Memory Management

Introduction

- **Main memory (RAM)** is an important resource that must be carefully managed
- Over the years, people discovered the concept of a **memory hierarchy**, in which computers have a few megabytes of very fast, expensive, volatile **cache memory**, a few gigabytes of medium-speed, medium-priced, volatile **main memory**, and a few terabytes of slow, cheap, nonvolatile magnetic or solid-state **disk storage**, not to mention **removable storage,** such as DVDs and USB sticks. It is the job of the operating system to abstract this hierarchy into a useful model and then manage the abstraction.
- The part of the operating system that manages (part of) the memory hierarchy is called the **memory manager.** Its job is to efficiently manage memory: keep track of which parts of memory are in use, allocate memory to processes when they need it, and deallocate it when they are done.

❖ *Memory management is the functionality of an operating system which handles or manages primary memory and moves programs back and forth between main memory and disk during execution*

❖ Memory management is a function of the OS. It helps us with:
  - Managing the system memory (primary memory)
  - Handling the movement of processes between main memory and disk during execution.
  - Keeping track of every memory location, whether allocated or free
  - Checking the amount of memory to be allocated to a process.
  - Deciding which process will get memory and at what time
  - Tracking when some memory is free or unallocated and updating its status

Mono-programming vs Multi-programming

- **Mono-programming (Uniprogramming)** implies that only a single task or program is in the main memory at a particular time. It was more common in the initial computers and mobiles where one can run only a single application at a time.
    - It allows only one program to sit in the memory at one time.
    - The size is small as only one program is present.
    - The resources are allocated to the program that is in the memory at that time.
- **Multiprogramming** is where several programs can be run at the same time. The programs reside in the main memory or the RAM of the system at a time, and the operating system that handles multiple programs is known as a multiprogramming operating system.
    - The memory can hold several programs at a time.
    - The resources are allocated to different programs.
    - The size of the memory is larger as compared to uniprogramming.

Modelling Multi-programming

- When multiprogramming is used, the CPU utilization can be improved.
- An important concept in multiprogramming is the degree of multiprogramming. The degree of multiprogramming describes the maximum number of processes that a single-processor system can accommodate efficiently
- Assume that that a process spends p percent of its time waiting for I/O. With n processes in memory the probability that all n processes are waiting for I/O (meaning the CPU is idle) is p^n. The CPU utilization is then given by

CPU Utilization = $1 - p^n$

Example: Assume n = 4 and p = 50% i.e. 0.5

CPU Utilization = $1 - 0.5^4 = 0.9375$

NOTE: In mono-programming, CPU Utilization = $1 - p$.

Often $p>0.5$ which shows CPU utilization is poor in mono-programming.

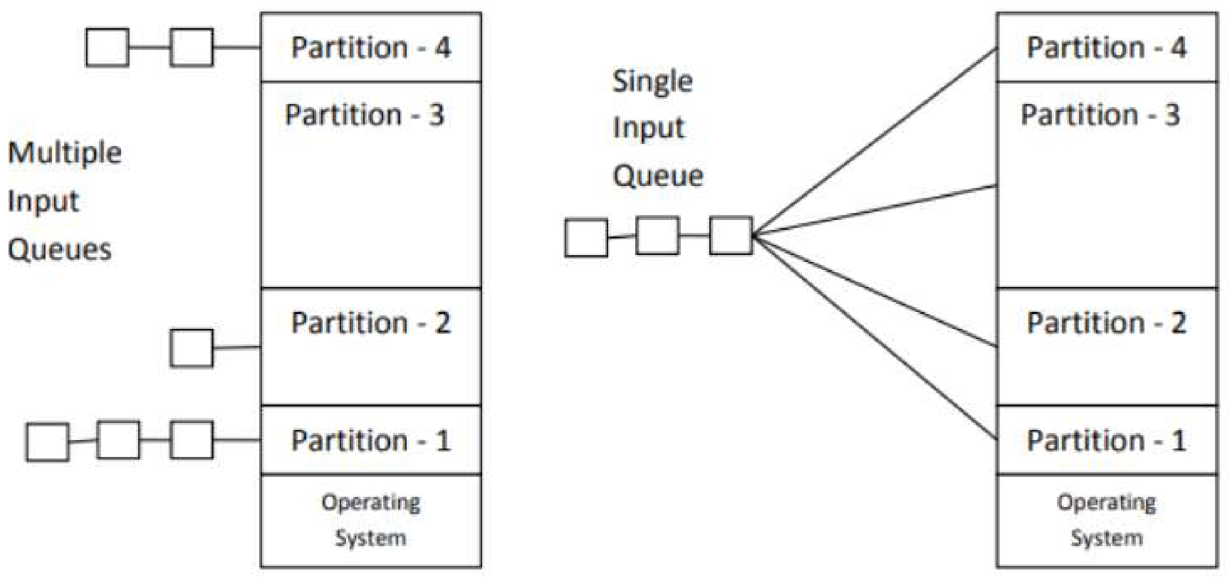
- To take maximum advantage of multiprogramming, several processes must reside in the computer's main memory at the same time. Thus, when one process requests input/output, the processor may switch to another process and continue to perform calculations without the delay associated with loading programs from secondary storage. When this new process yields the processor, another may be ready to use it.
- Many multiprogramming schemes have been implemented.
- Swapping can be used to run multiple programs concurrently. The process of bringing in each process in it's entirely in to memory, running it for a while and then putting in back to a disk is called **swapping.**

Multiprogramming with Fixed and Variable Partitions

In multi-programming environment several programs reside in main memory at a time and CPU passes its control rapidly between these programs. One way to support multiprogramming is to divide the main memory into several partitions each of which is allocated to single process. Depending on how and when partition are created there may be two types of partitions: static (fixed) and dynamic (variable)

Multiprogramming with Fixed Partitions

- Here memory is divided into fixed sized partitions.
- Size can be equal or unequal for different partitions.
- Generally unequal partitions are used for better utilizations.
- Each partition can accommodate exactly one process, means only single process can be placed in one partition.
- The partition boundaries are not movable.
- Whenever any program needs to be loaded in memory, a free partition big enough to hold the program is found. This partition will be allocated to that program or process.
- If there is no free partition available of required size, then the process needs to wait. Such process will be put in a queue.
- There are two ways to maintain queue
    1. Using multiple Input Queues.
        - Separate queue for each partition
    2. Using single Input Queue.
        - Single queue for all partitions



(a) (b)

Figure: (a) Fixed memory partition with separate input queue for each partition (b)Fixed memory partition with single input queue for all partitions

- The disadvantage of sorting the incoming jobs into separate queues beis requested apparent when the queue for a large partition is empty but the queue for a small partition is full, as is the case for partitions 1 and 3 in given above figure.
  - Here small jobs have to wait to get into memory, even though plenty of memory is free. An alternative organization is to maintain a single queue as in figure (b). Whenever a partition beis requested free, the job closest to the front of the queue that fits in it could be loaded into the empty partition and run.

Multiprogramming with Variable Partitions

- Here, memory is shared among operating system and various simultaneously running processes.
- Memory is not partitioned into any fixed partitioned. Also the number of partitioned is not fixed. Process is allocated exactly as much memory as it requires.
- Initially, the entire available memory is treated as a single free partition.
- Whenever any process enters in a system, a chunk of free memory big enough to fit the process is found and allocated. The remaining unoccupied space is treated as another free partition.
- If enough free memory is not available to fit the process, process needs to wait until required memory beis requested available.

- Whenever any process gets terminate, it releases the space occupied. If the released free space is contiguous to another free partition, both the free partitions are merged together in to single free partition.
- Better utilization of memory than fixed sized size partition.

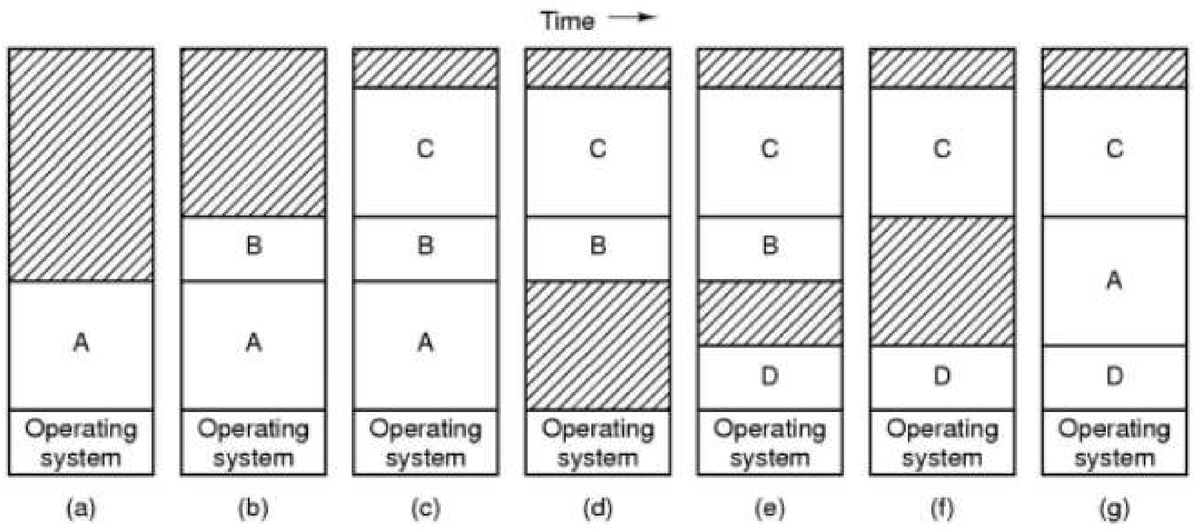


Figure : Variable Memory partitions Memory allocation changes as processes come into memory and leave it. The shaded regions are unused memory.

S.N.	Fixed Sized Partitioning	Variable Partitioning
1.	In multi-programming with fixed partitioning the main memory is divided into fixed sized partitions.	In multi-programming with variable partitioning the main memory is not divided into fixed sized partitions.
2.	Only one process can be placed in a partition.	In variable partitioning, the process is allocated a chunk of free memory.
3.	It does not utilize the main memory effectively.	It utilizes the main memory effectively.
4.	There is presence of internal fragmentation and external fragmentation.	There is external fragmentation.
5.	Degree of multi-programming is less.	Degree of multi-programming is higher.
6.	It is easier to implement.	It is less easy to implement.
7.	There is limitation on size of process.	There is no limitation on size of process.

.

Relocation and Protection

- In multiprogramming system, there are following two problems have to be solved to allow multiple applications to be in memory at the same time without interfering with each other.
  - **Relocation**
  - **Protection**

Relocation

When a program is run it does not know in advance what location it will be loaded at. Therefore, the program cannot simply generate static addresses (e.g. from jump instructions). Instead, they must be made relative to where the program has been loaded.

*Relocation is the ability to move process around in memory without affecting execution***.**

Types of relocation

- Static Relocation
- Dynamic Relocation

Protection

Once there are two programs in memory at the same time there is a danger that one program can write to the address space of another program. This is obviously dangerous and should be avoided.

Memory protection is a way to control access rights. The main purpose of memory protection is to prevent a process from accessing memory that has not been allocated.

- *Write Protection* - to prevent data & instructions from being over-written.
- *Read Protection* - To ensure privacy of data & instructions.
- OS needs to be protected from user processes, and user processes need to be protected from each other.
- Memory protection (to prevent memory overlaps) is usually supported by the hardware (limit registers), because most languages allow memory addresses to be computed at run-time.

Static and Dynamic Relocation

1. *Static Relocation* - Program must be relocated before or during loading of process into memory. Program must always be loaded into same address space in memory, or relocator must be run again.
2. *Dynamic Relocation* - Process can be freely moved around in memory. Virtual-to-physical address space mapping is done at run-time

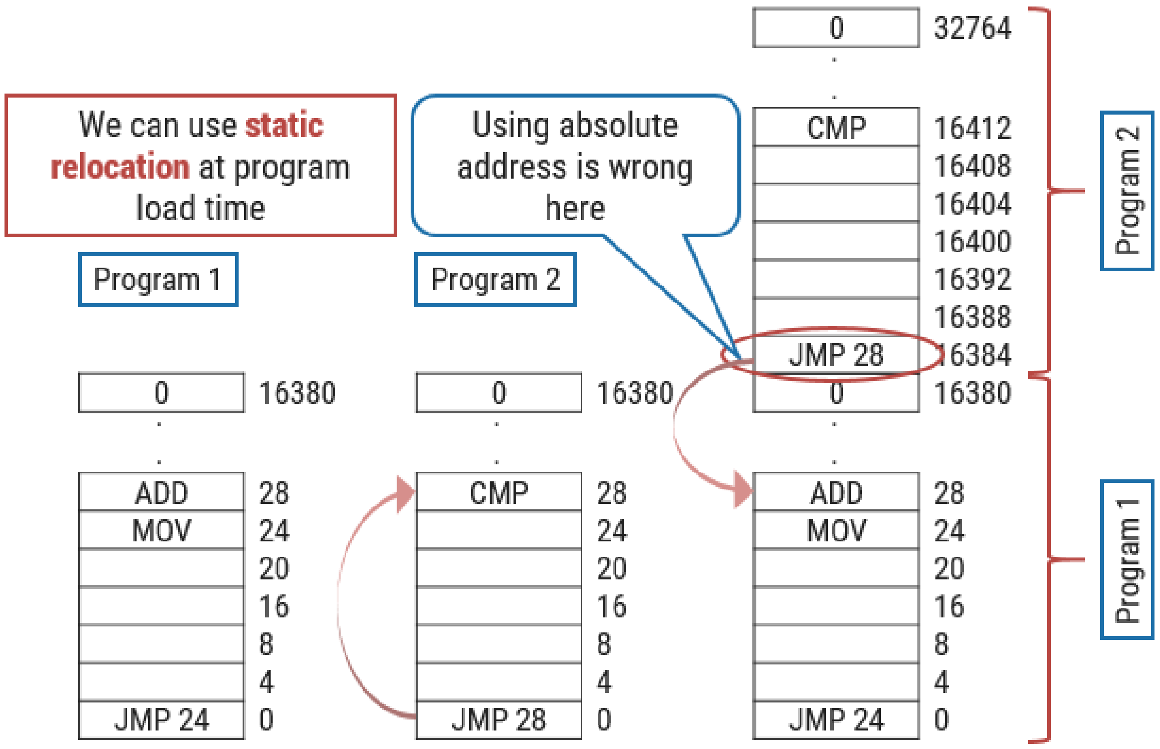


Figure: Multiprogramming without Memory abstraction

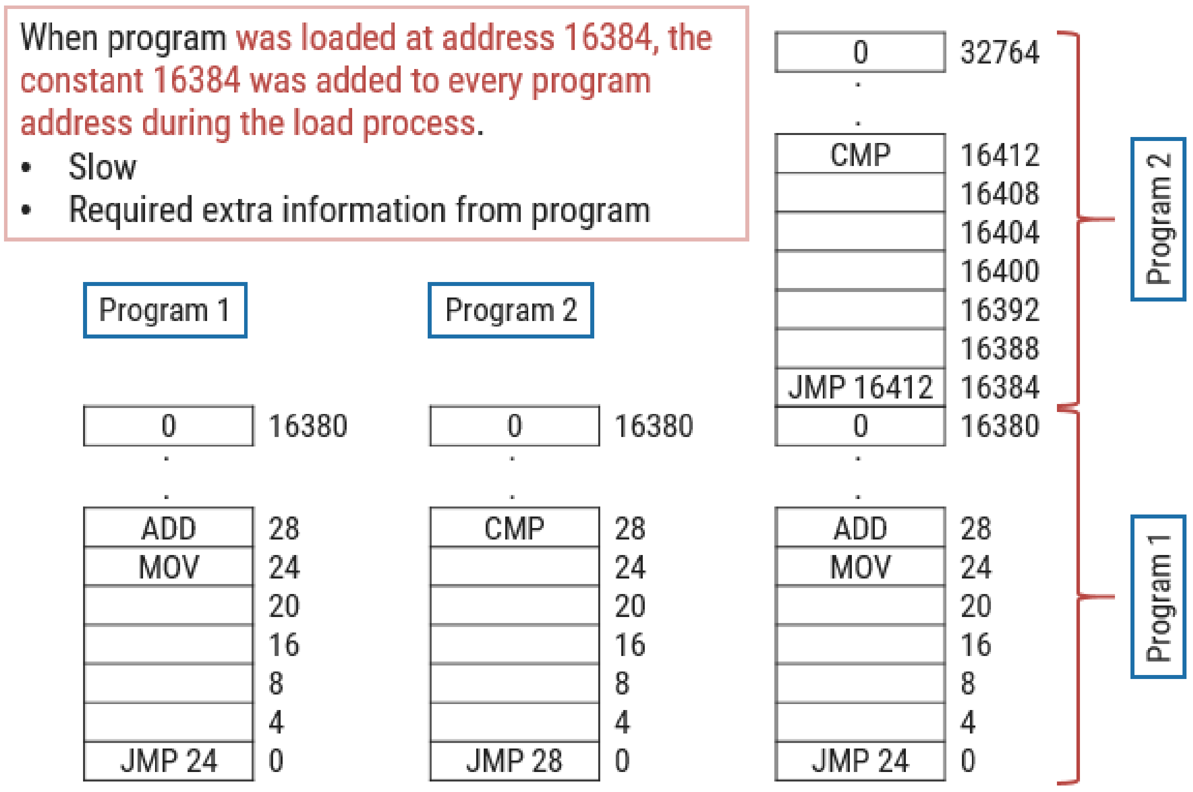

When program was loaded at address 16384, the constant 16384 was added to every program address during the load process.
• Slow
• Required extra information from program
Program 1
Program 2
0
16380
ADD
MOV
JMP 24
28
24
20
16
8
4
0
CMP
JMP 28
32764
16412
16408
16404
16400
16392
16388
JMP 16412
16384
Program 2
Program 1

Figure: Multiprogramming with Static Relocation

Address Space

- An **address space** is the set of addresses that a process can use to address memory. An address space is the range of valid addresses in memory that are available for program or process. Each process has its own address space.

