

**Computer Organization and Architecture: A Pedagogical Aspect**  
**Prof. Jatindra Kr. Deka**  
**Dr. Santosh Biswas**  
**Dr. Arnab Sarkar**  
**Department of Computer Science & Engineering**  
**Indian Institute of Technology, Guwahati**

**Lecture - 31**  
**Page Replacement Algorithms**

Welcome, in this lecture we will continue our discussion with virtual memories and caches. We will start with a bit of recap from what we discussed in the last class. So, particularly we will start by discussing a bit again, on virtually indexed physically tagged caches. We had said last day that the problem with physically indexed physically tagged caches was that, the TLB comes in the critical path for cache accesses.

So, therefore, cache access latencies are high because the TLB comes in the middle and we cannot access the cache, until the complete physical address is generated. For to do away with this, to improve the situation, so, virtually indexed virtually tagged caches, VIVT caches, we had proposed and there what happened is that the both the indexing and tagging of the cache was done based on virtual addresses ok. So, the logical address was used for both indexing and tagging of the cache

Now, this avoided TLBs to come into the critical path. So, TLB's are no more coming into the critical path; however, the problem again it was that both the indexing and tagging because, it is done with logical addresses, it has no connection with the physical address and where a particular cache block is placed in physical memory. So, the issue is that now the advantage of VIVT caches is that so, I don't have to go into the TLB. So, even if there is a miss of the TLB and I have to go to the memory to bring in the physical page number, even that is avoided and we don't need to go into the we don't need to go into the TLB for that, if the if the data is in cache we are fine we are happy. So, we don't go into the TLB to look for the physical address at all.

However, the problem as we said is that virtual addresses have no connection with physical addresses. So, a particular data in cache is now stored only with respect to what the logical address says and the logical address of different processes may be same. The physical address so, multiple processors have different physical addresses. So, the data corresponding to multiple processes will be stored in different locations in the physical memory. So, when I have

the cache that indexed and tagged based on physical addresses I don't have the problem that the same cache block can be stored in multiple different locations, or multiple different sets, in the cache the same cache block cannot be stored in multiple different sets in the cache. So, the same block in physical memory cannot be stored in multiple different sets in the cache. If I am address indexing and tagging the cache using physical addresses.

However because these are virtual addresses so, a block in physical memory can be stored in multiple different locations or multiple different sets in the cache. And this is a problem because the same virtual address can mean different physical addresses by different processors ok. So, therefore, the same cache physical cache block maybe stored in different locations and therefore, the cache needs to be flushed, every time there is a context switch and a different process comes into the CPU.

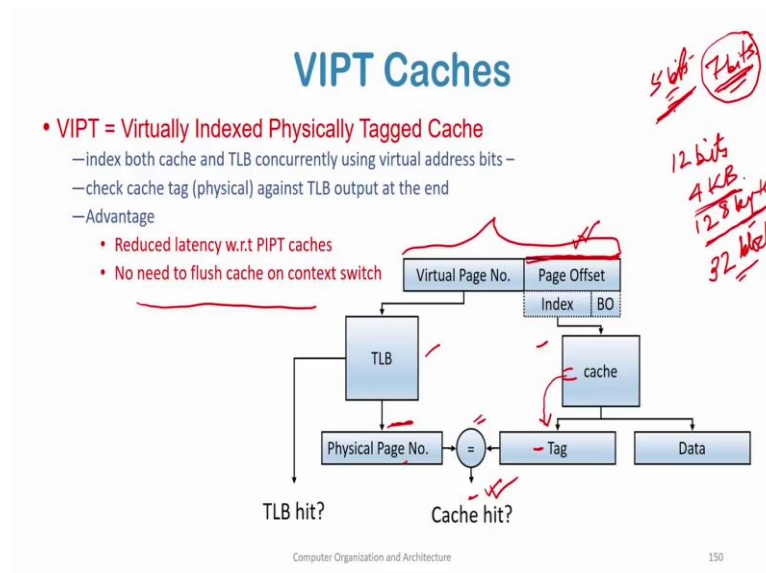
So, when one process is executing on the CPU, for that the for the virtual addresses of that process I am accessing the cache using virtual logical address of that processor process and now when there is a context don't a different process comes in and there the virtual address will mean entirely different set of physical addresses and therefore, the previous entire cache the cache needs to be flushed and, there can be a lot of cold misses; that means, the previous data is all rubbed is all deleted from the cache and therefore, when the new process comes in I will have nothing in the cache of nothing of the physical memory in cache and therefore, I have to repopulate everything in the cache corresponding to that process ok and this will lead to a lot of cold misses as it is called ok. Because the cache is cold and I will cold or empty and therefore, I have to bring in data from the physical memory into cache and so, that was the problem of virtually indexed virtually tagged caches.

