Dr. SNS RAJALAKSHMI COLLEGE OF ARTS & SCIENCE(Autonomous)

Coimbatore – 49.

DEPARTMENT OF COMPUTER APPLICATIONS

COURSE : OPERATING SYSTEM

CLASS : I BCA 'B'

Unit IV

MEMORY MANAGEMENT

3.1 Main Memory

3.1.1 Basic Hardware

- Program must be
  - → brought (from disk) into memory and
  - → placed within a process for it to be run.
- Main-memory and registers are only storage CPU can access directly.
- Register access in one CPU clock.
- Main-memory can take many cycles.
- Cache sits between main-memory and CPU registers.
- Protection of memory required to ensure correct operation.
- A pair of base- and limit-registers define the logical (virtual) address space (Figure 3.8 & 3.9).

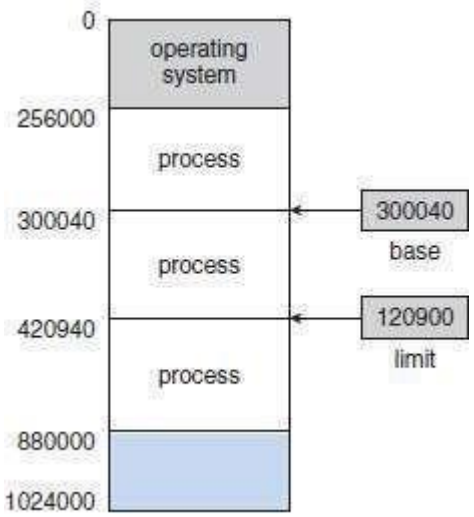


Figure 3.8 A base and a limit-register define a logical-address space

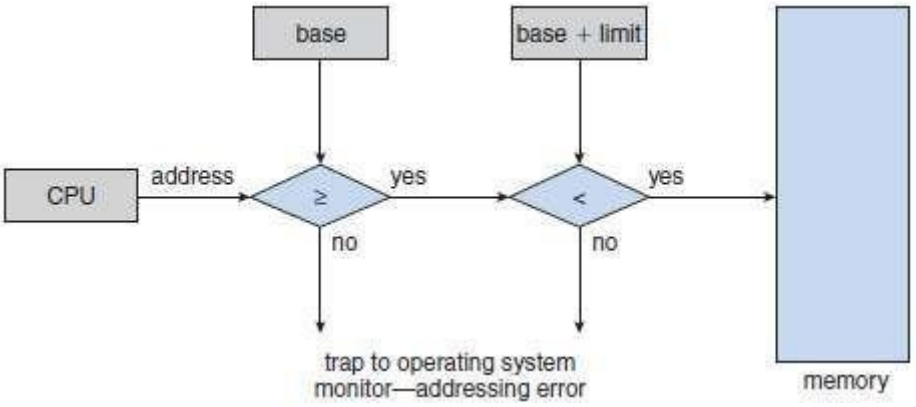


Figure 3.9 Hardware address protection with base and limit-registers

3.1.2 Address Binding

- Address binding of instructions to memory-addresses can happen at 3 different stages (Figure 3.10):

1) Compile Time

- If memory-location known a priori, absolute code can be generated.
- Must recompile code if starting location changes.

2) Load Time

- Must generate relocatable code if memory-location is not known at compile time.

3) Execution Time

- Binding delayed until run-time if the process can be moved during its execution from one memory-segment to another.
- Need hardware support for address maps (e.g. base and limit-registers).

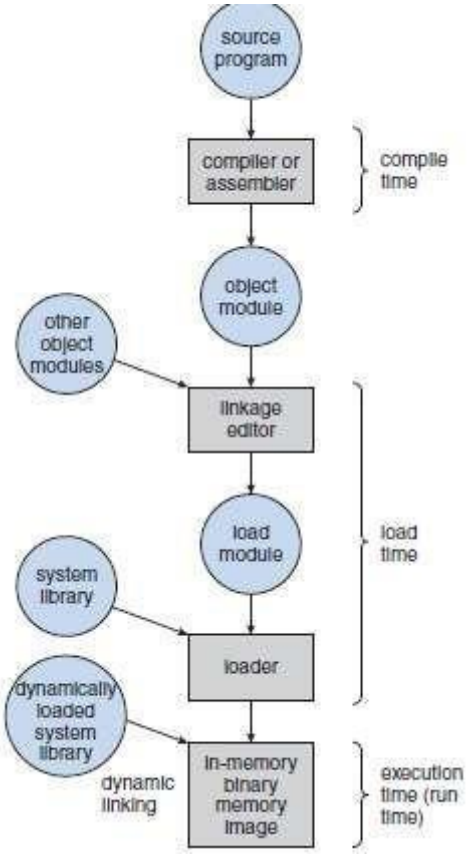


Figure 3.10 Multistep processing of a user-program

3.1.3 Logical versus Physical Address Space

- **Logical-address** is generated by the CPU (also referred to as virtual-address).

