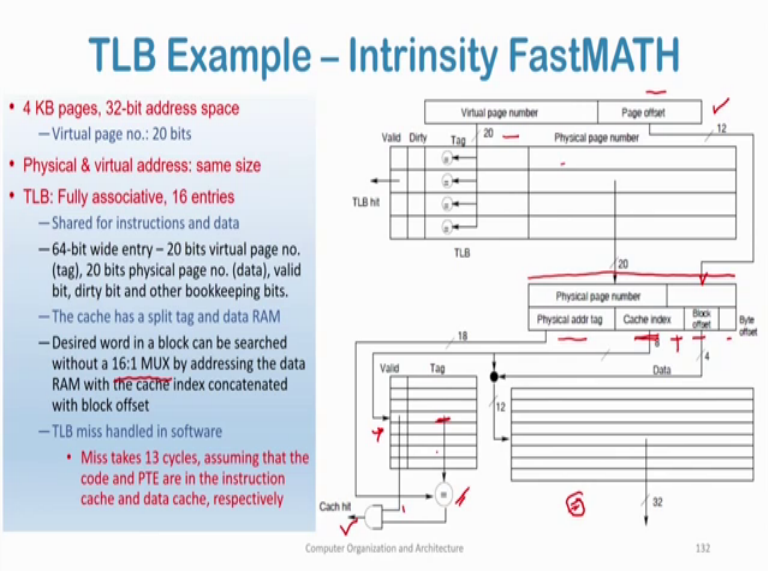
Computer Organization and Architecture: A Pedagogical Aspect

Prof. Jatindra Kr. Deka
Dr. Santosh Biswas
Dr. Arnab Sarkar
Department of Computer Science & Engineering
Indian Institute of Technology, Guwahati

Lecture – 30
Cache Indexing and Tagging Variations, Demand Paging

Welcome. In this lecture, we continue our discussion with Virtual Memories. Towards the end of the last lecture, we took an example of a practical architecture of intensity fast math. We will begin today's lecture by briefly recapitulate in that example.

(Refer Slide Time: 00:49)



So, we said that the intensity fast math architecture consists of a 32 bit address space in which you have a 20 bit virtual page number and 12 bit of page offset. And this 20 bit of virtual page number is goes to the TLB and is matched in parallel to a fully in a fully associative TLB. And if there is a tag match corresponding to the virtual page number, you generate a physical page number; the physical page number is also 20 bits. That means, the virtual the virtual address space and the physical address space has the same size and the page offset goes unchanged into the physical address.

So, we generate the physical page complete physical address here. And, after generating the physical address we go for the cache access and we do. So, by dividing the physical

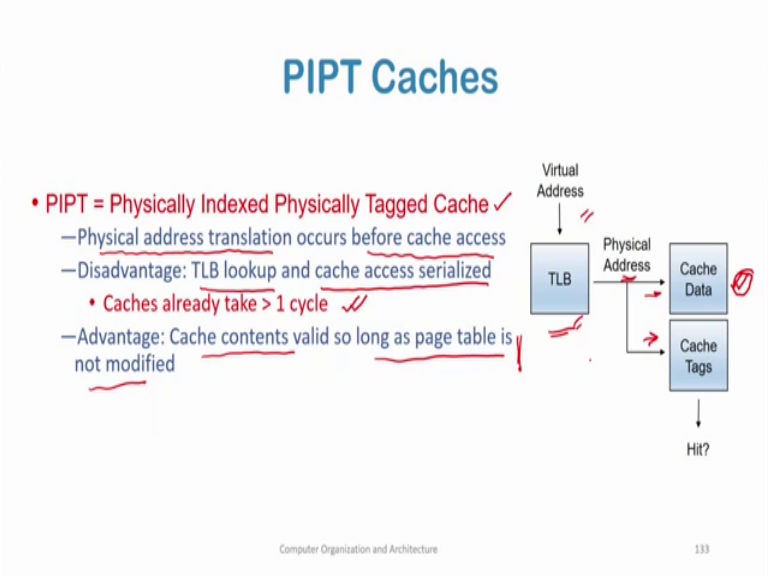
page address physical address in to different parts one is the physical address tag the cache index the block of certain byte offset. And we said in the last class that we looked at the cache, we looked at a split cache in which the tag part and the data part of the cache was divided physically, logically, it is one cache, but physically the tag part and the data part is divided and we said why we did that.

So, the physical address tag. So, I use the cache index and to go into the tag part and index the tag and when there is a match corresponding to this cache index, I take the tag value, I match the tag value with the physical address tag and then I also match with the valid bit and if the valid bit is on here, I get a cache hit. So, if the valid bit is on and if the tag matches with the physical address tag here and I combine I get a cache hit here.

And I said that I have I had, we have divided this tag part and the data part because we club the index and block offset to generate a 12 bit indexing of this data part of the cache. So, instead of a 8 bit indexing of the cache. So, we have a split tag part and a data part and we said that we had divided the data part to directly access the word within a cache instead of a block.

So, when we address this data part where the cache index is appended with the block offset and we use this 12 bit in indexing of the data part, I directly go into a word otherwise if we did not do. So, what we would have what would happen is that we would we would index the data part and we go into we would go into a block and then we would require a 16 cross 1 MUX to go into the particular required word within that block. So, by keeping this tag and data split into 2 parts; we can do away with the 16 cross 1 MUX.

(Refer Slide Time: 04:21)



Now, the point for starting with this example again is to reiterate that this was a physically indexed physically tagged cache that we were looking into. So, what is a physically indexed physically tagged cache? So, in a physically, index physically tagged cache the physical address translation occurs before cache access. So, first I take the virtual address go into the TLB generate the physical address and based on that physical address, I access the cache this is what happened with the intensity fast math architecture that we just looked.

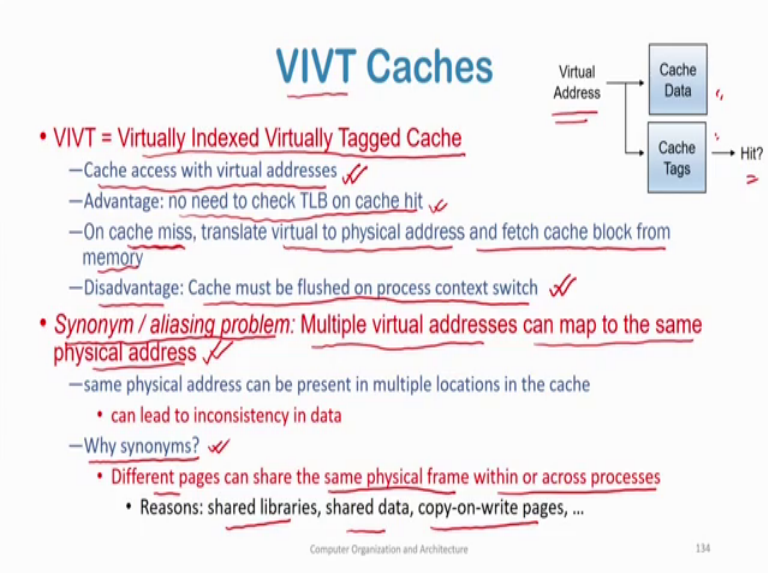
The only problem with this architecture is that the TLB comes into the critical path of data axis. So, suppose even if I have the data in cache, I have to go through the TLB and then obtain the physical address and then be able to access the cache, if the page is not present in the TLB, if there is a TLB miss, then we have to go to the main memory to fetch the page table entry required page table entry and get the physical address.