Now virtually indexed physically tagged caches was a compromise between these two. So, in virtually indexed physically tagged caches what do we do? We index both the cache and TLB concurrently using virtual address bits ok. So, the virtual page number part of the virtual address is used to go used to search the TLB for a hit. So, the TLB is fully is fully associative and so therefore, or in the all the entries in the TLB will be searched for the virtual page number and if the virtual page number is found the corresponding physical page number is taken.

Now concurrently I will use the physical page offset sorry, the virtual page offset which is same as the physical page offset. So, the physical page offset will be used to index the cache and if there is a if the physical page offset matches, if the physical page offset sorry I will I will use

the index to in I will go use the physical page offset to index the cache and then corresponding to that I will try to match the tag at that particular at that particular location that particular block, or a particular set of blocks in a set for a match of the physical page number that I got as output from the physical page numbers.

(Refer Slide Time: 07:01)



So, here my TLB has produced a physical page number from here, I am indexing the cache. So, I have gone to a particular location and found a certain tag and that tag I have obtained. If this tag matches with the physical page number then I have a cache hit. So, why this is a benefit this is a benefit because, the cache the cache and the TLB is accessed concurrently not sequentially one after another, but concurrently and therefore, I save time; the TLB does not come in the critical path. However, if there is a TLB miss this access still has to wait, this access still has to wait to get the physical page number from memory back and then only we can we can have check for a cache hit. So, therefore, this strategy is helpful when there is a TLB hit.

So, on an average it reduces access times, with respect to with respect to virtually indexed virtually tagged sorry with respect to physically indexed physically tagged cache because the TLB and cache are accessed concurrently. With respect to virtually indexed virtually tagged cache, it is it is not as efficient when one process is running because, if there is a TLB miss I have to go to the physical memory; however, if there is a TLB hit I have I can check for the cache hit without going into the physical memory and I save time.

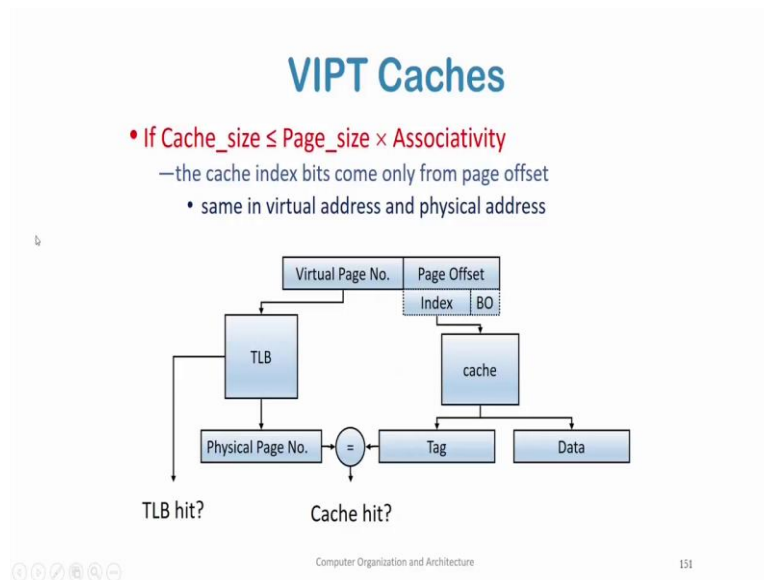
But this however, this approach avoids the need for need to flush the cache on a context switch ok. So, why because the physical page offset and the virtual page offset are same ok. So, therefore, when I am accessing the cache with the page offset part only of the virtual page number. So, this is the complete virtual address, this is the complete virtual address and I am accessing the cache only with the physical page offset part, I am indexing the cache only with the physical page offset part, if I am doing this then what essentially is happening is that, I am basically indexing the cache using basically using physical addresses only, because the page offset part of the virtual address and physical address is same.

