# UNIT IV

## MEMORY MANAGEMENT

**Memory Management**: Basic concept, Logical and Physical address map, Memory allocation: Contiguous Memory allocation – Fixed and variable partition–Internal and External fragmentation and Compaction; Paging: Principle of operation – Page allocation – Hardware support for paging, Protection and sharing, Disadvantages of paging.

**Virtual Memory:** Basics of Virtual Memory – Hardware and control structures – Locality of reference, Page fault , Working Set , Dirty page/Dirty bit – Demand paging, Page Replacement algorithms: Optimal, First in First Out (FIFO), Second Chance (SC), Not recently used (NRU) and Least Recently used (LRU).

---

## Memory Management

Memory management is the functionality of an operating system which handles or manages primary memory and moves processes back and forth between main memory and disk during execution. Memory management keeps track of each and every memory location, regardless of either it is allocated to some process or it is free. It checks how much memory is to be allocated to processes. It decides which process will get memory at what time. It tracks whenever some memory gets freed or unallocated and correspondingly it updates the status.

**Basic concept:**

- ***Memory accesses and memory management*** are a very important part of modern computer operation. Every ***instruction has to be fetched from memory*** before it can be executed, and most instructions involve ***retrieving data from memory or storing data in memory*** or both.
- The advent of ***multi-tasking OSs*** increases the complexity of memory management, because as ***processes are swapped in and out of the CPU***, so their ***code and data*** must be swapped in and out of memory, at ***high speeds and without interfering*** with any other processes.

### 1. Basic Hardware

- The CPU can only access its registers and main memory. It ***cannot make direct access to the hard driv***e, so any data stored in hard disk must first be transferred into the main memory before the CPU can work with it.
- ***Memory accesses to registers*** are very fast, generally one clock tick, and a CPU may be able to execute more than one machine instruction per clock tick.

- ***Memory accesses to main memory*** are comparatively slow, and may take a number of clock ticks to complete.
- An intermediary fast memory ***cache*** built into most modern CPUs. The basic idea of the cache is to transfer chunks of memory at a time from the main memory to the cache, and then to access individual memory locations one at a time from the cache.
- User processes must be restricted so that they ***only access memory locations that "belong" to that particular process***. This is usually implemented using a ***base register and a limit register for each process***, as shown in Figures below.
- Every memory access made by a user process is ***checked against these two registers,*** and if a memory access is attempted ***outside the valid range, then a fatal error*** is generated.
- For example:

Base register = 300040

limit register = 120900

then legal address = (300040+120900)= 420940(inclusive).

[legal address = base register+ limit register]

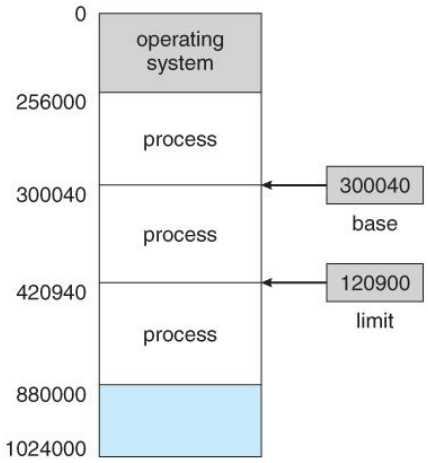



Figure - A **base and a limit register** define a logical address space

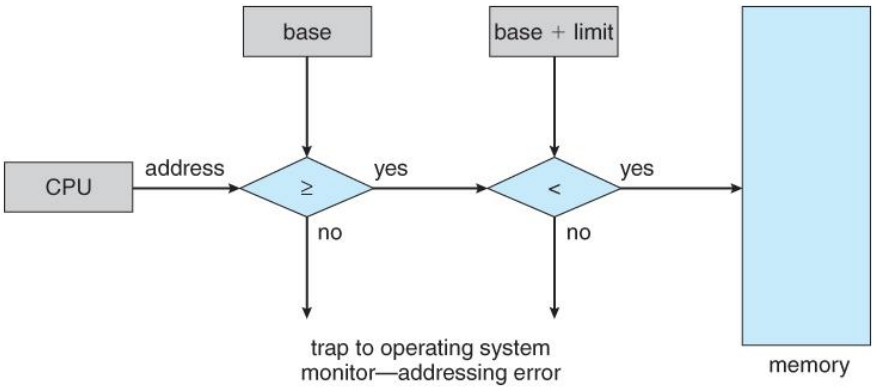



Figure - Hardware address protection with base and limit registers

## 2. Address Binding

- User programs typically refer to ***memory addresses with symbolic names*** such as "i", "count", and "averageTemperature". These ***symbolic names must be mapped or bound to physical memory addresses***, which typically occurs in several stages:
    - **Compile Time -** If you know that during compile time where process will reside in memory, then **absolute address** is generated i.e **physical address is embedded to the executable of the program during compilation**. Loading the executable as a process in memory is very fast. But if the generated address space is preoccupied by other process, then the program crashes and it becomes necessary to **recompile the program to change the address space**. DOS programs use compile time binding.
    - **Load Time -**(loading the executable file from hard disk to main memory) If it is not known at the compile time, where process will reside, then **relocatable address** will be generated. **Loader translates the relocatable address to absolute address**. The base address of the process in main memory is added to all logical addresses by the loader to generate absolute address. In this if the **base address of the process changes** then we need to **reload the process again**.
    - **Execution (run) Time -** If a program can be **moved around in memory during the course of its execution**, then binding must be **delayed until execution time**. This requires special hardware and is the method implemented by most modern OSs. [The instructions are in memory and are being processed by the CPU. Additional memory may be allocated and/or deallocated at this time.]
- Figure below shows the various stages of the binding processes and the units involved in each stage:

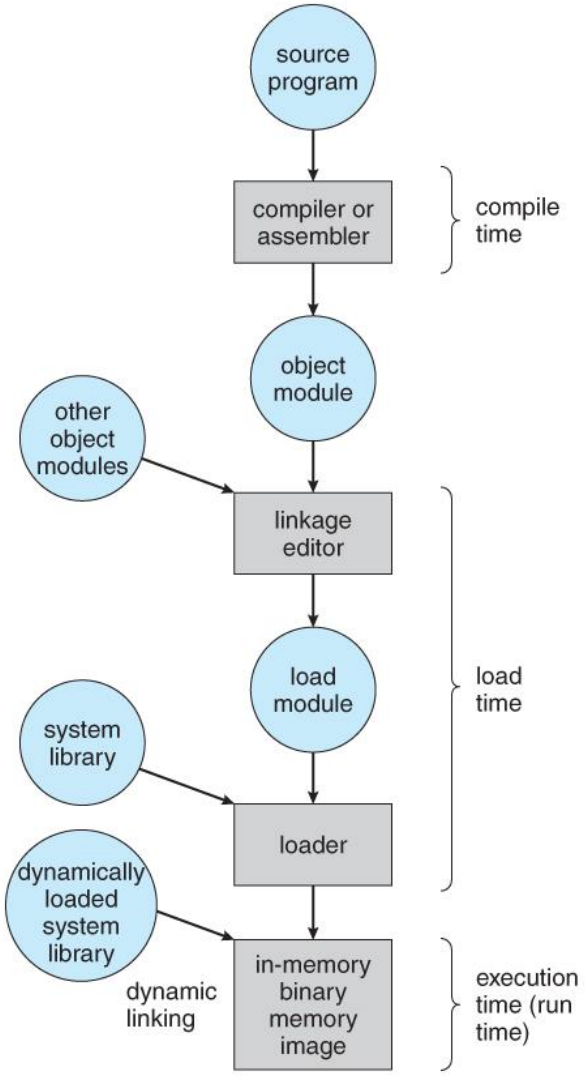



Figure - Multistep processing of a user program

### 3. Logical Versus Physical Address Space

Address uniquely identifies a location in the memory. We have two types of addresses that are logical address and physical address. The logical address is a virtual address and can be viewed by the user. The user can't view the physical address directly. The logical address is used like a reference, to access the physical address. The fundamental difference between logical and physical address is that **logical address** is generated by CPU during a program execution whereas, the **physical address** refers to a location in the memory unit.

| BASIS FOR COMPARISON | LOGICAL ADDRESS | PHYSICAL ADDRESS |
|---|---|---|
| Basic | It is the **virtual address** generated by CPU | The physical address is a location in a memory unit. |
| Address Space | Set of all logical addresses generated by CPU in reference to a program is referred as Logical Address Space. | Set of all physical addresses mapped to the corresponding logical addresses is referred as Physical Address. |
| Visibility | The user can view the logical address of a program. | The user can never view physical address of program |
| Access | The user uses the logical address to access the physical address. | The user can not directly access physical address. |
| Generation | **The Logical Address is generated by the CPU** | **Physical Address is Computed by MMU** |

**Memory-Management Unit (MMU) or Address Translation Unit:**

The user program generates the logical address and thinks that the program is running in this logical address. But the program needs physical memory for its execution. Hence, the logical address must be mapped to the physical address before they are used.

The logical address is mapped to the physical address using a hardware device called **Memory-Management Unit (MMU)**. The set of all physical addresses corresponding to the logical addresses in a Logical address space is called **Physical Address Space**.

The address-binding methods used by MMU generates **identical** logical and physical address during **compile time** and **load time**. However, while **run-time** the address-binding methods generate **different** logical and physical address.

- 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.

**MMU scheme :** CPU-------> MMU------>Memory

1. CPU will generate logical address (for eg: 346)
2. MMU will generate relocation register(base register) for eg:14000
3. In Memory, physical address is located eg:(346+14000= 14346)

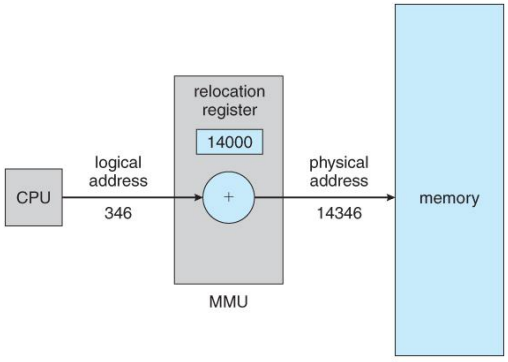



**Figure - Dynamic relocation using a relocation register**

***Exercise:***

Consider a logical address space of **eight pages of 1024 words** each, mapped onto a physical memory of **32 frames**.

a. How many bits are there in the logical address?

b. How many bits are there in the physical address?

