OPERATING SYSTEMS

Unit Objectives:

1. To provide a detailed description of various ways of organizing memory hardware.
2. To discuss various memory-management techniques, including paging and segmentation.
3. To describe the benefits of a virtual memory system.
4. To explain the concepts of demand paging, page-replacement algorithms, and allocation of page frames.
5. To discuss the principle of the working-set model.

Unit -5

Memory Management Strategies: Background; Swapping; Contiguous memory allocation; Paging; Structure of page table; Segmentation. Virtual Memory Management: Background; Demand paging; Copy-on-write; Page replacement; Allocation of frames; Thrashing.

Memory Management Strategies
Background

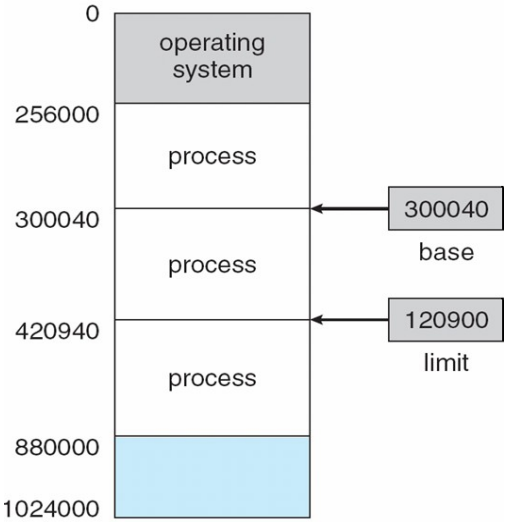
- Memory consists of a large array of words or bytes, each with its own address. The CPU fetches instructions from memory according to the value of the program counter.
- A typical instruction-execution cycle, for example, first fetches an instruction from memory.
- The instruction is then decoded and 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. We can ignore how a program generates a memory address.

Basic Hardware

- Main memory and the registers built into the processor 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.
- Registers that are built into the CPU are generally accessible within one cycle of the CPU clock. Completing a memory access may take many cycles of the CPU clock. The remedy is to add fast memory between the CPU and main memory called cache.
- The operating system must be protected from user processes and, in addition, user processes must be protected from one another. This protection must be provided by the hardware.
- We can provide this protection by using two registers, usually a **base** and a **limit**, as illustrated below. A pair of **base** and **limit** registers define the logical address space.

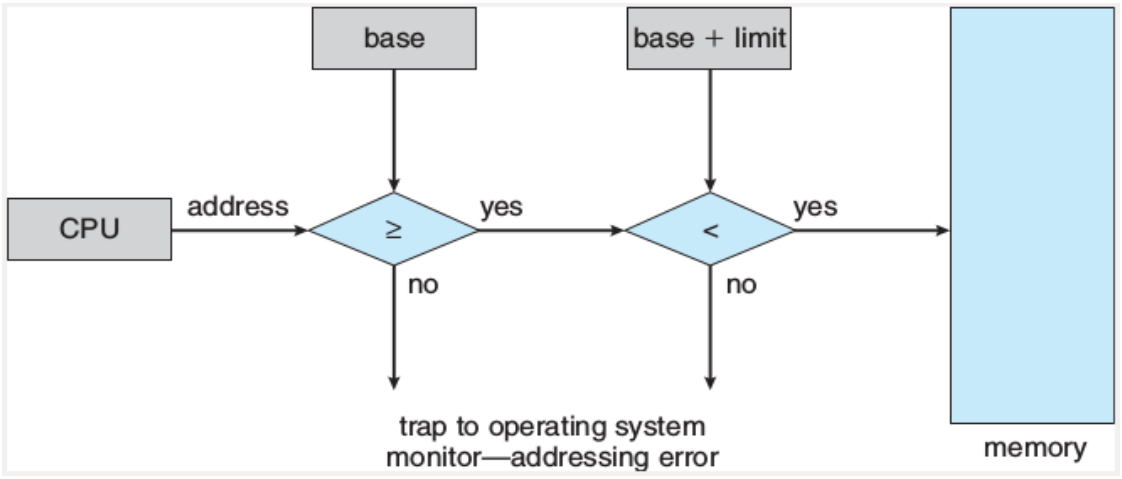
Base and Limit Registers

- 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 (inclusive).



Hardware Address Protection

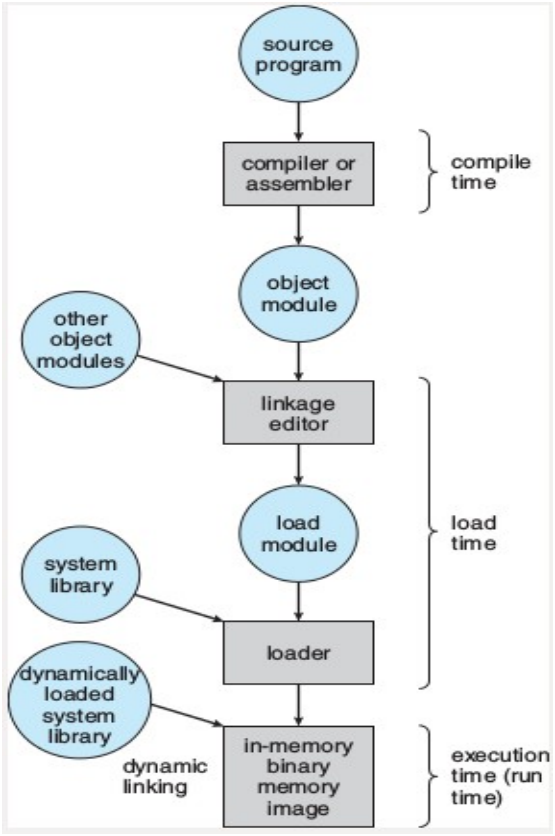
- Protection of memory space is accomplished by having the CPU hardware compare every address generated in user mode with the registers.
- Any attempt by a program executing in user mode to access operating-system memory or other users' memory results in a trap to the operating system, which treats the attempt as a fatal error.
- The base and limit registers can be loaded only by the Operating system.



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.
- Depending on the memory management in use, the process may be moved between disk and memory during its execution. The processes on the disk that are waiting to be brought into memory for execution form the **input queue**.
- Most systems allow a user process to reside in any part of the physical memory. Thus, although the address space of the computer starts at 00000, the first address of the user process need not be 00000.
- A user program will go through several steps-some of which may be optional-before being executed. Addresses may be represented in different ways during these steps.
- Addresses in the source program are generally symbolic. A compiler will typically bind these symbolic addresses to relocatable addresses (such as "14 bytes from the beginning of this module"). The linkage editor or loader will in turn bind the relocatable addresses to absolute addresses (such as 74014). Each binding is a mapping from one address space to another.

Multistep Processing of a User Program



Binding of Instructions and Data to Memory

Address binding of instructions and data to memory addresses can happen at three different stages.

Compile time. If you know at compile time where the process will reside in memory, then absolute code can be generated. For example, if you know that a user process will reside starting at location R, then the compiler generated code will start at that location and extend up from there.

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.

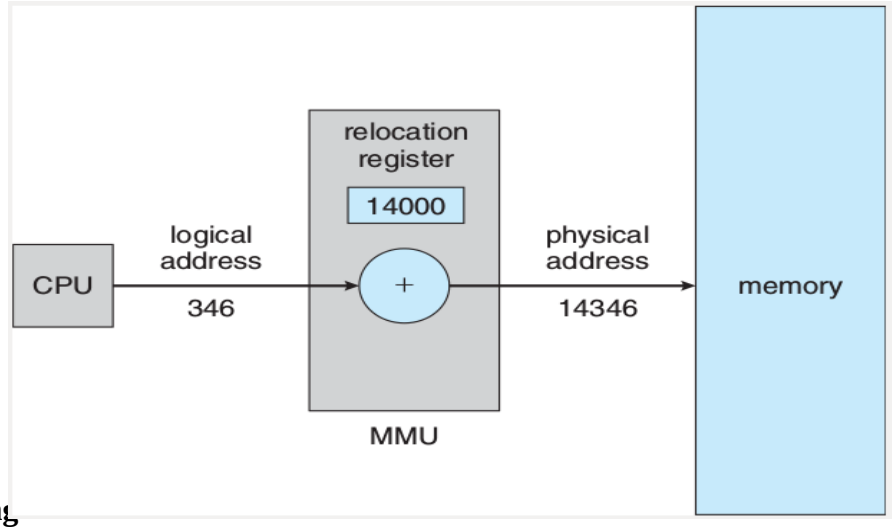
Execution time. If the process can be moved during its execution from one memory segment to another, then binding must be delayed until run time.

Logical vs. 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.
- 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**.

Memory-Management Unit (MMU)

- The run-time mapping from virtual(logical) to physical addresses is done by a hardware device called the Memory management unit(MMU).
- We illustrate this mapping with a simple MMU scheme that is a generalization of the base-register scheme.
- 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.
- 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.
- The user program never sees the real physical addresses. We now have two different types of addresses: logical addresses (in the range 0 to max) and physical addresses (in the range R+0 to R+max for a base value R).
- The user generates only logical addresses and thinks that the process runs in locations 0 to max. However, these logical addresses must be mapped to physical addresses before they are used.



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. Then control is passed to the newly loaded routine.
- The advantage of dynamic loading is that an unused routine is never loaded. This method is particularly useful when large amounts of code are needed to handle infrequently occurring cases, such as error routines.
- Dynamic loading results in better memory-space utilization.

Dynamic Linking

- **Static linking** – in which system language libraries and program code are combined by the loader into the binary program image.
- **Dynamic linking**, in contrast, is similar to dynamic loading. Here linking, rather than loading, is postponed until execution time.
- This feature is usually used with system libraries, such as language subroutine libraries. Without this facility, each program on a system must include a copy of its language library in the executable image.
- 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.
- This feature can be extended to library updates. A library may be replaced by a new version, and all programs that reference the library will automatically use the new version.
- One copy of library is shared by multiple programs. This system is also known as **shared libraries.**
- Unlike dynamic loading, dynamic linking generally requires help from the operating system.
- If the processes in memory are protected from one another, then the operating system is the only entity that can check to see whether the needed routine is in another process's memory space or that can allow multiple processes to access the same memory addresses.

Swapping

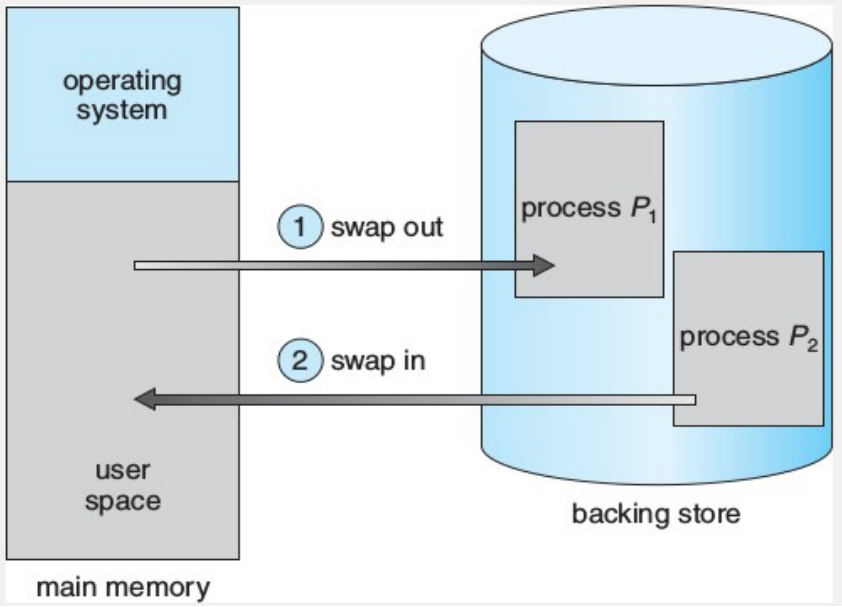
- A process must be in memory to be executed. A process, however, can be swapped temporarily out of memory to a backing store and then brought back into memory for continued execution.
- For example, assume a multiprogramming environment with a round-robin CPU-scheduling algorithm. When a quantum expires, 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.
- Swapping variant used for priority-based scheduling algorithms; lower-priority process is swapped out so that newly arrived higher-priority process can be loaded and executed. This variant of swapping is called **roll out, roll in**.
- Normally, a process that is swapped out will be swapped back into the same memory space it occupied previously. If binding is done at assembly or load time, then the process cannot be

easily moved to a different location. If execution-time binding is being used, however, then a process can be swapped into a different memory space.

- Swapping requires 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. To get an idea of the context-switch time, let us assume that the user process is 100 MB in size and the backing store is a standard hard disk with a transfer rate of 50MB per second. The actual transfer of the 100-MB process to or from main memory takes

  100MB/50MB per second= 2 seconds.
- Assuming an average latency of 8 milliseconds, the swap time is 2008 milliseconds. Since we must both swap out and swap in, the total swap time is about 4016 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.

Schematic View of Swapping



- Processes with Pending I/O – cannot be swapped out as I/O would occur to wrong process.
- Assume that I/O operation of process P1 is queued because the device is busy. If we swap out process P1 and swap in process P2, the I/O operation might attempt to use memory that now belongs to process P2.
- Two solutions to the problem:
  - Never swap a process with pending I/O.
  - Execute I/O operations only into the operating system buffers. Transfer between buffers and process memory then occur only when the process is swapped in.

- Standard swapping requires too much swapping time and provides too little execution time. Modified versions of swapping are found on many systems (i.e., UNIX, Linux, and Windows).
  - Swapping normally disabled.
  - Started if more than threshold amount of memory allocated.
  - Disabled again once memory demand reduced below threshold.