So, even if the data is there in cache we may need to go into memory, because the page table entry corresponding to this corresponding to this data is not present in the TLB. So, there is, so the caches. So, this is the disadvantage of TLB; lookup TLB lookup and cache access gets serialized and the cache takes greater than one cycle time and it may take multiple cycles, because if there is a TLB miss I need to go into the main memory assuming that the page table is stored in main memory in this architecture. So, I have to

go to the main memory, bring back the page table entry fill the TLB bring it to the TLB and then access again get the physical address and then go into the cache.

So, this happens even if the data is present in the cache; however, the advantage of this scheme is that cache contents remain valid. So, long as the page table is not modified, we will be able to appreciate this advantage a bit later when we go into seeing how this problem of this TLB being within the critical path of data access is solved.

(Refer Slide Time: 06:56)



So, we try to solve; that means, we try to do away with the we try to take the TLB out of critical path by using virtually addressed caches and the first type we will look into is the virtually indexed virtually tagged cache.

So, we look into the virtually indexed virtually tagged cache. So, instead of so, what it directly from the name, we understand; what happens is that instead of using the using a physical tag address and a physical indexing of the cache, I use the virtual address to both index and tag the cache I.

So, therefore, because I directly use virtual addresses I go to based on the virtual address, I break the virtual address again into tag part and index part and go into the data and tag part of the cache ok. So, the cache access is done you with the virtual addresses in virtually, indexed virtually tagged caches the advantage is that we do not need to check TLB on cache hit because I have a process that process generates virtual addresses

directly based on the virtual addresses, I will go that corresponding to these virtual addresses; address do I have the data corresponding to this virtual address.

So, the page the physical address could be anything, but that has been brought into the cache because that has been brought into the cache. Now, it is stored corresponding to the virtual address that data. So, I directly understand that corresponding to this process corresponding to the virtual address generated by this process is does the cache content my required data or not, I do not need to go to the TLB I do not need to I do not need to go to the TLB to address to get the physical address.

However on a cache miss, I need to do that on a cache miss I need to translate the virtual address to physical address by going through the TLB if there is a TLB means going to the memory bringing back the page table entry and then fetching the cache block from memory because on a cache miss, what I have to do I there is there is no there is if when there is a cache miss the data, I required is not in cache. So, I have to go to physical memory and bring back the data to cache. So, how do I do that for that I need to know the physical address of this data how do I do that? I the only way is to go to the TLB get to the page table entry get the physical address and then go to memory bring back the data and put into the appropriate entry in cache such that the virtual address next time can be able to access the data that I need.

The disadvantage of this scheme there are there are a few disadvantages the first big disadvantage is that the cache must be flushed on process context switch. So, remember that each process has the same virtual address space. So, it is very common that the same set of virtual addresses will be generated for each process and the virtual addresses of different processes may be meaning different things, it is local to the process virtual addresses are local to the process.

So, therefore, when there is a contact switch, I cannot keep the cache contents anymore, I have to flush the cache and I have to flush everything that was there in the cache; so right. And therefore, when the new process comes in, again, I will have a set of compulsory misses, I need to I will incur a set of compulsory cache misses always right. So, this is the first disadvantage that the cache needs to be flushed at every context which when I use virtually indexed virtually tagged caches.

The second problem is that of synonym or aliasing, it is called the synonym problem or the aliasing problem. The problem is that multiple virtual addresses can now map to the same physical address. So, the same physical address can be present in multiple location in the cache. So, the same physical address because what has happened is that suppose there are 2 virtual addresses of the same process of the same process, I have 2 virtual addresses which point to the same physical memory location.

Why can how can that happen that 2 virtual addresses can point to the 2 virtual addresses can point to the same physical memory location because can share same physical page frame within or across processes reasons we have, we saw we have shared libraries share data copy on write pages. So, in case of a for example, let us say I have a shared printf function which is called from different portions of the process.

So, instead of keeping different versions of that same printf function copy the same printf function, whenever I need, I can keep in physical memory one version of the printf function, the code of the printf function and then I can call I can have stubs within the virtual memory which will call the same printf function from physical memory, if it did not have this; what would what would I have to do? I had I will have to keep several versions of the printf function. So, whenever it is called whenever I when whenever my process prints something, I need to keep the code of the printf.

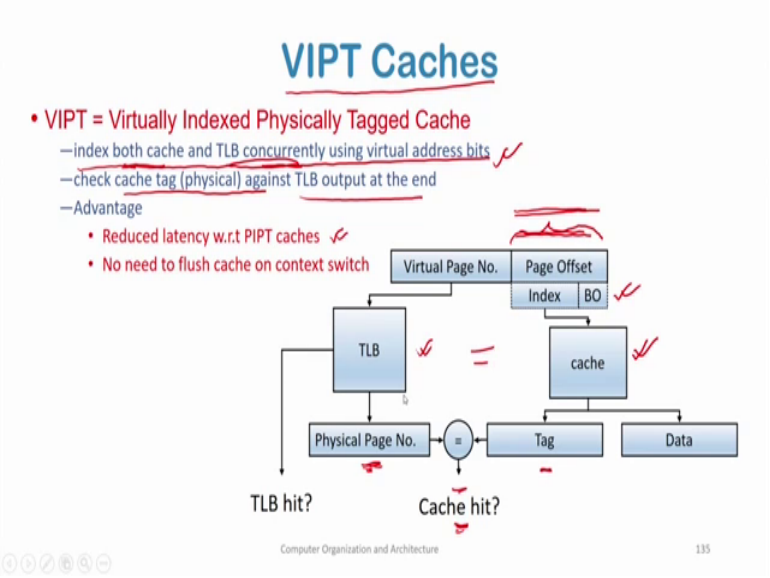
Now, now it is better to share this code of printf and keep it in one place. So, what does this sharing mean? This sharing means that different virtual addresses will now map to the same printf function which is the same set of physical locations in the physical memory. Therefore, multiple virtual addresses will point to the same physical address in this case and because I have a virtually addressed cache this same data the same. So, therefore, I will what I will have is that the same data or the same code will be there will be accessed from multiple virtual addresses.

So, therefore, the same physical address can be present in multiple locations in the cache. So, data corresponding to the rather, we can we can say it more precisely that data corresponding to the same physical address can be present in multiple locations in the cache now this may lead to potential inconsistency because it actually means the same physical location in the physical memory suppose one virtual memory writes in writes in to their data and the other one reads it.