***Answer:***

Addressing within a 1024-word page requires 10 bits because $1024 = 2^{10}$. Since the logical address space consists of $8 = 2^3$ pages, the logical addresses must be 10+3 = 13 bits.

Similarly, since there are $32 = 2^5$ physical pages, phyiscal addresses are 5 + 10 = 15 bits long.

a. Logical address: 13 bits

b. Physical address: 15 bits

**Ex-2:**

Consider a logical address space of 64 pages of 1024 words each, mapped onto a physical memory of 32 frames.

a. How many bits are there in the logical address?
b. How many bits are there in the physical address?

Answer:
a. Logical address: 16 bits
b. Physical address: 15 bits

**4. Dynamic Loading**

- Rather than loading an entire program into memory at once, dynamic loading **loads up each routine as it is called**. The advantage is that **unused routines need never be loaded**, **reducing total memory usage** and generating **faster program startup times**. The downside is the added complexity and overhead of checking to see if a routine is loaded every time it is called and then loading it up if it is not already loaded.

**5. Dynamic Linking and Shared Libraries (DLL)**

Main Memory refers to a physical memory that is the internal memory to the computer. The word main is used to distinguish it from external mass storage devices such as disk drives. Main memory is also known as RAM. The computer is able to change only data that is in main memory. Therefore, every program we execute and every file we access must be copied from a storage device into main memory.

**Dynamic Loading:**

All the programs are loaded in the main memory for execution. Sometimes complete program is loaded into the memory, but sometimes a certain part or routine of the program is loaded into the

main memory only when it is called by the program, this mechanism is called **Dynamic Loading**, this enhance the performance.

**Dynamic Linking:**

Also, at times one program is dependent on some other program. In such a case, rather than loading all the dependent programs, CPU links the dependent programs to the main executing program when its required. This mechanism is known as **Dynamic Linking**.

***Dynamic Linking Vs. static linking***

- With **static linking** library modules get fully included in executable modules, ***wasting both disk space and main memory usage***, because every program that included a certain routine from the library would have to have their own copy of that routine linked into their executable code.

## Swapping

- A process must be loaded into memory in order to execute.
- If there is not enough memory available to keep all running processes in memory at the same time, then some processes who are not currently using the CPU may have their memory swapped out to a fast local disk (Hard disk) called the ***backing store.***

### 1. Standard Swapping

- If **compile-time or load-time address binding** is used, then **processes must be swapped back into the same memory location** from which they were swapped out. If execution time binding is used, then the processes can be swapped back into any available location.
- Swapping is a **very slow process** compared to other operations.

  For example, if a user process occupied 10 MB and the transfer rate for the backing store were 40 MB per second, then it would take **1/4 second ( 250 milliseconds )** just to do the data transfer. **Adding in a latency lag of 8 milliseconds** and ignoring head seek time for the moment, and further recognizing that swapping involves moving old data out as well as new data in, the **overall transfer time required for this swap** is **512 milliseconds**, or over half a second. For efficient processor scheduling the CPU time slice should be significantly longer than this lost transfer time.

- To reduce swapping transfer overhead, it is **desired to transfer as little information as possible,** which requires that the system must know how much memory a process is using.
- It is important to swap processes out of memory **only when they are idle**, or only when there are **no pending I/O operations**. ( Otherwise the pending I/O operation could write into the wrong process's memory space. ) The solution is to **either swap only totally idle processes, or do I/O operations only into and out of OS buffers**, which are then transferred to or from process's main memory as a second step.
- Most modern OSes no longer use swapping, because it is too slow and there are faster alternatives available. ( e.g. Paging. ) However **some UNIX systems** will still invoke swapping if the system gets extremely full, and then discontinue swapping when the load reduces again.

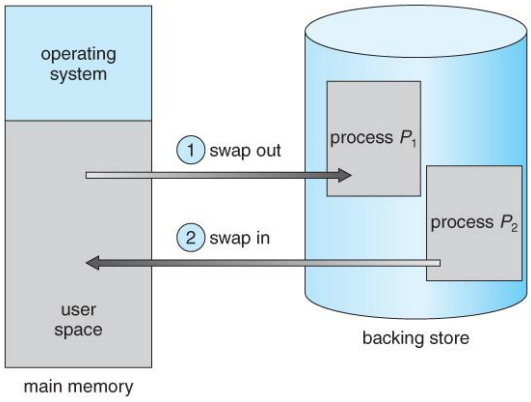



**Figure - Swapping of two processes using a disk as a backing store**

## 2. Swapping on Mobile Systems

- Swapping is typically not supported on mobile platforms, for several reasons:
  - Mobile devices typically **use flash memory** in place of more spacious hard drives for persistent storage, so there is **not as much space available**.
  - Flash memory can only be **written to a limited number of times** before it becomes unreliable.
  - **The bandwidth** to flash memory is also **lower**.

**Exercise on swapping:**

**Ex-1:**

Let us assume that the user **process is of size 2048KB** and on a standard hard disk where swapping will take place has a **data transfer rate around 1 MB** (1024KB) per second.

The actual transfer of the process to or from memory will take

=2048KB / 1024KB per second

= 2 seconds

= 2000 milliseconds (ie. $2*10^3$)

Now considering in and out time, it will take complete **4000 milliseconds** plus other overhead where the process competes to regain main memory.

**Ex-2:**

The user process is 100 MB in size and the backing store is a standard hard disk with a transfer rate of 50 MB per second. The actual transfer of the 100 MB process to or from main memory takes:

100 MB/50 MB per second = 2 seconds= 2 *$10^3$= 2000 milliseconds.

The swap time is 2000 milliseconds. Since we must swap both out and in, the total swap time is about 4000 milliseconds.

## Memory allocation

**Contiguous Memory allocation:**

- One approach to memory management is to load each process into a contiguous space.
- Main memory is split into two partitions:
    - Low memory : operating system is held
    - High memory: user processes are held

The operating system is allocated space first, usually at low memory locations, and then the remaining available memory is allocated to user processes as needed.

**1. Memory Protection ( or Memory Mapping and Protection )**

- The system shown in figure below allows protection against user programs accessing areas that they should not, allows programs to be relocated to different memory starting addresses as needed, and allows the memory space devoted to the OS to grow or shrink dynamically as needs change.

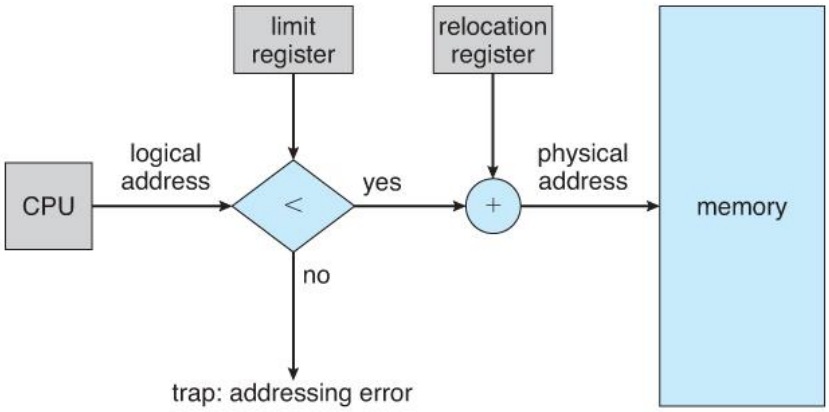


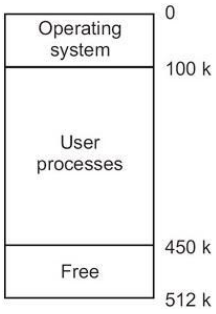

Figure - Hardware support for relocation and limit registers

**2. Memory Allocation**

Now we will discuss about the various memory allocation schemes.

**i. Single Partition Allocation**

In this scheme Operating system is residing in low memory and user processes are executing in higher memory.




**Advantages**

- It is simple.
- It is easy to understand and use.

**Disadvantages**

- It leads to poor utilization of processor and memory.
- Users job is limited to the size of available memory.

**ii. Multiple-partition Allocation**

One of the simplest methods for allocating memory is to divide memory into several fixed sized partitions. There are two variations of this.

- **Fixed Equal-size Partitions**
  It divides the main memory into equal number of fixed sized partitions, operating system occupies some fixed portion and remaining portion of main memory is available for user processes.

  ***Advantages***

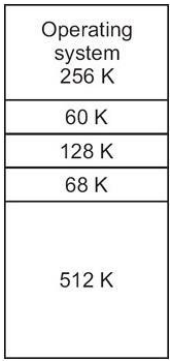
  - Any process whose size is less than or equal to the partition size can be loaded into any available partition.
  - It supports multiprogramming.

  ***Disadvantages***

  - If a program is too big to fit into a partition use overlay technique.
  - Memory use is inefficient, i.e., block of data loaded into memory may be smaller than the partition. It is known as **internal fragmentation**.

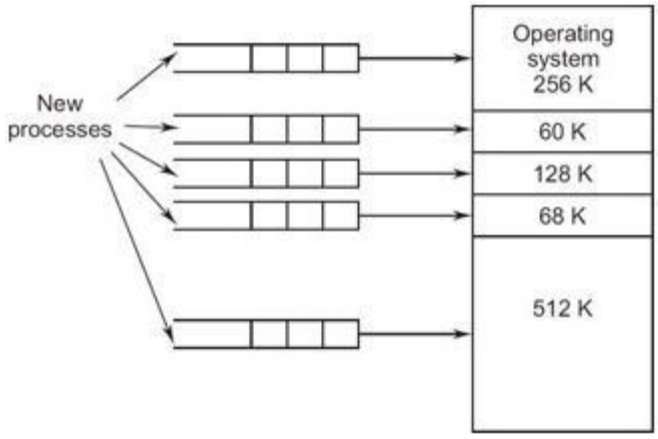
- **Fixed Variable Size Partitions**

  By using fixed variable size partitions we can overcome the disadvantages present in fixed equal size partitioning. This is shown in the figure below:




  With unequal-size partitions, there are two ways to assign processes to partitions.

  - **Use multiple queues:-** For each and every process one queue is present, as shown in the figure below. Assign each process to the smallest partition within which it will fit, by using the scheduling queues, i.e., when a new process arrives, it will be put in the queue it is able to fit without wasting the memory space, irrespective of other blocks queues.



