RGPVNOTES.IN

Subject Name: **Computer Architecture**

Subject Code: **IT-4005**

Semester: **4th**

UNIT-IV Computer memory system

Memory hierarchy

A memory unit is the collection of storage units or devices together. The memory unit stores the binary information in the form of bits. Generally, memory/storage is classified into 2 categories:

- **Volatile Memory**: This loses its data, when power is switched off.
- **Non-Volatile Memory**: This is a permanent storage and does not lose any data when power is switched off.

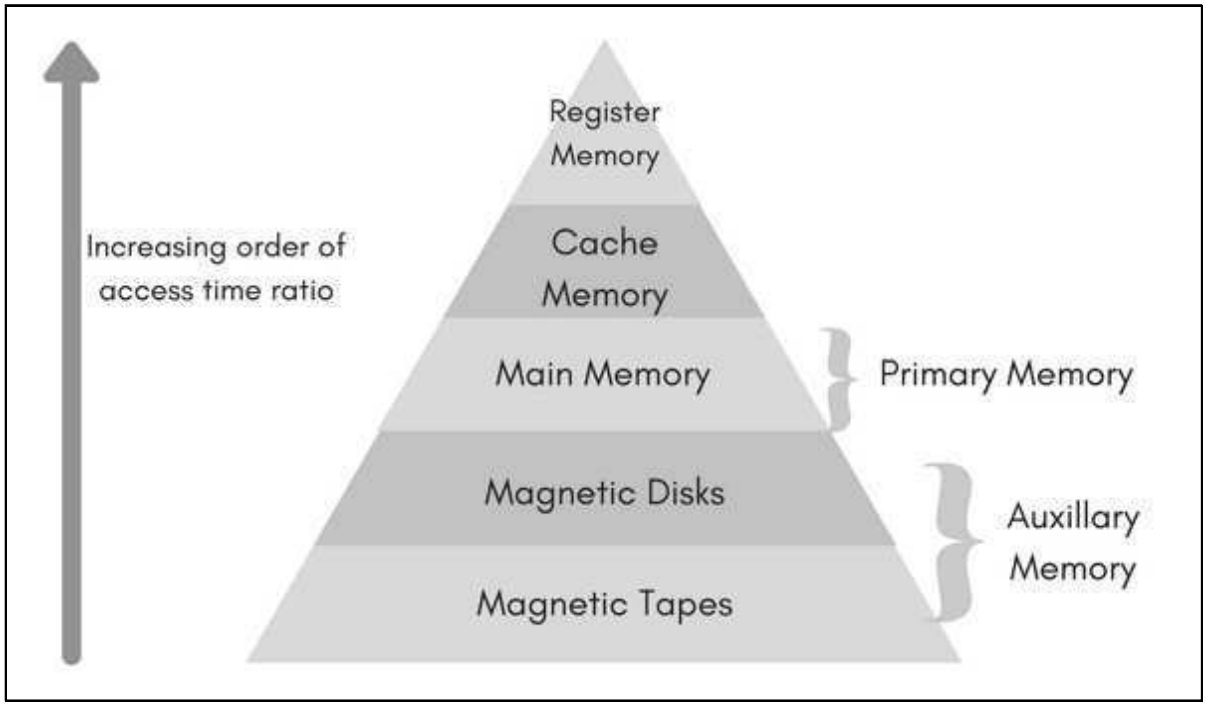

Fig 4.1 Memory Hierarchy

The memory hierarchy system consists of all storage devices contained in a computer system from the slow Auxiliary Memory to fast Main Memory and to smaller Cache memory. **Auxiliary memory** access time is generally **1000 times** that of the main memory, hence it is at the bottom of the hierarchy.

The **main memory** occupies the central position because it is equipped to communicate directly with the CPU and with auxiliary memory devices through Input/output processor (I/O).

