UNIT - III STORAGE MANAGEMENT

Main Memory-Contiguous Memory Allocation, Segmentation, Paging, 32 and 64 bit architecture Examples; Virtual Memory- Demand Paging, Page Replacement, Allocation, Thrashing; Allocating Kernel Memory, OS Examples.

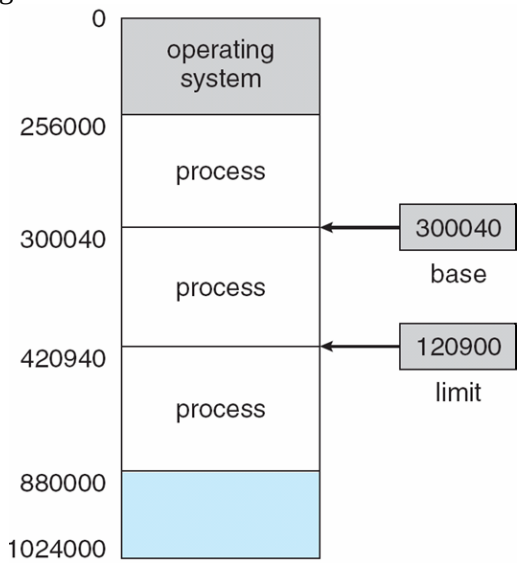
MAIN MEMORY:

MEMORY HARDWARE:

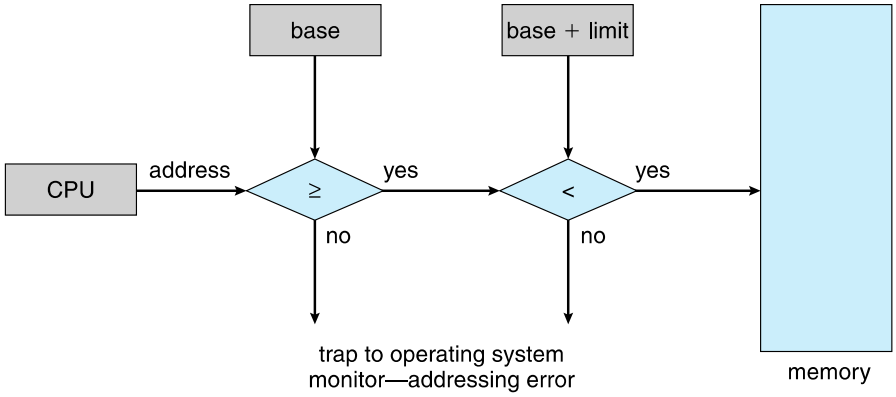
- Memory consists of a large array of bytes, each with its own address. The CPU fetches instructions from memory according to the value of the program counter.
- Main memory and the registers built into the processor itself are the only general-purpose storage that the CPU can access directly.
- Registers that are built into the CPU are generally accessible within one cycle of the CPU clock.
- A memory access may take many cycles of the CPU clock. In such cases, the processor normally needs to **stall**, since it does not have the data required to complete the instruction.
- The remedy is to add fast memory between the CPU and main memory, typically on the CPU chip for fast access called as CACHE.

MEMORY PROTECTION:

- For proper system operation we must protect the operating system from access by user processes.
- Each process has a separate memory space. Separate per-process memory space protects the processes from each other.
- The hardware protection of memory is provided by two registers
  - Base Register
  - Limit Register
- **The base register holds the smallest legal physical memory address, called the starting address of the process.**
- **The Limit register specifies the size of range of the process.**
- If the base register holds300040 and the limit register is 120900, then the program can legally access all addresses from 300040 through 420939



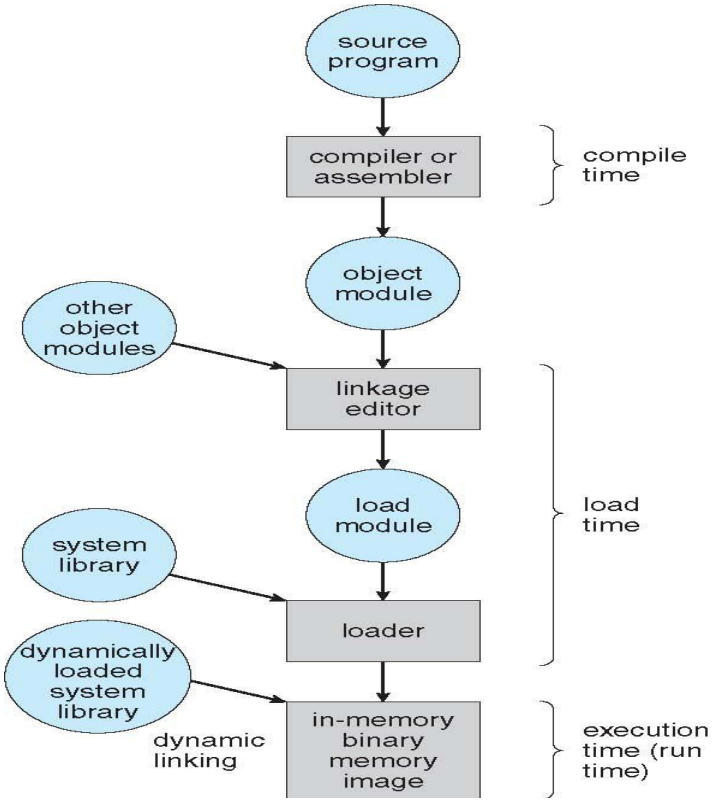
- Protection of memory space is accomplished by having the CPU hardware compare every address generated in user mode with the registers.
- Any attempt by a program executing in user mode to access operating-system memory or other users' memory results in a trap to the operating system, resulting in addressing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



- This scheme prevents a user program from modifying the code or data structures of either the operating system or other users.
- The address generated by the CPU for a process should lie between the Base address of the process and base + Limit of the process, Else the hardware sends an interrupt to the OS.

ADDRESS BINDING:

- Address binding is the process of mapping the program's logical or virtual addresses to corresponding physical or main memory addresses.
- Addresses in the source program are generally symbolic.
- A compiler typically **binds** these symbolic addresses to relocatable addresses.
- The linkage editor or loader in turn binds the relocatable addresses to absolute addresses
- Each binding is a mapping from one address space to another .

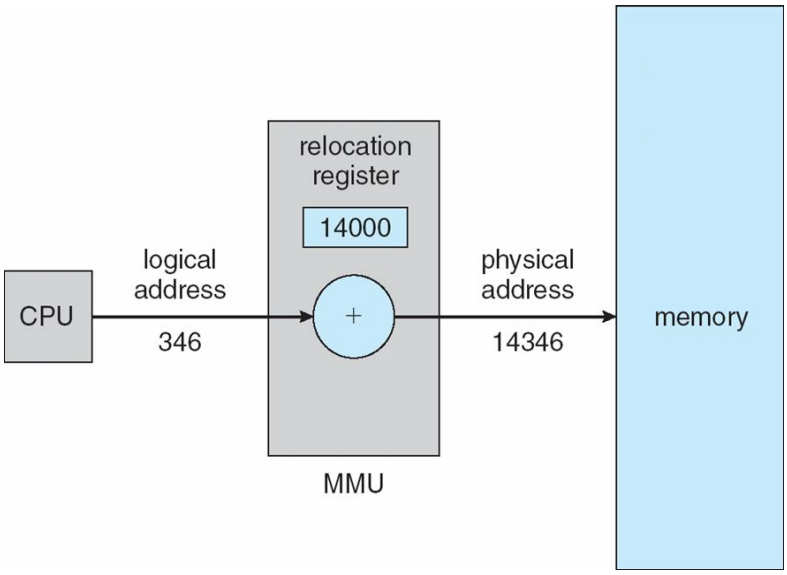


The binding of instructions and data to memory addresses can be done in three ways.

1) Compile time. If you know at compile time where the process will reside in memory, then **absolute code** can be generated.
2) Load time. If it is not known at compile time where the process will reside in memory, then the compiler must generate **relocatable code**.
3) Execution time. If the process can be moved during its execution from one memory segment to another, then binding must be delayed until run time.

LOGICAL VERSUS PHYSICAL ADDRESS SPACE:

- An address generated by the CPU is commonly referred to as a **logical address**. which is also called as virtual address
- The set of all logical addresses generated by a program is a **logical address space**.
- 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 set of all physical addresses corresponding to these logical addresses is a **physical address space**.
- The run-time mapping from virtual to physical addresses is done by a device called the **memory-management unit (MMU)**.



- The base register is also called as **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

DYNAMIC LOADING:

- Dynamic Loading is the process of loading a routine only when it is called or needed during runtime.
- Initially 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.
- The advantage of dynamic loading is that a routine is loaded only when it is needed.
- This method is particularly useful when large amounts of code are needed to handle infrequently occurring cases, such as error routines.

DYNAMIC LINKING AND SHAREDLIBRARIES:

- **Dynamically linked libraries** are system libraries that are linked to user programs when the programs are in execution.

- In Dynamic linking the linking of system libraries are postponed until execution time.
- Static Linking combines the system libraries to the user program at the time of compilation**.**
- Dynamic linking saves both the disk space and the main memory space.
- The libraries can be replaced by a new version and all the programs that reference the library will use the new version. **This system is called as Shared libraries** which can be done with dynamic linking.

SWAPPING:

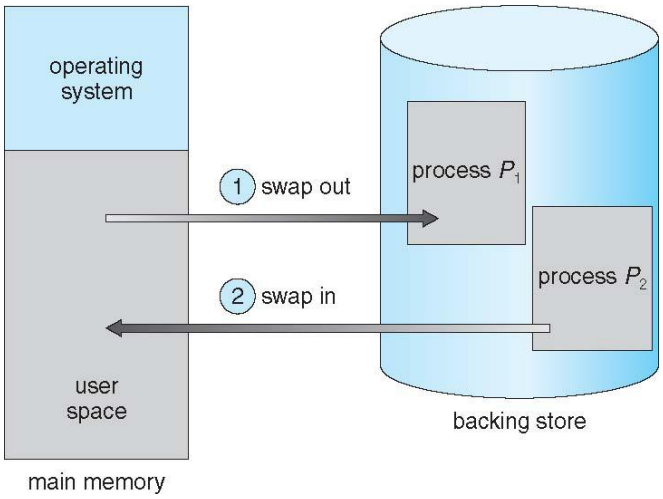
- A process must be in memory to be executed.
- **A process can be swapped temporarily out of memory to a backing store and then brought back into memory for continued execution. This process is called as Swapping.**
- Swapping allows the total physical address space of all processes to exceed the real physical memory of the system, and increases the degree of multiprogramming in a system.
- Swapping involves moving processes between main memory and a backing store. The backing store is commonly a fast disk.