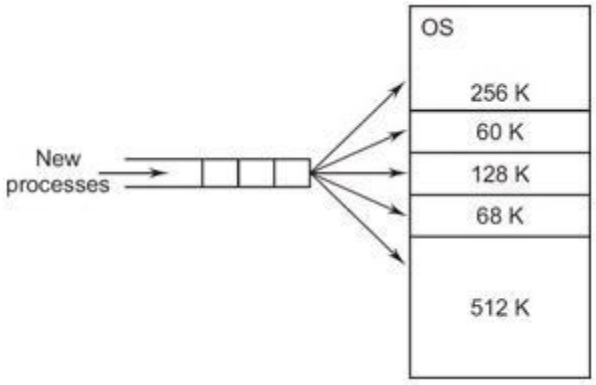

  Advantages

  - Minimize wastage of memory.

  Disadvantages

- This scheme is optimum from the system point of view. Because larger partitions remains unused.

- **Use single queue:-** In this method only one ready queue is present for scheduling the jobs for all the blocks irrespective of size. If any block is free even though it is larger than the process, it will simply join instead of waiting for the suitable block size. It is depicted in the figure below:




Advantages

- It is simple and minimum processing overhead.

Disadvantages

- The number of partitions specified at the time of system generation limits the number of active processes.
- Small jobs do not use partition space efficiently.

**iii. Dynamic Partitioning (Dynamic storage allocation problem)**

Even though when we overcome some of the difficulties in variable sized fixed partitioning, dynamic partitioning require more sophisticated memory management techniques. The partitions used are of variable length. That is when a process is brought into main memory, it allocates exactly as much memory as it requires. Each partition may contain exactly one process. Thus the degree of multiprogramming is bound by the number of partitions. In this method when a partition is free a process is selected from the input queue and is loaded into the free partition. When the process terminates the partition becomes available for another process.

**Fragmentation:**

**Types:**
1. Internal fragmentation

2. External fragmentation

If the free memory is present within a partition then it is called "internal fragmentation". Similarly if the free blocks are present outside the partition, then it is called "external fragmentation". Solution to the "external fragmentation" is **compaction**.

**Compaction**:

- An dynamic allocation approach is to keep a list of unused ( free ) memory blocks ( holes ), and to find a hole of a suitable size whenever a process needs to be loaded into memory. There are many **different strategies for finding the "best" allocation of memory to processes**, including the three most commonly discussed:

    1. **First fit** - It allocates the first free partition or hole which is large enough to accommodate the process. It finishes after finding the first suitable free partition.

        ***Advantage***

        Fastest algorithm because it searches as little as possible.

        ***Disadvantage***

        The remaining unused memory areas left after allocation become waste if it is too smaller. Thus request for larger memory requirement cannot be accomplished.

    ***2.*** **Best fit** - The best fit deals with allocating the smallest free partition which meets the requirement of the requesting process. This algorithm first searches the entire list of free partitions and considers the smallest hole that is adequate.

        ***Advantage***

        Memory utilization is much better than first fit as it searches the smallest free partition first available.

        ***Disadvantage***

        It is slower and may even tend to fill up memory with tiny useless holes.

3. **Worst fit** - It locates the largest available free portion so that the portion left will be big enough to be useful. It is the reverse of best fit.

   ***Advantage***

   Reduces the rate of production of small gaps.

   ***Disadvantage***

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

- The first or best fit are better than worst fit in terms of both time and storage utilization. First and best fits are about equal in terms of storage utilization, but first fit is faster.

**Exercise on dynamic memory allocation:**

1. 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?

**First-fit:**
212K is put in 500K partition
417K is put in 600K partition
112K is put in 200K partition
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 300K partition
426K must wait

| | | | | | |
|---|---|---|---|---|---|
| **First-fit** | | 212 | 112 | | 417 |
| **Best-fit** | | 417 | 112 | 212 | 426 |
| **Worst-fit** | | 417 | | 112 | 212 |
| **Memory blocks** | **100KB** | **500KB** | **200KB** | **300KB** | **1000KB** |

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

2. Given memory partitions of 200K, 425K, 100K, 700K, and 300K (in order), how would each of the First-fit, Best-fit, and Worst-fit algorithms place processes of 212K, 417K, 112K, and 426K (in order)? Which algorithm makes the most efficient use of memory?

**3. Fragmentation and Compaction**

- All the memory allocation strategies suffer from ***external fragmentation.*** External fragmentation occurs when total free RAM space is enough to load a process, but the process still cannot be loaded, because free blocks of RAM are not contiguous.
- ***Internal fragmentation*** also occurs with all memory allocation strategies when fixed sized memory blocks are available, whereas the process gets a block that is too much larger than the storage requirement of a process.
- If the programs in memory are relocatable, then the external fragmentation problem can be reduced via ***compaction***, i.e. moving all processes down to one end of physical memory. This only involves updating the relocation register for each process.
- Another solution is to allow processes to use **non-contiguous blocks of physical memory**, with a separate relocation register for each block.

# Paging

**Need for Paging:**

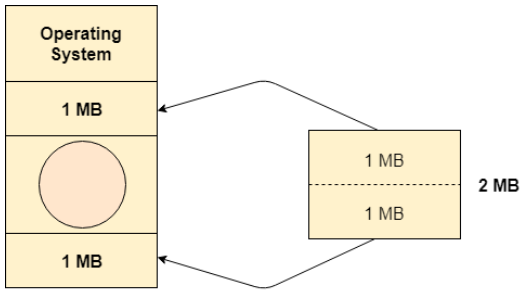
Let us consider a process P1 of size 2 MB and the main memory which is divided into three partitions. Out of the three partitions, two partitions are holes of size 1 MB each.

P1 needs 2 MB space in the main memory to be loaded. We have two holes of 1 MB each but they are not contiguous.

Although, there is 2 MB space available in the main memory in the form of those holes but that remains useless until it become contiguous.

We need to have some kind of mechanism which can store one process at different locations of the memory.

The Idea behind paging is to divide the process (ie. process address space) into blocks of same size called **pages** (size is power of 2, The size of the process is measured in the number of pages) so that, we can store them in the memory at different holes.




The process needs to be divided into two parts to get stored at two different places.

**Paging:**

Paging is a memory management scheme that eliminates the need for contiguous allocation of physical memory. This scheme permits the **physical address space** of a process to be **non – contiguous.**

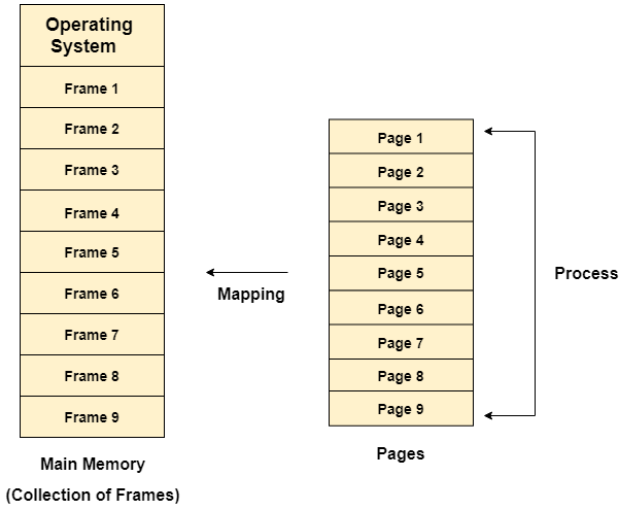
In Operating Systems, Paging is a storage mechanism used to retrieve processes from the secondary storage into the main memory in the form of pages.

The main idea behind the paging is to **divide each process in the form of pages**. The **main memory will also be divided in the form of frames**.

One page of the process is to be stored in one of the frames of the memory. The pages can be stored at the different locations of the memory but the priority is always to find the contiguous frames or holes.

Pages of the process are brought into the main memory only when they are required otherwise they reside in the secondary storage.

Considering the fact that the pages are mapped to the frames in Paging, **page size needs to be as same as frame size.**



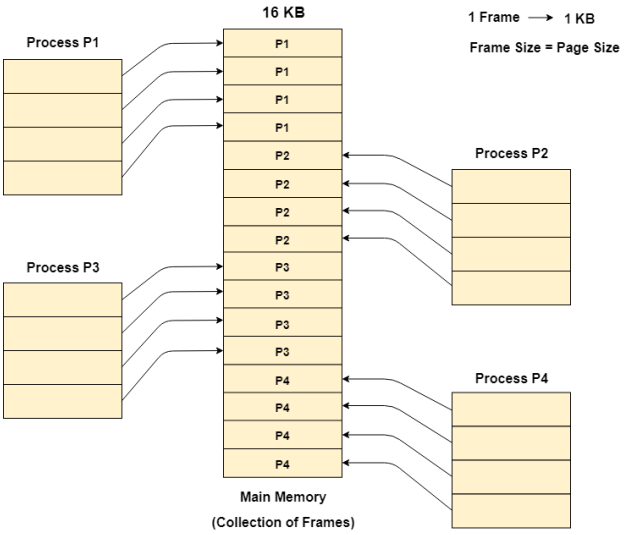

**Example**

Let us consider the main memory size 16 Kb and Frame size is 1 KB therefore the main memory will be divided into the collection of 16 frames of 1 KB each.

There are 4 processes in the system that is P1, P2, P3 and P4 of 4 KB each. Each process is divided into pages of 1 KB each so that one page can be stored in one frame.

Initially, all the frames are empty therefore pages of the processes will get stored in the contiguous way.

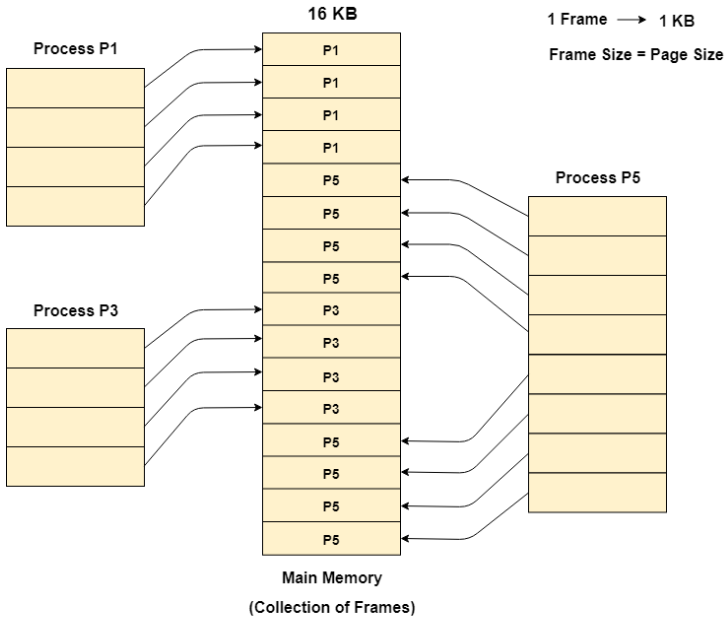
Frames, pages and the mapping between the two is shown in the image below.




