

**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 TLB is 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 do not 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 do not need to go into the we do not need to go into the TLB for that, if the if the data is in cache we are fine we are happy. So, we do not 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 addresses 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 do not 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.

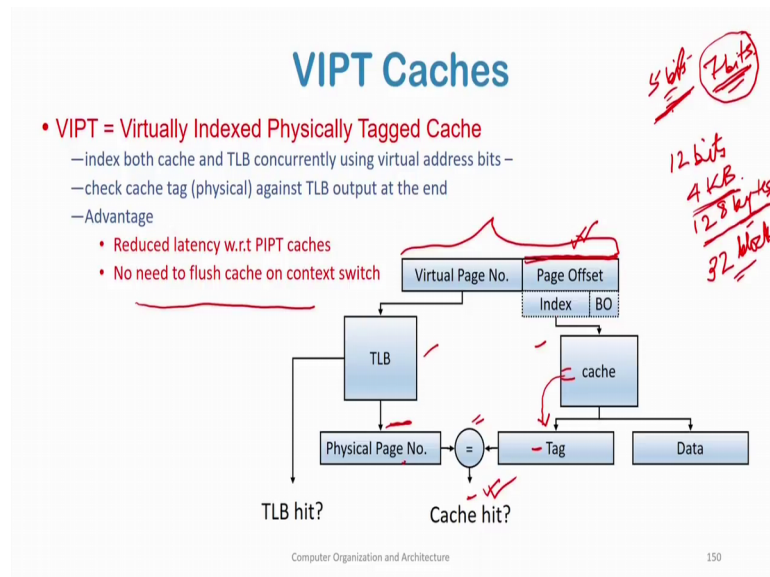
However because these are first virtual addresses so, a block in physical memory can be stored in multiple different locations or multiple different sets in the cache. And this is a problem because the same virtual address can mean different physical addresses by different processors ok. So, therefore, 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 which a different processor 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 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 index 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 use 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 took 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 index physical tagged cache because the TLB and cache are accessed concurrently. With respect to virtually indexed

virtually tagged cache, 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 physical TLB hit 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 to flush the cache on a context switch. So, why because the physical page offset and the virtual page offset are same. 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. So, the page size is 4 KB and let us say my cache block size is 128 bytes. So therefore, I have 8 into 4 32 cache blocks 32 blocks per page I have 32 blocks per page.

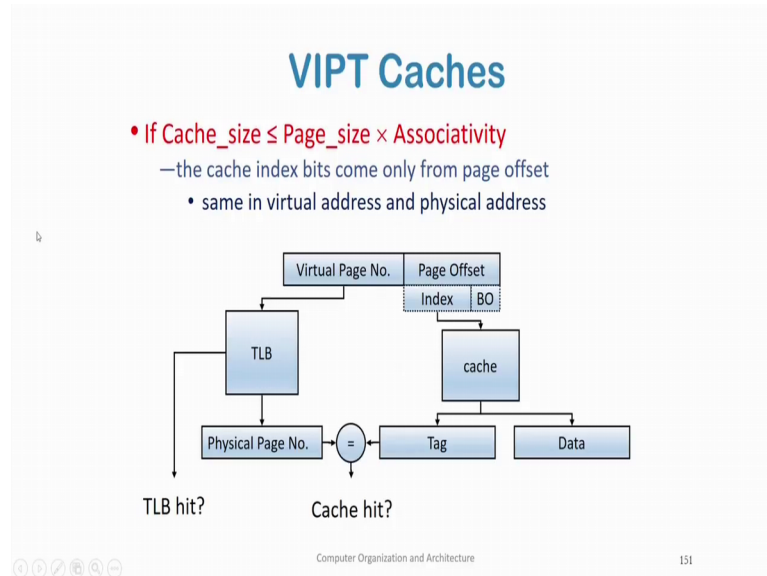
Now, each of these 32 blocks will go to a particular location in the cache, depending on what, so I have 32 128 bytes ok. So, I have 128 bytes in is the block size. So, this will require 7 bits this will require 7 bits and therefore, so this will require 7 bits and the other 5 bits will tell me where 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

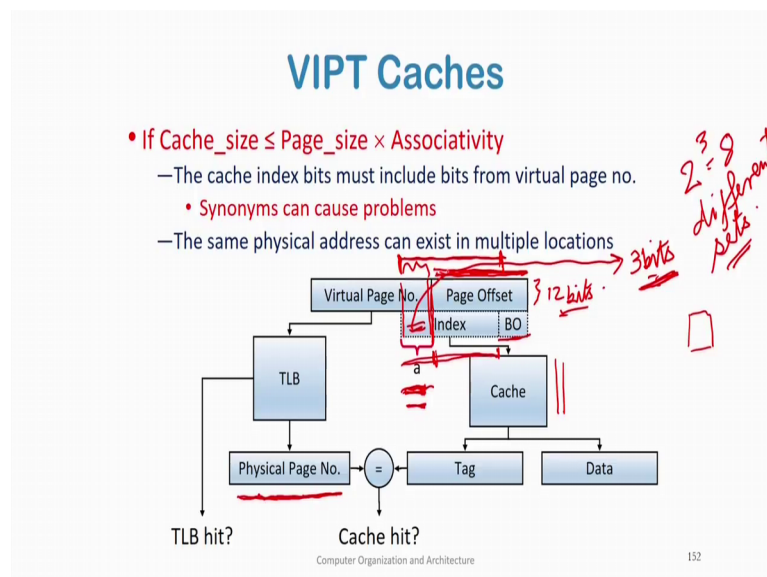


irrespective of what the virtual address is because, the physical page offset and the virtual page is same.

(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 used 3 more bits from the virtual page number to

index the cache, why because my cache was off size. So, I had 12 bits so, 12 bits can access any one of  $2$  to the power 12 locations. So, any one of  $2$  to the power 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 there 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  $2$  to the power 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:  $index(\text{Virtual address}) = index(\text{Physical address})$
  - Called page coloring
    - All physical page frames are colored =
    - 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

Computer Organization and Architecture 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 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 add if the in the equal to the index that 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 I will go and colour each page in the 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 it is colour ok.