The **cache memory** is used to store program data which is currently being executed in the CPU. Approximate access time ratio between cache memory and main memory is about **1 to 7-10**

Memory Access Methods

Each memory type is a collection of numerous memory locations. To access data from any memory, first it must be located and then the data is read from the memory location. Following are the methods to access information from memory locations:

1. **Random Access**: Main memories are random access memories, in which each memory location has a unique address. Using this unique address any memory location can be reached in the same amount of time in any order.
2. **Sequential Access**: This methods allows memory access in a sequence or in order.

3. **Direct Access**: In this mode, information is stored in tracks, with each track having a separate read/write head.

Main Memory

The memory unit that communicates directly within the CPU, Auxillary memory and Cache memory, is called main memory. It is the central storage unit of the computer system. It is a large and fast memory used to store data during computer operations. Main memory is made up of **RAM** and **ROM**, with RAM integrated circuit chips holing the major share.

- **RAM: Random Access Memory**
  - **DRAM**: Dynamic RAM, is made of capacitors and transistors, and must be refreshed every 10 - 100 ms. It is slower and cheaper than SRAM.
  - **SRAM**: Static RAM, has a six transistor circuit in each cell and retains data, until powered off.
  - **NVRAM**: Non-Volatile RAM, retains its data, even when turned off. Example: Flash memory.
- **ROM**: Read Only Memory, is non-volatile and is more like a permanent storage for information. It also stores the **bootstrap loader** program, to load and start the operating system when computer is turned on.
  PROM(Programmable ROM),
  EPROM(Erasable PROM) and
  EEPROM(Electrically Erasable PROM) are some commonly used ROMs.

Auxiliary Memory

Devices that provide backup storage are called auxiliary memory. **For example:** Magnetic disks and tapes are commonly used auxiliary devices. Other devices used as auxiliary memory are magnetic drums, magnetic bubble memory and optical disks.
It is not directly accessible to the CPU, and is accessed using the Input/Output channels.

Cache Memory

The data or contents of the main memory that are used again and again by CPU, are stored in the cache memory so that we can easily access that data in shorter time. Whenever the CPU needs to access memory, it first checks the cache memory. If the data is not found in cache memory then the CPU moves onto the main memory. It also transfers block of recent data into the cache and keeps on deleting the old data in cache to accomodate the new one.

Hit Ratio

The performance of cache memory is measured in terms of a quantity called **hit ratio**. When the CPU refers to memory and finds the word in cache it is said to produce a **hit**. If the word is not found in cache, it is in main memory then it counts as a **miss**.
The ratio of the number of hits to the total CPU references to memory is called hit ratio.
Hit Ratio = Hit/(Hit + Miss)

Associative Memory

It is also known as **content addressable memory (CAM)**. It is a memory chip in which each bit position can be compared. In this the content is compared in each bit cell which allows very fast table lookup. Since the entire chip can be compared, contents are randomly stored without considering addressing scheme. These chips have less storage capacity than regular memory chips.

Mapping and Concept of Virtual Memory

The transformation of data from main memory to cache memory is called mapping. There are 3 main types of mapping:

- Associative Mapping
- Direct Mapping
- Set Associative Mapping

Associative Mapping

The associative memory stores both address and data. The address value of 15 bits is 5 digit octal numbers and data is of 12 bits word in 4 digit octal number. A CPU address of 15 bits is placed in **argument register** and the associative memory is searched for matching address.