Paging

Let us consider that, P2 and P4 are moved to waiting state after some time. Now, 8 frames become empty and therefore other pages can be loaded in that empty place. The process P5 of size 8 KB (8 pages) is waiting inside the ready queue.

Given the fact that, we have 8 non contiguous frames available in the memory and paging provides the flexibility of storing the process at the different places. Therefore, we can load the pages of process P5 in the place of P2 and P4.




Paging

## 1. Basic Method

- The basic idea behind paging is to divide physical memory into a number of equal sized blocks called ***frames,*** and to divide a programs logical memory space into blocks of the same size called ***pages.***
- Any page ( from any process ) can be placed into any available frame.
- The ***page table*** is used to look up what frame a particular page is stored in at the moment. The **page table is stored in physical memory**. In the following example, for instance, page 2 of the program's logical memory is currently stored in frame 3 of physical memory:

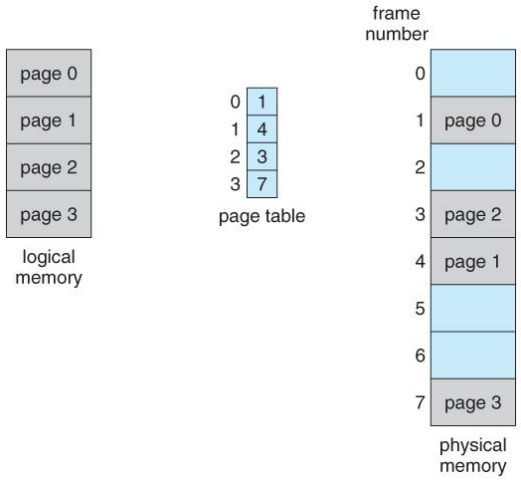



**Figure - Paging model of logical and physical memory**

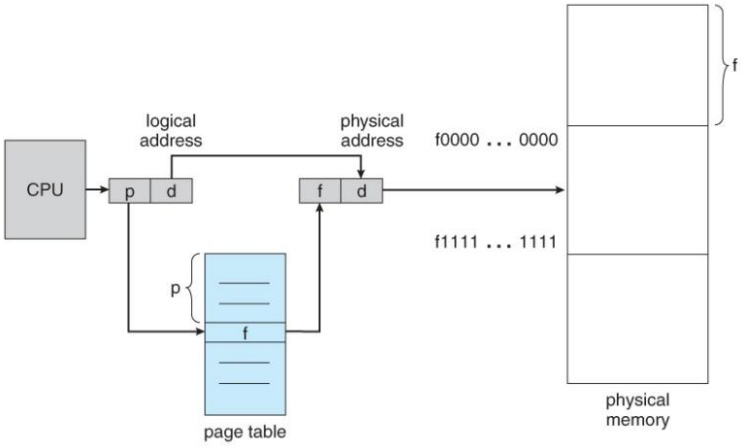



**Figure - Paging hardware**

- A **logical address consists of two parts**: A **page number** in which the address resides, and an **offset from the beginning of that page**.
- The **page table *maps the page number to a frame number***, to yield a **physical address which has two parts: The frame number and the offset within that frame.**
- Page numbers, frame numbers, and frame sizes are typically **powers of two**, allowing addresses to be split at a certain number of bits. For example, if the logical address size is **2^m** and the page size is **2^n,** then the **high-order m-n bits of a logical address** designate the **page number** and the **remaining n bits represent the offset**.


- Consider the following micro example, in which a process has 16 bytes of logical memory, mapped in 4 byte pages into 32 bytes of physical memory.

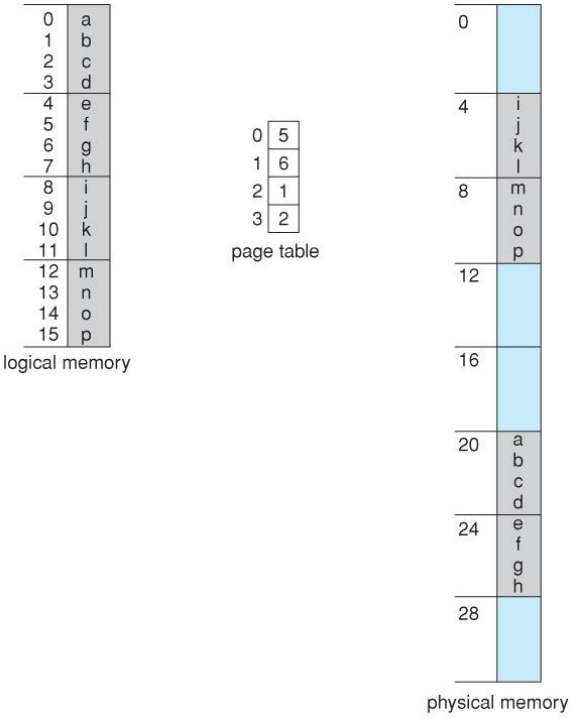
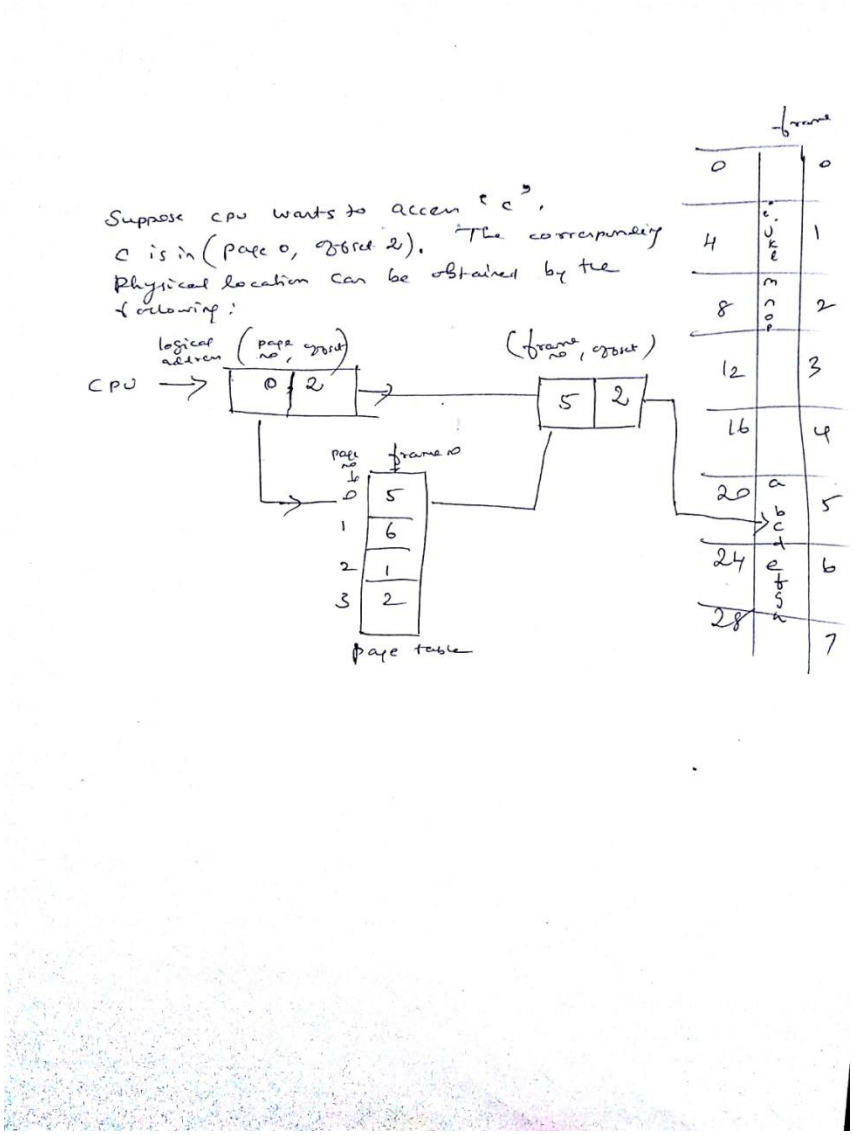



**Figure - Paging example for a 32-byte memory with 4-byte pages**




**Eg: Logical address 2 has (pageno 0, offset 2). By referring to the page table, pageno 0 is in frame no 5. The corresponding physical address is**

=(frame no * page size) +offset

=(5*4)+2

=22

- There is **no external fragmentation with paging**. All blocks of physical memory are used, and there are no gaps in between and no problems with finding the right sized hole for a particular chunk of memory.

- There is, **however, internal fragmentation**. Memory is allocated in chunks the size of a page, and on the average, the last page will only be half full, wasting on the average half a page of memory per process.
- When a process requests memory, free frames are allocated from a free-frame list, and inserted into that process's page table.
- Processes are blocked from accessing anyone else's memory because all of their memory requests are mapped through their page table. There is no way for them to generate an address that maps into any other process's memory space.