Now, what will I do is that. So, each cache block within this. So, the page will be composed of a 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 set 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 the 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)

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

• Consider the following reference string: 

|   |   |   |   |   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 2 | 1 | 6 | 4 | 0 | 1 | 0 | 3 | 1 | 2 | 1 |
| X | X | X | X | X | X |   |   | X | X | X |   |

|   |   |   |   |
|---|---|---|---|
| 0 | 2 | 1 | 6 |
|---|---|---|---|

 $\xrightarrow{4}$ 

|   |   |   |   |
|---|---|---|---|
| 4 | 2 | 1 | 6 |
|---|---|---|---|

 $\xrightarrow{0}$ 

|   |   |   |   |
|---|---|---|---|
| 4 | 0 | 1 | 6 |
|---|---|---|---|

 $\xrightarrow{3}$ 

|   |   |   |   |
|---|---|---|---|
| 4 | 0 | 3 | 6 |
|---|---|---|---|

 $\xrightarrow{1}$ 

|   |   |   |   |
|---|---|---|---|
| 4 | 0 | 3 | 1 |
|---|---|---|---|

 $\xrightarrow{2}$ 

|   |   |   |   |
|---|---|---|---|
| 2 | 0 | 3 | 1 |
|---|---|---|---|

• Fault Rate =  $9 / 12 = 0.75$

Computer Organization and Architecture 158

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)

## FIFO Issues

- Poor replacement policy
- Evicts the oldest page in the system
  - usually a heavily used variable should be around for a long time
  - FIFO replaces the oldest page - perhaps the one with the heavily used variable
- FIFO does not consider page usage

Computer Organization and Architecture 159

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 going 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 recent to 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 it 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 164

So, we looked into least recently used, so, I will not go into that anymore, but we will study what were the problems with you are we also saw this what are the problems with LRU and we said that the problem was in practical implementation, why is in 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 is the what is 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 time stamp 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 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.

So, if I do not implement this counter method or the stack method in hardware I have to implement that in software, meaning that whenever there is a memory reference, I have to go to the OS and update data structures, which is also impractical because memory references are a very frequent phenomena. So therefore, the exact version of ALU is not often used and instead approximate a LRU is commonly used.

(Refer Slide Time: 32:29)

## Replacement Hardware Support

- Use the reference bit in page table for each page
- On a reference to a page, this bit is set to 1
- This bit can be cleared by the OS
- This simple hardware has lead to a variety of algorithms to approximate LRU

So, one of the most common implementations is to use an approximate version of LRU, which uses reference bit in the page table for each page. So, what how does this operate? On a reference to a page this bit is set to 1 ok. So, each page has a reference bit when I am referring to this page when I am accessing this page and this page is in memory I am setting this bit from 0 to 1, if it is not already 1 ok.

Now, at and I have time periods or frames or intervals. So, fixed size intervals after which I check for the after which I set the bits of all pages, I said the reference bits of all pages to 0. So, I when at the start of a time interval, I set the reference bits of all pages to 0 and then within that particular interval every page that is referenced their reference bits are set to 1. And then when the when a particular page has to be replaced, I will try to use a page for which the reference bit is 0. So, a reference bit is 0 means that in the in the current time interval it has not been accessed; that means, it has it is it is of higher probability that it is of higher probability that this page will also not be used in the recent future because, it has not been used in the current timestamp.

This anyway does not mean that this is the least recently used, but it is the it is an approximate LRU to avoid the high hardware cost of LRU, I will just keep one reference bit on each memory reference if that reference bit is 0 I set it to 1 and during replacement I try to find a page for which the reference bit is 0; that means, which has not been used it is with the approx with the approximate assumption that this is not a very frequently used page because, this has not been accessed in the current time interval and therefore, this one could be a good page to replace ok.

Now if all bits are same for a for a given reference bit suppose I find out among all pages for which the reference bit is 0, I may like to select the one which has which was which came into memory at the earliest; that means, first in first out, I will use FIFO for a given value of the reference bit. So, if I get a set of pages with reference bit 0, I can choose the one which came into memory at the earliest ok. So, that will be that will be evicted; so among all pages which have the same reference bit, I use the FIFO strategy to find out the page which needs to be replaced.

(Refer Slide Time: 35:46)

## Sampled LRU

- Keep a reference byte for each page
- At set time intervals, take an interrupt and get the OS involved
  - OS reads the reference bit for each page
  - reference bit is stuffed into the beginning byte for page
  - all the reference bits are then cleared
- On page fault, replace the page with the smallest reference byte

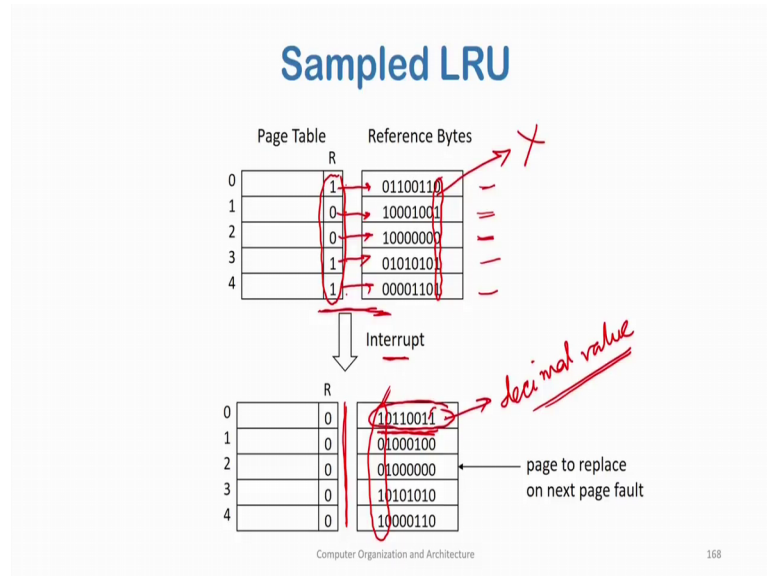
Computer Organization and Architecture 167

The next one is sampled LRU, sampled LRU who is an extension over the approximate LRU which we studied using just one reference bit. And it has slightly higher hardware cost in that, the first byte of each page is used by this strategy. Now what happens is this, that instead of in addition to the reference bit I have a reference byte for each page. So, the first byte of a page will be used as a reference byte reference byte for the page. Now at set time intervals I have similar time intervals as was for the simple approximate LRU I have similar time intervals.