Base and Limit Registers

- A solution, which solves both the relocation and protection problem is to equip the machine with two registers called **the base and limit registers**.
- The **base register** *stores the start address of the partition* and the **limit register** *holds the length of the partition*.
    - Any address that is generated by the program has the base register added to it.
    - In addition, all addresses are checked to ensure they are within the range of the partition.

- An additional benefit of this scheme is that if a program is moved within memory, only its base register needs to be amended. This is obviously a lot quicker than having to modify every address reference within the program.
- Only the OS can update base and limit registers.

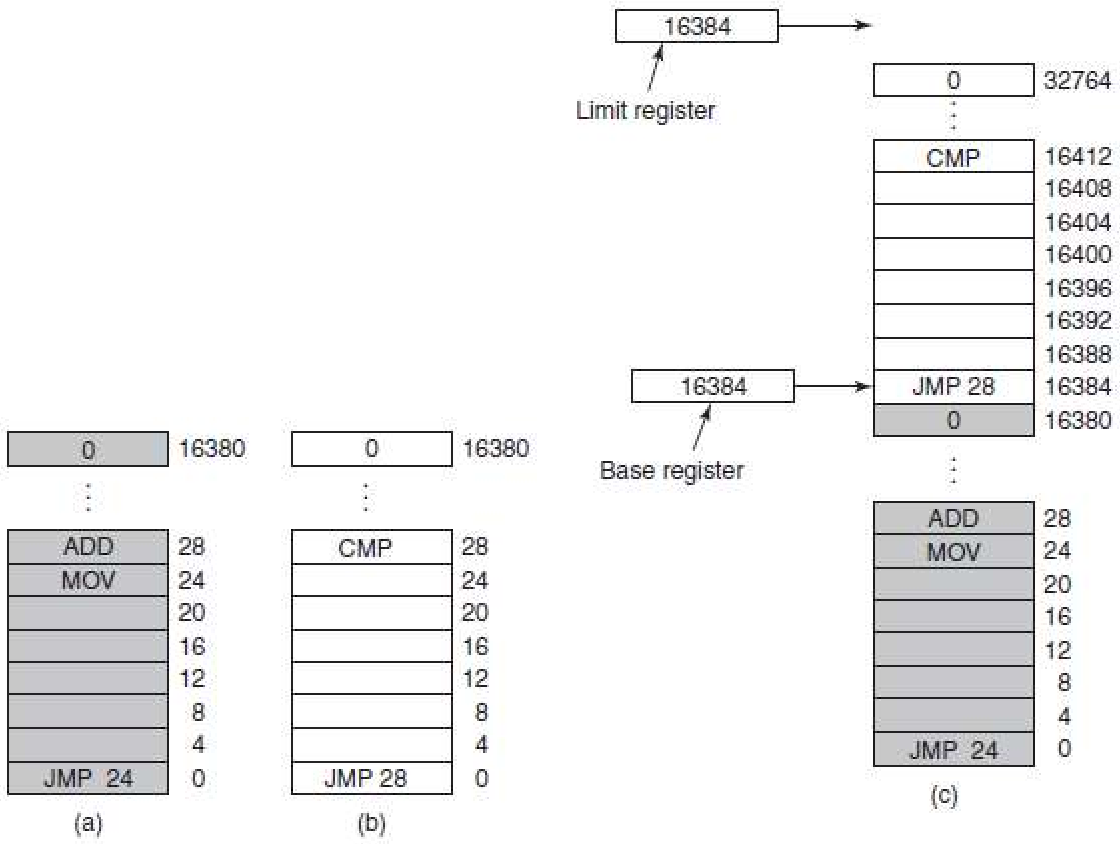
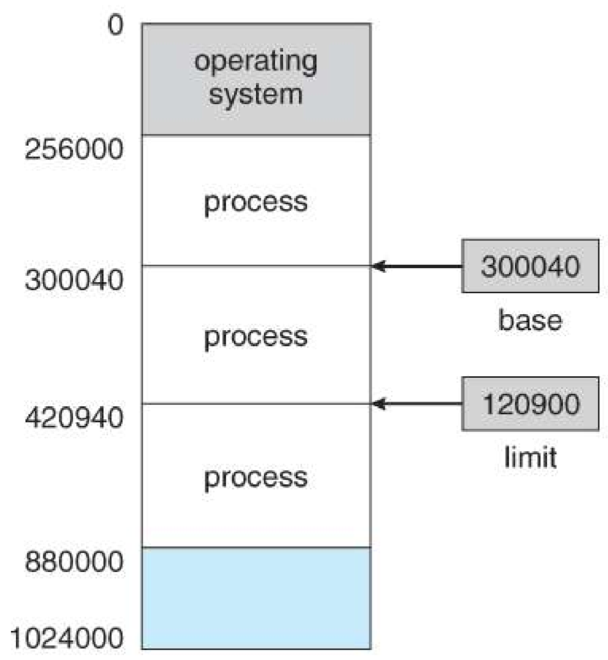


Figure: Base and Limit Registers

A base and a limit register define a logical address space



Steps in Dynamic Relocation

- Dynamic relocation refers to address transformations being done during execution of program
- Steps in dynamic relocation
  - Hardware adds relocation register (base) to virtual address to get a physical address
  - Hardware compare address with their register; address must be less than or equal limit
  - If test fails, the processor takes an address trap and ignores the physical address.

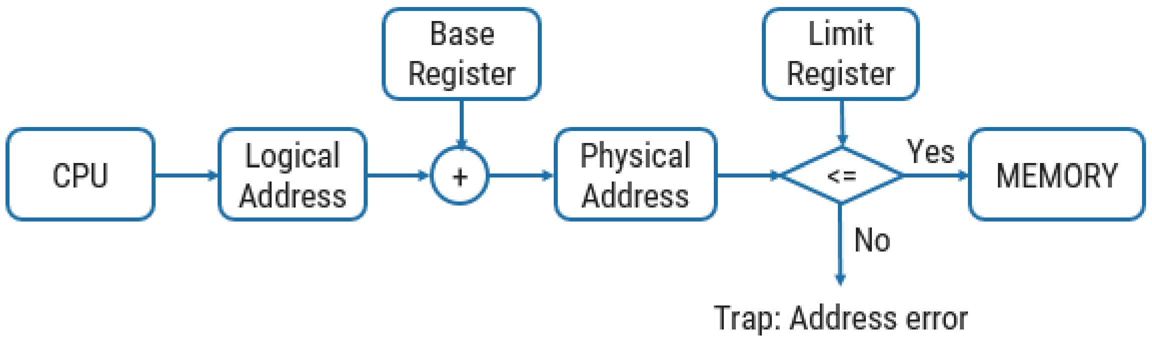


Figure: Steps in Dynamic Relocation

Logical Versus Physical Address Space

- The address generated by the CPU is a *logical address*, whereas the address actually seen by the memory hardware is a *physical address*.
- Addresses bound at compile time or load time have identical logical and physical addresses.
- Addresses created at execution time, however, have different logical and physical addresses.
- In this case the logical address is also known as a *virtual address*, and the two terms are used interchangeably.
- The set of all logical addresses used by a program composes the *logical address space*, and the set of all corresponding physical addresses composes the *physical address space*.
- The run time mapping of logical to physical addresses is handled by the *memory-management unit*, **MMU.**
- The MMU can take on many forms. One of the simplest is a modification of the base-register scheme described earlier.
- The base register is now termed a relocation register, whose value is added to every memory request at the hardware level.
- Note that user programs never see physical addresses. User programs work entirely in logical address space, and any memory references or manipulations are done using purely logical addresses. Only when the address gets sent to the physical memory chips is the physical memory address generated.

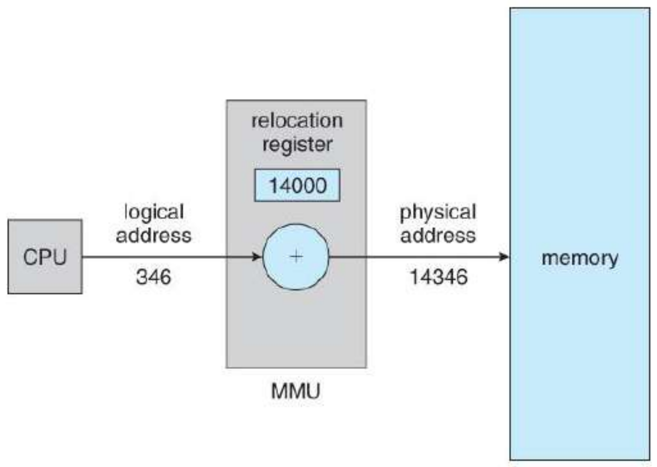


Figure: Dynamic relocation using a relocation register

Space Management

- Management of free spaces in memory.

Fragmentation

- As the process is loaded and unloaded from memory, these areas are fragmented into small pieces of memory that cannot be allocated to incoming processes. It is called fragmentation.
- There are mainly two types of fragmentation in the operating system. These are as follows:
  1. Internal Fragmentation
  2. External Fragmentation

Internal Fragmentation

- **Internal fragmentation** is memory that is allocated but not used. (The allocation method may require that processes sometimes get more memory than they need. For example, there may be a minimum allocation, or allocations may be made in chunks of a specific size.)
  - When a process is allocated to a memory block, and if the process is smaller than the amount of memory requested, a free space is created in the given memory block. Due to this, the free space of the memory block is unused, which causes internal fragmentation.

Example: Assume that memory allocation in RAM is done using fixed partitioning (i.e., memory blocks of **fixed sizes**). **2MB, 4MB, 4MB, and 8MB** are the available sizes. The Operating System uses a part of this RAM. Let's suppose a **process P1 with a size of 3MB** arrives and is given a memory block of 4MB. As a result, the 1MB of free space in this block is unused and cannot be used to allocate memory to another process ==> **internal fragmentation.**

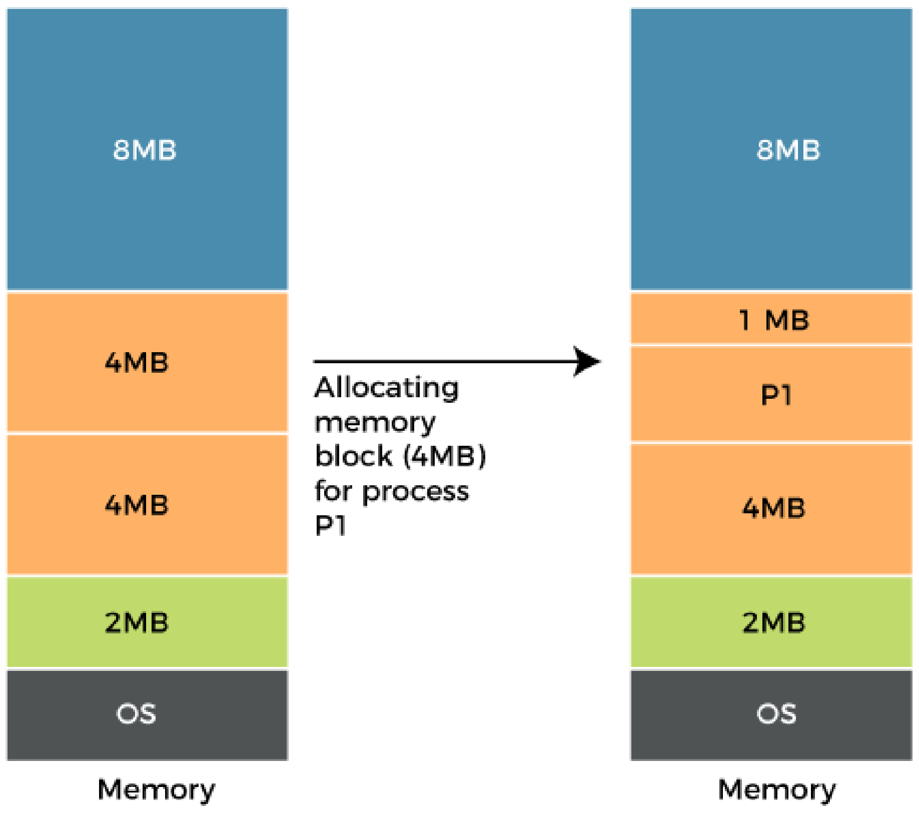


Figure: Internal Fragmentation

External Fragmentation

- **External fragmentation** is memory that is available but unusable. (Especially a collection of holes, each of which is too small to use for anything, but which would be enough to accommodate a process if it were possible to combine them together into one hole.)
- It occurs when the available space is not contiguous and the storage space is divided into a small number of holes
- External fragmentation happens when a dynamic memory allocation method allocates some memory but leaves a small amount of memory unusable. The quantity of available memory is substantially reduced if there is too much external fragmentation. There is enough memory space to complete a request, but it is not contiguous. It's known as external fragmentation.

Example: In the following diagram, we can see that there is sufficient space (50 KB) to run a process (P05) (need 45KB), but the memory is not contiguous ==> **external fragmentation.**

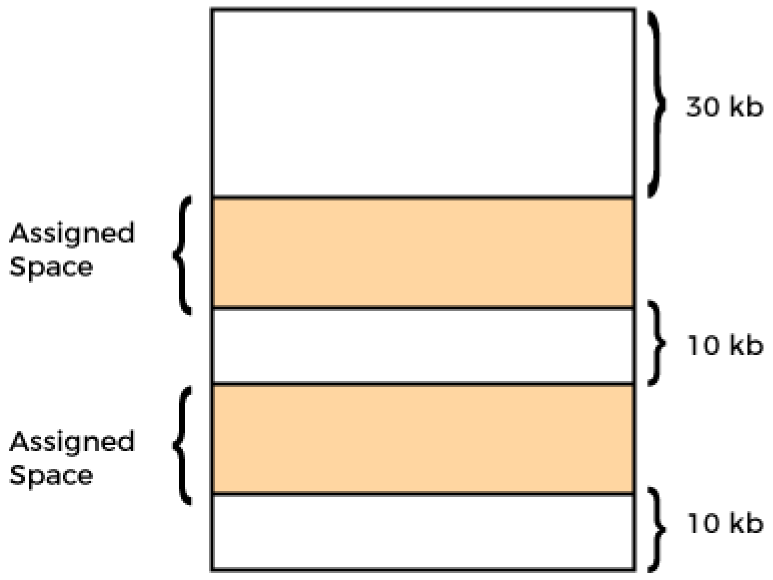


Figure: External Fragmentation

- **External Fragmentation** – total memory space exists to satisfy a request, but it is not contiguous
- **Internal Fragmentation** – allocated memory may be slightly larger than requested memory; this size difference is memory internal to a partition, but not being used

Memory Compaction

- 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.**
- **Compaction** is the technique by which the programs are relocated in such a way that the small chunks of free memory space are made contagious to eachother and clubbed together into a single partition that may be big enough to accommodate some more programs.