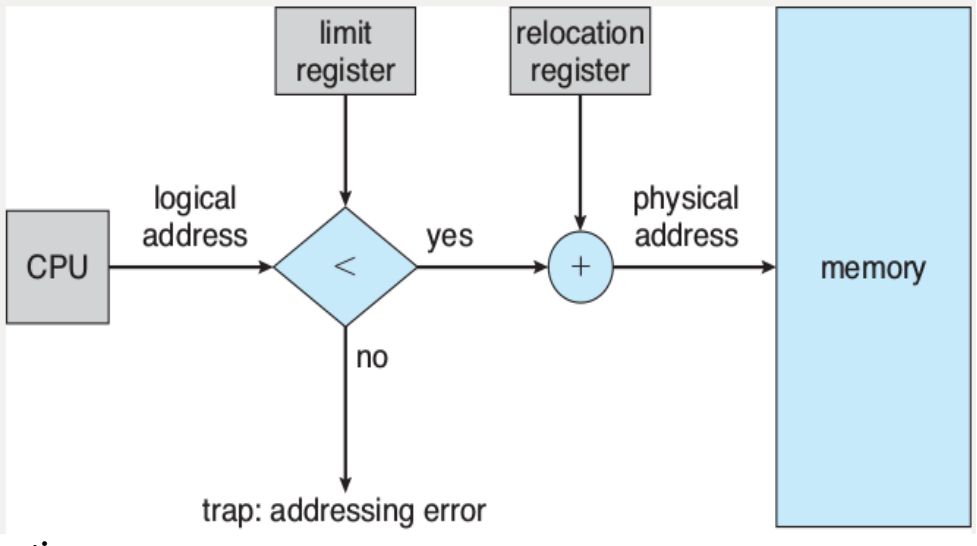
Contiguous Memory Allocation

- 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.
- 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.
- We usually want several user processes to reside in memory at the same time. We therefore need to consider how to allocate available memory to the processes that are in the input queue waiting to be brought into memory.
- In contiguous memory allocation, each process is contained in a single contiguous section of memory.

Memory Mapping and Protection

- 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.
- 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.

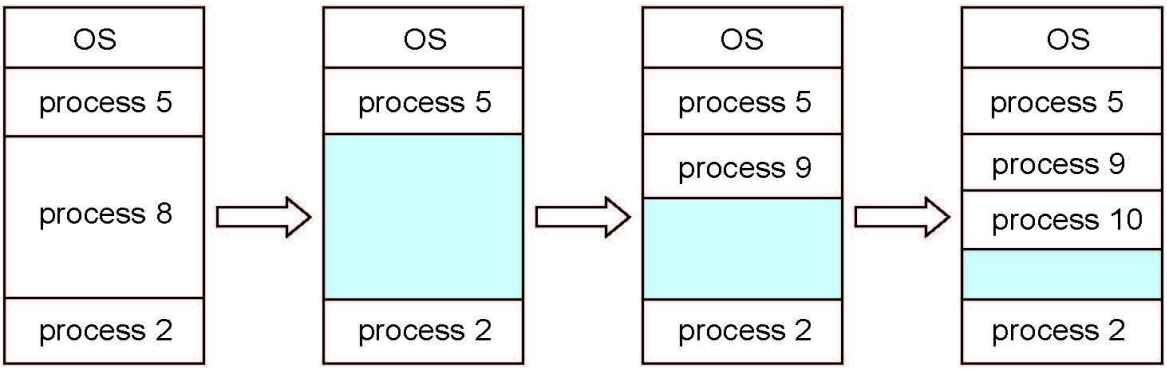
Hardware Support for Relocation and Limit Registers



Memory Allocation

- One of the simplest methods for allocating memory is to divide memory into several fixed-sized partitions . Each partition may contain exactly one process.

- In this multiple partition method, when a partition is free, a process is selected from the input queue and is loaded into the free partition. When the process terminates, the partition becomes available for another process.
- In the **variable-sized 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**.



- 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.
- 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.
- At any given time, then, we have a list of available block sizes(holes) and an input queue. Memory is allocated to processes until finally, the memory requirements of the next process cannot be satisfied.
- 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.

Dynamic Storage-Allocation Problem

How to satisfy a request of size n from a list of free holes?

First fit. Allocate the first hole that is big enough. Searching can start either at the beginning of the set of holes or at the location where the previous first-fit search ended. We can stop searching as soon as we find a free hole that is large enough.

Best fit. Allocate the smallest hole that is big enough. We must search the entire list, unless the list is ordered by size. This strategy produces the smallest leftover hole.

Worst fit. Allocate the largest hole. Again, we must search the entire list, unless it is sorted by size. This strategy produces the largest leftover hole, which may be more useful than the smaller leftover hole from a best-fit approach.

Fragmentation

- Both the first-fit and best-fit strategies for memory allocation suffer from external fragmentation. As processes are loaded and removed from memory, the free memory space is broken into little pieces.

- **External fragmentation** exists when there is enough total memory space to satisfy a request but the available spaces are not contiguous; storage is fragmented into a large number of small holes.
- 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 avoid this problem is to break the physical memory into fixed-sized blocks. With this approach, the memory allocated to a process may be slightly larger than the requested memory. The difference between these two numbers is **internal fragmentation**-unused memory that is 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.
- Another possible solution to the external-fragmentation problem is to permit the logical address space of the processes to be noncontiguous, thus allowing a process to be allocated physical memory wherever such memory is available.

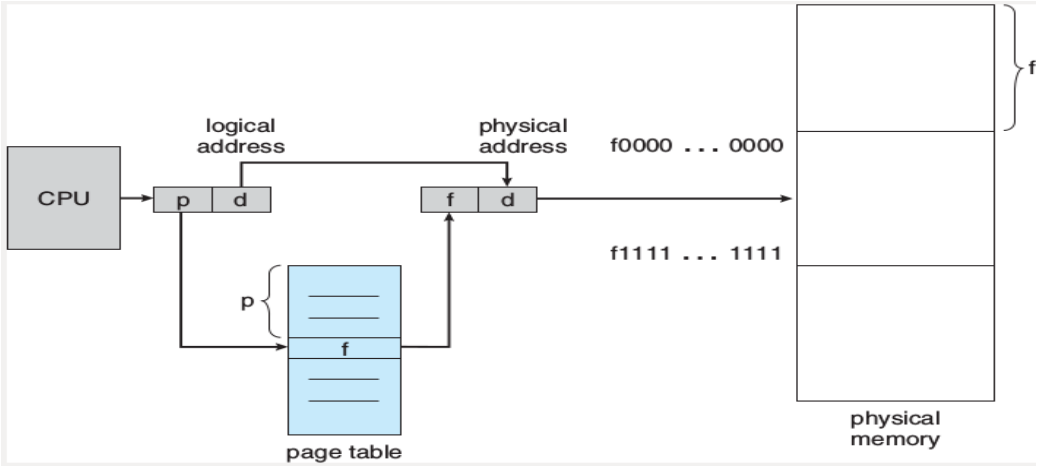
Paging

Paging is a memory-management scheme that permits the physical address space of a process to be noncontiguous. Paging avoids external fragmentation and the need for compaction.

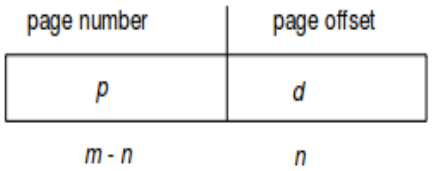
Basic Method

- 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 same size as the memory 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 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.

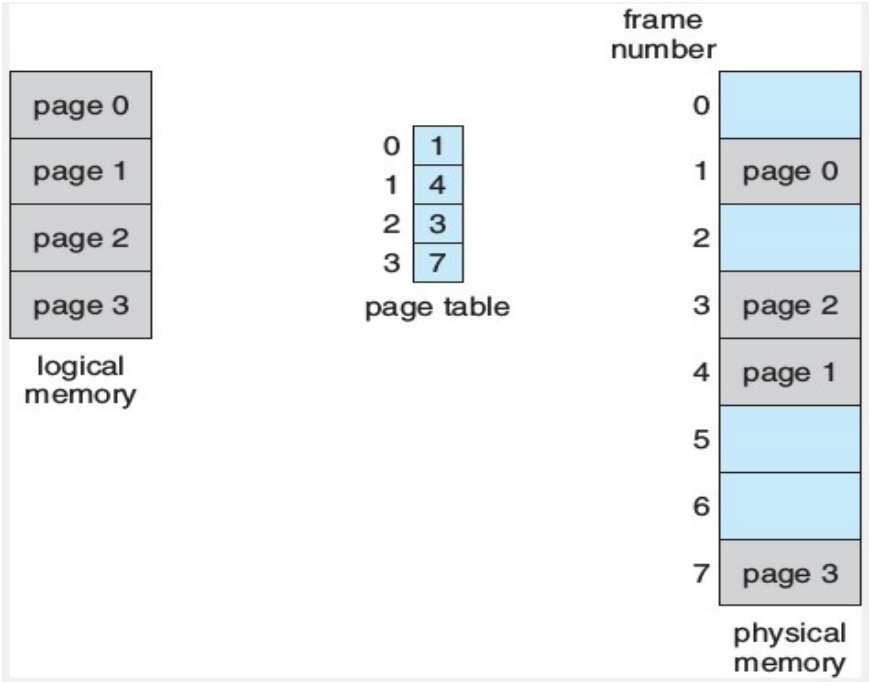
Paging Hardware



- 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:

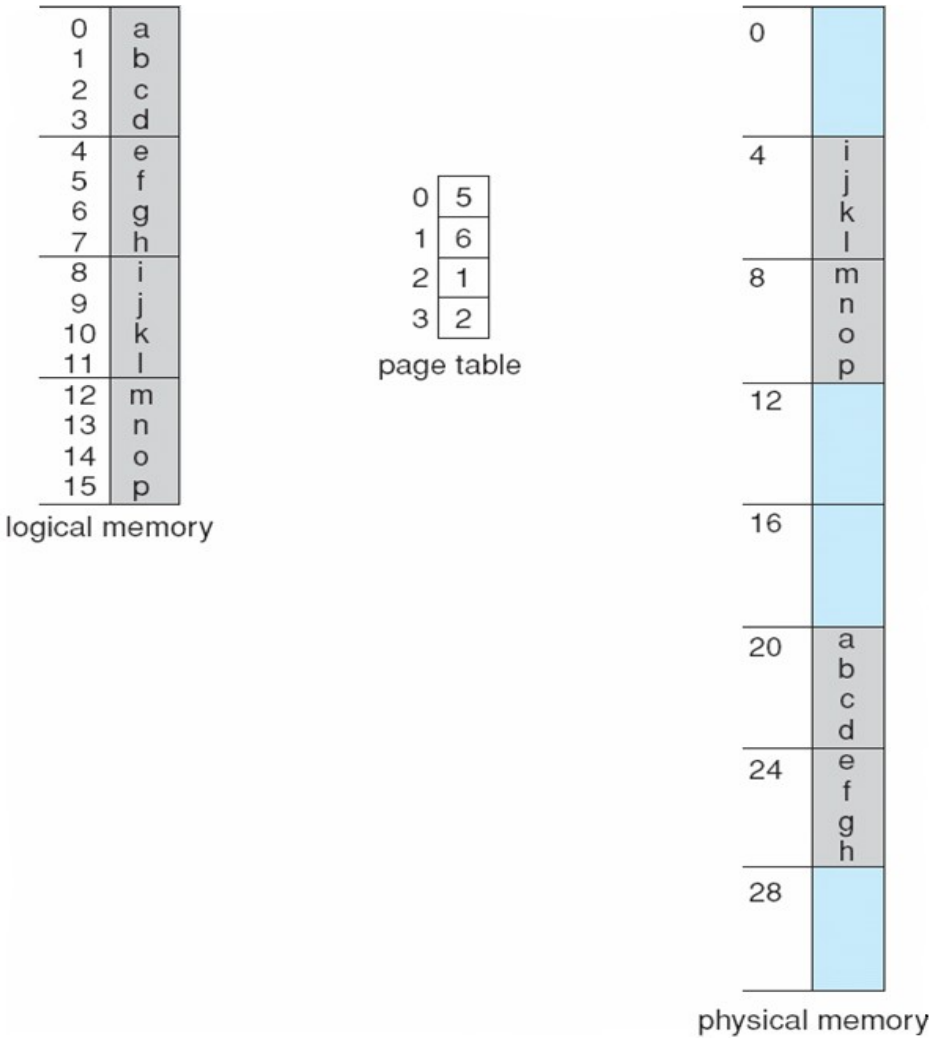


- where p is an index into the page table and d is the displacement within the page.



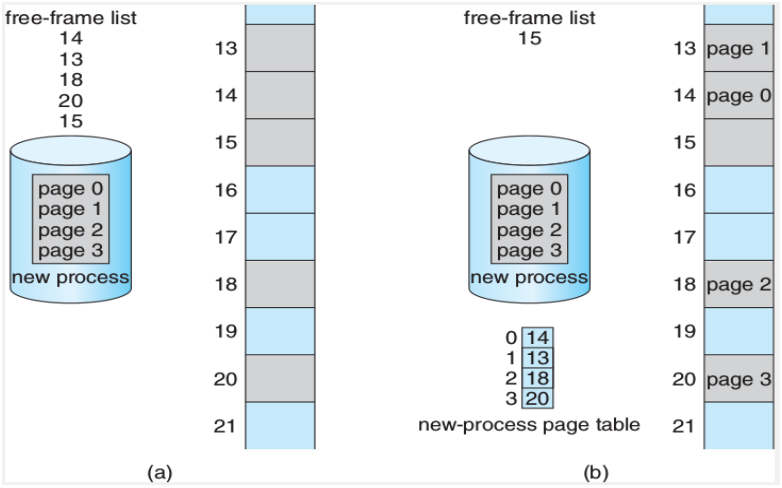
Paging Example

- 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).
- Logical address 0 is page 0, offset 0. Indexing into the page table, we find that page 0 is in frame 5. Thus, logical address 0 maps to physical address 20 [= (5 x 4) + 0].
- Logical address 3 (page 0, offset 3) maps to physical address 23 [= (5 x 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 x 4) + 0].
- Logical address 13 maps to physical address 9.



- When we use a paging scheme, we have no external fragmentation, However, we may have some internal fragmentation.
- If the memory requirements of a process do not happen to coincide with page boundaries, the last frame allocated may not be completely full.
- 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.
- In the worst case, a process would need n pages plus 1 byte. It would be allocated n + 1 frames.
- Usually, each page-table entry is 4 bytes long, but that size can vary as well. A 32-bit entry can point to one of 232 physical page frames. If frame size is 4 KB, then a system with 4-byte entries can address 244 bytes (or 16 TB) of physical memory.
- 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. Frame numbers are put in the page table for this process.

Free Frames



- An important aspect of paging is the clear separation between the user's view of memory and the actual physical memory.
- User program views memory as one single space, containing only this program. In fact user program is scattered throughout the physical memory, which also holds other programs.
- The mapping of logical addresses to physical is hidden from the user and is controlled by the operating system.
- The operating system maintains a copy of the page table for each process. This copy is used to translate logical addresses to physical addresses.
- Operating system maintains a data structure called a **Frame table**: Has one entry for each physical frame, indicating whether the frame is free or allocated and, if allocated, to which page of which process.