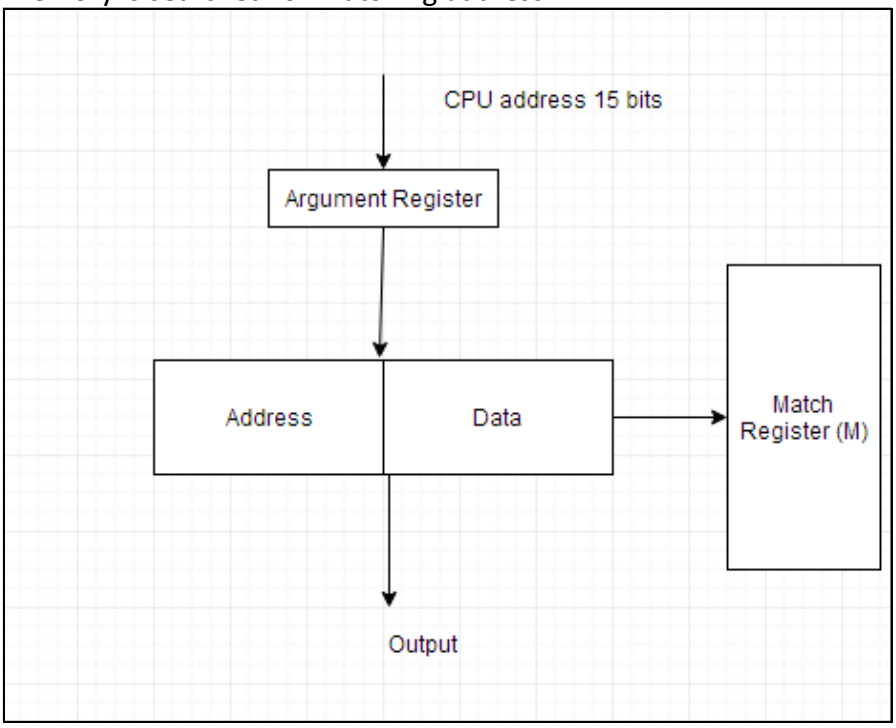

Fig: 4.2 **Associative Mapping**

Direct Mapping

The CPU address of 15 bits is divided into 2 fields. In this the 9 least significant bits constitute the **index** field and the remaining 6 bits constitute the **tag** field. The number of bits in index field is equal to the number of address bits required to access cache memory.

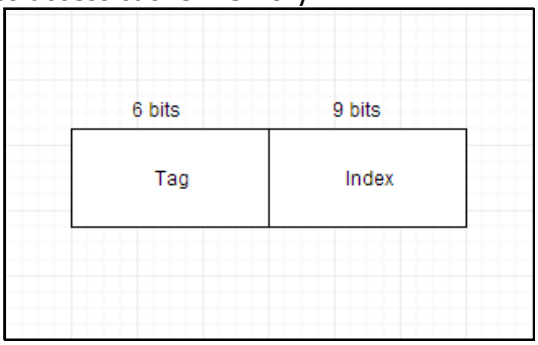

Fig 4.3 **Direct Mapping**

Set Associative Mapping

The disadvantage of direct mapping is that two words with same index address can't reside in cache memory at the same time. This problem can be overcome by set associative mapping.

In this we can store two or more words of memory under the same index address. Each data word is stored together with its tag and this forms a set.

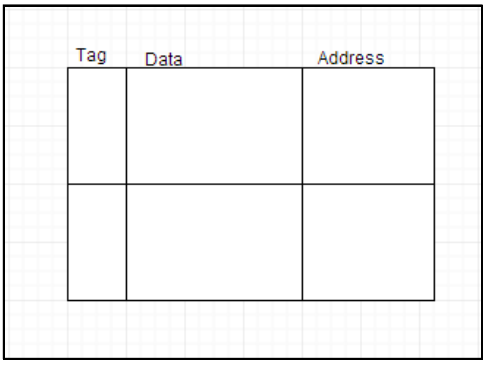

Fig 4.4 **Set Associative Mapping**

Write Policy

Cache is the technique of storing a copy of data temporarily in rapidly-accessible storage media (also known as memory) local to the central processing unit (CPU) . There are three main caching techniques that can be deployed, each with their own pros and cons.

- **Write-through cache** directs write I/O onto cache and through to underlying permanent storage before confirming I/O completion to the host. This ensures data updates are safely stored on, for example, a shared storage array, but has the disadvantage that I/O still experiences latency based on writing to that storage. Write-through cache is good for applications that write and then re-read data frequently as data is stored in cache and results in low read latency.

- **Write-around cache** is a similar technique to write-through cache, but write I/O is written directly to permanent storage, bypassing the cache. This can reduce the cache being flooded with write I/O that will not subsequently be re-read, but has the disadvantage is that a read request for recently written data will create a “cache miss” and have to be read from slower bulk storage and experience higher latency.