- It is usually not done because it requires a lot of CPU time.

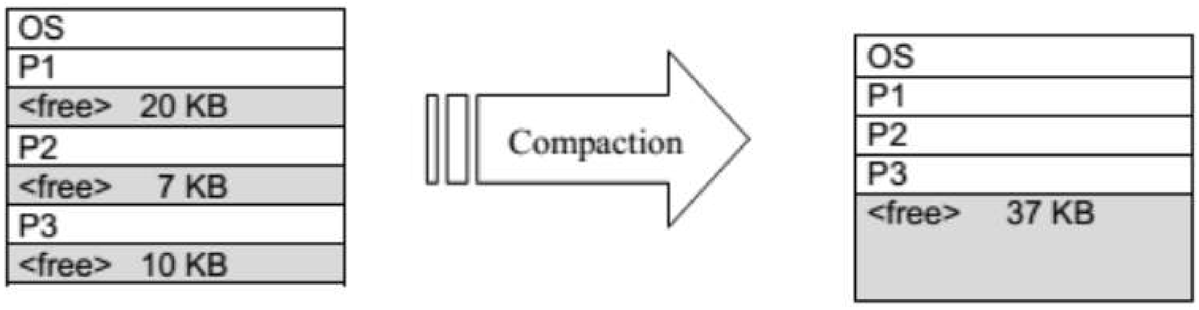


Figure: Memory Compaction

— Also called as defragmentation

Managing Free Memory

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 Linked lists.

Two ways to keep track of memory usage (free memory):-

1. Memory Management with Bitmaps
2. Memory Management with Linked Lists

Memory Management with Bitmaps

- With a bitmap, memory is divided into allocation units.
  - Allocation units can be as small as a few words and as large as several kilobytes
- Corresponding to each allocation unit is a bit in the bitmap.
- Bit is 0 if the unit is free and 1 if it is occupied (or vice versa).
- Following figure (a) and (b) shows part of memory and the corresponding bitmap

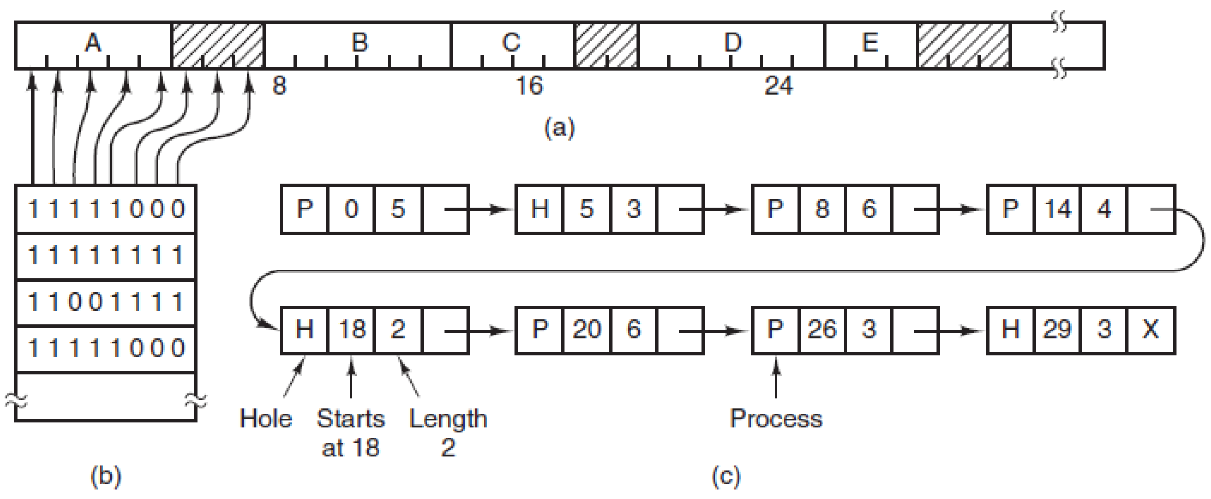


Figure: (a) A part of memory with five processes and three holes. The tick marks show the memory allocation units. The shaded regions (0 in the bitmap) are free. (b) The corresponding bitmap. (c) The same information as a list.

— 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 is that *when it has been decided to bring a k-unit process into memory, the memory 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 either contains a process or is an empty hole between two processes.
- 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 /entry.
- The memory of above figure (a) is represented in figure (c) as a linked list of segments.
- The segment list is kept sorted by address.

- Sorting this way has the advantage that when a process terminates or is swapped out, updating the list is straightforward.
- A terminating process normally has two neighbors (except when it is at the very top or bottom of memory)

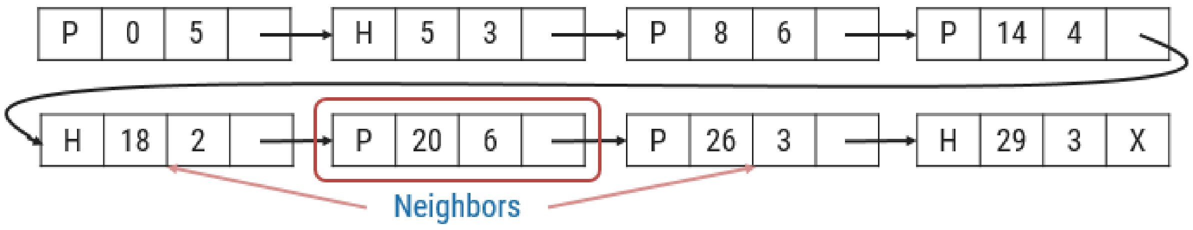


Figure: Showing the Neighbors of a process in Linked Representation

  - The neighbors may be either processes or holes, leading to the four combinations shown in following figure:

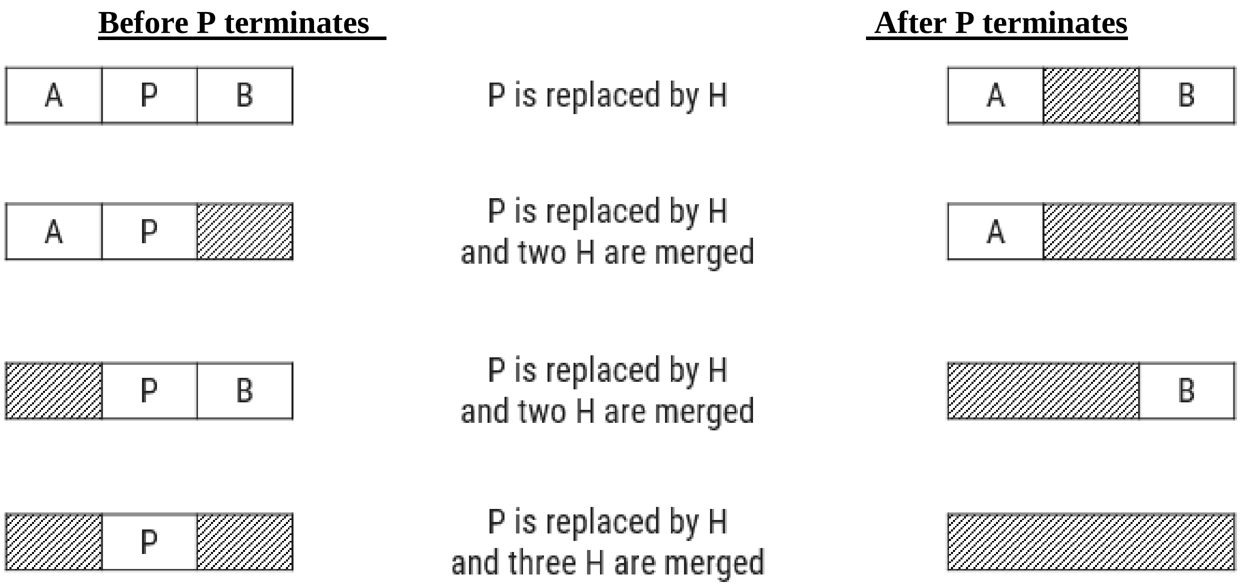

Before P terminates
After P terminates
A P B
P is replaced by H
A B
A P
P is replaced by H and two H are merged
A
P B
P is replaced by H and two H are merged
B
P
P is replaced by H and three H are merged

Figure: Four neighbor combinations for a terminating process, P

Memory Allocation Strategies

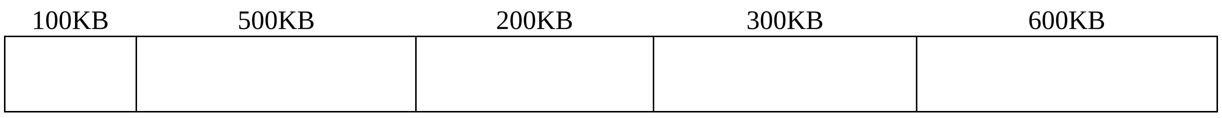
- When the processes and holes are kept on a list sorted by address, several algorithms can be used to allocate memory for a created process (or an existing process being swapped in from disk). We assume that the memory manager knows how much memory to allocate.
- When we need to allocate memory, storing the list in segment address order allows us to implement various strategies.
- Some of the memory allocation strategies are as follows:
    1. **First fit**
    2. **Next fit**
    3. **Best fit**
    4. **Worst fit**
    5. **First fit**

First Fit

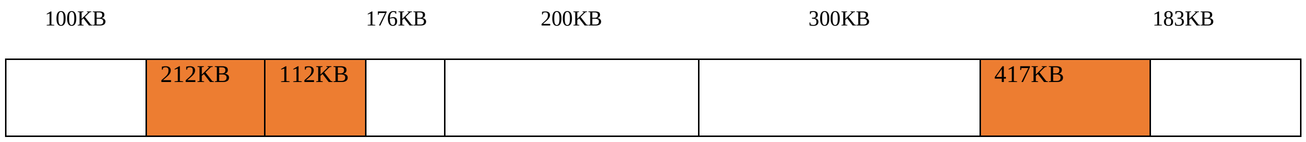
- The simplest algorithm is **first fit**. The memory manager scans along the list of segments until it finds a hole that is big enough.
- Search starts from the starting location of the memory.
- First Available hole that is large enough to hold the process is selected for allocation.
- The hole is then broken up into *two pieces*, one *for process* and another *for unused memory (*except in the statistically unlikely case of an exact fit).

Example:

Suppose processes of 212KB, 417KB, 112KB and 426KB arrives in order and the memory is partitioned with fixed partition as follows.



After following First fit strategy the memory is allocated as follows:



Here the process of size 426 will not get any partition

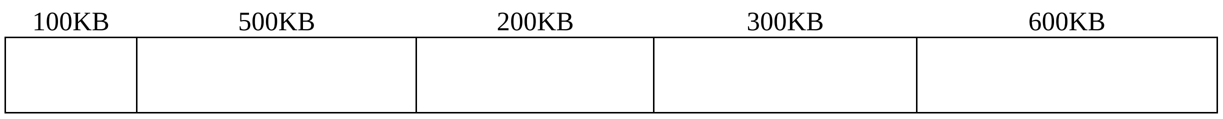
- ➢ First fit is a fast algorithm because it searches as little as possible
- ➢ In this strategy the memory loss is higher
  - — Very large hole may be selected for small process.

Next Fit

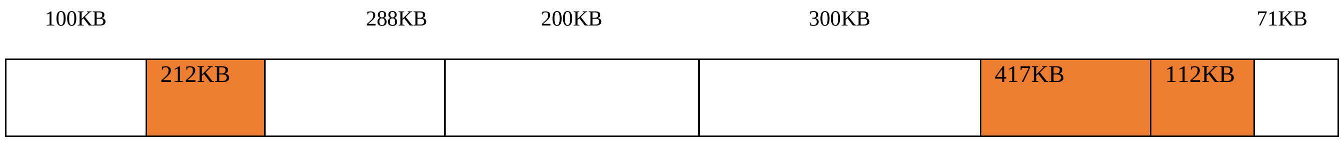
- — A minor variation of first fit is **next fit**.
- — It works the same way as first fit, except that *it keeps track of where it is whenever it finds a suitable hole.*
- — The *next time it is called to find a hole, it starts searching the list from the place where it left off last time,* instead of always at the beginning, as first fit does.

Example:

Suppose processes of 212KB, 417KB, 112KB and 426KB arrives in order and the memory is partitioned with fixed partition as follows:



After following Next fit strategy the memory is allocated as follows:



Here the process of size 426 will not get any partition for allocation

- — Search time is less in this strategy.
- — Memory manager must keep track of last allotted hole to process.
- — It give slightly worse performance than First fit strategy

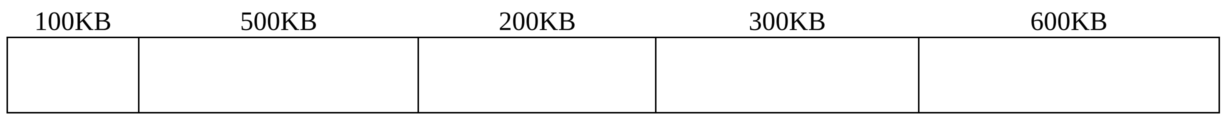
Best Fit

- — Another well-known and widely used algorithm is **best fit**.
- — Best fit searches the entire memory.
- — From beginning to end, the smallest hole that is large enough to hold the process is selected for allocation.

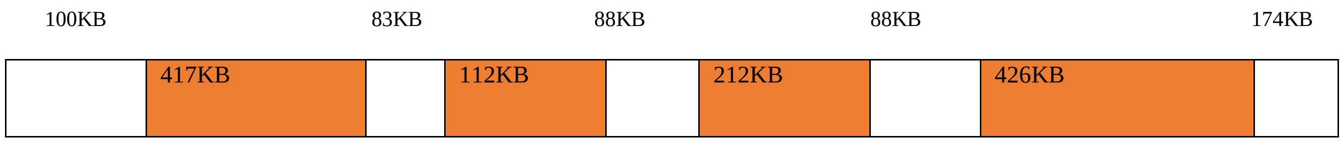
- Rather than breaking up a big hole that might be needed later, best fit tries to find a hole that is close to the actual size needed, to best match the request and the available holes.

Example:

Suppose processes of 212KB, 417KB, 112KB and 426KB arrives in order and the memory is partitioned with fixed partition as follows:



After following Best fit strategy the memory is allocated as follows:



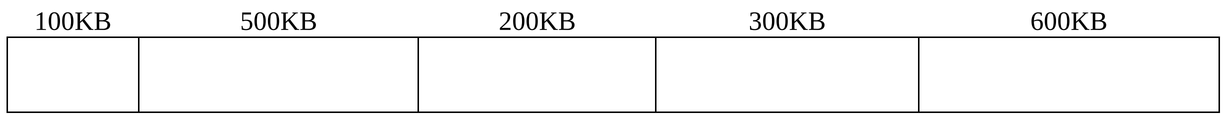
- In this technique, search time is high because the entire memory is searched every time
- Best fit is slower than first fit because it must search the entire list every time it is called
- Somewhat surprisingly, it also results in more wasted memory than first fit or next fit because it tends to fill up memory with tiny, useless holes.

Worst Fit

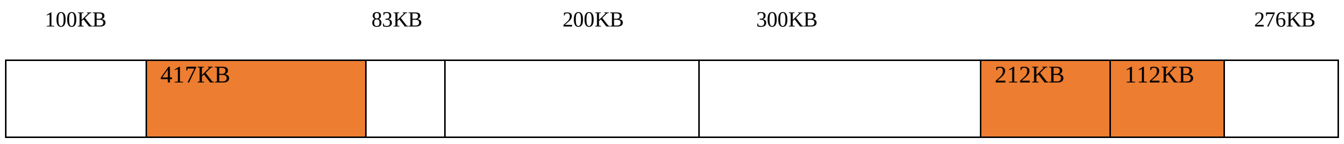
- In worst fit, the largest available hole is always taken, so that the new hole will be big enough to be useful
- Worst fit searches the entire memory.
- The largest hole that is large enough to hold the process is selected for allocation.

Example:

Suppose processes of 212KB, 417KB, 112KB and 426KB arrives in order and the memory is partitioned with fixed partition as follows:



After following Worst fit strategy the memory is allocated as follows:



- Here the process of size 426 will not get any partition for allocation

— In this technique, search time is high because the entire memory is searched every time

— If a process requiring larger memory arrives at later stage then it cannot be accomodated as the larger hole is already split and occupied.

