanoseconds).

For a 98-percent hit ratio, we have

$$\begin{aligned}\text{effective access time} &= 0.98 \times 120 + 0.02 \times 220 \\ &= 122 \text{ nanoseconds.}\end{aligned}$$

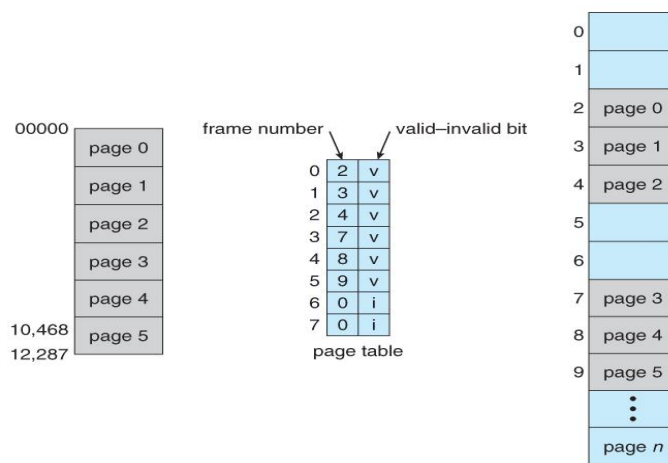
This increased hit rate produces only a 22 percent slowdown in access time.

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 (or memory-protection violation).

We can create hardware to provide read-only, read-write, or execute-only protection; or, by providing separate protection bits for each kind of access, we can allow any combination of these accesses. Illegal attempts will be trapped 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.

For example, that in a system with a 14-bit address space (0 to 16383), we have a program that should use only addresses 0 to 10468. Given a page size of 2 KB, we have the situation shown in figure below.



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

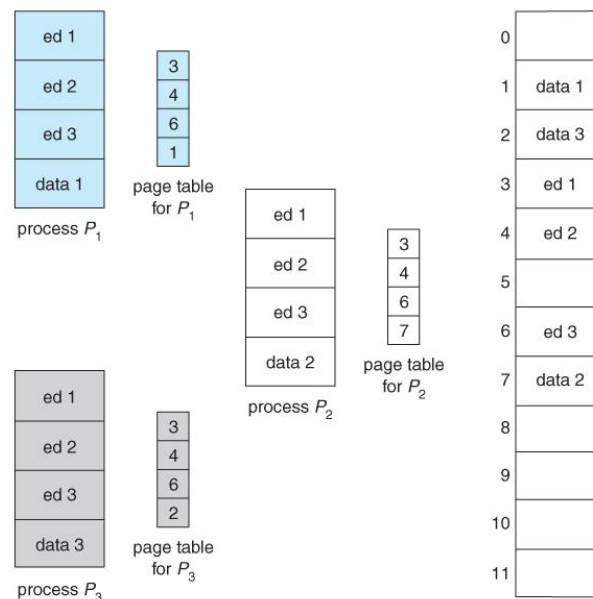
Addresses in pages 0, 1, 2, 3, 4, and 5 are mapped normally through the page table. Any attempt to generate an address in pages 6 or 7, however, will find that the valid -invalid bit is set to invalid, and the computer will trap to the operating system (invalid page reference).

Notice that this scheme has created a problem. Because the program extends only to address 10468, any reference beyond that address is illegal. However references to page 5 are classified as valid, so accesses to addresses up to 12287 are valid. Only the addresses from 12288 to 16383 are invalid. This problem is a result of the 2-KB page size and reflects the internal fragmentation of paging.

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, as shown in the following figure. Here we see a three-page editor-each page 50 KB in size (the large page size is used to simplify the figure)-being shared among three processes. Each process has its own data page.



(Sharing of code in a paging environment)

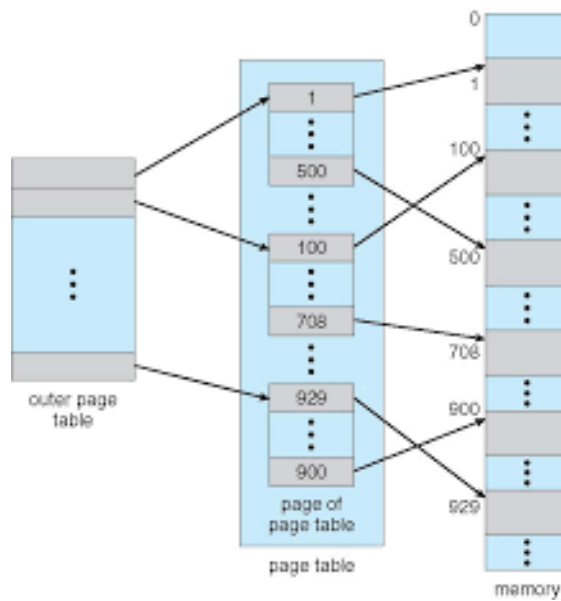
Re-entrant code is non-self-modifying code: it never changes during execution. Thus, two or more processes can execute the same code at the same time. Each process has its own copy of registers and data storage to hold the data for the process's execution. The data for two different processes will of 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. Thus, to support 40 users, we need only one copy of the editor (150 KB), plus 40 copies of the 50 KB of data space per user. The total space required is now 2,150 KB instead of 8,000 KB-a significant savings.