Hardware Support

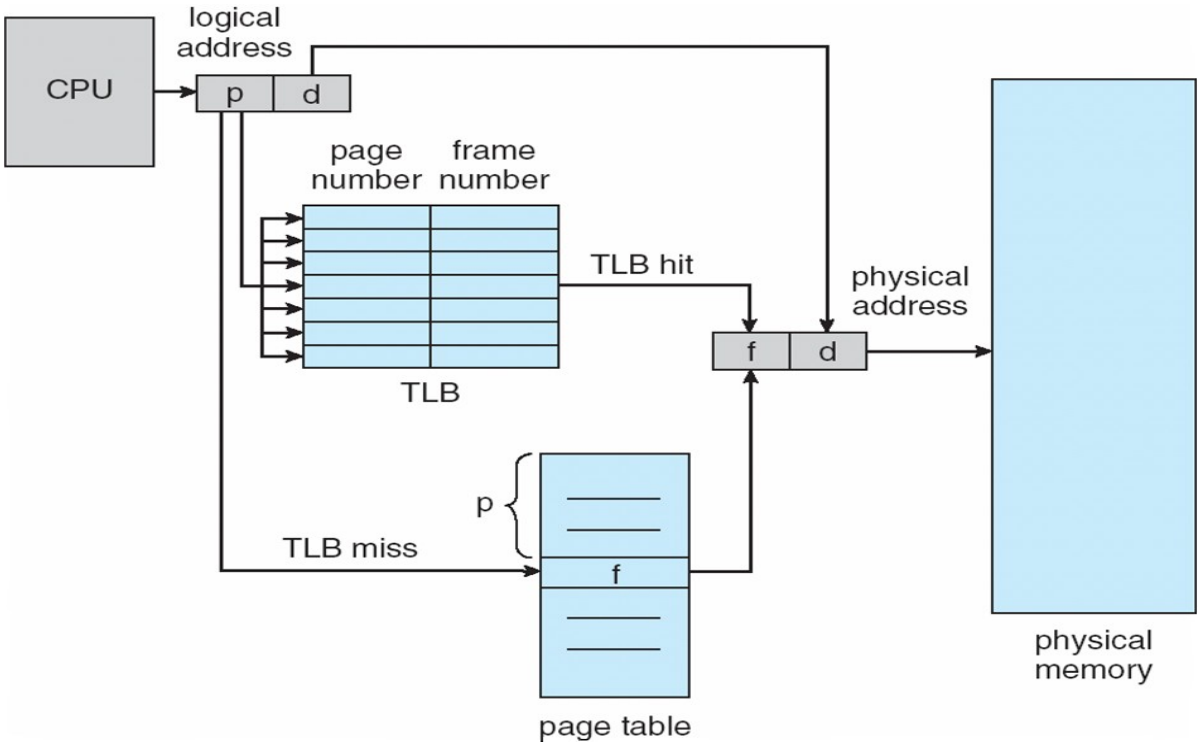
- The hardware implementation of the page table can be done in several ways. In the simplest case, the page table is implemented as a set of dedicated registers. The CPU dispatcher reloads these registers, just as it reloads the other registers.
- The use of registers for the page table is satisfactory if the page table is reasonably small. Most contemporary computers, however, allow the page table to be very large.
- So, the page table is kept in main memory, and a page-table base register (PTBR) points to 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. In this scheme every data/instruction access requires two memory accesses, one for the page table and one for the data/instruction.

Translation look-aside buffer(TLB)

- The standard solution to this problem is to use a special, small, fast-lookup hardware cache, called an associative memory or **Translation look-aside buffer(TLB)**.
- 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.

- 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.
- 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.
- 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.
- An ASID allows TLB to contain entries for several different processes simultaneously.

Paging Hardware With TLB



Effective Access Time

- 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:

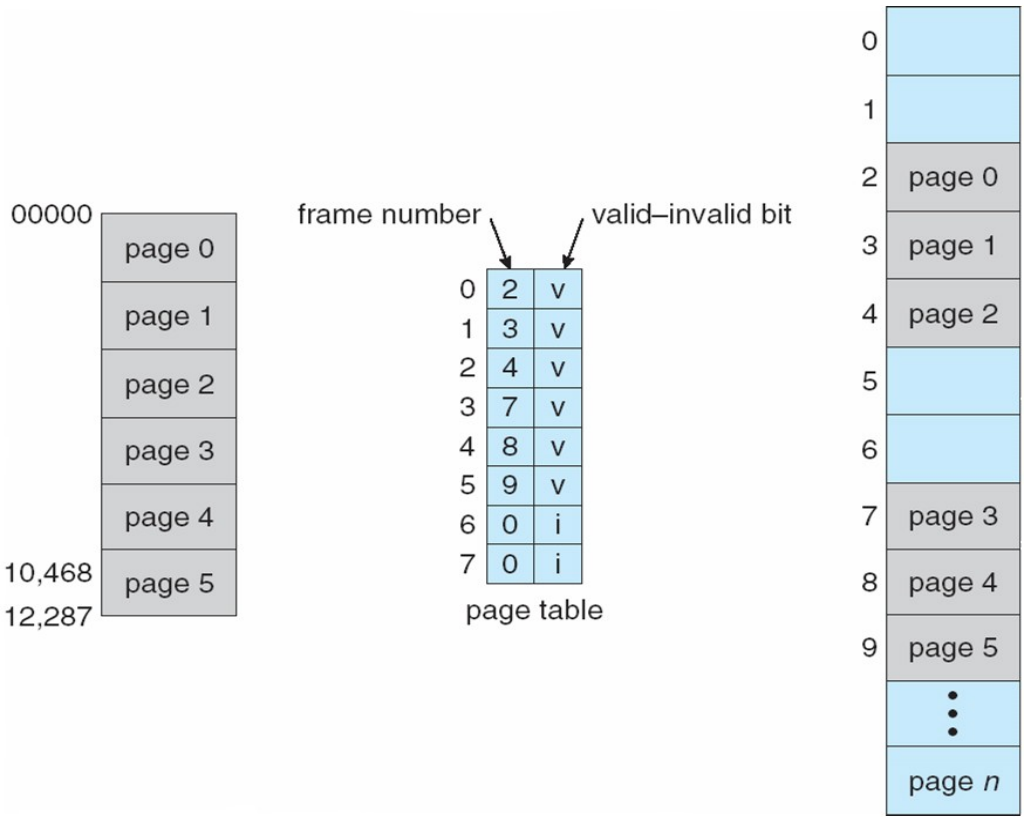
effective access time = 0.80 x 120 + 0.20 x 220

= 140 nanoseconds.

Memory 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.
- 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 page. When the bit is set to"invalid," the page is not in the process's logical address space.
- In a system with 14-bit address space (0 to 16383), we have a program that should use only addresses 0 to 10468.
- 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.
- References to page 5 are classified as valid, so accesses to addresses up to 12287 are valid. This problem is a result of 2KB page size and reflects the internal fragmentation of paging.

Valid (v) or Invalid (i) Bit In A Page Table

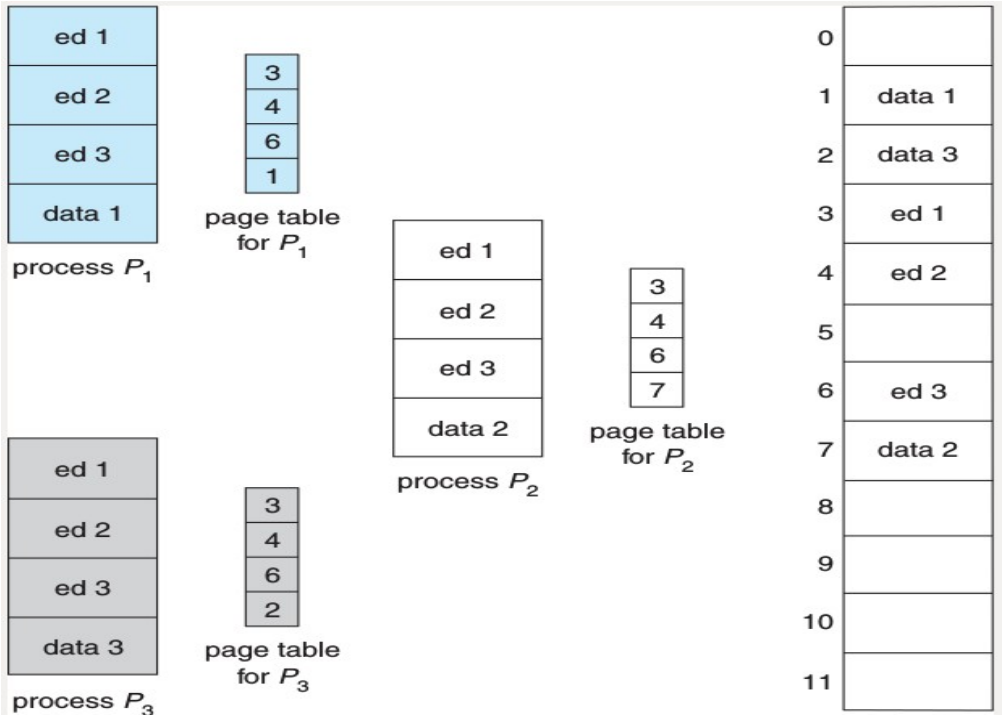


Shared Pages

- An advantage of paging is the possibility of sharing common code. 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 **pure code**(or reentrant code), however, it can be shared, as shown below. Here we see a three-page editor-each page 50 KB in size being shared among three processes. Each process has its own data page.
- Pure code never changes during execution. Thus, two or more processes can execute the same code at the same time. 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.
- Other heavily used programs can also be shared -compilers, window systems, run-time libraries, database systems, and so on.

Shared Pages Example

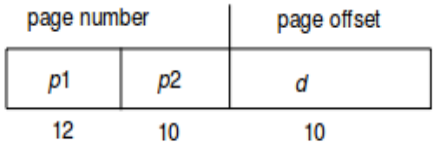


Structure of the Page Table

- 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.
- We would not want to allocate page table contiguously in main memory. One simple solution to this problem is to divide the page table into smaller pieces. We can accomplish this division in several ways.

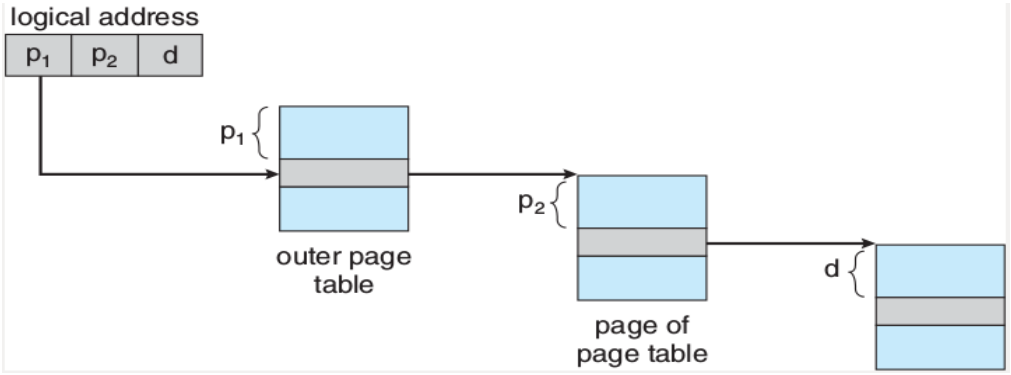
Hierarchical Page Tables

- One way is to use a **two-level paging algorithm**, in which the page table itself is also paged.
- For example, consider again the system with a 32-bit logical address space and a page size of 1 KB. A logical address is divided into a page number consisting of 22 bits and a page offset consisting of 10 bits. Because we page the page table, the page number is further divided into a 12-bit page number and a 10-bit page offset. Thus, a logical address is as follows:

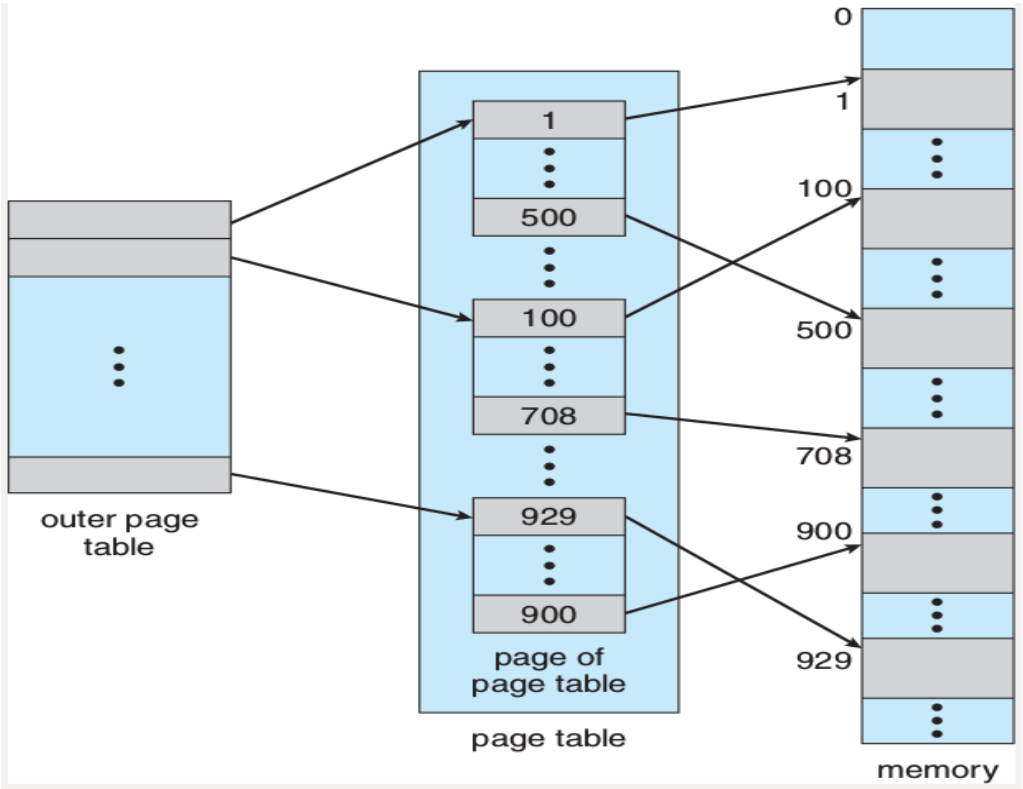


- where P_1 is an index into the outer page table and P_2 is the displacement within the page of inner the page table. The address-translation method for this architecture is shown below. Because address translation works from the outer page table inward, this scheme is also known as a **forward mapped page table.**