Virtual Memory

- Virtual memory is an area of a computer system's secondary memory storage space (such as a hard disk or solid state drive) which acts as if it were a part of the system's RAM or primary memory.
- **Virtual memory** is a memory management technique which allows to execute programs that may not fit entirely in the system memory. It is implemented with the help of secondary storage devices by transferring data from main memory to secondary memory and vice-versa.
- Computer has finite amount of main memory (RAM), It is possible to run out of memory when too many programs running at one time. In this situation virtual memory come into picture.
- Ideally, the data and instructions needed to run applications is stored in RAM, where they can be accessed quickly by the CPU. But when large applications (programs) are being run, or when many applications (programs) are running at once, the system's RAM may become full (insufficient).
- Virtual memory enables a system to run larger programs or run more programs at the same time without running out of RAM. Specifically, the system can operate as if its total RAM resources were equal to the amount of physical RAM, plus the amount of virtual RAM.

- Virtual memory serves two purposes. First, it allows us to extend the use of physical memory by using disk. Second, it allows us to have memory protection, because each virtual address is translated to a physical address.

The basic idea behind virtual memory is that each program has its own address space, which is broken up into chunks called **pages**.

Each page is a contiguous range of addresses.

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.

The OS keeps those parts of the program currently in use in main memory, and rest on the disk.

When the program references a part of its address space that is in physical memory, the hardware performs the necessary mapping on the fly.

In a system using virtual memory, the physical memory is divided into page frames and the virtual address space is divided into equally sized partitions called pages.

When the program references a part of its address space that is not in physical memory, the operating system is alerted to go get the missing piece and re-execute the instruction that failed.

Virtual memory works just fine in a multiprogramming system, with bits and pieces of many programs in memory at once. While a program is waiting for pieces of itself to be read in, the CPU can be given to another process

Following are the situations, when entire program is not required to be loaded fully in main memory.

- User written error handling routines are used only when an error occurred in the data or computation.
- Certain options and features of a program may be used rarely.
- Many tables are assigned a fixed amount of address space even though only a small amount of the table is actually used.
- The ability to execute a program that is only partially in memory would counter many benefits.
- Less number of I/O would be needed to load or swap each user program into memory.
- A program would no longer be constrained by the amount of physical memory that is available.
- Each user program could take less physical memory, more programs could be run the same time, with a corresponding increase in CPU utilization and throughput

NOTES:

- Since RAM is more expensive than virtual memory, it would seem – all things being equal – that computers should be equipped with as little RAM and as much virtual memory as possible.
- But in fact the characteristics of virtual memory are different than those of physical memory. The key difference between virtual memory and physical memory is that RAM is very much faster than virtual memory.
- So a system with 2 GB of physical RAM and 2 GB of virtual memory will not offer the same performance as a similar system with 4 GB of physical RAM.

- *Virtual memory can be implemented in a number of different ways by operating system, and the two most common approaches are **paging** and **segmentation**.*

Advantages of Virtual Memory

- Allows more applications to be run at the same time.
- Allows larger applications to run in systems that do not have enough physical RAM alone to run them.
- Provides a way to increase memory which is less costly than buying more RAM.
- Provides a way to increase memory in a system which has the maximum amount of RAM that its hardware and operating system can support.

Disadvantages of Virtual Memory

- Does not offer the same performance as RAM.
- Can negatively affect the overall performance of a system.
- Takes up storage space which could otherwise be used for long term data storage.

Paging

- The virtual address space consists of fixed-size units called **pages**.
- **Paging** is a mechanism used to retrieve process from the secondary storage (disk) into the main memory (RAM) in the form of pages.
- Paging is a non-contiguous memory allocation technique.

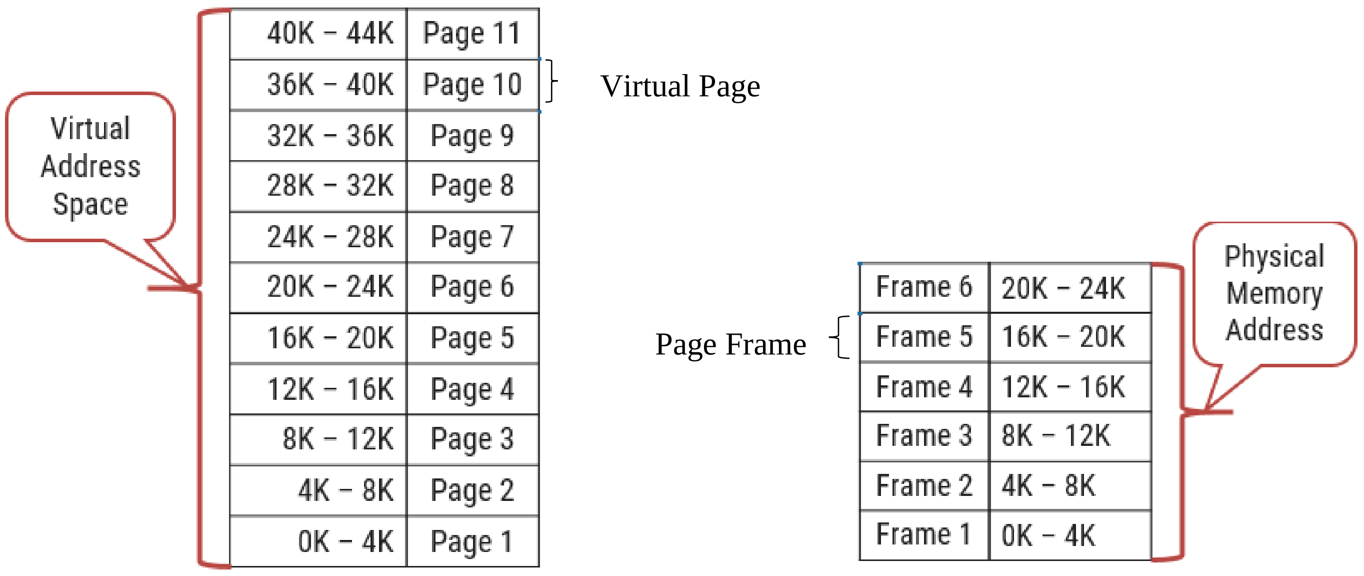


Figure: Virtual addresses space and Physical memory address (Page and Frame Size is considered 4KB)

- The main idea behind paging is to divide each process in the form of pages. The main memory is also divided in the form of frames. One page of the process is to be stored in one of the frames of the memory.
  - Considering the frame size of 4 KB, 44 KB of virtual address space and 24 KB of physical memory, we get 11 virtual pages and 6 page frames (shown in above figure).
  - In above figure, the range marked 0K–4K means that the virtual or physical addresses in that page are 0 to 4095. Every page begins on a multiple of 4096 (since 4KB = 4096 Bytes) and ends 4095 addresses higher, so 4K–8K really means 4096–8191 and 8K to 12K means 8192 – 12287.
- The page can be stored at the different locations of the memory but the priority is always to find contiguous frames or holes.
- Pages of the process are brought into the main memory only when they are required otherwise they reside in secondary storage.
- The size of the each frame must be equal. Considering the fact that the pages are mapped to the frames in Paging, page size needs to be as same as frame size
- Different operating system defines different frame sizes.

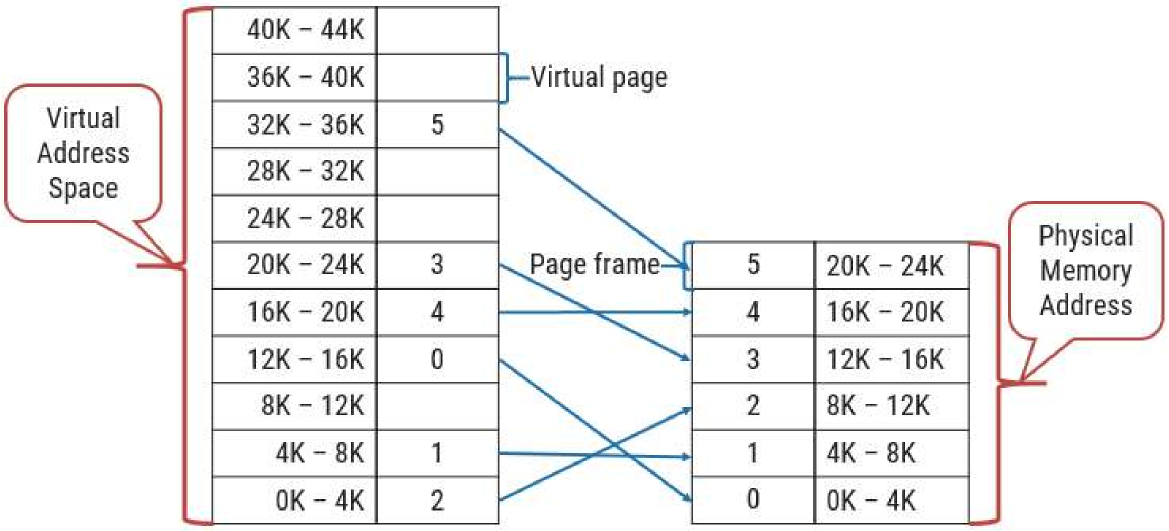


Figure: The relation between virtual addresses and physical memory addresses

- The relation between virtual addresses and physical memory addresses is given by the **page table**. Page table keeps the track of pages and page frames.

NOTES:

- **Logical Address** is generated by CPU while program is running. The logical address is virtual address as it does not exist physically therefore it is also known as **Virtual Address**
- **Physical Address** identifies a physical location of required data in memory.
- The hardware device called **Memory Management Unit (MMU)** is used for *mapping (converting) logical address to its physical address*.

⇒ ***When a virtual memory is used, the virtual address is presented to an MMU that maps the virtual addresses to the physical memory addresses.***

- In above figure, only 24 KB of physical memory is available and 44KB program cannot be loaded into memory in their entirety and run. So, the program is kept in virtual memory.
    - Only the required pages are loaded into physical memory
    - Transfers between RAM and disk are always in units of a page.

Internal operation of MMU

⇒ The virtual address is split into a virtual page number (high-order bits) and an offset (low-order bits).

⇒ For example, with a 16-bit address and a 4-KB page size, the upper 4 bits could specify one of the 16 virtual pages and the lower 12 bits would then specify the byte offset (0 to 4095) within the selected page.

⇒ The page number is used as an index into the page table, yielding the number of the page frame corresponding to that virtual page.

⇒ If the Present/absent bit is 0, it is page fault; trap to OS is caused to bring required page into main memory

⇒ If the bit is 1, the page frame number found in the page table is copied to the high-order bits (3 bits in this example) of the output register, along with the offset (12 bits in this example). Together they form a physical address (15 bits in this example). The output register is then put onto the memory bus as the physical memory address.

- *Together page frame number and offset creates physical address*
- **Physical Address = Page Frame Number + Offset of Virtual Address**

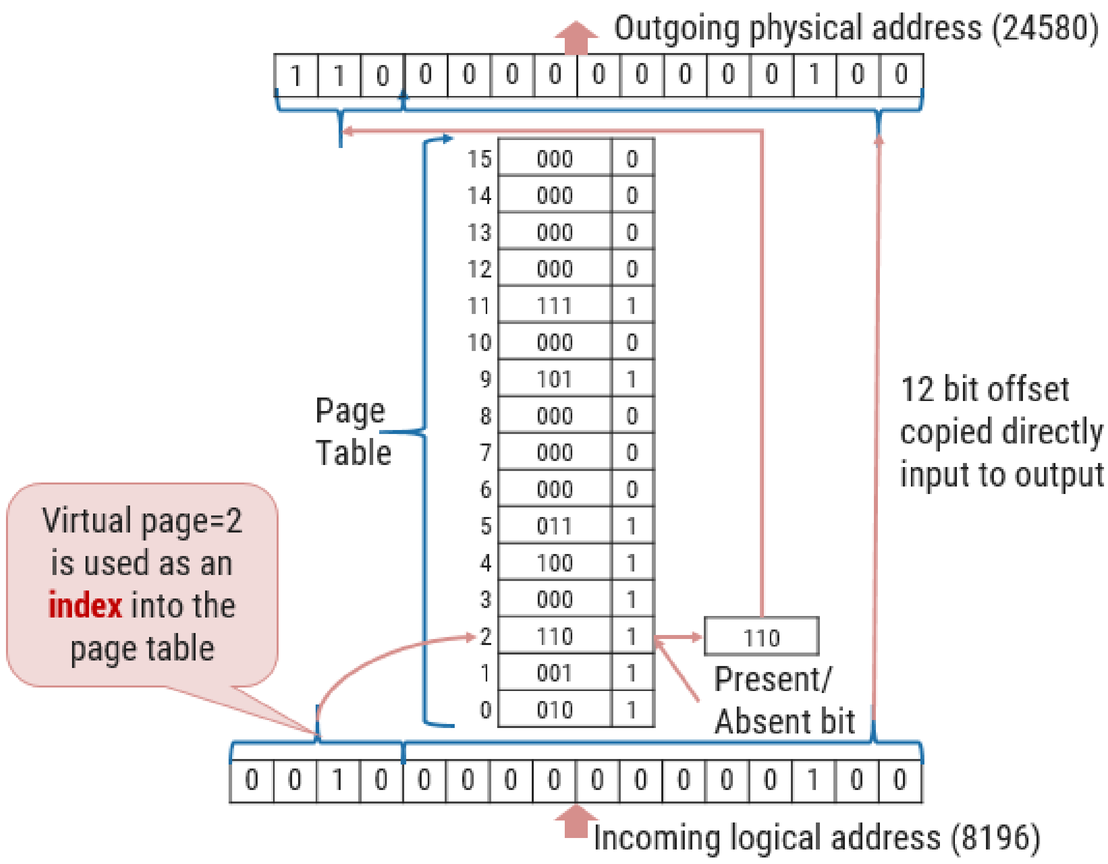


Figure: The internal operation of the MMU with 16 4-KB pages

❖ NOTE: When the program references an unmapped address the MMU causes the CPU to trap to the operating system. This trap is called a **page fault**.

Page Tables

- ☑ Page table is a **data structure** which performs the mapping of page number to the frame number (translates virtual address into equivalent physical address).
- ☑ The virtual page number is used as an index into the Page table to find the entry for that virtual page and from the Page table physical page frame number is found.
- ☑ Thus the purpose of page table is to map virtual pages onto page frames.
- ☑ The page table is a function, with the virtual page number as argument and the physical frame number as result
- ☑ The page table of the process is held in the kernel space.
- ☑ Each Process has a separate page table

- The virtual address is split into a virtual page number (high-order bits) and an offset (low-order bits).
- From the page table entry, the page frame number (if any) is found.
- The page frame number is attached to the high-order end of the offset, replacing the virtual page number, to form a physical address that can be sent to the memory.
- Thus, the purpose of the page table is to map virtual pages onto page frames.
- Mathematically speaking, the page table is a function, with the virtual page number as argument and the physical frame number as result. Using the result of this function, the virtual page field in a virtual address can be replaced by a page frame field, thus forming a physical memory address.

Structure of a Page Table Entry

The exact layout of an entry in the page table is highly machine dependent, but the kind of information present is roughly the same from machine to machine. Following figure represent a sample page table entry.

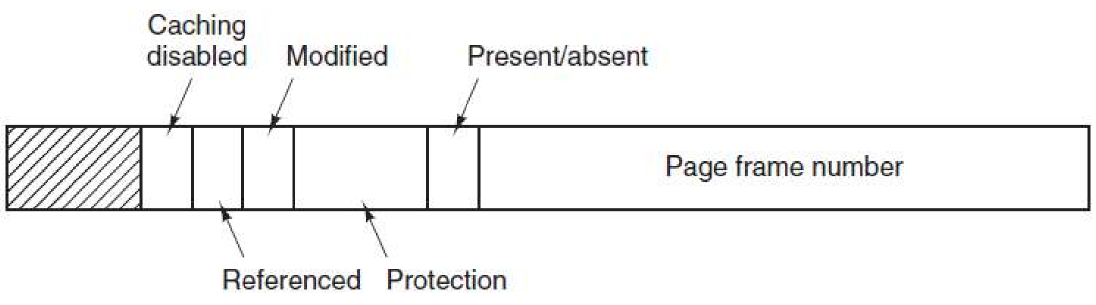
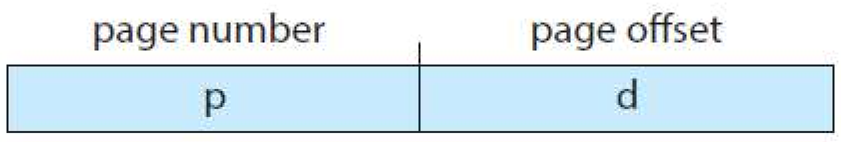


Figure: A typical page table entry.