EXAMPLE: Consider a multiprogramming environment with Round robin Scheduling**.** When the quantum time of a process expires the memory manager will swap out the process just finished and swap another process into the memory space.

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.



- The context-switch time in such a swapping system is fairly high.
- The major part of the swap time is transfer time. The total transfer time is directly proportional to the amount of memory swapped.
- If we want to swap a process, we must be sure that it is completely idle. A process that is waiting for any event such as I/O Operation to occur should not be swapped.
- A variant of swapping policy is used for priority based scheduling algorithms. If a higher priority process arrives and want service the memory manager can then swap the lower priority process and then load and execute the higher priority process.
- When the higher priority process finishes then the lower priority process can be swapped back in. This is also called as **Roll in and Roll out.**

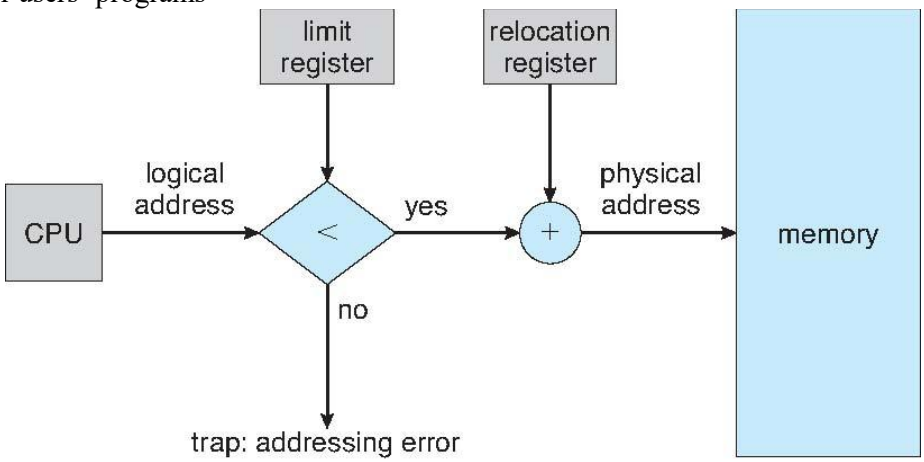
CONTIGUOUS MEMORY ALLOCATION:

- The main memory must accommodate both the operating system and the various user processes
- The memory is usually divided into two partitions: one for the resident operating system and one for the user processes.

- In Multiprogramming several user processes to reside in memory at the same time.
- The OS need to decide how to allocate available memory to the processes that are in the input queue waiting to be brought into memory.
- **In contiguous memory allocation, each process is contained in a single section of memory that is contiguous to the section containing the next process.**

MEMORY PROTECTION:

- We can prevent a process from accessing memory of other process.
- If we have a system with a relocation register together with a limit register we accomplish our goal.
- The relocation register contains the value of the smallest physical address; the limit register contains the range of logical addresses
- 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.
- Every address generated by a CPU is checked against these registers, we can protect both the operating system and the other users' programs



MEMORY ALLOCATION:

In Contiguous memory allocation the memory can be allocated in two ways

1) Fixed partition scheme
2) Variable partition scheme

Fixed partition scheme:

- One of the simplest methods for allocating memory is to divide memory into several fixed-sized
- Partitions. Each partition may contain exactly one process.
- Thus, the degree of multiprogramming is bound by the number of partitions.
- In this **multiple partition method**, when a partition is free, a process is selected from the input
- queue and is loaded into the free partition.
- When the process terminates, the partition becomes available for another process.
- **INTERNAL FRAGMENTATION:** In fixed size partitions, each process is allocated with a partition, irrespective of its size. **The allocated memory for a process may be slightly larger than requested memory; this memory that is wasted internal to a partition, is called as internal fragmentation.**

Variable Partition scheme:

- 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**.
- When a process is allocated space, it is loaded into memory, and it can then compete for CPU time. When a process terminates, it releases its memory, which the operating system may then fill with another process from the input queue.

- Os will have a list of available block sizes and an input queue. The operating system can order the input queue according to a scheduling algorithm.
- 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.
- When a process arrives and needs memory, the system searches the set for a hole that is large enough for this process.
- If the hole is too large, it is split into two parts. One part is allocated to the arriving process; the other is returned to the set of holes.
- When a process terminates, it releases its block of memory, which is then placed back in the set of holes. If the new hole is adjacent to other holes, these adjacent holes are merged to form one larger hole.
- The 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**

There are many solutions to this problem.

- **First fit: Allocate the first hole that is big enough.**
- **Best fit: Allocate the smallest hole that is big enough.**
- **Worst fit: Allocate the largest hole.**

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

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**, **so that the memory cannot be allocated to the process.**

COMPACTION: 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.

50 PERCENT RULE: The analysis of first fit, reveals that 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 rule**.

NON CONTIGUOUS MEMORY ALLOCATION:

The 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 memory is available.

Two complementary techniques achieve this solution:

- segmentation
- paging

1. SEGMENTATION:

Segmentation is a memory-management scheme that supports the programmer view of memory.

A logical address space is a collection of segments.

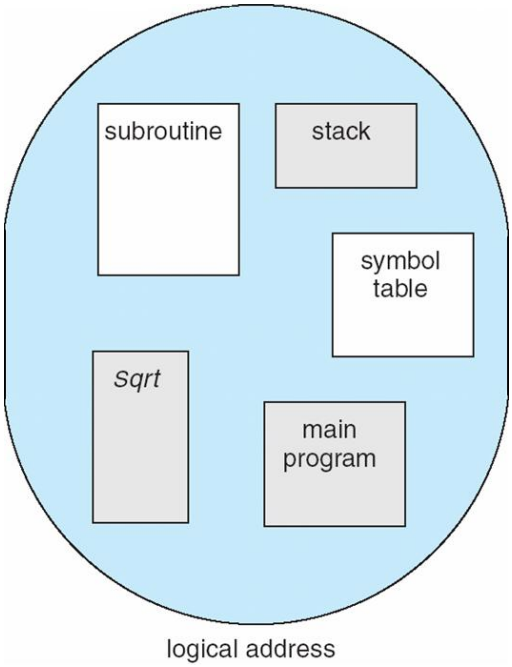
Each segment has a name and a length.

The logical addresses specify both the segment name and the offset within the segment. Each address is specified by two quantities: a segment name and an offset

Segment-number, offset>.

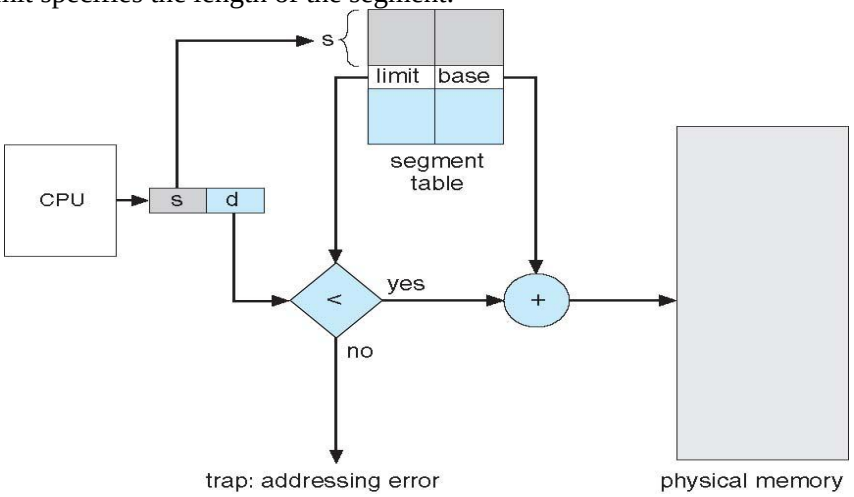
A C compiler might create separate segments for the following:

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



SEGMENTATION HARDWARE:

- The programmer can refer to objects in the program by a two-dimensional address (segment number and offset); the actual physical memory a one dimensional sequence of bytes.
- The two-dimensional user-defined addresses should be mapped into one-dimensional physical addresses.
- The mapping of logical address to physical address is done by a table called 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
- 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 between 0 and limit then hardware 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 an array of base–limit register pairs.
- Segmentation can be combined with paging.

Example: Consider five segments numbered from 0 through 4. 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).

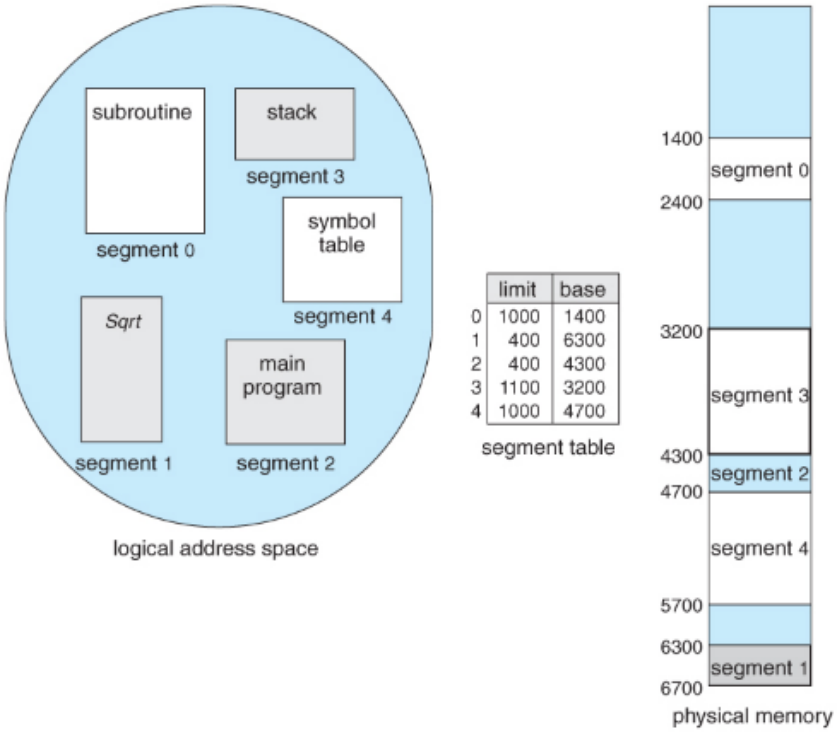


Figure 8.9 - Example of segmentation

- The 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.