Now, I have 2 copies of the data in cache; both meaning the same physical memory location, yes, both meaning the same physical memory location, I have 2 data elements and these 2 data elements are mapped by different virtual addresses. So, therefore, in a virtual address cache they will be stored twice in different locations in the cache, and because I have the same physical data element in two points. In 2 places within the cache, this may lead to potential data inconsistencies. So, this may lead to potential data inconsistency.

So, these are the 2 major problems with virtual index virtually tagged caches. So, it tries to solve the problem of bringing or of taking out TLB from the critical path of a data access this is why this was the motivation of bringing in virtually addressed caches, but it introduced 2 new problems. The first one was that of was that the cache she needs to be flushed at every contact switch. And that may lead to potential latencies due to cache misses of a later process which could possibly have been potentially some of these cache. Misses can be avoided if I did not have to flush the cache and the second of the second problem was that of synonym or aliasing which meant that the same physical location can the data corresponding to the same physical location can be present in multiple locations in the cache being pointed to by different virtual addresses and this may lead to inconsistency of data.

(Refer Slide Time: 16:01)



Now, to handle these problems; so, to handle these problems while keeping the advantage, people looked into virtually indexed physically tagged caches. So, in this what happens? Both the in the index both cache and TLB concurrently using virtual address bits. So, what happened is in previously was that previously what happened was that I used the virtual address and using the virtual I broke the virtual address and then for the tag and data, I used the virtual address for both the tag and the indexing of the cache.

Now, here what happens? I will I will do the indexing of the cache principally using the offset part of this virtual address ok. So, this part of the virtual address will be used for indexing the cache. So, it is virtually indexed because I use the virtual address to index the cache this is virtua, this is virtual indexing.

Now, in parallel when I am indexing the cache at the same time I take the virtual page number go into the TLB and get the page number physical page number, if there is a page hit, I go to get the physical page number, otherwise, I need to go to the main memory, but if there is a page hit say, then I get immediately get the page number and therefore, I can match with the tag that is the tag that is available here and understand if there is a cache hit.

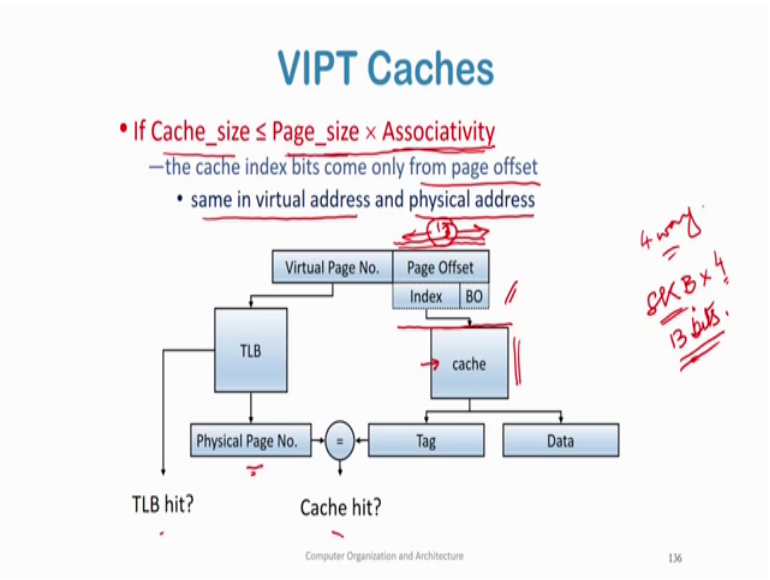
So, essentially there is no latency involved I am still. So, I am still having the advantage that I had with virtually indexed and virtually tagged caches why because this TLB access and the cache access this cache indexing and the TLB indexing is happening in parallel concurrently in hardware. And therefore, if there is a TLB miss, I just index the cache get the data and to check whether this cache this was a cache hit subsequent. So, after this indexing is done I just checked the page number obtained from TLB does it match with the CAC tag part of the cache if there if it is. So, I get a cache hit.

So, so, both; so, I index both the cache and the TLB concurrently using virtual address bits and then check the tag physical cache tag against TLB output at the end. Here the advantage is that I have reduced latency with respect to physically index physically tagged caches, because I do not generate the whole page number. And then a subsequent to generating the entire page address, I access the cache, I do not do that.

I am doing this in parallel and we do not need to flush the cache on a contact switch why is it. So, I do not need to flush the cache on a contact switch because the page offset the

page offset corresponding to the page offset corresponding to the virtual address remains unchanged in the physical address and therefore, there the problem of synonym if this is the case here, what the case has the case for the case here; I have no problem of synonym either.

(Refer Slide Time: 19:41)



So, now what has happened the first scenario what we actually looked at in the last slide where I had completely been able to avoid the problem of synonym was this case was this case. Now elaborate this case one this case. So, this case can happen only if the entire cache can be indexed by only using the page offset bits of the virtual address. So, if the cache can be fully indexed by only using the page offset part of the virtual address, I have no problem of synonym as we will discuss in detail.

Now, when does this happen; when the cache size is less than the page size into associativity. So, when the cache size is at most page size into associativity, then I can use only these bits why because page size tells me; how many bits I use for the offset, let us say, I have I have a page size of 8 kb, then I use 13 bits for indexing the page I have 13 bits for this one, this part has 13 bits, 13 bits. And then let us say, it is a 4 way set associative cache, if it is a 4 way set associative cache, then the size of the cache is. So, I have 4 into 8 KB is the size of the cache 4; 4 into 8.

So now, what happens these this associate this 4 way sets; set associative means that I will go to the particular I only need to identify the set in the cache using these 13 bits, I

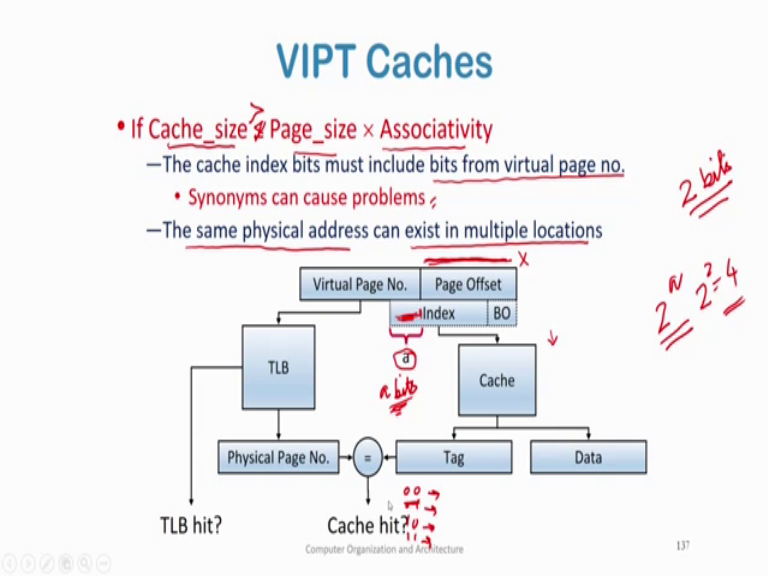
will be able to find the set in the cache and all the 4 sets all the 4 blocks within this set will be searched in parallel to find a cache hit. So, therefore, if the cache size is less than page size in through into associativity, I can only use this page offset part of the virtual address to index the cache. And I do not have any problem of synonym in this case, why because this page offset part the cache index will come only from the page offset and this page offset is same both in the virtual address and the physical address.