  Physical-address is the address seen by the memory-unit.

- Logical & physical-addresses are the same in compile-time & load-time address-binding methods. Logical and physical-addresses differ in execution-time address-binding method.
- MMU (Memory-Management Unit)
  - Hardware device that maps virtual-address to physical-address (Figure 3.11).
  - The value in the relocation-register is added to every address generated by a user-process at the time it is sent to memory.
  - The user-program deals with logical-addresses; it never sees the real physical-addresses.

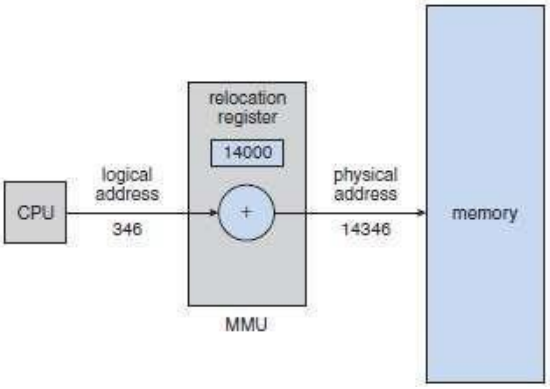


Figure 3.11 Dynamic relocation using a relocation-register

3.1.4 Dynamic Loading

- This can be used to obtain better memory-space utilization.
- A routine is not loaded until it is called.
- This works as follows:
  1) Initially, all routines are kept on disk in a relocatable-load format.
  2) Firstly, the main-program is loaded into memory and is executed.
  3) When a main-program calls the routine, the main-program first checks to see whether the routine has been loaded.
  4) If routine has been not yet loaded, the loader is called to load desired routine into memory.
  5) Finally, control is passed to the newly loaded-routine.
- Advantages:
  1) An unused routine is never loaded.
  2) Useful when large amounts of code are needed to handle infrequently occurring cases.

3) Although the total program-size may be large, the portion that is used (and hence loaded) may be much smaller.

4) Does not require special support from the OS.

3.1.5 Dynamic Linking and Shared Libraries

- Linking postponed until execution-time.
- This feature is usually used with system libraries, such as language subroutine libraries.
- A stub is included in the image for each library-routine reference.
- The **stub** is a small piece of code used to locate the appropriate memory-resident library-routine.
- When the stub is executed, it checks to see whether the needed routine is already in memory. If not, the program loads the routine into memory.
- Stub replaces itself with the address of the routine, and executes the routine.
- Thus, the next time that particular code-segment is reached, the library-routine is executed directly, incurring no cost for dynamic-linking.
- All processes that use a language library execute only one copy of the library code.

Shared libraries

- A library may be replaced by a new version, and all programs that reference the library will automatically use the new one.
- Version info. is included in both program & library so that programs won't accidentally execute incompatible versions.

3.2 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 into memory for continued execution.

• **Backing-store** is a fast disk which is large enough to accommodate copies of all memory-images for all users.

• **Roll out/Roll in** is a swapping variant used for priority-based scheduling algorithms.

- Lower-priority process is swapped out so that higher-priority process can be loaded and executed.
- Once the higher-priority process finishes, the lower-priority process can be swapped back in and continued (Figure 3.12).

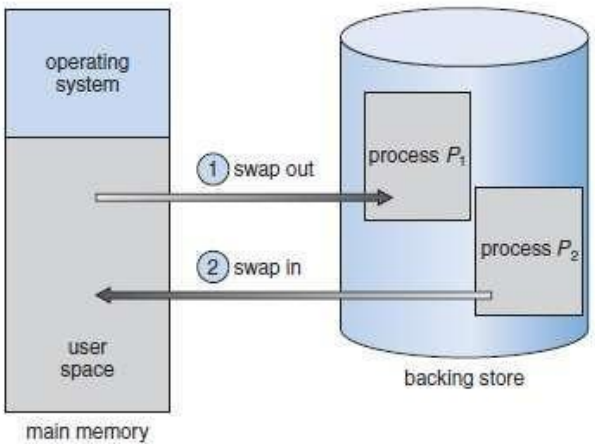


Figure 3.12 Swapping of two processes using a disk as a backing-store

• Swapping depends upon address-binding:

1) If binding is done at load-time, then process cannot be easily moved to a different location.

2) If binding is done at execution-time, then a process can be swapped into a different memory- space, because the physical-addresses are computed during execution-time.

• Major part of swap-time is transfer-time; i.e. total transfer-time is directly proportional to the amount of memory swapped.

• Disadvantages:

1) Context-switch time is fairly high.

2) If we want to swap a process, we must be sure that it is completely

idle. Two solutions:

i) Never swap a process with pending I/O.

ii) Execute I/O operations only into OS buffers.

3.3 Contiguous Memory Allocation

• Memory is usually divided into 2 partitions:

→ One for the resident OS.

→ One for the user-processes.

• Each process is contained in a single contiguous section of memory.