- **Write-back cache** is where write I/O is directed to cache and completion is immediately confirmed to the host. This results in low latency and high throughput for write-intensive applications, but there is data availability exposure risk because the only copy of the written data is in cache.

Cache Performance

Assuming a hit time of one CPU clock cycle, program execution will continue normally on a cache hit. The total number of stall cycles depends on the number of cache misses and the miss penalty.
Memory stall cycles = Memory accesses x miss rate x miss penalty

To include stalls due to cache misses in CPU performance equations, we have to add them to the “base” number of execution cycles.
CPU time = (CPU execution cycles + Memory stall cycles) x Cycle time.

We Assume that 33% of the instructions in a program are data accesses. The cache hit ratio is 97% and the hit time is one cycle, but the miss penalty is 20 cycles.
Memory stall cycles = Memory accesses x Miss rate x Miss penalty
= 0.33 I x 0.03 x 20 cycles
= 0.2 I cycles
If I instructions are executed, then the number of wasted cycles will be 0.2 x I. This code is 1.2 times slower than a program with a “perfect” CPI of 1!

CPU time = (CPU execution cycles + Memory stall cycles) x Cycle time

Processor performance traditionally outpaces memory performance, so the memory system is often the system bottleneck. For example, with a base CPI of 1, the CPU time from the last page is:

CPU time = (I + 0.2 I) x Cycle time

if we could double the CPU performance so the CPI becomes 0.5, but memory performance remained the same.

CPU time = (0.5 I + 0.2 I) x Cycle time

The overall CPU time improves by just 1.2/0.7 = 1.7 times

Virtual Memory

Virtual memory is the separation of logical memory from physical memory. This separation provides large virtual memory for programmers when only small physical memory is available. Virtual memory is used to give programmers the illusion that they have a very large memory even though the computer has a small main memory. It makes the task of programming easier because the programmer no longer needs to worry about the amount of physical memory available.

Address Space

Address space is the amount of memory allocated for all possible addresses for a computational entity, such as a device, a file, a server, or a networked computer. Address space may refer to a range of either physical or virtual addresses accessible to a processor or reserved for a process. As unique identifiers of single entities, each address specifies an entity's location (unit of memory that can be addressed separately). On a computer, each computer device and process is allocated address space, which is some portion of the processor's address space. A processor's address space is always limited by the width of its address bus and registers. Address space may be differentiated as either flat, in which addresses are expressed as incrementally increasing integers starting at zero, or segmented, in which addresses are expressed as separate segments augmented by offsets (values added to produce secondary addresses). In some systems, address space can be converted from one format to the other through a process known as thunking.

Address Mapping

Address binding is the process of mapping from one address space to another address space. Logical address is address generated by CPU during execution whereas Physical Address refers to location in memory unit(the one that is loaded into memory).Note that user deals with only logical address(Virtual address). The logical address undergoes translation by the MMU or address translation unit in particular. The output of this process is the appropriate physical address or the location of code/data in RAM.

An address binding can be done in three different ways:

Compile Time – It work is to generate logical address(also known as virtual address). If you know that during compile time where process will reside in memory then absolute address is generated.

Load time – It will generate physical address.If at the compile time it is not known where process will reside then relocatable address will be generated. In this if address changes then we need to reload the user code.

Execution time- It helps in differing between physical and logical address.This is used if process can be moved from one memory to another during execution(dynamic linking-Linking that is done during load or run time).

MMU(Memory Management Unit)-

The run time mapping between Virtual address and Physical Address is done by hardware device known as

MMU.In memory management, Operating System will handle the processes and moves the processes between disk and memory for execution . It keeps the track of available and used memory.

Instruction-execution cycle Follows steps:

1. First instruction is fetched from memory e.g. ADD A,B
2. Then these instructions are decoded i.e., Addition of A and B
3. And further loading or storing at some particular memory location takes place.