Address-Translation Scheme



Two-Level Page-Table Scheme

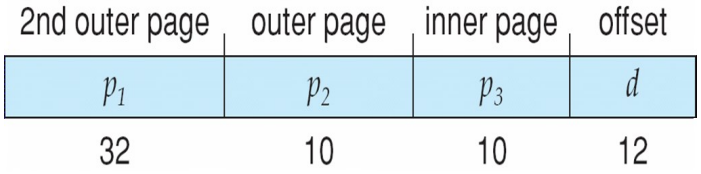


Three level paging scheme

- For a system with a 64-bit logical address space, a two-level paging scheme is no longer appropriate. To illustrate this point, let us suppose that the page size in such a system is 4 KB (2^{12}). In this case, the page table consists of up to 2^{52} entries. If we use a two-level paging scheme, then the inner page tables can conveniently be one page long, or contain 2^{10} 4-byte entries. The addresses look like this:

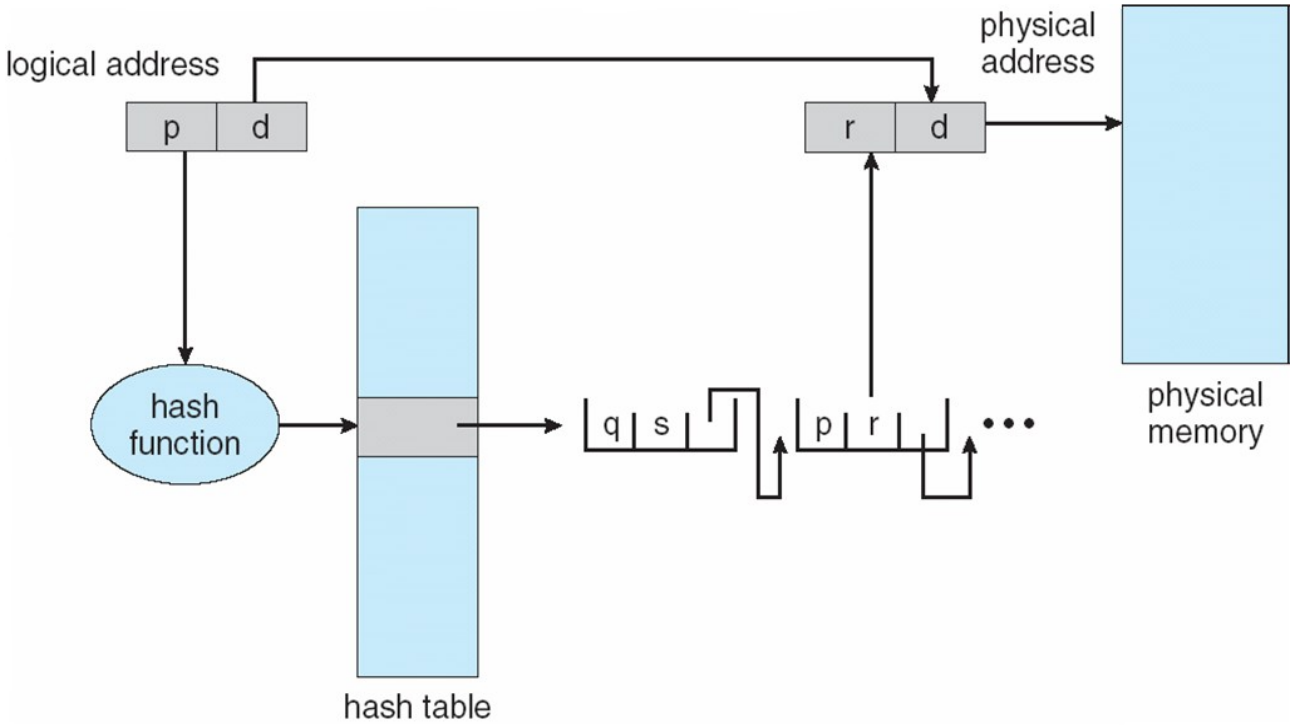
outer page	inner page	offset
p_1	p_2	d
42	10	12

- The outer page table consists of 2^{42} entries, or 2^{44} bytes. The obvious way to avoid such a large table is to divide the outer page table into smaller pieces. We can page the outer page table, giving us a three-level paging scheme.



Hashed Page Tables

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



Inverted Page Table

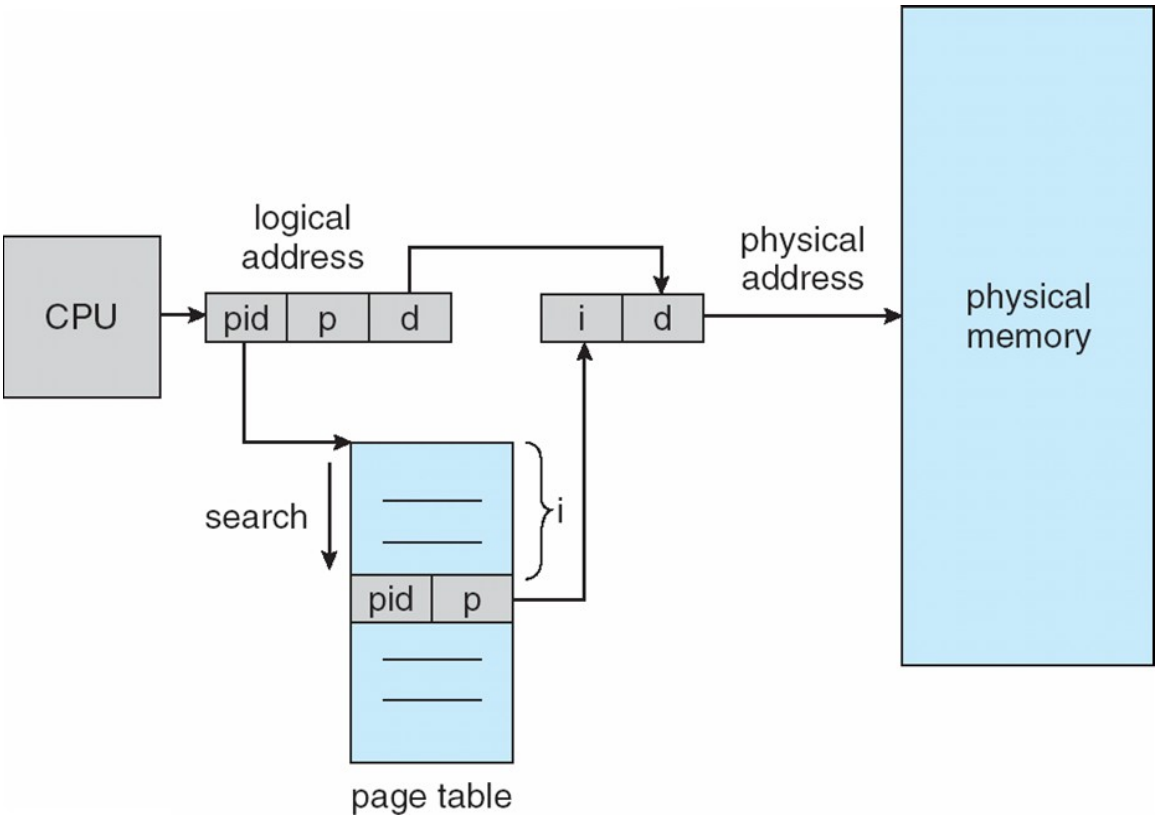
- Usually, each process has an associated page table. The page table has one entry for each page that the process is using.
- This table representation is a natural one, since processes reference pages through the pages' virtual addresses. The operating system must then translate this reference into a physical memory address.
- 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.

Each virtual address in the system consists of a triple:

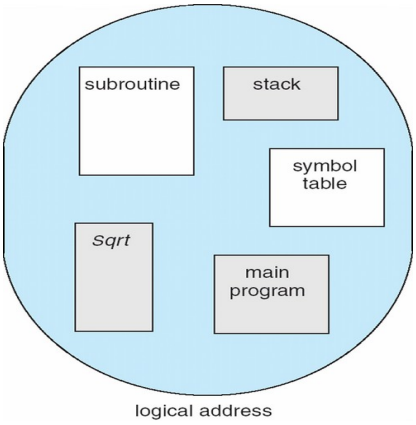
<process-id, page-number, offset>

- When memory reference occurs part of the virtual address consisting of <process-id, page-number> is presented to the memory subsystem. The inverted page table is then searched for a match.
- If match is found at entry i, then the physical address <i,offset> is generated. If no match is found an illegal address has been generated.
- Although this scheme decreases the amount of memory needed to store each page table, it increases the amount of time needed to search the table when a page reference occurs.
- Because the inverted page table is sorted by the physical address, but lookups occur on virtual addresses.



Segmentation

- As we have already seen, the user's view of memory is not the same as the actual physical memory. The user's view is mapped onto physical memory.
- Users prefer to view memory as a collection of variable-sized segments.



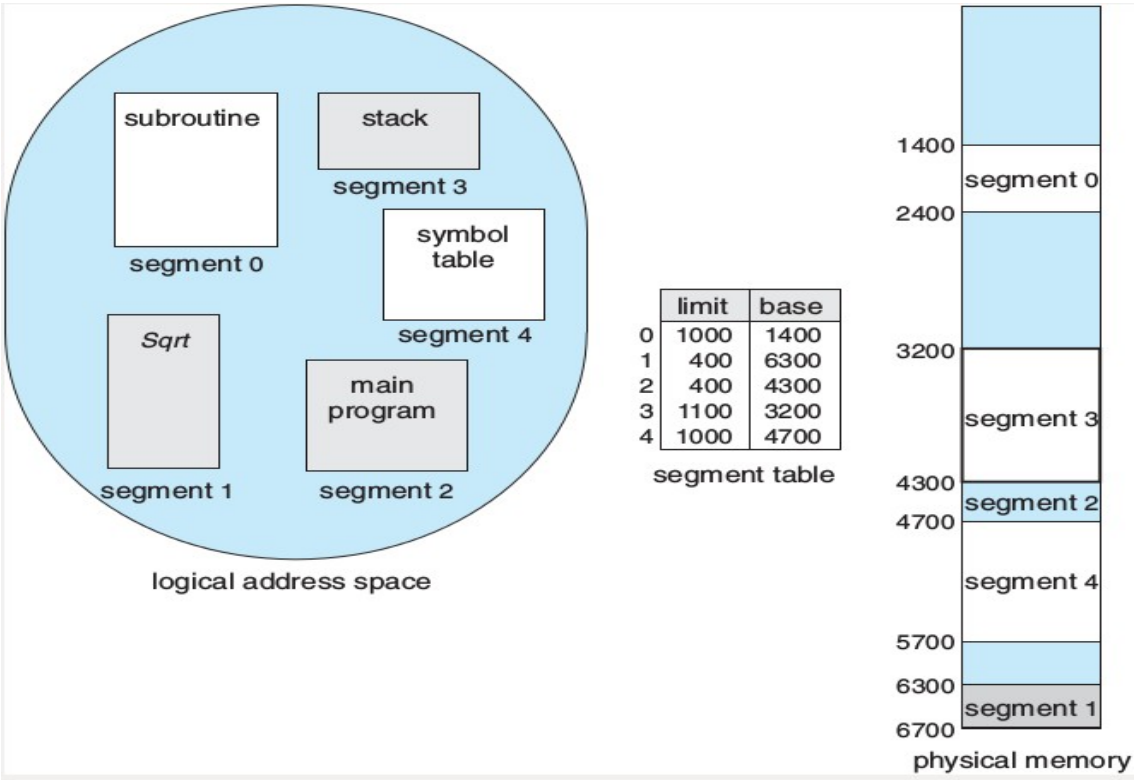
Basic Method

- 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.
- 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:

  <segment-number, offset>
- A C compiler might create separate segments for the following:
  - The code
  - Global variables
  - The heap, from which memory is allocated
  - The stacks used by each thread
  - The standard C library

Segmentation Architecture

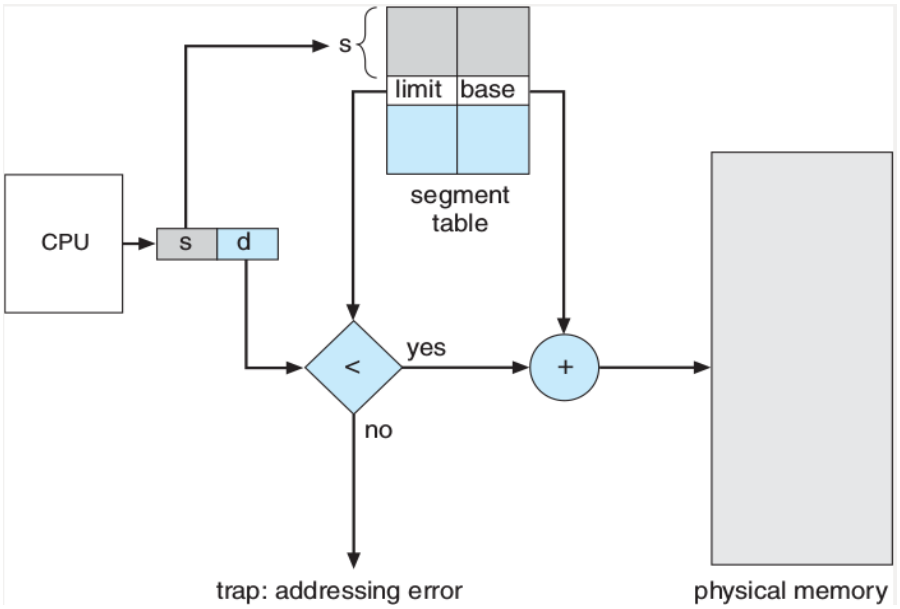
- The use of a segment table is illustrated below. A logical address consists of two parts: a segment number, s and an offset into that segment, d. The segment number is used as an index to the segment table. The offset d of the logical address must be between 0 and the segment limit. If it is not, we 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.
- As an example, consider the situation shown in above. We have 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 1000 bytes long.

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.