So, if the cache is accessed only using the physical you only using the page offset part of the virtual memory, then what happens is that let us say I have the page offset of 12 bits 12 bits. So, the page size is 4 KB and let us say my cache block size is 128 bytes. So therefore, I have  $8 \times 4$ , 32 cache blocks 32 blocks per page I have 32 blocks per page.

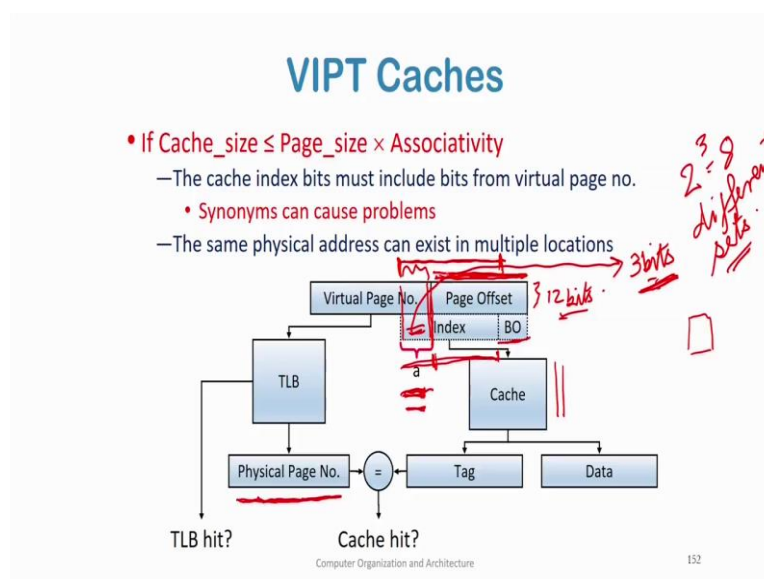
Now, each of these 32 blocks will go to a particular location in the cache, depending on what, so I have 32 128 bytes ok. So, I have 128 bytes in is the block size. So, this will require 7 bits this will require 7 bits and therefore, so this will require 7 bits and the other 5 bits will tell me where it will go in the cache ok. So, the cache the cache is no more than 12. So, the cache is also 4 KB in size; the cache is 4 KB in size and I know depending on what the other 5 bits are so, 7 bits the lower 7 bits are for accessing the cache and the higher significant 5 bits will tell me which particular block in the cache, this cache block is going to go, which particular line in cache will this particular cache block.

So, these 7 bits each enumeration of the second bits will identify a particular cache block and this cache block will go to a designated location, or designated line in cache depending on the value of the more significant of the higher significant 5 bits. Now, therefore, each cache block will have a designated location in cache and, the cache block cannot sit cannot be located in to multiple locations in the cache, depending on means irrespective of what the virtual address is because, the physical page offset and the virtual page is same.

(Refer Slide Time: 12:00)



(Refer Slide Time: 12:04)



When the situation changes, when we want to increase the size of the cache, now, when we want to increase the size of the cache, then I need to use a part of the virtual page number this part of the virtual this part these many bits let us say these are 3 bits. So, in addition to the 12 bits that I have 12 bits that I have in addition to I was using 12 bits. Now in addition to 12 bits let us say I use 3 more bits from the virtual page number to index the cache, why because my cache was of size. So, I had 12 bits so, 12 bits can access any one of  $2^{12}$  locations. So, any one of  $2^{12}$  blocks it can access.

