Memory Management

Compiled By, Er. Nabaraj Bahadur Negi

Contents

- Introduction, Monoprogramming vs. Multi-programming, Modelling Multiprogramming, Multiprogramming with fixed and variable partitions, Relocation and Protection.
- Memory management (Bitmaps & Linked-list), Memory Allocation Strategies
- Virtual memory: Paging, Page Table, Page Table Structure, Handling Page Faults, TLB's
- Page Replacement Algorithms: FIFO, Second Chance, LRU, Optimal, LFU, Clock, WS-Clock, Concept of Locality of Reference, Belady's Anomaly
- Segmentation: Need of Segmentation, its Drawbacks, Segmentation with Paging (MULTICS)

Memory Management

- Memory management is the basic concept of OS, which manages the **main memory.**
- Main memory (RAM) is a critical and limited resource in any computer system.
- Allocation and de-allocation of the process take place between main memory and secondary memory. Memory management keeps the record of each process in the main memory.
- Both programmers and users ideally want memory that is infinitely large, extremely fast, and non-volatile (i.e., retains data after power is off), but in reality, all three can't be achieved at once.
- The Operating System (OS) is responsible for managing the computer's memory hierarchy—this task is handled by the memory manager (**or memory management unit, MMU**).

Key Responsibilities of the Memory Manager:

1. Tracking Memory Usage

- The memory manager keeps track of which parts of memory are in use (allocated to processes or system) and which are free.
- It maintains data structures (such as bitmaps, linked lists, or tables) to monitor memory allocation.

2. Allocating Memory

1. When a process needs memory (e.g., when a program runs or requests more space), the memory manager **allocates** available memory to that process.
2. Allocation must be efficient and prevent conflicts (no two processes should get the same memory).

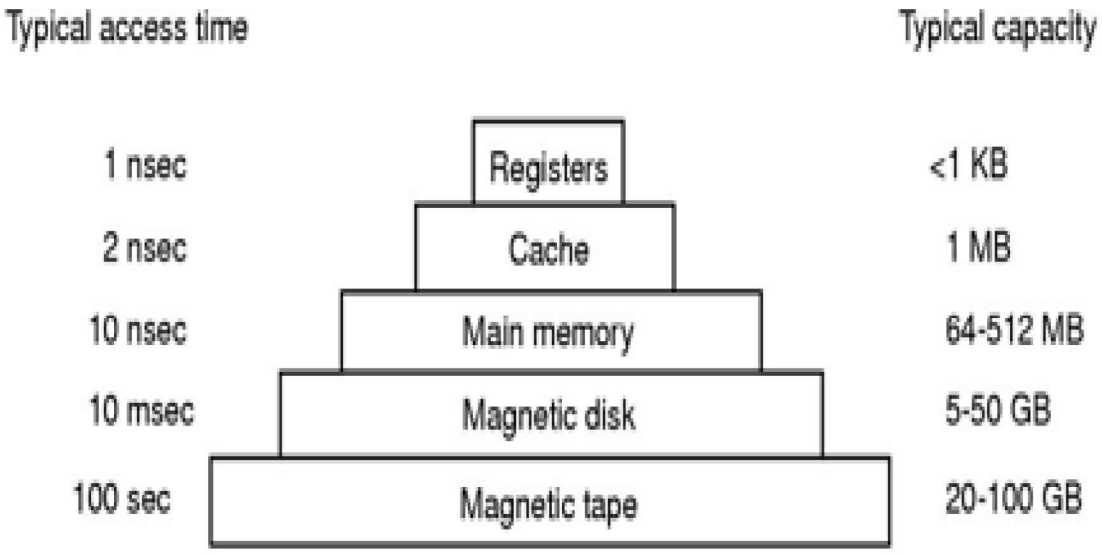
3. Deallocating Memory

- When a process finishes or releases memory, the OS deallocates (frees up) that memory so it can be used by other processes.
- Proper deallocation prevents memory leaks (wasted memory that is no longer used but not released).

4. Swapping and Virtual Memory Management

- If main memory (RAM) is too small to hold all active processes, the OS uses swapping:
  - Moves some processes/data to disk storage (slower, but larger and non-volatile).
  - Brings them back into RAM as needed.
- This is part of the virtual memory concept, which gives the illusion of a larger main memory by using disk space as an extension.

Computer Hardware Review



- Typical memory hierarchy, numbers shown are rough approximations

- Memory management systems can be divided into two basic classes:
  - Those that move processes back and forth between main memory and disk during execution (swapping and paging) and
  - Those that don’t.

Monoprogramming without Swapping or Paging

- The simplest possible memory management scheme is to run just one program at a time, sharing the memory between that program and the OS.
- Three simple ways of organizing memory

-An operating system with one user process,

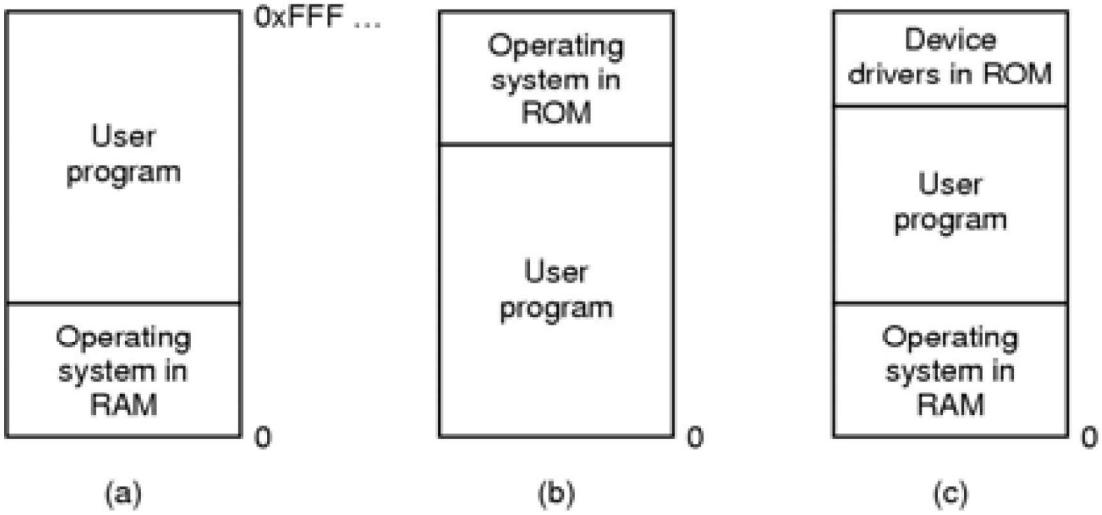


Figure 4-1: Three simple ways of organizing memory with an operating system and one user process. Other possibilities also exist.

- The OS may be at the bottom of memory in RAM (a). Or it may be in ROM at the top of memory (b) or the device drivers may be at the top of memory in a ROM and the rest of the system in RAM down below (c).
- The first model was formerly used on mainframes and minicomputers but is rarely used any more.
- The second model is used on some palmtop computers and embedded systems.
- The third model was used by early personal computers (e.g., running MS-DOS), where the portion of the system in the ROM is called the BIOS.
- When the system is organized in this way, **only one process at a time** can be running.

- As soon as the user types a command, the OS copies the requested program from disk to memory and executes it.
- When the process finishes, the OS displays a prompt character and waits for a new command.
- When it receives the command, it loads a new program into memory, overwriting the first one.

Multiprogramming

- Today, almost all computer systems allow multiple processes to run at the same time.
- When multiple processes running at the same time, means that when one process is blocked waiting for input/output to finish, another one can use the Central Processing Unit.
- Therefore, multiprogramming increases the central processing unit utilization.
- Now-a-day, both network server and client machines have the ability to run multiple processes at the same time.
- To achieve multiprogramming, the simplest way is just to divide memory up into n partitions (normally unequal partitions).
- Multiprogramming lets the processes use the Central Processing Unit when it would be. The Central Processing Unit (CPU) utilization can be improved when multiprogramming is used.

Logical and Physical Address

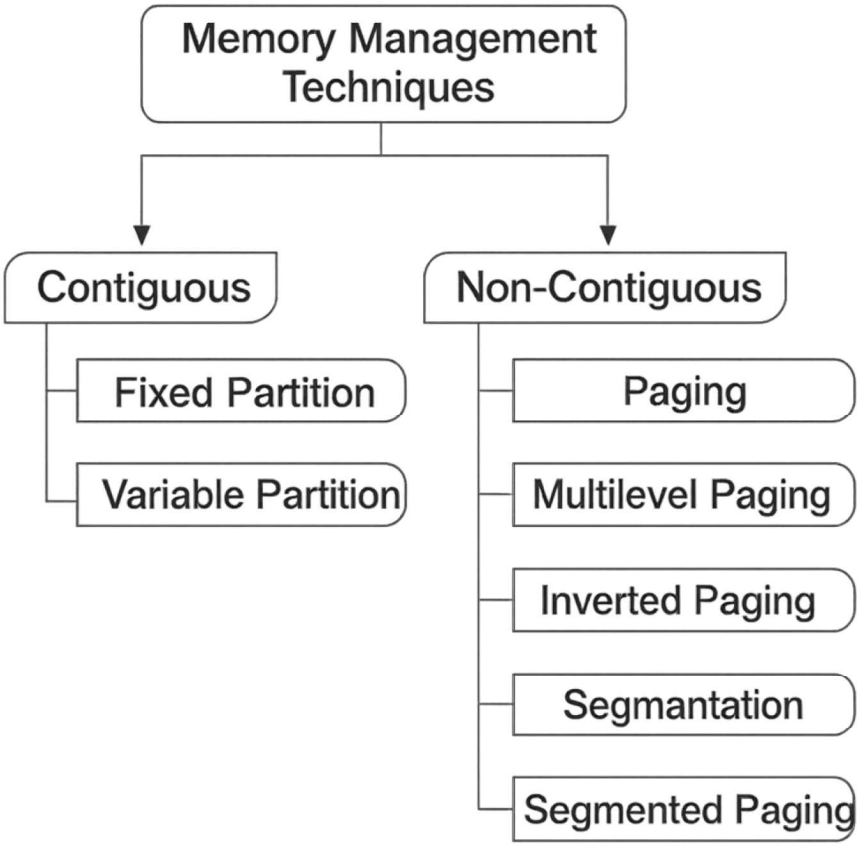
Logical Address

- Address generated by **CPU** while a program is running is referred as **Logical Address**.
- The logical address is virtual as it does not exist physically. Hence, it is also called as **Virtual Address**. This address is used as a reference to access the physical memory location.
- The set of all logical addresses generated by a program's perspective is called **Logical Address Space**.
- The logical address is mapped to its corresponding physical address by a hardware device called **Memory-Management Unit**. The address-binding methods used by MMU generates identical logical and physical address during compile time and load time.

Physical Address

- **Physical Address** identifies a physical location in a memory.
- MMU (**Memory-Management Unit)** computes the physical address for the corresponding logical address. MMU also uses logical address computing physical address. The user never deals with the physical address.
- The logical address is mapped to the physical address using a hardware called Memory-Management Unit.
- The set of all physical addresses corresponding to the logical addresses in a Logical address space is called **Physical Address Space**.

Memory Management Techniques



Feature	Contiguous Memory Allocation	Noncontiguous Memory Allocation
Memory Allocation	Allocates single continuous block of memory for each process	Allocates memory in multiple separate blocks (not necessarily continuous)
Address Translation	Not required (simple calculation)	Required (needs mapping and translation)
Fragmentation	Suffers from external fragmentation	Only paging suffers from internal fragmentation; avoids external fragmentation
Process Placement	Process must be loaded into one large, available block	Process can be loaded into any available blocks
Memory Utilization	Can lead to memory wastage	Utilizes memory more efficiently
Flexibility	Less flexible; hard to accommodate large processes	More flexible; easier to fit processes into available memory
Swapping	Swapped-in processes go to original area	Swapped-in processes can go anywhere in memory
Techniques Used	Fixed partitioning, variable partitioning	Paging, segmentation, multi-level paging, etc.
Overhead	Low overhead	Higher overhead due to address translation
Complexity	Simple to implement	More complex to implement

Memory Allocation

Multiprogramming with Fixed Partitions

- Multiprogramming refers to running multiple processes concurrently in a single computer system.
- Instead of executing one process at a time (monoprogramming), multiprogramming allows a CPU to switch between processes when a currently running process is waiting (for instance, for I/O operations to finish), thereby improving the overall efficiency and CPU utilization.

Fixed Partitions:

- In fixed partition multiprogramming, the memory is divided into a fixed number of partitions.
- These partitions are defined statically, often at system initialization, and can be equal or unequal in size.
- Each partition holds exactly one process at a time.

- Memory is divided into fixed-size regions (partitions) when the system boots up.
- Each partition can hold one process.
- When a process enters the system, it is allocated to an available partition.
- When no partitions are available, a new process must wait in a queue.
- CPU switches between processes based on their availability and state (blocked or running).
- **Example,** Assume the memory is divided into 4 partitions: 100KB, 150KB, 200KB, and 300KB.
- A process requiring 180KB arrives. It can only fit into the partitions 200KB or 300KB.
- The operating system chooses one of these partitions, and the process occupies it until it terminates or is swapped out.

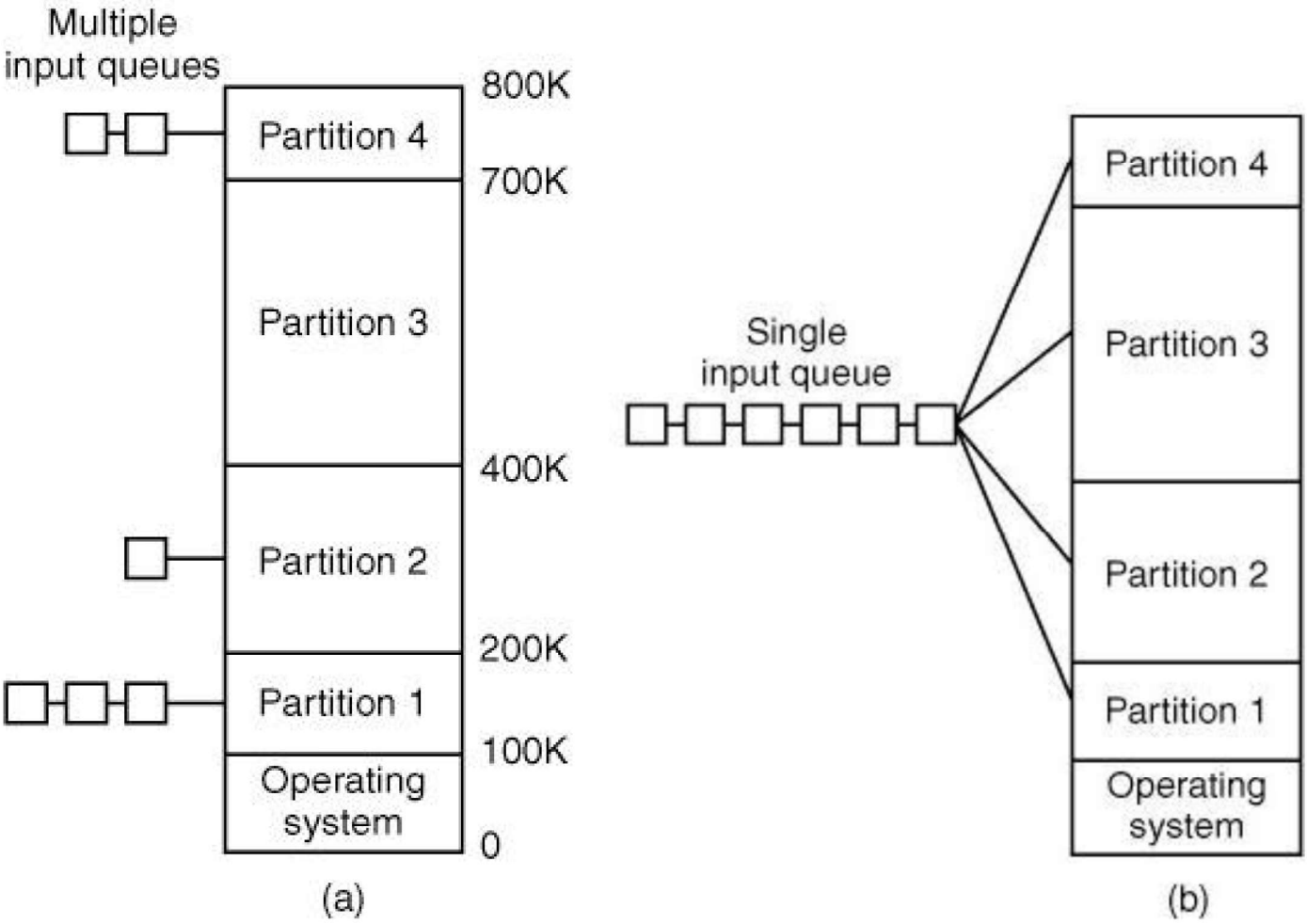


Figure 4-2. (a) Fixed memory partitions with separate input queues for each partition.(b) Fixed memory partitions with a single input queue.

Separate Queues for Each Partition (Figure a)

- Memory partitions are fixed in size (here: partition 1, 2, 3, and 4 with sizes indicated).
- Each partition has its own dedicated queue.
- Jobs are sorted into these queues based on their size and the matching partition's size

Disadvantages:

- A major drawback occurs when one partition's queue is empty, leaving that partition idle, while other partitions have waiting jobs.
- For example, in Figure (a):
    - Partition 4 (largest) queue is empty- no jobs are waiting, causing wasted resources as it remains idle.
    - Partition 1 (smallest) queue has many waiting jobs, causing delays.

Single Shared Input Queue (Figure b)

- All incoming jobs wait in a single queue, irrespective of their sizes.
- When a partition becomes free, the scheduler checks this single queue from the front.
- The first job in line that fits the partition is selected and executed.

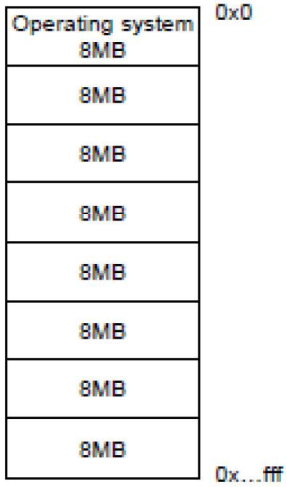
Advantages:

- This approach significantly improves resource utilization.
- It eliminates the problem of having empty queues and idle partitions when jobs are waiting elsewhere.