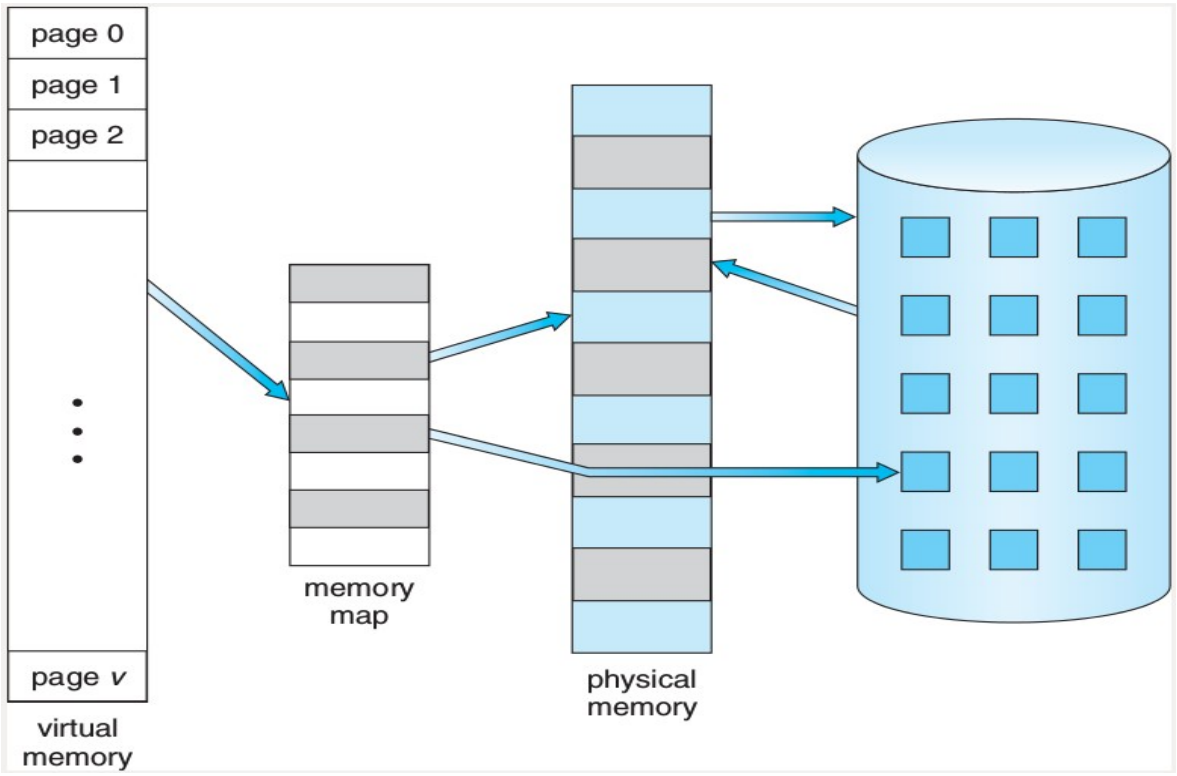


Virtual Memory Management

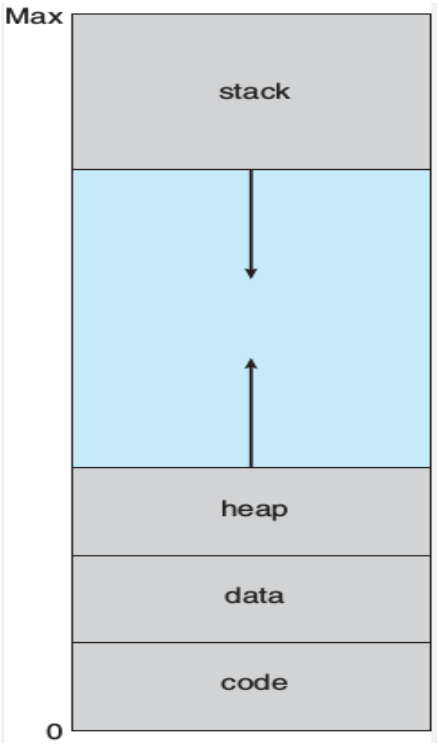
Background

- An examination of real programs shows us that, in many cases, the entire program is not needed in memory. For instance, consider the following:
  - 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.
  - Certain options and features of a program may be used rarely.
- 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.

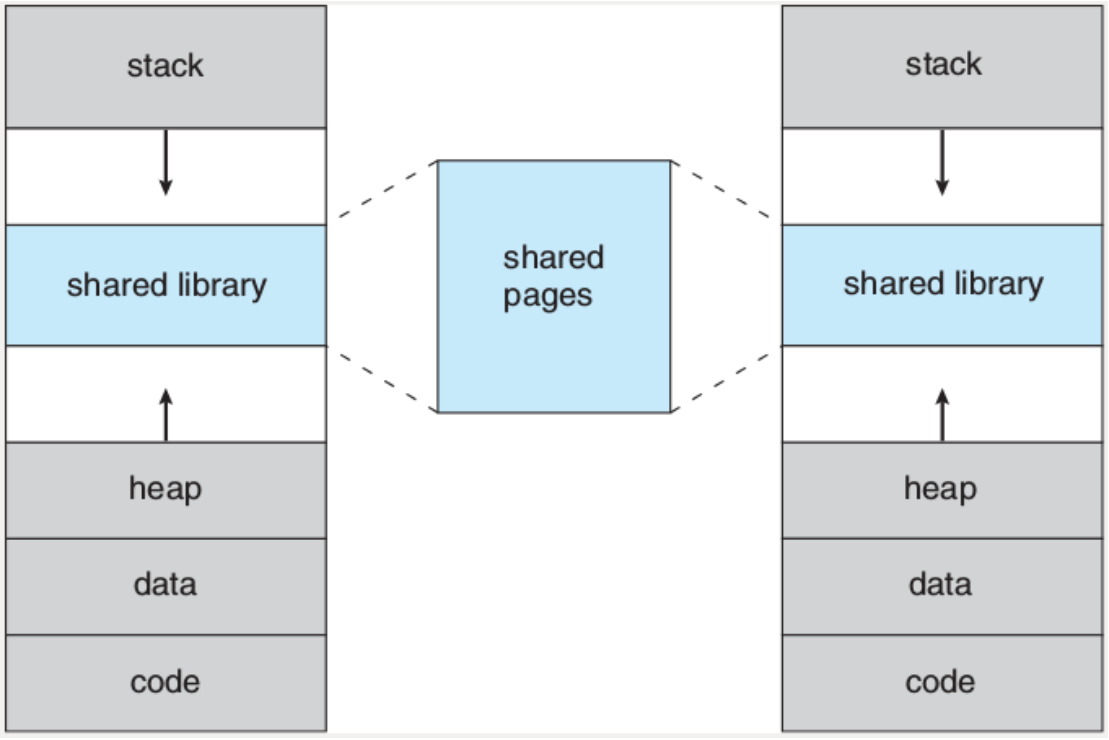
Virtual Memory That is Larger Than Physical Memory



- Virtual memory involves the separation of logical memory as perceived by users from physical memory. This separation allows an extremely large virtual memory to be provided for programmers when only a smaller physical memory is available.
- **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.
- We allow for the heap to grow upward in memory as it is used for dynamic memory allocation. Similarly, we allow 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.

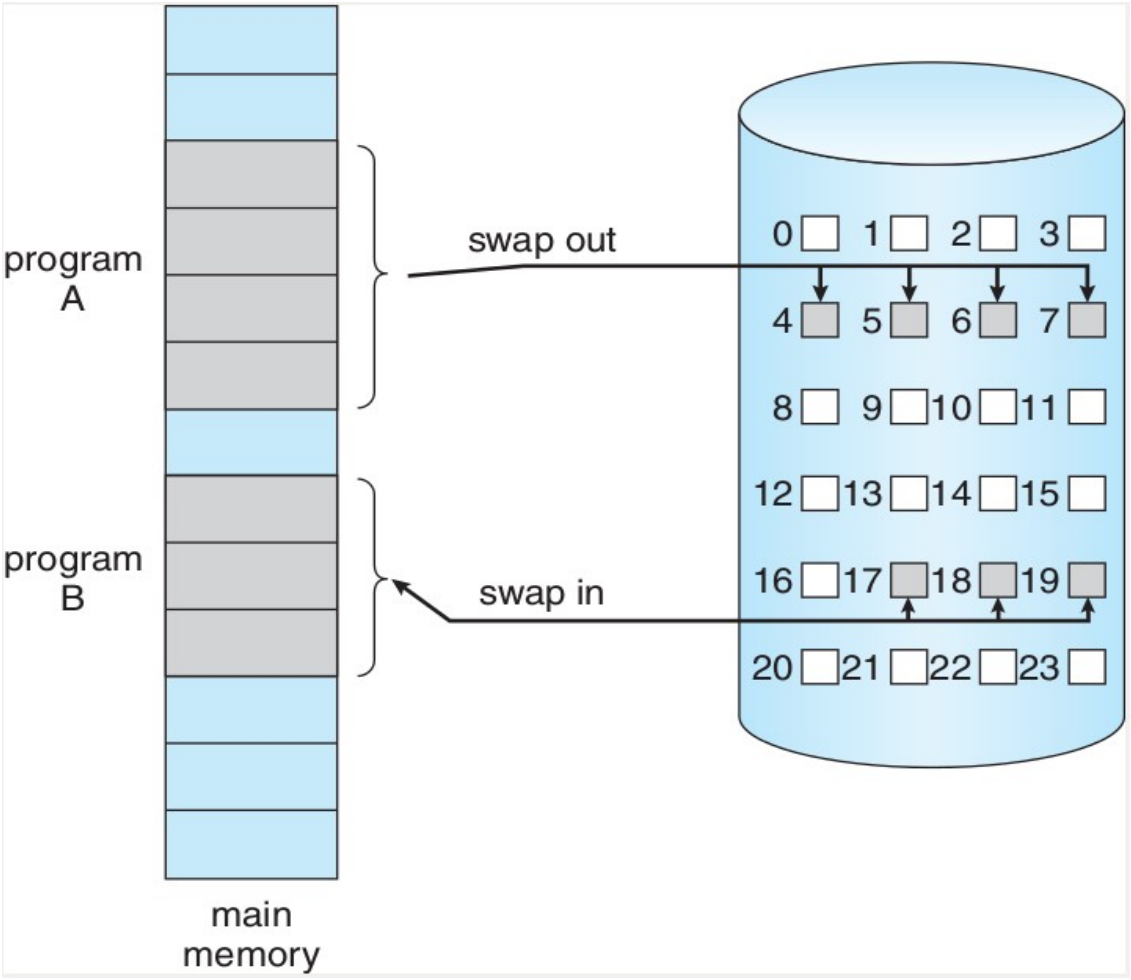


- Virtual memory allows files and memory to be shared by two or more processes through page sharing.
- System libraries can be shared by several processes through mapping of the shared object into a virtual address space. Although each process considers the shared libraries to be part of its virtual address space, the actual pages where the libraries reside in physical memory are shared by all the processes.
- 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.



Demand Paging

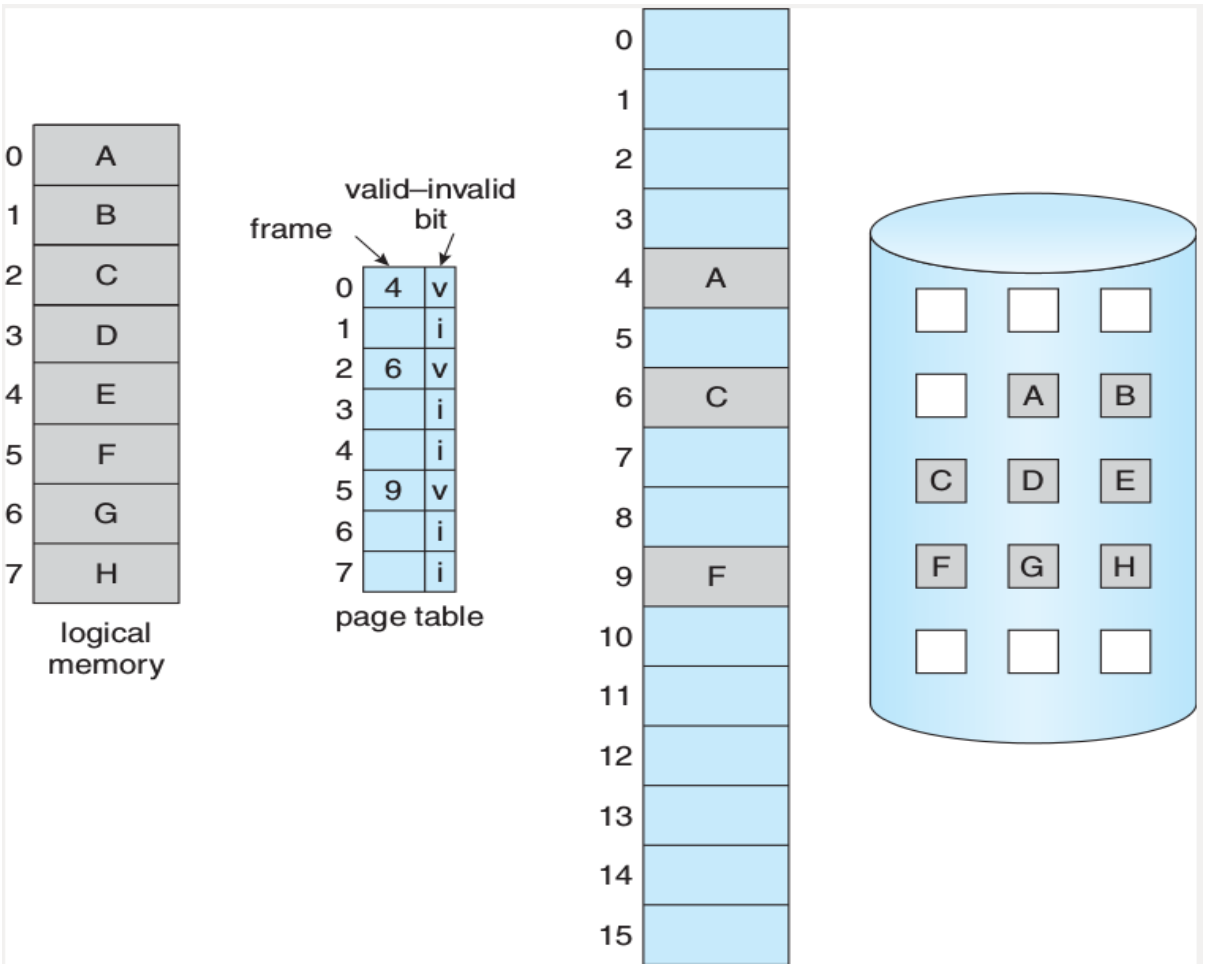
- Consider how an executable program might be loaded from disk into memory. One option is to load the entire program into physical memory at program execution time.
- An alternative strategy is to load pages only as they are needed. This technique is known as **demand paging**. Pages that are never accessed are thus never loaded into physical memory.
- A demand-paging system is similar to a paging system with swapping. When we want to execute a process, we swap it into memory. Rather than swapping the entire process into memory, however, 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.



Basic Concepts

- Pager brings only required pages into memory. Thus decreases 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.
- The valid-invalid bit scheme can be used for this purpose. 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.
- Access to a page marked invalid causes a **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.