- **Page frame Number:** It gives the frame number in which the current page you are looking for is present.
- **Present/Absent bit:** Present or absent bit says whether a particular page you are looking for is present or absent. If it is not present, that is called Page Fault. It is set to 0 if the corresponding page is not in memory. Sometimes this bit is also known as valid/invalid bits.
- **Protection bits:** Protection bit says that what kind of protection you want on that page. In the simplest form, 0 for read/write and 1 for read only.
- **Modified bit:** Modified bit says whether the page has been modified or not. If the page in memory has been modified, it must be written back to disk. This bit is also called as dirty bit as it reflects the page's state.
- **Referenced bit:** A references bit is set whenever a page is referenced, either for reading or writing. Its value helps operating system in page replacement algorithm.
- **Cashing Disabled bit:** This feature is important for pages that maps onto device registers rather than memory. With this bit cashing can be turned off.

Basic idea about Paging Technique

- Every address generated by the CPU is divided into two parts: a page number (p) and a page offset (d):



- The page number is used as an index into a page table.
- The base address of the frame is combined with the page offset to define the physical memory address

- The following outlines the steps taken by the MMU to translate a logical address generated by the CPU to a physical address:
  - Extract the page number p and use it as an index into the page table.
  - Extract the corresponding frame number f from the page table.
  - Replace the page number p in the logical address with the frame number f.

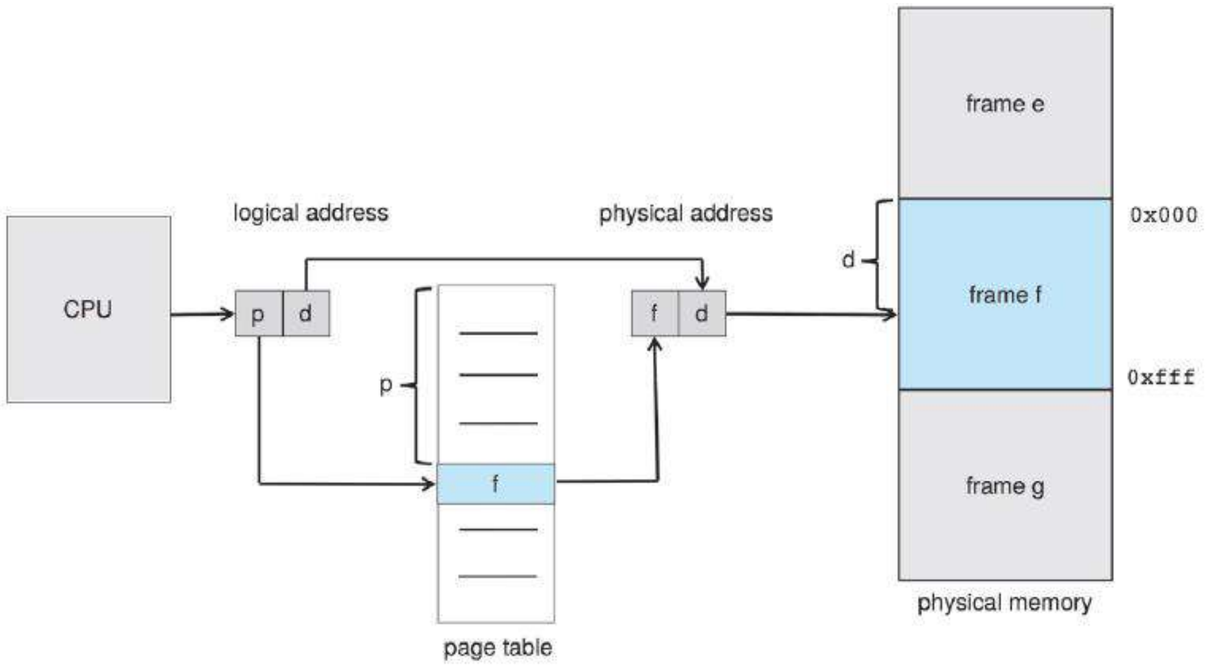
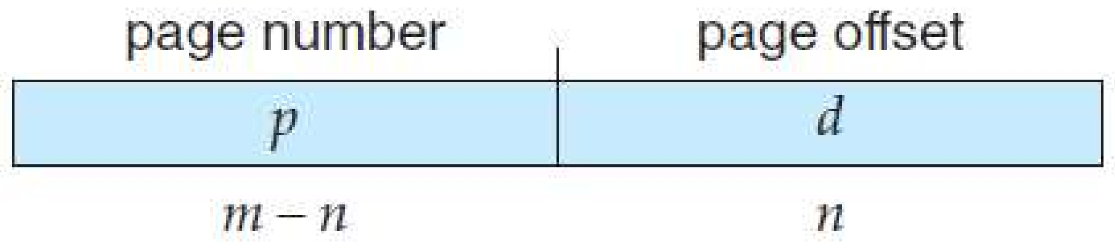


Figure : Paging hardware

- If the size of the logical address space is 2^m, and a page size is 2^n bytes, then the high-order (m−n) bits of a logical address designate the page number, and the n low-order bits designate the page offset. Thus, the logical address is as follows:



NOTES

- In general, if the given address consists of 'n' bits, then using 'n' bits, 2^n locations are possible.
- Then, size of memory = 2^n x Size of one location.
- If the memory is byte-addressable, then size of one location = 1 byte.
  - Thus, size of memory = 2^n bytes.
- If the memory is word-addressable where 1 word = m bytes, then size of one location = m bytes.
  - Thus, size of memory = 2^n x m bytes.

Important Formulas

The following list of formulas is very useful for solving the numerical problems based on paging.

For Main Memory

- Physical Address Space = Size of main memory
- Size of main memory = Total number of frames x Page size
- Frame size = Page size
- If number of frames in main memory = 2^n, then number of bits in frame number = n bits
- If Page size = 2^n Bytes, then number of bits in page offset = n bits
- If size of main memory = 2^n Bytes, then number of bits in physical address = n bits

For Process

- Virtual Address Space = Size of process
- Number of pages the process is divided = Process size / Page size
- If process size = 2^n bytes, then number of bits in virtual address space = n bits

For Page Table

- Size of page table = Number of entries in page table x Page table entry size
- Number of entries in pages table = Number of pages the process is divided
- Page table entry size = Number of bits in frame number + Number of bits used for optional fields if any

Handling Page Faults

Page Fault is the condition in which a running process refers to a page that is not loaded in the main memory.

- The present/absent bit of page-table entry for a page that is brought into memory is set as 1 but the page-table entry for a page that is not currently in memory is simply set 0 (or marked invalid with 'i' as in following figure)
- If the process tries to access a page that was not brought into memory, a **page fault** is occurred. The paging hardware, in translating the address through the page table, will notice that the present/ absent bit is 0, causing a trap to the operating system.
- If a page is needed that was not originally loaded up, then a page fault trap is generated, which must be handled in following sequence of steps:
    1. An internal table (usually kept with the process control block) is checked for this process to determine whether the reference was a valid or an invalid memory access.
    2. If the reference was invalid, the process is terminated. If it was valid but it has not been brought in that page, the page is brought into the memory.
    3. A free frame is found (by taking one from the free-frame list, for example).
    4. A secondary storage operation is scheduled to read the desired page into the newly allocated frame.
    5. When the storage read is complete, the internal table kept with the process and the page table are modified to indicate that the page is now in memory.

6. The instruction that was interrupted by the trap is restarted. The process can now access the page as though it had always been in memory.

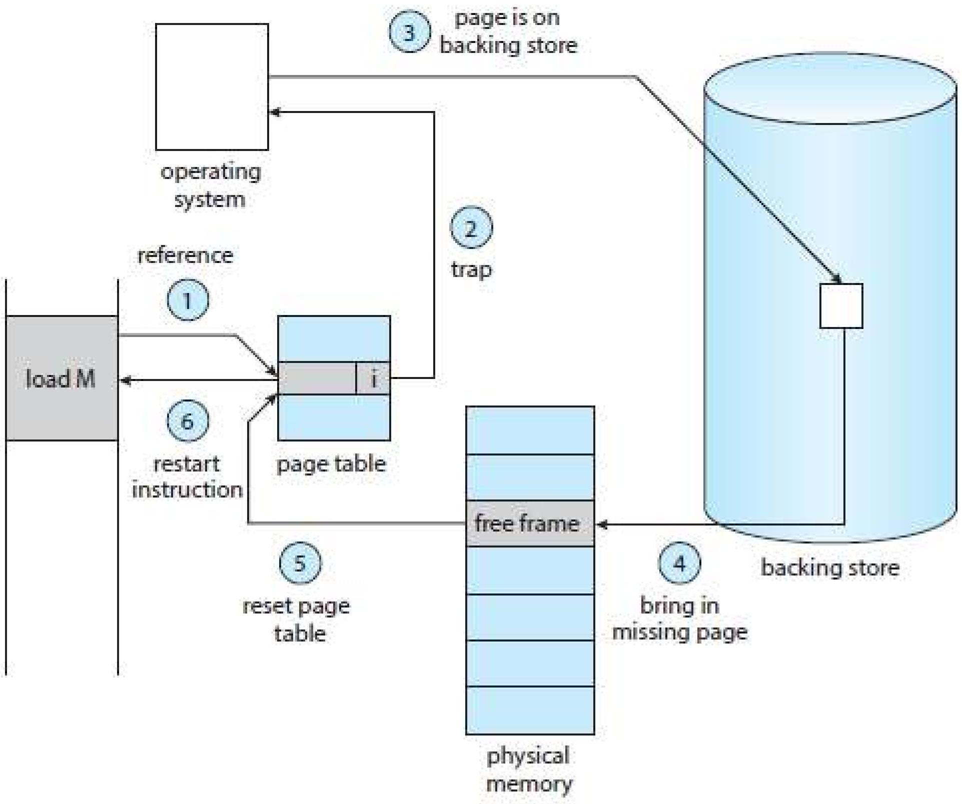


Figure: Steps in handling Page Fault

Issues in Paging

- The mapping from virtual address to physical address must be fast.
- The virtual address space is large, the page table will be large.

Translation Lookaside Buffer (TLB)

- One major disadvantage of paging is increase in the effective access time due to increased number of memory accesses. One memory access is required to get the frame number from the page table. Another memory access is required to get the word from the page.
  - Two memory accesses are needed to access data (one for the page-table entry and one for the actual data).
  - Thus, memory access is slowed by a factor of 2, a delay that is considered intolerable under most circumstances
- The standard solution to this problem is TLB
- Translation Lookaside Buffer (TLB) is a special, small, fast-lookup hardware cache.
- The TLB is associative, high-speed memory.
- Each entry in the TLB Each entry in the TLB consists of two parts: a key (or tag) and a value.
- TLB contains page table entries that have been most recently used.
  - Given a virtual address, the processor examines the TLB if a page table entry is present (TLB hit), the frame number is retrieved and the real address is formed.
  - If a page table entry is not found in the TLB (TLB miss), the page number is used as index while processing page table. TLB first checks if the page is already in main memory, if not in main memory a page fault is issued then the TLB is updated to include the new page entry.
- Steps in TLB hit:
  - CPU generates virtual (logical) address.
  - It is checked in TLB (present).
  - Corresponding frame number is retrieved, which now tells where in the main memory page lies.

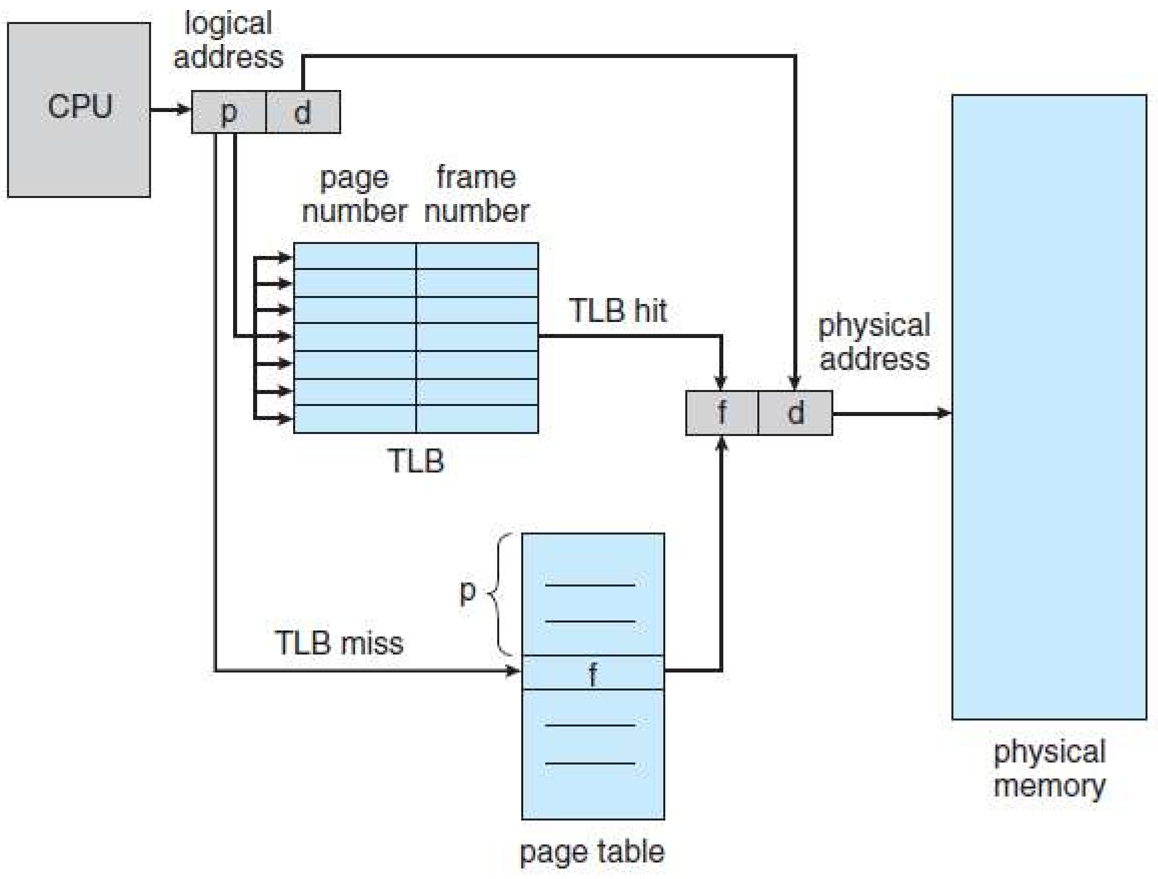


Figure : Paging hardware with TLB

- <u>Steps in TLB miss:</u>
    - CPU generates virtual (logical) address.
    - It is checked in TLB (not present).
    - Now the page number is matched to page table residing in main memory (assuming page table contains all PTE).
    - Corresponding frame number is retrieved, which now tells where in the main memory page lies.
    - The TLB is updated with new PTE (if space is not there, one of the replacement technique is requested into picture i.e either FIFO, LRU or MFU etc)

- **Effective Memory Access Time (EMAT)** : TLB is used to reduce effective memory access time as it is a high speed associative cache.

  EMAT = h(c+m) + (1-h)*(c+2m)*

  where, h = hit ratio of TLB

  m = Memory access time

  c = TLB access time

Some Definitions

- **Demand paging:** Demand paging is a technique used in virtual memory systems where the pages are brought in the main memory only when required or demanded by the CPU.
- **Pre-paging:** Many paging systems try to keep track of each process's working set and make sure that it is in memory before letting the process run. Loading pages before allowing processes to run is called pre-paging.
- **Working Set:** The set of pages that a process is currently using is known as working set.
- **Thrashing:** Thrashing is a condition or a situation when the system is spending a major portion of its time in servicing the page faults, but the actual processing done is very negligible.

Page Replacement Algorithms

- Page Replacement Algorithm decides which page to remove, also called swap out when a new page needs to be loaded into the main memory.
- Page Replacement happens when a requested page is not present in the main memory and the available space is not sufficient for allocation to the requested page.
- While it would be possible to pick a random page to evict at each page fault, system performance is much better if a page that is not heavily used is chosen.
    - If a heavily used page is removed, it will probably have to be brought back in quickly, resulting in extra overhead.