Now, why is this not a problem because now the page offset will tell me which set in the cache I need to go find and then the virtual page number I will go to the TLB, I will get the physical number and then based on the page offset I have indexed the cache. So, the page offset plus physical page number then tells me my data.

So, it is as if it is very similar to the physically indexed physically tagged cache in its operation because in the I have I cannot have a case in which the same physical memory location can exist in two location multiple locations in the cache why, because this page offset part of the physical memory tells me where in the cache I will have. And from that cache I will just take check for the tag and this check for the tag will be based on page number and I will get a cache hit R.

Now, this problem is solved only therefore, for small caches. So, if the I can so, because I only have these 13 bits essentially to address the cache to index the cache in this case the number of page offset bits, I can only use to address the cache either I have to increase page sizes or I have to keep the cache very small now both has limits.

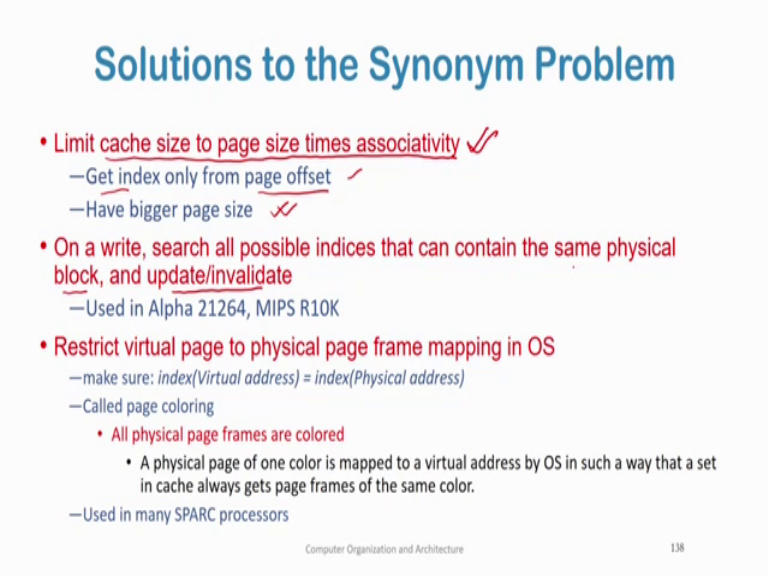
(Refer Slide Time: 23:50)



So, therefore, sometimes what happens in the so, in practice, what happens is that I have to have a cache who size whose size is such that it cannot be indexed solely by the bits in the page offset part, I cannot use only this to access the entire cache. So, I have to borrow a few of the page virtual page number bits as well and here in again comes a bit of a problem.

So, when the cache size is greater than page size into associativity when the cache size is greater than page size into associativity, then only these offset page offset bits cannot is not sufficient to index the cache the cache index bits must include bits from the virtual page number. So, this part, now here I can have, I will again have the problem of synonyms which I had for virtually index virtually tagged caches. So, therefore, the same physical address can now exist. Again this problem has come back the same physical address can exist in multiple locations in the cache.

(Refer Slide Time: 25:10)



Now, how to people try to solve this problem of synonyms now the first way as we told is that you limit cache size to page size times associativity you do this we already discussed this. So, how do you do that? So, then you get the index only from the page offset part have a bigger page size or small caches, what is the second solution approach on a right search all possible indices that can contain the same physical block and update slash invalidate ok. So, what am I doing in this case let us say I have a bits here. So, what are the different sets in cache in into which the same physical address can belong the same physical address can belong in at most 2 to the power a different sets ok.

For example, let us say a equals to 2 bits so; that means, so; that means, that to index the cache I have used parts of the page offset for this part for this part the physical address and the virtual address is same, I have no problem for these 2 bits, this a part is 2 bits, this these 2 bits, I can have 4 different sets in which the same physical address can reside I can have 2 to the power 2 equals to 4 different places in which this a particular physical page can then reside.

So, now, for all these 4 locations for all these 4 sets I have to check. So, on a write on a write what happens we have to search all possible indices that can contain the same physical block on I want to write some data into the cache and then what happens during this, I have to search all possible places to retain consistency so that the same physical address is not present in multiple locations in the cache, I have to check in all possible

positions in the cache where this data can reside. So, I have to check all the 4 different sets in which this particular in which this particular data can reside and this technique is used in architecture such as alpha and MIPS; this architecture are 10 K.

The third strategy is by restricting the virtual page to physical page frame mapping in the operating system ok. So, here I am restricting the placement of physical page frames into the cache, what do I want to ensure that I will derive the same set in the cache by indexing the virtual address as I would do by indexing the physical address this is what I want to ensure and this is done through a mechanism called page coloring.

So, in this scheme, I statically colored the physical I statically color all physical page frames into using different colors. So, what will be the number of colors the number of colors will be at least equals to this 2 to the power a so; that means, if I have 2 bits here I will statically color the physical page frames in physical memory with 4 different colors. So, of a set of page frames will be in red the say another set of page frames will be black another set of page frames will be blue another set of page frames will be yellow. So, I will use 4 different types of colors corresponding to each page frame. So, if the page frames in physical memory will be separated into 4 sets and each set will get a different color ok.

Then a physical page of one color so, after coloring is done a physical page of one color is mapped to a virtual address by the OS in such a way that a set in cache always gets page frames of the same color. So, a physical page of one color so, I have already statically colored; I have already statically colored the page frames ok. Now a physical page of one color is mapped by the virtual address map to a virtual address. So, what will happen? The OS will allocate physical pages to virtual addresses column. So, when I need physical addresses.

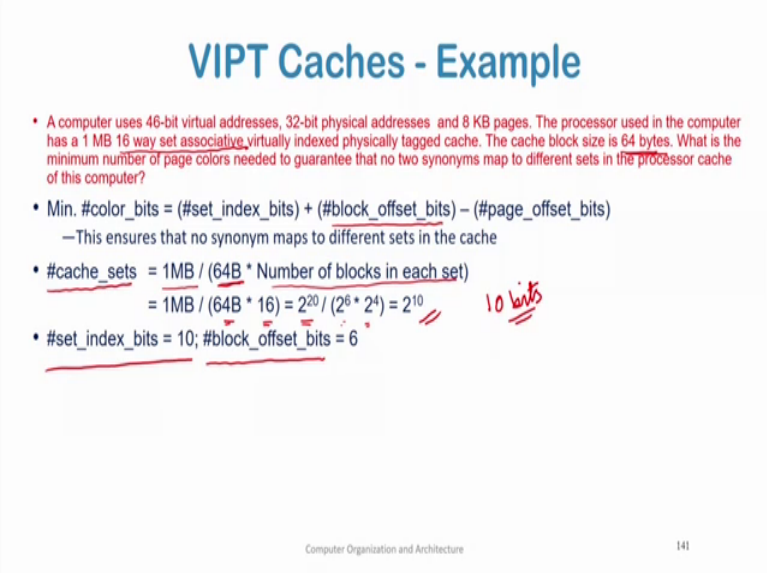
So, now, it will restrict those will restrict which virtual addresses can get which physical pages which virtual page numbers can get which physical page numbers there will be a restriction on that what will be the restriction it will map, such physical it will map physical page frames to virtual pages in such a way that a set in cache always gets page frames of the same color that a set in cache always gets page frames of the same color.