Basic Hardware

As main memory and registers are built into processor and CPU can access these only.So every instructions should be written in direct access storage devices.

1. If CPU access instruction from register then it can be done in one CPU clock cycle as registers are built into CPU.
2. If instruction resides in main memory then it will be accessed via memory bus that will take lot of time. So remedy to this add fast memory in between CPU and main memory i.e. adding cache for transaction.
3. Now we should insure that process resides in legal address.
4. Legal address consists of base register(holds smallest physical address) and limit register(size of range).

How processes are mapped from disk to memory

1. Usually process resides in disk in form of binary executable file.
2. So to execute process it should reside in main memory.
3. Process is moved from disk to memory based on memory management in use.
4. The processes waits in disk in form of ready queue to acquire memory.

Procedure of mapping of disk and memory

Normal procedure is that process is selected from input queue and loaded in memory. As process executes it accesses data and instructions from memory and as soon as it completes it will release memory and now memory will be available for other processes.

MMU scheme –

CPU------- MMU------Memory

Paging

Paging is a **memory management scheme**. Paging allows a process to be stored in a memory in a **non-contiguous** manner. Storing process in a non-contiguous manner solves the problem of **external fragmentation**.

For implementing paging the **physical and logical memory spaces** are divided into the same fixed-sized blocks. These fixed-sized blocks of physical memory are called **frames**, and the fixed-sized blocks of logical memory are called **pages**.

When a process needs to be executed the process pages from logical memory space are loaded into the frames of physical memory address space. Now the address generated by **CPU** for accessing the frame is divided into two parts i.e. **page number** and **page offset**.

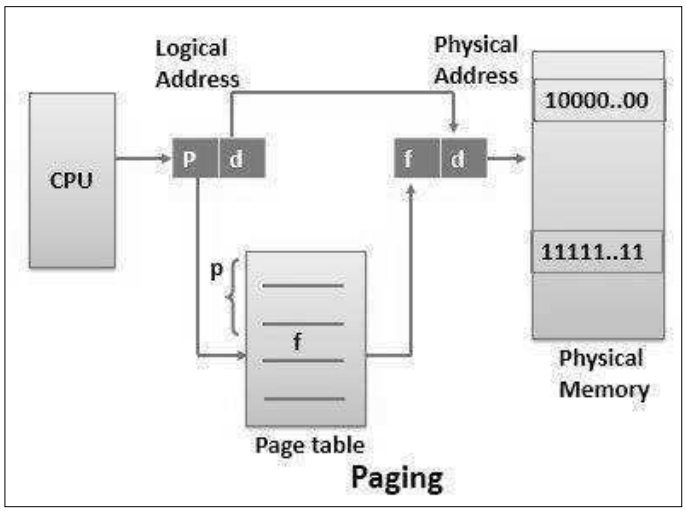

Fig 4.5 Paging

The **page table** uses page number as an index; each process has its separate page table that maps logical address to the physical address. The page table contains base address of the page stored in the frame of physical memory space. The base address defined by page table is combined with the page offset to define the frame number in physical memory where the page is stored.

Segmentation

Like Paging, **Segmentation** is also a **memory management scheme**. It supports the user's view of the memory. The process is divided into the **variable size segments** and loaded to the logical memory address space. The logical address space is the collection of variable size segments. Each segment has its **name** and **length**. For the execution, the segments from logical memory space are loaded to the physical memory space.