- A page fault occurs when a page referenced by the CPU is not found in the main memory.
- The required page has to be brought from the secondary memory into the main memory.
- A page has to be replaced if all the frames of main memory are already occupied.
- **Page replacement** is a process of swapping out an existing page from the frame of a main memory and replacing it with the required page.
- Page replacement is required when-
    - All the frames of main memory are already occupied.
    - Thus, a page has to be replaced to create a room for the required page.
- Some of the page replacements Algorithms:
    - FIFO Page Replacement Algorithm
    - Second Chance Page Replacement Algorithm
    - Optimal Page Replacement Algorithm
    - LFU Page Replacement Algorithm
    - LRU Page Replacement Algorithm
    - Clock Page Replacement Algorithm
    - WS -Clock Page Replacement Algorithm

A good page replacement algorithm is one that minimizes the number of page faults

Hit Rate and Miss Rate

- 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**
    - ***Hit Ratio = Total number of page hits / Total number of references***
    - ***Miss Ratio = Total number of page misses / Total number of references***

      OR

      Miss Ratio = 1 – Hit Ratio

Concept of Locality of Reference

- Experiments show that some pages are accessed more frequently than others.
- **Locality of Reference**, also known as the **principle of locality**, is the *tendency of a processor to access the same set of memory locations repetitively over a short period of time.*
- The property of Locality of Reference is mainly shown by loops and subroutine calls in a program.
- There are two basic types of locality of reference – **tempora**l and **spatial.**

  Temporal locality
    - Related with time
    - *Recently referenced items are likely to reference in the near future.*
    - If at one point a particular memory location is referenced, then it is likely that the same location will be referenced again in the near future.
    - Recently accessed data/ instruction are likely to be accessed again
    - Example:
        - Reference of certain variable in each iteration (data)
        - Cycle through loops (instruction)

  Spatial locality
    - Related with space
    - *Items with nearby addresses tend to be referenced close together in the near future.*
    - If a particular storage location is referenced at a particular time, then it is likely that nearby memory locations will be referenced in the near future.
    - If memory location (address) n is accessed, the address close to n is likely to be accessed (usually $n \pm \epsilon$)

- Example:
  - Reference of array elements in succession (data)
  - Reference of instructions in sequence (instruction)

First in First out Page (FIFO) Replacement Algorithm

☞ This is the simplest page replacement algorithm.

☞ In this algorithm, the OS maintains a queue that keeps track of all the pages in memory, with the oldest page at the front and the most recent page at the back.

☞ When there is a need for page replacement, the FIFO algorithm, swaps out the page at the front of the queue, that is the page which has been in the memory for the longest time.

- As the name suggests, this algorithm works on the principle of "First in First out".
- It replaces the oldest page that has been present in the main memory for the longest time.
- It is implemented by keeping track of all the pages in a queue.

Example:

A system uses 3 frames for storing process pages in main memory. It uses the First in First out (FIFO) page replacement policy. Assume that all the page frames are initially empty. What is the total number of page faults that will occur while processing the page reference string given below:

3, 1, 2, 1, 6, 5, 1, 3

Also calculate the hit ratio and miss ratio

Solution,
Total memory references = 8

Initially, all of the frames are empty so page faults occur at 3,1,2.

When page 1 is requested, it is in the memory so no page fault occurs.

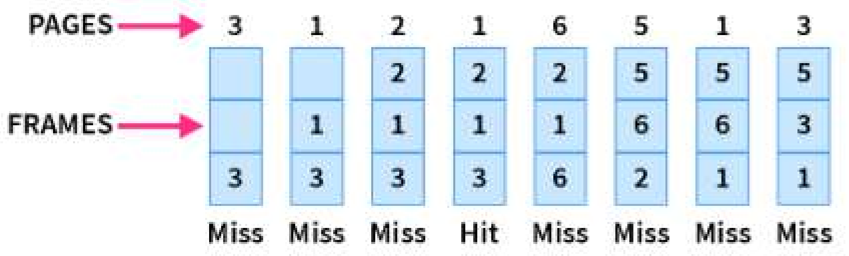
When page 6 is referenced, it is not present and a page fault occurs. Since there are no empty slots, we remove the front of the queue, i.e 3.

For page 5 , it is also not present and hence a page fault occurs. The front of the queue i.e 1 is removed.

For page 1, it is not found in memory and again a page fault occurs. The front of the queue i.e 2 is removed.

When page 3 is referenced, it is again not found in memory, a page fault occurs, and page 6 is removed being on top of the queue

So, total page faults = 7



Queue:

3	1	2	6	5	1	3

<u>Calculating Hit ratio</u>

Total number of page hits

= Total number of references – Total number of page misses or page faults

= 8 – 7

= 1

Thus, Hit ratio

= Total number of page hits / Total number of references

= 1 / 8

= 12.5%

<u>Calculating Miss ratio</u>

Total number of page misses or page faults = 6

Thus, Miss ratio

= Total number of page misses / Total number of references

= 7 / 8

= 87.5%

Alternatively,

Miss ratio

= 1 – Hit ratio

= 1 – 0.125

= 0.875

= 87.5%

Exercise: Consider the page reference string of size 12: 1, 2, 3, 4, 5, 1, 3, 1, 6, 3, 2, 3 with frame size 4(i.e. maximum 4 pages in a frame). If First in First out (FIFO) page replacement is used, what will be total number of page faults? Also calculate Hit Ratio.

Advantages

- FIFO Algorithm is simple and easy to implement.
- It does not cause more overheads

Disadvantages

- Poor performance.
- It doesn’t consider the frequency of use or last used time, simply replaces the oldest page.
- It suffers from Belady’s Anomaly.

Belady’s Anomaly

- Bélády’s anomaly is the name given to the phenomenon where increasing the number of page frames results in an increase in the number of page faults for a given memory access pattern.
- This phenomenon is commonly experienced when using the first-in first-out (FIFO) page replacement algorithm. In FIFO, the page fault may or may not increase as the page frames increase, but in optimal and stack-based algorithms like LRU, as the page frames increase, the page fault decreases.
- <u>Example (Belady's anomaly in FIFO page replacement) :</u>
  The reference String is given as 0 1 5 3 0 1 4 0 1 5 3 4. Let's analyze the behavior of FIFO algorithm in two cases.

When Number of frames = 3

Pages	**0**	**1**	**5**	**3**	**0**	**1**	**4**	**0**	**1**	**5**	**3**	**4**
Frame 3			5	5	5	1	1	1	1	1	3	3
Frame 2		1	1	1	0	0	0	0	0	5	5	5
Frame 1	0	0	0	3	3	3	4	4	4	4	4	4
Miss/Hit	Miss	Miss	Miss	Miss	Miss	Miss	Miss	Hit	Hit	Miss	Miss	Hit

Number of Page Faults = 9

When Number of frames = 4

Pages	**0**	**1**	**5**	**3**	**0**	**1**	**4**	**0**	**1**	**5**	**3**	**4**
Frame 4				3	3	3	3	3	3	5	5	5
Frame 3			5	5	5	5	5	5	1	1	1	1
Frame 2		1	1	1	1	1	1	0	0	0	0	4
Frame 1	0	0	0	0	0	0	4	4	4	4	3	3
Miss/Hit	Miss	Miss	Miss	Miss	Hit	Hit	Miss	Miss	Miss	Miss	Miss	Miss

Number of Page 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.

Note – It is not necessary that every string reference pattern cause Belady's anomaly in FIFO but there is certain kind of string references that worsen the FIFO performance on increasing the number of frames.

Second Chance Page Replacement Algorithms

- It is modified form of the FIFO page replacement algorithm.
- A reference (R) bit is used to give information about whether the page has been used or not.
  - Initially all reference bits are 0 (when page is brought to frame during miss)
  - As the page is referenced (when page is available in memory i.e. during hit), reference bit is set to 1 indicating that the page is in use.
  - A page with reference bit 1 is not replaced and given second chance
- As its name suggests, Second-chance gives every page a "second-chance" – an old page that has been referenced is probably in use, and should not be swapped out over a new page that has not been referenced.
- Second Chance Algorithm looks at the front of the queue as FIFO does, but instead of immediately paging out that page, it checks to see if its referenced bit is set.

— If it is not set (zero), the page is swapped out.

— Otherwise, the referenced bit is cleared, the page is inserted at the back of the queue (as if it were a new page) and this process is repeated.

- When a page gets a second chance, its reference bit is cleared, and its arrival time is reset to the current time.

Example:

Suppose reference string : 2,3,2,1,5,2,4,5,3 and number of frames = 3. Initially all frames are empty.

Pages	**2**	**3**	**2**	**1**	**5**	**2**	**4**	**5**	**3**
Frame 3				1(0)	1(0)	1(0)	4(0)	4(0)	3(0)
Frame 2		3(0)	3(0)	2(1)	2(0)	2(1)	2(1)	2(1)	2(1)
Frame 1	2(0)	2(0)	2(1)	3(0)	5(0)	5(0)	5(0)	5(1)	5(1)
Miss/Hit	Miss	Miss	Hit	Miss	Miss	Hit	Miss	Hit	Miss

Queue

2	3	1	2	5	4	2	5	3

— When page 2 and 3 are referenced, page fault occurs and they are brought to frames and reference bits are initially set to 0s.

— When page 2 is referenced, the page hit occurs and the reference bit is set to 1

— There is page fault when page 1 is referenced

— When page 5 is requested, page fault occurs and there is no free frame. Therefore, a page should be replaced. Here, page 2 is in front of queue and it's reference bit is 1. Since reference bit is set it is given second chance (the reference bit is set to 0 and placed at the page is placed at last of queue). Now page 3 is at front of queue and it's reference bit is 0. So this page is replaced.

— Again page 2 is referenced - it's page hit and the reference bit is set to 1

— When page 4 is referenced, page fault occurs and the page 1 is there at front of queue. Since the reference bit of this page 1 is 1, it is replaced.

— Again page 5 is referenced - it's page hit and the reference bit is set to 1

— There is page fault when page 3 is again referenced because it was swapped out. At this time page 2 is at front of the queue and it's reference bit is 1. So, again, second chance is given to page 2 . Now page 5 is at front the queue and it's reference bit is also set, second chance is given to page 5 also. Page 4 is replaced this time.

Here, **Number of Page Faults = 6**

Miss ratio = Total number of page misses / Total number of references

6/9 = 0.66

Exercise: Trace Second Chance Page Replacement Algorithm based on following information. Also compute hit ratio.

Reference string: 0 4 1 4 2 4 3 4 2 4 0 4 1 4 2 4 3 4 and 3 frames.

Advantages

- Improvement over FIFO
- Allows frequently used page to stay in memory.

Disadvantages

- It suffers from Belady’s Anomaly.
- Overhead of updating and checking reference bits.

Least Recently Used (LRU) Page Replacement Algorithm

- Least Recently Used page replacement algorithm keeps track of page usage over a short period of time.
- It works on the idea that the pages that have been most heavily used in the past are most likely to be used heavily in the future too.
- When a page fault occurs, **throw out the page that has been unused for the longest time.** This strategy is called LRU (Least Recently Used) paging.
- This idea is based on *locality of reference*. Any page that has been unused for long time is more likely to remain unused. Therefore, it is better to replace that page.
- To fully implement LRU, it is necessary to maintain a linked list of all pages in memory, with the most recently used page at the front and the least recently used page at the rear.
    - The list must be updated on every memory reference.
    - Finding a page in the list, deleting it, and then moving it to the front is a very time consuming operation

Example: Let’s see the performance of the LRU on the reference string of 3, 1, 2, 1, 6, 5, 1, 3 with 3- frames

Page Requests	**3**	**1**	**2**	**1**	**6**	**5**	**1**	**3**
Frame 3			2	2	2	5	5	5
Frame 2		1	1	1	1	1	1	1
Frame 1	3	3	3	3	6	6	6	3
Miss/Hit	M	M	M	H	M	M	H	M

Initially, since all the frames are empty, pages 3, 1, 2 cause a page fault and take the empty frames

When page 1 is requested, it is in the memory and no page fault occurs.

For page 6, it is not in the memory, so a page fault occurs and the least recently used page 3 is replaced.

When page 5 is referenced, it again causes a page fault and page 2 is removed as it is now the least recently used page

When page 1 is requested again, it is in the memory and hence page hit occurs.

When page 3 is requested, the page fault occurs again and this time page 6 is replaced as the least recently used one.

Total number of page faults = 6

Advantages

— Efficient.

— Doesn't suffer from Belady’s Anomaly.

Disadvantages

— Complex Implementation.

— Time consuming.

— Requires hardware support.

Exercise:

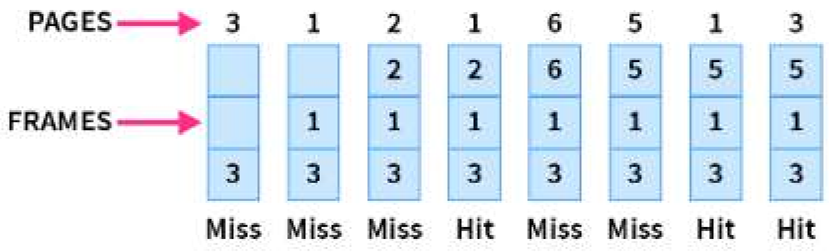
Consider reference string : 1,2,3,4,5,1,3,1,6,3,2,3 and number of frames = 4 . Calculate page fault ratio.

Optimal Page Replacement Algorithm

- Optimal Page Replacement algorithm is the best page replacement algorithm as it gives the least number of page faults.
- It is also known as OPT, clairvoyant replacement algorithm, or Belady's optimal page replacement policy.
- In this algorithm, pages are replaced which would not be used for the longest duration of time in the future
- This algorithm replaces the page that will not be referred by the CPU in future for the longest time.
- Each page can be labeled with the number of instructions that will be executed before that page is first referenced. The optimal page replacement algorithm says that the page with the highest label should be removed. If one page will not be used for 8 million instructions and another page will not be used for 6 million instructions, removing the former pushes the page fault that will fetch it back as far into the future as possible
- It is practically impossible to implement this algorithm. This is because the pages that will not be used in future for the longest time cannot be predicted.
- Optimal page replacement is best, but not possible in practice as the operating system cannot know future requests.
- The utilization of optimal Page replacement is to set up a benchmark so that other replacement algorithms can be analyzed against it.

Example:

Page reference string: 3, 1, 2, 1, 6, 5, 1, 3 with 3 frames



PAGES	3	1	2	1	6	5	1	3
FRAMES			2	2	6	5	5	5
		1	1	1	1	1	1	1
	3	3	3	3	3	3	3	3
	Miss	Miss	Miss	Hit	Miss	Miss	Hit	Hit

Initially, since all the slots are empty, pages 3, 1, 2 cause a page fault and take the empty frames.

When page 1 is requested, it is in the memory and no page fault occurs.

When page 6 is referenced, it is not in the memory, so a page fault occurs and page 2 is replaced as it is not going to be used again.

When page 5 is requested, it is also not in the memory and causes a page fault. Similar to above page 6 is removed as it is not going to be used again.

When page 1 and page 3 are requested, they are in the memory so no page fault occurs.

Advantages

— Excellent efficiency

— Less complexity

— Easy to use and understand

— Simple data structures can be used to implement

— Used as the benchmark for other algorithms

Disadvantages

— More time consuming

— Difficult for error handling

— Impractical since it needs future knowledge of the program, which is not possible.

Exercise:

Consider reference string: 7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 7 0 1 and number of frames = 4 . Calculate page fault ratio. (Answer: Page Fault ratio = 8/20)

Least Frequently Used (LFU) Page Replacement Algorithm

- This algorithm selects a page for replacement if the page has not been used often in the past or replaces the page with smallest frequency of references
- For each page loaded in memory, a counter is assigned which counts the frequency of reference.
- Initially frequency of all pages are 0.
- When the page is referenced it's frequency is incremented by 1.
- In LFU policy, the least frequently used page is replaced.
- If the frequency of pages is the same, then the page that has arrived first is replaced first.