2. PAGING:

- Paging involves breaking physical memory into fixed-sized blocks called **frames** and breaking logical memory into blocks of the same size called **pages**.
- Paging avoids external fragmentation and the need for compaction, whereas segmentation does not.
- When a process is to be executed, its pages are loaded into any available memory frames

PAGING HARDWARE:

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

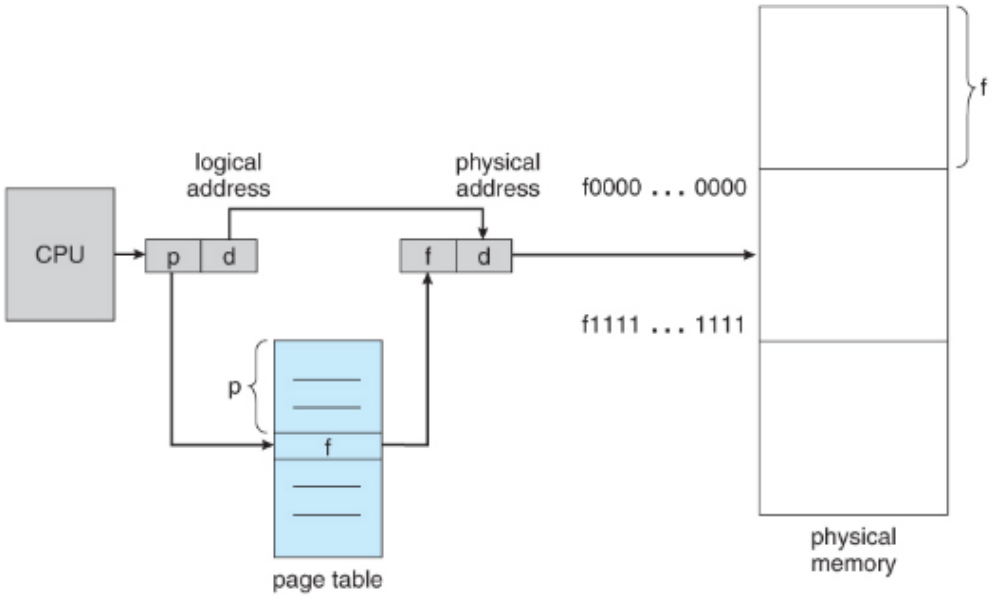


Figure 8.10 - Paging hardware

The page size is defined by the hardware. The size of a page is a power of 2, varying between 512 bytes and 1 GB per page.

If the size of the logical address space is 2*m*, and a page size is 2*n* bytes, then the high-order *m*− *n* bits of a logical address designate the page number, and the *n* low-order bits designate the page offset.

The logical address is given by

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

PAGING MODEL:

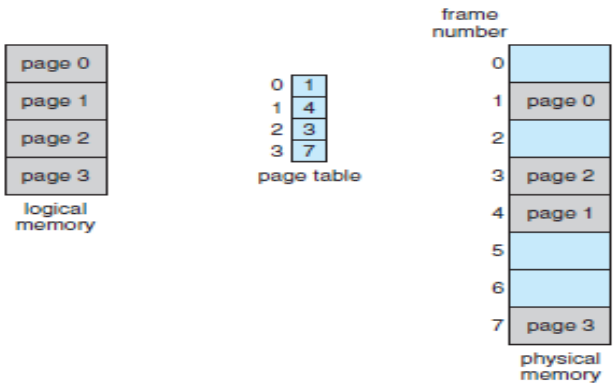


Figure 8.11 Paging model of logical and physical memory.

PAGING EXAMPLE:

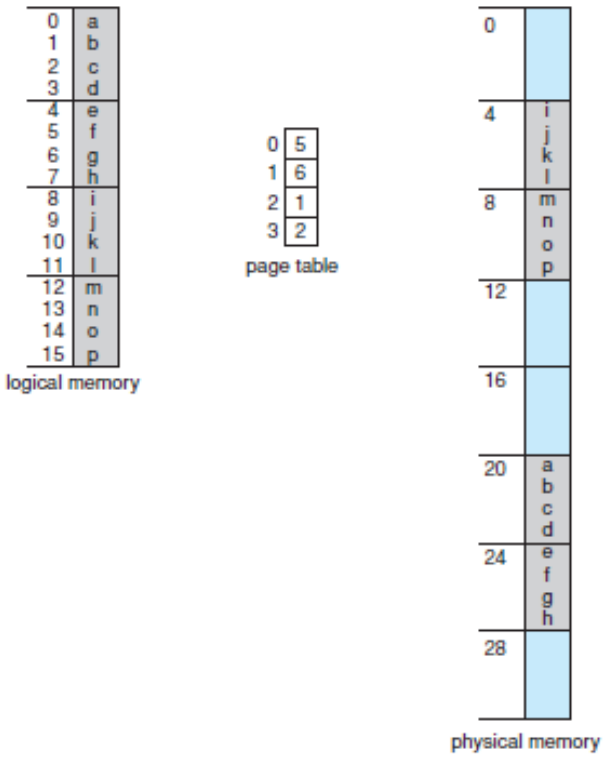


Figure 8.12 Paging example for a 32-byte memory with 4-byte pages.

- Consider the memory with the logical address, n= 2 and m = 4. Using a page size of 4 bytes and a physical memory of 32 bytes
- Logical address 0 is page 0, offset 0. Indexing into the page table, we find that page 0 is in frame 5. Thus, logical address 0 maps to physical address 20 [= (5 × 4) + 0].
- Logical address 3 (page 0, offset 3) maps to physical address 23 [= (5 × 4) + 3].
- Logical address 4 is page 1, offset 0; according to the page table, page 1 is mapped to frame 6.
- Thus, logical address 4 maps to physical address 24 [= (6 × 4) + 0].

FREE FRAME LIST:

- Each page of the process needs one frame. Thus, if the process requires n pages, at least n frames must be available in memory.
- If n frames are available, they are allocated to this arriving process.
- The operating system is managing physical memory and knows the allocation details of physical memory—which frames are allocated, which frames are available, how many total frames there are, and so on.
- This information is generally kept in a data structure called a frame table.

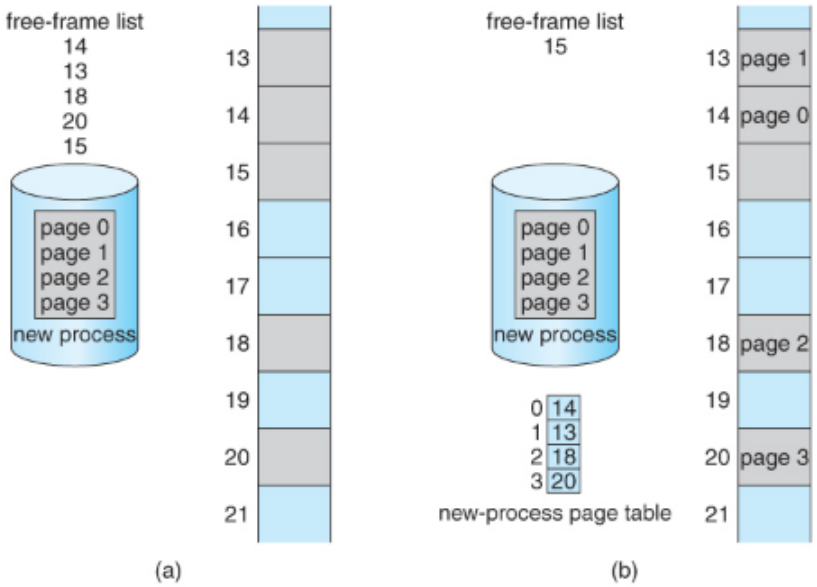
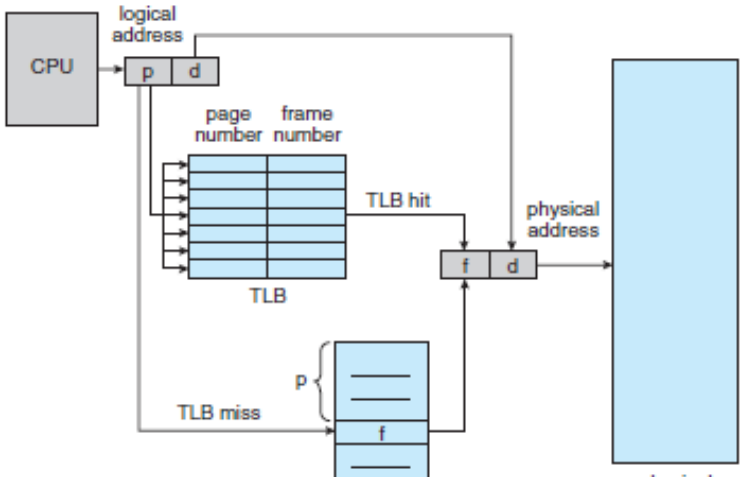


Figure 8.13 - Free frames (a) before allocation and (b) after allocation

HARDWARE SUPPORT:

- The hardware implementation of the page table can be done in several ways. The page table is implemented as a set of dedicated registers if the size of the page table is too small.
- If the size of the page table is too large then the page table is kept in main memory and a page table base register is used to point to the page table.
- When the page table is kept in main memory then two memory accesses are required to access a byte.
- One for accessing the page table entry, another one for accessing the byte.
- Thus the overhead of accessing the main memory increases.
- The standard solution to this problem is to use a special, small, fast lookup hardware cache called a translation look-aside buffer (TLB).



- Each entry in the TLB consists of two parts: a key (or tag) and a value.
- 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 **(TLB HIT),** its frame number is immediately available and is used to access memory.
- If the page number is not in the TLB (TLB **miss**), a memory reference to the page table must be made. When the frame number is obtained, we can use it to access memory.
- The percentage of times that the page number of interest is found in the TLB is called the hit ratio.
- The access time of a byte is said to be effective when the TLB hit ratio is high.
- Thus the effective access time is given by

Effective access time = TLB hit ratio* Memory access time +TLB miss ratio* (2*memory access time)

PROTECTION:

- Memory protection in a paged environment is accomplished by protection bits associated with each frame.
- One bit can define a page to be read–write or read-only. When 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.
- When the bit is set to invalid, the page is not in the process's logical address space.
- Page-table length register (PTLR),is used to indicate the size of the page table. This value is checked against every logical address to verify that the address is in the valid range for the process