Disadvantages:

- Risk of allocating small jobs into larger partitions, resulting in wasted space (internal fragmentation).

- Two fixed partitioning schemes,
  - Equal-size partitions
  - Unequal-size partitions



Equal-size partitions

- Memory is divided into multiple partitions of the **same size**.

Problems:

- **Big programs**: Cannot run if their size exceeds the fixed partition size.
- **Possible solution:** Split large programs into smaller parts and load each part separately from disk (overlay technique).
- **Small programs**: Occupy an entire partition, even though the program may be smaller than the partition.
- Leads to wasted memory space within partitions, called **"internal fragmentation"**

Unequal-size partitions

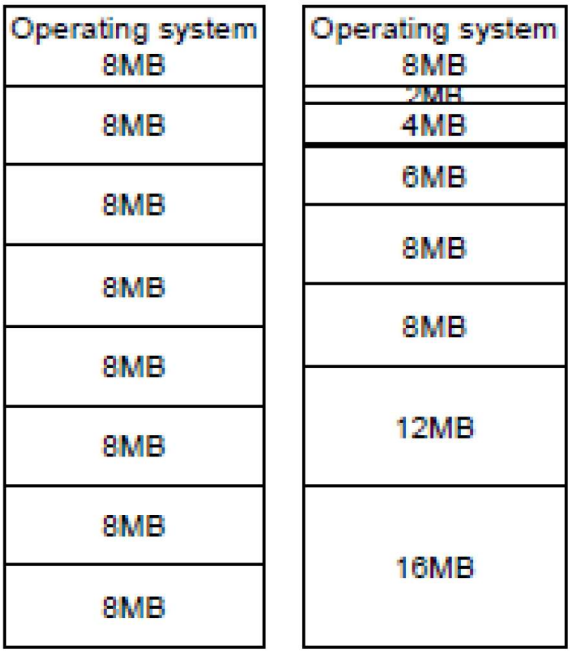
- Memory partitions are of different sizes.
- This approach addresses some limitations of equal-size partitioning by accommodating programs of varying sizes more efficiently.

Key Advantages:

- **Handling Larger Programs:** Large programs can easily fit into appropriately large partitions.
- Reduces the need to load large programs in parts from disk, simplifying execution.
- **Reducing Internal Fragmentation:** Small programs can fit into smaller partitions, significantly reducing wasted space.
- Results in more efficient utilization of memory space overall compared to equal-size partitions.

Potential drawbacks:

- Complexity in managing multiple partition sizes.
- Still possible internal fragmentation, although generally less severe.



Fragmentation

- Fragmentation occurs in memory management when the available memory becomes inefficiently divided, causing issues when allocating space to new processes.
- Fragmentation can be classified into two types:
  - **Internal Fragmentation**
  - **External Fragmentation**
- Internal fragmentation occurs when the allocated memory block is slightly larger than the requested memory.
- The excess memory inside the allocated block remains unused and is wasted, as it's too small to be allocated to another process.

- In systems with fixed-sized memory allocation units, such as:
  - **Fixed partitions** (equal or unequal sized partitions).
  - **Paging** (a modern memory management method where memory is divided into fixed-size frames)

Example :

Process 1: Requires 10KB, allocated block of 12KB → 2KB wasted

Process 2: Requires 10KB, allocated block exactly 10KB → 0KB wasted

Process 3: Requires 12KB, allocated block of 13KB → 1KB wasted

Total internal fragmentation = 3KB (2KB + 1KB).

- External fragmentation occurs when free memory is divided into small, scattered blocks that collectively have enough space for new processes, but individually, each block is too small for allocation.

- Occurs in systems with variable-sized memory allocation units, such as:
  - **Dynamic partitioning** (multiple partitions with varying sizes allocated dynamically).
  - **Segmentation** (memory divided into logical variable-sized segments).
- Algorithms such as **First-fit** and **Best-fit** also lead to external fragmentation, since allocation and freeing of memory creates scattered, unusable memory fragments.

Example: Initially, memory is free and contiguous.

- Process A (20KB) allocated, Process B (10KB) allocated, Process C (15KB) allocated. Now, Processes A and C finish execution and leave holes of 20KB and 15KB separately.
- Total free memory = 35KB, but it's split into two blocks (20KB + 15KB).
- If a new process (Process D) needs 30KB contiguous memory, it cannot be allocated, despite having 35KB free space.

Relocation and Protection in Memory Management

- When an operating system uses multiprogramming (running multiple programs at the same time), two important challenges emerge:
  - **Relocation**
  - **Protection**
- **Relocation** is the process of adjusting the memory addresses within a program based on where the program is loaded into memory.
- Programs can be loaded into different areas (partitions) of memory at different times, so their addresses must be adjusted accordingly.

Why is Relocation Needed?

- When you compile and link a program, it assumes it starts at address 0 (or some fixed address).
- However, when multiple programs run, they can't all start at the same address because they'll overwrite each other. Each program needs to be loaded into a different area of memory.

Example : Suppose the program has an instruction: CALL 100 (This means the program calls a function located at memory address 100).

- Suppose the OS loads your program into memory starting at address 100,000 (100K).
- If your program directly tries to jump to address 100, it actually tries to access address 100, which isn't correct. The actual address should be 100,000 + 100 = 100,100.
- Similarly, if your program is loaded at address 200,000 (200K), the same instruction should now jump to 200,000 + 100 = 200,100.

How do computers solve the relocation problem?

To solve this, CPUs use two special hardware registers:

- **Base Register**: Holds the actual starting address where your program is loaded in memory.
- **Limit Register**: Holds the size (length) of memory allocated to your program.

Program Loaded at	Base Register	Limit Register (size of memory)
100K	100K	50K (program can use 100K–150K)
200K	200K	50K (program can use 200K–250K)

Now, when your program executes the instruction: CALL 100

The CPU automatically calculates the actual memory address as:

Effective Address = Base Register + Instruction Address

- If Base = 100K, CALL 100 → Effective address = 100K + 100
- If Base = 200K, CALL 100 → Effective address = 200K + 100

The program's instructions **do not change**. The CPU hardware adjusts addresses dynamically.

- A pair of **base** and **limit registers** define the logical address space

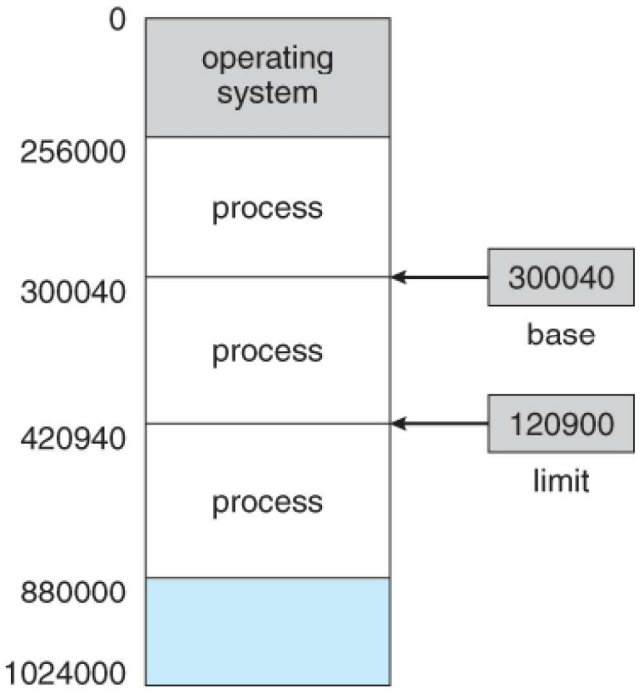


Figure 4-3 : A base and a limit register define a logical address space

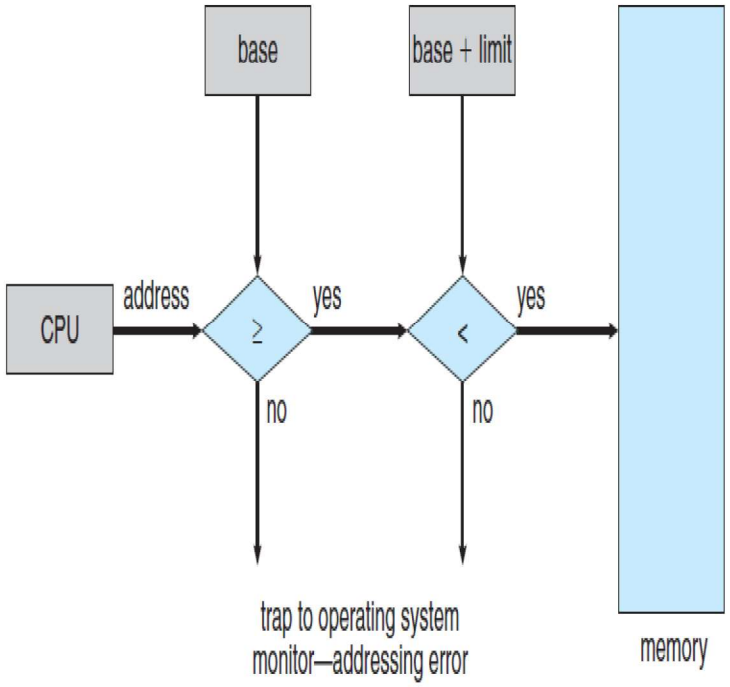


Figure 4-4 : Hardware Address Protection

Disadvantages of Using Base and Limit Registers:

- **Slower Speed**: Every memory reference requires:
  - **One addition** (base + logical address)
  - **One comparison** (to ensure within limits)
- **Reason Addition is Slower**: Additions can be slow because they require handling **carry propagation** (carrying numbers from one digit to another in binary arithmetic), unless specialized hardware is used.
- **Less Common Today**: Due to this overhead, modern computers rarely use the simple base-and-limit mechanism. Modern architectures use more advanced methods (like paging) for memory protection and management.

Memory Protection

- **Memory protection** ensures that a process (program in execution) only accesses the memory allocated to it and does not interfere with other processes.
- To achieve this, the operating system uses some special bits (protection bits) associated with each memory frame (physical memory block).

Protection Bits:

- Each memory frame typically has **protection bits** to indicate how the memory can be accessed by a process:
  - **Read-Only Bit:** The frame can be read, but not modified.
  - **Read-Write Bit:** The frame can be both read and written.
  - **Execute-Only Bit:** (optional) The frame contains executable code but cannot be modified or read as data directly.

These protection bits make sure that processes can't accidentally or intentionally modify sensitive data or code.

Valid-Invalid Bit (V/I bit):

- Another important bit is the **valid-invalid bit** attached to each entry in the **page table** (the data structure used to translate logical addresses to physical addresses):
- **Valid (V):** The page is part of the process's logical address space, meaning the process can legally access this page.
- **Invalid (I):** The page is not part of the process's logical address space, so any attempt to access it will be trapped (stopped and handled as an error by the OS).

Simplified Real-Life Analogy:

Think of memory protection as hotel-room keycards:

- Your keycard (**page table entry**) has permissions:
  - **Valid bit:** It opens your assigned room (legal access).
  - **Invalid bit:** It doesn't open other rooms (illegal access).
- **Protection bit:** Specifies what you can do:
  - **Read-only:** You can enter and look around but not change anything.
  - **Read-Write:** You can enter, rearrange furniture, etc.

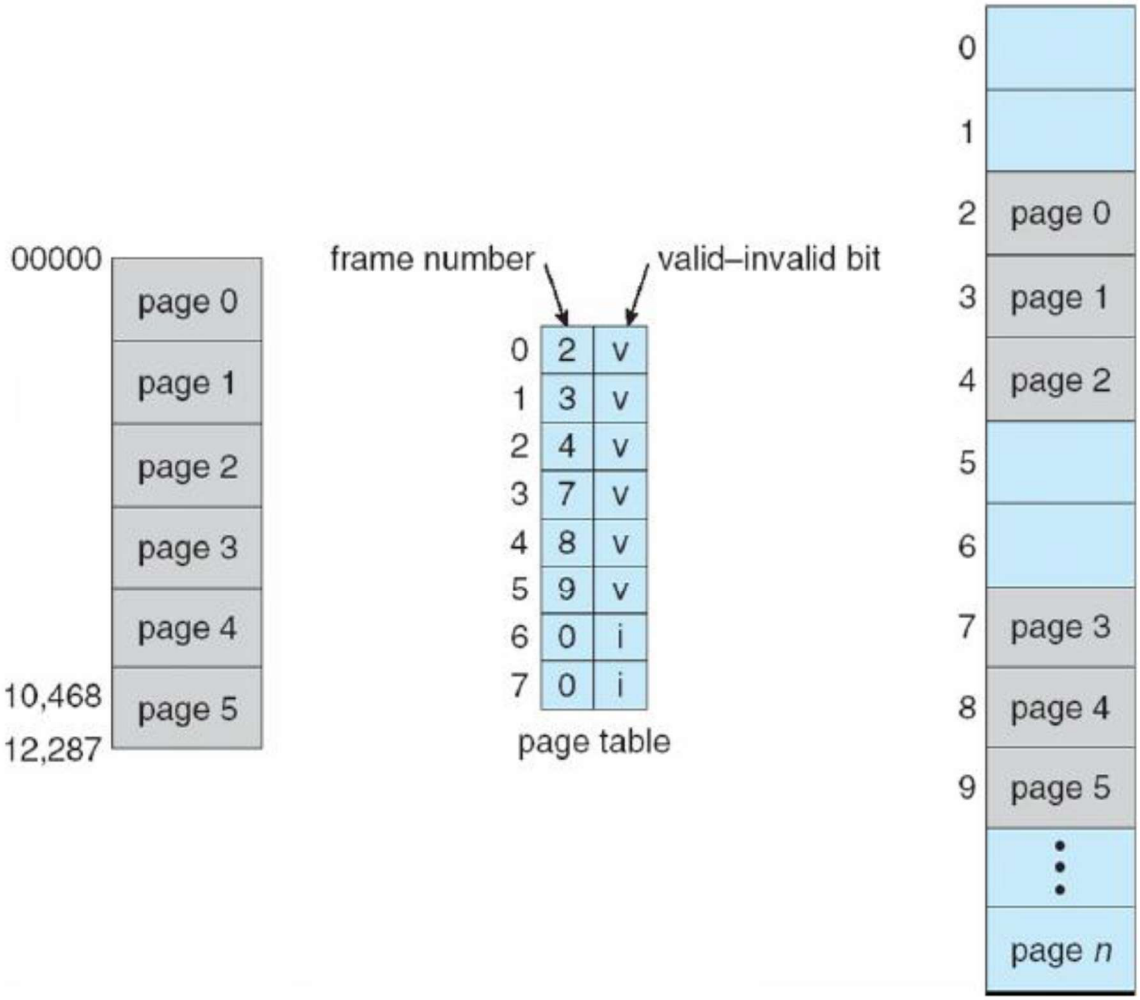


Figure 4-5 : Valid (v) or Invalid (i) Bit In A Page Table

Shared Pages

- **Shared code**

– One copy of read-only (**reentrant**) code shared among processes (i.e., text editors, compilers, window systems)

– Similar to multiple threads sharing the same process space

– Also useful for inter-process communication if sharing of read-write pages is allowed

- **Private code and data**

– Each process keeps a separate copy of the code and data

– The pages for the private code and data can appear anywhere in the logical address space

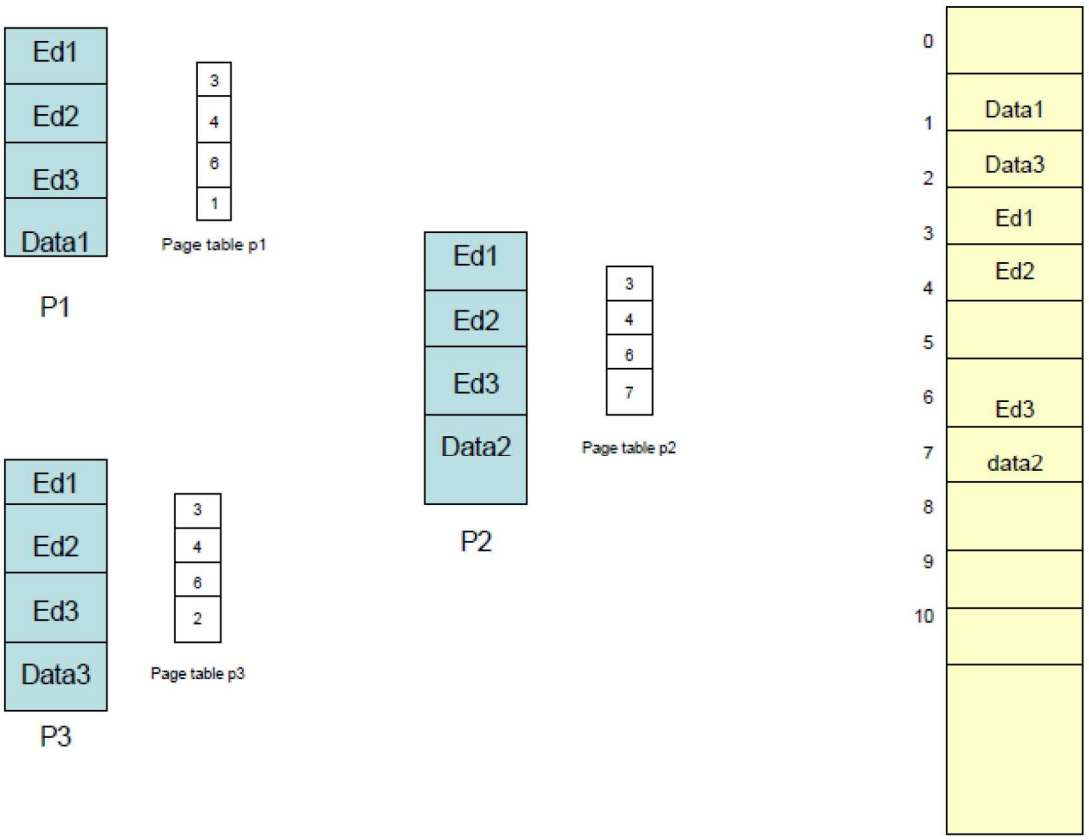


Figure 4-6 : Shared Pages Example

- Two general approaches to memory management can be used, depending on the available hardware:
  - –**Swapping** (the simplest strategy that consists of bringing in each process in its entirety, running it for a while, then putting it back on the disk) and
  - –**Virtual memory** (which allows programs to run even when they are only partially in main memory).
- **Swapping** is a memory management scheme in which any process can be temporarily swapped from main memory to secondary memory so that the main memory can be made available for other processes.
- It is used to improve main memory utilization.
- At some later time, the system swaps back the process from the secondary storage to main memory.