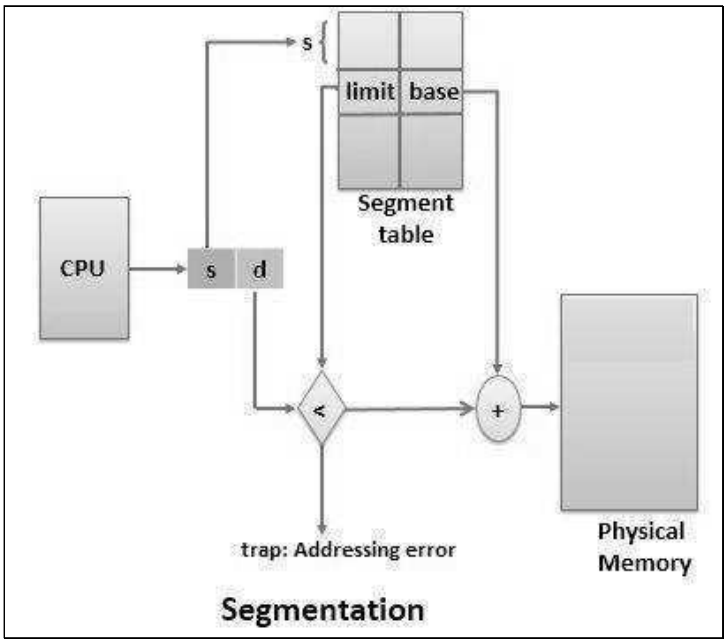

Fig 4.6 Segmentation

The address specified by the user contain two quantities the **segment name** and the **Offset**. The segments are numbered and referred by the **segment number** instead of segment name. This segment number is used as an index in the **segment table**, and **offset** value decides the length or **limit** of the segment. The segment number and the offset together combinely generates the address of the segment in the physical memory space.

TLB

A translation lookaside buffer (TLB) is a memory cache that stores recent translations of virtual memory to physical addresses for faster retrieval. When a virtual memory address is referenced by a program, the search starts in the CPU. First, instruction caches are checked. If the required memory is not in these very fast caches, the system has to look up the memory's physical address. At this point, TLB is checked for a quick reference to the location in physical memory. When an address is searched in the TLB and not found, the physical memory must be searched with a memory page crawl operation. As virtual memory addresses are translated, values referenced are added to TLB. When a value can be retrieved from TLB, speed is enhanced because the memory address is stored in the TLB on processor. Most processors include TLBs to increase the speed of virtual memory operations through the inherent latency-reducing proximity as well as the high-running frequencies of current CPU's.

TLBs also add the support required for multi-user computers to keep memory separate, by having a user and a supervisor mode as well as using permissions on read and write bits to enable sharing. TLBs can suffer performance issues from multitasking and code errors. This performance degradation is called a cache thrash. Cache thrash is caused by an ongoing computer activity that fails to progress due to excessive use of resources or conflicts in the caching system.

Page Fault

A page fault occurs when a program attempts to access data or code that is in its address space, but is not currently located in the system RAM. So when page fault occurs then following sequence of events happens :