And we take an interrupt and get the OS involved. So, at the end of each time interval there what I was doing, I was I was setting all the reference bits to 0. Here, additionally what I do is I get the OS involved and what does the OS do here? The OS reads the reference bit for each page; the OS reads the reference bit for each page and the reference which is stuffed, so, at the beginning byte for the page. So, in addition I already have the reference bits for each page.

So, at the beginning of the interval I read the reference bit of each page and, stuff it into the reference byte. And then I all reference bits are again clear; very similar to the previous scheme, this one is very similar to the previous scheme, the only difference being that before setting all the reference bits of each page to 0, I copied the reference bits of each page and stuff it into the reference byte. And then on a page fault I replace the page with the smallest reference byte.

(Refer Slide Time: 37:54)



So, how does this scheme work? Now let us say these are my reference bytes for each page. So, these are distinct pages; page 1, page 2, page 3, page 4, page 5 and this is the page table. So, this is the page table and the page table contains my reference bits; these are my reference bits for these pages.

Now, what happens is that now when at the end of so, I mean what has happened is that here, my time interval has come to an end; the current time interval. So, at the beginning of every time interval, the OS takes charge and let us say this one is the end of one time interval, there is an interrupt and the OS has taken charge. What does the OS do? It first throws these bits. So, these the least bits are thrown away and what am I doing? I am stuffing all these bits to these places ok.

So, I am stuffing these bits so, when I stuff this bits, so, see these bits here are the same as the MSBs here, as the MSB here, then I am clearing all my reference bits ok, I am clearing all my reference bits. Now what happens? The values this byte; if I take the numerical value of this byte. So, I take this value of this each of these reference byte; this reference byte. So, I take the numerical decimal value say decimal value of this byte. Now what does this decimal value tell? This decimal value tells me in the last 8 intervals what was the access pattern of it.

And this MSB has the highest weight meaning that it could be that in the last few intervals this page was not used, but this page was used immediately here. So, even if

these bits are 0, I should keep this page. So, and let us say so therefore, the numerical value of this byte tells me how good is this page for replacement; lower the value of this byte better is a better is this page or candidate for replacement. So, I should replace a page having least value for this byte. Why? Because this tells me this byte tells me two things. A very low value of this byte tells me that this page was not accessed recently and this page was not accessed possibly many times recently.

So, this page was not accessed very recently and this page was not accessed possibly not many times recently. So, this is what it tells me the numerical value of this byte. So, I will replace that page having low numerical lowest numerical value of this byte. Again for all pages having the same value of this numerical byte, so, having the same numerical value all pages for which the numerical value of this byte is same, I will choose that page which came into the memory at the earliest. So, among all pages having the same value of this numerical byte numerical value of this byte, I will use the FIFO strategy to choose in for choosing the page which has to be replaced.

(Refer Slide Time: 41:33)

**Clock Algorithm (Second Chance)**

- On page fault, search through pages
- If a page's reference bit is set to 1  
—set its reference bit to zero and skip it (give it a second chance)
- If a page's reference bit is set to 0  
—select this page for replacement
- Always start the search from where the last search left off

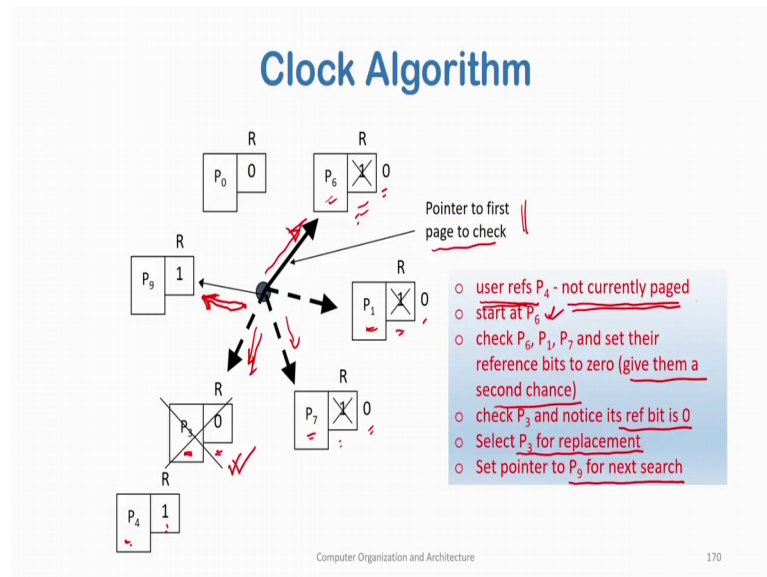
Computer Organization and Architecture 169

So, we will now look at the clock algorithm or the second chance page replacement algorithm. So, it is an extension, it is it uses the reference bits you uses the reference bits in a different way. So, how does this operate? So, on a page fault it searches through pages and then if the pages reference bit is set to 1, then it sets it to 0 and skips it. So, it gives the this page as second chance. So, if it is 1 it does not it does not replace it, but it



sets it to 0 ok. Now if a pages reference bit is 0 this is selected for replacement, if it is 0 then among all pages that is 0 I use the FIFO strategy to replace the one which has 0. Now it searches it starts the search from where the know where the last search was left off.