3.3.1 Memory Mapping & Protection

• Memory-protection means

→ protecting OS from user-process and

→ protecting user-processes from one another.

• Memory-protection is done using

→ **Relocation-register**: contains the value of the smallest physical-address.

→ **Limit-register**: contains the range of logical-addresses.

• 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 (Figure 3.13).

• When the CPU scheduler selects a process for execution, the dispatcher loads the relocation and limit-registers with the correct values.

• Because every address generated by the CPU is checked against these registers, we can protect the OS from the running-process.

• The relocation-register scheme provides an effective way to allow the OS size to change dynamically.

• **Transient OS code**: Code that comes & goes as needed to save memory-space and overhead for unnecessary swapping.

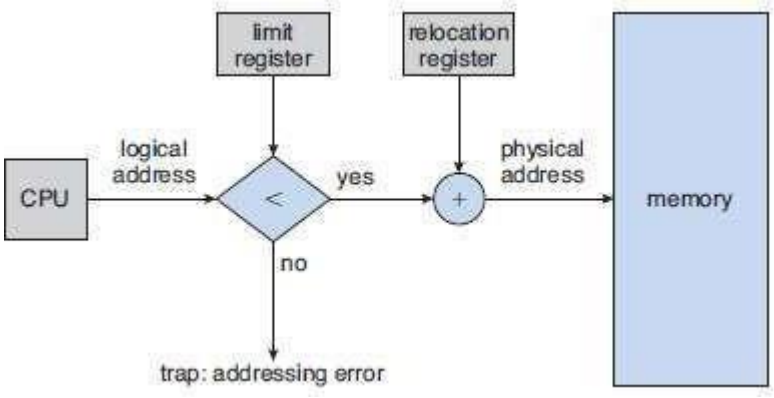


Figure 3.13 Hardware support for relocation and limit-registers

3.3.2 Memory Allocation

- Two types of memory partitioning are: 1) Fixed-sized partitioning and
  2) Variable-sized partitioning

1) Fixed-sized Partitioning

- The memory is divided into fixed-sized partitions.
- Each partition may contain exactly one process.
- The degree of multiprogramming is bound by the number of partitions.
- When a partition is free, a process is
  - → selected from the input queue and
  - → loaded into the free partition.
- When the process terminates, the partition becomes available for another process.

2) Variable-sized Partitioning

- The OS keeps a table indicating
  - → which parts of memory are available and
  - → which parts are occupied.
- A **hole** is a block of available memory.
- Normally, memory contains a set of holes of various sizes.
- Initially, all memory is
  - → available for user-processes and
  - → considered one large hole.
- When a process arrives, the process is allocated memory from a large hole.
- If we find the hole, we
  - → allocate only as much memory as is needed and
  - → keep the remaining memory available to satisfy future requests.

- Three strategies used to select a free hole from the set of available holes.

1) 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.

2) 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.

3) 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.

- First-fit and best fit are better than worst fit in terms of decreasing time and storage utilization.

3.3.3 Fragmentation

- Two types of memory fragmentation: 1) Internal fragmentation and
  2) External fragmentation

1) Internal Fragmentation

- The general approach is to
  → break the physical-memory into fixed-sized blocks and
  → allocate memory in units based on block size (Figure 3.14).
- The allocated-memory to a process may be slightly larger than the requested-memory.
- The difference between requested-memory and allocated-memory is called internal fragmentation i.e. Unused memory that is internal to a partition.

2) External Fragmentation

- External fragmentation occurs when there is enough total memory-space to satisfy a request but the available-spaces are not contiguous. (i.e. storage is fragmented into a large number of small holes).
- Both the first-fit and best-fit strategies for memory-allocation suffer from external fragmentation.
- Statistical analysis of first-fit reveals that
  → given N allocated blocks, another 0.5 N blocks will be lost to fragmentation. This property is known as the **50-percent rule**.
- Two solutions to external fragmentation (Figure 3.15):

  1) Compaction

  - The goal is to shuffle the memory-contents to place all free memory together in one large hole.
  - Compaction is possible only if relocation is
    → dynamic and
    → done at execution-time.

  2) Permit the logical-address space of the processes to be non-contiguous.

  - This allows a process to be allocated physical-memory wherever such memory is available.
  - Two techniques achieve this solution:
    1) Paging and
    2) Segmentation.