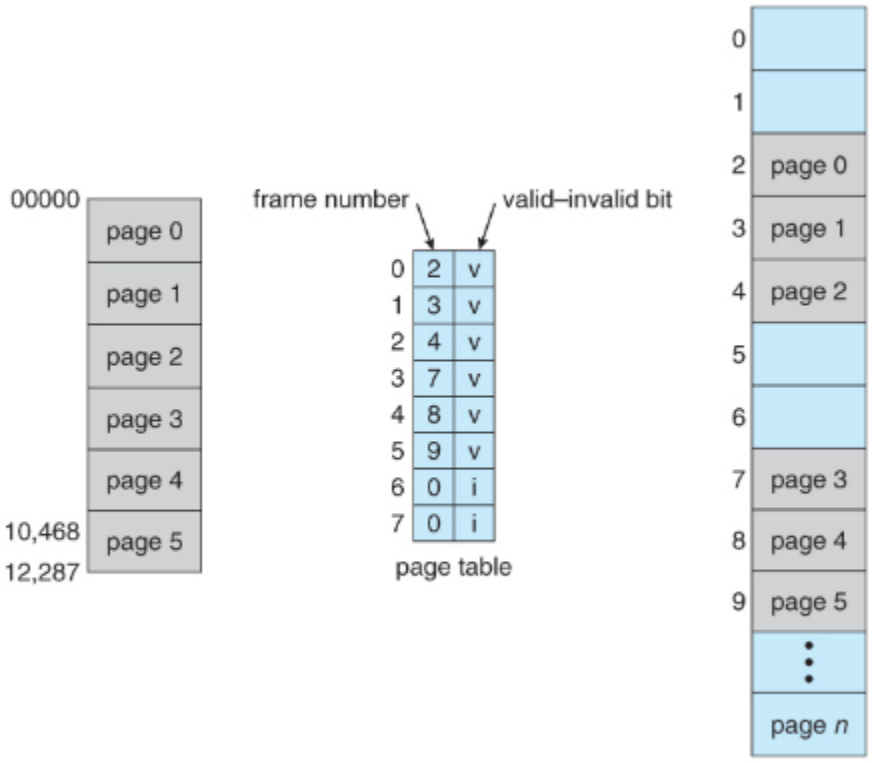


Figure 8.15 - Valid (v) or invalid (i) bit in page table

SHARED PAGES:

- An advantage of paging is the possibility of sharing common code.
- If the code is reentrant code (or pure code), however, it can be shared.
- Reentrant code is non-self-modifying code: it never changes during execution. Thus, two or more processes can execute the same code at the same time

EXAMPLE: Consider three processes that share a page editor which is of three pages. Each process has its own data page.

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.

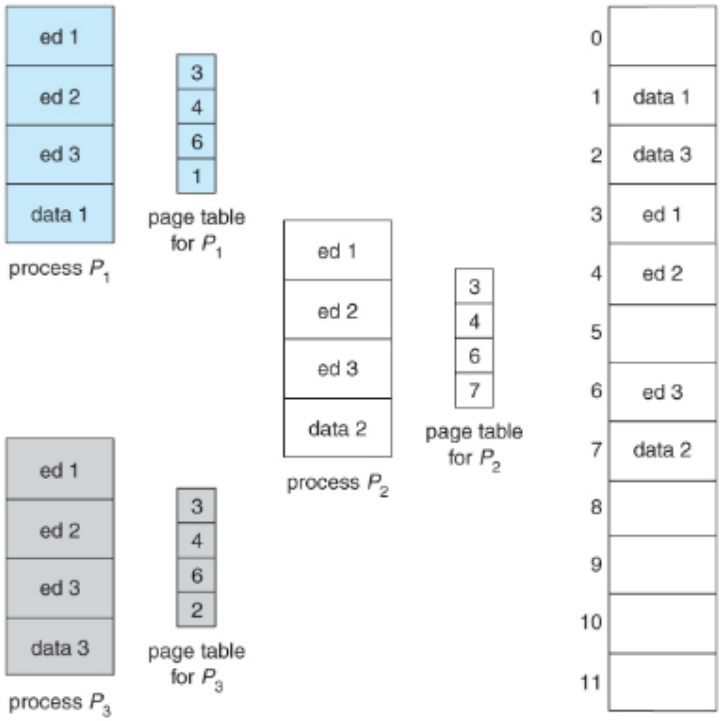


Figure 8.16 - Sharing of code in a paging environment

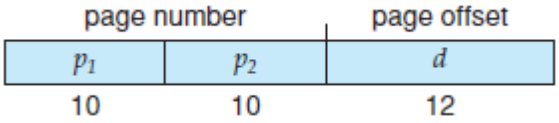
STRUCTURE OF PAGE TABLE:

The structure of page table includes

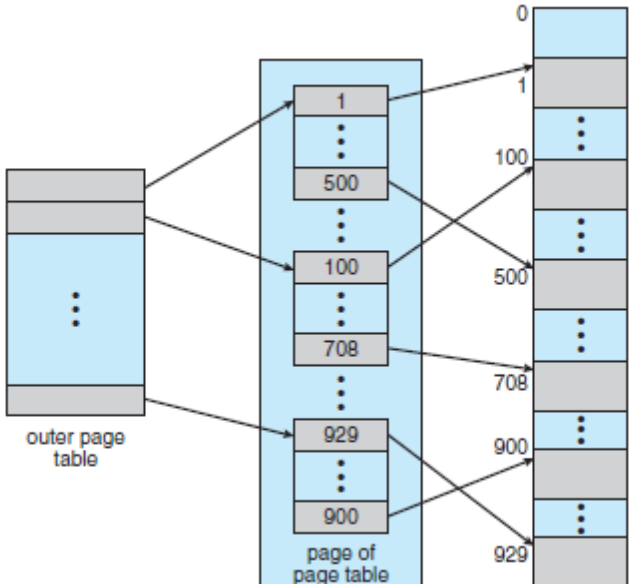
- Hierarchical paging
- Hashed page table
- Inverted page table.

HIERARCHIAL PAGING:

- In Computer a system that has a 32 bit logical address space the page table becomes too large. Each process may need upto 4MB of Physical address space for the page table alone.
- One simple solution to this problem is to divide the page table into smaller pieces. One way is to use a two-level paging algorithm, in which the page table itself is also paged.
- Consider the system with a 32-bit logical address space and a page size of (2 12)4 KB.
- A logical address is divided into a page number consisting of 20 bits and a page offset consisting of 12 bits.
- The page number is further divided into a 10-bit page number and a 10-bit page offset.



- Here p1 is an index into the outer page table and p2 is the displacement within the page of the inner page table.



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

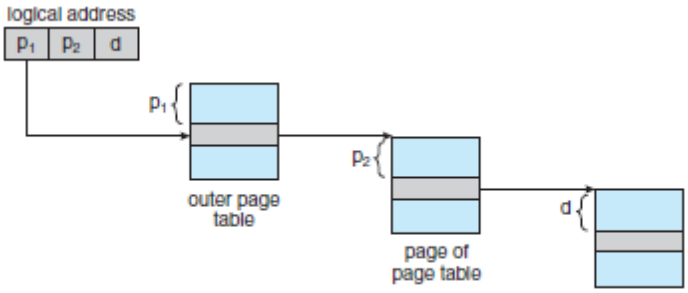
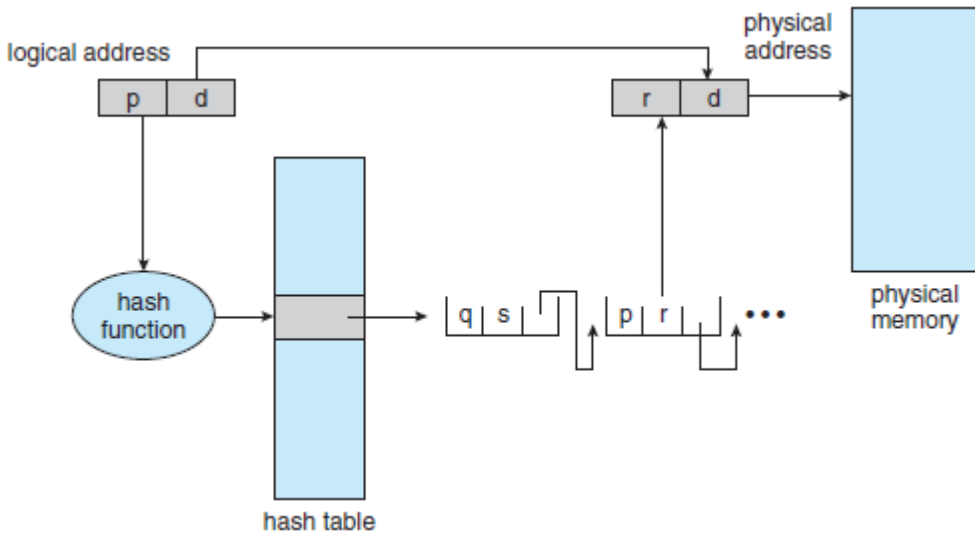


Figure 8.18 Address translation for a two-level 32-bit paging architecture.

HASHED PAGE TABLES:

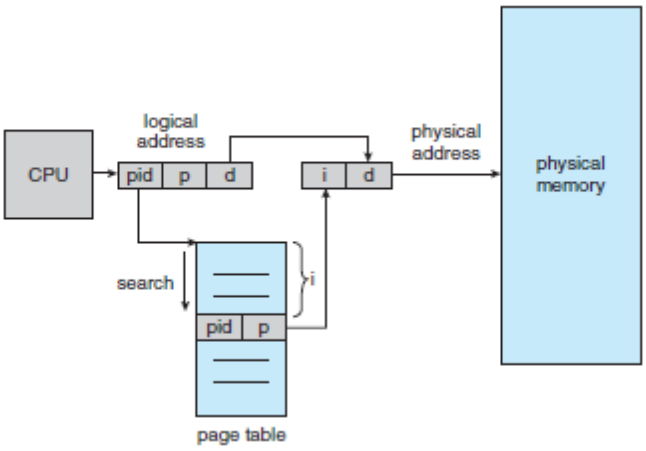
- A common approach for handling address spaces larger than 32 bits is to use a **hashed page table**, with the hash value being the virtual page number.
- Each entry in the hash table contains a linked list of elements that hash to the same location
- Each element consists of three fields:
    - virtual page number,
    - value of the mapped page frame
    - pointer to the next element in the linked list
- 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.



- A variation to hashed page table is **clustered page tables**, which are similar to hashed page tables except that each entry in the hash table refers to several pages (such as 16) rather than a single page.

INVERTED PAGE TABLES:

- Each process has an associated page table. The page table has one entry for each page that the process is using.
- The table is sorted by virtual address, the operating system 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.
- To solve this problem, we can use an inverted page table.
- An inverted page table has one entry for each 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 Logical address in the system consists of a triple :< **process-id, page-number, offset>.**
- Each inverted page-table entry is a pair **<process-id, page-number**>
- When a 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 a match is found—say, at entry i—then the physical address <i, offset> is generated..