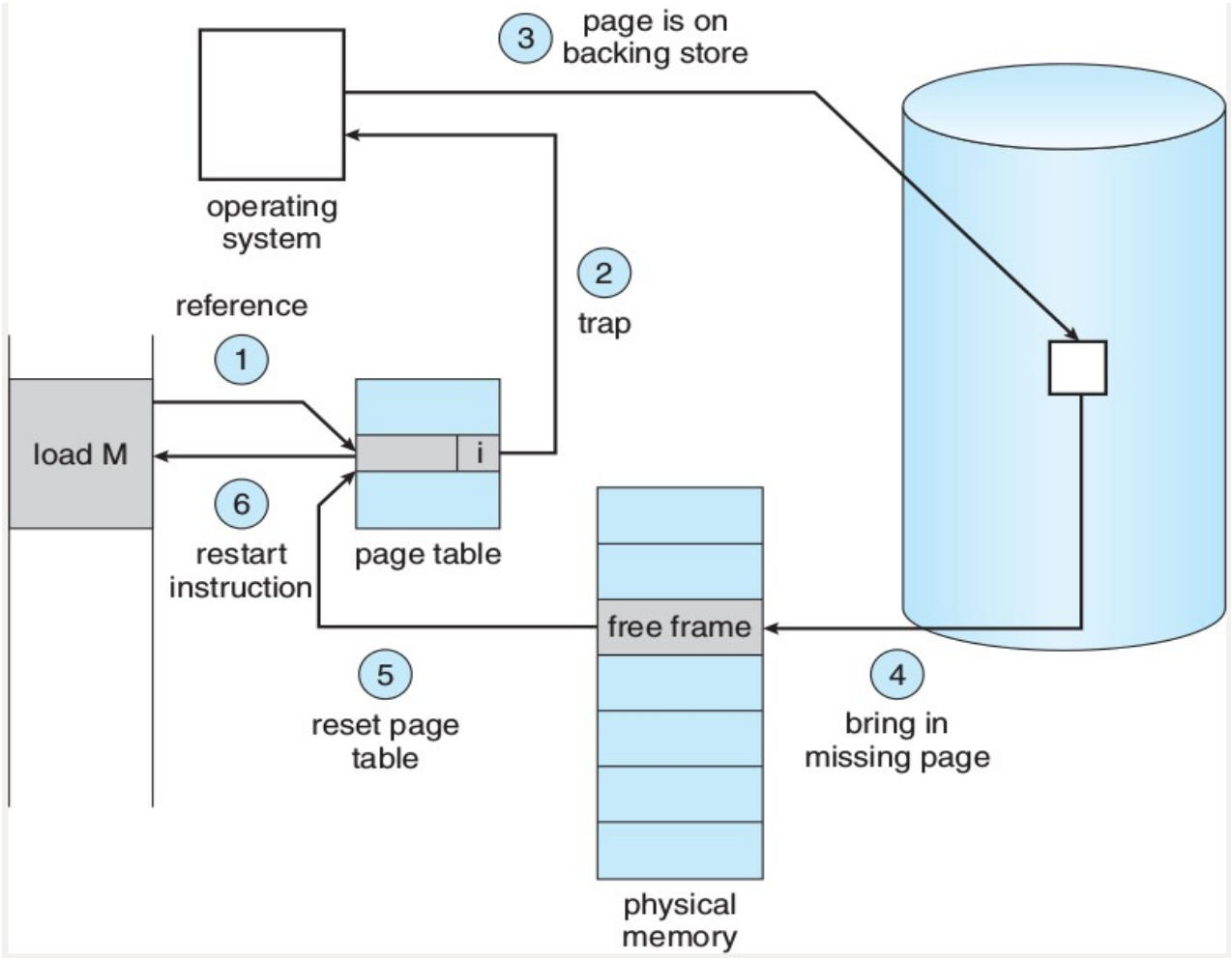
Page Table When Some Pages Are Not in Main Memory



Page Fault

- The procedure for handling page fault is straightforward:
  - We check an internal table (usually kept with the process control block) to determine whether the reference was a valid or an invalid memory access.
  - If the reference was invalid, we terminate the process. If it was valid, we have not yet brought in that page, we now page it in.
  - We find a free frame (by taking one from the free-frame list, for example). We schedule a disk operation to read the desired page into the newly allocated frame.
  - 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.
  - 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**.

Steps in Handling a Page Fault



Aspects of Demand Paging

- Theoretically, a given instruction could access multiple pages, possibly causing multiple page faults per instruction. This situation would result in unacceptable system performance.
  - Consider fetch and decode of instruction which adds 2 numbers from memory and stores result back to memory.
- But programs tend to have locality of references which results in reasonable performance from demand paging.
- The hardware to support demand paging is the same as the hardware for paging and swapping:
  - Page table. This table has the ability to mark an entry invalid through a valid-invalid bit or a special value of protection bits.
  - Secondary memory. This memory holds those pages that are not present in main memory. It is known as the swap device, and the section of disk used for this purpose is known as swap space.

Instruction Restart

- 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, except that the desired page is now in memory and is accessible.

- A page fault may occur at any memory reference. If the page fault occurs on the instruction fetch, we can restart by fetching the instruction again. If a page fault occurs while we are fetching an operand, we must fetch and decode the instruction again and then fetch the operand.
- As a worst-case example, consider a three-address instruction such as ADD the content of A to B, placing the result in C. These are the steps to execute this instruction:
  - Fetch and decode the instruction (ADD).
  - Fetch A
  - Fetch B.
  - Add A and B.
  - Store the sum in C.
- If we fault when we try to store in C (because C is in a page not currently in memory), we will have to get the desired page, bring it in, correct the page table, and restart the instruction. The restart will require fetching the instruction again, decoding it again, fetching the two operands again, and then adding again.

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/byte.
- 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 then is

  effective access time= (1 - p) x ma + p x 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/0 subsystem (I/0 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.

Demand Paging Example

- Three major components of the page-fault service time:
  - Service the page-fault interrupt.
  - Read in the page.
  - Restart the process.
- 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) x (200) + p (8 milliseconds)

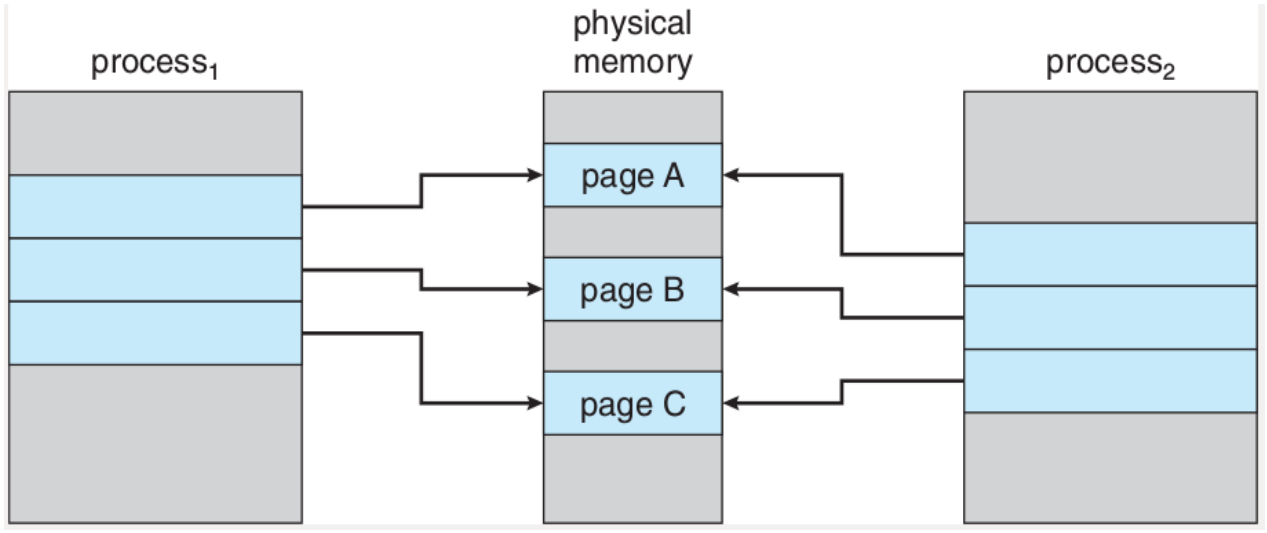
  = (1 - p) X 200 + p X 8,000,000

  = 200 + 7,999,800 X p.
- We see that the effective access time is directly proportional to the page fault rate. If one access out of 1,000 causes a page fault, the effective access time is 8.2 microseconds. The computer will be slowed down by a factor of 40 because of demand paging.

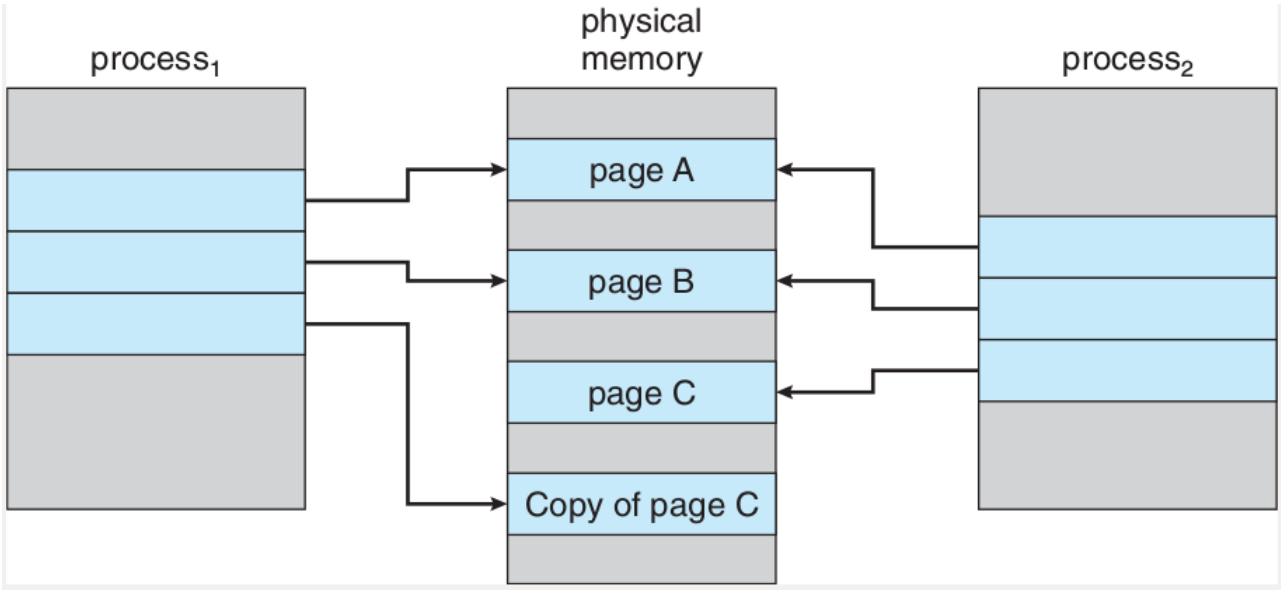
Copy-on-Write

- This technique provides for rapid process creation and minimizes the number of new pages that must be allocated to the newly created process.
- Traditionally, fork() worked by creating a copy of the parent's address space for the child, duplicating the pages belonging to the parent.
- Many child processes invoke the exec() system call immediately after creation, then copying of the parent's address space may be unnecessary.
- Instead, we can use a technique known as copy-on-write, which works by allowing the parent and child processes initially to share the same pages. These shared pages are marked as copy-on-write pages, meaning that if either process writes to a shared page, a copy of the shared page is created.
- For example, assume that the child process attempts to modify a page containing portions of the stack, with copy-on-write, the operating system will create a copy of this page, mapping it to the address space of the child process.
- The child process will then modify its copied page and not the page belonging to the parent process. All unmodified pages can be shared by the parent and child processes.

Before Process 1 Modifies Page C



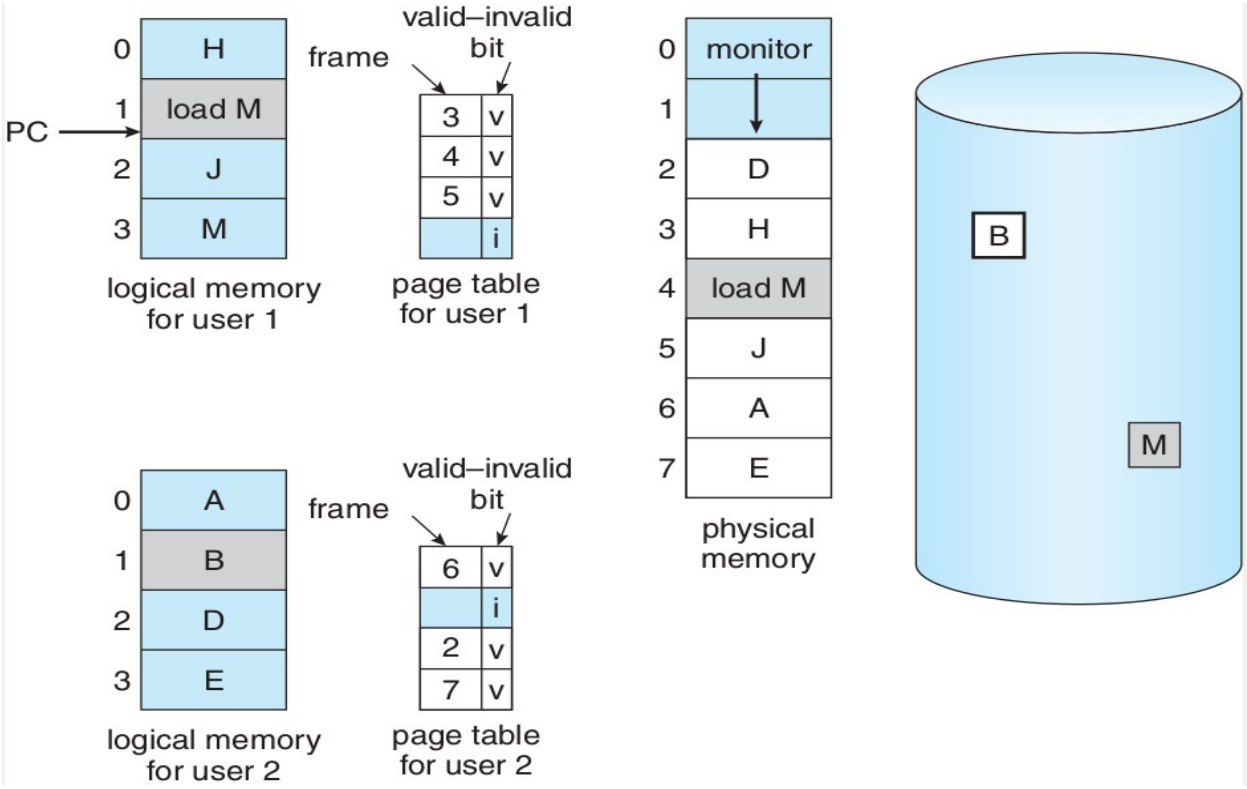
After Process 1 Modifies Page C



Page Replacement

- 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.
- 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. So this option is not the best choice.
- The operating system could instead swap out a process, freeing all its frames and reducing the level of multiprogramming. This option is a good one in certain circumstances.