(Refer Slide Time: 42:38)



So, how does this scheme work? So, let us say the previous search ended with the pointer; that means, the last page which was accessed was page 6 ok, which was accessed which was page number 6. Now this is the pointer to the first page to check. So, for all pages I have a circular link list, for all pages I have a circular link list. And for you suppose the user references page number 4 and that is not currently in the physical memory. So, I have to replace to bring in page 4, I have to replace some page. So, last time the search ended just before page 6, in the physical frame containing page 6. So, now, I will start from the frame containing page 6, P 6. So, this one becomes the first page to check.

Now, therefore so, first I find that the reference bit is 1 and not 0. So, I make the reference between 0 and I do not replace this page, then I come to the next frame; this frame contains P 1, the reference bit is 1 not 0. So, I only set it to 0 and go to the and check the next frame, I come to the next frame, it contains page number 7 and the reference bit is and the reference bit is 1. So, again I set it to 0 and I again proceed, when I come to page when I come to the next page frame I see that it contains page 3 and a

reference bit for page 3 is 0; which means that in the current time interval page 3 was not accessed.

So, therefore, I select page 3 for replacement and after replacement I so, I select page 3 for replacement and it goes to P 4 page number 4 comes in page of page 3 and the reference bit again is set to 1. And then for the next search it now points to this page frame. So, my current search ended at this page frame and then my next therefore, search will start from the next page frame. So, user references page 4 this is not currently page, we start at page 6, we check P 6 P 6 then P 1 then P 7 and set their reference bits to 0, give them a second chance. And then check page 3 and notice that the reference bit is 0 and then we select P 3 for replacement and set the pointer to P 9 for the next search. Now what is good about this algorithm is that if all pages are 1, then ultimately in the next round I will I will select this page.

This one has a low overhead why? Because I am not searching for all pages which has a reference bit of 0. So, possibly which has a reference bit of 0 and I am going on searching for the next page which has a reference bit of 0. So, if all pages have a reference bit of 1, it will the search will circle through all page frames in physical memory and then come back to the first page which was for which the memory bit was set to 0 from 1 and it will be the 1 to be evicted. So, although this page is referenced, but I have been able to give a second chance and because all pages have been recently used I could not find a page with the reference bits 0 and I choose I will choose that one for replacement.

(Refer Slide Time: 46:20)

**Dirty Pages**

- If a page has been written to, it is *dirty*
- Before a dirty page can be replaced it must be written to disk
- A *clean* page does not need to be written to disk  
—the copy on disk is already up-to-date
- We would rather replace an old, clean page than an old, dirty page

Computer Organization and Architecture 171

Now, one aspect which the second chance algorithm does not take care, or ignores is that was is this page dirty or not is this has this page been modified has been written to or not, if a page has not been written to it is dirty. So, before a dirty page can be replaced it must be written to disk. Before a dirty page has been replaced it must be written to disk and this has higher overhead. A clean page does not need to be written to the disk and therefore, it has much lower overhead as replacement I can just discard this page because it is not written and therefore, it does not need to be written back to disk, it can just be discarded and in it is place a new page can come to ok.

The page on disk is already up to date. So, we would rather replace an old clean page old and clean page will rather an old clean page than an old dirty page. So, if I if both pages are old I will choose one which is which is which is clean and which is not dirty, because it will I do not have to write that page back to disk.

(Refer Slide Time: 47:29)

### Modified Clock Algorithm

- Very similar to Clock Algorithm ✓
- Instead of 2 states (ref'd and not ref'd) we will have 4 states
  - (0, 0) - not referenced clean ✗
  - (0, 1) - not referenced dirty ✓
  - (1, 0) - referenced but clean
  - (1, 1) - referenced and dirty
- Order of preference for replacement goes in the order listed above

Computer Organization and Architecture 172

So, now, we will go and see an extension of the modified clock replacement algorithm, in which we use the 2 bits both the reference bits and the dirty bit. It is similar to the clock algorithm, but now each page instead of having 2 states whether it is referenced or not referenced will all have 4 states, whether it is referenced or not referenced and also whether it is dirty or non dirty. So, if a page has its reference bit 0 and it is dirty bit also 0, it means that this page was not referenced in the current interval and is also clean; that means, it is while when replaced I do not need to write this page back on to disk.


If it is 0 1 it is not referenced and but it is dirty. So, it is not referenced in the current time interval, but in the last wherever it was last used whenever, it was last used it was written to therefore, if I choose this page for replacement I need to write this page first back into disk then bring a new page. Then it the next one is 1 0, the page has been referenced in the current interval, but is clean that means, I do not I can discard this page. All; however, it was referenced so, it could be that it will again be referenced in the near future; however, because I have no one to replace I may need to replace this if the other options are not there.

So, and if it is 1 1 it is both referenced and then dirty this one is the is the late least preferred set of pages, if the if a page has the pages which have this 1 1 set they will be the least preferred set of pages to be replaced. So, the order of preference for replacement goes from the one above. So, how will the modified clock replacement algorithm work?

(Refer Slide Time: 49:38)

## Modified Clock Algorithm

- Add a second bit to page table - dirty bit
- Hardware sets this bit on write to a page
- OS can clear this bit
- Now just do clock algorithm and look for best page to replace
- This method may require multiple passes through the list



Computer Organization and Architecture 173

So, add a second bit to the page table the dirty bit, hardware sets this bit on right to a page fine, the OS can clear this bit ok. Now, just do clock algorithm and look for the best page to replace. So, what do you do? In one round of the clock you try to find 0 a page which is 0 0 and in this one if you have for all those pages which are 0 1 you set it to 0 0 ok. So, if you have the least significant bit 1 in the first round you set it to 0 and, you go on looking for a page which is which has a page which has both bits 0. If you find a page which has both bit 0, you use it for replacement. If you see that if both bits are not 0, then you find out whether the least significant bit is non-zero, if the least significant bit is non-zero you set it to 0 fine. Now, in the in the first round if now no page is found, then you find try to find a page for which is which is if you try to find a page which is then 1 0. So, you go on making multiple passes in the order of preference setting 1 bit to 0 at that time and hence you will you will you may require multiple passes to passes through the list to get to a desired page to get the desired page for replacement.