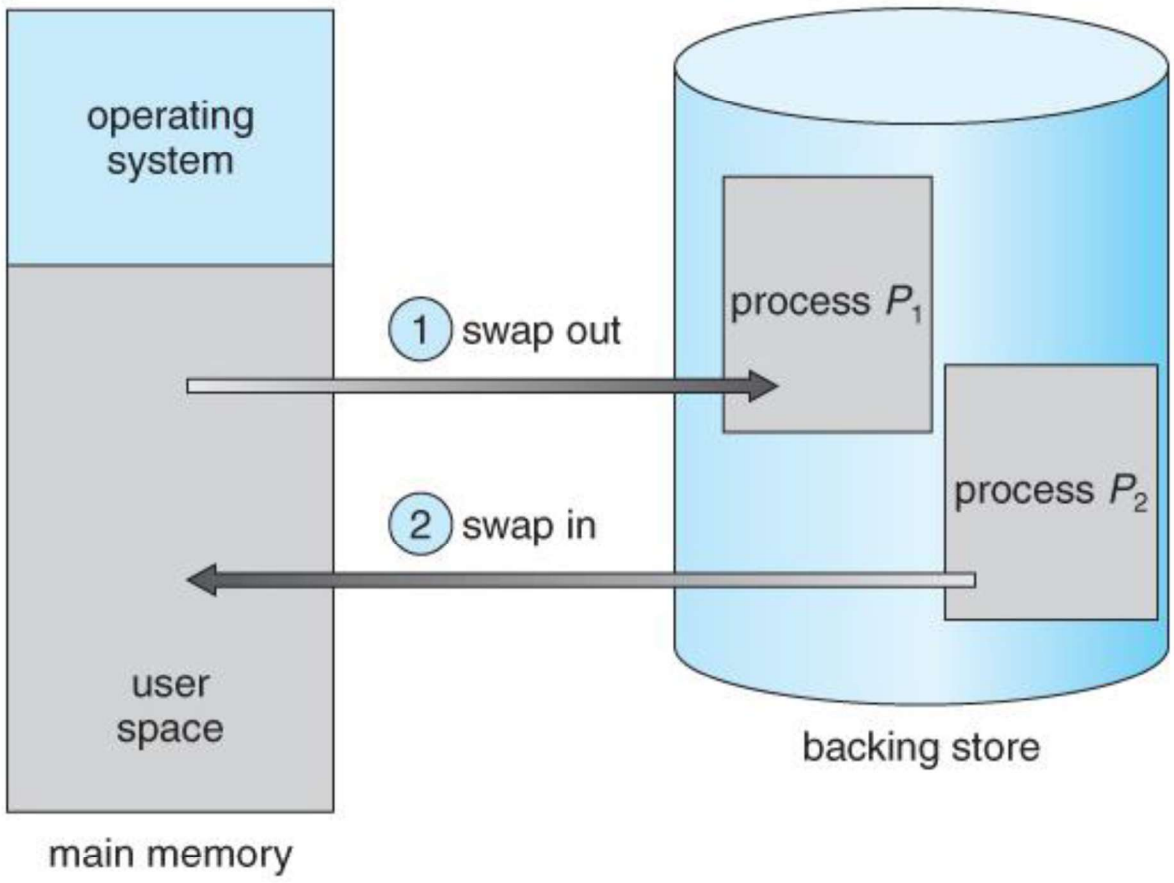


Figure 4-7 : Swapping

- The operation of a swapping system is shown below:

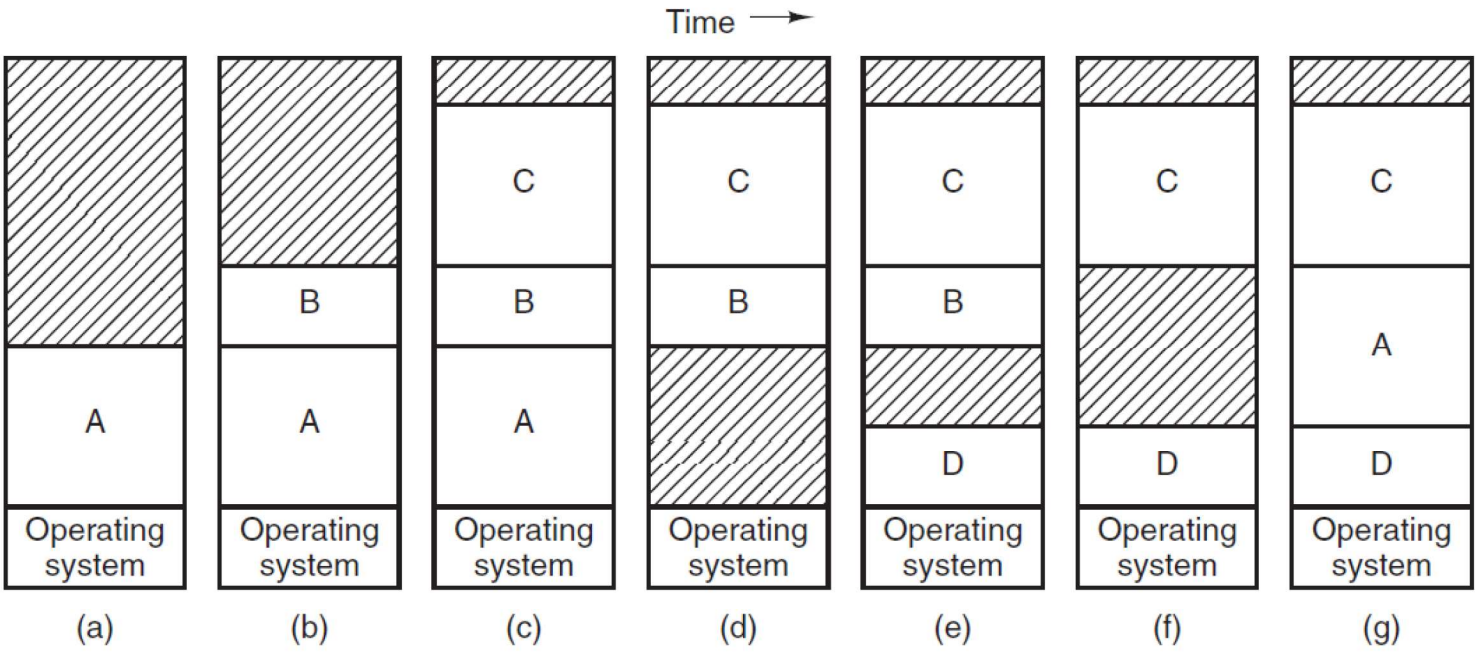


Figure 4.8 : Memory allocation changes as processes come into memory and leave it. The shaded regions are unused memory.

- Initially, only process *A* is in memory. Then processes *B* and *C* are created or swapped in from disk.
- In Fig.4-8(d) *A* is swapped out to disk. Then *D* comes in and *B* goes out. Finally *A* comes in again.
- Since *A* is now at a different location, addresses contained in it must be relocated, either by software when it is swapped in or (more likely) by hardware during program execution.
- When swapping creates multiple holes in memory, it is possible to combine them all into one big one by moving all the processes downward as far as possible. This technique is known as **memory compaction**. It is usually not done because it requires a lot of CPU time.
- when swapping processes to disk, only the memory actually in use should be swapped; it is wasteful to swap the extra memory as well.

- In Fig 4-9 (a) we see a memory configuration in which space for growth has been allocated to two processes.

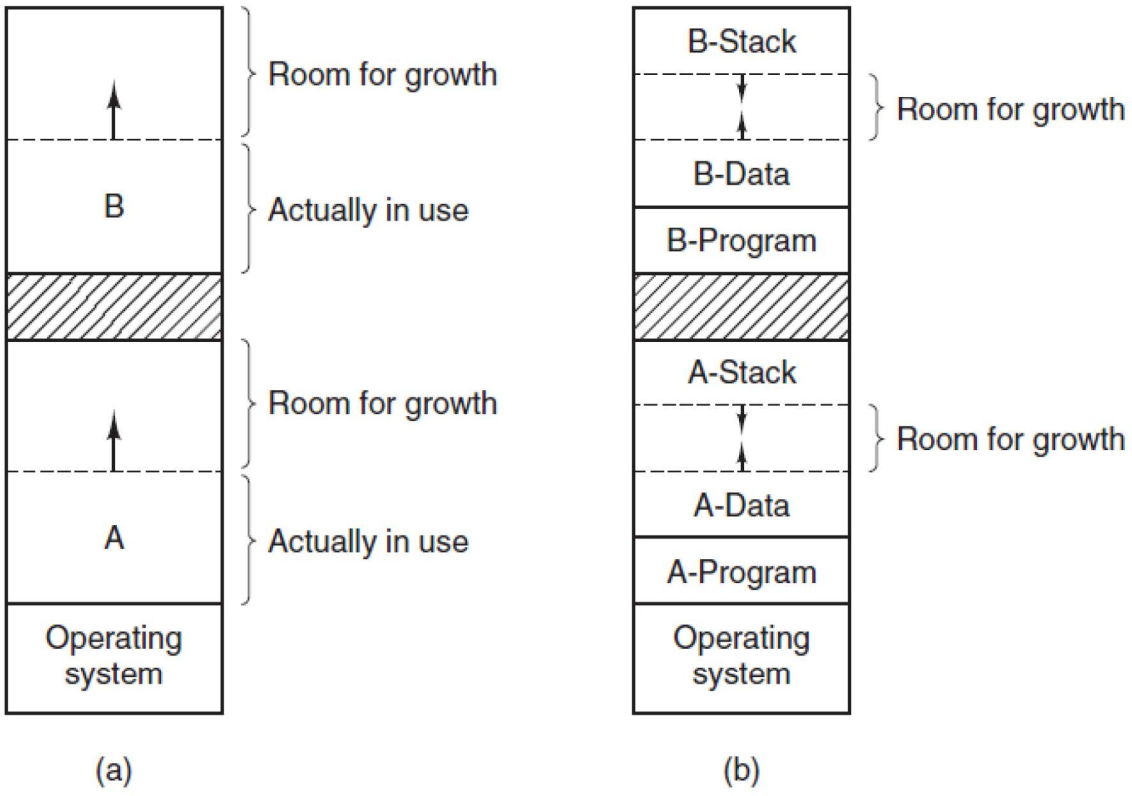


Figure 4-9 : (a) Allocating space for a growing data segment. (b) Allocating space for a growing stack and a growing data segment.

- If processes can have two growing segments—for example, the data segment being used as a heap for variables that are dynamically allocated and released and a stack segment for the normal local variables and return addresses
- In (b) we see that each process illustrated has a stack at the top of its allocated memory that is growing downward, and a data segment just beyond the program text that is growing upward.
- The memory between them can be used for either segment. If it runs out, the process will either have to be moved to a hole with sufficient space, swapped out of memory until a large enough hole can be created, or killed.

Managing Free Memory

Memory management with Bitmaps

- When memory is assigned dynamically, the operating system must manage it.
- In general terms, there are two ways to keep track of memory usage: bitmaps and free lists.
- With a bitmap, memory is divided up into allocation units, perhaps as small as a few words and perhaps as large as several kilobytes.
- Corresponding to each allocation unit is a bit in the bitmap, which is 0 if the unit is free and 1 if it is occupied (or vice versa).
- The next figure shows part of memory and the corresponding bitmap.

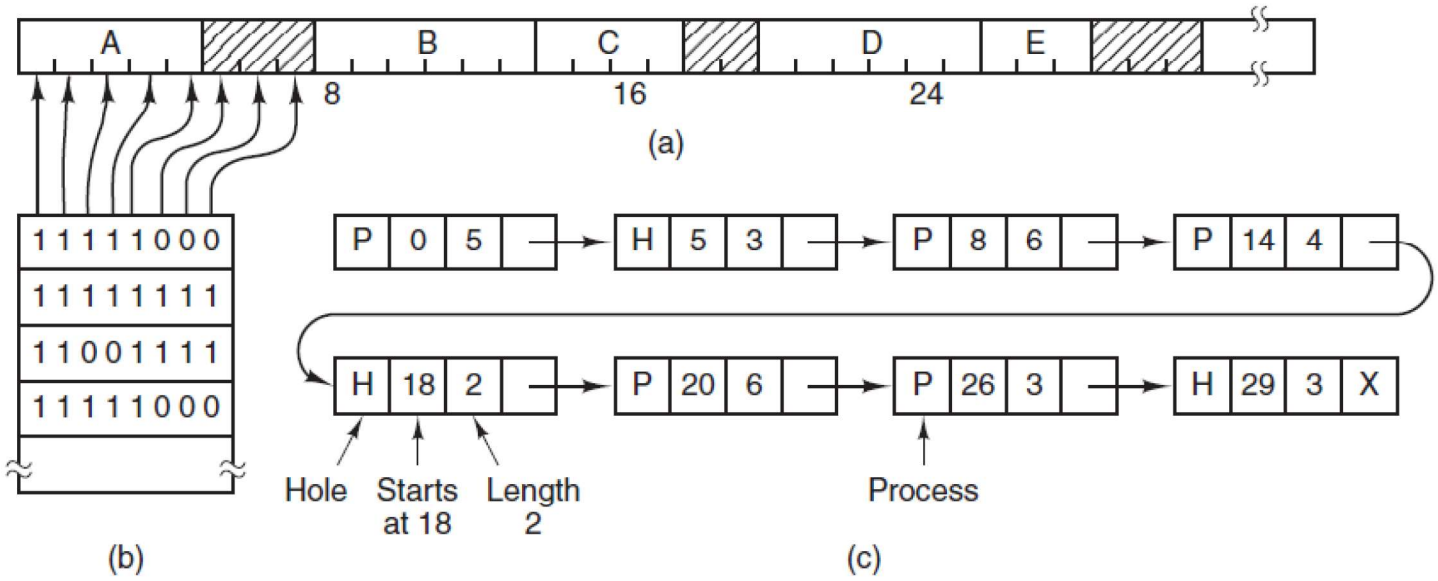


Figure 4-10 : (a) A part of memory with five processes and three holes. The tick marks show the memory allocation units. The shaded regions (0 in the bitmap) are free. (b) The corresponding bitmap. (c) The same information as a list.

- The size of the allocation unit is an important design issue.
- The smaller the allocation unit, the larger the bitmap.
- A bitmap provides a simple way to keep track of memory words in a fixed amount of memory because the size of the bitmap depends only on the size of memory and the size of the allocation unit.
- The main problem with it is that when it has been decided to bring a k unit process into memory, the mem manager must search the bitmap to find a run of k consecutive 0 bits in the map
- searching a bitmap for a run of a given length is a slow operation

Memory Management with Linked Lists

- Another way of keeping track of memory is to maintain a linked list of allocated and free memory segments, where a segment is either a process or a hole between two processes.
- When processes and holes are kept on a list that is sorted by address, several algorithms can be used to allocate memory for a newly created process or an existing process being swapped in from the disk.
- There are different types of algorithms like, first fit, best fit, next fit, worst fit, quick fit.
- The memory of Fig. 4-11(a) is represented in Fig. 4-11(c) as a linked list of segments. Each entry in the list specifies a hole (H) or process (P), the address at which it starts, the length, and a pointer to the next item.

Figure 4-11 : Four neighbor combinations for the terminating process, X.

- A terminating process normally has two neighbors (except when it is at the very top or very bottom of memory).
- These may be either processes or holes, leading to the four combinations shown below.
    - In (a) updating the list requires replacing a P by an H.
    - In (b) and also in (c), two entries are coalesced into one, and the list becomes one entry shorted.
    - In (d), three entries are merged and two items are removed from the list.
- The process of merging two adjacent holes to form a single larger hole is called **coalescing** .
- It is possible to combine all the holes into one big one by moving all the processes downward as far as possible; this technique is called memory **compaction**.
- When the processes and holes are kept on a list sorted by address, several algorithms can be used to allocate memory for a created process (or an existing process being swapped in from disk).

Memory Allocation Strategies

First fit

- The process manager scans along the list of segments until it finds a hole that is big enough.
- The hole is then broken up into two pieces, one for the process and one for the unused memory, except in the statistically unlikely case of an exact fit.
- First fit is a fast algorithm because it searches as little as possible.

Next fit

- It works the same way as first fit, except that it keeps track of where it is whenever it finds a suitable hole.
- The next time it is called to find a hole, it starts searching the list from the place where it left off last time, instead of always beginning, as first fit does.
- Simulations by bays (1977) show that next fit gives slightly worse performance than first fit.

Best fit

- Best fit searches the entire list and takes the smallest hole that is adequate.
- Rather than breaking up a big hole that might be needed later, best fit tries to find a hole that is close to the actual size needed.
- Best fit is slower that first fit because it must search the entire list every time it is called.
- Somewhat surprisingly, it also results in more wasted memory than first fit or next fit because it tends to fill up memory with tiny, useless holes (first fit creates larger holes on average).

worst fit

- To get around the problem of breaking up nearly exact matches into a process and a tiny hole, one could think about worst fit, that is, always take the largest available hole, so that the hole broken off will be big enough to be useful.
- Simulation has shown that worst fit is not a very good idea either.

Quick fit

- Quick fit maintains separate lists for some of the more common sizes requested.
- For example, it might have a table with n entries, in which the first entry is a pointer to the head of a list of 4-kb holes, the second entry is a pointer to a list of 8-kb holes, the third entry a pointer to 12-kb holes, and so on.
- Holes of say, 21-kb, could either be put on the 20-kb list or on a special list of odd-sized holes.
- With quick fit, finding a hole of the required size is extremely fast, but it has the same disadvantage as all the other scheme that sort by hole size.