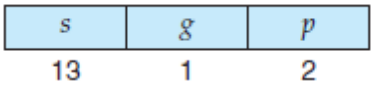
INTEL 32 AND 64-BIT ARCHITECTURES [SEGMENTATION WITH PAGING]

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

- The linear address is then given to the paging unit, which in turn generates the physical address in main memory.
- Thus, the segmentation and paging units form the equivalent of the memory-management unit

IA-32 Segmentation:

- The IA-32 architecture allows a segment to be as large as 4 GB, and the maximum number of segments per process is 16 K.
- The logical address space of a process is divided into two partitions. Information about the process that are private to the process is kept in the local descriptor table (LDT);
- The information that is shared among other processes is kept in the global descriptor table (GDT).
- Each entry in the LDT and GDT consists of an 8-byte segment descriptor with detailed information about a particular segment, including the base location and limit of that segment.
- The logical address is a pair (selector, offset), where the selector is a 16-bit number:



- s designates the segment number, g indicates whether the segment is in the GDT or LDT, and p deals
- with protection.
- The offset is a 32-bit number specifying the location of the byte.
- The segment registers points to the appropriate entry in the LDT or GDT.
- The base and limit information about the segment is used to generate a linear address.

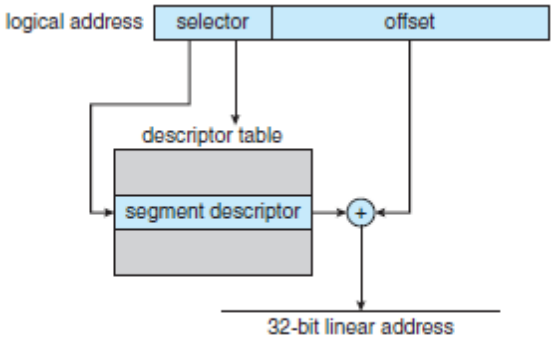
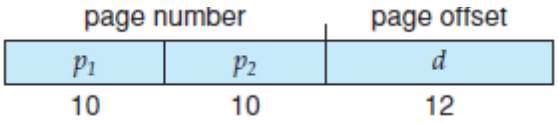


Figure 8.22 IA-32 segmentation.

IA -32 PAGING:

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



- The 10 high-order bits reference an entry in the outermost page table, which IA-32 terms the page directory.
- The page directory entry points to an inner page table that is indexed by the contents of the innermost 10 bits in the linear address.
- Finally, the low-order 12 bits refer to the offset in the 4-KB page pointed to in the page table.
- **Intel adopted a page address extension (PAE), which allows 32-bit processors to access a physical address space larger than 4 GB**
- PAE also increased the page-directory and page-table entries from 32 to 64 bits in size, which allowed the base address of page tables and page frames to extend from 20 to 24 bits
- The 10 high-order bits reference an entry in the outermost page table, which IA-32 terms the page directory.
- The page directory entry points to an inner page table that is indexed by the contents of the innermost 10 bits in the linear address.

- Finally, the low-order 12 bits refer to the offset in the 4-KB page pointed to in the page table.
- **Intel adopted a page address extension (PAE), which allows 32-bit processors to access a physical address space larger than 4 GB**
- PAE also increased the page-directory and page-table entries from 32 to 64 bits in size, which allowed the base address of page tables and page frames to extend from 20 to 24 bits

VIRTUAL MEMORY:

- Virtual memory is a memory management technique that allows the execution of processes that are not completely in memory.
- In some cases during the execution of the program the entire program may not be needed, such as error conditions, menu selection options etc.
- The virtual address space of a process refers to the logical view of how a process is stored in memory.

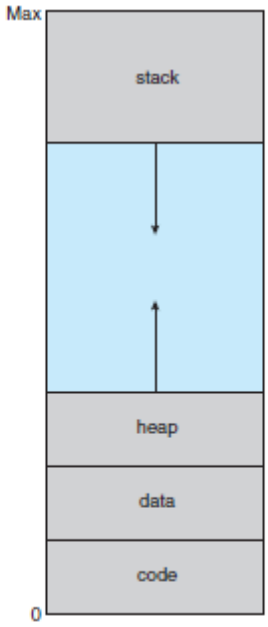


Figure 9.2 Virtual address space.

- The heap will grow upward in memory as it is used for dynamic memory allocation.
- The stack will 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.**
- Sparse address space can be filled as the stack or heap segments grow or if we wish to dynamically link libraries
- Virtual memory allows files and memory to be shared by two or more processes through page sharing

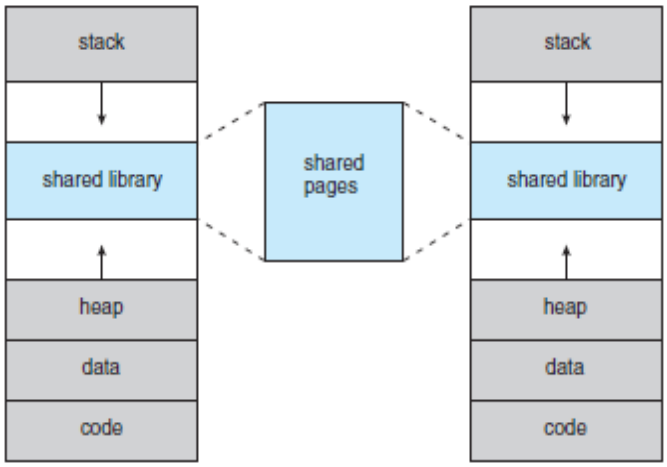


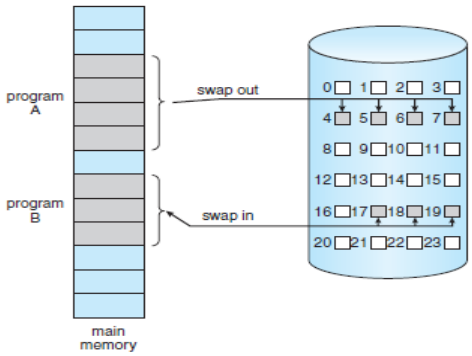
Figure 9.3 Shared library using virtual memory.

ADVANTAGES:

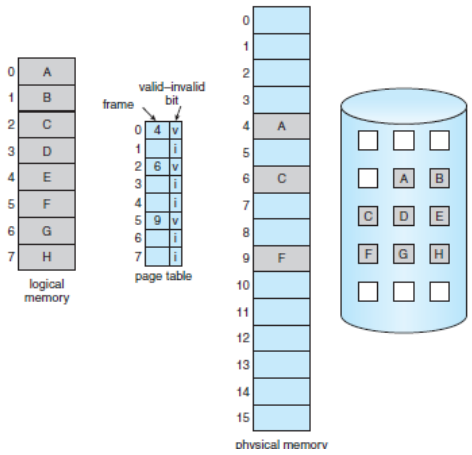
- One major advantage of this scheme is that programs can be larger than physical memory
- Virtual memory also allows processes to share files easily and to implement shared memory.
- Increase in CPU utilization and throughput.
- Less I/O would be needed to load or swap user programs into memory

DEMAND PAGING:

- **Demand paging is the process of loading the pages only when they are demanded by the process during execution. Pages that are never accessed are thus never loaded into physical memory.**
- A demand-paging system is similar to a paging system with swapping where processes reside in secondary memory



- When we want to execute a process, we swap it into memory. Rather than swapping the entire process into memory we use a **lazy swapper** that never swaps a page into memory unless that page will be needed.
- Lazy swapper is termed to as pager in demand paging.
- When a process is to be swapped in, the pager guesses which pages will be used before the process is swapped out again. Instead of swapping in a whole process, the pager brings only those pages into memory.
- Os need the 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.
- If the bit is set to ―valid,‖ the associated page is both legal and in memory.
- If the bit is set to ―invalid,‖ the page either is not valid or is valid but is currently on the disk.



PAGE FAULT: If the process tries to access a page that was not brought into memory, then it is called as a page fault. Access to a page marked invalid causes a page fault.
The paging hardware, will notice that the invalid bit is set, causing a trap to the operating system.

PROCEDURE FOR HANDLING THE PAGE FAULT:
1. 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. Find a free frame
4. 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. Restart the instruction that was interrupt. Though it had always been in memory.

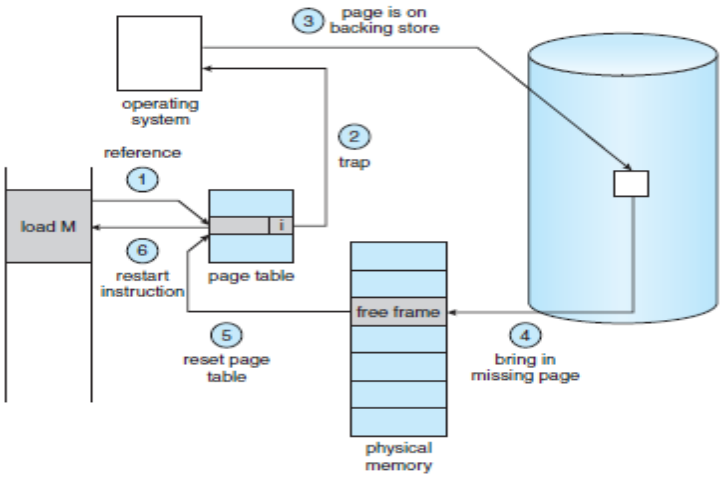


Figure 9.6 Steps in handling a page fault.

PURE DEMANDPAGING: The process of executing a program with no pages in main memory is called as pure demand paging. This never brings a page into memory until it is required.
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. The secondary memory is usually a high-speed disk. It is known as the swap device, and the section of disk used for this purpose is known as **swap space.**

PERFORMANCE OF DEMAND PAGING:

- Demand paging can affect the performance of a computer system.
- The effective access time for a demand-paged memory is given by
- **effective access time = (1 − p) × ma + p × page fault time.**
- The memory-access time, denoted *ma*, ranges from 10 to 200 nanoseconds.
- If there is no page fault then the effective access time is equal to the memory access time.
- If a page fault occurs, we must first read the relevant page from disk and then access the desired word.
- There are three major components of the page-fault service time:
    1. Service the page-fault interrupt.
    2. Read in the page.
    3. 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) × (200) + p (8 milliseconds)
  = (1 − p) × 200 + p × 8,000,000
  = 200 + 7,999,800 × p.
- **Effective access time is directly proportional to the page-fault rate.**

PAGE REPLACEMENT:

NEED FOR PAGE REPLACEMENT:

- Page replacement is basic to demand paging
- If a page requested by a process is in memory, then the process can access it. If the requested page is not in main memory, then it is page fault.
- When there is a page fault the OS decides to load the pages from the secondary memory to the main memory. It looks for the free frame. If there is no free frame then the pages that are not currently in use will be swapped out of the main memory, and the desired page will be swapped into the main memory.
- **The process of swapping a page out of main memory to the swap space and swapping in the desired page into the main memory for execution is called as Page Replacement.**

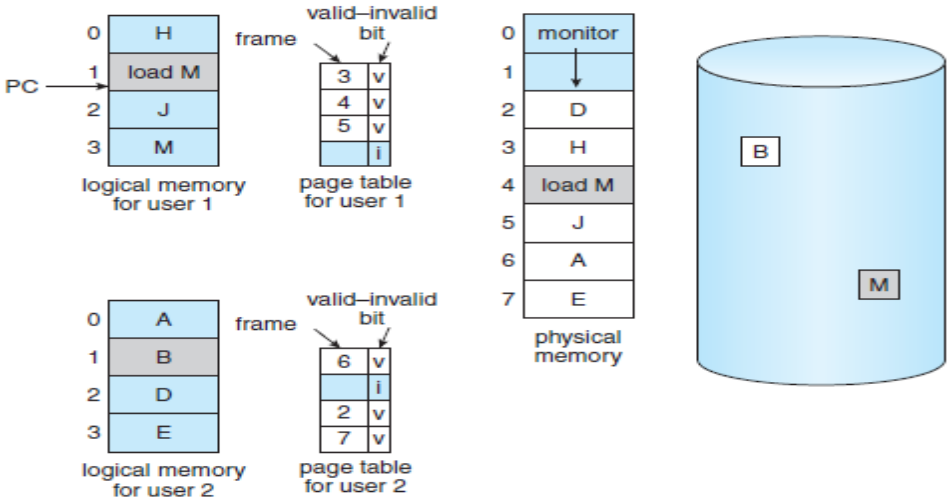


Figure 9.9 Need for page replacement.