Structure of the Page Table

- **Hierarchical Paging**

Most modern computer systems support a large logical address space (2^{32} to 2^{64}). In such an environment, the page table itself becomes excessively large. For example, consider a system with a 32-bit logical address space. If the page size in such a system is 4 KB (2^{12}), then a page table may consist of up to 1 million entries ($2^{32} / 2^{12}$). Assuming that each entry consists of 4 bytes, each process may need up to 4MB of physical address space for the page table alone. Clearly, we would not want to allocate the page table contiguously in main memory. One simple solution to this problem is to divide the page table into smaller pieces. To accomplish this division a two-level paging algorithm is used.



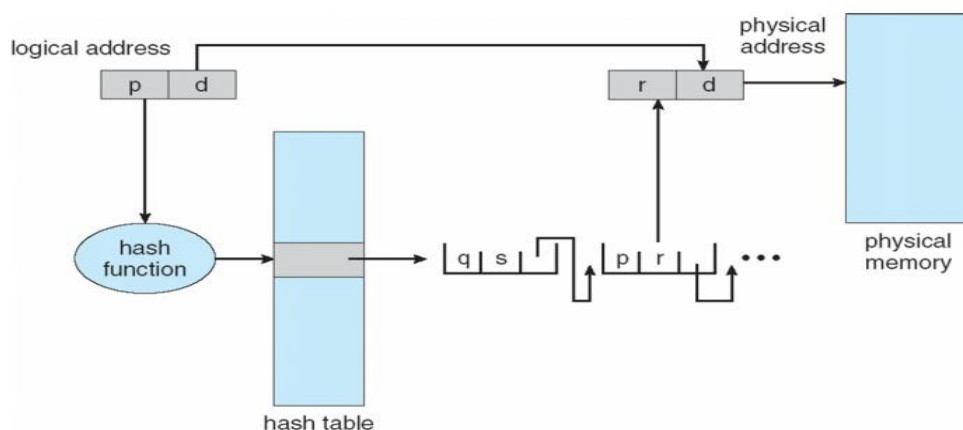
(A two-level page-table scheme)

- **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. This scheme is shown in figure,

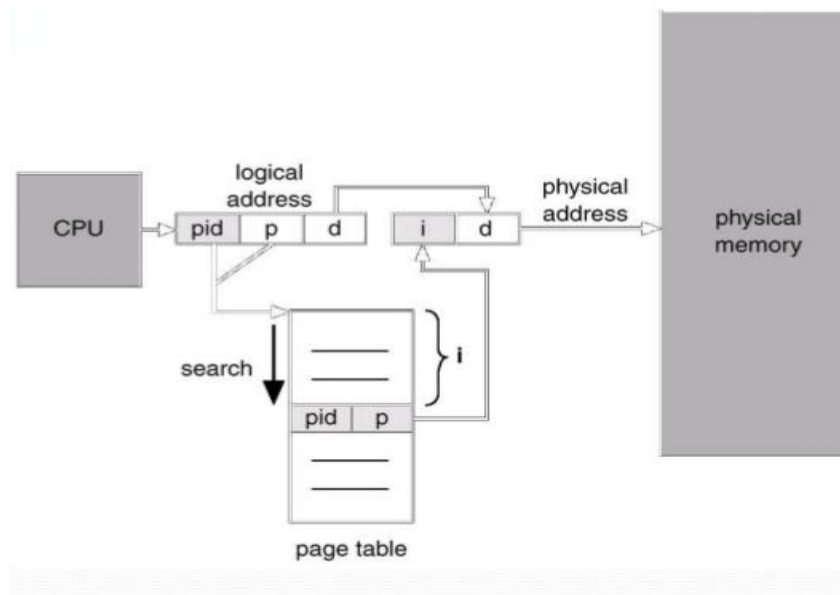
Hashed Page Table



- **Inverted Page Tables**

Usually, each process has an associated page table. The page table has one entry for each page that the process is using (or one slot for each virtual address, regardless of the latter's validity). This table representation is a natural one, since processes reference pages through the pages' virtual addresses. The operating system must then translate this reference into a physical memory address. Since the table is sorted by virtual address, the operating system is able to calculate where in the table the associated physical address entry is located and to use that value directly. One of the drawbacks of this method is that each page table may consist of millions of entries. These tables may consume large amounts of physical memory just to keep track of how other physical memory is being used.