So, my vert suppose, these 2 bits there are of these a bits I can have what I these 2 bits can be can be 0 0 0 1 1 0 and 1 1. So, I have 4 colors one color is 0 0 other is 0 1 1 0 and

1 1. So, what will happen it will, if this virtual address is 0 0 I will only I will only allow I will only allow. So, if a virtual address or virtual page number or virtual address has this particular bits 0 0, I will only allow pages with color 0 0, let us say 0 0 is red, I will only allow pages of colors 0 0 to be mapped to this virtual address by this, I will be able to ensure that corresponding to this virtual address or the physical address the physical addresses will always be mapped to this same set ok, this is what is the concept of page coloring.

So, a physical page of one color is mapped to a virtual address the physical page of one color is mapped to a virtual address by the OS in such a way that a set in cache always gets page frames of the same color though. So, a set in cache will always get page frames of the same color. So, this is how I avoid the problem of synonyms in virtually indexed physically tagged caches and it is used in many sparc processors as well.

(Refer Slide Time: 32:29)



So, before proceeding further we will take an example with virtually indexed physically tagged caches. So, I have a computer which uses forty 6 bit virtual addresses it uses 32 bit physical addresses and an 8 KB and 8 KB pages. So, page size is 8 KB physical address space is 32 bit virtual address space is forty 6 bit and the processor used in the computer has a 1 MB 16 way set associative virtually indexed physically tagged cache. So, I have a VIP T cache, 1 MB is the size of the cache and it is 16 way set associative and the cache block size is 64 bytes ok.

So, the question is; what is the minimum number of page colors that will be needed to guarantee that no 2 synonyms map to different sets in the processor cache of this computer. So, how do I determine this? Firstly, what is the minimum number of page colors required the minimum number of colored bits is given by the set index bits plus block offset bits minus page offset bits. So, set index bits is what I the number of sets in the cache. So, the number of sets the number of bits required to index a set in the cache is will be the number of hash set index bits or the number of set index bits that will be required for the cache is the number of bits that will be required to identify an individual set in the cache.

Block offset bits are what the number of blocks in each set and then these together when added has to be subtracted from the page offset bits because page offset bits tell me the this part which is not varying the page offset bits is constant. This is same for the virtual address and the physical address. So, that does not vary, the remaining bits vary and those bits must be used to obtain the number of colors.

So, this ensures this num this minimum number of colors these colors ensure that no synonym maps to different sets in cache. So, no synonyms map to different sets in cache ok. So, what was a synonym the synonym the synonym meant that I have 2 virtual addresses pointing to the same physical address and because the virtual addresses are different and this VIP T spurs partially virtually addressed cache. So, therefore, potentially this can happen that these 2 virtual addresses will map to different locations in the cache. And therefore, the same physical page the same physical data same data in physical memory can reside in 2 different locations in the cache, because of this mapping because of these bits because of these bits present in the address.

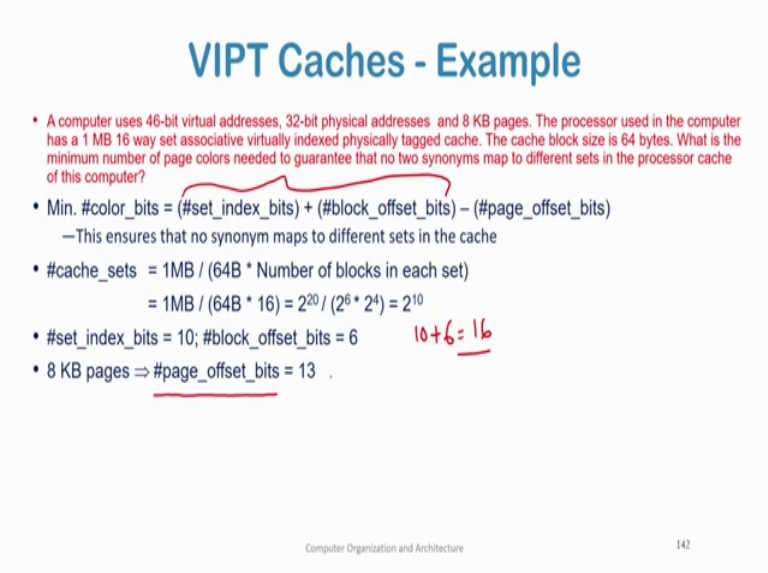
Now, by coloring I ensure that a physical page of one color will only go to a particular set in cache. So, if my page physical page frame is red I know that it has to be if the virtual address will always map it is in such a way that it gets to that that set always gets the same page color. So, then what happens the number of cache sets, how do I determine the number of cache sets? I have the total cache size is one MB and I have 64 I have a 64 byte.

So, a cache block is 64 bytes; so, 64 bytes in a block. So, block offset is 64 bytes 64 bytes and the number of blocks in each set the number of blocks in each set is 16, I have

a 16 way set associative cache. So, 16 way set associative cache. So, the number of blocks in each set is 16 and therefore, one MB is 2 to the power 20; 64 is 2 to the power 6 and 2 to the power 4. This is the number of blocks in each set. And therefore, the number of sets in the cache is 2 to the power 10.

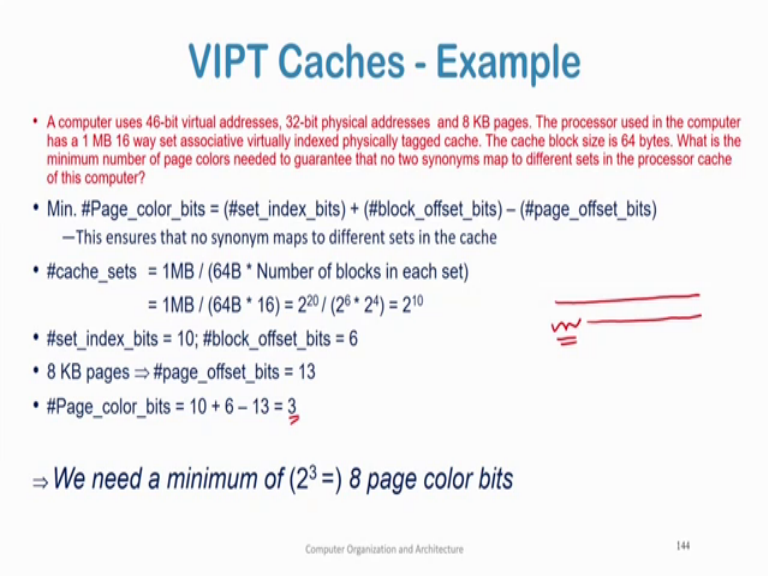
Therefore, I require 10 bits I require 10 bits for set index bits. So, set index bits is 10 set index bits is 10 and block offset bits as we saw here block offset number of blocks in the in a in a set number of number of bytes in a block is the block offset, sorry, sorry, the number of bytes in a block is the block offset. So, the block offset bit is 6.