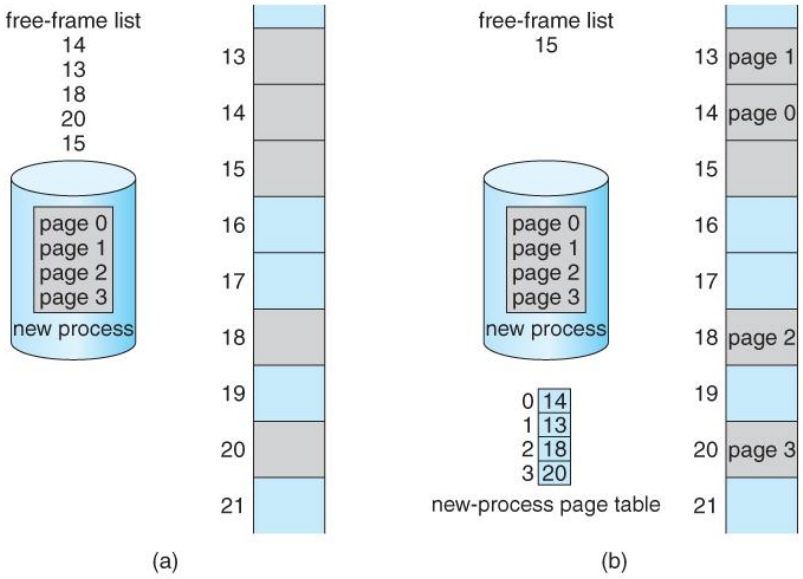



**Figure - Free frames (a) before allocation and (b) after allocation**

## 2. Hardware Support (TLB)

- Page lookups must be done for every memory reference, and whenever a process gets swapped in or out of the CPU, its page table must be swapped in and out too, along with the instruction registers, etc. It is therefore appropriate to provide hardware support for this operation, in order to make it as fast as possible and to make process switches as fast as possible also.
    - A very special high-speed memory device called the ***translation look-aside buffer (TLB)*** *is used*

- The benefit of the TLB is that it can **search an entire table for a key value in parallel**, and if it is found anywhere in the table, then the corresponding lookup value is returned.

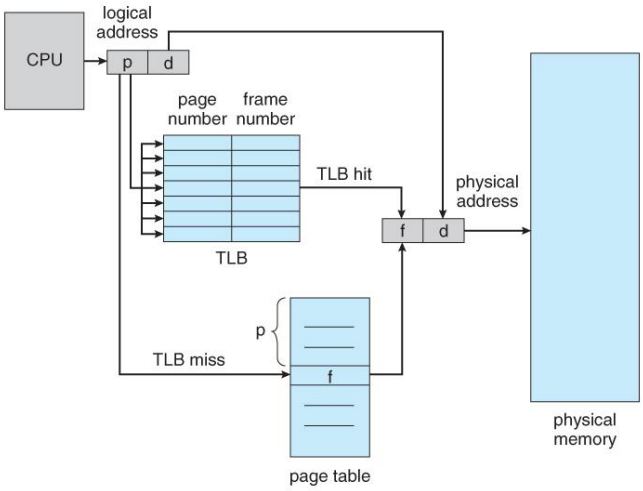



**Figure - Paging hardware with TLB**

- The TLB is very **expensive**, however, **very small**. ( Not large enough to hold the entire page table. ) It is therefore **used as a cache device**.
    - Addresses are first checked against the TLB, and if the info is not there **( a TLB miss** ), then the frame is looked up from main memory and the TLB is updated.
    - If the TLB is full, then replacement strategies range from ***least-recently used, LRU*** to random.
- The **percentage of time that the desired information is found in the TLB** is termed the ***hit ratio***.

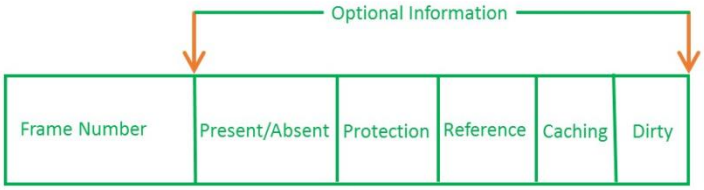
**Exercise:**

- For example, suppose that it takes 100 nanoseconds to access main memory, and only 20 nanoseconds to search the TLB. So a **TLB hit takes 120 nanoseconds**, and a **TLB miss takes 220 nanoseconds,** So with an **80% TLB hit ratio**, the **average memory access time** would be:

  0.80 * **120** + 0.20 * **220** = 140 nanoseconds

### 3. Protection

- The page table can also help to protect processes from accessing memory that they shouldn't, or their own memory in ways that they shouldn't.
- A **bit or bits can be added to the page table** to **classify a page as read-write, read-only, read-write-execute**, or some combination of these sorts of things. Then each memory reference can be checked **to ensure it is accessing the memory in the appropriate mode.**
- Page table has **page table entries** where each page table entry stores a frame number and optional status bits. The following figure shows page table entries:



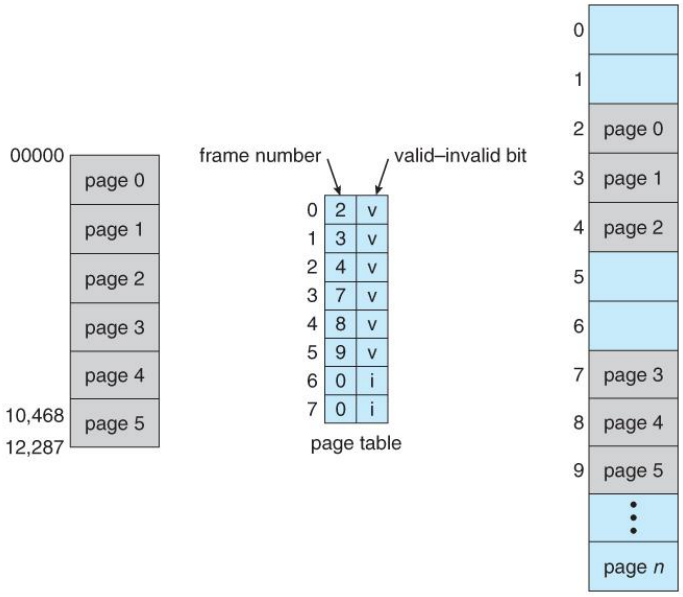




**Figure - Valid (v) or invalid (i) bit in page table**

1. **Frame Number** – It gives the frame number in which the current page you are looking for is present.

2. **Present/Absent bit (valid/invalid bits)** – Present or absent bit says whether a particular page you are looking for is present or absent. In case 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**.

3. **Protection bit** – Protection bit says that **what kind of protection** you want on that page. So, these bit for the protection of the page frame (**read, write etc).**

4. **Referenced bit** – Referenced bit will say whether **this page has been referred in the last clock cycle or not**. It is set to 1 by hardware when the page is accessed.

5. **Modified bit** (**Dirty bit**)– Modified bit says **whether the page has been modified or not**. Modified means sometimes you might try to write something on to the page. If a page is modified, then whenever you should replace that page with some other page, then the modified information should be kept on the hard disk or it has to be written back or it has to be saved back. **It is set to 1 by hardware on write-access to page** which is used to avoid writing when swapped out. Sometimes this modified bit is also called as the **Dirty bit**.

**4. Shared Pages**

- Paging systems can make it very easy to **share blocks of memory**, by **simply duplicating page numbers in multiple page frames**. This may be done with either code or data.

- If code is ***reentrant***, that means that it does not write to or change the code in any way ( it is non self-modifying ), and it is therefore safe to re-enter it. More importantly, it means the **code can be shared by multiple processes, so long as each has their own copy of the data and registers**.

- In the example given below, three different users are running the editor simultaneously, but the code is only loaded into memory ( in the frames ) one time.

- Some systems also implement shared memory in this fashion.

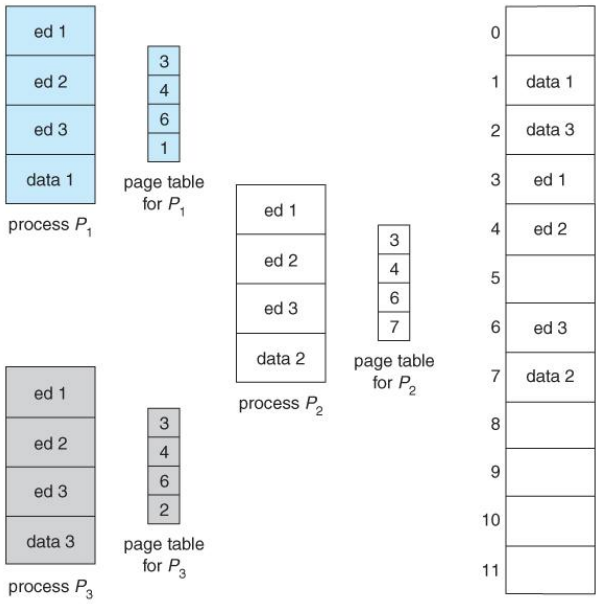
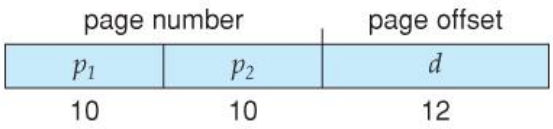



**Figure - Sharing of code in a paging environment**

## 6. Structure of the Page Table

### 6.1 Hierarchical Paging

- Let us assume, we have a memory address of 32 bits. The 32 bit address is divided into two segments, which is 20 bits for page no, 12 bits for page offset.
- As page offset is 12, the page size is $2^{12}= 2^2 * 2^{10}= 4* 1024= 4096$ . ie., each page size is 4096 bits= 4KB.
- 20 bit page number can have combination $2^{10}* 2^{10}= 1K * 1K= 1$ MB. so, 1 Mega number of locations are possible. ie, we need 1MB of locations (contiguous locations) to store a page table. But, getting 1 MB continuous free memory locations may not be possible sometimes. Also, takes more time to search the page table. One option is to use a **two-tier paging system**, i.e. **to page the page table**.
- Hence, we suggest not to have 20 bits for the page table, hence divide it into 10 bits for page table P1 and another 10 bits for page table P2. Hence each page table size would become $2^{10}= 1$KB.