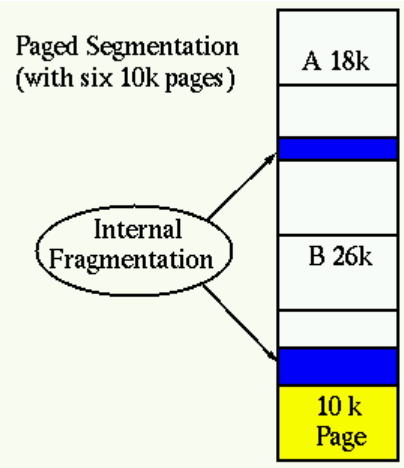


Figure 3.14: Internal fragmentation

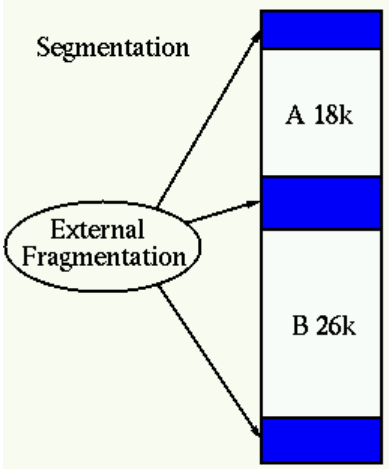


Figure 3.15: External fragmentation

3.13 Paging

- Paging is a memory-management scheme.
- This permits the physical-address space of a process to be non-contiguous.
- This also solves the considerable problem of fitting memory-chunks of varying sizes onto the backing-store.
- Traditionally: Support for paging has been handled by hardware.

  Recent designs: The hardware & OS are closely integrated.

3.13.1 Basic Method

- Physical-memory is broken into fixed-sized blocks called **frames**(Figure 3.16). Logical-memory is broken into same-sized blocks called **pages.**
- When a process is to be executed, its pages are loaded into any available memory-frames from the backing-store.
- The backing-store is divided into fixed-sized blocks that are of the same size as the memory-frames.

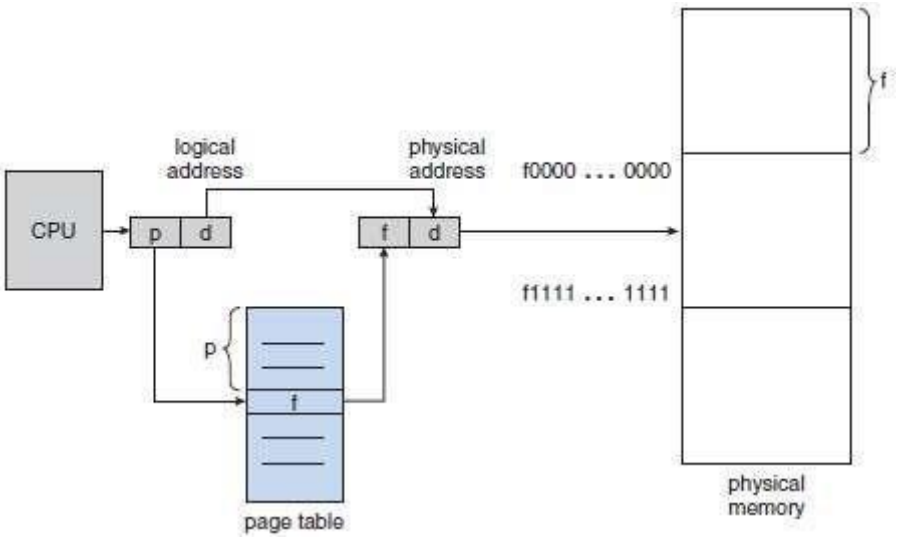


Figure 3.16 Paging hardware

- The page-table contains the base-address of each page in physical-memory.
- Address generated by CPU is divided into 2 parts (Figure 3.17):

  1) Page-number(p) is used as an index to the page-table and

  2) Offset(d) is combined with the base-address to define the physical-address. This physical-address is sent to the memory-unit.

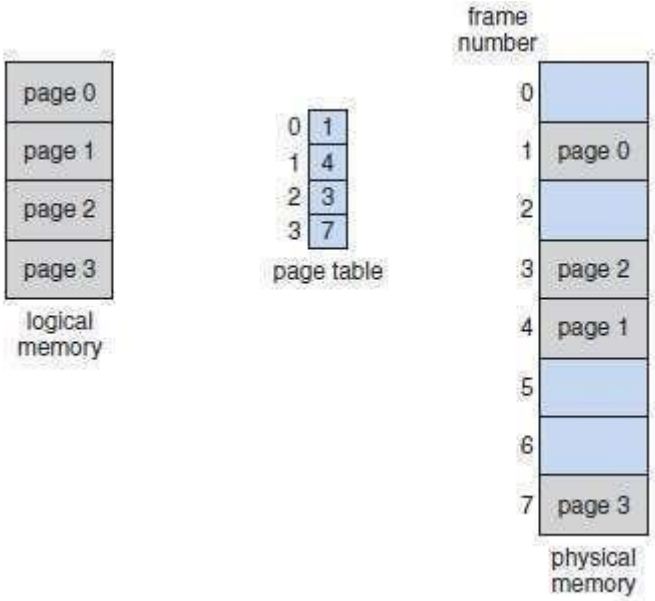


Figure 3.17 Paging model of logical and physical-memory

• The page-size (like the frame size) is defined by the hardware (Figure 3.18).

• 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.