To solve this problem, we can use an *inverted page table*. 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. Figure shows the operation of an inverted page table.



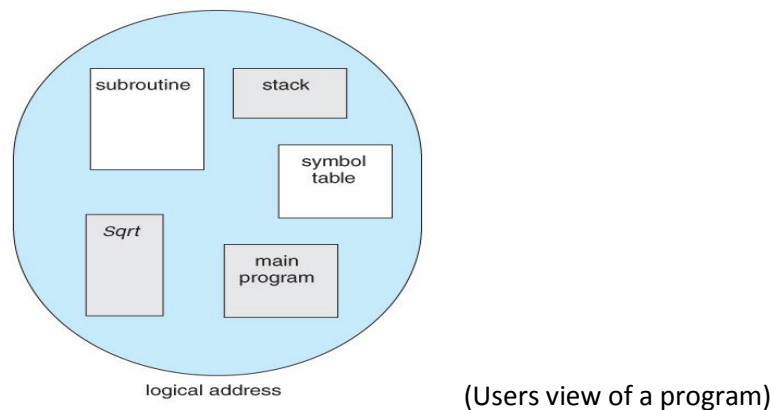
(Inverted page table.)

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.

Segmentation

An important aspect of memory management that became unavoidable with paging is the separation of the user's view of memory from the actual physical memory. The user's view of memory is not the same as the actual physical memory. The user's view is mapped onto physical memory. This mapping allows differentiation between logical memory and physical memory. Users

prefer to view memory as a collection of variable-sized segments, with no necessary ordering among segments.



Segmentation is a memory-management scheme that supports this user 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 user therefore specifies each address by two quantities: a segment name and an offset. (Contrast this scheme with the paging scheme, in which the user specifies only a single address, which is partitioned by the hardware into a page number and an offset, all invisible to the programmer.)

For simplicity of implementation, 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*:

$\langle \text{segment number}, \text{offset} \rangle$

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

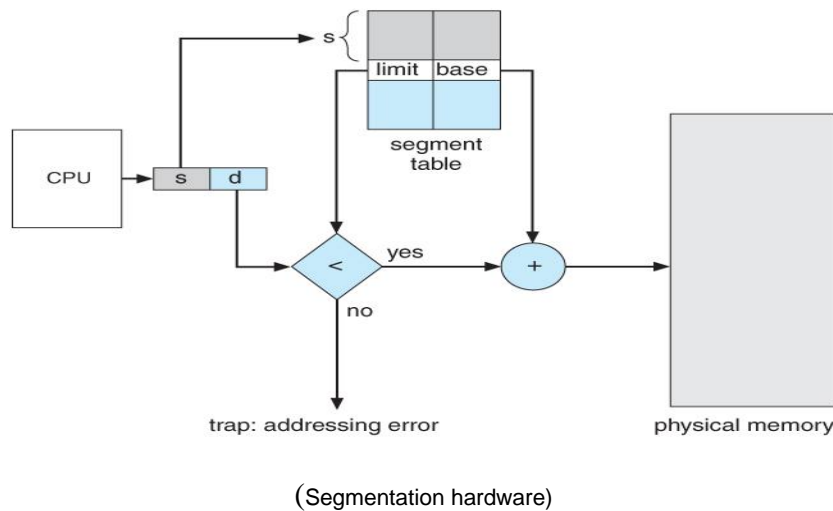
1. The code
2. Global variables
3. The heap, from which memory is allocated
4. The stacks used by each thread
5. The standard C library

Libraries that are linked in during compile time might be assigned separate segments. The loader would take all these segments and assign them segment numbers.