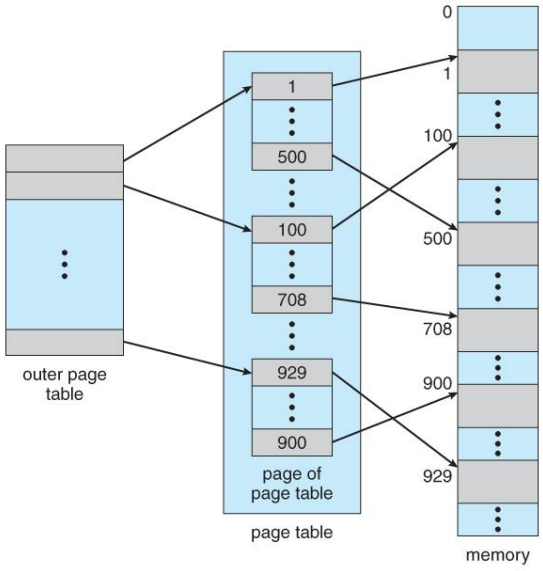



**Figure A two-level page-table scheme**

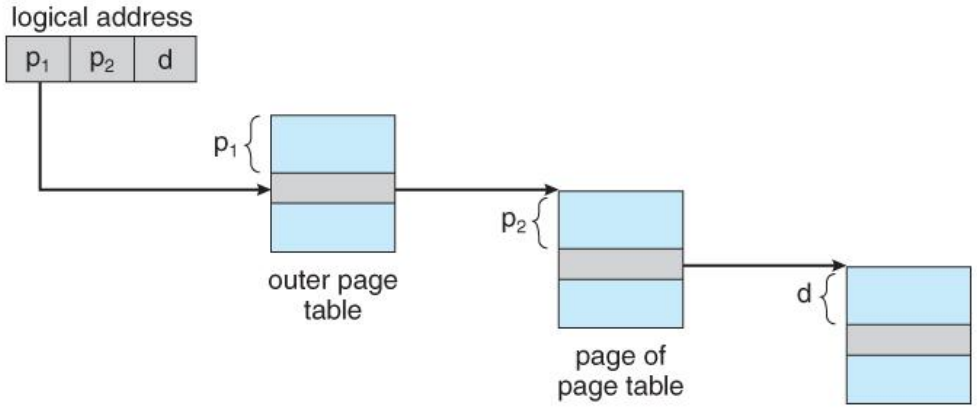



**Figure - Address translation for a two-level 32-bit paging architecture**

## 6.2 Hashed Page Tables

- One common data structure for accessing data that is sparsely distributed over a broad range of possible values is with ***hash tables*.** Figure below illustrates a *hashed page table* using chain-and-bucket hashing:

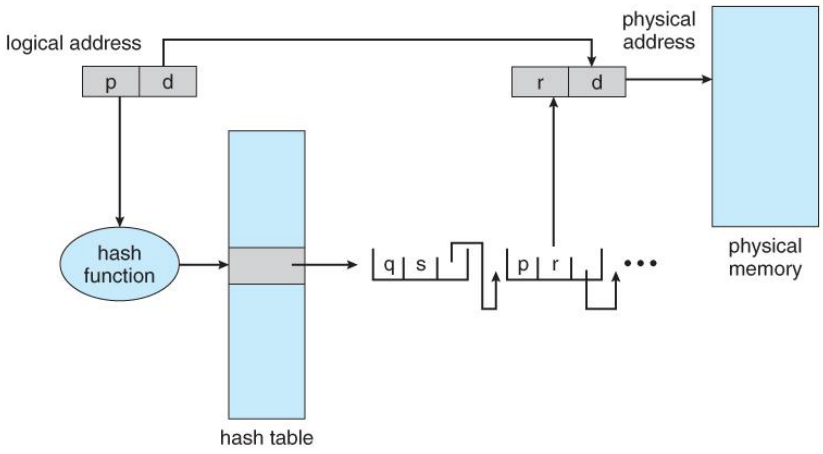



**Figure - Hashed page table**

### 6.3 Inverted Page Tables

- Another approach is to use an *inverted page table*. Our system is a multitasking system, with multiple processes are running at the same time. Hence. if we want to access 10th page and 4th offset, the system cannot identify, which process's 10th page and 4th offset the user is referring to. Hence we need to attach process id with the logical address. For example the logical address contains (pid, page no, offset). The pid and page no are used collectively to refer to particular process frame no.

- Access to an inverted page table can be **slow**, as it may be necessary to **search the entire table in order to find the desired page**. **Hashing the table can help speedup the search process.**

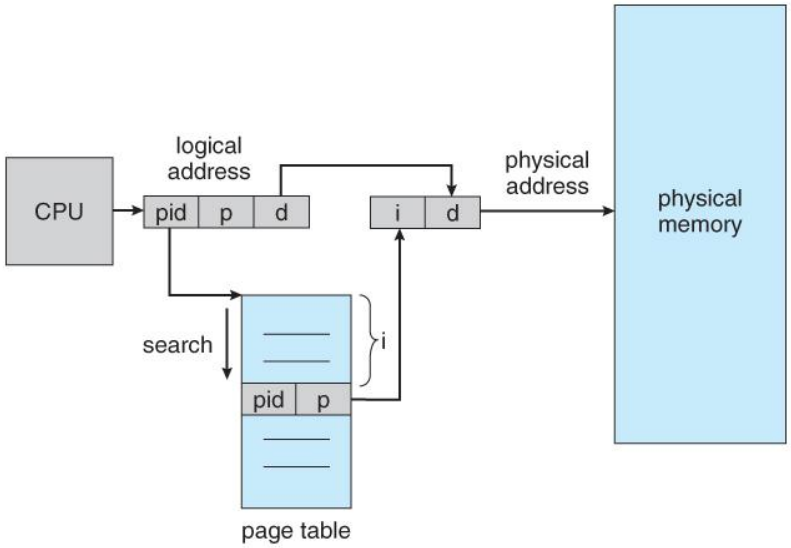



**Figure - Inverted page table**

**Advantages of Paging**

- Paging **reduces external fragmentation**.
- Paging is **simple to implement** and assumed as an efficient memory management technique.
- Due to **equal size of the pages and frames**, **swapping becomes very easy**.

**Disadvantages of Paging**

- Though paging reduces external fragmentation, still **suffer from internal fragmentation**.
- **Page table requires extra memory space**, so may not be good for a system having small RAM.

# Virtual Memory

## Basics of Virtual Memory:

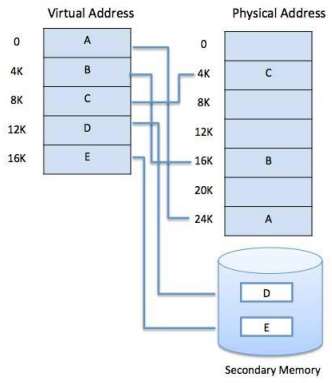
A computer can address more memory than the amount physically installed on the system. This extra memory is actually called **virtual memory** and it is a **section of a hard disk** that's set up to emulate the computer's RAM.

The main visible advantage of this scheme is that **programs can be larger than physical memory**.

***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.
- **Less number of I/O** would be needed to **load or swap** each user program into memory.
- 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**.

Modern microprocessors use a memory management unit, or MMU, which is built into the hardware. The MMU's job is to translate virtual addresses into physical addresses. A basic example is given below −




## Demand Paging

Virtual memory is commonly implemented by demand paging.

A demand paging system is quite similar to a **paging system with swapping,** where processes reside in secondary memory and pages are loaded only on demand, not in advance.

## 1. Background

- To avoid (external) memory fragmentation, paging is breaking process memory requirements into pages and is storing the pages non-contiguously in memory. However the entire process still had to be stored in memory somewhere.
- In practice, most real processes do not need all their pages, or not all at once, for several reasons:
    1. Error handling code is not needed unless that specific error occurs, some of which are quite rare.
    2. Only a small fraction of the arrays are actually used in practice.
    3. Certain features of certain programs are rarely used
- The ability to load only the portions of processes that were actually needed has several benefits:
    - Programs could be written for a much larger address space ( virtual memory space ) than physically exists on the computer.
    - Because each process is only using a fraction of their total address space, there is more memory left for other programs, improving CPU utilization and system throughput.
    - Less I/O is needed for swapping processes in and out of RAM, speeding things up.

  Figure below shows the general layout of ***virtual memory***, which can be much larger than physical memory:

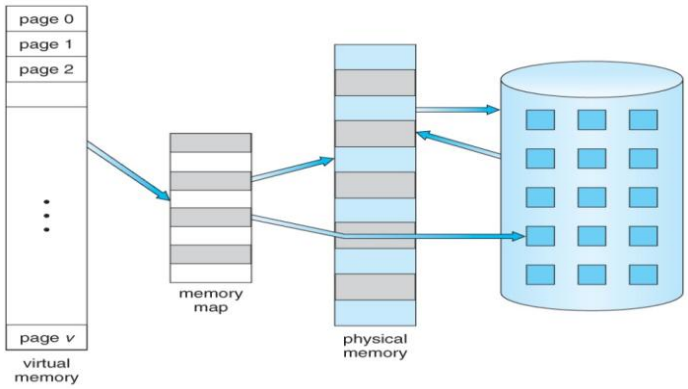



**Figure - Diagram showing virtual memory that is larger than physical memory**

- Figure below shows ***virtual address space***, which is the programmers logical view of process memory storage. The actual physical layout is controlled by the process's page table.
- Note that the address space shown in Figure is ***sparse*** - A great hole in the middle of the address space is never used, unless the stack and/or the heap grow to fill the hole.

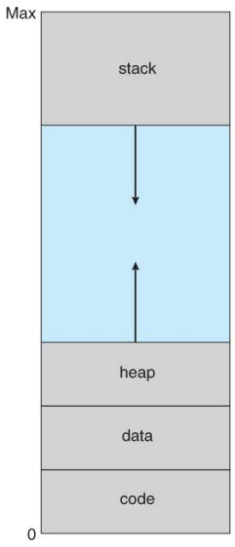



**Figure 9.2 - Virtual address space**

## 2. Demand Paging

- The basic idea behind ***demand paging*** is that when a process is swapped in, its pages are not swapped in all at once. Rather they are swapped in only when the process needs them. (on demand) This is termed a ***lazy swapper***, although a ***pager*** is a more accurate term.

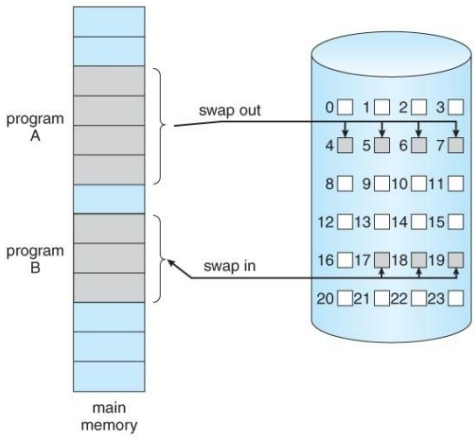



**Figure - Transfer of a paged memory(program A) to contiguous disk space**

## 2.1 Basic Concepts

- The basic idea behind paging is that when a process is swapped in, the pager only loads into memory those pages that the process currently needs.
- **Pages that are not loaded** into memory are **marked as invalid in the page table**, using the invalid bit.
- If the process only accesses pages that are loaded in memory ( ***memory resident*** pages), then the process runs exactly as if all the pages were loaded in to memory.