Example:

Given five memory partitions of 100KB, 500KB, 200KB, 300KB, 600KB (in order), how would the first-fit, best-fit, and worst-fit algorithms place processes of 212 KB, 417 KB, 112 KB, and 426 KB (in order)? Which algorithm makes the most efficient use of memory?

Solution:

First-fit:

212K is put in 500K partition

417K is put in 600K partition

112K is put in 288K partition (new partition 288K = 500K - 212K)

426K must wait

Best-fit:

212K is put in 300K partition

417K is put in 500K partition

112K is put in 200K partition

426K is put in 600K partition

Worst-fit:

212K is put in 600K partition

417K is put in 500K partition

112K is put in 388K partition

426K must wait

In this example, best-fit turns out to be the best.

Virtual Memory

- Many years ago, people were first confronted with programs that were too big to fit in the available memory.
- A solution adopted in the 1960s was to split programs into little pieces, called **overlays**.
- Overlay 0 would start running first, When it was done, it would call another overlay.
- Some overlay systems were highly complex, allowing many overlays in memory at once. The overlays were kept on the disk and swapped in and out of memory by the overlay manager.
- Although the actual work of swapping overlays in and out was done by the operating system, the work of splitting the program into pieces had to be done manually by the programmer.

- Splitting large programs up into small, modular pieces was time consuming, boring, and error prone.
- The method that was devised has come to be known as **virtual memory**. The basic idea behind virtual memory is that each program has its own address space, which is broken up into chunks called **pages**.
- Virtual memory, or virtual storage is a memory management technique that provides an "idealized abstraction of the storage resources that are actually available on a given machine" which "creates the illusion to users of a very large memory".
- These pages are mapped onto physical memory, but not all pages have to be in physical memory at the same time to run the program.

Paging

- **Paging** is a storage mechanism that allows OS to retrieve processes from the secondary storage into the main memory in the form of pages.

- Memory Management Unit generates physical address from virtual/logical address provided by the program.

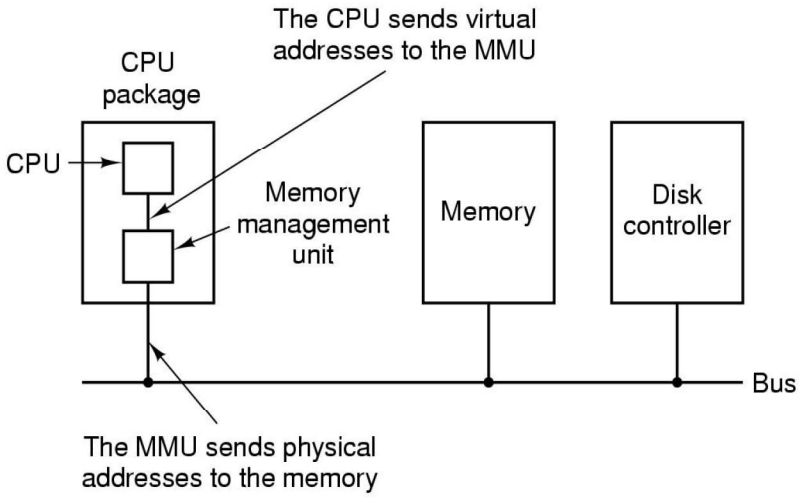


Figure 4-12 : MMU maps virtual addresses to physical addresses and puts them on memory bus

- When a program uses an instruction like:
  - –MOV REG, 1000
- It does this to copy the contents of memory address 1000 to REG.
- Addresses can be generated using indexing, base registers, segment registers, etc.
- These program-generated addresses are called virtual addresses and form the **virtual address space**.
- When virtual memory is used, the virtual addresses do not directly go to the memory bus.
- Instead, they go to an **MMU** (**Memory Management Unit**) that maps the virtual addresses onto the physical memory addresses

Pages and Page Frames

- The virtual address space consists of fixed-size units called **pages**. The corresponding units in the physical memory are called **page frames**.
- The size of a frame should be kept the same as that of a page to have maximum utilization of the main memory and to avoid external fragmentation.
- Virtual addresses divided into pages
    - 512 bytes-64 KB range
    - Transfer between RAM and disk is in whole pages
    - Example on next slide

Mapping of pages to page frames

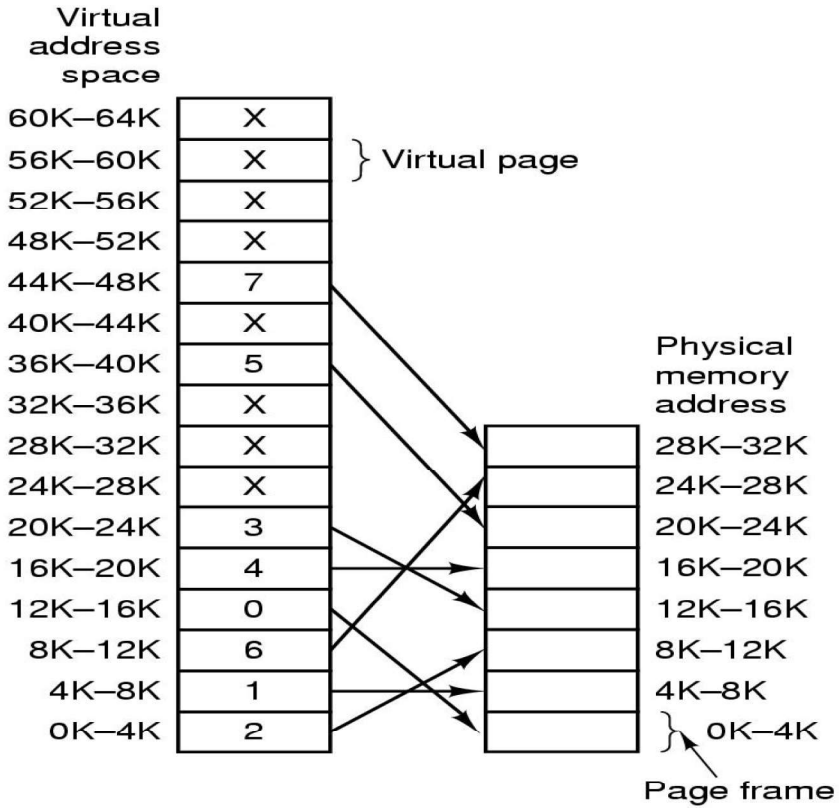


Figure 4-13 : The relation between virtual addresses and physical memory addresses is given by the page table. Every page begins on a multiple of 4096 and ends 4095 addresses higher, so 4K–8K really means 4096–8191 and 8K to 12K means 8192–12287.

Page Fault Processing

- Present/absent bit tells whether page is in memory
- What happens If address is not in memory?
- Trap to the OS
    - OS picks page to write to disk
    - Brings page with (needed) address into memory
    - Re-starts instruction

Address Translation Scheme

- The addresses generated by the machine while executing in user mode are logical addresses. The paging hardware translates these addresses to physical addresses.
- Address generated by CPU is divided into:
- **Page number (p):** used as an index into a page table which contains base address of each page in physical memory.
- **Page offset (d):** combined with base address to define the physical memory address that is sent to the MMU.

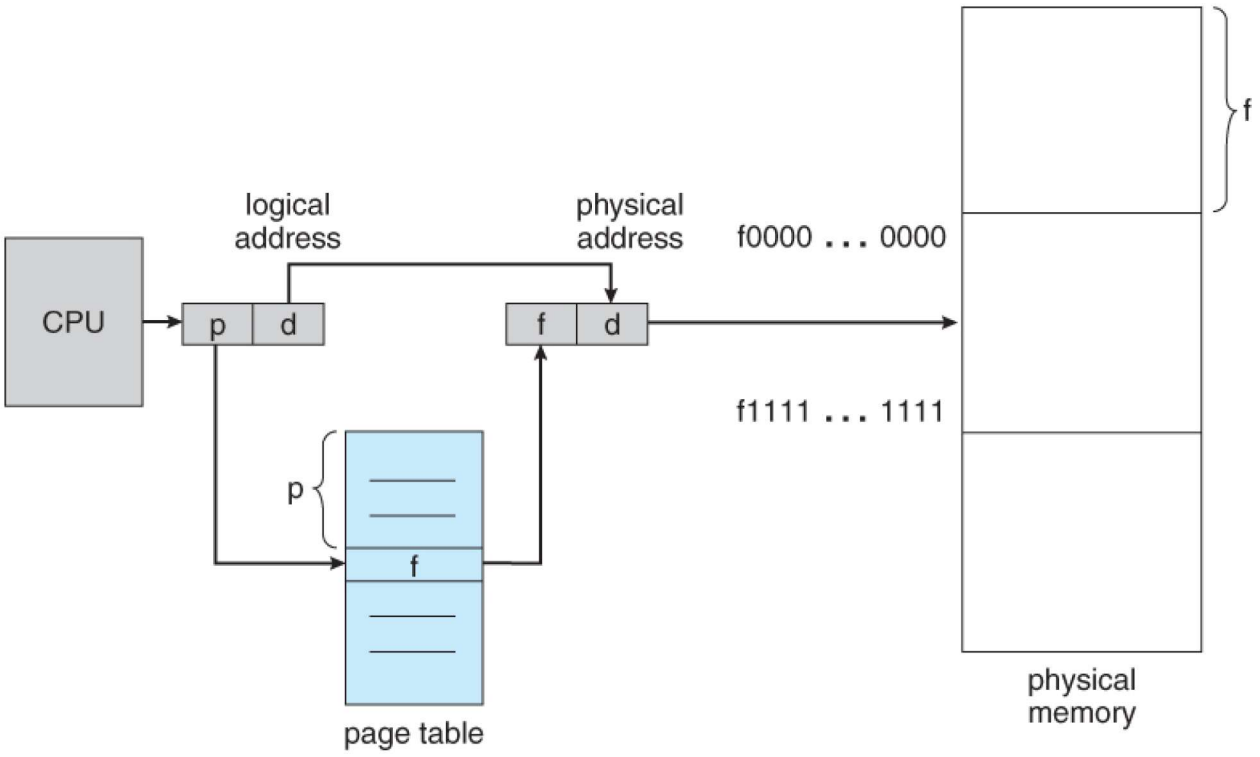


Figure 4-14 : Address Translation (logical to physical address)

Page Table

- The **page table** is used to look up what frame a particular page is stored in at the moment. or
- A **page table** is the data structure used by a virtual memory system in a computer operating system to store the mapping between virtual addresses and physical addresses
- Virtual address={virtual page number, offset}
- Virtual page number used to index into page table to find page frame number
- If present/absent bit is set to 1, attach page frame number to the front of the offset, creating the physical address
- which is sent on the memory bus

Address translation Example-MMU operation

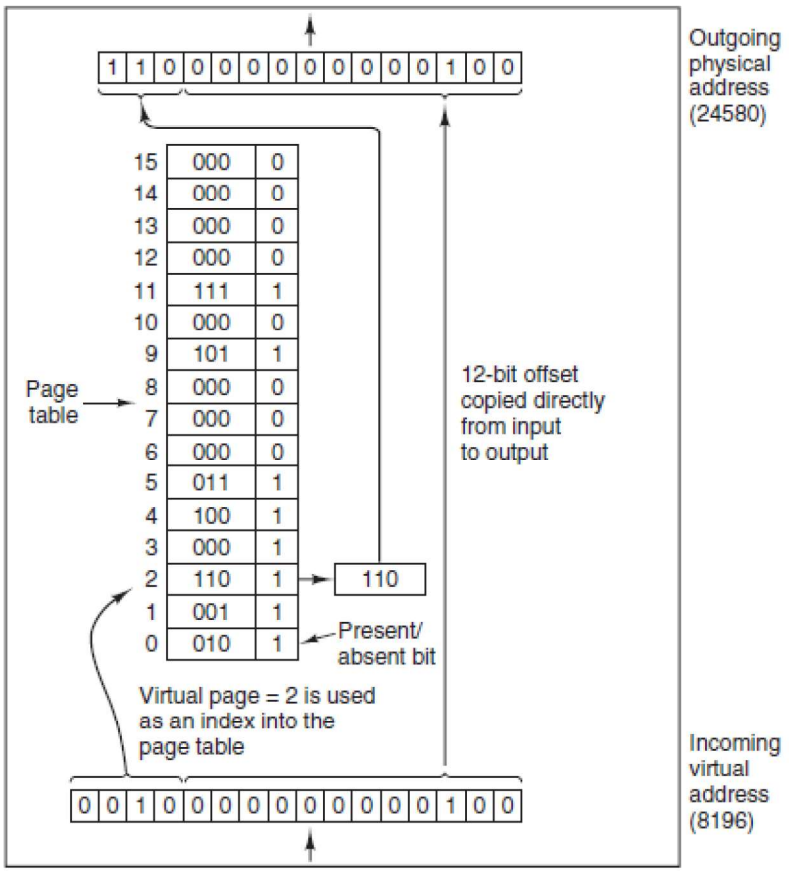


Figure 4-15 : The internal operation of the MMU with 164-KB pages.

Page Tables for Large Memories

Multi-level page tables

- Want to avoid keeping the entire page table in memory because it is too big.
- Hierarchy of page tables does this, page table of page tables.
- In (a) we have a 32-bit virtual address that is partitioned into a 10-bit PT1 field, a 10-bit PT2 field, and a 12-bit Offset field.
- Since offsets are 12 bits, pages are 4KB, and there are a total of 2^20 of them.
- The secret to the multilevel page table method is to avoid keeping all the page tables in memory all the time.
- In particular, those that are not needed should not be kept around.

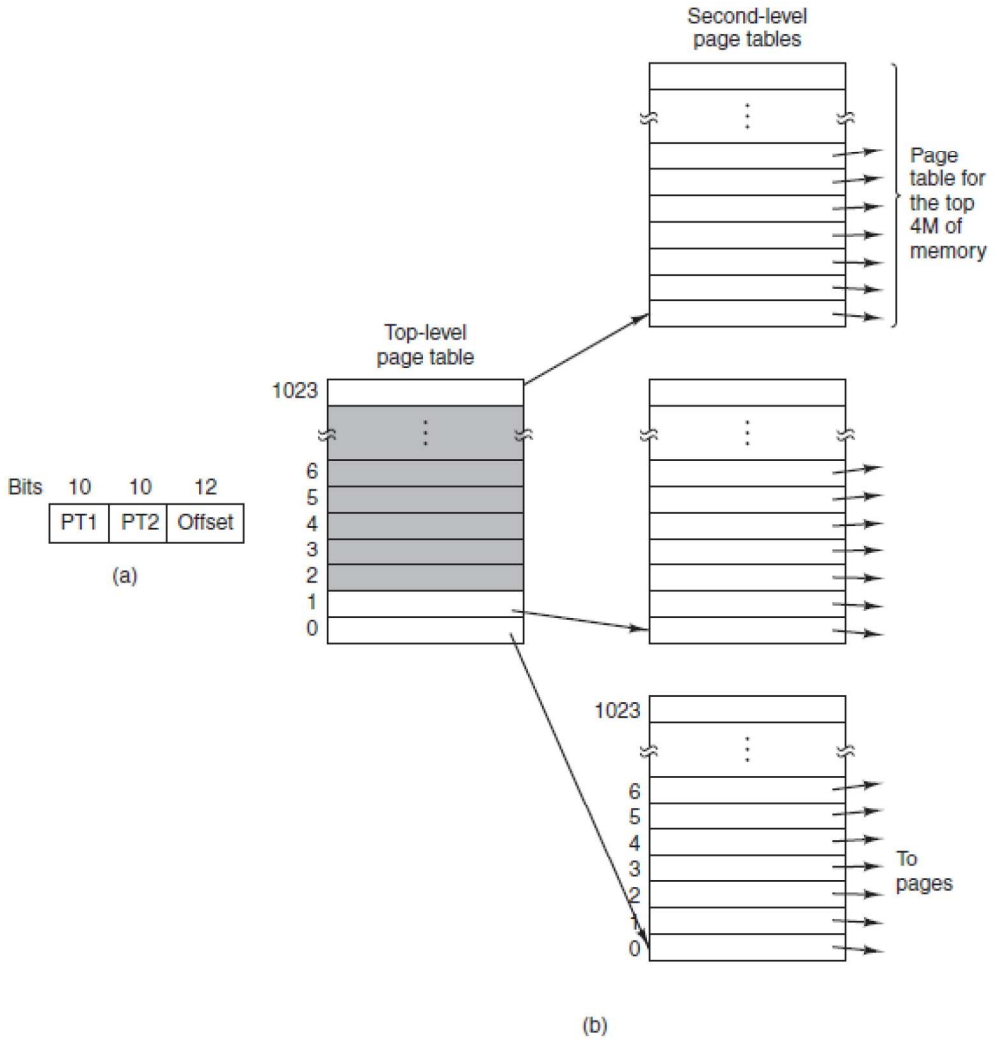


Figure 4-16 : (a) A 32-bit address with two page table fields. (b) Two-level page tables.

- In (b) we see how the two-level page table works.
  - On the left we have the top-level page table, with 1024 entries, corresponding to the 10-bit PT1 field.
  - When a virtual address is presented to the MMU, it first extracts the PT1 field and uses this value as an index into the top-level page table.
  - Each of these 1024 entries represents 4M because the entire 4-gigabyte virtual address space has been chopped into chunks of 1024 bytes.
- The entry located by indexing into the top-level page table yields the address of the page frame # of a second-level page table.

Hashed Page Tables

- Hashed page tables are a memory management technique often used to efficiently handle address spaces larger than 32 bits (like those in 64-bit architectures), where traditional linear or hierarchical page tables would consume too much memory.

How It Works

- **Hashing Virtual Page Numbers:**
  - The virtual page number (VPN) from a virtual address is passed through a hash function.
  - This hash function maps the VPN to an index in the hashed page table.
- **Structure of Hashed Page Table:**
  - Each entry (or "bucket") in the table points to a chain (linked list) of elements.
  - This is needed because different VPNs might hash to the same table index (a "collision").

- **Element Structure:**
  - Each element in the chain contains:
    - **The virtual page number (VPN)**-to check for a match.
    - **The value of the mapped physical page frame**-the translation result if it's a match.
    - **A pointer to the next element** in the chain- for handling collisions via chaining.
- **Lookup Process:**
  - To translate a virtual address, hash its VPN to find the relevant bucket.
  - Traverse the linked list (chain) at that bucket, comparing stored VPNs to the incoming VPN.
  - If a match is found, the corresponding physical frame number is returned.
  - If no match is found, it means the page is not present (page fault).