Hardware

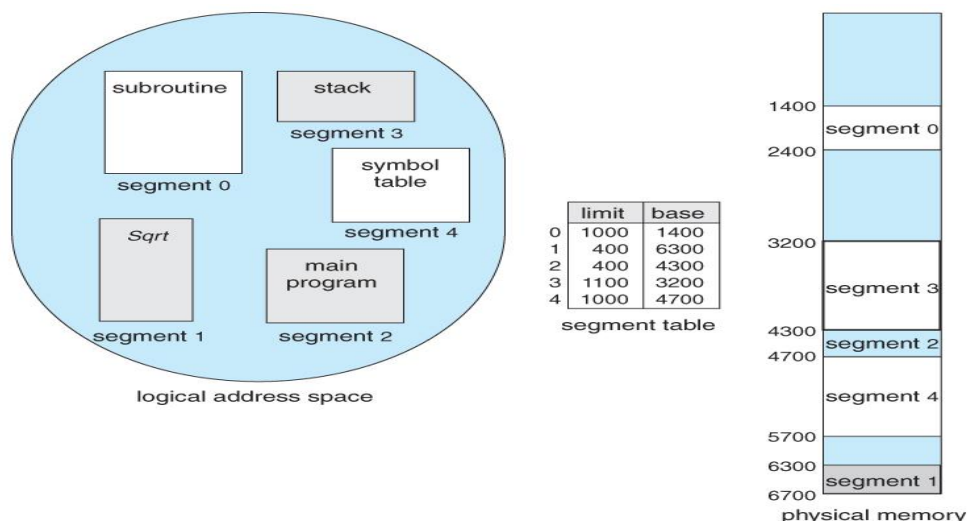
Although the user can now refer to objects in the program by a two-dimensional address, the actual physical memory is still, of course, a one-dimensional sequence of bytes. Thus, we must define an

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 (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.

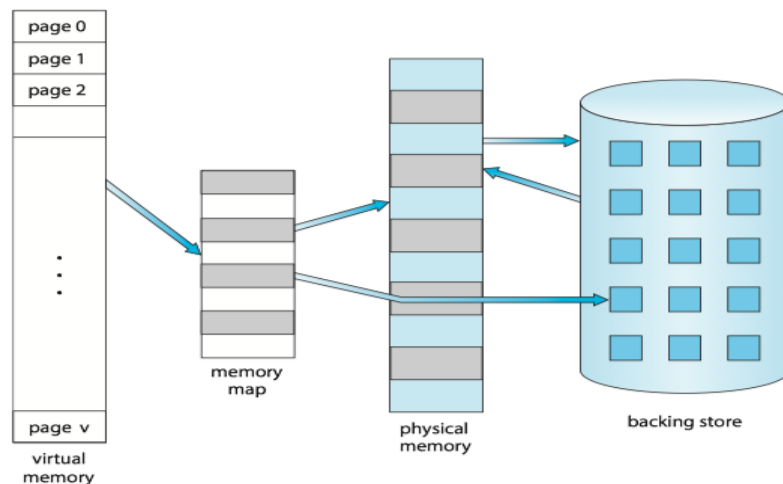
As an example, consider the situation shown in the following figure. There are five segments numbered from 0 through 4. The segments are stored in physical memory as shown. The segment table has a separate entry for each segment, giving the beginning address of the segment in physical memory (or base) 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.



Virtual Memory Management

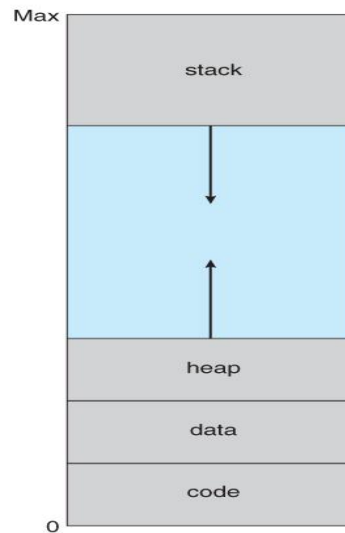
Virtual memory is a technique that allows the execution of processes that are not completely in memory. One major advantage of this scheme is that programs can be larger than physical memory. Further, virtual memory abstracts main memory into an extremely large, uniform array of storage, separating logical memory as viewed by the user from physical memory. This technique frees programmers from the concerns of memory-storage limitations. Virtual memory also allows processes to share files easily and to implement shared memory. In addition, it provides an efficient mechanism for process creation.

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 (figure below). Virtual memory makes the task of programming much easier, because the programmer no longer needs to worry about the amount of physical memory available; the programmer can concentrate instead on the problem to be programmed.



(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 - address0, and exists in contiguous memory, as shown in the following figure. Physical memory may be organized in page frames and that the physical page frames assigned to a process may not be contiguous. It is up to the memory management unit (MMU) to map logical pages to physical page frames in memory.

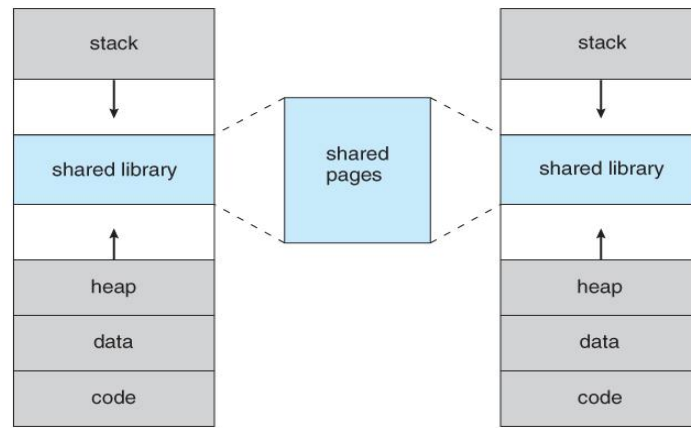


(Virtual address space.)