(Refer Slide Time: 51:17)

### Page Replacement - Example $\Leftarrow$

- Consider a computer system with ten physical page frames. The system is provided with an access sequence  $(a_1, a_2, \dots, a_{20}, a_1, a_2, \dots, a_{20})$ , where each  $a_i$  is a distinct virtual page number. Determine the difference in the number of page faults between the last-in-first-out page replacement policy and the optimal page replacement policy.

*Handwritten notes on the slide:*

$a_1, a_2, \dots, a_{10} \rightarrow 10$  page faults.

$a_{11}, a_{12}, \dots, a_{20} \rightarrow 10$  page faults.

Ans = 1

$a_1, a_2, \dots, a_{10}, a_{11}, a_{12}, \dots, a_{20}$  (with  $a_{11}$  to  $a_{20}$  circled)

LIFO = 31 page faults

Opt = 30 page faults

Page hits

$a_{11}, a_{12}, \dots, a_{20}$  (with  $a_{11}$  to  $a_{20}$  circled)

Computer Organization and Architecture 174

So, now before proceeding we will take a small example. Consider a computer system with 10 physical page frames, so, I have 10 physical page frames. The system is provided with an access sequence  $a_1, a_2, \dots, a_{20}, a_1, a_2, \dots$ . Pages  $a_1$  up to  $a_{20}$  are accessed one after the other and again in sequence  $a_1$  up to  $a_{20}$ ; where each  $a_i$  is a distinct virtual page number. So, each  $a_i$  is a distinct virtual page number. Determine the difference in the number of page faults between the last in first out page replacement policy and the optimal page replacement policy.

So, let us first see what will happen in the last in first out page replacement policy? I have 10 physical page frames. So,  $a_1, a_2, \dots, a_{10}$  will result in compulsory misses. So, 10 page faults. So, compulsory page faults 10 page faults ok. Now  $a_{11}$  who so it is last in first out. So, last in is  $a_{10}$ ; so  $a_{11}$  will replace  $a_{10}$ ,  $a_{12}$  will replace who will who will it replace it will replace  $a_{11}$  ok. And then there will be again these 10 page faults up to  $a_{20}$  up to  $a_{20}$  so, 10 page faults ok. Now after this what do you have? You in the in the page frames you have  $a_1$  to  $a_9$ , so, after this sequence after this one is done, you have after the first set of 1 to 20 references you in the in the physical memory you have pages  $a_1, a_2, a_9$  and  $a_{20}$ ; this is what you have. Now, therefore, the next set of 9 accesses do not result in a page fault in the LRU, in the least last in first out page replacement policy these will be hits. So, this will be page hits, this will not be page for this will be hits.

Now, a 10 will again result in a fault. Now who will a 10, a 10 will be who will it replace? So, this is a last in last in first out page replacement policy. So, therefore, so therefore, a 10 will a 10 when it is accessed it will it will be replaced by the one which came in last; who came in last a 20 because, a 1 to a 9 resulted in page hits they were not brought in. So, last in, in this system of in this system that means, in the set of 10 page frames that I have currently this set of 10 page frames that I have currently a 20 is the last in page.

So, a 10 will be again replaced by this one. So, a 10 will replace a 20, a 11 will again replace a 10 will again replace a 10 will again replace a 10 and likewise, a 20 will result in a page fault and it will replace a 19 likewise. So, therefore, how many page faults will you have? You will you will have here, let us see these are 10 page faults again these are 10 page faults and, then from a 20, a 10 to a 20 you will have 11 more page faults. So, this one will be 11 page faults.

So, for the last in for the LIFO strategy for the LIFO strategy, you will have 31 page faults; for the LIFO strategy, you will have 31 page faults. Now when you use the optimal page replacement, you see that this 20 page faults will still be there. In the optimal page replacement, these 20 faults will still be there because these are all compulsory their page faults ok. Now in the in the optimal policy also when 11 comes it will replace page 10 because, it will see in future and find out whom to replace ok. It will find out that a 1 and to a 9 will be replaced will be needed later. So, it will go on replacing them ok. So, in a similar way as the as the LIFO strategy it will also do the same and at the end of the first 20 accesses it will also have the optimal policy will also have a 1, a 2, a 9 and a 20 in the in the memory ok.

Now, what will happen a 1 to a 9 will be page hits, when a 20 comes in now which 1 will which page will the optimal page replacement policy replace? It will replace any one of the pages between a 1 and a 9, but it will not replace a 20 ok. So, why will that happen? Because a 20 it knows that will be needed again. So, a 10, a 11, a 10, a 11, a 12 these when this will be accessed it will go on accessing these one, it will it will keep a 20 in the memory.

So, that when a 20 is needed again later it will be found in the physical memory; a 20 the access of a 20 will not result in a page fault for the optimal page replacement algorithm

because, it has an oracle to know which page will be used in the future. So, a 20 will not result in a page fault and therefore, the optimal policy will incur only 30 page faults. So, the question determines the difference in the number of page faults between the LIFO page replacement policy and the optimal page replacement policy, the answer will be 1, the difference is 1.