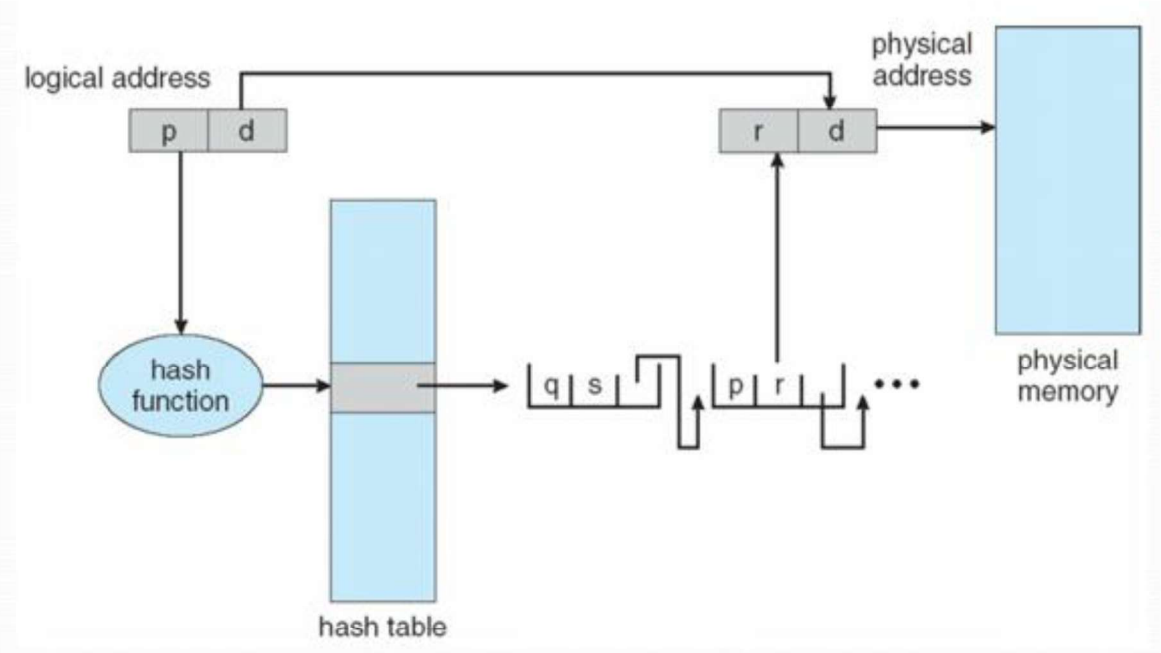


Figure 4-17 : Hashed Page Table

Inverted Page Tables

Traditional Page Tables:

- Each process has its own page table.
- Tracks: All possible logical (virtual) pages for that process.
- Problem: For large address spaces or many processes, the total number of page tables (and their size) becomes huge-high memory overhead.

Inverted Page Table Approach:

- Instead of maintaining a separate page table for every process:
- One global page table for the whole system.
- Tracks: All physical pages in RAM.

- Structure:
  - One entry per physical page (not per virtual page).
  - Each entry contains:
    - The virtual address (page number) currently stored in that physical frame.
    - The process ID (or similar info) that owns this page.
    - Additional metadata (like control bits for access, dirty, referenced, etc).

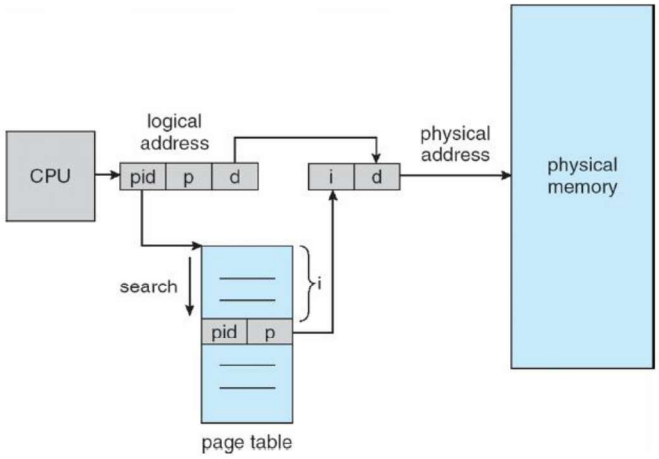


Figure 4-18 : Inverted Page Table

Structure of Page Table Entry

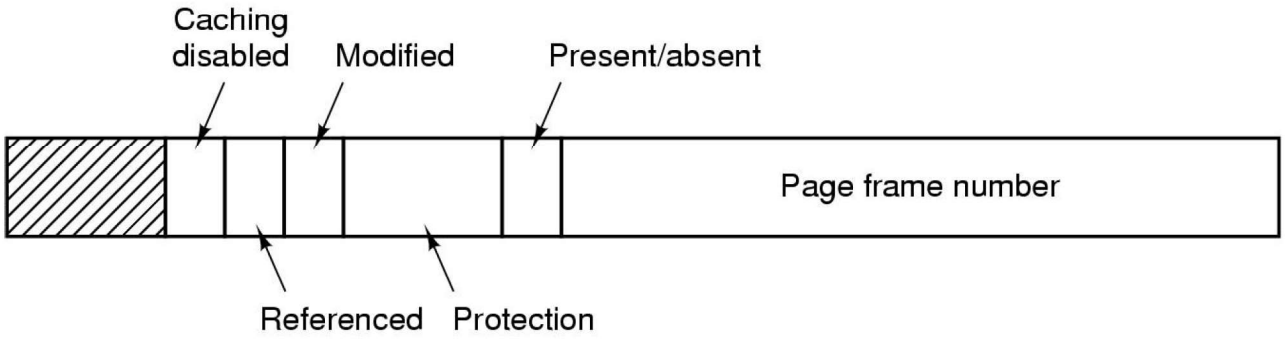


Figure 4-19 : A typical page table entry.

- The exact layout of a page table entry is highly machine dependent, but the kind of information present is roughly the same from machine to machine.
- The size varies from computer to computer, but 32 bits is a common size.

.

Page Frame Number:

- Indicates the actual location in physical memory (RAM) where the virtual page is stored.
- The main goal of address translation is to find this number so the MMU can translate a virtual address to a physical address.

Present/Absent Bit :

- Tells if the page is currently in physical memory.
- If set to 1: The entry is valid, and the virtual page is in RAM-translation can proceed.
- If set to 0: The page is not in memory; accessing it will cause a **page fault**, prompting the operating system to load the page from disk (swap or backing store).

Protection Bits :

- Control what type of access is allowed for this page.
- Example: In the simplest case, 1 bit: 0 = Read/Write, 1 = Read-Only.
- Prevents unauthorized writes or executes, protecting memory and enforcing security.

Modified Bit (Dirty Bit):

- Shows whether the page has been written to (modified) since it was loaded.
- Automatically set by hardware when the page is written.
- Helps the OS decide: If a page is to be swapped out and the bit is set, it needs to be written back to disk; otherwise, it can just be discarded.

Referenced Bit:

- Tracks whether the page has been accessed (read or written).
- Set by hardware on any access to the page.
- Used by the OS in page replacement algorithms (e.g., LRU, NRU) to identify recently used pages and decide which page to evict from memory.

Speeding Up Paging

Problems for paging:

- The mapping from virtual address to physical address must be fast.
- If the virtual address space is large, the page table will be large. 32 bit addresses now and 64 bits becoming more common

- A **translation lookaside buffer** (**TLB**) is a memory cache that is used to reduce the time taken to access a user memory location.
- It is a part of the chip's memory-management unit (MMU).
- The CPU generates the logical address, which contains the page number and the page offset.

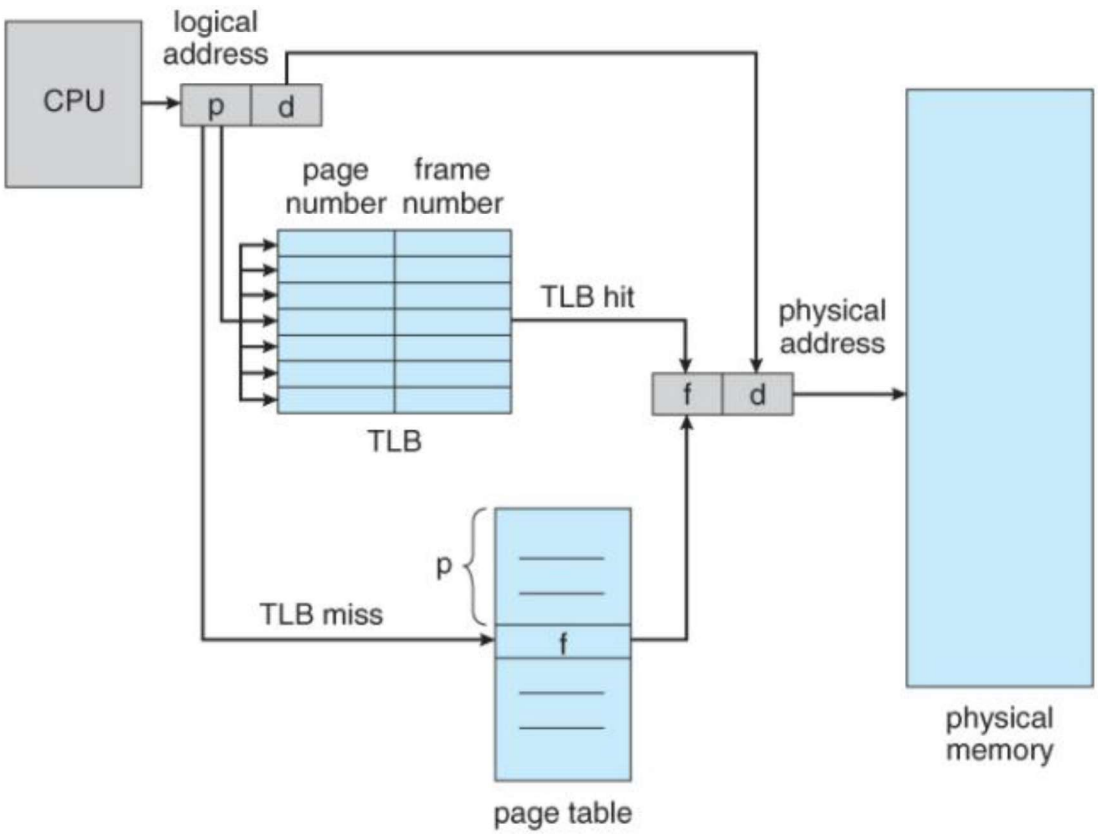


Figure 4-20 :TLB (Translation Look Aside Buffer)

- The page number is used to index into the page table, to get the corresponding page frame number, and once we have the page frame of the physical memory(also called main memory), we can apply the page offset to get the right word of memory.
- The thing is that page table is stored in physical memory, and sometimes can be very large, **so to speed up the translation of logical address to physical address** , we use TLB, **which is made of expensive and faster associative memory**, So instead of going into page table first, we go into the TLB and use page number to index into the TLB.
- **TLB Hit:** If the TLB has the page frame number for the requested page, this is called a **TLB hit** (no need to check the page table).
- **TLB Miss:** If the TLB does not have the needed entry, it's a **TLB miss** (so the system must look up the page frame in the main page table).

- **Page Fault:** Happens when a program tries to access a page that is not in physical memory (RAM), so the operating system must load it from secondary storage (like a hard drive).
- **Cache Hit:** Occurs when the data requested by the CPU is found in the cache memory, allowing for much faster access than fetching from main memory.
- **Cache Miss :** Happens when the data is not in cache memory; the system then retrieves it from main memory, which is slower, possibly involving page table or TLB lookups.

Page Replacement Algorithms

- First-in First-Out (FIFO) page replacement
- Second chance page replacement
- Optimal page replacement
- Least Recently Used (LRU) page replacement
- Least Frequently Used (LFU) page replacement
- Clock page replacement
- WSClock page replacement
- Not recently used (NRU) page replacement

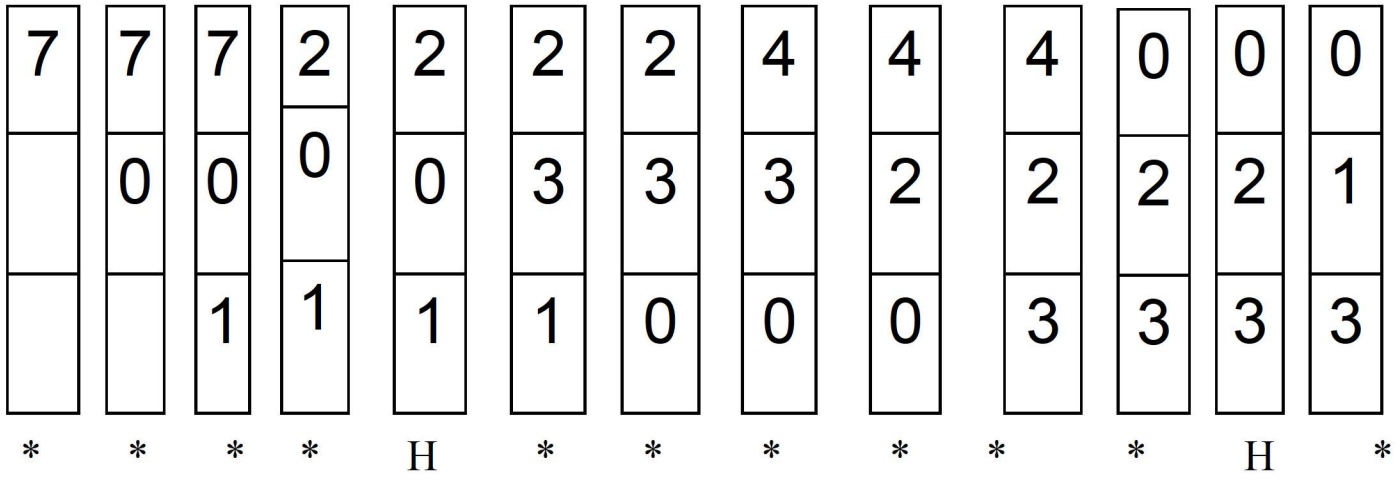
- If a process requests for page and that page is found in the main memory then it is called **page hit**, otherwise **page miss** or **page fault**.

First-in, first-out (FIFO)

- The simplest page-replacement algorithm is a FIFO algorithm.
- The first-in, first-out (FIFO) page replacement algorithm is a low-overhead algorithm that requires little book-keeping on the part of the operating system.
- The idea is obvious from the name - the operating system keeps track of all the pages in memory in a queue, with the most recent arrival at the back, and the earliest arrival in front.
- When a page needs to be replaced, the page at the front of the queue (the oldest page) is selected.
    - While FIFO is cheap and intuitive, it performs poorly in practical application. Thus, it is rarely used in its unmodified form. This algorithm experiences **Belady's anomaly**, in which increasing the number of frames available can actually ***increase*** the number of page faults that occur.

Example:- Consider the main memory consists of 3 frames for FIFO Page Replacement algorithm and input pattern of memory pages.

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



Here, Miss/Faults represented as * or M and Hit represented as H , Total references=13

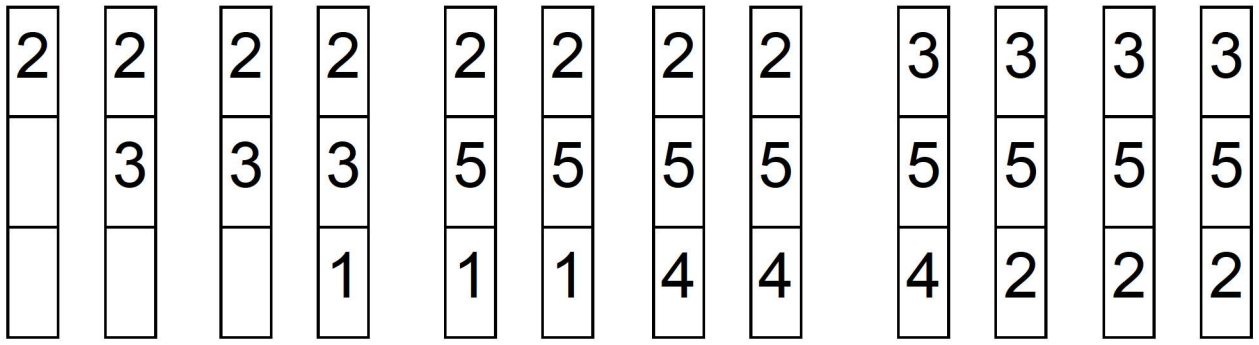
Total Page Hits=2 , [Hit ratio= no. of hits / no. of references=>2/13]

Total Page faults/page miss=11 , [Miss ratio= no. of miss / no. of references=>11/13]

Second Chance

Example:- Consider the main memory consists of 3 frames for Second chance Page Replacement algorithm and input pattern of memory pages.

2, 3, 2, 1, 5, 2, 4, 5, 3, 2, 3, 5



Total Page faults= 7

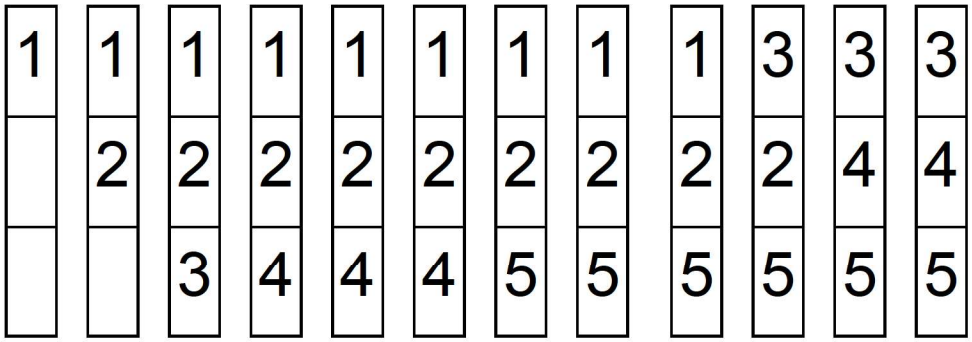
Total Page hits=5

Optimal page replacement(OPT)

- The theoretically optimal page replacement algorithm (also known as OPT, clairvoyant replacement algorithm, or Belady's optimal page replacement policy) is an algorithm that works as follows:
  - When a page needs to be swapped in, the operating system swaps out the page whose next use will occur farthest in the future.
  - For example, a page that is not going to be used for the next 6 seconds will be swapped out over a page that is going to be used within the next 0.4 seconds.
- This algorithm cannot be implemented in the general-purpose operating system because it is impossible to compute reliably how long it will be before a page is going to be used, except when all software that will run on a system is either known beforehand and is amenable to the static analysis of its memory reference patterns, or only a class of applications allowing run-time analysis.