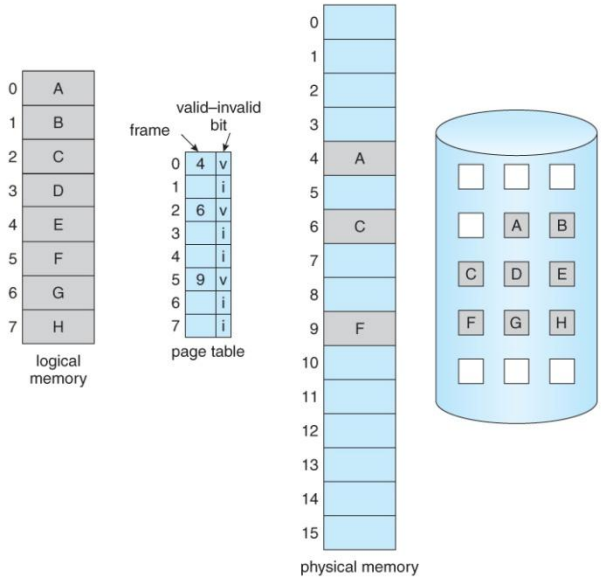



**Figure - Page table when some pages are not in main memory.**

- On the other hand, if a page is needed that was not originally loaded up, then a ***page fault trap*** is generated, which must be **handled in a series of steps**:
    1. The memory address requested is first checked, to make sure it was a valid memory request.
    2. If the reference was invalid, the process is terminated. Otherwise, the page must be paged in.
    3. A free frame is located, possibly from a free-frame list.
    4. A disk operation is scheduled to bring in the necessary page from disk.
    5. When the I/O operation is complete, the process's page table is updated with the new frame number, and the invalid bit is changed to indicate that this is now a valid page reference.
    6. The instruction that caused the page fault must now be restarted from the beginning.

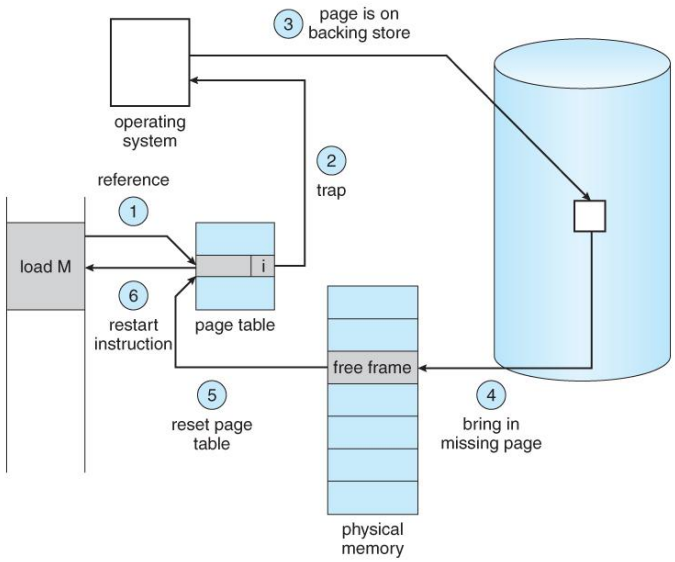



**Figure - Steps in handling a page fault**

- In an extreme case, **NO pages are swapped in** for a process **until they are requested** by **page faults**. This is known as ***pure demand paging.***
- In theory, each instruction could generate multiple page faults. In practice this is very rare, due to ***locality of reference***.
- The hardware necessary to support virtual memory is the same as for paging and swapping: A page table and secondary memory.

## 2.2 Performance of Demand Paging

- There is some **slowdown and performance hit** whenever **a page fault occurs** and the system has to go get it from secondary memory
- There are many steps that occur when servicing a page fault, and some of the steps are optional or variable. Suppose that a normal memory access requires 200 nanoseconds, and that servicing **a page fault takes 8 milliseconds**. ( 8,000,000 nanoseconds ) With a ***page fault rate*** of p, ( on a scale from 0 to 1 ), the **effective access time** is now:

  ( 1 - p ) * ( 200 ) + p * 8000000

  = 200 + 7,999,800 * p

  which ***clearly*** depends heavily on p!

## 2.3 Advantages of demand paging

- Large virtual memory.
- More efficient use of memory.
- There is no limit on degree of multiprogramming.

**Disadvantages**

- The amount of processor overhead for handling page interrupts are greater than in the case of the simple paged management techniques.

## 2.4 Copy-on-Write

- Copy-on-write finds its main use in virtual memory operating systems; when a process creates a copy of itself, the pages in memory that are modified by the process are marked copy-on-write.
- Copy-on-write is an optimization strategy designed **to save memory**. it sounds like: everyone has a single shared copy of the same data *until it's written*, and then a copy is made.

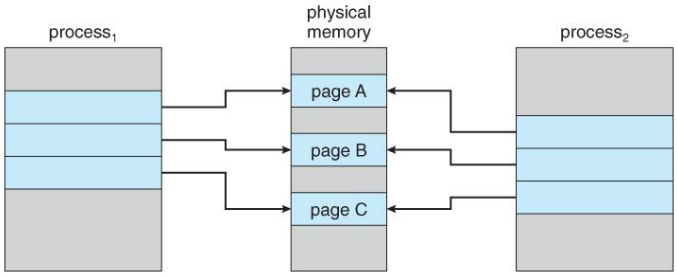



**Figure - Before process 1 modifies page C.**

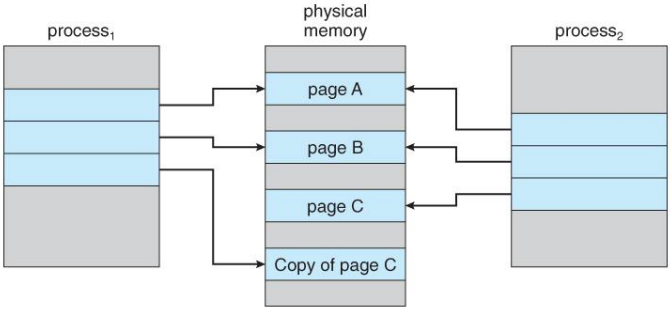



**Figure - After process 1 modifies page C.**

- **fork() with copy-on-write:**

  Copy-on-Write allows both parent and child processes to initially share the same pages in memory. If either process modifies a shared page, only then is the page copied.

  After fork, a child is copied by parent process. since they use same virtual address, and their virtual addresses map to the same physical addresses, they can share read only memory.

  When child would like to write data to memory, OS would allocate a new frame, and copy the data of that frame to the new frame, then child process can write data to this new frame. so for that page, although parent and child process has the same virtual address, the mapping physical address is different.

- advantage: save space and the time of allocating whole space.
- disadvantage: complex and has overhead.

- **note:**

  even if one byte of a frame is modified, it will cause copy-on-write.

## Locality of reference:

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.

***Types:***

**1. Temporal locality:**

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**.

**2. Spatial locality:**

If at one point a particular storage location is referenced, then it is likely that **nearby memory locations will be referenced in the near future.**

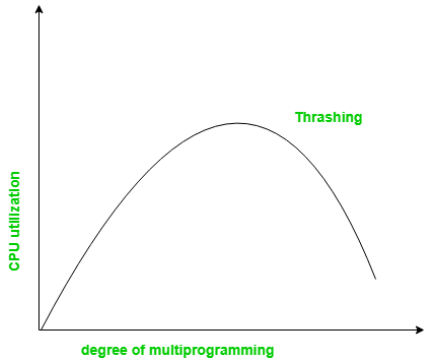
---

## Thrashing:

What happens if memory gets overcommitted?

- Suppose the pages being actively used by the current threads don't all fit in physical memory.
- Each page fault causes one of the active pages to be moved to disk, so another page fault will occur soon.
- The system will spend all its time reading and writing pages, and won't get much work done.
- This **situation is called *thrashing***; it was a serious problem in early demand paging systems.

**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.



The basic concept involved is that if a process is allocated too few frames, then there will be too many and too frequent page faults. As a result, no useful work would be done by the CPU and the CPU utilisation would fall drastically. The long-term scheduler would then try to improve the CPU utilisation by loading some more processes into the memory thereby increasing the degree of multiprogramming. This would result in a further decrease in the CPU utilization triggering a chained reaction of higher page faults followed by an increase in the degree of multiprogramming, called Thrashing.

**How to deal with thrashing?**

- If a single process is too large for memory, there is nothing the OS can do. That process will simply thrash.
- If the problem arises because of the sum of several processes:
  Figure out how much memory each process needs.
  Change scheduling priorities to run processes in groups that fit comfortably in memory: must shed load.

***1. Working Sets*:** conceptual model **to prevent thrashing**.

- Informal definition: the collection of pages a process is using actively, and which must thus be memory-resident to prevent this process from thrashing.
- If the sum of all working sets of all runnable threads exceeds the size of memory, then stop running some of the threads for a while.
- Divide processes into two groups: **active and inactive**:
  - When a process is active its entire working set must always be in memory: never execute a thread whose working set is not resident.
  - When a process becomes inactive, its working set can migrate to disk.
  - Threads from inactive processes are never scheduled for execution.
  - The collection of active processes is called the *balance set*.
  - The system must have a mechanism for gradually moving processes into and out of the balance set.
  - As working sets change, the balance set must be adjusted.

- **How to compute working sets?**
  - Let working set parameter be T: **all pages referenced in the last T seconds comprise the working set.**

***2. Page Fault Frequency*:** another approach **to preventing thrashing**.

- Per-process replacement; at any given time, each process is allocated a fixed number of physical page frames.
- Monitor the **rate at which page faults are occurring** for each process.
- If the **rate gets too high** for a process, assume that its **memory is overcommitted**; **increase the size** of its memory pool.
- If the **rate gets too low** for a process, assume that its **memory pool can be reduced in size**.
- If the sum of all memory pools don't fit in memory, deactivate some processes.

## Page replacement algorithms

### 1. Page Replacement

- Since we only load the pages that are actually needed by each process at any given time, there is room to load many more processes.
- What happens if some process suddenly decides it needs more pages and there aren't any free frames available? There are several possible solutions to consider:
    1. Put the process requesting more pages into a **wait** queue until some free frames become available.
    2. **Swap** some process out of memory completely, freeing up its page frames.
    3. Find some page in memory that **isn't being used right now**, and **swap that page** only out to disk, **freeing up a frame** that can be allocated to the process requesting it. This is known as ***page replacement***. There are many different algorithms for page replacement.