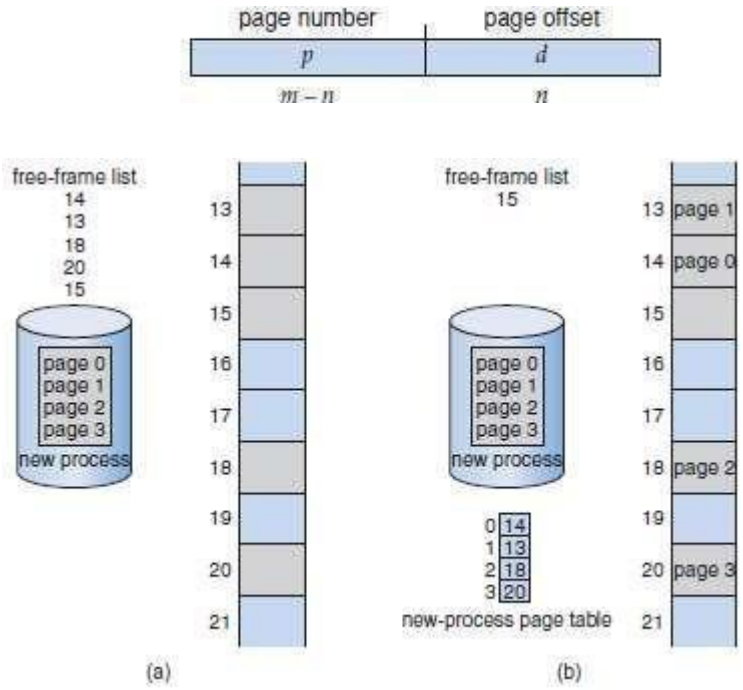


Figure 3.18 Free frames (a) before allocation and (b) after allocation

3.13.2 Hardware Support for Paging

- Most OS's store a page-table for each process.
- A pointer to the page-table is stored in the PCB.

Translation Lookaside Buffer

- The TLB is associative, high-speed memory.
- The TLB contains only a few of the page-table entries.
- Working:
  - 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
    → used to access memory.
  - If page-number is not in TLB (**TLB miss**), a memory-reference to page table must be made.
  - The obtained frame-number can be used to access memory (Figure 3.19).
  - 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 OS must select one for replacement.
- Percentage of times that a particular page-number is found in the TLB is called **hit ratio**.
- Advantage: Search operation is fast.
  Disadvantage: Hardware is expensive.
- Some TLBs have wired down entries that can't be removed.
- Some TLBs store ASID (address-space identifier) in each entry of the TLB that uniquely
  → identify each process and
  → provide address space protection for that process.

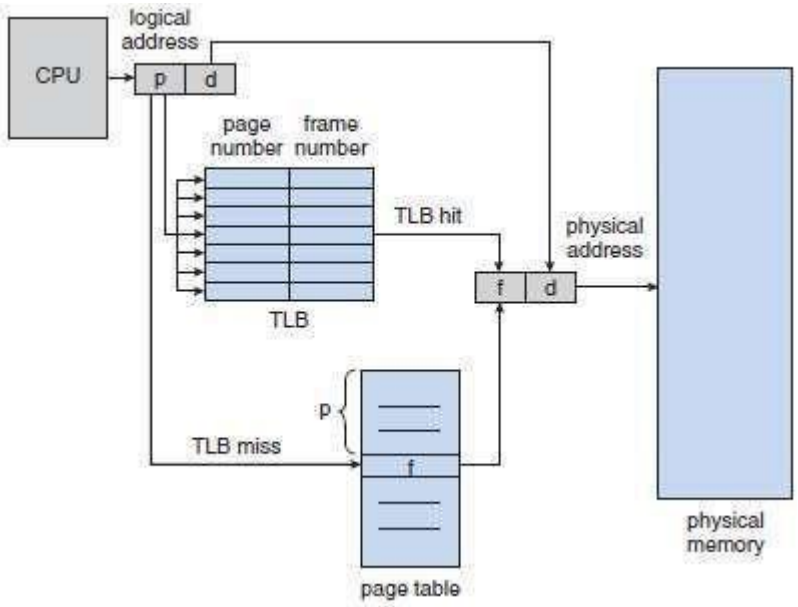


Figure 3.19 Paging hardware with TLB

3.13.3 Protection

- Memory-protection is achieved by **protection-bits** for each frame.
- The protection-bits are kept in the page-table.
- One protection-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.
- Firstly, the physical-address is computed. At the same time, the protection-bit is checked to verify that no writes are being made to a read-only page.
- An attempt to write to a read-only page causes a hardware-trap to the OS (or memory-protection violation).

Valid Invalid Bit

- This bit is attached to each entry in the page-table (Figure 3.20).
    1) **Valid bit**: The page is in the process' logical-address space.
    2) **Invalid bit**: The page is not in the process' logical-address space.
- Illegal addresses are trapped by use of valid-invalid bit.
- The OS sets this bit for each page to allow or disallow access to the page.