Example:- Consider the main memory consists of 3 frames for OPT Page Replacement algorithm and input pattern of memory pages.

1,2,3,4,1,2,5,1,2,3,4,5



Total page faults=7

Total page Hits=5

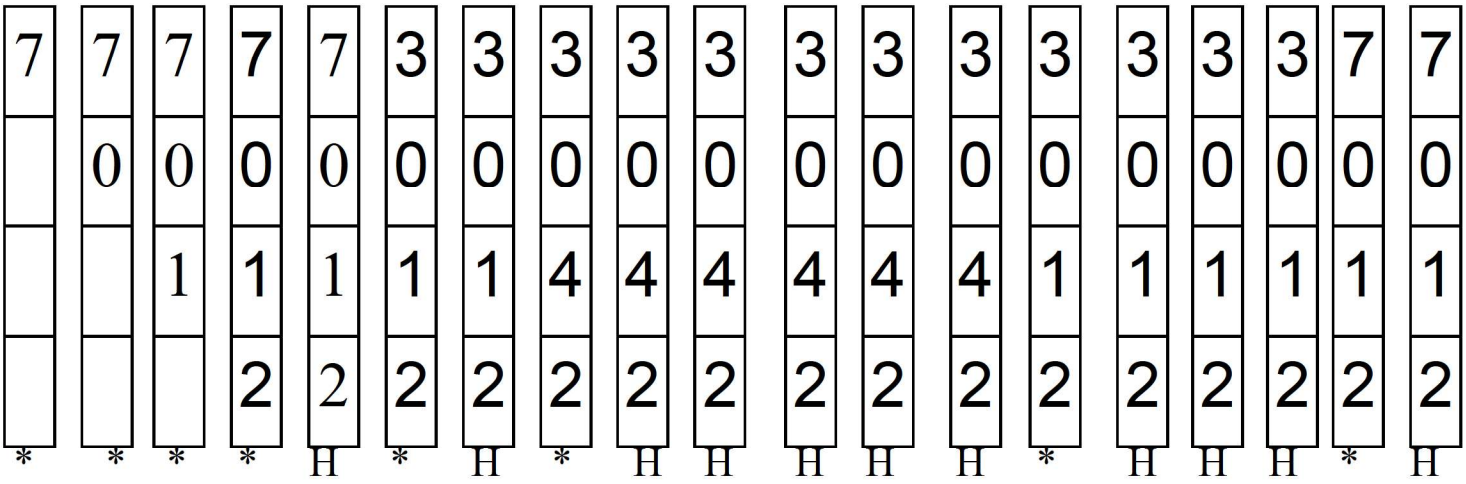
Least Recently Used (LRU)

- LRU works on the idea that pages that have been most heavily used in the past few instructions are most likely to be used heavily in the next few instructions too.
- While LRU can provide near-optimal performance in theory (almost as good as Adaptive Replacement Cache), it is rather expensive to implement in practice.
- There are a few implementation methods for this algorithm that try to reduce the cost yet keep as much of the performance as possible.
- The most expensive method is the linked list method, which uses a linked list containing all the pages in memory.
- At the back of this list is the least recently used page, and at the front is the most recently used page.

- The difficulty is that the list must be updated on every memory reference.
- Another method that requires hardware support is as follows: suppose the hardware has a 64-bit counter that is incremented at every instruction.
- Whenever a page is accessed, it gains a value equal to the counter at the time of page access.
- Whenever a page needs to be replaced, the operating system selects the page with the lowest counter and swaps it out.
- With present hardware, this is not feasible because the OS needs to examine the counter for every page in memory.

Example:- Consider the main memory consists of 4 frames for LRU Page Replacement algorithm and input pattern of memory pages.

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



Page Faults=8

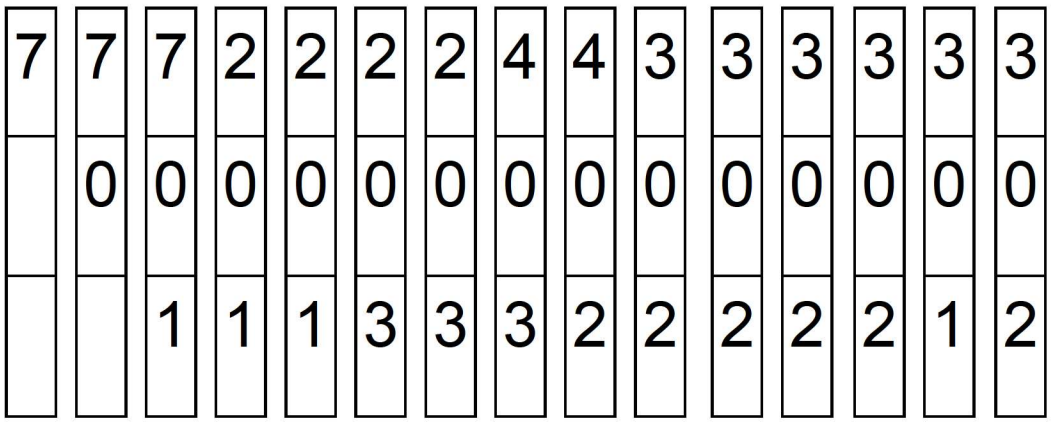
Page Hits=11

Least Frequently Used (LFU)

- Least Frequently Used (LFU) is a type of cache algorithm used to manage memory within a computer.
- The standard characteristics of this method involve the system keeping track of the number of times a block is referenced in memory.
- When the cache is full and requires more room, the system will purge the item with the lowest reference frequency.

Example (using 3 frames):- Reference String is:

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

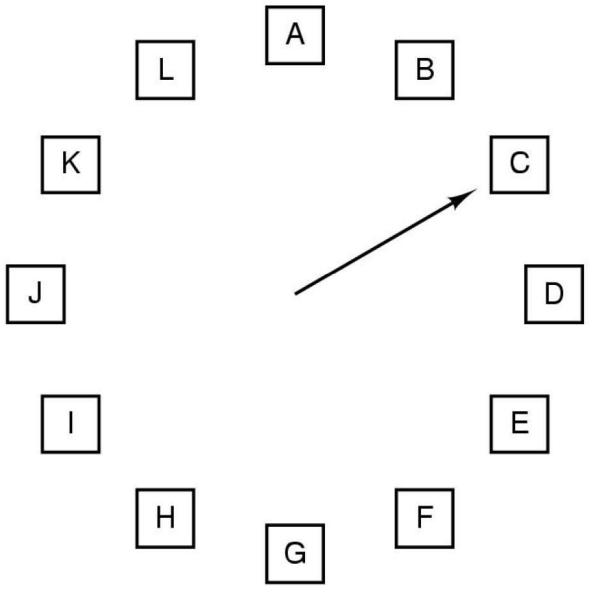


Total page Hits=5

Total page Faults=10

Clock

- Keep all the page frames on a circular list in the form of a clock



When a page fault occurs, the page the hand is pointing to is inspected. The action taken depends on the R bit:

R = 0: Evict the page

R = 1: Clear R and advance hand

- When a page fault occurs, the page being pointed to by the hand is inspected.
- If R =0, the page is evicted, the new page is inserted into the clock in its place, and the hand is advanced one position.
- If R =1, it is cleared and the hand is advanced to the next page. This process is repeated until a page is found with R = 0. Not surprisingly, this algorithm is called **clock**.
- **Does not use the age** of a page to decide which to evict (unlike LRU, Least Recently Used).
- **Faster:** Does not manipulate a list; just moves the hand around the clock.
- **Does not distinguish** between pages that have not been referenced for different lengths of time; only whether they were referenced since the last cycle.

The Working Set Model

- The working set of a process is **the set of pages expected to be used by that process during a specific time interval** (the "window" or "Δ" interval).
- The working set model **isn't a strict page replacement algorithm**; it's more like a **medium-term scheduling** concept that helps manage memory allocation for processes.
- The concept defines what memory a process requires in a given time interval-helping the operating system decide how many pages to keep in RAM for a process to run efficiently.

Demand Paging:

- A memory management scheme where pages are loaded only when needed.
- The OS tries to execute the first instruction of a process, causes a page fault (since the page isn't in memory). Continue until all **necessary pages (working set)** are in memory.
- An optimization over demand paging. The OS tries to ensure the entire working set is loaded before the process runs, reducing page faults called **pre-paging**.

Thrashing:

- If memory is too small to hold the entire working set, the process will continually fault pages in and out—this is called **thrashing** (excessive paging activity with little actual progress)**.** CPU spends more time on paging than executing, leading to poor performance.

Locality of reference:

- Instead of loading the entire process in the main memory, OS can load only those number of pages in the main memory that are frequently access the same set of pages for a period of time by the CPU.
- This allows the operating system to **optimize memory usage** by loading only the frequently used pages into main memory.
- Similarly, the OS only needs to maintain the **page table entries** corresponding to these active pages.

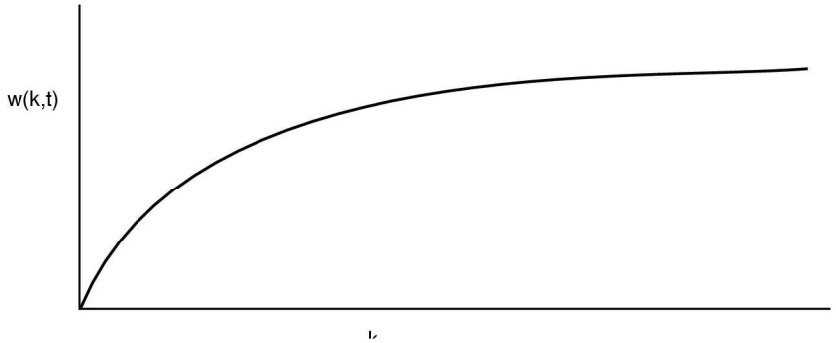


Figure 4-21: The working set is the set of pages used by the *k* most recent memory references. The function *w*(*k*, *t*) is the size of the working set at time *t*.

Working Set Page Replacement

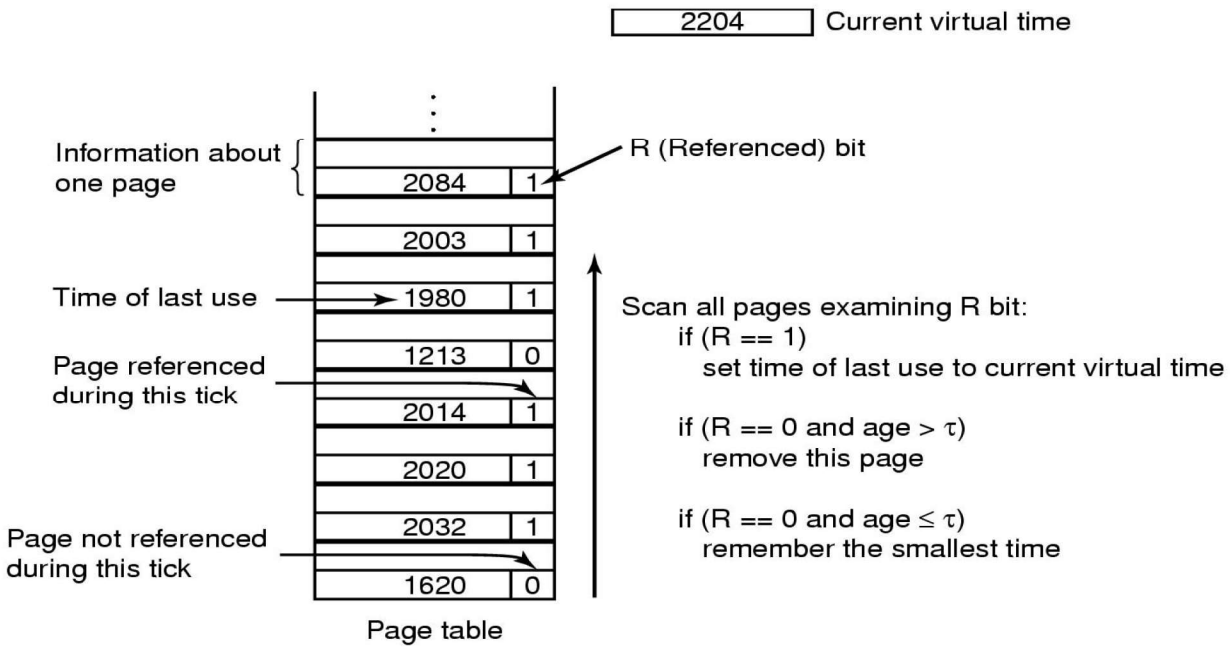


Figure 4-22. The working set algorithm.

Weakness with WS algorithm

- Need to scan entire page table at each page fault to find a victim
- Use clock idea with working set algorithm

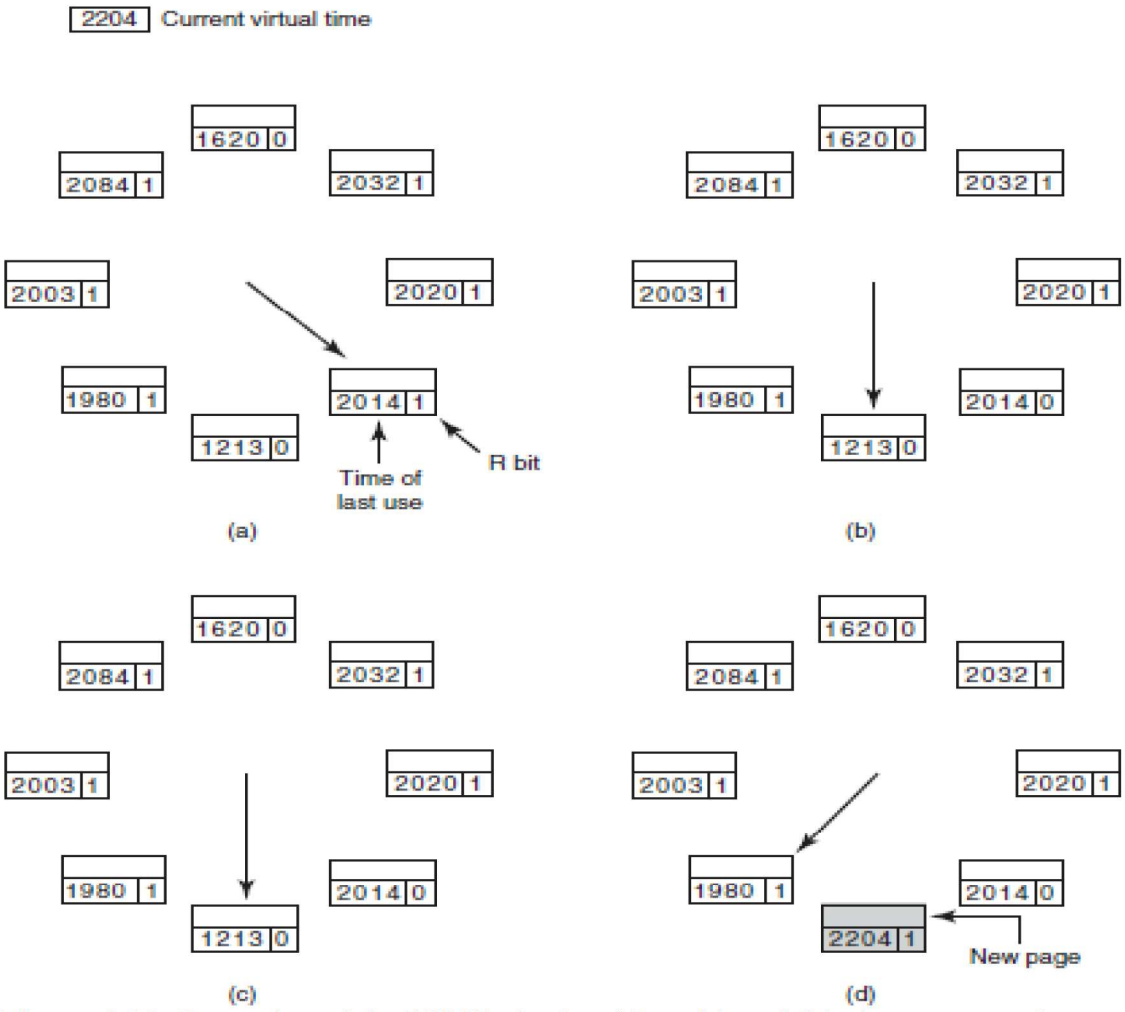


Figure 4-23: Operation of the WSClock algorithm. (a) and (b) give an example of what happens when *R* = 1. (c) and (d) give an example of *R* = 0.

The WSClock Page Replacement Algorithm

- If the hand comes all the way around to its starting point there are two cases to consider:

- At least one write has been scheduled.
  - Hand keeps moving looking for clean page. Finds it because a write eventually completes- evicts first clean page hand comes to.
- No writes have been scheduled.
  - Evict first (clean) page

Summary of Page Replacement Algorithms

Algorithm	Comment
Optimal	Not implementable, but useful as a benchmark
NRU (Not Recently Used)	Very crude approximation of LRU
FIFO (First-In, First-Out)	Might throw out important pages
Second chance	Big improvement over FIFO
Clock	Realistic
LRU (Least Recently Used)	Excellent, but difficult to implement exactly
NFU (Not Frequently Used)	Fairly crude approximation to LRU
Aging	Efficient algorithm that approximates LRU well
Working set	Somewhat expensive to implement
WSClock	Good efficient algorithm

.

Belady's Anomaly

- Belady's Anomaly is the phenomenon of increasing the number of page faults on increasing the number of frames in main memory.
- Following page replacement algorithms suffer from Belady's Anomaly.
  - FIFO Page Replacement Algorithm
  - Random Page Replacement Algorithm
  - Second Chance Algorithm
- "Algorithms suffer from Belady's Anomaly" does not mean that always the number of page faults will increase on increasing the number of frames in main memory.
- This unusual behavior is observed only sometimes.