A barring the byte offset again, so, ok. So, it will it will it the page of so I will in I will need few more bits, I will need 3 more bits. So, 3 more bits I have used to increase the size of the cache. Now what is the problem that this has brought into? Now a particular now a particular block cache block a particular block in physical memory can sit in multiple locations in the cache, why? Because these 3 bits these 3 bits will now depend on what the virtual address says. This, previously what was happening I was only the using this physical offset part of the cache.

And therefore, when I am when I am when I am appending it to the physical page number and the physical page offset, I know that corresponding to this physical address, I my cache block will sit in a particular set or particular block in the cache only. Now these 3 additional bits have created this problem that given for a given physical address depending on what the values of these 3 bits are, it can it the same the same cache block the same block in physical memory can go into different sets, different sets or different blocks, depending on what type of set associative or direct mapped or what it is. So, let us say if we have a set associative cache and therefore, the index part will tell me which set which set in cache my particular physical my particular physical block will go into.


Now, this page offset part remains same, but these 3 these 3 bits become different. Now these 3 bits therefore this, what happens due to this is that  $2^3$  or 8 different location 8 different sets ok. Now these 3 bits mean 8 different sets for a given this part remaining same, this part of the address remaining same, even the physical page number remaining same, when the physical page number remains same and this part remains same; that means, I am going to the same physical address, I am trying to access the same physical address.

However, depending on what the value of this 3 bits are the same physical address. So, the block which contains this physical address can go into 8 different sets in the cache. So, I will I will reiterate based on this physical page number and this page offset let us say a situation, in which this physical page number is same and the physical page offset is same and I am trying to index the cache using this index. Now this part of the index is going to remain same for this physical address. However, for the same physical page the physical page number could be different sorry the physical for the same physical page number depending on what these 3 bits are the same physical block can go into 8 different sets in cache and this as he had told is the synonym problem.

(Refer Slide Time: 16:55)

# Solutions to the Synonym Problem

- Limit cache size to page size times associativity
  - Get index only from page offset
  - Have bigger page size
- On a write, search all possible indices that can contain the same physical block, and update/invalidate
  - Used in Alpha 21264, MIPS R10K
- Restrict virtual page to physical page frame mapping in OS
  - make sure:  $\text{index}(\text{Virtual address}) = \text{index}(\text{Physical address})$
  - Called page coloring
    - All physical page frames are colored  $\neq$ 
      - A physical page of one color is mapped to a virtual address by OS in such a way that a set in cache always gets page frames of the same color.
  - Used in many SPARC processors



153

And one of the ways in which we had discussed 3 ways, I will today I will just recap the last one; one of the ways to handle this synonym problem was page colouring. And we said what was page colouring it is to restrict virtual page to physical page frame mapping in OS, we will restrict virtual page to physical page frame mapping in the OS.

So, how will we do this? We will try to make sure that the index that the virtual address produces is ok. So, virtual address meaning the virtual address meaning this one, this entire thing is basically part of the virtual address. This is the virtual address, so, that is why it is a virtually indexed cache. So, this is part of the virtual address. So, the index that the virtual address produces, we will try to make it same as if the physical address would create.

And how will we do that? We will do that using a scheme called page colouring in which all physical page frames are coloured. So, how are they coloured. So, now the physical memory if we see let us say this is the physical memory and this one is let us say the pages in it these are the pages in it, physical memory and these are the pages in it. Now this one will require what one this page offset. So, page offset page offset will address each location within this page, within each page the page offset can locate. Now which page will be given by the page number?

Now let us say we coloured the physical memory into 8 colours. So, colouring means I will give a unique ID. So, let us say this one is given 0 0 0, this one is given 0 0 1. So, 0 1 0, 0 1 1, 1 0 0, 1 0 1, 1 1 0, 1 1 1. So, I give 8 different colours. So, now, again for the next set of next

set of pages, I will again give colours to it 0 0 0, 0 0 1, 0 1 0, 0 1 1, 0 1 1 and likewise it will go on ok. So, I will go and colour each page in the physical memory. So, statically before in the system, I will know that this page has colour 0 0 0, page 2 has colour 0 0 1, page 3 has colour 0 1 0 likewise. And again this one will have again colour 0 0 0, this one will have again colour 0 0 1. So, for each page I will know what is its colour ok.