STEPS IN 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. Continue the user process from where the page fault occurred.

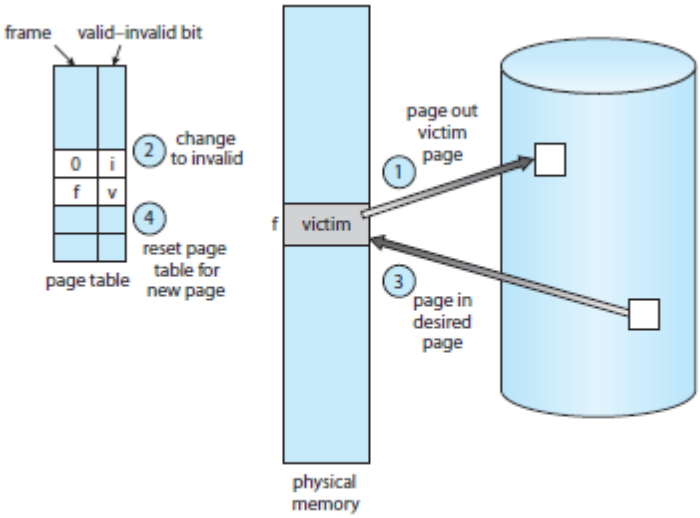


Figure 9.10 Page replacement.

- 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.
- This overhead can be reduced by using a modify bit (or dirty bit).
- When this scheme is used, each page or frame has a modify bit associated with it in the hardware.
- **MODIFY BIT:** The modify bit for a page is set by the hardware whenever any 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.

PAGE REPLACEMENT ALGORITHMS:

- 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.
- The string of memory references made by a process is called a **reference string**.

There are many different page-replacement algorithms that includes

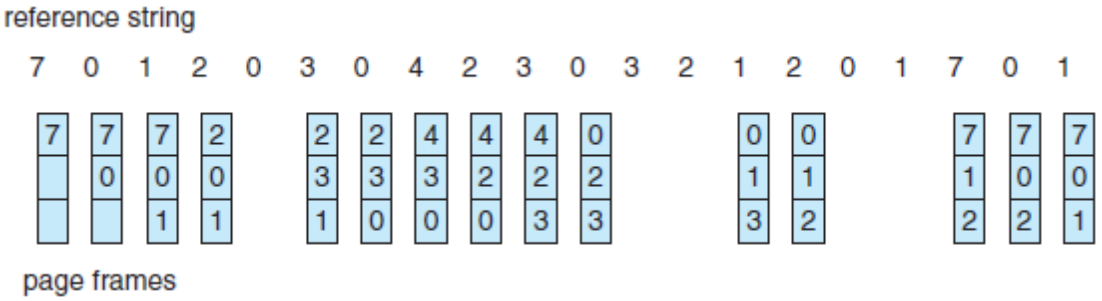
- FIFO page Replacement
- Optimal Page Replacement
- LRU Page Replacement
- LRU Approximation page Replacement algorithm
- Counting Based Page Replacement Algorithm
- Page Buffering Algorithm

FIFO PAGE REPLACEMENT:

- The simplest page-replacement algorithm is a first-in, first-out (FIFO) algorithm.
- A FIFO replacement algorithm replaces the oldest page that was brought into main memory.

EXAMPLE: Consider the Reference string 7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1 for a memory with three frames.

- The three frames are empty initially.
- The first three references (7, 0, 1) cause page faults and are brought into these empty frames.
- The algorithm has 15 faults.



- Page 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.
- Because of this replacement, the next reference, to 0, will fault. Page 1 is then replaced by page 0. The process continues until all the pages are referenced.

Advantages:

- The FIFO page-replacement algorithm is easy to understand and program

Disadvantages:

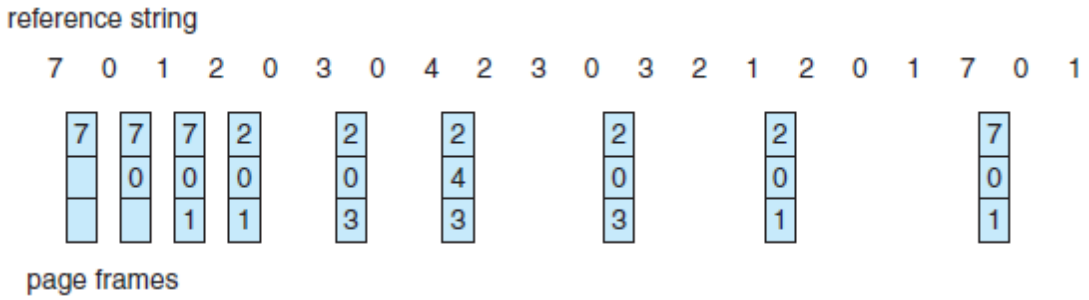
- The Performance is not always good.
- It Suffers from Belady's Anomaly.

BELADY'S ANOMALY: The page fault increases as the number of allocated memory frame increases. This unexpected result is called as Belady's Anomaly.

OPTIMAL PAGE REPLACEMENT

- This algorithm has the lowest page-fault rate of all algorithms and will never suffer from Belady's anomaly
- Optimal page replacement algorithm Replace the page that will not be used for the longest period of time

EXAMPLE: Consider the Reference string 7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1 for a memory with three frames.



- The Optimal replacement algorithm produces Nine faults
- 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.

Advantages:

- optimal replacement is much better than a FIFO algorithm

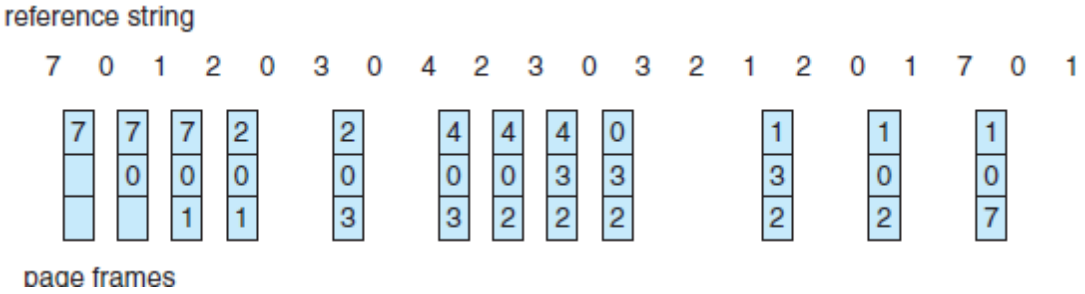
Disadvantage:

- The optimal page-replacement algorithm is difficult to implement, because it requires future knowledge of the reference string.

LRU PAGE REPLACEMENT

- The Least Recently used algorithm replaces a page that has not been used for a longest period of time.
- LRU replacement associates with each page the time of that page's last use.
- It is similar to that of Optimal page Replacement looking backward in time, rather than forward.

EXAMPLE: Consider the Reference string 7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1 for a memory with three frames.



- The LRU algorithm produces twelve faults
- The first three references cause faults that fill the three empty frames.
- The reference to page 2 replaces page 7, because page 7 has not been used for a longest period of time, when we look backward.
- The Reference to page 3 replaces page 1, because page 1 has not been used for a longest period of time.
- 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.

Advantages:

- The LRU policy is often used as a page-replacement algorithm and is considered to be good.
- LRU replacement does not suffer from Belady's anomaly.

Disadvantage:

- The problem is to determine an order for the frames defined by the time of last use.
- Two implementations are feasible:

• **Counters**. We associate with each page-table a time-of-use field and add to the CPU a logical clock or counter. The clock is incremented for every memory reference. Whenever a reference to a page is made, the contents of the clock register are copied to the time-of-use field in the page-table entry for that page. So we can find the ―time‖ of the last reference to each page.

• **Stack**. Another approach to implementing LRU replacement is to keep a stack of page numbers. 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

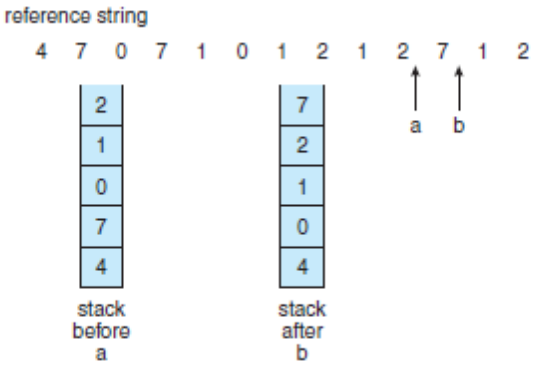


Figure 9.16 Use of a stack to record the most recent page references.

STACK ALGORITHM:

A stack algorithm is an algorithm for which it can be shown that the set of pages in memory for *n* frames is always a ***subset*** of the set of pages that would be in memory with *n* + 1 frames.

LRU APPROXIMATION PAGE REPLACEMENT ALGORITHM:

- The system provides support to the LRU algorithm in the form of a bit called Reference bit.

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).