Example: Consider FIFO page replacement

- Reference string : 0,1, 5, 3, 0, 1, 4, 0, 1, 5, 3, 4

    - Case 1: Using 3 frames

Ref.	0	1	5	3	0	1	4	0	1	5	3	4
Frame 1	0	0	0	3	3	3	4	4	4	4	4	4
Frame 2		1	1	1	0	0	0	0	0	5	5	5
Frame 3			5	5	5	1	1	1	1	1	3	3
	M	M	M	M	M	M	M	H	H	M	M	H

Number of Page Miss/Faults = 9

- Case 2: Using 4 frames

Ref.	0	1	5	3	0	1	4	0	1	5	3	4
Frame 1	0	0	0	0	0	0	4	4	4	4	3	3
Frame 2		1	1	1	1	1	1	0	0	0	0	4
Frame 3			5	5	5	5	5	5	1	1	1	1
Frame 4				3	3	3	3	3	3	5	5	5
	M	M	M	M	H	H	M	M	M	M	M	M

Number of Page Miss/Faults = 10

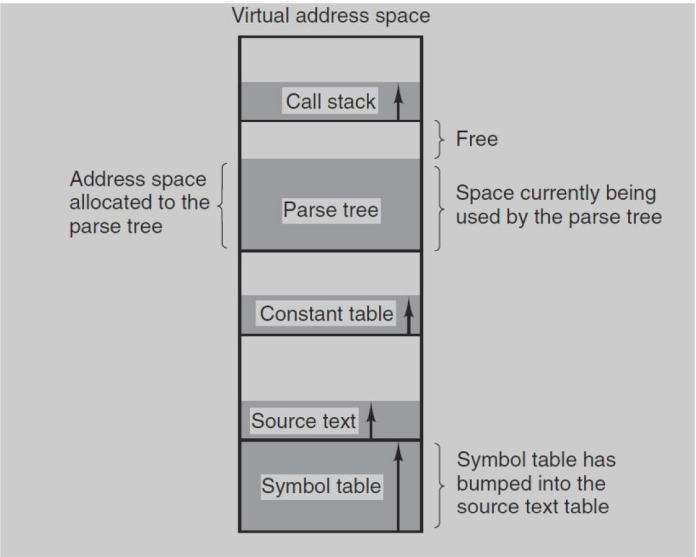
Therefore, in this example, the number of page faults is increasing by increasing the number of frames hence this suffers from Belady's Anomaly.

Segmentation

- **Segmentation** is a memory management technique where the memory is divided into variable-sized units called **segments**.
- Each process is divided into different segments such as:
  - Code segment
  - Data segment
  - Stack segment
  - Heap segment
- Each process has a **segment table** associated with it, which stores information about its segments.
- The main entries in the segment table are:
  - **Base**: The starting physical address of the segment in memory.
  - **Limit**: The length of the segment, which helps in bounds checking to prevent illegal memory access.

- A compiler is a good example to understand segmentation, as it deals with several different types of data structures that can be stored in separate segments. These might include:
    1. **Source text**: The actual program code to be compiled (e.g., for listings in batch systems).
    2. **Symbol table**: Stores variable names and attributes.
    3. **Constant table**: Holds integer and floating-point constants.
    4. **Parse tree**: Represents the syntactic structure of the program.
    5. **Call stack**: Manages procedure calls and return addresses during compilation.
- Each of the first four tables grows continuously as compilation proceeds. The last one grows and shrinks in unpredictable ways during compilation.

- In a one-dimensional memory, these five tables would have to be allocated contiguous chunks of virtual address space, as in Fig. below.



- If a program uses a large number of variables, the symbol table segment might need more memory, while other segments like the parse tree or constants may use less.

- In systems with segmentation, each segment can grow or shrink independently, allowing more flexible memory usage.

Segment Addressing:

- Each logical address generated by a program is represented as a **pair (segment number, offset)**.
- **Segment number (s):** Number of bits required to represent the segment.
- **Segment offset (d):** Number of bits required to represent the size of the segment.
- The segment number is used to index into the segment table to find the base address.
- The offset is added to the base to get the physical address.
- If the offset exceeds the limit, a **segmentation fault** occurs.

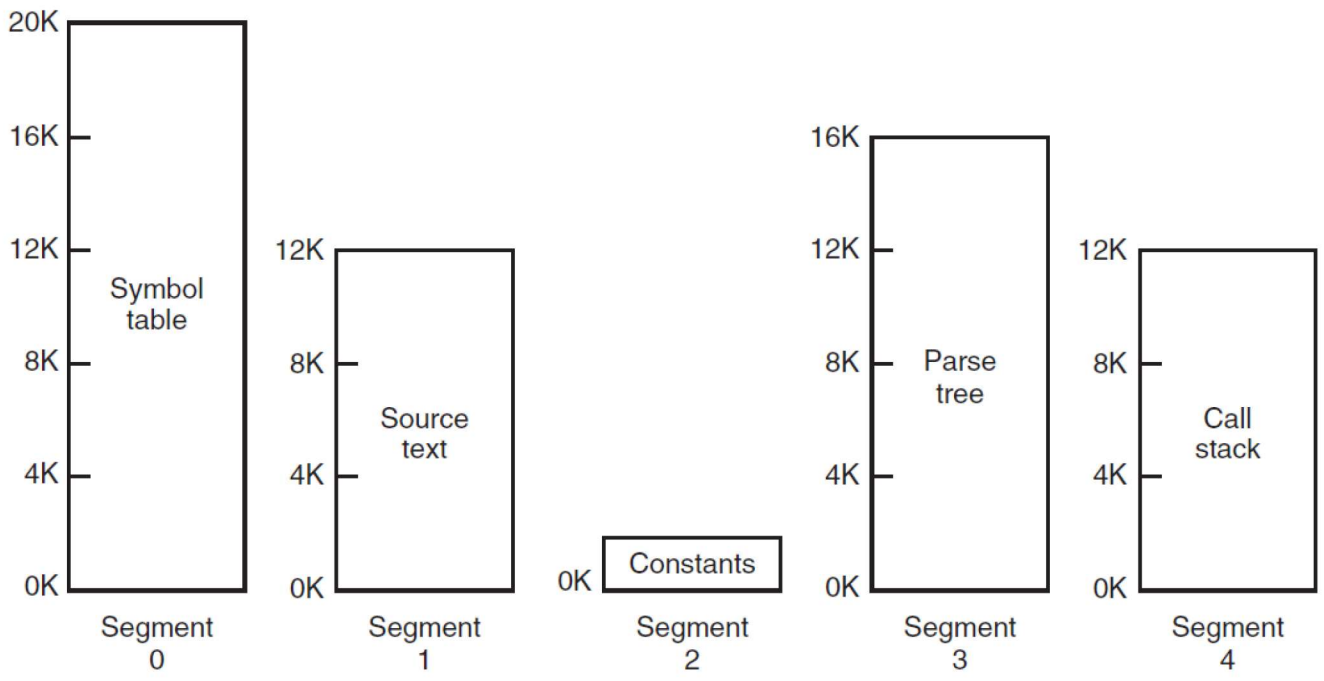
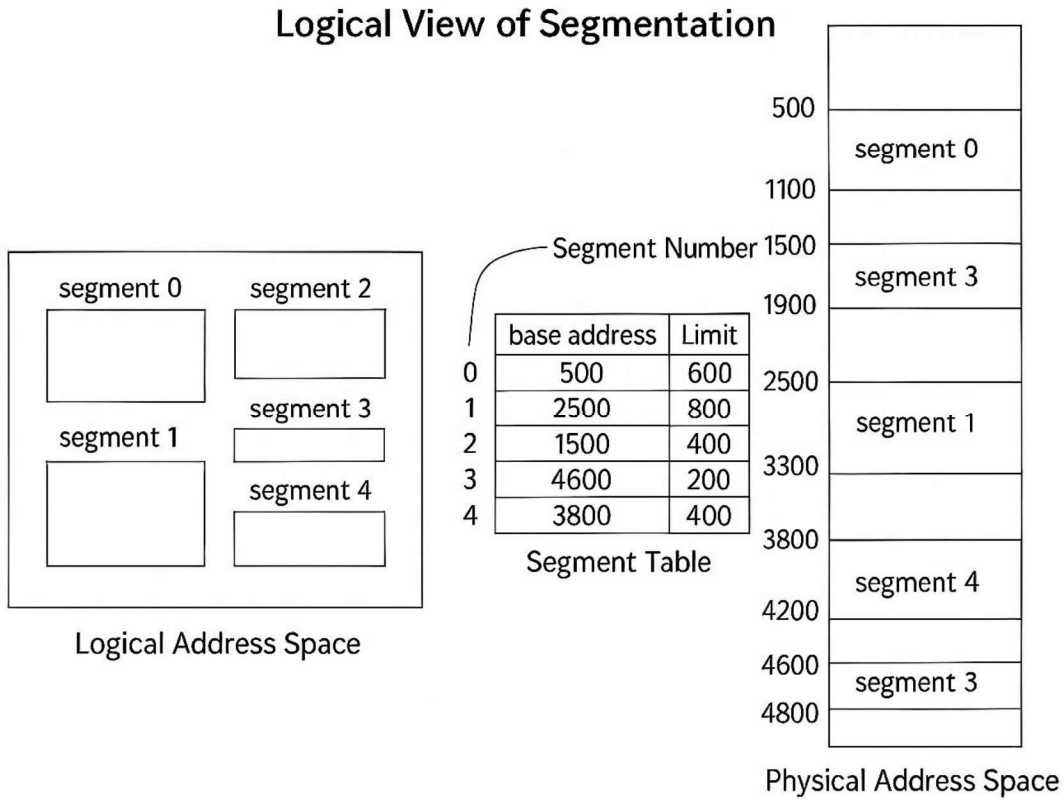
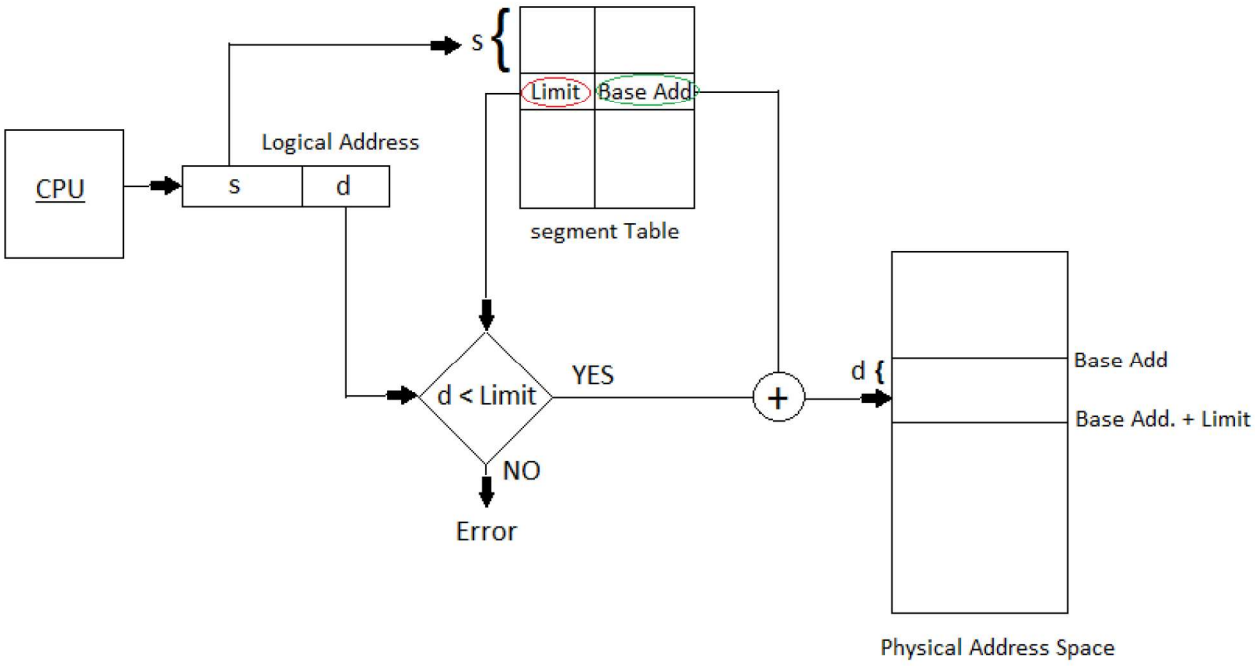


Figure 4-24: A segmented memory allows each table to grow or shrink independently of the other tables.

Segmentation Table Mapping to Physical Address



Translate Logical Address to Physical Address



- To convert a logical address to a physical address:

  If Offset < Limit:

  Physical Address = Base + Offset

  Else:

  Trap: Segmentation Fault (Address out of bounds)

Example:

- Logical Address = <Segment 1, Offset 50>, Base = 500, Limit = 600. Since , 50 < 600, it's valid. **Physical Address = 500 + 50 = 550**
- Logical Address = <Segment 1, Offset 650>, Here, Offset > Limit (650 > 600), **Segmentation Fault** (Invalid memory access).

Advantages of Segmentation

- **Dynamic Data Structure Support:** Simplifies the handling of data structures that grow and shrink dynamically.
- **Logical Addressing:** The address space of segment n is expressed as (n, local address) where (n, 0) is the starting address of the segment.
- **Independent Compilation:** Segments can be compiled separately, which supports modular programming and development.
- **Shared Procedures or Data:** Facilitates sharing of procedures or data between multiple processes.
- **Segment-wise Protection:** Each segment can have its own protection bits (e.g., read, write, execute), enhancing security and access control.
- **Smaller Segment Table:** Segment tables are typically smaller than page tables in paging, reducing memory overhead.

Drawbacks of Segmentation

- **External Fragmentation**: Because segments are variable-sized, it leads to gaps (unused memory) in RAM over time.
- **Complex Memory Management**: Allocating, deallocating, and compacting variable-sized segments increases OS complexity.
- **Performance Overhead**: Segment table lookups and bounds checking add overhead during address translation.
- **Limited Segment Size**: Segments are bounded by the system's max segment size, which may restrict large data handling.

Consideration	Paging	Segmentation
Need the programmer be aware that this technique is being used?	No	Yes
How many linear address spaces are there?	1	Many
Can the total address space exceed the size of physical memory?	Yes	Yes
Can procedures and data be distinguished and separately protected?	No	Yes
Can tables whose size fluctuates be accommodated easily?	No	Yes
Is sharing of procedures between users facilitated?	No	Yes
Why was this technique invented?	To get a large linear address space without having to buy more physical memory	To allow programs and data to be broken up into logically independent address spaces and to aid sharing and protection

Figure 4-25: Comparison of paging and segmentation.

Implementation of Pure Segmentation

- Paging uses fixed-size blocks (pages), while segmentation uses variable-sized blocks (segments).
- Figure (a) shows an example of physical memory initially containing five segments. Now consider what happens if segment 1 is evicted and segment 7, which is smaller, is put in its place.
- We arrive at the memory configuration of Fig.(b). Between segment 7 and segment 2 is an unused area—that is, a hole.
- Then segment 4 is replaced by segment 5, as in Fig.(c), and segment 3 is replaced by segment 6, as in Fig.(d).

- After the system has been running for a while, memory will be divided up into a number of chunks, some containing segments and some containing holes. This situation is known as **external fragmentation** or **checkerboarding**, which leads to **wasted memory** in the form of small unusable holes, wastes memory in the holes. It can be dealt with by **compaction**, as shown in Fig.(e).

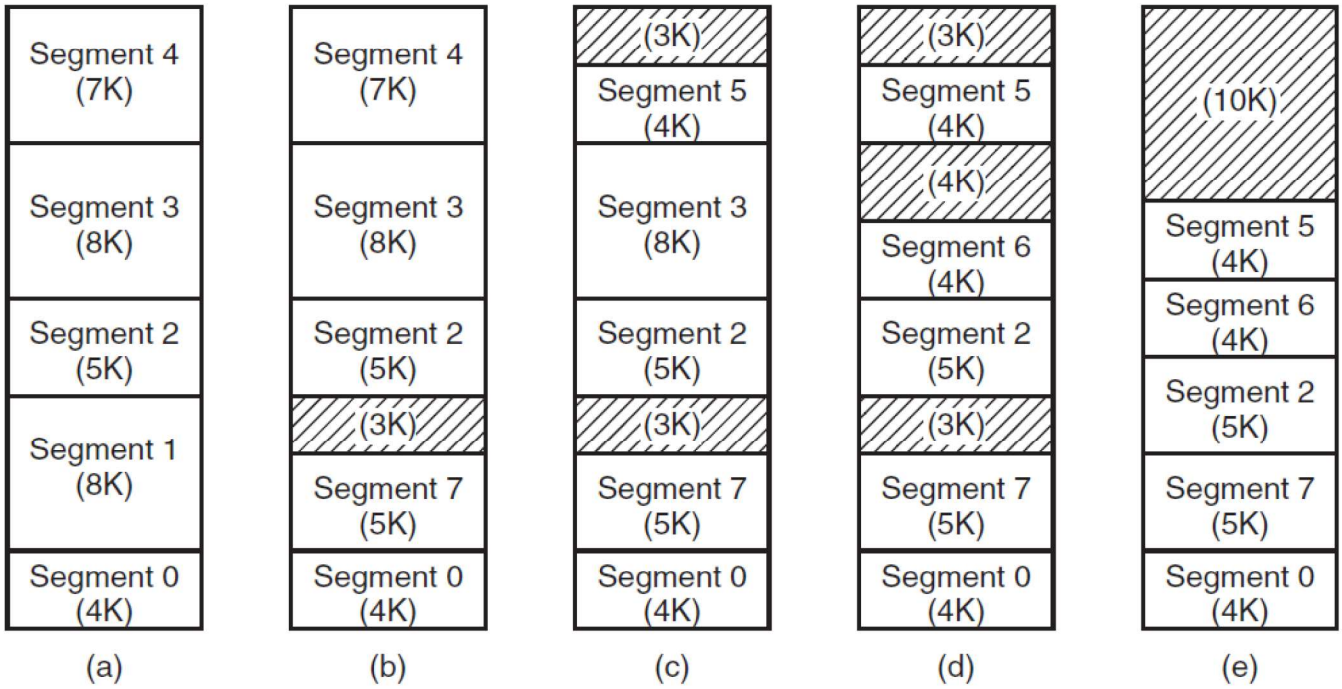


Figure 4-26 : (a)-(d) Development of checkerboarding. (e) Removal of the checkerboarding by compaction.