(Refer Slide Time: 37:54)



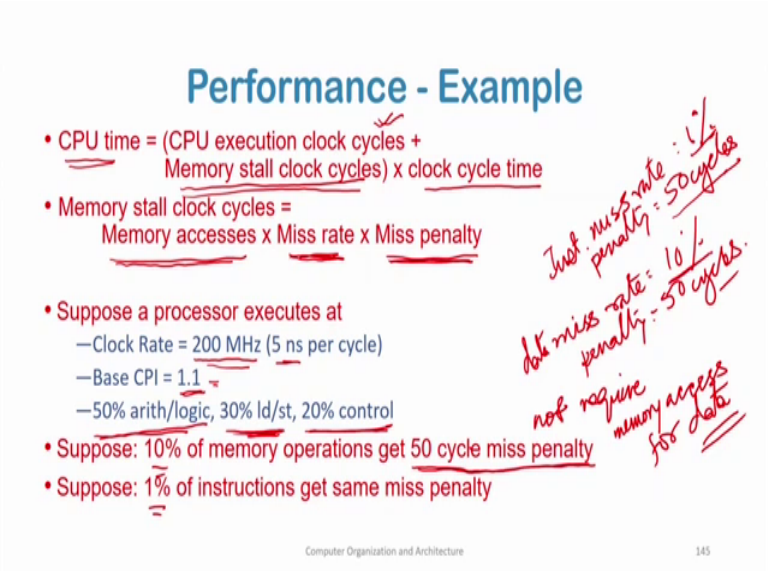
So, now then what happens is that um. So, now, so, this plus this plus this is 10 plus 6 equals to 16. Now I have 8 KB pages. So, number of page offset bits is 13.

(Refer Slide Time: 38:12)



And so, number of page colored bits that will be required is 10 plus 6 minus 13 is 3. Hence, we need a minimum of 2 to the power 3 because I have 3 bits. So, the page offset is 13 bits, but I am using 16 bits to access the index the cache. And therefore, these 3 bits bring in the problem of synonyms. And therefore, I colored my page frames of my physical memory into 8 different colors such that when I map a page of the same color, I will map the virtual address to physical address will be mapped in such a way that a set in cache always gets pages of the same color.

(Refer Slide Time: 39:02)



So, now before proceeding further we will we will now look at page replacement algorithms, but just before that we will take a look back will recapitulate with 2 small examples into the performance factor of paging and caching and together how CPU instruction time is determined will just take a 2 small examples.

So, CPU time taken by taken per instruction; let us say CPU time is given by the CPU execution clock cycles. So, the number of times the CPU is doing work and the number of time slots in which the CPU is doing work number of cycles in which the CPU is doing work plus the memory stall clock cycles in to clock cycle time. So, number of clock cycles in which the CPU is working plus the number of memory stall cycles in which it is waiting for data or instructions. So, the CPU time for a program will be given by the number of clock cycles in which the program code is getting executed and the along with the number of clock cycles for which the code is a program is waiting for instructions or data multiplied by the clock cycle time.

And now the memory stall cycles; again the memory stall cycles again will be given by memory accesses into miss rate into miss penalty; what will be the memory stall cycles number of memory accesses for instructions or data among these accesses; what is the rate at which I miss the miss rate; what is the amount of which let us say in the cache how many are missed in the memory how many I miss. So, that I have to go to the lower level of the memory at a particular level of memory what is the miss rate and if I have a miss what is the amount of penalty that I have to incur and with this we will take an example suppose a processor executes at a crock clock rate of 200 megahertz. So, my cycle time is 5 nanoseconds, fine.

And I have a base CPI of 1.1. So, if everything goes well and if everything goes well, I have a hit in the in the memory and I am executing everything fine my pipeline is working I have no stalls then I have a CPI cycles per instruction is 1.1 and in this computer in the in this program that we are considering 50 percent are arithmetic logic instructions in this program, 30 percent are load store; that means, data access and 20 percent are control instruction.

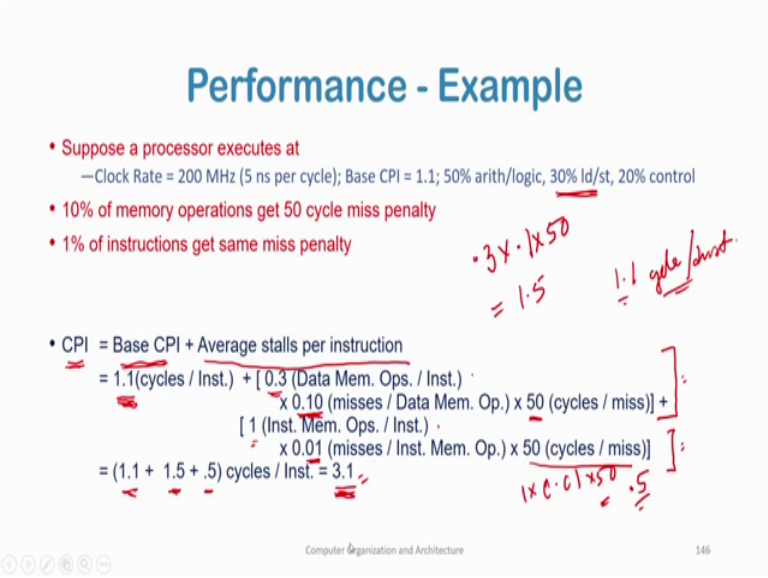
So, therefore, 70 percent instructions do not require do not require memory do not require memory access memory access for data do not require memory access for data only for instructions all instructions everybody required to access memory for

instructions off which some may be missed. So, I do not have the instruction at a certain level of memory, I could have to go to the next level of memory to get the instruction and for the 30 percent load store instructions, I have to access memory for data for all instructions, I have to access memory for the instruction itself and for 30 percent of the instructions, I have to access memory for data.

Now, let us say suppose I have a 10 percent memory operations, let us say 10 percent memory operations, let us say 10 percent memory operations get 50 cycle miss penalty. So, 10 percent of memory operations get see 50 cycles of miss penalty. So, the miss rate is 10 percent miss rate is 10 percent and miss penalty miss penalty equals to 50 cycles ok.

Next one percent of instructions get the same miss penalty. So, this is the data miss rate and another is the instruction miss rate and that is equals to one percent and miss penalty is same penalty is same 50 cycles. So, both for instructions and data the penalty is 50 cycles, but the instruction miss rate is 1 percent the data miss rate at a given level of memory is 10 percent.