Now, what will I do is that. So, each cache block within this. So, the page will be composed of an integral number of cache blocks. So, in this in this particular page there will be a number of blocks. So, not cache block, but number of blocks. So, each page again will have a number of blocks like we had said in the previous case that our page had was composed of 32 blocks. So, each page had 32 blocks. Now here all these blocks will have a colour of 0 0 1; all these colour all these all these blocks in physical memory within this page will also have the same colour as the page.

So, now for each block in physical memory I know what colour it is ok. Now we I will use the scheme; a physical page of one colour is mapped to a virtual address by the OS in such a way that that a set in cache always gets page frames of the same colour. Now a physical page of one colour is mapped to a virtual address; so, this physical page will be mapped to a virtual address ok. Now if this page I will always map to a virtual address such that those 3 bits, those 3 bits, these 3 bits in a, we in a these 3 bits will also have 0 0 1 ok.

So, this physical page will be mapped to a virtual will map to such a virtual address, such that those 3 bits in a, of the virtual address will have will be 0 0 1 ok. Now what happens if for this virtual address therefore, I know that those 3 bits will be 0 0 1; so, the virtual I am restricting what, I am restricting during the mapping of the physical. So, for a virtual address I will map a physical page frame.

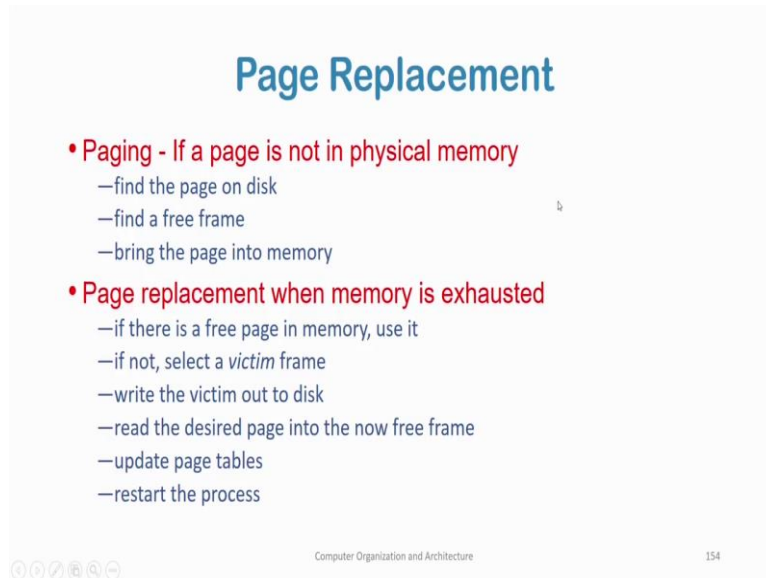
Now when I am doing this mapping between virtual address to physical address, I will I will map such a physical page number to a virtual page number that those for in that virtual page number those 3 bits of a will be 0 0 1, if this is the physical page I am referring to. So, only those virtual addresses will be able to get these physical page frames, if that virtual address or those set of that virtual page number.

So, this page number will be given to such a virtual page number, in which those 3 bits of a will be 0 0 1. So, by this scheme I will always be able to ensure that this page will go to the same set in cache. So, when this page goes to the same set in cache, I will be able to avoid the



synonym problem. Now, we will quickly study page replacement and go and look at page replacement again. So, that we discuss one more important problem which is Belady's anomaly and progress from there.