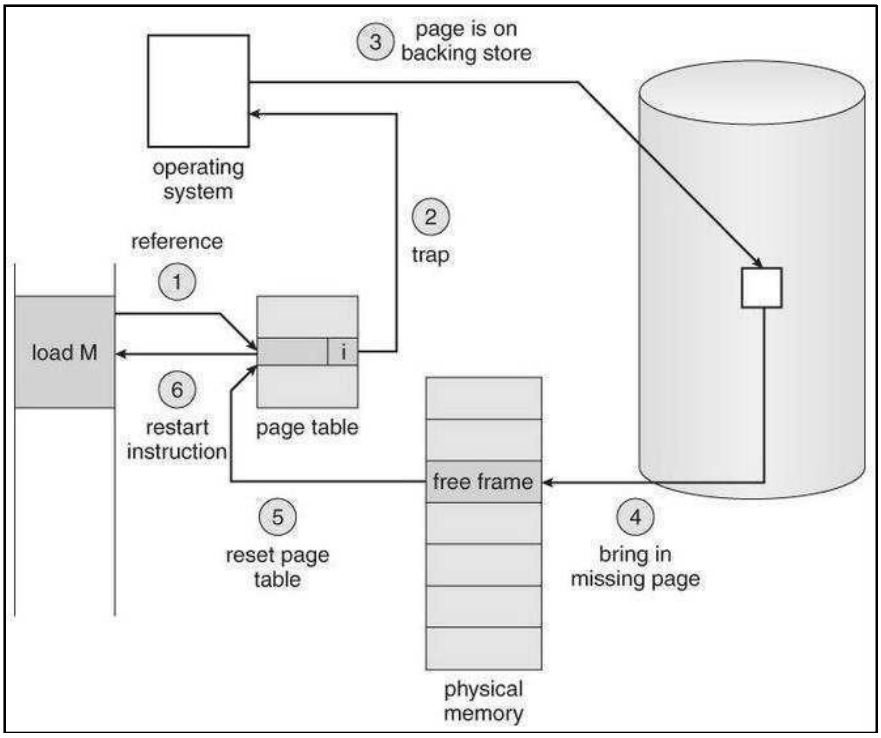

Fig 4.7 TLB

- The computer hardware traps to the kernel and program counter (PC) is saved on the stack. Current instruction state information is saved in CPU registers.
- An assembly program is started to save the general registers and other volatile information to keep the OS from destroying it.
- Operating system finds that a page fault has occurred and tries to find out which virtual page is needed. Some times hardware register contains this required information. If not, the operating system must retrieve PC, fetch instruction and find out what it was doing when the fault occurred.
- Once virtual address caused page fault is known, system checks to see if address is valid and checks if there is no protection access problem.
- If the virtual address is valid, the system checks to see if a page frame is free. If no frames are free, the page replacement algorithm is run to remove a page.
- If frame selected is dirty, page is scheduled for transfer to disk, context switch takes place, fault process is suspended and another process is made to run until disk transfer is completed.
- As soon as page frame is clean, operating system looks up disk address where needed page is, schedules disk operation to bring it in.
- When disk interrupt indicates page has arrived, page tables are updated to reflect its position, and frame marked as being in normal state.
- Faulting instruction is backed up to state it had when it began and PC is reset. Faulting is scheduled, operating system returns to routine that called it.
- Assembly Routine reloads register and other state information, returns to user space to continue execution

Effective Access Time

The percentage of times that the page number of interest is found in the TLB is called the hit ratio. An 80-percent hit ratio, for example, means that we find the desired page number in the TLB 80 percent of the time. If it takes 100 nanoseconds to access memory, then a mapped-memory access takes 100 nanoseconds when the page number is in the TLB. If we fail to find the page number in the TLB then we must first access memory for the page table and frame number (100 nanoseconds) and then access the desired byte in memory (100 nanoseconds), for a total of 200 nanoseconds. (We are assuming that a page-table lookup

takes only one memory access, but it can take more)

To find the effective memory-access time, we weight the case by its probability: effective access time = 0.80 × 100 + 0.20 × 200 = 120 nanoseconds

Replacement Algorithm

Page Replacement Algorithms :

- **First In First Out (FIFO) –**

  This is the simplest page replacement algorithm. In this algorithm, operating system keeps track of all pages in the memory in a queue, oldest page is in the front of the queue. When a page needs to be replaced page in the front of the queue is selected for removal.
  For example-1, consider page reference string 1, 3, 0, 3, 5, 6 and 3 page slots.

  Initially all slots are empty, so when 1, 3, 0 came they are allocated to the empty slots —> **3 Page Faults.**

  when 3 comes, it is already in memory so —> **0 Page Faults.**

  Then 5 comes, it is not available in memory so it replaces the oldest page slot i.e 1. —>**1 Page Fault.**

  Finally 6 comes, it is also not available in memory so it replaces the oldest page slot i.e 3 —>**1 Page Fault.**

  Example-2, Let's have a reference string: a, b, c, d, c, a, d, b, e, b, a, b, c, d and the size of the frame be 4.

Time req. / Page frames		1	2	3	4	5	6	7	8	9	10	11	12	13	14
		a	b	c	d	c	a	d	b	e	b	a	b	c	d
0	a	**a**	a	a	a	a	a	a	a	**e**	e	e	e	e	**d**
1	b		**b**	b	b	b	b	b	b	b	b	**a**	a	a	a
2	c			**c**	c	c	c	c	c	c	c	c	**b**	b	b
3	d				**d**	d	d	d	d	d	d	d	d	**c**	c
FAULTS		**x**	**x**	**x**	**x**					**x**		**x**	**x**	**x**	**x**

Fig 4.8 FIFO

There are 9 page faults using FIFO algorithm.