Note in this figure that allows for the heap to grow upward in memory as it is used for dynamic memory allocation. Similarly, it allows for the stack to grow 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. Using a sparse address space is beneficial because the holes can be filled as the stack or heap segments grow or if it wishes to dynamically link libraries (or possibly other shared objects) during program execution.

In addition to separating logical memory from physical memory, virtual memory allows files and memory to be shared by two or more processes through page sharing. This leads to the following benefits:

- System libraries can be shared by several processes through mapping of the shared object into a virtual address space. Although each process considers the shared libraries to be part of its virtual address space, the actual pages where the libraries reside in physical memory are shared by all the processes (Figure below). Typically, a library is mapped read-only into the space of each process that is linked with it.
- Similarly, virtual memory enables processes to share memory. 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. Processes sharing this region consider it part of their virtual address space, yet the actual physical pages of memory are shared, much as is illustrated in the figure.
- Virtual memory can allow pages to be shared during process creation with the `fork()` system call thus speeding up process creation.

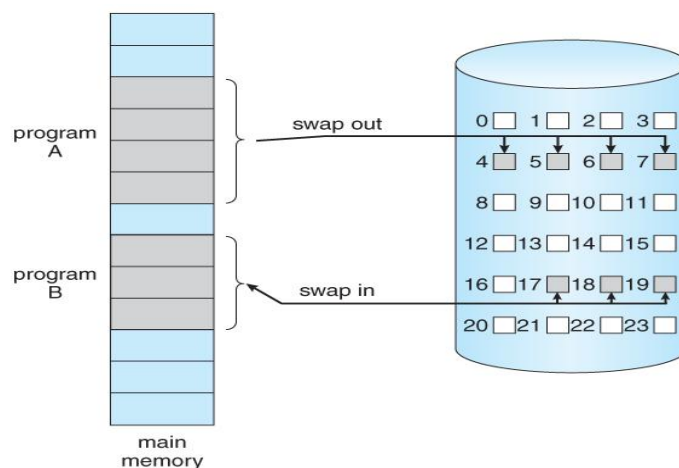


(Shared library using virtual memory.)

Demand Paging

Consider how an executable program might be loaded from disk into memory. One option is to load the entire program in physical memory at program execution time. However, a problem with this approach is that it may not initially need the entire program in memory. Suppose a program starts with a list of available options from which the user is to select. Loading the entire program into memory results in loading the executable code for all options, regardless of whether an option is ultimately selected by the user or not. An alternative strategy is to load pages only as they are needed. This technique is known as *demand paging* and is commonly used in virtual memory systems. With demand-paged virtual memory, pages are only loaded 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 (Figure below) where processes reside in secondary memory (usually a disk). When we want to execute a process, we swap it into memory. A swapper manipulates entire processes, whereas a *page* is concerned with the individual pages of a process.

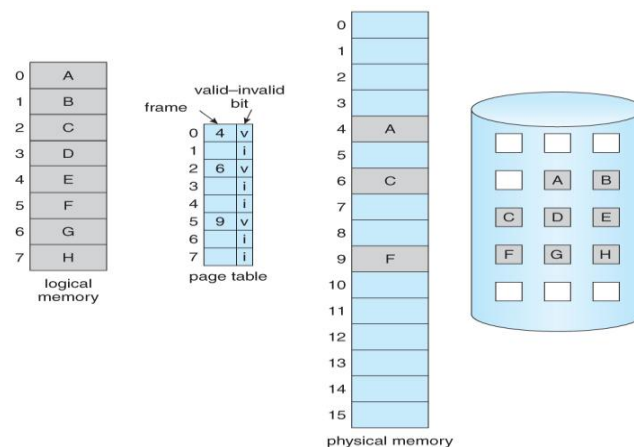


(Transfer of a paged memory to contiguous disk space)

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. Thus, it avoids reading into memory pages that will not be used anyway, decreasing the swap time and the amount of physical memory needed.

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. 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. This situation is depicted in the following figure.