Need For Page Replacement



Basic Page Replacement

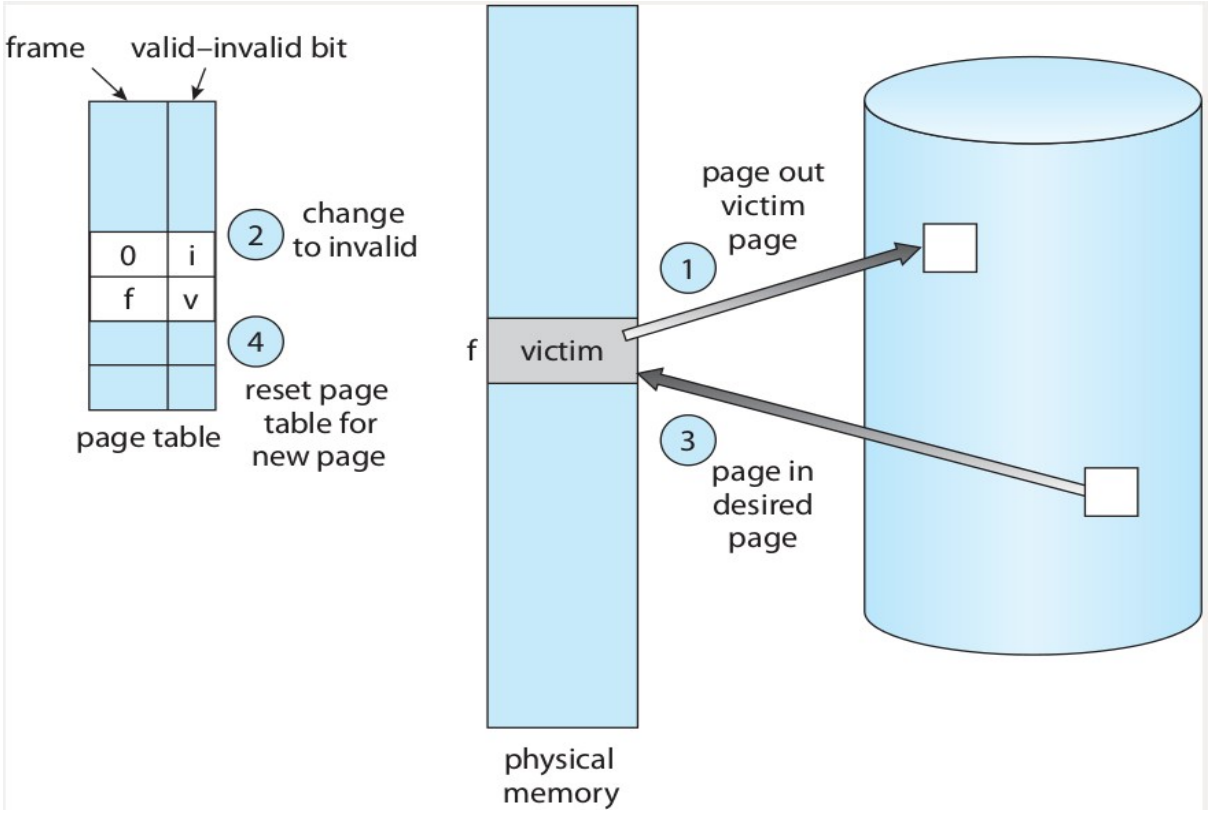
- Page replacement takes the following approach.
  - If no frame is free, we find one that is not currently being used and free it.
  - We can free a frame by writing its contents to swap space and changing the page table (and all other tables) to indicate that the page is no longer in memory.
- We can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



Page Replacement and Frame Allocation Algorithms

- Notice that, 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.
- We can reduce this overhead by using a modify bit. When this scheme is used, each page or frame has a **modify bit** associated with it in the hardware.

- The modify bit for a page is set by the hardware whenever any word or byte in the page is written into, indicating that the page has been modified.
- When we select a page for replacement, we examine its modify bit. If the bit is set, we know that the page has been modified since it was read in from the disk. In this case, we must write the page to the disk.
- If the modify bit is not set, however, the page has not been modified since it was read into memory. In this case, we need not write the memory page to the disk: it is already there.
- If we have a user process of twenty pages, we can execute it in ten frames simply by using demand paging and using a replacement algorithm to find a free frame whenever necessary.
- We must solve two major problems to implement demand paging:we must develop a **frame allocation algorithm** and a **page replacement algorithm**.
- That is, if we have multiple processes in memory, we must decide how many frames to allocate to each process; and when page replacement is required, we must select the frames that are to be replaced.

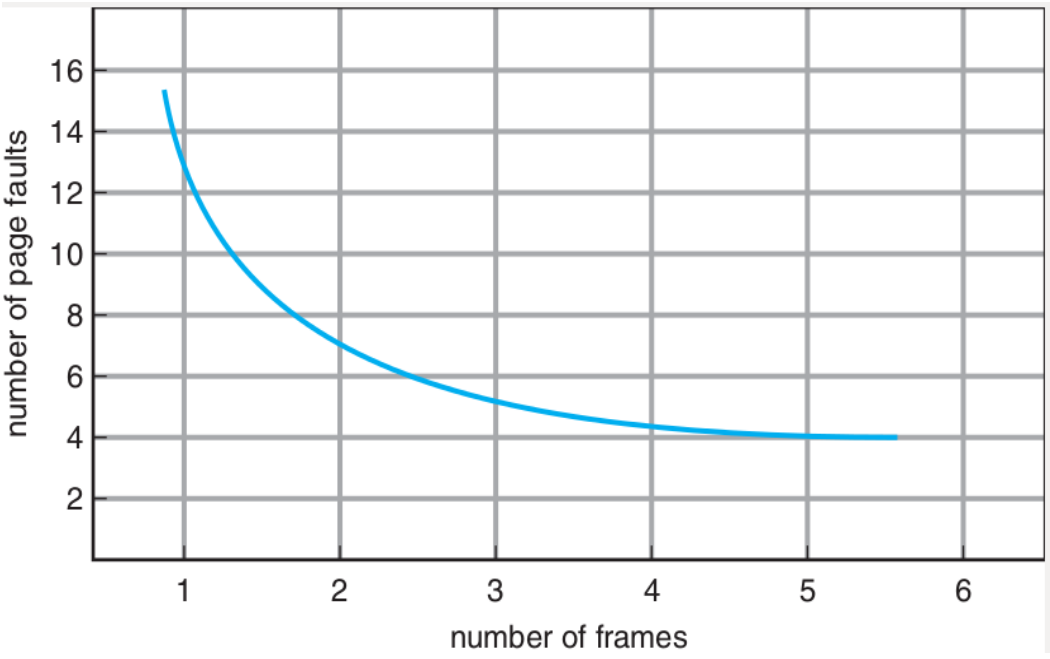
Page-replacement algorithm

- We evaluate an algorithm by running it on a particular string of memory references and computing the number of page faults.
- The string of memory references is called a reference string. First, for a given page size, we need to consider only the page number, rather than the entire address. For example, if we trace a particular process, we might record the following address sequence:

  0100, 0432, 0101, 0612, 0102, 0103, 0104, 0101, 0611, 0102,
  0103, 0104, 0101, 0610, 0102, 0103, 0104, 0101, 0609, 0102, 0105
- At 100 bytes per page, this sequence is reduced to the following reference string:

  1, 4, 1, 6, 1, 6, 1, 6, 1, 6, 1
- To determine the number of page faults for a particular reference string and page-replacement algorithm, we also need to know the number of page frames available. Obviously, as the number of frames available increases, the number of page faults decreases.

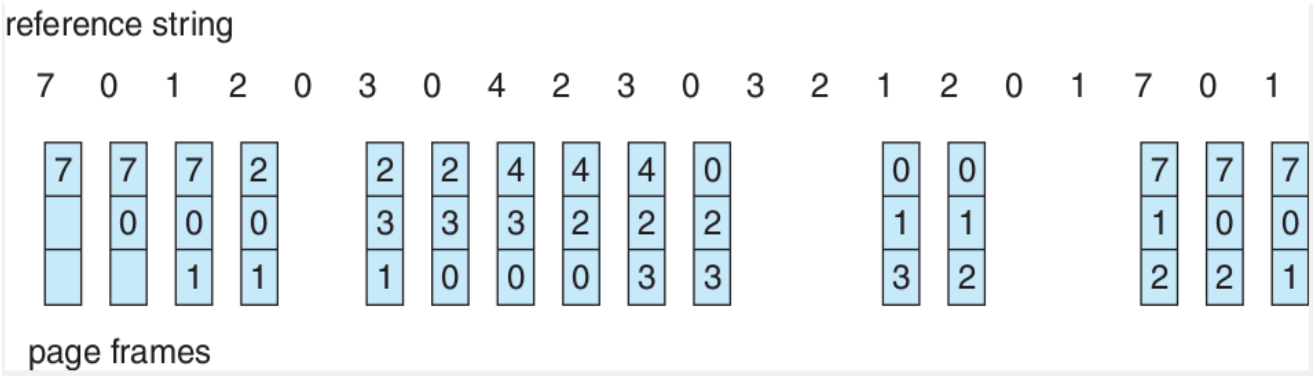
Graph of Page Faults Versus The Number of Frames



First-In-First-Out (FIFO) Algorithm

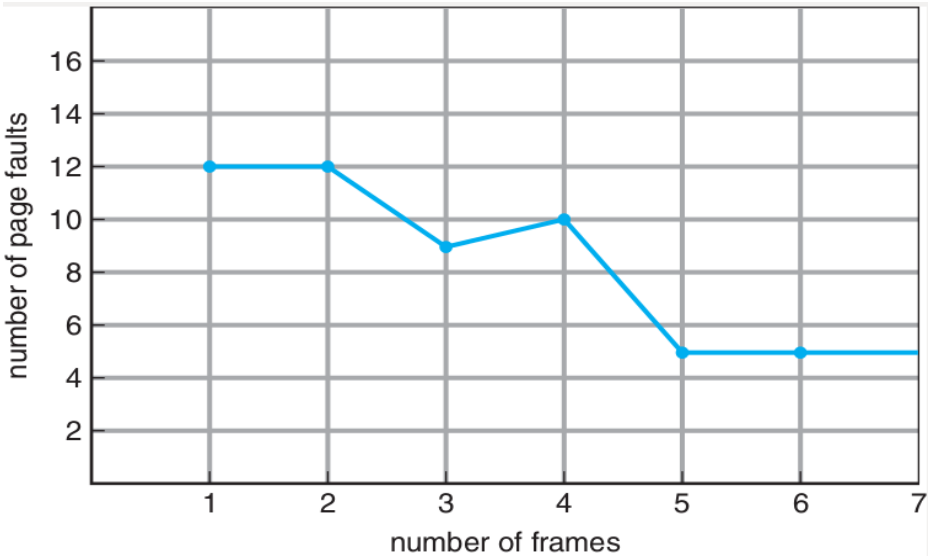
- A FIFO replacement algorithm associates with each page the time when that page was brought into memory. When a page must be replaced, the oldest page is chosen.
- Consider the following reference string.

  7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1
- 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 it is now first in line. This process continues as shown below. Total number of page faults is 15.



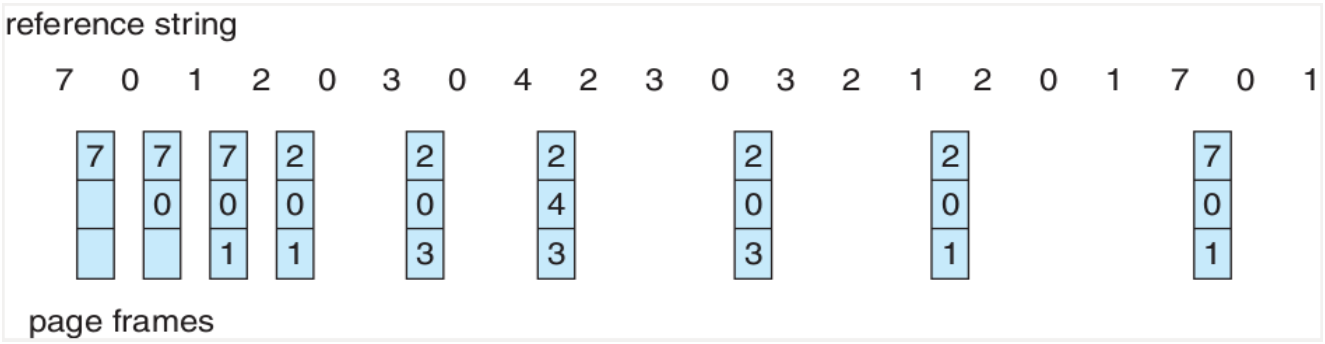
FIFO Illustrating Belady's Anomaly

- To illustrate the problems that are possible with a FIFO page-replacement algorithm, we consider the following reference string: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5
- Graph shows the curve of page faults for this reference string versus the number of available frames. Notice that the number of faults for four frames (ten) is greater than the number of faults for three frames (nine)! This most unexpected result is known as **Belady's** anomaly.