(Refer Slide Time: 23:40)



## Page Replacement

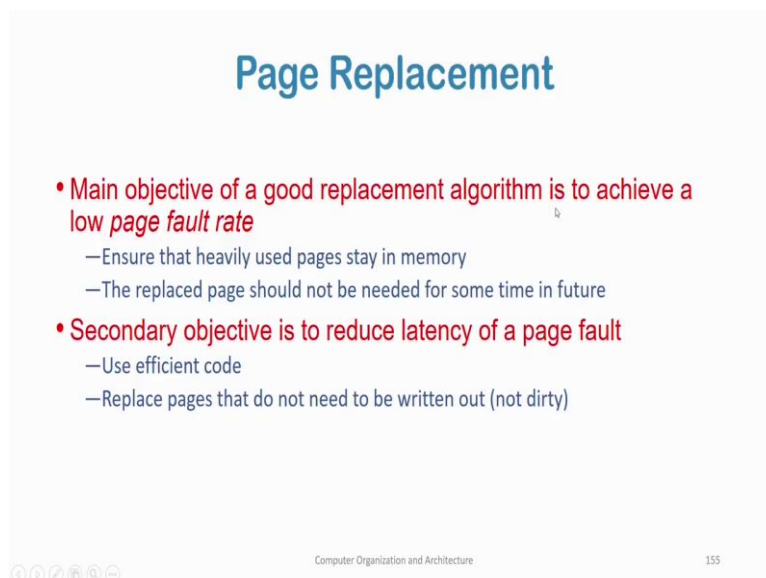
- **Paging - If a page is not in physical memory**
  - find the page on disk
  - find a free frame
  - bring the page into memory
- **Page replacement when memory is exhausted**
  - if there is a free page in memory, use it
  - if not, select a *victim* frame
  - write the victim out to disk
  - read the desired page into the now free frame
  - update page tables
  - restart the process

Computer Organization and Architecture

154

So, we had already told why page replacement is required.

(Refer Slide Time: 23:41)



## Page Replacement

- **Main objective of a good replacement algorithm is to achieve a low *page fault rate***
  - Ensure that heavily used pages stay in memory
  - The replaced page should not be needed for some time in future
- **Secondary objective is to reduce latency of a page fault**
  - Use efficient code
  - Replace pages that do not need to be written out (not dirty)

Computer Organization and Architecture

155

And to reduce page fault rates.

(Refer Slide Time: 23:46)

## Reference String

- Reference string is the sequence of pages being referenced
- If user has the following sequence of addresses  
—123, 215, 600, 1234, 76, 96
- If the page size is 100, then the reference string is  
—1, 2, 6, 12, 0, 0

↓

Computer Organization and Architecture 156

And we said what reference string are is. So, these are the set of pages that that the that a processor is accessing.

(Refer Slide Time: 23:57)

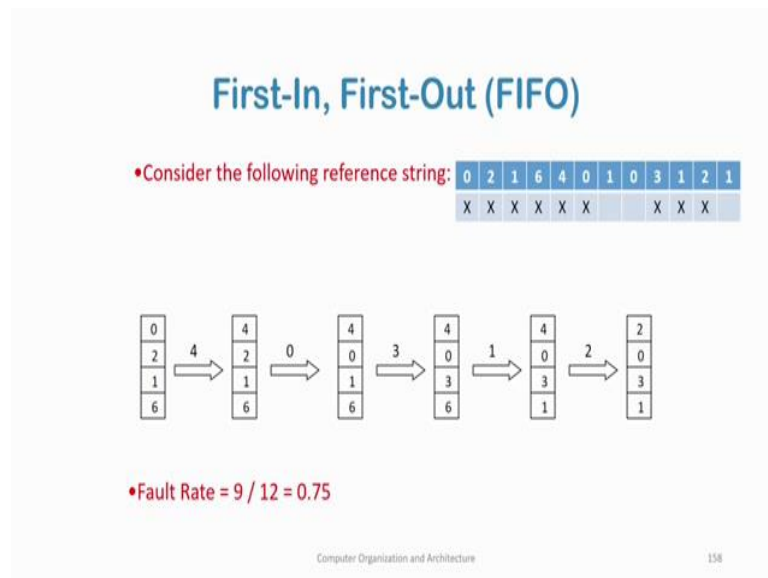
## First-In, First-Out (FIFO)