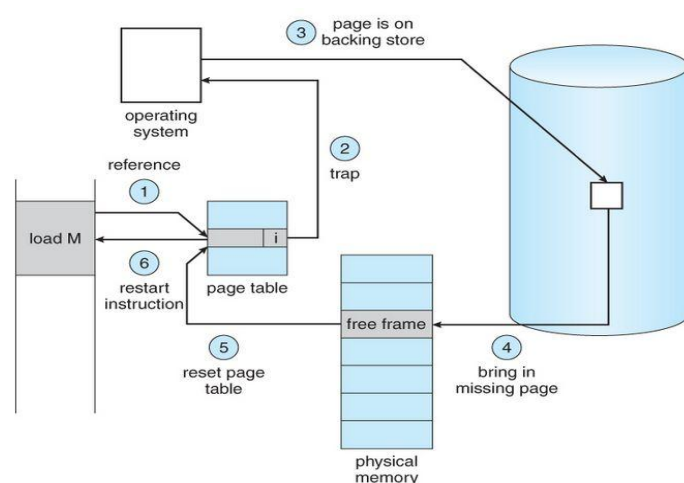


(Page table when some pages are not in main memory)

If the process tries to access a page that was not brought into memory, that page is marked page fault. The paging hardware, in translating the address through the page table, will notice that the invalid bit is set, causing a trap to the operating system. This trap is the result of the operating system's failure to bring the desired page into memory. The procedure for handling this page fault is straightforward.

1. We check an internal table (usually kept with the process control block) 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 (by taking one from the free-frame list, for example).
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.



(Steps in handling a page fault)

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 ($0 \leq p \leq 1$). 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

$$\text{effective access time} = (1 - p) \times ma + p \times \text{page fault time}.$$

To compute the effective access time, we must know how much time is needed to service a page fault. 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

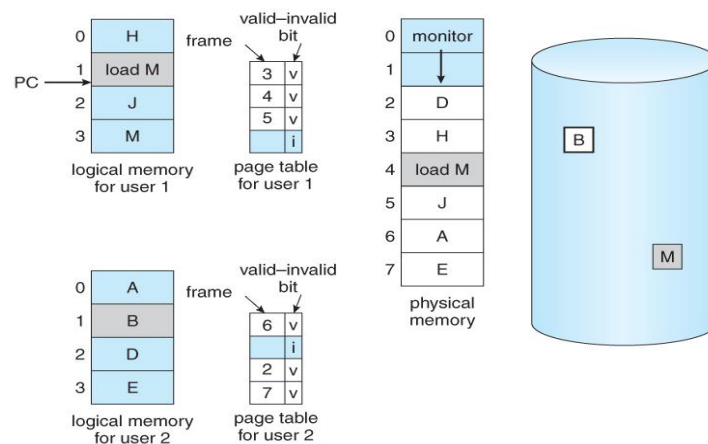
10. Correct the page table and other tables to show that the desired page is now in memory.
11. Wait for the CPU to be allocated to this process again.
12. Restore the user registers, process state, and new page table, and then resume the interrupted instruction.

An additional aspect of demand paging is the handling and overall use of swap space. Disk I/O to swap space is generally faster than that to the file system. It is faster because swap space is allocated in much larger blocks, and file lookups and indirect allocation methods are not used. The system can therefore gain better paging throughput by copying an entire file image into the swap space at process start up and then performing demand paging from the swap space.

Page Replacement

Consider the system memory is not used only for holding program pages. Buffers for I/O also consume a considerable amount of memory. This use can increase the strain on memory-placement algorithms. Deciding how much memory to allocate to I/O and how much to program pages is a significant challenge. Some systems allocate a fixed percentage of memory for I/O buffers, whereas others allow both user processes and the I/O subsystem to compete for all system memory.

Over-allocation of memory manifests itself as follows. While a user process is executing, a page fault occurs. The operating system determines where the desired page is residing on the disk but then finds that there are no free frames on the free-frame list; all memory is in use. This shows in the following figure.



(Need for page replacement)

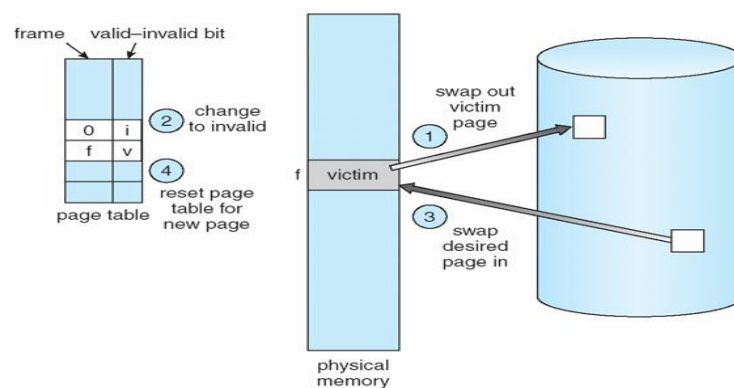
The operating system has several options at this point. It could terminate the user process. However, demand paging is the operating system's attempt to improve the computer system's utilization and throughput. The operating system could instead swap out a process, freeing all its frames and reducing the level of multiprogramming.