(Refer Slide Time: 58:05)

### Belady's Anomaly

- Reference string: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5
- 3 frames (3 pages can be in memory at a time per process):

| FIFO |   |     |
|------|---|-----|
| 1    | 1 | 4 5 |
| 2    | 2 | 1 3 |
| 3    | 3 | 2 4 |

9 page faults

---

- 4 frames:

|   |   |     |
|---|---|-----|
| 1 | 1 | 5 4 |
| 2 | 2 | 1 5 |
| 3 | 3 | 2   |
| 4 | 4 | 3   |

10 page faults

- FIFO Replacement manifests Belady's Anomaly:  
—more frames ⇒ more page faults

Computer Organization and Architecture 175

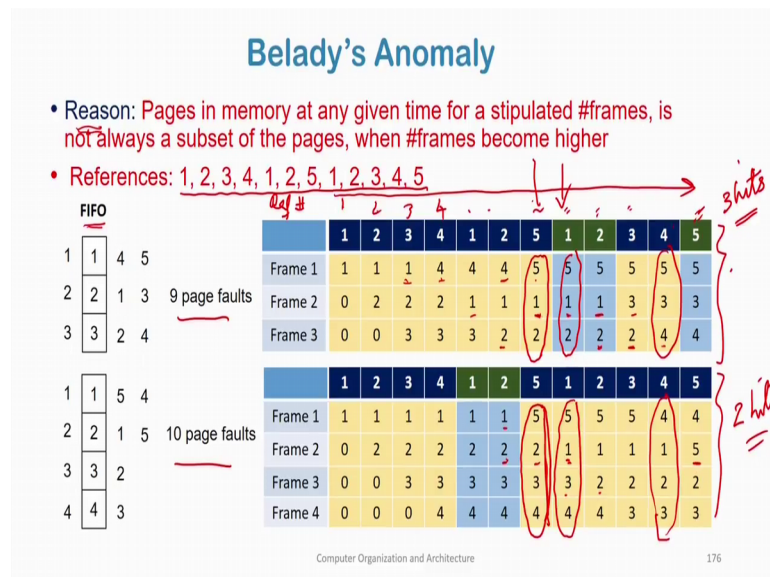
Now we go on to understand one aspect of page replacement, which was important for which is important from a theoretical perspective and, also because this was quite a concern for early page replacement designer page replacement policy designers, who used a FIFO for their replacement algorithms. So, the people who used FIFO will encounter an anomaly which is which is popularly, or commonly known as Belady's anomaly.

Now, what is this? Let us say that you have 3 page frames, let us say you have 3 page frames in memory. Ok. The your whole memory consists of only 3 page frames, in one situation and 4 page frames in another situation. So, in one case you have a system containing 4 frames in memory and in one case you have 3 frames in memory. Now, sometimes it so, happens that for example, for this reference string it so happens that FIFO replacement for the FIFO replacement policy, when you have lower number of frames in memory, you will have lesser lower number of page faults, then when you have more number of frames in memory, you will have more page faults.



Now this should not happen right because, you have more space in physical memory when you have more space 4 frames in physical memory means that you have higher space the capacity of the physical memory is higher, than when you only have 3 frames in physical memory ok. So, the capacity of the physical memory is higher, but you still are encoding higher number of page faults. So, this is an anomaly which is referred to as Belady's anomaly.

(Refer Slide Time: 60:08)



Now we will try to see in more detail why this happens? Now the reason for Belady's anomaly has been found to be this; pages in memory at any given time for the stipulated number of frames is not always a subset of the pages when the number of frames become higher. So, why does Belady's anomaly happen? Because, suppose I have 3 frames, at any given time the number of frames the pages that are there in memory is not a subset of the of the pages that are there when my number of number of frames are higher.

So, when my number of frames are lower, if thus the pages that I am containing are not a subset when my number of pages are higher then Belady's anomaly occur. We will take an example we will take the example of this reference string and see why and how this happens. So, these are the set of memory references and these are the set of memory references shown here, 1 2 3 4 1 2 5 1 2 3 4 5. So, these are the references shown here. Now at time 0, so this 1 is time 0 time 1 time 2 first reference second reference third reference fourth reference like this.

So, ref hash; this one is 1 2 3 4 like this it progresses so, at the first reference. So, I am bringing in page number page 1 to frame 1 and what happens is that it comes to frame 1 and frame 2 and frame 3 are empty. So, there is a page fault; there then 2 comes in and goes to the second empty frame, the frame 2 is empty and 2 goes here, for the 3rd reference 3 comes in frame number 3, for the 4th reference because I am using a FIFO policy first in first out. So, 1 was first in so, 4 replaces 1 and it again incurs a miss. Now, 1 comes in; now, 1 has already been replaced in the previous access. So, 1 again encounters a miss and it replaces 2.

Now, 2 is accessed again, 2 again incurs a miss because, 1 has replaced 2 and therefore, it replaces 3 then 5 is accessed now, 5 incurs a miss. So, the first in among these 3 pages is 4. So, therefore 5 replaces 4 in frame number 1. So, then what happens? Then we find that 1 comes in and 1 is already there. So, therefore, this is green and this is not a miss this is a hit then 2 comes in 2 is already there in frame number 3. So, this is again a hit, then 3 comes in; however, 3 has been replaced and because and because 1 is currently the one with the which is which is earliest which came into the memory earliest. So, 3 is replaces 1 and then 4 comes in and 2 is now the one which is the earliest one which has come into memory. So, 4 replaces 2 and then 5 comes in 5 is already there in memory. So, this one is a hit.

Now, when I have 4 frames in memory these are the set of accesses. Now, what has happened? See so, 1 comes in miss, 2 comes in miss, 3 comes in miss, 4 comes in miss, then 1 comes in 1 is a hit because it is already there I do not need to replace, 2 comes in 2 is already there it is a hit is not a miss, then 5 comes in 5 replaces 1 because 1 is the earliest one which came in it is in the FIFO order 1 is replaced, then 1 comes in; 1 is then 2 is the one in the FIFO order which needs to be replaced.

So, 1 replaces 2 then 2 comes in 2 replaces 3 and then 3 comes in then 3 replaces 4 and all other accesses are miss. Now, we see that when you have when you have 3 frames in memory, when you have 3 frames in memory the number of page faults are 9; you have 3 hits ok. And when you have 4 frames in memory, you have you have only 2 hits. So, therefore, you have 10 page faults, here you have 9 faults and here you have 10 faults.