- The oldest page in physical memory is the one selected for replacement ✓
- Very simple to implement
  - keep a list
    - victims are chosen from the tail
    - new pages in are placed at the head

Computer Organization and Architecture 157

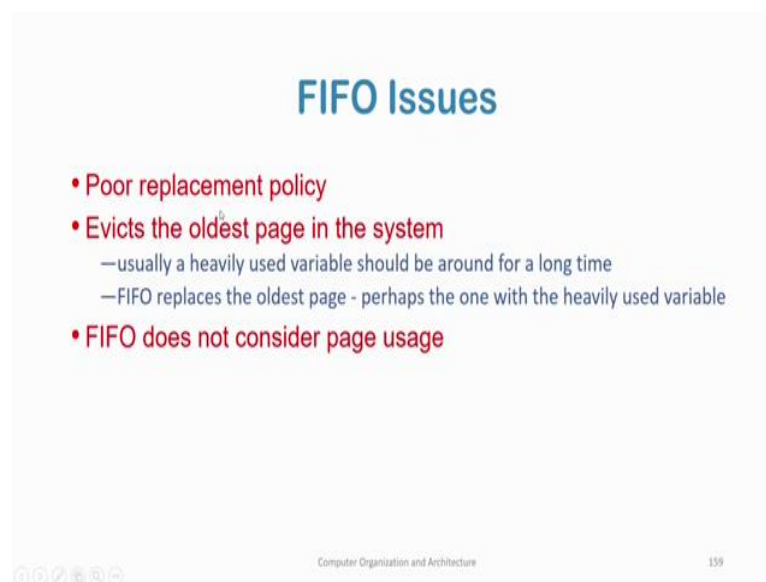
Then we discussed different page replacement policies, the first one we discussed was first in first out, in which the oldest page in physical memory is the one selected for replacement. So, the oldest page in physical memory at any given time is used for replacement.

(Refer Slide Time: 24:17)



And we discussed this we will discuss again with respect to Belady's anomaly. So, I am not going into first in first out.

(Refer Slide Time: 24:26)



We will look at FIFO issues again with respect to Belady's anomaly.

(Refer Slide Time: 24:29)

## Optimal Page Replacement

- **Basic idea**
  - replace the page that will not be referenced for the longest time in future
- **This gives the lowest possible fault rate**
- **Impossible to implement**
- **Does provide a good measure for other techniques**

Computer Organization and Architecture 160

So, I will not go into this again, for optimal replacement we said that we replace the page which will not be referenced for the longest time in future. So, I will have to know using an oracle as to which pages will be accessed in future, this is not possible and hence this optimal page replacement policy is not realizable in practice, but we use this to measure or evaluate and compare other algorithms against, how good it is. Because we cannot do better than the optimal, we will use it to compare other algorithms with respect to this one.

(Refer Slide Time: 25:07)

## Least Recently Used (LRU)

- **Basic idea**
  - replace the page in memory that has not been accessed for the longest time in the past
- **Optimal policy looking back in time** =
  - as opposed to forward in time
  - fortunately, programs tend to follow similar behavior

Computer Organization and Architecture 162

Then we came into least recently used and we said that in least recently used, we replace that page in memory that has not been accessed for the longest time in the past. So, at any point in time I will then the page frames which page frames will be contained, the page the pages in the physical memory will be the one which is most recently used. So, the least recently used one will be will be will be replaced, when I need to replace a page. So, when I need to replace a page when do I need to replace the page? When there are no free frames in memory and to get a new page into the physical memory, I have to replace an existing page in the page in the physical memory, send it to the secondary memory, if it is dirty and then bring in a new page.

So, we said that LRU was the is the optimal algorithm is an optimal algorithm, when with the restriction that I can look back in time, but I cannot look forward; that means, this is this is a this is practical because, looking back is possible; looking forward in what will happen in future is not possible, but what has happened we already know, therefore looking back in future this is the optimal algorithm. Why? Because it always keeps the most recently more most recently used pages at any given time.