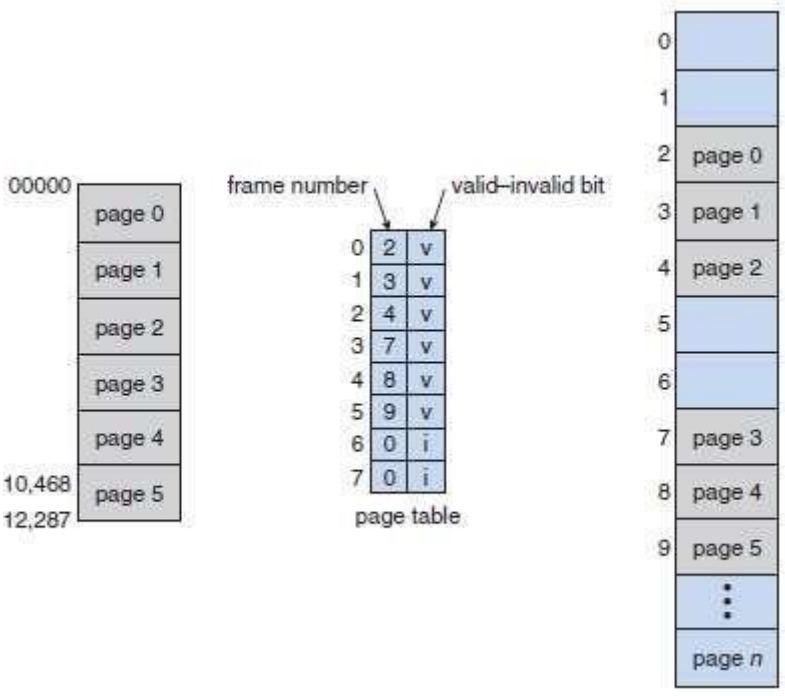


Figure 3.20 Valid (v) or invalid (i) bit in a page-table

3.13.4 Shared Pages

- Advantage of paging:

  1) Possible to share common code.

- Re-entrant code is non-self-modifying code, it never changes during execution.
- Two or more processes can execute the same code at the same time.
- Each process has its own copy of registers and data-storage to hold the data for the process's execution.
- The data for 2 different processes will be different.
- Only one copy of the editor need be kept in physical-memory (Figure 3.21).
- Each user's page-table maps onto the same physical copy of the editor, but data pages are mapped onto different frames.
- Disadvantage:

  1) Systems that use inverted page-tables have difficulty implementing shared-memory.

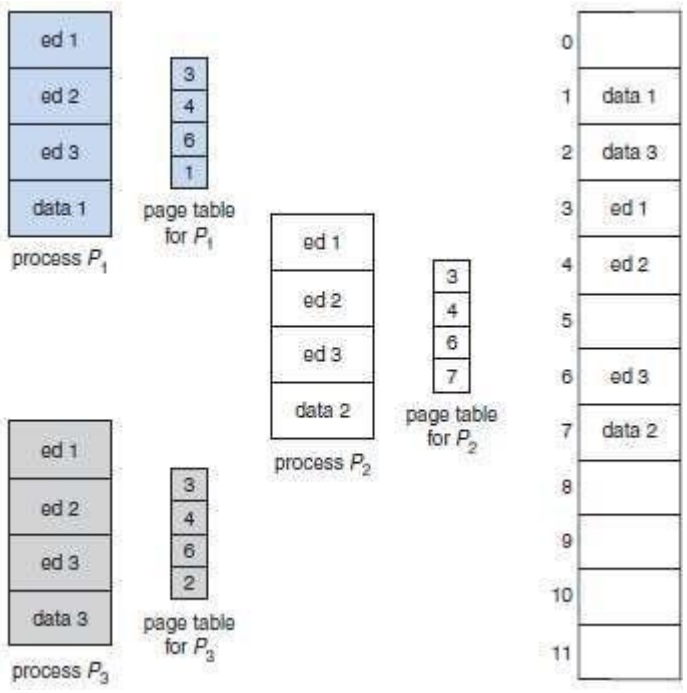


Figure 3.21 Sharing of code in a paging environment

3.14 Structure of the Page Table

1) Hierarchical Paging
2) Hashed Page-tables
3) Inverted Page-tables

3.14.1 Hierarchical Paging

• Problem: Most computers support a large logical-address space (2^{32} to 2^{64}). In these systems, the page-table itself becomes excessively large.

Solution: Divide the page-table into smaller pieces.

Two Level Paging Algorithm

• The page-table itself is also paged (Figure 3.22).

• This is also known as a forward-mapped page-table because address translation works from the outer page-table inwards.