Now, let us see why this has happened? If you see what has happened. So, at this position I have 5 2 3 4 in the in the 4 frames and, here I have 5 1 2. Now, 5 1 2 is not a

subset of 5 2 3 4 because, 5 1 2 because 5 1 2 is not a subset of 5 2 3 4 when 1 is accessed. So, when 1 is accessed 1 is there in memory here, but at this point in time 1 is not there in the physical memory when I have 4 frames. So, I have a miss here when 1 is accessed, but I have a hit here when 1 is accessed, why because if we go back to the definition again the pages in physical memory at any given time.

So, these are each different times, at any given time the pages in physical memory for a stipulated number of frames. So, pages are so for 3 frames for a stipulated number of frames 3, it is not always a subset of the pages that are there the pages that are there, when your number of frames become higher, when your number of frames become higher the pages are not a subset. For example, here 5 2 3 4 is not a superset of 5 1 2 and therefore, for this case 1 is a hit and for this case it is not.

For the same reason again 2 is a hit so this 5 1 3 4 is so, 5 1 2 is not a subset of 5 1 3 4 and therefore is this 1 was a hit here, but this resulted in a miss. Again in this case 5 3 4 is not a subset of 4 1 2 4 1 2 3 and therefore, when 5 is accessed here 5 results in a miss here however, 5 is a hit here ok. So, Belady's anomaly occurs because, pages in memory at any given time for the stipulated number of frames is not always a subset of the pages when number of frames become higher.

(Refer Slide Time: 67:41)

**Belady's Anomaly**

- The optimal algorithm and LRU do not exhibit Belady's anomaly
- The optimal algorithm always maintains the most frequently used pages to be accessed in future and, *recently*
- LRU always maintains the most frequently used pages accessed in the past, *recently*
- Irrespective of the number of frames
- Most frequently used pages for a lower number of frames is always a subset of the pages when we have a higher number of frames

Computer Organization and Architecture 177

Now, the optimal algorithm and LRU both the optimal algorithm and the LRU do not exhibit Belady's anomaly. The optimal algorithm always maintains the most frequently

used pages to be accessed in future. So, optimal algorithm what does it maintain? At any point in time the optimal algorithm maintains the most frequently used pages sorry, the most recently. The most recently used the most recently used to be used pages to be accessed in future and this one LRU keeps the most recently used pages most recently most recently used pages accessed in the past.

So, because LRU at any given in time point at any given point in time, whatever be the number of frames that are available. What does an LRU do? It replaces the least recently used page. So, therefore, what are the pages that are there in the memory at any given time the most recently used pages are there and what does optimal algorithm keep the most recently used pages that will be accessed in future. Whatever, I have written here may not be exactly correct, but I am telling it that optimal algorithm will keep at any point in time the most recently to be accessed pages in future and LRU keeps the most recently used pages accessed in the past.

Irrespective of the number of frames: So, irrespective of the number of frames both these algorithms keep the most recently accessed pages. Now most recently used pages, most recently used pages for a lower number of frames is always the subset of the pages we have for a higher number of frames. Suppose I have a certain number of frames and, I and at a given time I have a certain number of pages in memory ok. Now the most recently you so, let us say I am using the LRU algorithm. So, the most recently used pages will be there in memory for 3 frames.

Now the most recently used pages at that point in time for 4 frames will always be a subset, will always be a superset of the most recently used pages if I have 3 frames. The most recently used pages for 4 frames will always be a superset of the most recently used pages for 3 frames. This is why Belady's anomaly is not there in the least recently used algorithm neither is it therefore, the optimal algorithm.

(Refer Slide Time: 70:38)

## Page Buffering

- It is expensive to wait for a dirty page to be written out
- To get process started quickly, always keep a pool of free frames (buffers)
- On a page fault
  - select a page to replace
  - write new page into a frame in the free pool
  - mark page table
  - restart the process
  - If selected page for replacement is dirty
    - write dirty page out to disk
  - place frame holding replaced page in the free pool

Computer Organization and Architecture 178

Now, we will go into another concept called page buffering. Now, page buffering it is expensive to wait for a dirty page to be written out. Now, let us say during a replacement I will need to replace a dirty I have to replace a dirty page. Now, it is expensive to wait for the dirty page to be written out. Instead what we can do, to get this process quickly started we can always keep a pool of free frames. Now, these frames are free, how do we make it free on a page how do we do this business? On a page fault, we select a page to replace and then write a new page into a frame in the free pool. So, at any given time I have a pool of free frames.

So, there I can just close my eyes and bring a page into this free pool without looking into anything else. But how do I maintain this free pool? I will first choose a page for replacement, mark the page table restart the process after replacement. So, I have brought a page from the disk and put it into this free frame, but then I take this page which I have used for replacement and put it into the free frame pool. So, at any point in time, I have a kitty of free frames that are ready for replacement. And for the current access I am using a free frame from by free frame pool, but I am using a free frame. So, I will replenish that free frame pool by one frame after replacement.

So, I will restart the process after replacement and then I will quietly do my business of creating one more free frame. So, if the selected page frame is dirty, then after replacement I will I am I will I will I will I will put that page into this and then put this page

into the free frame pool and because this dirty page was not put into the memory during replacement, this overhead is not visible to the page replacement. And this dirty page by putting this dirty page into the disk happens after the replacement and after the process has started and when this process is working in the CPU, I am replacing this dirty page into the disk I am putting this dirty page into the disk and creating the free frame pool. So, place frame holding the replaced bit in the free pool ok. So, this is how buffering will work.

Now, one more important aspect is that one more enhancement that we can do with this is that, when I am replacing when I am replacing I may not just I may not destroy the contents of the page, I have put it in the free frame, I have done the replacement is if required, but in the free frame let us keep the page intact do not destroy the contents of the page. If that page is required to be replaced in any case this page is clean and can be used because, it is there in the free frame pool, but by chance if this free frame which I have kept it in the free frame pool but by chance let us say this page is accessed by one process.