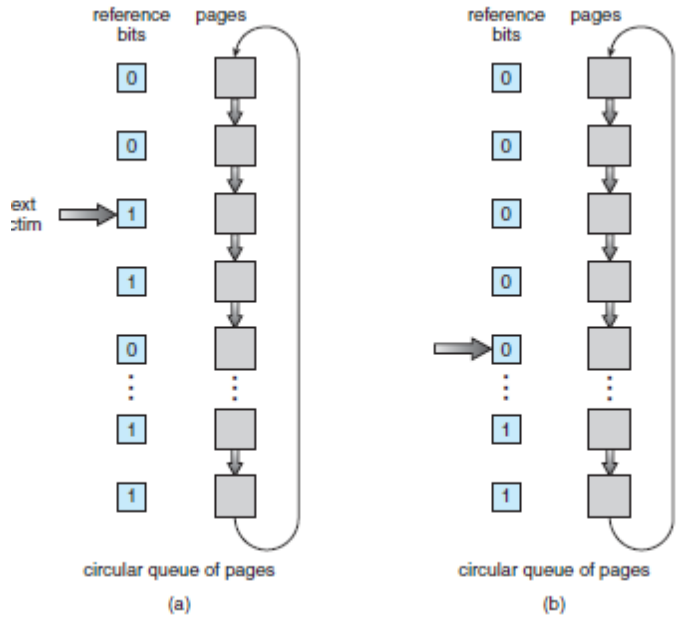
- Reference bits are associated with each entry in the page table.
- Initially, all bits are cleared (to 0) by the operating system. As a user process executes, the bit associated with each page referenced is set (to 1) by the hardware.
- After some time, we can determine which pages have been used and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

a) Additional-reference-bits algorithm

- The additional ordering information can be gained 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 (say, every 100 milliseconds), a timer interrupt transfers control to the operating system.
- These 8-bit shift registers contain the history of page use for the last eight time periods.
- If the shift register contains 00000000, for example, then the page has not been used for eight time periods.
- A page that is used at least once in each period has a shift register value of 11111111.
- A page with a history register value of 11000100 has been used more recently than one with a value of 01110111.
- Thus the page with the lowest number is the LRU page, and it can be replaced.

b) Second-Chance Algorithm:

- The basic algorithm of second-chance replacement is a FIFO replacement algorithm.
- When a page has been selected, however, 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 FIFO page.
- When a page gets a second chance, its reference bit is cleared, and its arrival time is reset to the current time. Thus, a page that is given a second chance will not be replaced until all other pages have been replaced
- 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.
- Once a victim page is found, the page is replaced, and the new page is inserted in the circular queue in that position



c. Enhanced Second-Chance Algorithm

- We can enhance the second-chance algorithm by considering the reference bit and the modify as an ordered pair
- The order is {Reference bit, Modify bit}
- we have the following four possible classes:
- 0, 0) neither recently used nor modified—best page to replace
- 0, 1) not recently used but modified—not quite as good
- 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
- Here we give preference to those pages that have been modified in order to reduce the number of I/Os required. Thus the modified pages will not be replaced before writing it to the disk.

COUNTING-BASED PAGE REPLACEMENT

- We can keep a counter of the number of references that have been made to each page.

This method includes two schemes

- **Least frequently used (LFU) page-replacement:** 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.
- **Most frequently used (MFU)** page-replacement algorithm: 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 ALGORITHM:

- systems commonly keep a pool of free frames
- When a page fault occurs, a victim frame is chosen as and 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.
- 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

ALLOCATION OF FRAMES:

- Allocation of frames deals with how the operating system allocates the fixed amount of free memory among the various processes.
- Consider a single-user system with 128 KB of memory composed of pages 1 KB in size.
- This system has 128 frames. The operating system may take 35 KB, leaving 93 frames for the user process.
- Under pure demand paging, all 93 frames would initially be put on the free-frame list. When a user process started execution, it would generate a sequence of page faults.
- The first 93 page faults would all get free frames from the free-frame list. When the free-frame list was exhausted, a page-replacement algorithm would be used to select one of the 93 in-memory pages to be replaced with the 94th, and so on.
- When the process terminated, the 93 frames would once again be placed on the free-frame list.
- Operating system allocates all its buffer and table space from the free-frame list.
- When this space is not in use by the operating system, it can be used to support user paging.

a) Minimum Number of Frames

OS cannot allocate more than the total number of available frames (unless there is page sharing). It must also allocate at least a minimum number of frames as required by the process.

- The reason for allocating at least a minimum number of frames involves performance. As the number of frames allocated to each process decreases, the page-fault rate increases, slowing process execution.
- Consider a machine in which all memory-reference instructions may reference only one memory address. 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 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.
- When the first indirection occurs, a counter is set to the maximum number of indirections allowed; the counter is then decremented for each successive indirection for this instruction. If the counter is decremented to 0, a trap occurs.
- The minimum number of frames is defined by the computer architecture and the maximum number is defined by the amount of available physical memory.

b) Allocation Algorithms

The Operating system makes use of various allocation algorithms to allocate the frames to the user process.

- Equal Allocation
- Proportional Allocation

EQUAL ALLOCATION:

- The easiest way to split *m* frames among *n* processes is to give everyone an equal share, *m*/*n* frames.
- 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**.

DISADVANTAGE:

- Consider a system with a 1-KB frame size. If a small student process of 10 KB and an interactive database of 127 KB are the only two processes running in a system with 62 free frames, each process will be allocated with 31 frames.
- The student process does not need more than 10 frames, so the other 21 are wasted.

PROPORTIONAL ALGORITHM:

- The Os allocates the available memory to each process according to its size.
- Let the size of the virtual memory for process *pi* be *si* , given by
- If the total number of available frames is *m,* we allocate *ai* frames to process *pi*, where *ai* is approximately $a_i = s_i/S \times m$.
- Consider a system with a 1-KB frame size. If a small student process of 10 KB and an interactive database of 127 KB are the only two processes running in a system with 62 free frames.
- we would split 62 frames between two processes, one of 10 pages and one of 127 pages, by allocating 4 frames and 57 frames, respectively, since $10/137 \times 62 \approx 4$, and $127/137 \times 62 \approx 57$.
- Thus both processes share the available frames according to their ―needs,‖ rather than equally.
- A high-priority process is treated the same as a low-priority process.
- we may want to give the high-priority process more memory to speed its execution, to the detriment of low-priority processes.

- One solution is to use a proportional allocation scheme wherein the ratio of frames depends not on the relative sizes of processes but rather on the priorities of processes or on a combination of size and priority.

c) Global versus Local Allocation

- Another important factor in the way frames are allocated to the various processes is page replacement.

Page-replacement algorithms are divided into two broad categories:

  - **global replacement**
  - **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;
- Local replacement requires that each process select from only its own set of allocated frames.

EXAMPLE: consider an allocation scheme where in we allow high-priority processes to select frames from low-priority processes for replacement.

- A process can select a replacement from among its own frames or the frames of any lower-priority process. This approach allows a high-priority process to increase its frame allocation
- 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.
- In local replacement, the set of pages in memory for a process is affected by the paging behavior of only that process.

d) Non-Uniform Memory Access

- In some systems, a given CPU can access some sections of main memory faster than it can access others. These performance differences are caused by how CPUs and memory are interconnected in the system.
- A system is made up of several system boards, each containing multiple CPUs and some memory.
- 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.
- The systems in which the memory access time are uniform is called as Uniform memory access.
- Systems in which memory access times vary significantly are known collectively as **non-uniform memory access (NUMA)** systems, and they are slower than systems in which memory and CPUs are located on the same motherboard.
- Managing which page frames are stored at which locations can significantly affect performance in NUMA systems.
- If we treat memory as uniform in such a system, CPUs may wait significantly longer for memory access.

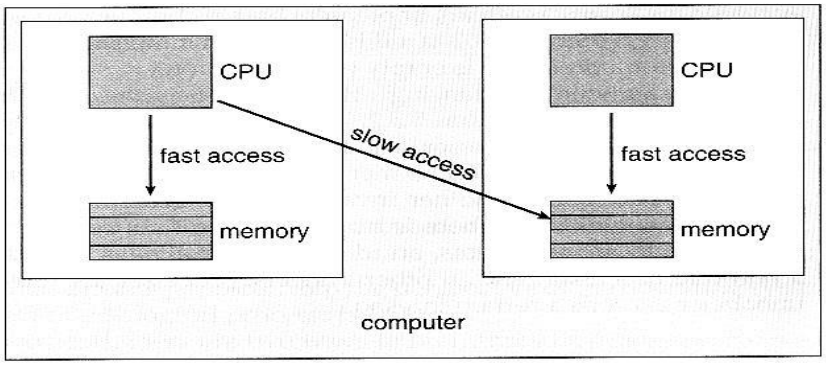


Figure 5.9 NUMA and CPU scheduling.

- The goal is to have memory frames allocated ―as close as possible‖ to the CPU on which the process is running so that the memory access can be faster.
- In NUMA systems the scheduler tracks 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

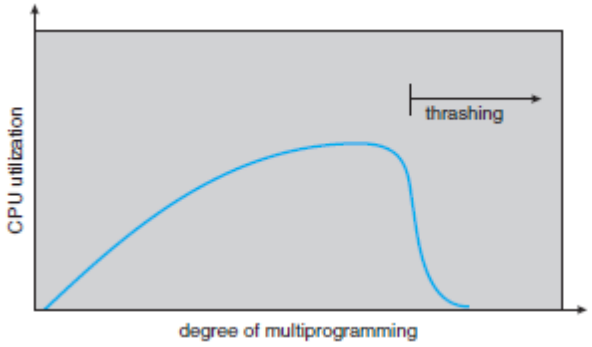
the process close to the CPU on which it is being scheduled, then improved cache hits and decreased memory access times will result.

THRASHING:

- **If the process does not have the number of frames it needs to support pages in active use, it will quickly page-fault. At this point, it must replace some page. If all its pages are in active use, it must replace a page that will be needed again right away. So it quickly faults again, and again, and again, replacing pages that it must bring back in immediately. This high paging activity is called thrashing.**
- A process is thrashing if it is spending more time paging than executing.

Causes 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.
- 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.
- **A global page-replacement algorithm is used**; it replaces pages without regard to the process to which they belong.
- 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 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.

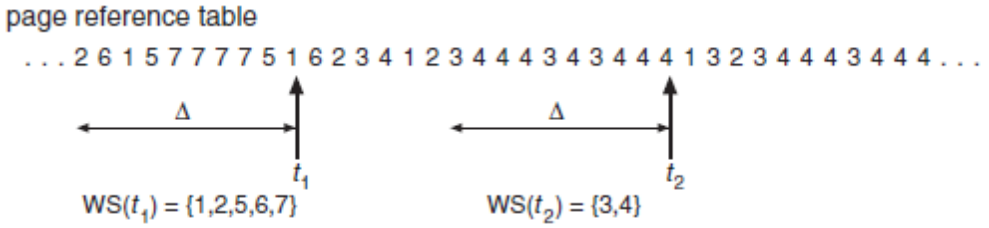


- At this point, to increase CPU utilization and stop thrashing, we must ***decrease*** the degree of Multi programming.
- We can limit the effects of thrashing by using a **local replacement algorithm.** With local replacement, if one process starts thrashing, it cannot steal frames from another process, so the page fault of one process does not affect the other process.
- To prevent thrashing, we must provide a process with as many frames as it needs. The Os need to know how many frames are required by the process.
- The working-set strategy starts by looking at how many frames a process is actually using. This approach defines the **locality model** of process execution.
- A locality is a set of pages that are actively used together. A program is generally composed of several different localities, which may overlap.
- Suppose we allocate enough frames to a process to accommodate its current locality. It will fault for the pages in its locality until all these pages are in memory; then, it will not fault again until it changes localities.
- If we do not allocate enough frames to accommodate the size of the current locality, the process will thrash, since it cannot keep in memory all the pages that it is actively using.