Example:

Reference String: 7 0 2 4 3 1 4 7 2 0 4 3 0 3 2 7 number of frames = 3

Page Requests	**7**	**0**	**2**	**4**	**3**	**1**	**4**	**7**	**2**	**0**	**4**	**3**	**0**	**3**	**2**	**7**
Frame 3			2	2	2	1	1	1	2	2	2	3	3	3	3	3
Frame 2		0	0	0	3	3	3	7	7	0	0	0	0	0	2	7
Frame 1	7	7	7	4	4	4	4	4	4	4	4	4	4	4	4	4
Miss/Hit	M	M	M	M	M	M	H	M	M	M	H	M	H	H	M	M

Referenced Pages	**Reference Frequencies**						
7	0	1	0	1	0		
0	0	1	0	1	2	0	
2	0	1	0	1	0	1	0
4	0	1	2	3			
3	0	1	0	1	2		
1	0	1	0				

Firstly, there are three empty frames in the memory. Therefore, when 7, 0, 2 come into the frame, they are allocated to the empty frame in order of their arrival. It occurs page fault because 7, 0, 2 are not present in the memory.

When 4 is requested, it is not present in the memory. Therefore, a page fault occurs, replacing 7, which is the least frequently used page.

When 3 is requested, it is not present in the memory. Therefore, a page fault occurs, replacing 0, which is the least frequently used page.

When 1 is requested, it is not present in the memory. Therefore, a page fault occurs, replacing 2, which is the least frequently used page.

When 4 is requested, it is present in the memory. Therefore, a page hit occurs, and no replacement occurs.

When 7 is requested, it is not present in the memory. Therefore, a page fault occurs, replacing 3, which is the least frequently used page.

When 2 is requested, it is not present in the memory. Therefore, a page fault occurs, replacing 1, which is the least frequently used page.

When 0 is requested, it is not present in the memory. Therefore, a page fault occurs, replacing 7, which is the least frequently used page.

When 4 is requested, it is present in the memory. Therefore, a page hit occurs, and no replacement occurs.

When 3 is requested, it is not present in the memory. Therefore, a page fault occurs, replacing 2, which is the least frequently used page.

When 0 and 3 are requested, it is present in the memory. Therefore, a page hit occurs, and no replacement occurs.

When 2 is requested, it is not present in the memory. Therefore, a page fault occurs, replacing 0, which is the least frequently used page.

When 7 is requested, it is not present in the memory. Therefore, a page fault occurs, replacing 2, which is the least frequently used page.

Total number of page faults or page misses = 12

Clock Page Replacement Algorithm

- In second chance algorithm pages are constantly moving around on its list. So it is not working efficiently.
- A better approach is to keep all the page frames on a circular list in the form of a clock.
- When a page fault occurs, the page being pointed to by the hand is inspected.
  - If its reference bit (R bit) is 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 is 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
- It differs from second chance algorithm only in the implementation

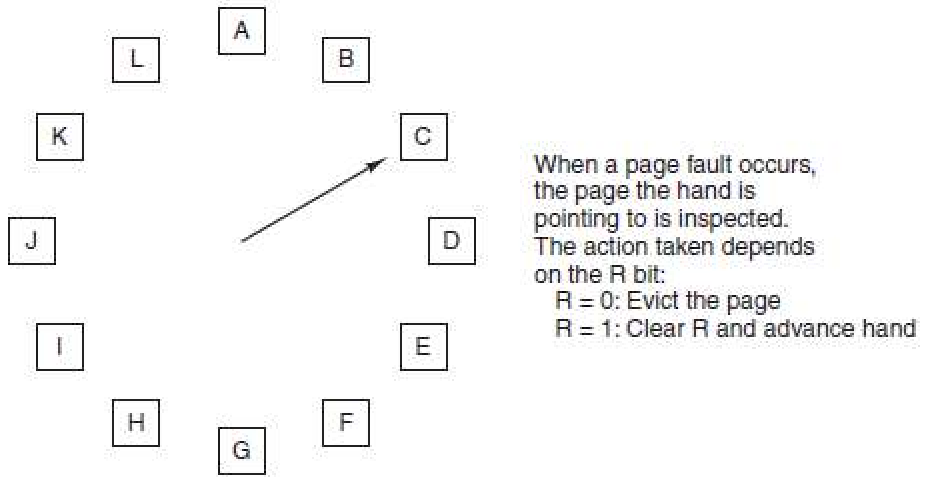

A
B
C
D
E
F
G
H
I
J
K
L
When a page fault occurs, the page the hand is pointing to is inspected. The action taken depends on the R bit:
R = 0: Evict the page
R = 1: Clear R and advance hand

Figure: The Clock Page Replacement Algorithm

Example: Discussed in class

Working Set (WS) Page Replacement Algorithm

- The set of pages that a process is currently using is its **working set**. A program causing page faults every few instructions is said to be **thrashing**.
- In a multiprogramming system, processes are often moved to disk. The question arises of what to do when a process is brought back in again.
- Many paging systems try to keep track of each process' working set and make sure that it is in memory before letting the process run. This approach is called the *working set model.* Loading the pages before letting processes run is also called **prepaging.**

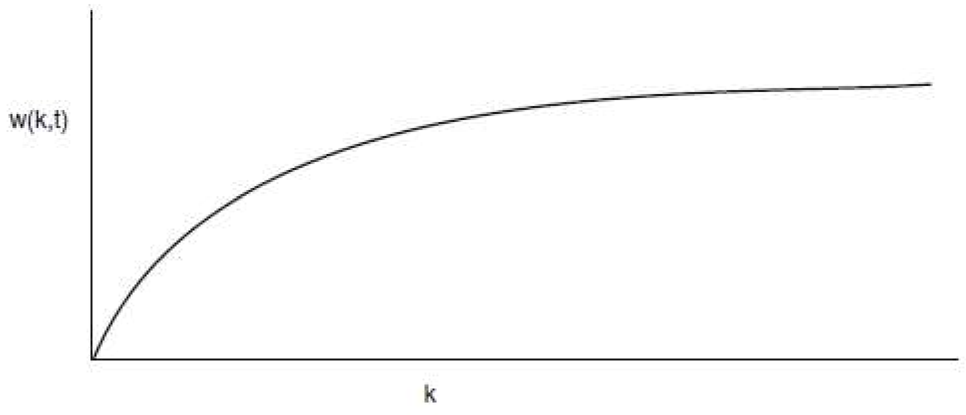


Figure: The working set is the set of pages used by the k most recent memory references. The function w(k, t) is the size of the working set at time t.

The algorithm works as follows:

- Clear the reference bit periodically
- On every page fault, scan the page table and examine the R bit
    - if it is 1, the current virtual time is written into the Time of last use field in the page table
    - if R is 0, ts age (the current virtual time minus its Time of last use) is computed and compared to τ. If the age is greater than τ , the page is no longer in the working set and the new page replaces it.
    - The scan continues updating the remaining entries.

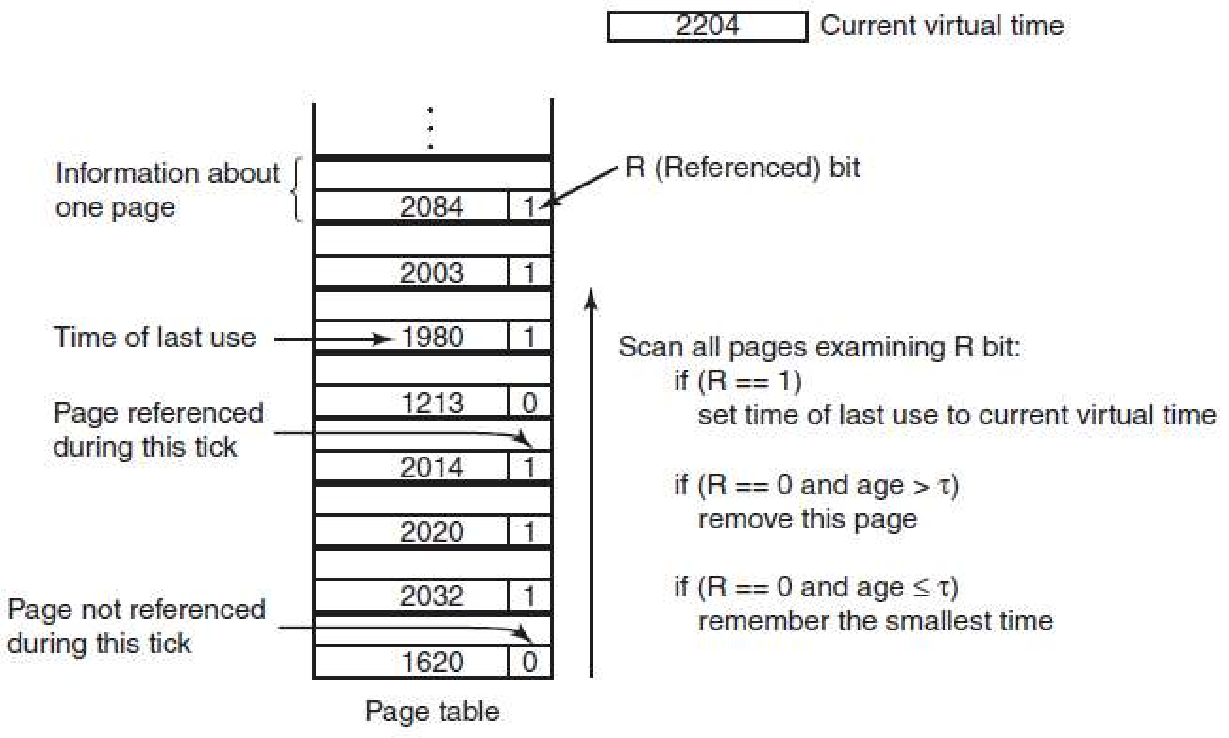

2204
Current virtual time
Information about one page
2084
1
R (Referenced) bit
2003
1
Time of last use
1980
1
1213
0
Page referenced during this tick
2014
1
2020
1
2032
1
Page not referenced during this tick
1620
0
Page table
Scan all pages examining R bit:
if (R == 1)
set time of last use to current virtual time
if (R == 0 and age > τ)
remove this page
if (R == 0 and age ≤ τ)
remember the smallest time

Figure: The working set algorithm.

WS-Clock Page Replacement Algorithm

- This algorithm is combination of Working Set and Clock page replacement algorithms
- Due to its simplicity of implementation and good performance, it is widely used in practice.
- The data structure needed is a circular list of page frames, as in the clock algorithm
- Each entry contains the Time of last use field from the basic working set algorithm, as well as the R bit (shown) and the M bit (not shown).
- At each page fault the page pointed to by the hand is examined first.
    - If the R bit is set to 1, the R bit is then set to 0, the hand advanced to the next page
    - if the page pointed to has R = 0,
        - If the age is greater than τ and the page is clean, the page frame is simply claimed and the new page put there
        - if the page is dirty, the write to disk is scheduled, but the hand is advanced and the algorithm continues with the next page.

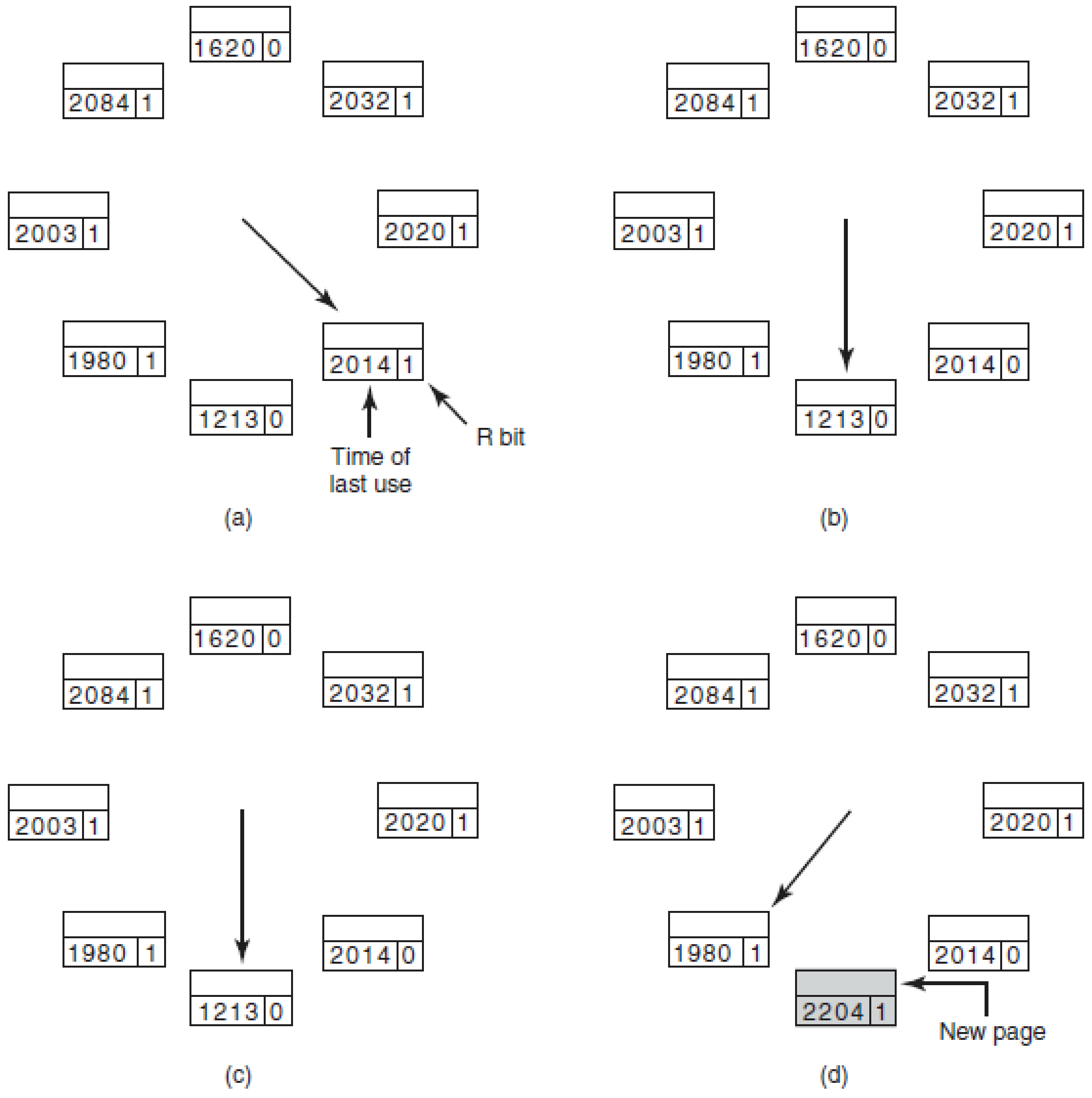


Figure : Operation of the WSClock algorithm. (a) and (b) give an example of what happens when R = *1. (c) and (d) give an example of* R = *0.*

Segmentation

- Segmentation is a memory management technique in which each job is divided into several segments of different sizes, one for each module that contains pieces that perform related functions.
- Each segment is actually a different logical address space of the program.

- Like Paging, Segmentation is another non-contiguous memory allocation technique.
- In segmentation, process is not divided blindly into fixed size pages. Rather, the process is divided into modules for better visualization.
- Segmentation is a variable size partitioning scheme.
- In segmentation, secondary memory and main memory are divided into partitions of unequal size.
- The size of partitions depend on the length of modules.
- The partitions of secondary memory are called as segments.
- Segmentation includes segment table for address mapping

Need of Segmentation

From the user's view, a program has segments such as the main method, with a set of objects, methods, arrays, procedures, global variables, templates etc. So these modules or data elements are the segments to which user refers it by its name without concerning at what address these modules or elements are stored in memory. Neither the user is worried about in which sequence the modules or data elements are stored in memory.

In paging, it is possible that the operating system divides the same segment (e.g. function) into different pages and those pages may or may not be loaded into the memory at the same time . Operating system also does not care about the User's view of the process. Due to this technique system's efficiency decreases.

Segmentation is a better technique because it divides the process into segments.