• For example (Figure 3.23):

- Consider the system with a 32-bit logical-address space and a page-size of 4 KB.
- A logical-address is divided into
  - → 20-bit page-number and
  - → 12-bit page-offset.
- Since the page-table is paged, the page-number is further divided into
  - → 10-bit page-number and
  - → 10-bit page-offset.
- Thus, a logical-address is as follows:

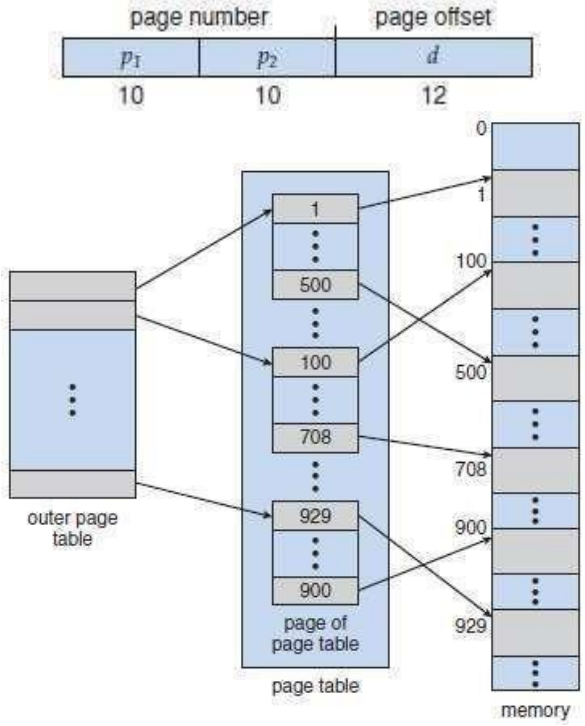


Figure 3.22 A two-level page-table scheme

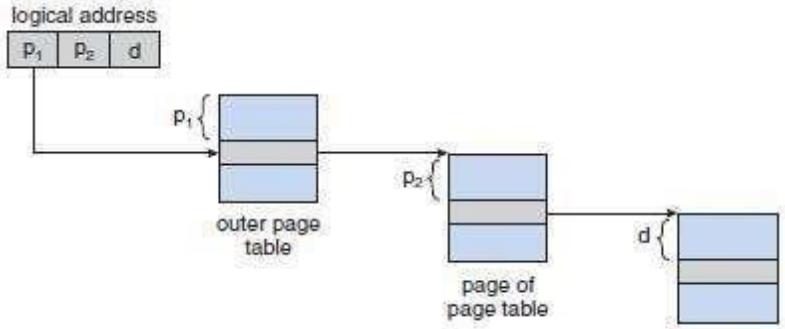


Figure 3.23 Address translation for a two-level 32-bit paging architecture

3.14.2 Hashed Page Tables

- This approach is used for handling address spaces larger than 32 bits.
- The hash-value is the virtual page-number.
- Each entry in the hash-table contains a linked-list of elements that hash to the same location (to handle collisions).
- Each element consists of 3 fields:
    1) Virtual page-number
    2) Value of the mapped page-frame and
    3) Pointer to the next element in the linked-list.
- The algorithm works as follows (Figure 3.24):
    1) The virtual page-number is hashed into the hash-table.
    2) The virtual page-number is compared with the first element in the linked-list.
    3) If there is a match, the corresponding page-frame (field 2) is used to form the desired physical-address.
    4) If there is no match, subsequent entries in the linked-list are searched for a matching virtual page-number.

Clustered Page Tables

- These are similar to hashed page-tables except that each entry in the hash-table refers to several pages rather than a single page.
- Advantages:
    1) Favorable for 64-bit address spaces.
    2) Useful for address spaces, where memory-references are noncontiguous and scattered throughout the address space.

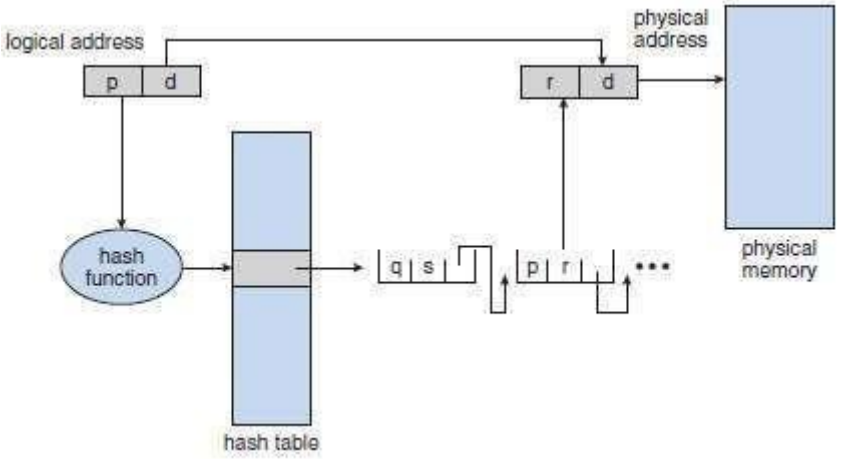


Figure 3.24 Hashed page-table

3.14.3 Inverted Page Tables

- Has one entry for each real page of memory.
- Each entry consists of

  → virtual-address of the page stored in that real memory-location and

  → information about the process that owns the page.