• Basic Page Replacement

Page replacement takes the following approach. If no frame is free, we find one that is not currently being used and free it. We can free a frame by writing its contents to swap space and changing the page table (and all other tables) to indicate that the page is no longer in memory (Figure below). We

can now use the freed frame to hold the page for which the process faulted. We modify the page-fault service routine to include page replacement:

1. Find the location of the desired page on the disk.
2. Find a free frame:
 - a. If there is a free frame, use it.
 - b. If there is no free frame, use a page-replacement algorithm to select a *victim frame*.
 - c. Write the victim frame to the disk; change the page and frame tables accordingly.
3. Read the desired page into the newly freed frame; change the page and frame tables.
4. Restart the user process.



(Page replacement)

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.

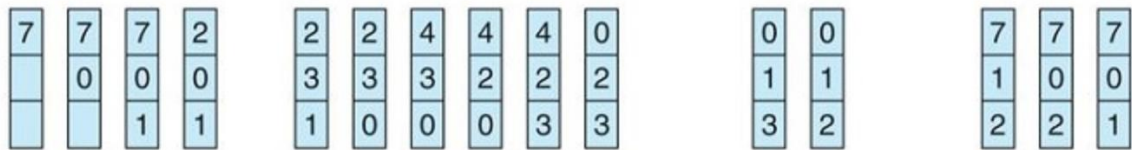
• FIFO Page Replacement

The simplest page-replacement algorithm is a first-in, first-out (FIFO) algorithm. A FIFO replacement algorithm associates with each page the time when that page was brought into memory. When a page must be replaced, the oldest page is chosen. Notice that it is not strictly necessary to record the time when a page is brought in. We can create a FIFO queue to hold all pages in memory. We replace the page at the head of the queue. When a page is brought into memory, we insert it at the tail of the queue.

For our example reference string, our three frames are initially empty. The first three references (7, 0, 1) cause page faults and are brought into these empty frames. The next reference (2) replaces page 7, because page 7 was brought in first. Since 0 is the next reference and 0 is already in memory, we have no fault for this reference. The first reference to 3 results in replacement of page 0, since reference string it is now first in line. Because of this replacement, the next reference, to 0, will fault. Page 1 is then replaced by page 0. This process continues as shown in the following figure. Every time a fault occurs, we show which pages are in our three frames. There are fifteen faults altogether.

reference string

7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 7 0 1



page frames

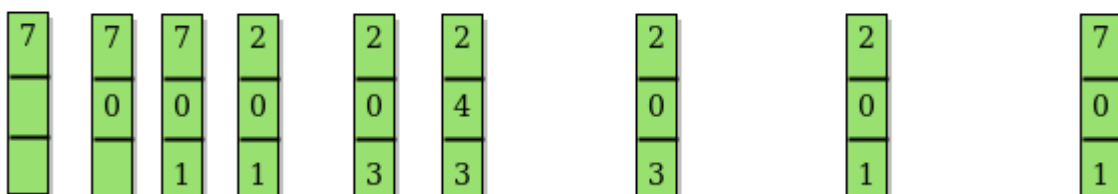
The FIFO page-replacement algorithm is easy to understand and program. However, its performance is not always good. On the one hand, the page replaced may be an initialization module that was used a long time ago and is no longer needed. On the other hand, it could contain a heavily used variable that was initialized early and is in constant use. Even if we select for replacement a page that is in active use, everything still works correctly. After we replace an active page with a new one, a fault occurs almost immediately to retrieve the active page. Some other page must be replaced to bring the active page back into memory. Thus, a bad replacement choice increases the page-fault rate and slows process execution. It does not, however, cause incorrect execution.

• Optimal Page Replacement

Use of this page-replacement algorithm guarantees the lowest possible page fault rate for a fixed number of frames. It simply specifies, replace the page that will not be used for the longest period of time.

For example, on our sample reference string, the optimal page-replacement algorithm would yield nine page faults, as shown in the following figure. The first three references cause faults that fill the three empty frames. The reference to page 2 replaces page 7, because page 7 will not be used until reference 18, whereas page 0 will be used at 5, and page 1 at 14. The reference to page 3 replaces page 1, as page 1 will be the last of the three pages in memory to be referenced again. With only nine page faults, optimal replacement is much better than a FIFO algorithm, which results in fifteen faults. (If we ignore the first three, which all algorithms must suffer then optimal replacement is twice as good as FIFO replacement.) In fact, no replacement algorithm can process this reference string in three frames with fewer than nine faults.