Segmentation with Paging: MULTICS

- To overcome the drawbacks of pure segmentation, Segmentation with Paging was introduced.
- **MULTICS** (Multiplexed Information and Computing Service) used this hybrid approach, developed by IBM mainframe systems.
- It was a pioneer in introducing segmentation to modern architectures.
- Every process can have multiple virtual address spaces (or segments).
- Each segment has its own page table.
- Advantage: Each segment can have the full virtual address space allowed by number of address bits.
- Disadvantage: High context switch penalty when switching from one segment to another, even within the same process.

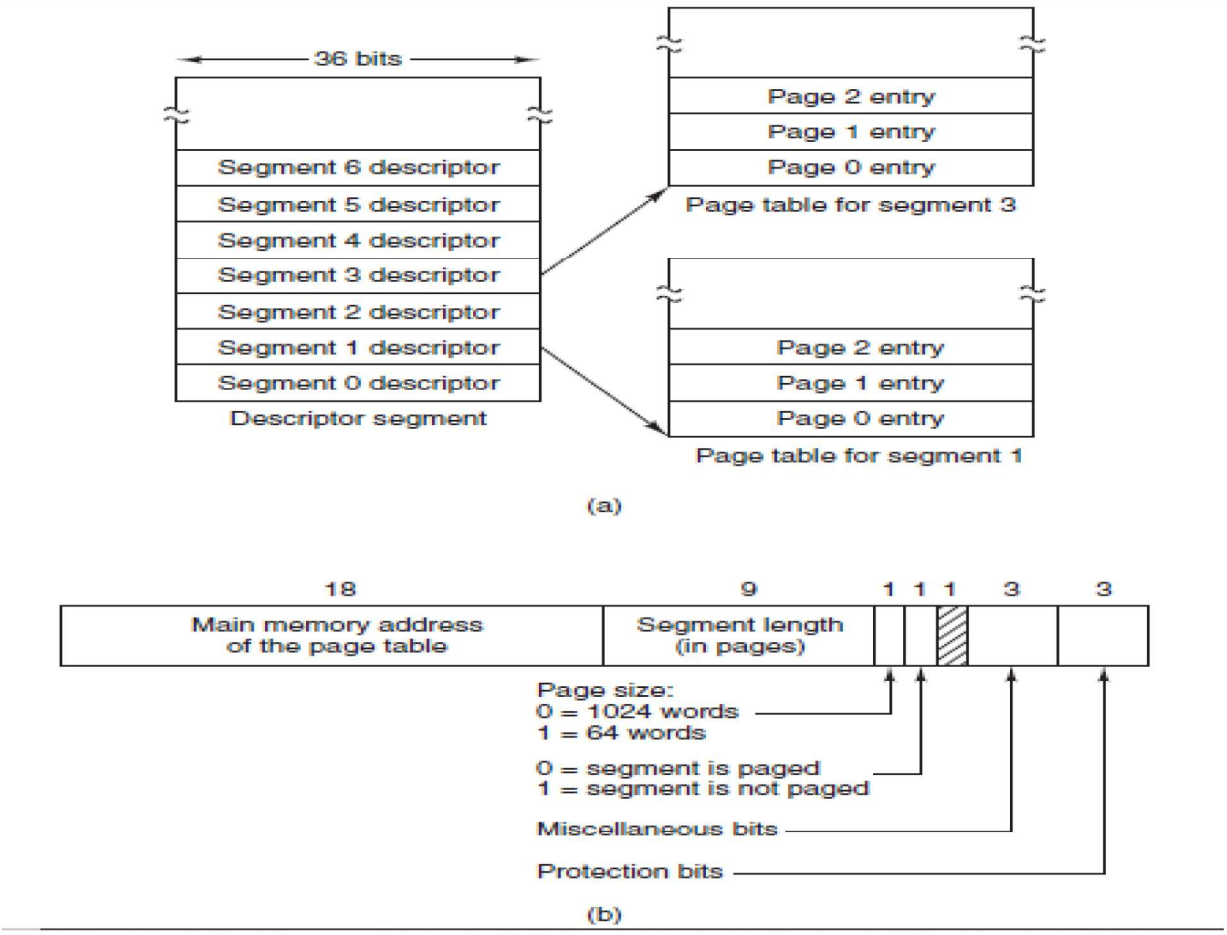


Figure 4-27: The MULTICS virtual memory. (a) The descriptor segment pointed to the page tables. (b) A segment descriptor. The numbers are the field lengths.

- Each MULTICS program had a segment table, with one descriptor per segment.
- A segment descriptor contained an indication of whether the segment was in main memory or not. If any part of the segment was in memory, the segment was considered to be in memory, and its page table was in memory.
- If the segment was in memory, its descriptor contained an 18-bit pointer to its page table, as in Fig. (a).
- The descriptor also contained the segment size, the protection bits, and other items.
- Figure (b) illustrates a segment descriptor. The address of the segment in secondary memory was not in the segment descriptor but in another table used by the segment fault handler.
- Each segment descriptor points to a page table.

- An address in MULTICS consisted of two parts: the segment and the address within the segment. The address within the segment was further divided into a page number and a word within the page.
  - **Segment Number :** It points to the appropriate Segment Number.
  - **Page Number:** It Points to the exact page within the segment.
  - **Page Offset:** Used as an offset within the page frame

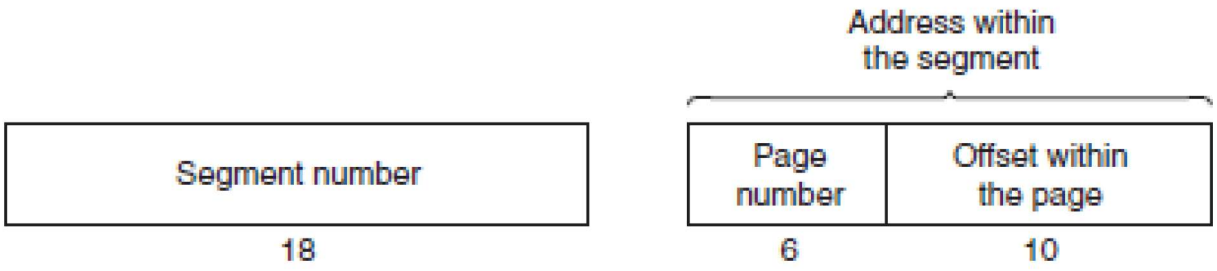


Figure 4-28: A 34-bit MULTICS virtual address.

Example: The Intel x86

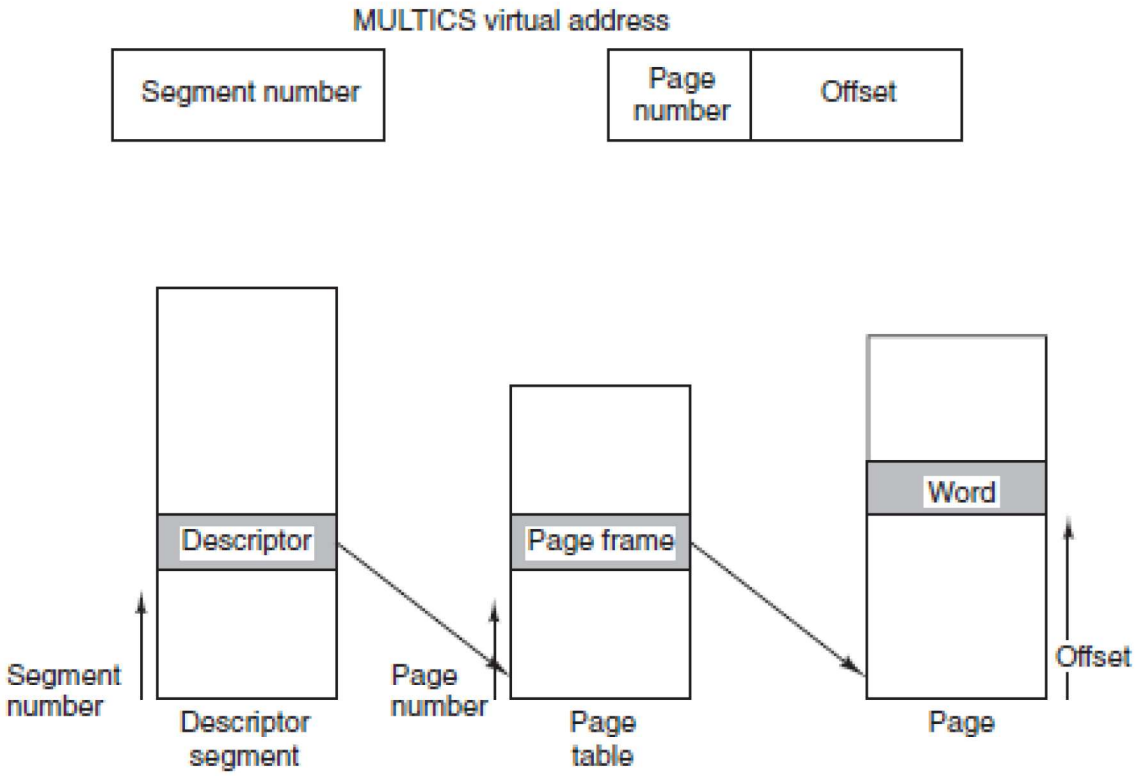


Figure 4-29: Conversion of a two-part MULTICS address into a main memory address.

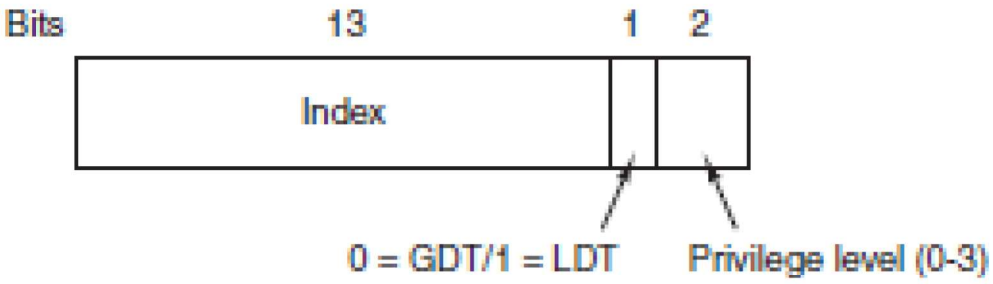


Figure 4-30: An x86 selector.

- The x86 virtual memory consists of two tables, called the **LDT** (**Local Descriptor Table**) and the **GDT** (**Global Descriptor Table**).
- Each program has its own LDT, but there is a single GDT, shared by all the programs on the computer. The
- LDT describes segments local to each program, including its code, data, stack, and so on, whereas the GDT describes system segments, including the operating system itself.

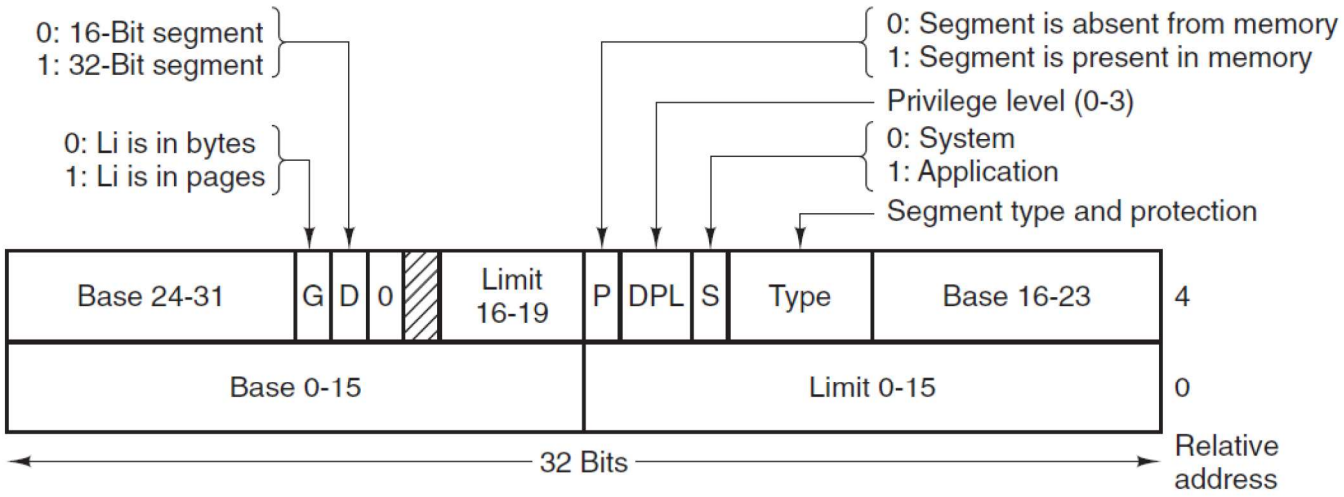


Figure 4-31: x86 code segment descriptor. Data segments differ slightly.

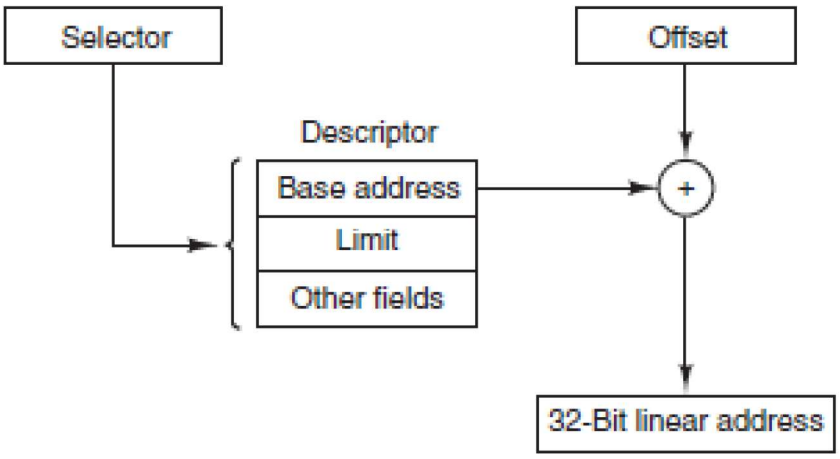


Figure 4-32: Conversion of a (selector, offset) pair to a linear address

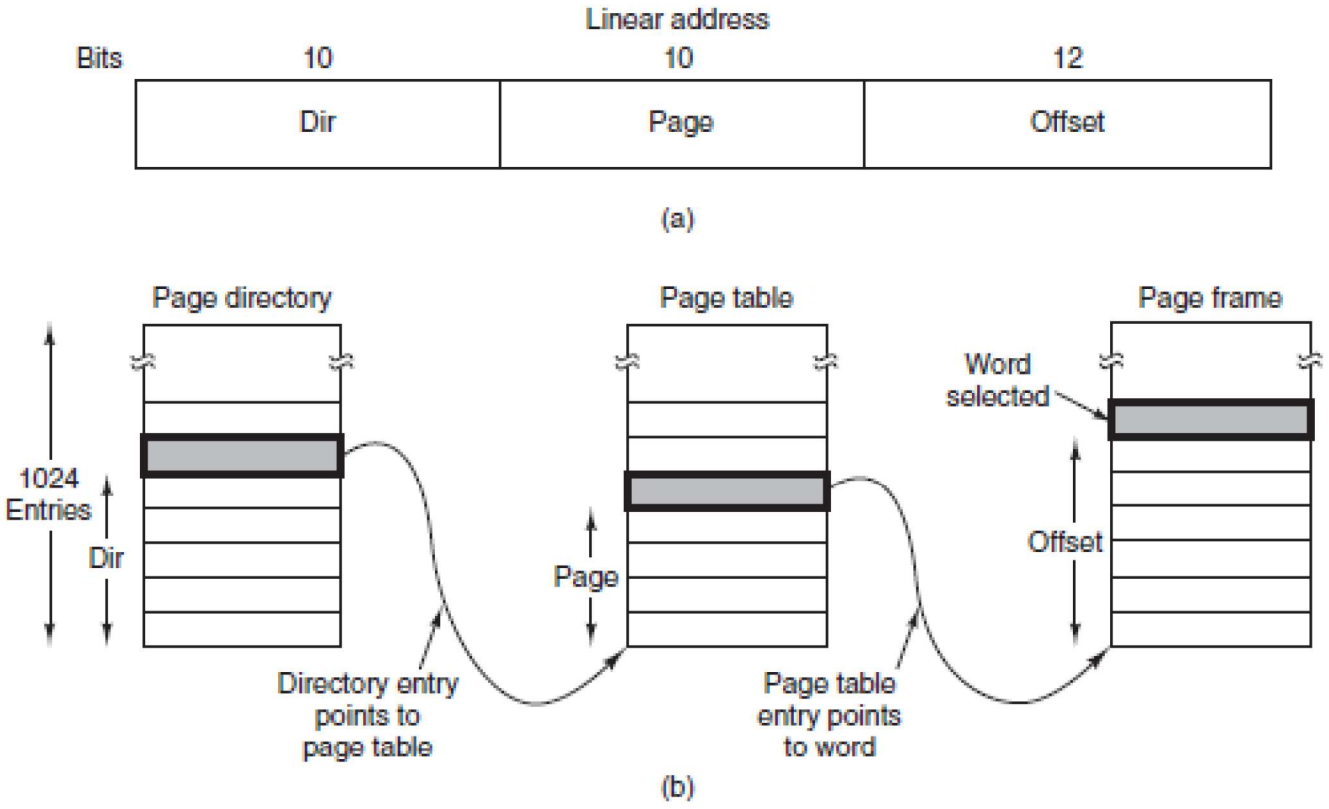


Figure 4-33: Mapping of a linear address onto a physical address.

- In fig.(A) linear address divided into three fields, dir, page, and offset.
- The **dir** field is used to index into the page directory to locate a pointer to the proper page table.
- The **page** field is used as an index into the page table to find the physical address of the page frame.
- The **offset** is added to the address of the page frame to get the physical address of the byte or word needed.