Optimal Algorithm

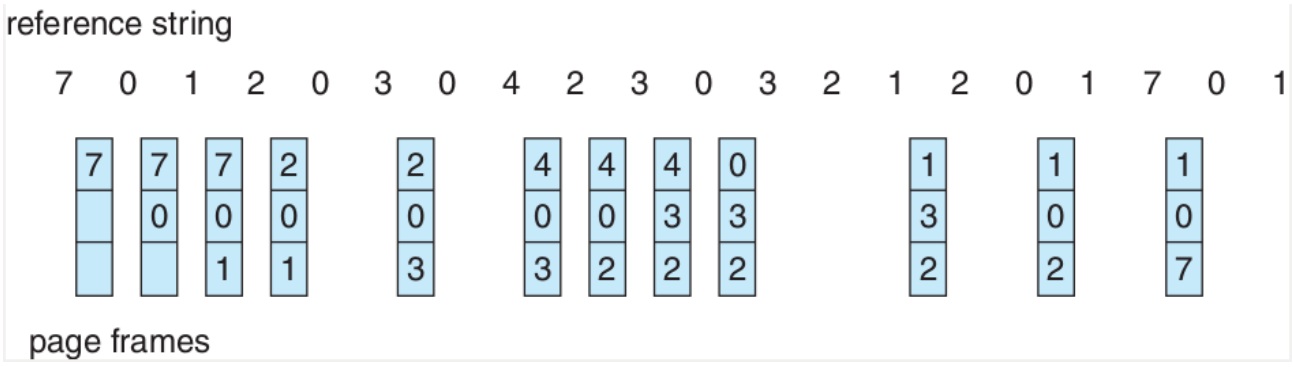
- Replace the page that will not be used for the longest period of time.
- Use of this page-replacement algorithm guarantees the lowest possible page-fault rate for a fixed number of frames.



- 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.
- 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.

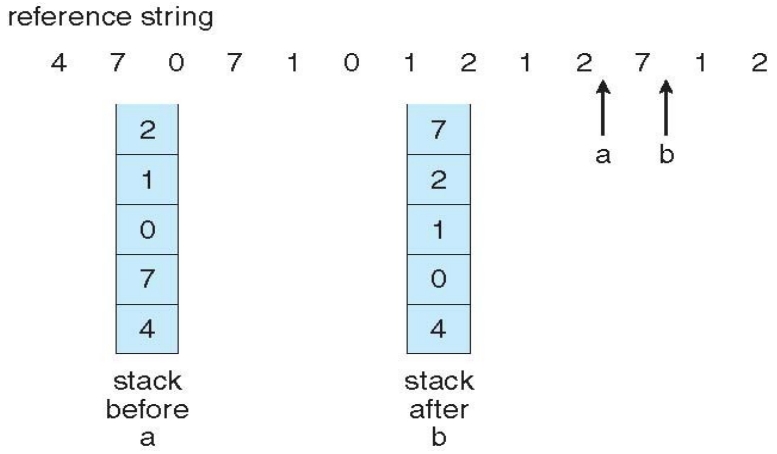
Least Recently Used (LRU) Algorithm

- 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.
- When a page must be replaced, LRU chooses the page that has not been used for the longest period of time. We can think of this strategy as the optimal page-replacement algorithm looking backward in time, rather than forward.
- The result of applying LRU replacement to our example reference string is shown below. The LRU algorithm produces twelve faults. When the reference to page 4 occurs, 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. Despite these problems, LRU replacement with twelve faults is much better than FIFO replacement with fifteen.



- The problem here is to determine an order for the frames defined by the time of last use. Two implementations are feasible:
  - Counters. Associate with each page-table entry a time-of-use field and add to the CPU a counter. The counter is incremented for every memory reference. Whenever a reference to a page is made, the contents of the counter are copied to the time-of-use field in the page-table entry for that page. We replace the page with the smallest time value.

- Stack Whenever a page is referenced, it is removed from the stack and put on the top. In this way, the most recently used page is always at the top of the stack and the least recently used page is always at the bottom.



LRU Approximation Algorithms

- 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.

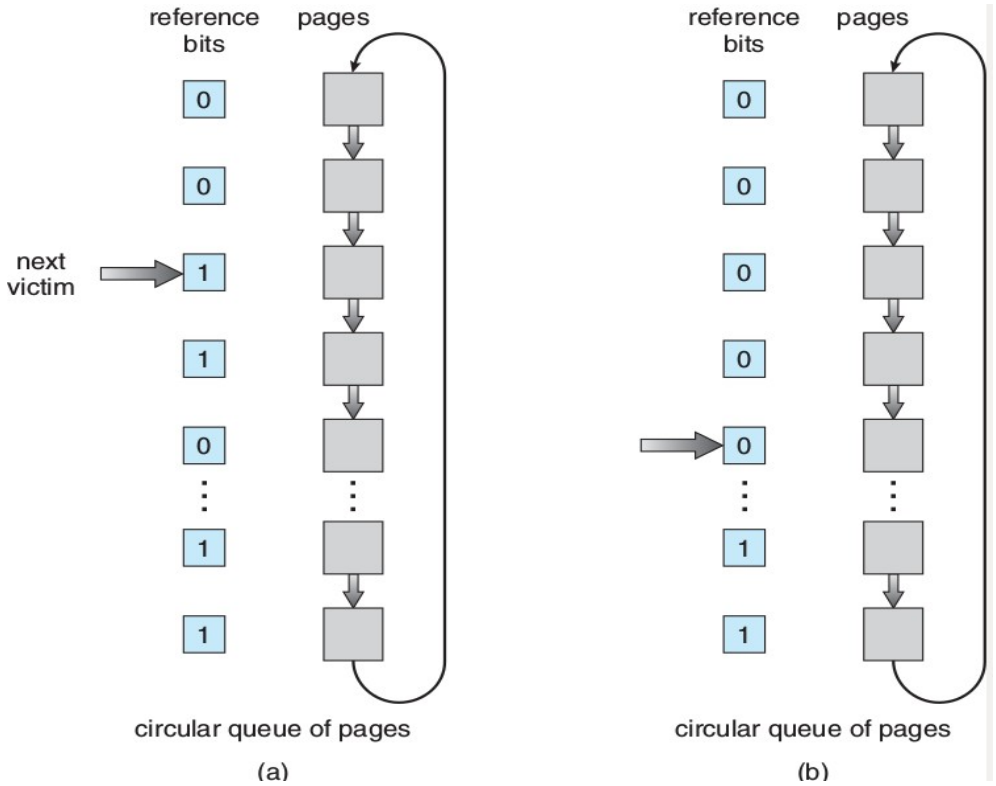
Additional-Reference-Bits Algorithm

- We can gain additional ordering information by recording the reference bits at regular intervals.
- We can keep an 8-bit byte for each page in a table in memory. At regular intervals, a timer interrupt transfers control to the operating system.
- The operating system shifts the reference bit for each page into the high-order bit of its 8-bit byte, shifting the other bits right by 1 bit and discarding the low-order bit.
- If the shift register contains 00000000, for example, then the page has not been used for eight time periods; A page with a register value of 11000100 has been used more recently than one with a value of 01110111.
- If we interpret these 8-bit bytes as unsigned integers, the page with the lowest number is the LRU page, and it can be replaced.

Second-Chance Page-Replacement Algorithm

- When a page has been selected for replacement, we inspect its reference bit. If the value is 0, we proceed to replace this page.
- But if the reference bit is set to 1, we give the page a second chance and move on to select the next page. When a page gets a second chance, its reference bit is cleared, and its arrival time is reset to the current time.
- One way to implement the second-chance algorithm is as a circular queue. A pointer indicates which page is to be replaced next. When a frame is needed, the pointer advances until it finds a page with a 0 reference bit. As it advances, it clears the reference bits.
- Once a victim page is found, the page is replaced, and the new page is inserted in the circular queue in that position.

- Notice that, in the worst case, when all bits are set, the pointer cycles through the whole queue, giving each page a second chance.



Enhanced Second-Chance Algorithm

- We can enhance the second-chance algorithm by considering the reference bit and the modify bit as an ordered pair.
  - (0, 0) neither recently used nor modified -best page to replace.
  - (0, 1) not recently used but modified-not quite as good, because the page will need to be written out before replacement.
  - (1, 0) recently used but clean -probably will be used again soon.
  - (1, 1) recently used and modified -probably will be used again soon, and the page will be need to be written out to disk before it can be replaced.
- Each page is in one of these four classes. We replace the first page encountered in the lowest nonempty class.
- Notice that we may have to scan the circular queue several times before we find a page to be replaced.

Counting-Based Page Replacement

- We can keep a count 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.
- 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.
- An expansion of this idea is to maintain a list of modified pages. Whenever the paging device is idle, a modified page is selected and is written to the disk. Its modify bit is then reset.
- This scheme increases the probability that a page will be clean when it is selected for replacement and will not need to be written out.
- 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 out to the disk, the old page can be reused directly from the free-frame pool if it is needed before that frame is reused.

Allocation of Frames

Minimum Number of Frames

- 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.
- In addition, remember that when a page fault occurs before an executing instruction is complete, the instruction must be restarted.
- For example, consider a machine in which all memory-reference instructions may reference only one memory address.
- In this case, we need at least one frame for the instruction and one frame for the memory reference.
- 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. The worst-case scenario occurs in computer architectures that allow multiple levels of indirection. To overcome this difficulty, we must place a limit on the levels of indirection.

Allocation Algorithms

- The easiest way to split m frames among n processes is to give every process 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**.
- An alternative is to recognize that various processes will need differing amounts of memory. 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 P_i be S_i, and define $S= \sum S_i$

s_i = size of process p_i
$S = \sum s_i$
m = total number of frames
a_i = allocation for $p_i = \frac{s_i}{S} \times m$

$m = 64$
$s_1 = 10$
$s_2 = 127$
$a_1 = \frac{10}{137} \times 62 \approx 4$
$a_2 = \frac{127}{137} \times 62 \approx 57$

- Of course, we must adjust each a_i to be an integer that is greater than the minimum number of frames required by the instruction set, with a sum not exceeding m. With proportional allocation, we would split 64 frames between two processes, one of 10 pages and one of 127 pages, by allocating 4 frames and 57 frames, respectively.

Global vs. 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.
- With global replacement, a process may happen to select only frames allocated to other processes, thus increasing the number of frames allocated to it.
- One problem with a global replacement algorithm is that a process cannot control its own page-fault rate.
- The set of pages in memory for a process depends not only on the paging behavior of that process but also on the paging behavior of other processes.
- Under local replacement, the set of pages in memory for a process is affected by the paging behavior of only that process. Global replacement generally results in greater system throughput and is therefore the more common method.

Non-Uniform Memory Access

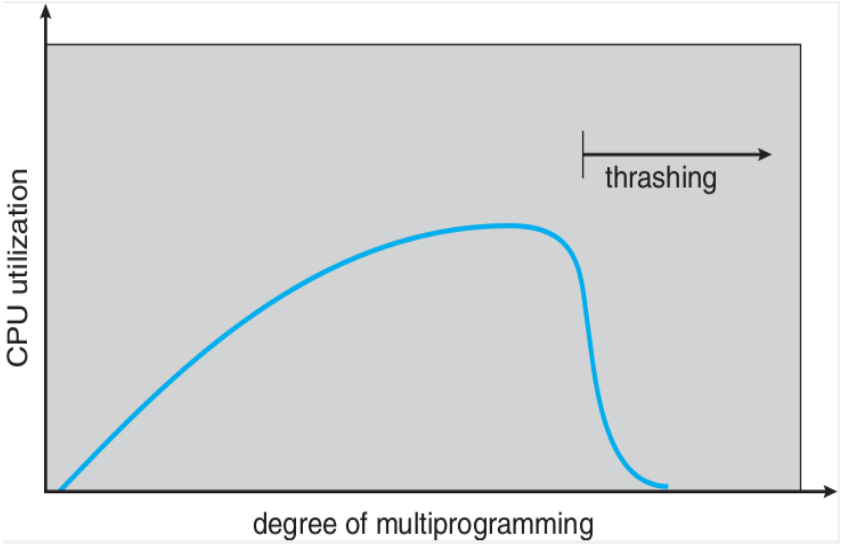
- In systems with multiple CPUs, a given CPU can access some sections of main memory faster than it can access others.
- Frequently, such a system is made up of several system boards, each containing multiple CPUs and some memory.
- As you might expect, the CPUs on a particular board can access the memory 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 systems(NUMA).
- Managing which page frames are stored at which locations can significantly affect performance in NUMA systems. The goal of these changes is to have memory frames allocated "as close as possible" to the CPU on which the process is running.
- The algorithmic changes consist of having the scheduler track the last CPU on which each process ran. If the scheduler tries to schedule each process onto its previous CPU, and the memory-management system tries to allocate frames for the process close to the CPU on which it is being scheduled, then improved cache hits and decreased memory access times will result.

Thrashing

- The high paging activity is called thrashing. A process is thrashing if it is spending more time on paging than on executing.

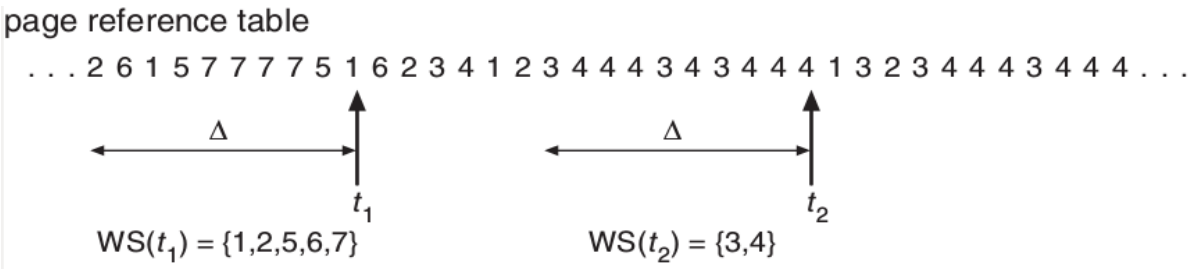
Cause of Thrashing

- The operating system monitors CPU utilization. If CPU utilization is too low, we increase the degree of multiprogramming by introducing a new process to the system.
- A global page-replacement algorithm is used; 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, and the CPU scheduler tries to increase the degree of multiprogramming even more. Thrashing has occurred, and system throughput plunges. The page- fault rate increases tremendously.



Working-Set Model

- This model uses a parameter Δ, to define the working-set window.
- The idea is to examine the most recent Δ page references. The set of pages in the most recent Δ 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 Δ time units after its last reference.
- Thus, the working set is an approximation of the program's locality.
- For example, given the sequence of memory references shown below, if Δ = 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}.



Keeping Track of the Working Set

- The most important property of the working set, is its size. If we compute the working-set size, WSS_i, for each process in the system, we can then consider that

$$D=\sum WSS_i$$

- Where D is the total demand for frames. Each process is actively using the pages in its working set. If the total demand is greater than the total number of available frames (D > m), thrashing will occur, because some processes will not have enough frames.
- If the sum of the working-set sizes increases, exceeding the total number of available frames, the operating system selects a process to suspend. The process's pages are written out (swapped), and its frames are reallocated to other processes. The suspended process can be restarted later.
- This working-set strategy prevents thrashing while keeping the degree of multiprogramming as high as possible. The difficulty with the working-set model is keeping track of the working set.

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.