Belady's anomaly – Belady's anomaly proves that it is possible to have more page faults when increasing the number of page frames while using the First in First Out (FIFO) page replacement algorithm. For example, if we consider reference string 3, 2, 1, 0, 3, 2, 4, 3, 2, 1, 0, 4 and 3 slots, we get 9 total page faults, but if we increase slots to 4, we get 10 page faults.

- **Optimal Page replacement –**

  In this algorithm, pages are replaced which are not used for the longest duration of time in the future.
  Let us consider page reference string 7 0 1 2 0 3 0 4 2 3 0 3 2 and 4 page slots.

  Initially all slots are empty, so when 7 0 1 2 are allocated to the empty slots —> **4 Page faults**

0 is already there so —> **0 Page fault.**

when 3 came it will take the place of 7 because it is not used for the longest duration of time in the future.—>**1 Page fault.**

0 is already there so —> **0 Page fault.**.

4 will takes place of 1 —> **1 Page Fault.**

Now for the further page reference string —> **0 Page fault** because they are already available in the memory.

Example-2, Let's have a reference string: a, b, c, d, c, a, d, b, e, b, a, b, c, d and the size of the frame be 4.

Time req. / Page frames	1	2	3	4	5	6	7	8	9	10	11	12	13	14
	a	b	c	d	c	a	d	b	e	b	a	b	c	d
0 a	**a**	a	a	a	a	a	a	a	**a**	e	e	e	e	**d**
1 b		**b**	b	b	b	b	b	b	b	b	a	a	a	a
2 c			**c**	c	c	c	c	c	c	c	c	b	b	b
3 d				**d**	d	d	d	d	**e**	d	d	d	c	c
FAULTS	**x**	**x**	**x**	**x**					**x**					**x**

Fig 4.9 OPR

There are 6 page faults using optimal algorithm.

Optimal page replacement is perfect, but not possible in practice as operating system cannot know future requests. The use of Optimal Page replacement is to set up a benchmark so that other replacement algorithms can be analyzed against it.

- **Least Recently Used –**

In this algorithm page will be replaced which is least recently used.
Let say the page reference string 7 0 1 2 0 3 0 4 2 3 0 3 2 . Initially we have 4 page slots empty. Initially all slots are empty, so when 7 0 1 2 are allocated to the empty slots —> **4 Page faults**

0 is already their so —> **0 Page fault.**

when 3 came it will take the place of 7 because it is least recently used —>**1 Page fault**

0 is already in memory so —> **0 Page fault**.

4 will takes place of 1 —> **1 Page Fault**

Now for the further page reference string —> **0 Page fault** because they are already available in the memory.

Example-2, Let's have a reference string: a, b, c, d, c, a, d, b, e, b, a, b, c, d and the size of the frame be 4.

Time req. / Page frames		1	2	3	4	5	6	7	8	9	10	11	12	13	14
		a	b	c	d	c	a	d	b	e	b	a	b	c	d
0	a	**a**	a	a	a	a	a	a	a	a	e	e	e	e	e
1	b		**b**	b	b	b	b	b	b	b	b	a	a	a	a
2	c			**c**	c	c	c	c	c	**e**	c	c	b	b	**d**
3	d				**d**	d	d	d	d	d	d	d	d	**c**	c
FAULTS		**x**	**x**	**x**	**x**					**x**				**x**	**x**

Fig 4.10 LRU

There are 7 page faults using LRU algorithm.

RGPVNOTES.IN

We hope you find these notes useful.

You can get previous year question papers at https://qp.rgpvnotes.in .

If you have any queries or you want to submit your study notes please write us at rgpvnotes.in@gmail.com

LIKE & FOLLOW US ON FACEBOOK
facebook.com/rgpvnotes.in