Now I can keep a pointer that this page is still there although it is there in the free frame pool, it is it is actually there in physical memory still now, it has not been replaced. So, I can I can access it from the free frame pool itself. So, I will get the page in main memory I have to I do not have to go to the disk to get this page. So, even if I put pages in the free frame I will just I will just keep a mark at as to what pages are there in the free frame pool at any given time and if there are access I will be able to put that use that page from the free frame and use that instead of going to the disk.

(Refer Slide Time: 74:58)

### Allocation of Frames

- Each process needs minimum number of frames
- Two major allocation schemes
  - fixed allocation ✓
  - priority allocation ✓
- Many variations

|||

Computer Organization and Architecture 179

Now, allocation of frames; now still now we are saying that a page does not a page frames have no connection with the with the processes. Now, this is entirely not true. So, each process requires a minimum number of frames. So, typically what do we do is that we allocate a certain number of physical page frames to each process. However, this is done using many schemes, there can be a fixed allocation scheme, there can be there can be a prioritized allocation scheme and it has many variations.

(Refer Slide Time: 75:39)

### Fixed Allocation

- Equal allocation – Example, given 100 frames and 5 processes, give each process 20 frames
  - Keep some as free frame buffer pool
- Proportional allocation – Allocate fairly based on process size
  - Dynamic as degree of multiprogramming, process sizes are subject to change

- $s_i$  = size of process  $p_i$  ✓
- $S = \sum s_i$
- $m$  = total number of frames
- $a_i$  = allocation for  $p_i = \frac{s_i}{S} \times m$

$m = 64$  ✓  
 $s_1 = 10$  ✓  
 $s_2 = 127$  ✓  
 $a_1 = \frac{10}{137} \times 62 \approx 4$  ✓  
 $a_2 = \frac{127}{137} \times 62 \approx 57$  ✓  
205

Computer Organization and Architecture 180

For example, the first strategy is to use the fixed allocation scheme. In the fixed allocation scheme let us say I have 100 frames in physical memory after these 100 frames are available after allocating frames to the OS and I have 5 processes. So, the degree of multiprogramming is 5, I give each process 20 frames, I divide the frames into this into this 5 processes. So, this one will be called a fixed allocation, fixed allocation. Otherwise we can do a proportional allocation, I allocate fairly based on the process size. So, for example, let us say let  $s_i$  be the size of process  $p_i$  and capital  $S$  be the total size of the process  $p_i$  in say number of pages.

And  $S$  is the summation of the pages summation of the number of pages over all processes that are currently there that are currently active. And let  $m$  be the total number of frames that are there in physical memory. So, then the number of frames that will be allocated to a certain process to process  $p_i$   $a_i$ , the number of frames  $a_i$  that will be allocated to process the process  $p_i$  will be given by  $s_i$  divided by capital  $S$  into  $m$ . So, this will be so I allocate a number of frames proportional to the size of the process. For example, let us say I have 64, I have 64 frames in physical memory and  $s_1$  has a size of 10 virtual pages,  $s_2$  has a size of 27 virtual pages that it requires.

So, therefore  $s_1$  plus  $s_2$  is 137 so, total number of pages summation  $s_i$  is 137 and out of which  $s_1$  has a size of only 10. So, out of so I allocate and let us say 2 pages out after out of this 2 this 64 minus 1 is 64 minus 2, so 2 pages I am using for OS. So, out of this remaining 62 pages out of this remaining 62 pages I will allocate 4 pages to process  $a_1$  and I will allocate the remaining 57 pages to process  $a_2$ . Why? Because  $p_2$ , the process  $p_2$  has a much larger size  $s_2$  equals to 127, 127 pages.

So, that the number of frames that is expected to that this page process 2 is expected to require is much more than  $p_1$  because,  $p_1$  has a much smaller size with respect to in terms of the number of pages  $p_1$  has a much smaller size. So, I will allocate when allocating frames, I will allocate  $p_1$  only 4 page frames and, I will allocate  $p_2$  57 page frames because, their sizes are also very skewed.

Now, if I allocate the same number of frames we understand that the memory the memory utilization will not be that good.



(Refer Slide Time: 79:25)

## Priority Based Allocation

- Use a proportional allocation scheme using priorities rather than size
- If process  $P_i$  generates a page fault,
  - select for replacement one of its frames, if available
  - Otherwise, select for replacement a frame from a process with lower priority number

Computer Organization and Architecture 181

Now the next is priority based allocation. Now, priority based allocation is proportional allocation using priorities, now instead of instead of sizes I use priorities. So, if process  $P_i$  generates a page fault select for replacement 1 of it is frames is available, otherwise select for replacement of frame from a process with lower priority number.

So, now, what it does? So, I have allocated let us say based on proportion I have first allocated the frames to different processes. Now, during replacement, if I have free frames allocated to this process I do a local replacement, I will choose a frame from the frames I will choose a page for replacement from the frames which are allocated to this process. However, if none of the frames are free. So, if all frames allocated to this processes are busy; that means, are allocated and no frame is free, then I will choose a frame the free frame from a lower priority process. So, I will take a frame from a lower priority process ok.

(Refer Slide Time: 80:45)

## Thrashing

- If a process does not have “enough” pages, the page-fault rate becomes high
  - Page fault to get page
  - Replace existing frame
  - But quickly need replaced frame back
  - This leads to:
    - Low CPU utilization
    - Operating system thinking that it needs to increase the degree of multiprogramming
    - Another process added to the system
- **Thrashing** : *When a process spends more time swapping pages in and out of memory than actual execution on CPU*

Computer Organization and Architecture

182

This is the priority based allocation scheme. Now, after this we come to the concept we understand the concept of thrashing. So, this we will continue in the next lecture.