### 1.1 Basic Page Replacement

- The page-fault handling must be modified to free up a frame if necessary, as follows:
    1. Find the location of the desired page on the disk
    2. Find a free frame:
        a. If there is a free frame, use it.
        b. If there is no free frame, use a page-replacement algorithm to select an existing frame to be replaced, known as the ***victim frame***.
        c. Write the victim frame to disk. Change all related page tables to indicate that this page is no longer in memory.
    3. Read in the desired page and store it in the frame. Adjust all related page and frame tables to indicate the change.
    4. Restart the process that was waiting for this page.

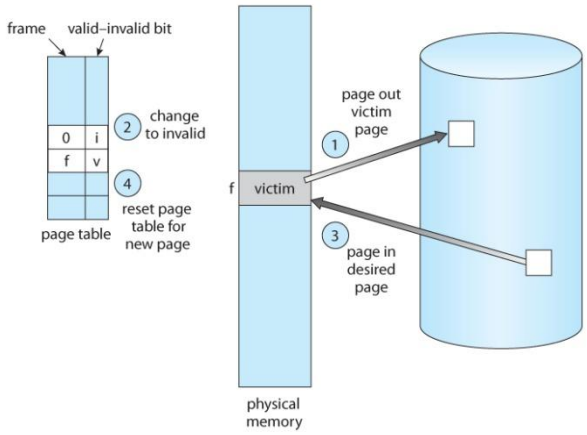



**Figure - Page replacement.**

- Note that step 2c adds an extra disk write to the page-fault handling, effectively doubling the time required to process a page fault. This can be alleviated somewhat by assigning a ***modify bit,*** or ***dirty bit*** to each page, indicating whether or not it has been changed since it was last loaded in from disk. If the dirty bit has not been set, then the page is unchanged, and does not need to be written out to disk. Otherwise the page write is required.
- To implement a successful demand paging system ***page-replacement algorithm can be used.*** It deals with how to select a page for replacement when there are no free frames available.
- Algorithms are evaluated using a given string of memory accesses known as a ***reference string***

**Types of Page Replacement algorithms:**

**1. FIFO Page Replacement**

- A simple page replacement strategy is ***FIFO***, i.e. first-in-first-out.
- As new pages are brought in, they are added to the tail of a queue, and the page at the head of the queue is the next victim. In the following example, for 3 frames, 20 page requests result in 15 page faults:

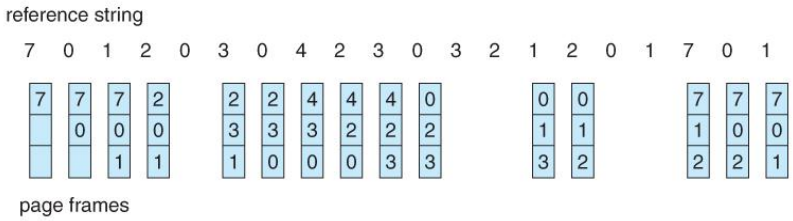



**Figure - FIFO page-replacement algorithm.**

- Although FIFO is simple and easy, it is not always optimal, or even efficient.

- An interesting effect that can occur with FIFO is ***Belady's anomaly***, in which **increasing the number of frames available can actually *increase* the number of page faults that occur**

**2. Optimal Page Replacement (*OPT or MIN*)**

- The discovery of Belady's anomaly lead to the search for an ***optimal page-replacement algorithm***, which is simply that which yields the lowest of all possible page-faults, and which does not suffer from Belady's anomaly.
- Such an algorithm does exist, and is called ***OPT or MIN.*** This algorithm is simply "Replace the page that **will not be used for the longest time** in the future."
- Figure illustrates OPT for our sample string, yielding 9 page faults, ( as compared to 15 for FIFO )

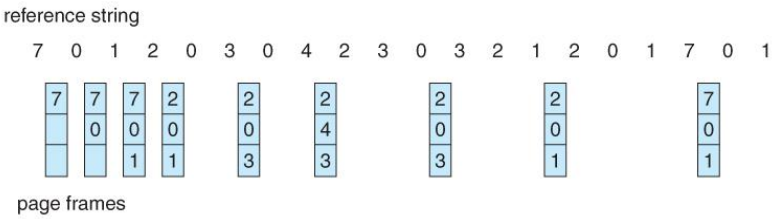



**Figure - Optimal page-replacement algorithm**

**3. LRU Page Replacement**

- The prediction behind ***LRU,*** the ***Least Recently Used,*** algorithm is that the page that **has not been used in the longest time** is the one that will not be used again in the near future.
- Figure illustrates LRU for our sample string, yielding 12 page faults, ( as compared to 15 for FIFO and 9 for OPT. )

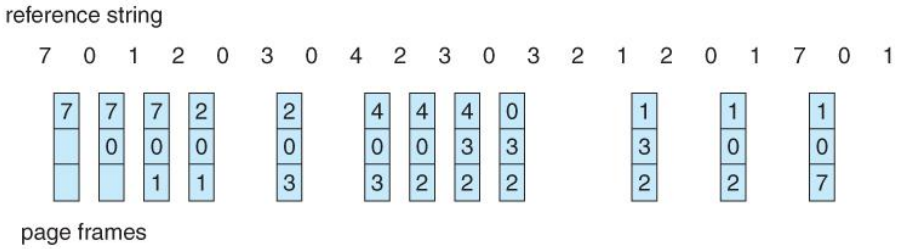



**Figure - LRU page-replacement algorithm.**

- LRU is considered a good replacement policy, and is often used. The problem is how exactly to implement it. There are two simple approaches commonly used:

1. **Counters.** Every memory access increments a counter, and the current value of this counter is stored in the page table entry for that page. Then finding the LRU page involves simple searching the table for the page with the smallest counter value.
2. **Stack.** Another approach is to use a stack, and whenever a page is accessed, pull that page from the stack and place it on the top. The LRU page will always be at the bottom of the stack.

## 4. LRU-Approximation Page Replacement

- Unfortunately full implementation of LRU requires hardware support, and few systems provide the full hardware support necessary.
- However many systems offer some degree of HW support, enough to approximate LRU fairly well.
- In particular, many systems provide a ***reference bit*** for every entry in a page table, which is set anytime that page is accessed. Initially all bits are set to zero, and they can also all be cleared at any time. One bit of precision is enough to distinguish pages that have been accessed since the last clear from those that have not, but does not provide any finer grain of detail.

### 4.1 Additional-Reference-Bits Algorithm

- Finer grain is possible by storing the most recent 8 reference bits for each page in an 8-bit byte in the page table entry, which is interpreted as an unsigned int.
    - At periodic intervals ( clock interrupts ), the OS takes over, and right-shifts each of the reference bytes by one bit.
    - The high-order ( leftmost ) bit is then filled in with the current value of the reference bit, and the reference bits are cleared.
    - At any given time, the page with the smallest value for the reference byte is the LRU page.
- Obviously the specific number of bits used and the frequency with which the reference byte is updated are adjustable, and are tuned to give the fastest performance on a given hardware platform.

### 4.2 Second-Chance Algorithm

- The ***second chance algorithm*** is essentially a FIFO, except the reference bit is used to give pages a second chance at staying in the page table.
    - When a page must be replaced, the page table is scanned in a FIFO ( circular queue ) manner.
    - If a page is found with its reference bit not set, then that page is selected as the next victim.

- If, however, the next page in the FIFO **does** have its reference bit set, then it is given a second chance:
  - The reference bit is cleared, and the FIFO search continues.
  - If some other page is found that did not have its reference bit set, then that page will be selected as the victim, and this page ( the one being given the second chance ) will be allowed to stay in the page table.
  - If , however, there are no other pages that do not have their reference bit set, then this page will be selected as the victim when the FIFO search circles back around to this page on the second pass.
- If all reference bits in the table are set, then second chance degrades to FIFO, but also requires a complete search of the table for every page-replacement.
- As long as there are some pages whose reference bits are not set, then any page referenced frequently enough gets to stay in the page table indefinitely.
- This algorithm is also known as the ***clock*** algorithm, from the hands of the clock moving around the circular queue.

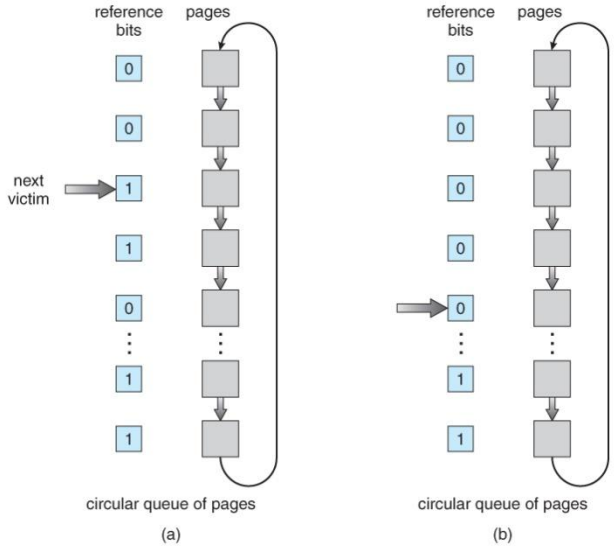



**Figure - Second-chance ( clock ) page-replacement algorithm.**

5. **NRU(Not Recently Used) Page Replacement Algorithm**

This algorithm requires that each page have two additional **status bits 'R' and 'M' called reference bit and change bit** respectively. The reference bit(R) is automatically set to 1 whenever the page is referenced. The change bit (M) is set to 1 whenever the page is modified. These bits are stored in the PMT and are updated on every memory reference. When a page fault occurs, the memory manager inspects all the pages and divides them into 4 classes based on R and M bits.

- **Class 1: (0,0)** − neither recently used nor modified - the best page to replace.

- **Class 2: (0,1)** − not recently used but modified - the page will need to be written out before replacement.
- **Class 3: (1,0)** − recently used but clean - probably will be used again soon.
- **Class 4: (1,1)** − recently used and modified - probably will be used again, and write out will be needed before replacing it.

  This algorithm searches the page table in a circular fashion, looking for the first page it can find in the lowest numbered category. I.e. it first makes a pass looking for a ( 0, 0 ), and then if it can't find one, it makes another pass looking for a ( 0, 1 ), etc.

Advantages:

- It is easy to understand.
- It is efficient to implement.