(Refer Slide Time: 44:03)



So, therefore, how do I get the average CPI average cycles required for instruction is the base CPI base number of cycles required for instruction plus average number of stalls per instruction how do I get that?

So, we knew that the base cycle base CPI is 1.1. So, if there are hits at the inns memory, if there is a hit in the memory then if there is always a hit in the memory then the I require 1.1 cycle per instruction ok. So, this is what it said.

In addition I have stalls. So, what is the average stalls per instruction? That is we said that 30 percent or 0.3; 30 percent are data memory operations ok. So, 30 percent are load store; that means, data access are 30 percent out of this 30 percent, there is the miss rate of 10 percent and each of them will incur. So, this out of this 30 percent I have a miss rate of 10 percent. So, 30 percent I will load store, but out of this 30 percent; that means, 90 percent of the times, I get my data that I require at a certain level of memory let us say main memory we are considering now.

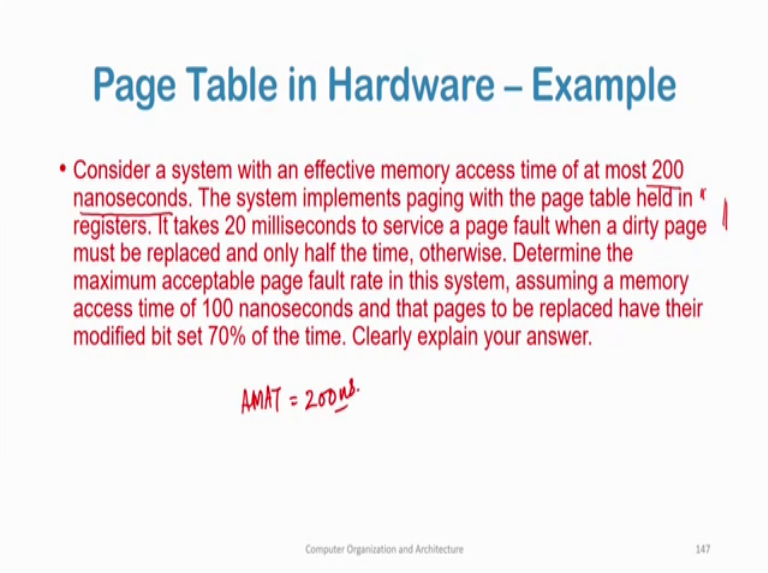
And or cache memory let us say we are accessing now and otherwise I have to go in 10 percent of the times; however, when I am accessing data memory, I have to go to the next level of memory that is main memory and when I go to the main memory I incur 50 cycles ok. So, the overhead the effective overhead for going into of going accessing memory for data is this. So, 30 percent data instructions out of which 10 percent misses and each of that incurs 50 cycles for instructions all instructions. So, 100 percent all instructions must access the memory out of that 1 percent misses the memory and then when I have a miss in the memory I incur 50 cycles I incur 50 cycles.

So, what is the total CPI; what is the total CPI it is the addition over the base CPI plus the memory access overhead due to data, and data misses memory access overhead due to data misses plus memory access overhead due to instruction misses. So, if I calculate, how did I get this 1.5? So, I got 0.3 into 0.1 into 50 which is equals to 1.5 similarly for this case I have 1 into 0.01 into 50 which is equals to 0.5 ok. So, I got this in this means and therefore, the CPI is 3.1.

So, we can understand that although the base CPI that that everything is a hit if everything is a hit, then I require only 1.1 cycles per instruction, but due to the overheads caused by the misses in the memory in the cache misses in the cache, and because I have to go to the memory, sometimes 10 percent of the times in case of data and 1 percent of the times in case of instructions the effective cycle time for each instruction the effective amount of time required to execute each instruction becomes 3.1 cycles ok.

So, this was the amount of performance degradation.

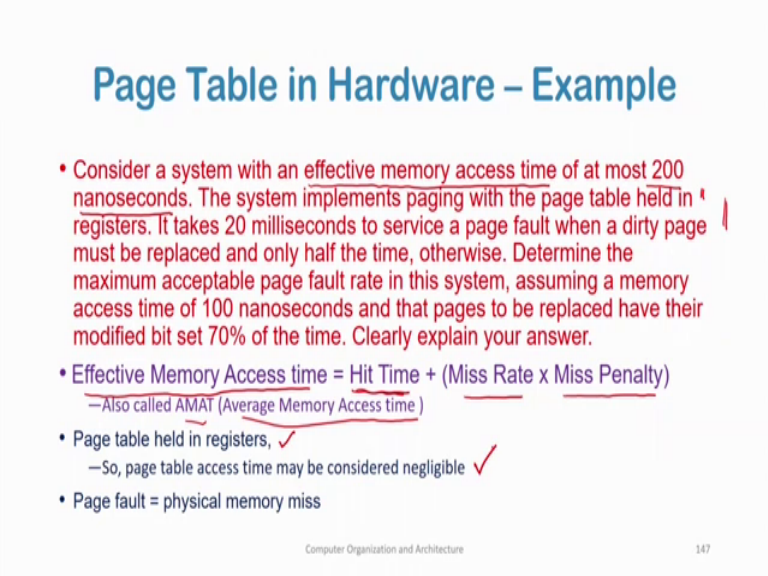
(Refer Slide Time: 48:04)



We will take another example before going into page replacement. So, consider a system with an effective memory access time of at most 200 nanoseconds. So, effective memory access time or average memory access time is equal to 200 nanoseconds the system implements paging with a page table held in registers. So, I have a hardware page table this means that I have a hardware page table and this means that I do not require to I do not have any overhead for accessing the page table, it is negligible the overhead for accessing the page table is negligible, because the page table is held in hardware registers.

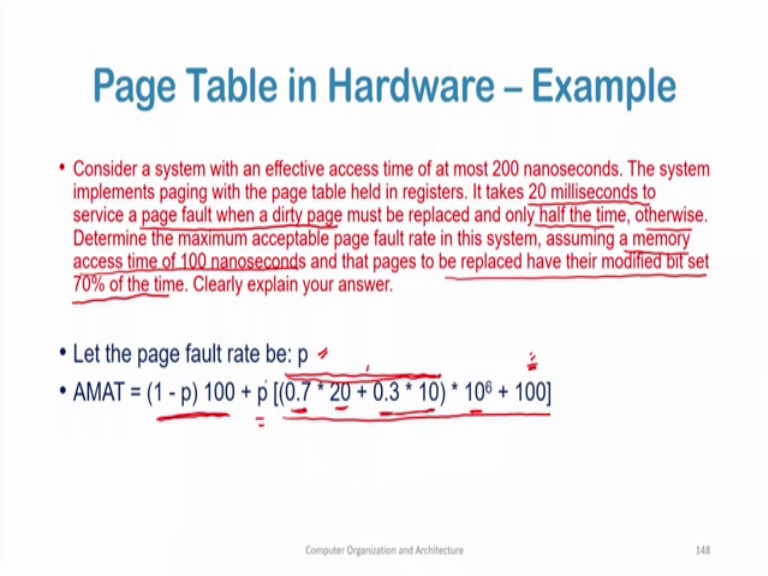
It takes 20 milliseconds to service a page fault when a dirty page must be replaced and only half the time otherwise determine the maximum acceptable page fault rate in the system assuming a memory access time of 100 nanoseconds and that pages to be replaced have their modified bit set 70 percent of the time.