User view of program

- Consider a program is divided into 5 segments as

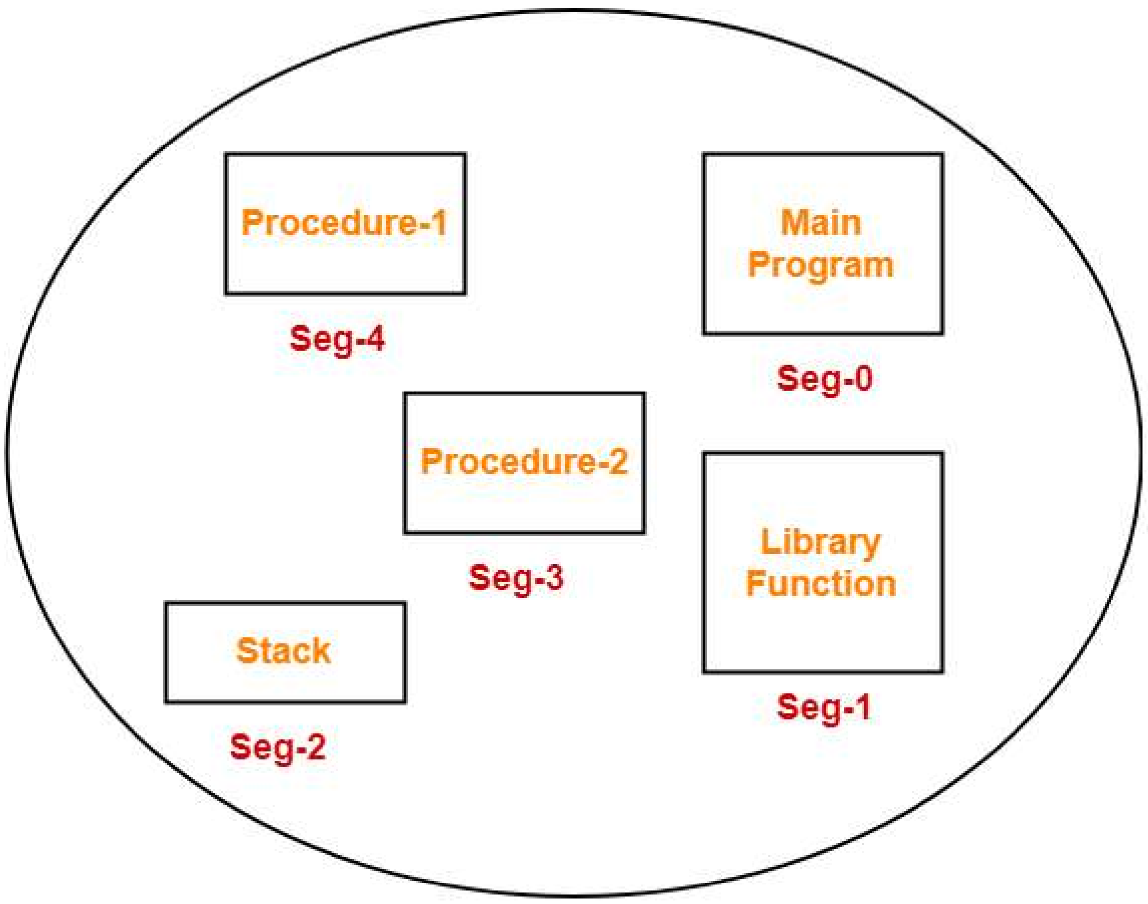
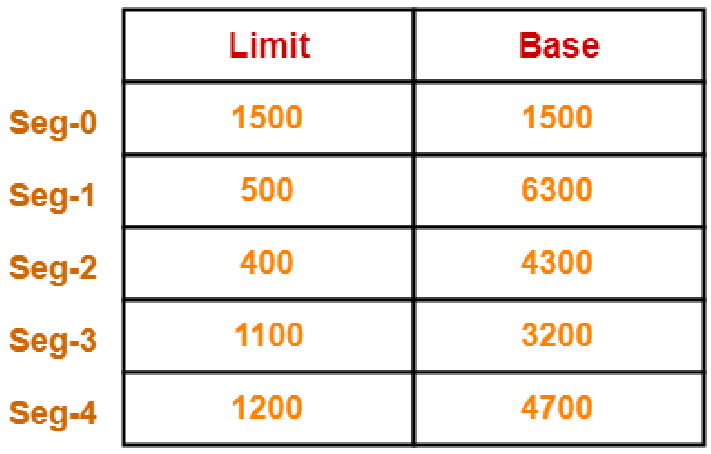


Figure : Logical Address Space

Segment table

- **Segment table** is a table that stores the information about each segment of the process.
- It has two columns.
  - First column stores the size or length of the segment.
  - Second column stores the base address or starting address of the segment in the main memory.
- Segment table is stored as a separate segment in the main memory.
- Segment table base register (STBR) stores the base address of the segment table.

For the above illustration, consider the segment table is-



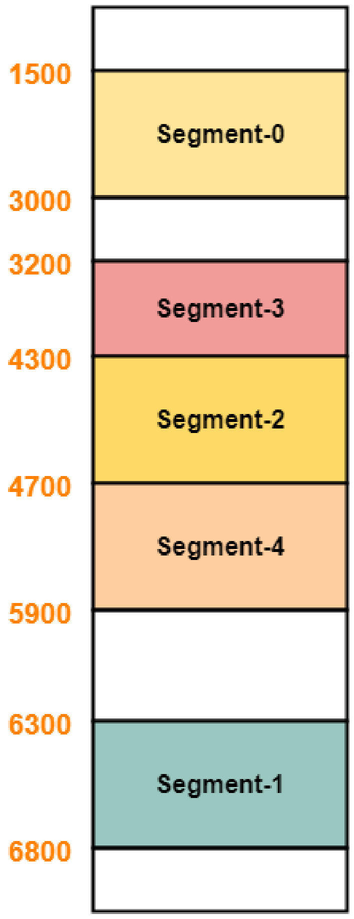
	Limit	Base
Seg-0	1500	1500
Seg-1	500	6300
Seg-2	400	4300
Seg-3	1100	3200
Seg-4	1200	4700

Figure: Segment Table

— Limit indicates the length or size of the segment.

— Base indicates the base address or starting address of the segment in the main memory.

In accordance to the above segment table, the segments are stored in the main memory as-



Translating Logical Address into Physical Address

- CPU always generates a logical address. A physical address is needed to access the main memory.
- A logical address consisting of two parts-
  - ➢ Segment Number (s)
    - Segment Number specifies the specific segment of the process from which CPU wants to read the data.
  - ➢ Segment Offset (d)
    - Segment Offset specifies the specific word in the segment that CPU wants to read.

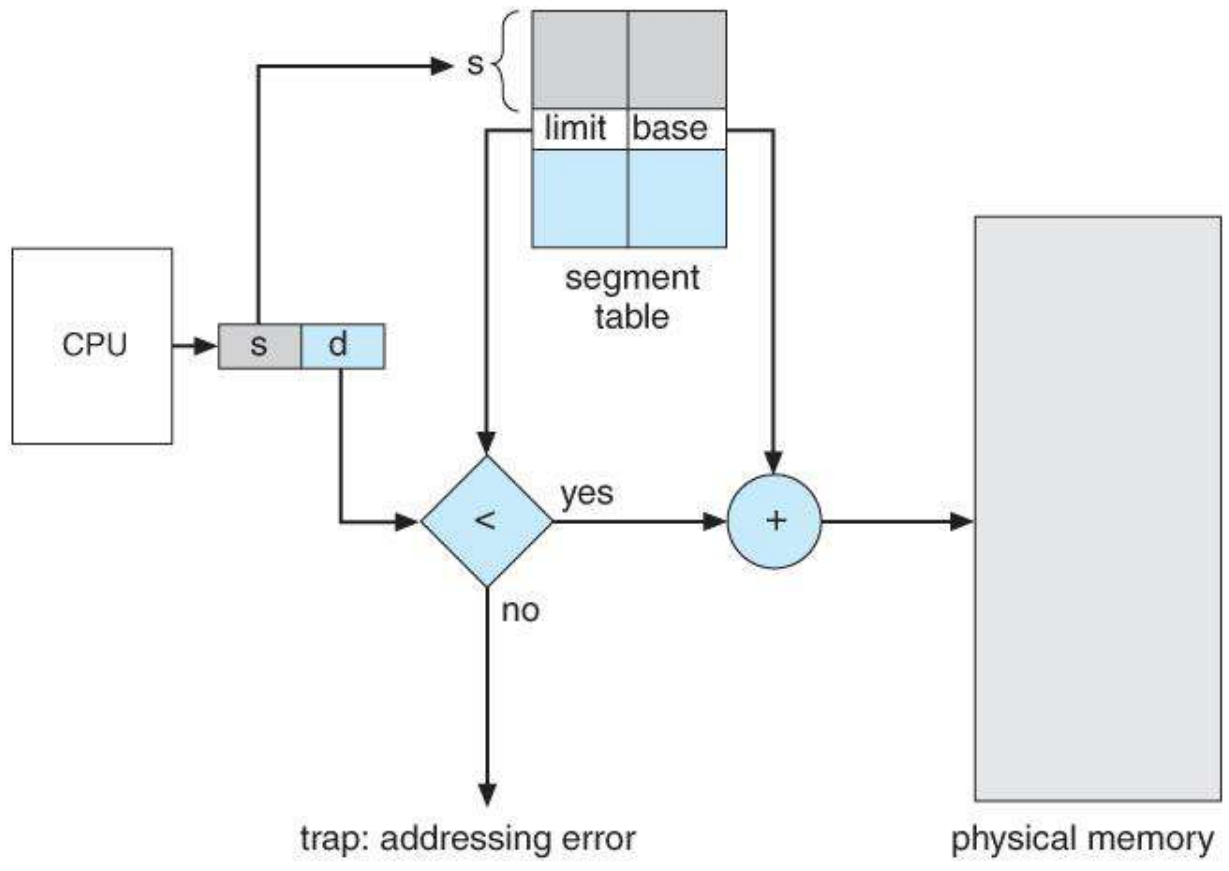


Figure: Translating Physical Address into Logical Address in Segmentation (Segmentation Hardware)

➢ For the generated segment number, corresponding entry is located in the segment table. Then, segment offset is compared with the limit (size) of the segment.

☞ If segment offset is found to be greater than or equal to the limit, a trap is generated.

- If segment offset is found to be smaller than the limit, then request is treated as a valid request.
    - The segment offset must always lie in the range [0, limit-1],
    - Then, segment offset is added with the base address of the segment.
    - The result obtained after addition is the address of the memory location storing the required word.

Advantages

The advantages of segmentation are-

- It allows to divide the program into modules which provides better visualization.
- Segment table consumes less space as compared to Page Table in paging.
- There is no internal fragmentation.

Disadvantages

The disadvantages of segmentation are-

- There is an overhead of maintaining a segment table for each process.
- The time taken to fetch the instruction increases since now two memory accesses are required. First, we need to access the segment table which is also stored in the main memory and second, combine the base address of the segment with the segment offset and then get the physical address which is again stored in the main memory.
- Segments of unequal size are not suited for swapping.
- It suffers from external fragmentation as the free space gets broken down into smaller pieces with the processes being loaded and removed from the main memory.

Paging vs Segmentation

S.N.	Paging	Segmentation
1	Non-Contiguous memory allocation	Non-contiguous memory allocation
2	Paging divides program into fixed size pages.	Segmentation divides program into variable size segments.
3	In paging, the user just provides a single integer as the address,that is divided by the hardware into a page number and offset.	In the segmentation method, the user specifies the address in two quantities: segment number and offset.
4	Paging is faster than segmentation	Segmentation is slower than paging
5	Paging is closer to Operating System	Segmentation is closer to User
6	It suffers from internal fragmentation	It suffers from external fragmentation
8	Logical address is divided into page number and page offset	Logical address is divided into segment number and segment offset
9	Page table is used to maintain the page information.	Segment Table maintains the segment information

Segmentation with paging: MULTICS

- ☞ Pure segmentation is not very popular and not being used in many of the operating systems. However, Segmentation can be combined with Paging to get the best features out of both the techniques.
- ☞ In Segmented Paging, the main memory is divided into variable size segments which are further divided into fixed size pages.
- ☞ If the segments are large, it may be inconvenient, or even impossible, to keep them in main memory in their entirety. This leads to the idea of paging them, so that only those pages of a segment that are actually needed have to be around. MULTICS (Multiplexed Information and Computing Service) supports this scheme.

☞ **In segmentation with paging** :

- Process is first divided into segments and then each segment is divided into pages.
- These pages are then stored in the frames of main memory.
- A page table exists for each segment that keeps track of the frames storing the pages of that segment.
- Each page table occupies one frame in the main memory.
- Number of entries in the page table of a segment = Number of pages that segment is divided.
- A segment table exists that keeps track of the frames storing the page tables of segments.
- Number of entries in the segment table of a process = Number of segments that process is divided.
- The base address of the segment table is stored in the segment table base register (STBR).

Address translation scheme in MULTICS (Paging with Segmentation)

- CPU generates a logical address consisting **Segment Number (s)**, and **Segment Offset (d). Where the segment Offset is the combination of Page Number(p)** and **Page Offset (d')**
  - **Segment Number** specifies the specific segment from which CPU wants to reads the data.
  - **Page Number** specifies the specific page of that segment from which CPU wants to read the data.
  - **Page Offset** specifies the specific word on that page that CPU wants to read.
- For the generated segment number, corresponding entry is located in the segment table.
  - Segment table provides the frame number of the frame storing the page table of the referred segment.
  - The frame containing the page table is located.
- For the generated page number, corresponding entry is located in the page table.
  - Page table provides the frame number of the frame storing the required page of the referred segment.

  - The frame containing the required page is located.

— The frame number combined with the page offset forms the required physical address.

  - For the generated page offset, corresponding word is located in the page and accessed.

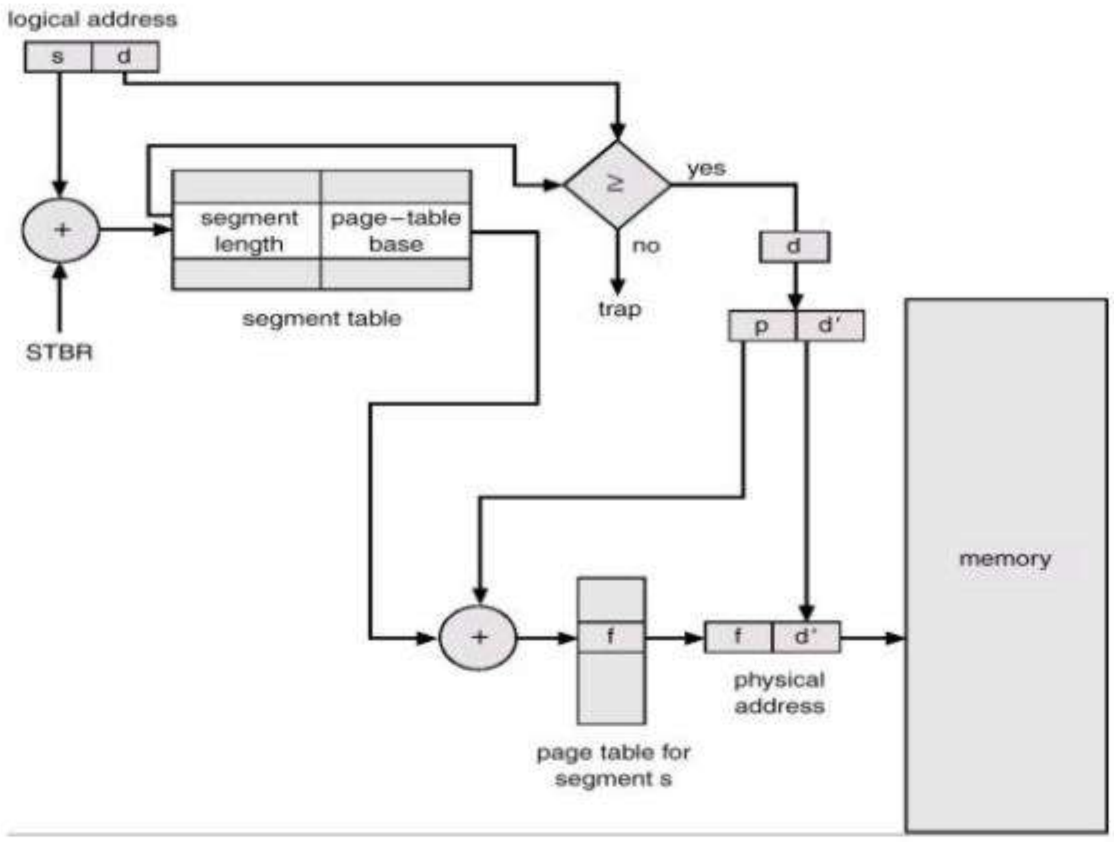


Figure: Address Translation Scheme in MULTICS (Segmentation with Paging)