7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 7 0 1

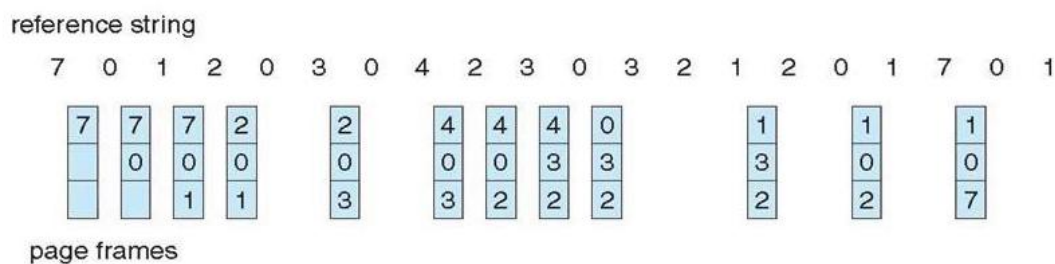


Unfortunately, the optimal page-replacement algorithm is difficult to implement, because it requires future knowledge of the reference string. As a result, the optimal algorithm is used mainly for comparison studies. For instance, it may be useful to know that, although a new algorithm is not optimal it is within 12.3 percent of optimal at worst and within 4.7 percent on average.

• LRU Page Replacement

The key distinction between the FIFO and OPT algorithms (other than looking backward versus forward in time) is that the FIFO algorithm uses the time when a page was brought into memory, whereas the optimal algorithm uses the time when a page is to be *used*. If we use the recent past as an approximation of the near future, then we can replace the page that *has not been used* for the longest period of time. This approach is the *least-recently-used (LRU) algorithm*.

The result of applying LRU replacement to our example reference string is shown in the following figure. The LRU algorithm produces twelve faults. Notice that the first five faults are the same as those for optimal replacement. When the reference to page 4 occurs, however, LRU replacement sees that, of the three frames in memory, page 2 was used least recently. Thus, the LRU algorithm replaces page 2, not knowing that page 2 is about to be used. When it then faults for page 2, the LRU algorithm replaces page 3, since it is now the least recently used of the three pages in memory. Despite these problems, LRU replacement with twelve faults is much better than FIFO replacement with fifteen.



The LRU policy is often used as a page-replacement algorithm and is considered to be good. The major problem is how to implement LRU replacement. An LRU page-replacement algorithm may require substantial hardware assistance. The problem is to determine an order for the frames defined by the time of last use.

• LRU-Approximation Page Replacement

Few computer systems provide sufficient hardware support for true LRU page replacement. Some systems provide no hardware support, and other page replacement algorithms (such as a FIFO algorithm) must be used. Many systems provide some help, however, in the form of a *reference bit*. The reference bit for a page is set by the hardware whenever that page is referenced (either a read or a write to any byte in the page). Reference bits are associated with each entry in the page table.

Initially, all bits are cleared (to 0) by the operating system. As a user process executes, the bit associated with each page referenced is set (to 1) by the hardware. After some time, we can determine which pages have been used and which have not been used by examining the reference bits, although we do not know the *order* of use. This information is the basis for many page-replacement algorithms that approximate LRU replacement.

• Counting-Based Page Replacement

Here keep a counter of the number of references that have been made to each page and develop the following two schemes.

- ✓ The least frequently used (LFU) page-replacement algorithm requires that the page with the smallest count be replaced. The reason for this selection is that an actively used page should have a large reference count. A problem arises, however, when a page is used heavily during the initial phase of a process but then is never used again. Since it was used heavily, it has a large count and remains in memory even though it is no longer needed. One solution is to shift the counts right by 1 bit at regular intervals, forming an exponentially decaying average usage count.
- ✓ The most frequently used (MFU) page-replacement algorithm is based on the argument that the page with the smallest count was probably just brought in and has yet to be used.

• Page-Buffering Algorithms

Systems commonly keep a pool of free frames. When a page fault occurs, a victim frame is chosen as before. However, the desired page is read into a free frame from the pool before the victim is written out. This procedure allows the process to restart as soon as possible, without waiting for the victim page to be written out. When the victim is later written out, its frame is added to the free-frame pool.

Another modification is to keep a pool of free frames but to remember which page was in each frame. Since the frame contents are not modified when a frame is written to the disk, the old page can be reused directly from the free-frame pool if it is needed before that frame is reused. No I/O is needed in this case. When a page fault occurs, we first check whether the desired page is in the free-frame pool. If it is not, we must select a free frame and read into it. This technique is used in the VAX/VMS system along with a FIFO replacement algorithm.