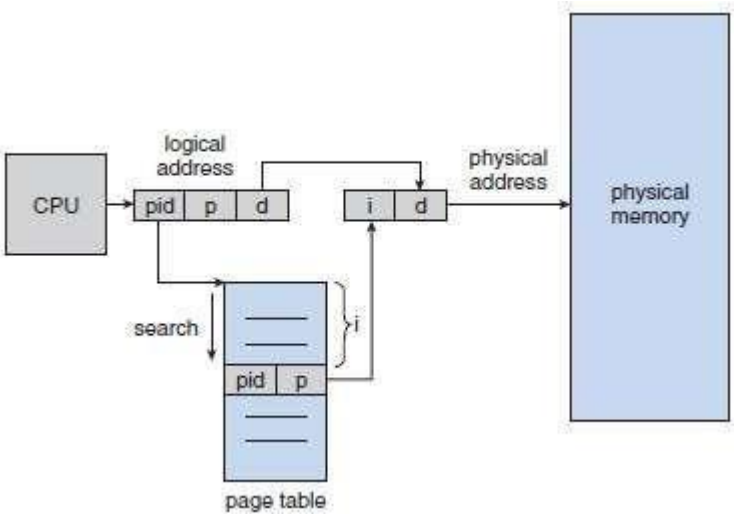


Figure 3.25 Inverted page-table

- Each virtual-address consists of a triplet (Figure 3.25):

  <process-id, page-number, offset>.
- Each inverted page-table entry is a pair <process-id, page-number>
- The algorithm works as follows:

  1) When a memory-reference occurs, part of the virtual-address, consisting of <process-id, page-number>, is presented to the memory subsystem.

  2) The inverted page-table is then searched for a match.

  3) If a match is found, at entry i-then the physical-address <i, offset> is generated.

  4) If no match is found, then an illegal address access has been attempted.
- Advantage:

  1) Decreases memory needed to store each page-table
- Disadvantages:

  1) Increases amount of time needed to search table when a page reference occurs.

  2) Difficulty implementing shared-memory.

3.15 Segmentation

3.15.1 Basic Method

- This is a memory-management scheme that supports user-view of memory(Figure 3.26).
- A logical-address space is a collection of segments.
- Each segment has a name and a length.
- The addresses specify both
  - → segment-name and
  - → offset within the segment.
- Normally, the user-program is compiled, and the compiler automatically constructs segments reflecting the input program.

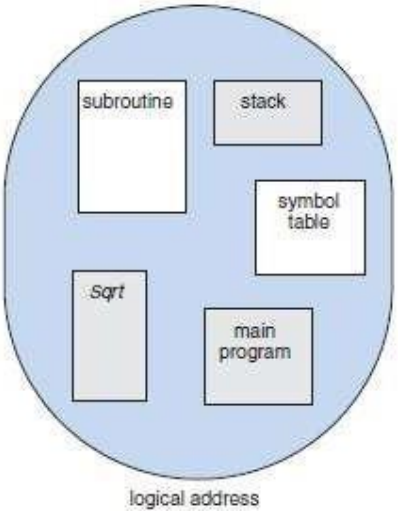
For ex:

→ The code

→ Global variables

→ The heap, from which memory is allocated

→ The stacks used by each thread



→ The standard C library

Figure 3.26 Programmer's view of a program

3.15.2 Hardware Support

- Segment-table maps 2 dimensional user-defined addresses into one-dimensional physical-addresses.
- In the segment-table, each entry has following 2 fields:
  1) **Segment-base** contains starting physical-address where the segment resides in memory.
  2) **Segment-limit** specifies the length of the segment (Figure 3.27).
- A logical-address consists of 2 parts:

1) **Segment-number(s)** is used as an index to the segment-table .

2) **Offset(d)** must be between 0 and the segment-limit.

• If offset is not between 0 & segment-limit, then we trap to the OS(logical-addressing attempt beyond end of segment).

• If offset is legal, then it is added to the segment-base to produce the physical-memory address.

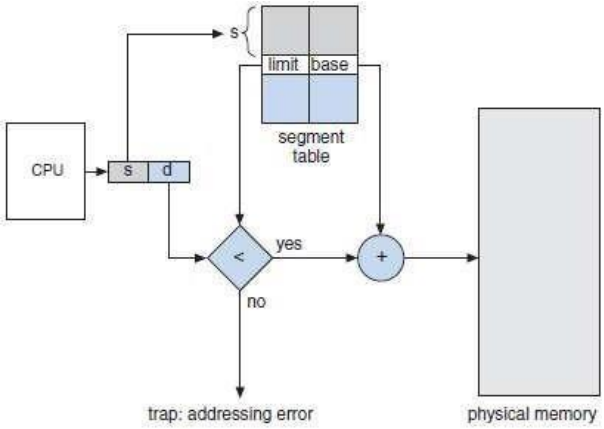


Figure 3.27 Segmentation hardware