Working-Set Model

- The **working-set model** is based on the assumption of locality.
- 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.

EXAMPLE: Consider the sequence of memory references shown. 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*}*.



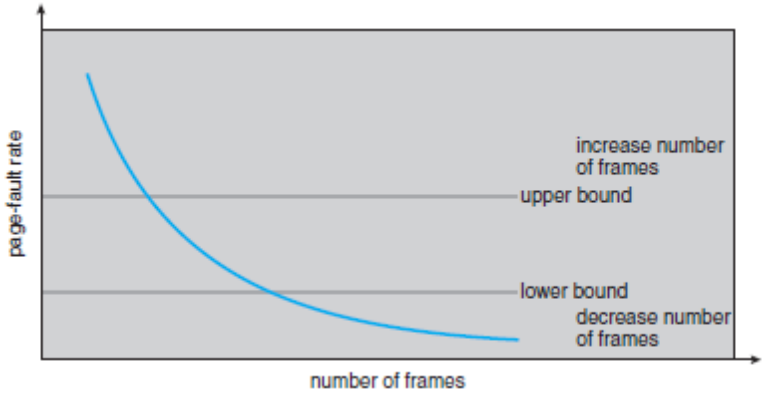
- If Δ is too small, it will not encompass the entire locality;
- If Δ is too large, it may overlap several localities.
- If Δ is infinite, the working set is the set of pages touched during the process execution.
- The most important property of the working set, then, is its size. If we compute the working-set size, *WSSi* , for each process in the system, we can then consider that
- Here *D* is the total demand for frames.
- Thus, process *i* needs *WSSi* frames. 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 there are enough extra frames, another process can be initiated.
- If the sum of the working-set sizes increases, exceeding the total number of available frames, the operating system selects a process to suspend.
- This working-set strategy prevents thrashing while keeping the degree of multiprogramming as high as possible.

Page-Fault Frequency

- The page fault frequency is calculated by the total number of faults to the total number of references.

Page fault frequency = No. of page Faults / No. of References

- 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.
- Establish an 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.
- If the page-fault rate increases and no free frames are available, we must select some process and swap it out to backing store.
- The freed frames are then distributed to processes with high page-fault rates

ALLOCATING KERNEL MEMORY

- When a process running in user mode requests additional memory, pages are allocated from the list of free page frames maintained by the kernel.
- If a user process requests a single byte of memory, internal fragmentation will result, as the process will be granted an entire page frame.
- Kernel memory is often allocated from a free-memory pool different from the list used to satisfy ordinary user-mode processes. There are two primary reasons for this:
- The kernel requests memory for data structures of varying sizes, some of which are less than a page in size
- Certain hardware devices interact directly with physical memory and consequently may require memory residing in physically contiguous pages.

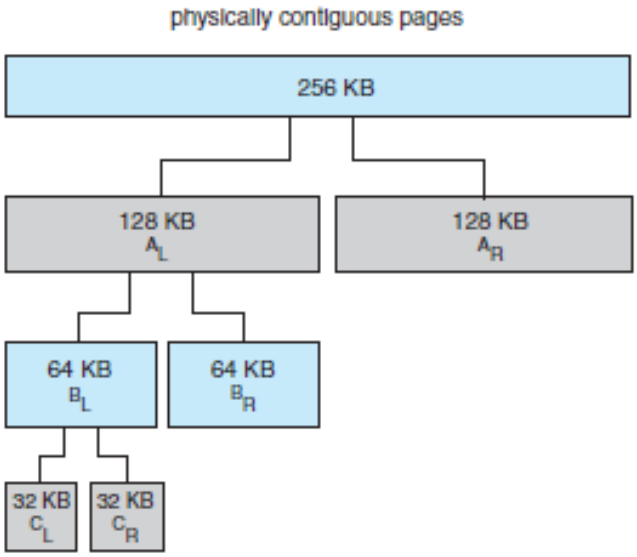
We examine two strategies for managing free memory that is assigned to kernel processes:

- Buddy system
- Slab allocation

Buddy Systems:

The buddy system allocates memory from a fixed-size segment consisting of physically contiguous pages.
Memory is allocated from this segment using a **power-of-2 allocator**, which satisfies requests in units sized as a power of 2 (4 KB, 8 KB, 16 KB, and so forth).

EXAMPLE: Assume the size of a memory segment is initially 256 KB and the kernel requests 21 KB of memory. The segment is initially divided into two **buddies**—which we will call *AL* and *AR*—each 128 KB in size.



One of these buddies is further divided into two 64-KB buddies— *BL* and *BR*. However, the next-highest power of 2 from 21 KB is 32 KB so either *BL* or *BR* is again divided into two 32-KB buddies, *CL* and *CR*.
One of these Buddies is used to satisfy the request.

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

Disadvantage: It Leads to Internal Fragmentation.

The obvious drawback to the buddy system is that rounding up to the next highest power of 2 is very likely to cause fragmentation within allocated segments.

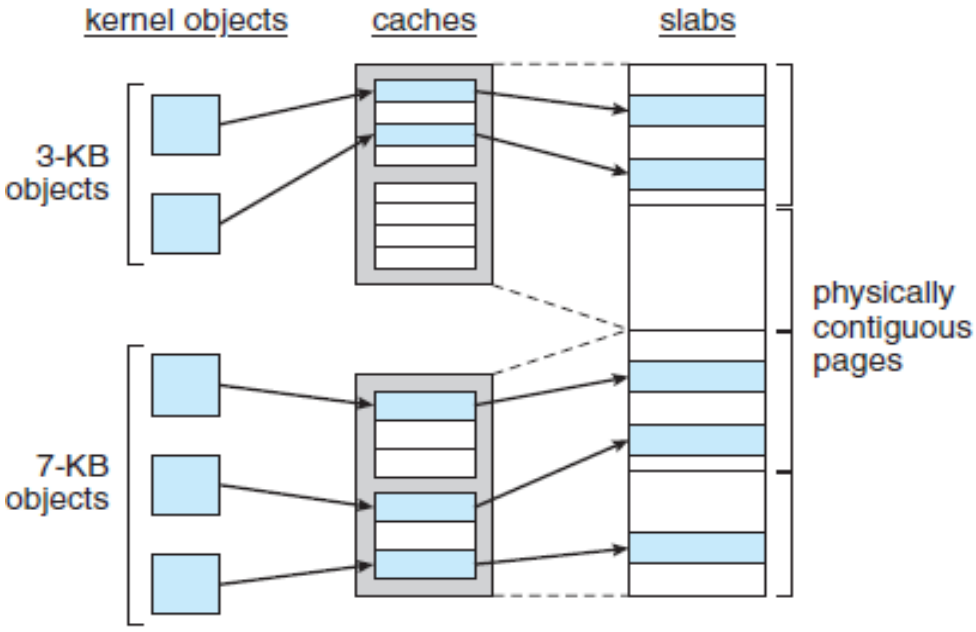
Slab Allocation

- A **slab** is made up of one or more physically contiguous pages.

- There is a single cache for each unique kernel data structure.

Example: A separate cache for the data structure representing process descriptors, a separate cache for file objects, a separate cache for semaphores.

A **cache** consists of one or more slabs.



- The slab-allocation algorithm uses caches to store kernel objects
- When a cache is created, a number of objects—which are initially marked as free—are allocated to the cache.
- Initially, all objects in the cache are marked as free. When a new object for a kernel data structure is needed, the allocator can assign any free object from the cache to satisfy the request. The object assigned from the cache is marked as used..
- In Linux, a slab may be in one of three possible states:
    1. **Full**. All objects in the slab are marked as used.
    2. **Empty**. All objects in the slab are marked as free.
    3. **Partial**. The slab consists of both used and free objects.
- The slab allocator first attempts to satisfy the request with a free object in a partial slab. If none exists, a free object is assigned from an empty slab. If no empty slabs are available, a new slab is allocated from contiguous physical pages and assigned to a cache.

Advantages: The slab allocator provides two main benefits:

i) No memory is wasted due to fragmentation.

ii) Memory requests can be satisfied quickly.

OS EXAMPLES:

CASE STUDY: How Windows and Solaris implement virtual memory.

1. Windows:

- Windows implements virtual memory using demand paging with **clustering**.
- **Clustering handles page faults by bringing in not only the faulting page but also several pages following the faulting page.**
- When a process is first created, it is assigned a working-set minimum and maximum.
- The **working-set minimum** is the minimum number of pages the process is guaranteed to have in memory.
- If sufficient memory is available, a process may be assigned as many pages as its **working-set maximum.**
- The virtual memory manager maintains a list of free page frames.
- Associated with this list is a threshold value that is used to indicate whether sufficient free memory is available.
- If a page fault occurs for a process that is below its working-set maximum, the virtual memory manager allocates a page from this list of free pages.

- If a process that is at its working-set maximum incurs a page fault, it must select a page for replacement using a local LRU page-replacement policy.
- When the amount of free memory falls below the threshold, the virtual memory manager uses a tactic known as **automatic working-set trimming** to restore the value above the threshold.
- If a process has been allocated more pages than its working-set minimum, the virtual memory Manager removes pages until the process reaches its working-set minimum.

2.Solaris:

- In Solaris, when a thread incurs a page fault, the kernel assigns a page to the faulting thread from the list of free pages it maintains.
- Associated with this list of free pages is a parameter—lotsfree—that represents a threshold to begin paging.
- The lotsfree parameter is typically set to 1⁄64 the size of the physical memory. Four times per second, the kernel checks whether the amount of free memory is less than lotsfree.
- If the number of free pages falls below lotsfree, a process known as a **pageout** starts up.
- The front hand of the clock scans all pages in memory, setting the reference bit to 0.
- Later, the back hand of the clock examines the reference bit for the pages in memory, appending each page whose reference bit is still set to 0 to the free list and writing to disk its contents if modified.
- Solaris maintains a cache list of pages that have been ―freed‖ but have not yet been overwritten.
- The free list contains frames that have invalid contents. Pages can be reclaimed from the cache list if they are accessed before being moved to the free list.