(Refer Slide Time: 26:38)

**LRU Issues**

- **How to keep track of last page access?**
  - requires special hardware support
- **2 major solutions**
  - counters
    - hardware clock “ticks” on every memory reference
    - the page referenced is marked with this “time”
    - the page with the smallest “time” value is replaced
  - stack
    - keep a stack of references
    - on every reference to a page, move it to top of stack
    - page at bottom of stack is next one to be replaced

Computer Organization and Architecture 154

So, we looked into least recently used, so, I will not go into that anymore, but we will study what were the problems with we also saw this what are the problems with LRU and we said that the problem was in practical implementation, why is such a thing why is it difficult to implement in practice because, at each point in time for each page in the physical memory, I have to keep when it was accessed because, I am I am I am evicting the page, I am replacing

the page which is least recently used. So, I need to know among all pages in physical memory which one is the least recently used.

So, what the logical way of doing that? I will have to keep a global clock and whenever memory is being accessed I will I will take the stamp of the global clock and put that stamp on this physical page. So, when I need to evict pages, I will I will I will have a timestamp associated with each physical page in memory, whenever the page is accessed I provide it a time stamp I provide the time at which it was accessed and therefore, I will know which one among all the pages in physical memory, which one was least recently used, which was accessed farthest back in time. And that will be evicted. So, this will this has a lot of hardware cost because and also overheads because, at each access I have to update the value of this time stamp corresponding to that page. And this is hard.

So, anyhow the solutions are this so, I keep the hardware clock ticks on every memory reference. So, this is this keeps global time. So, with respect to memory reference for each memory reference irrespective of which process does this reference, I keep a global clock and I go on incrementing a global clock. Now the value of this global clock is the time stamp which is attached to a page whenever it is referenced and the page with the smallest time value is replaced.

Now this is a very costly solution as we said; a simpler solution is this: We keep a stack of references and the stack is maintained as a doubly linked list and, on each reference to a page we what do we do? So, when a page is referenced and it is found in physical memory, it will be in a certain position in the stack. So, I take that take this take this reference this page and the node in the stack corresponding to this page I take it out and put it on the top of the stack. So, this will require the updation of the 6 pointers. And now at any point in time because, whatever when whenever a page is being accessed I am taking that page out and putting on top of the stack, now what is happening is that when so, what is happening is that when I need to replace a page, the page which is at the bottom of the stack is a least recently used one and that is replaced.

(Refer Slide Time: 30:07)

## LRU Issues

- Both techniques require additional hardware
  - As memory reference are very frequent phenomena
  - Impractical to invoke software on every memory reference
- LRU is not used very often
- Instead, approximate LRU is more commonly used

Computer Organization and Architecture 165

So, both techniques require additional hardware and memory because memory references are a very frequent phenomena. It is the overhead, the overhead if we implement it in software in software means whenever, there is a memory reference I have to go to the OS and update either I have to do a stack operation, or I have to do more costly continuously I have to do or I have to do I have to update the timestamp corresponding to that page.

Now, in the first approach when I am using counters, what happens is that when I need to replace a page I have to search all my timestamps corresponding to all pages to find the find the page with the least value of the timestamp, which is very costly. Now when I am using a stack at each memory reference, I have to take that node from the stack and put it on the top of the stack. If this one has a slightly higher overhead possibly than the counter one than the counter one; that means, just updating the timestamps, but when I am I need to replace a page the stack is lower overhead that stack has lower overhead, why? Because, we do not need to search the entire all pages in physical memory to find the least timestamp page.

And that I do not need to do. I will just go to the bottom of the stack and evict that node out ok. So, the so during the replacement stack is better; however, for both I need to implement both stack and this counter one in hardware memory. So, both these techniques therefore, require extra hardware as memory references are a very frequent phenomena it is impractical to invoke the OS on every memory reference.