(Refer Slide Time: 49:17)



So, what is effective memory access time? We said here effective memory access time. So, effective memory access time is the hit time. So, memory hit time plus memory miss rate into memory miss penalty this is also called the AMAT or the average memory access time as we just told we said that the page table is held in registers. So, page table access time may be considered to be negligible and what do we mean by a page fault, it is a physical memory miss.

(Refer Slide Time: 49:52)



Now, how do I determine this effective memory access time? Now this effective memory access time will be given as this. So, let the page fault rate be p; that means, then the rate at which I miss the main memory no access I try to access the main memory and then the page is not there in the main memory this happens at the rate of p.

So, 1 minus p is the hit time is the memory hit time and we said that the access time mem that we have a memory access time of 100 nanoseconds. So, for accessing the memory main memory, I require 100 nanoseconds, if I have a hit in the memory, I require 100 nanoseconds plus miss rate as we told miss rate into miss penalty. So, miss rate is p because the page fault rate is p and this is the miss penalty what do we have in this miss penalty part. So, in this miss penalty part I have to do what I have to bring this page from secondary memory into the main memory.

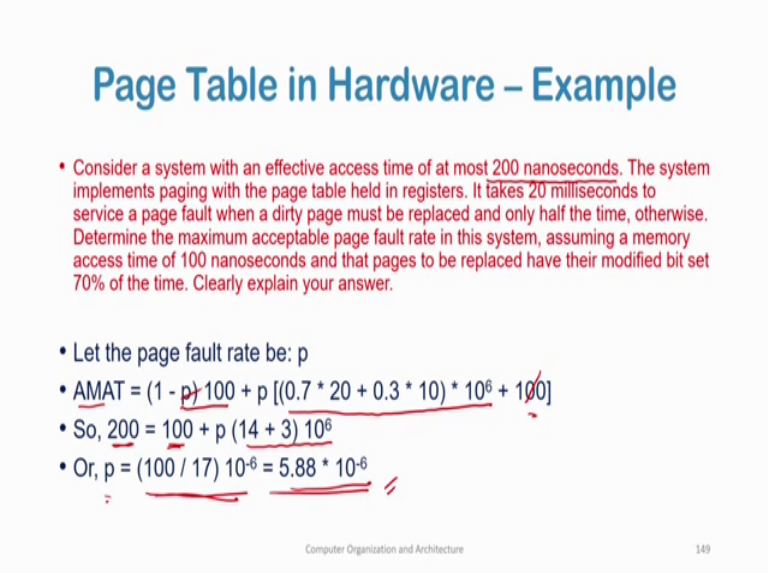
Now, in doing so, we said that it takes 20 milliseconds to service a page fault when the page is dirty and the page can replaced in half the time that is within 10 milliseconds otherwise. So, if the page is not dirty, then what then during page replacement I can just I can just find out the page to be replaced and that if that page is not dirty I will just discard this page I do not have to write the page back to the secondary memory. And then bring my required page frame I do not need to do this, I do not need to bring in my required page only after writing my previous page I do not need to do that why because the page is not dirty and if that is. So, it takes only 10 milliseconds; however, if the page is dirty. That means, I have to write back the page, I find the page which must be replaced the; I write back the page into physical memory and in the page frame of for corresponding to this page I bring the new page in.

That takes 20 milliseconds and we said that that pages to be replaced have then modified bit; that means, the dirty bit set 70 percent of the time when I have a page for 70 percent of the time I require 20 milliseconds and the remaining 30 percent of the time, I only require 10 milliseconds because in the this case the page is not dirty.

So, the effective time that is required to address to require to service the fault is this. So, to bring back whichever page, I need back into memory, I need this and then and then what happens then, I need to again I will have a page hit, I have to access the memory again to get my data ok. So, I have a page fault I will go to the secondary memory bring my page and then access the memory again this time, I will have a page it. So, I will only

incur 100 nanosecond and get my data and here these are in millisecond. So, I need to multiply this with 10 to the power 6 to convert it to nanoseconds.

(Refer Slide Time: 53:27)

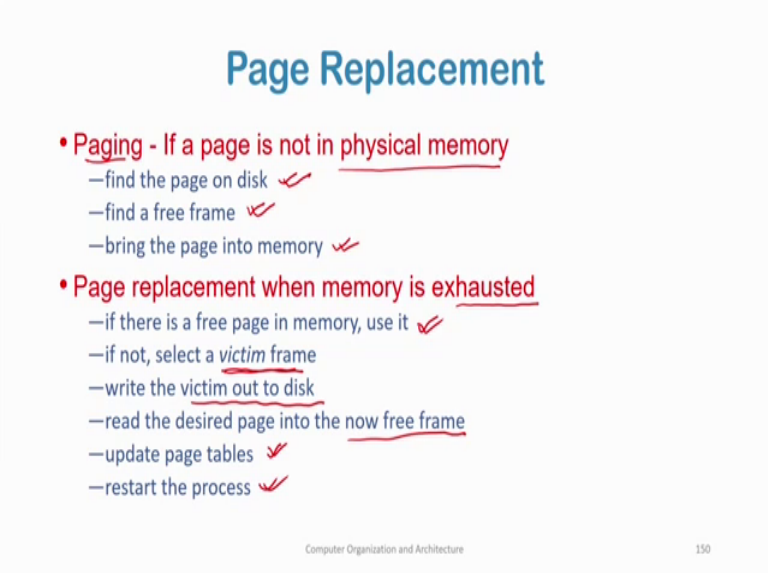


So, therefore, I we have first said that my affective memory access time is 200 nanoseconds. So, AMAT or average memory access time is 200 nanoseconds. So, 200 equals 200 plus p into. So, 7 by 10 into 20 which is which is basically 14. So, 14 plus 3; so, 17 into 10 to the power 6; so, this part gives 17 into 10 to the power 6. So, 17 into 10 to the power 6 plus 100; so, this way if you break up, if you break this up, this becomes 100 minus 100 minus p in to 100 and this one again has a p into 100. So, this one and this one cuts off and therefore, I have what do I have I have 200 equals 200 plus p into 17 into 10 to the power 6. So, therefore, p is given by 100 by 17 into 10 to the power minus 6 or 5.88 into 10 to the power minus 6.

So, therefore, what this means is that if the page fault rate is at most 5.88 into 10; 10 to the power minus 6. So, I have one page fault every 5.88 into 10 to the power minus 6 accesses if the page fault rate is as low as this. So, this means that if the page fault rate is at most 5.88 into 10 to the power minus 6, then I will have a effective memory at most time which is at most 200.

So, the memory access time effective memory access time incurring page faults together if the effective memory access time will be less than or equal to 200 200 nanoseconds if the page fault rate is as low as this.

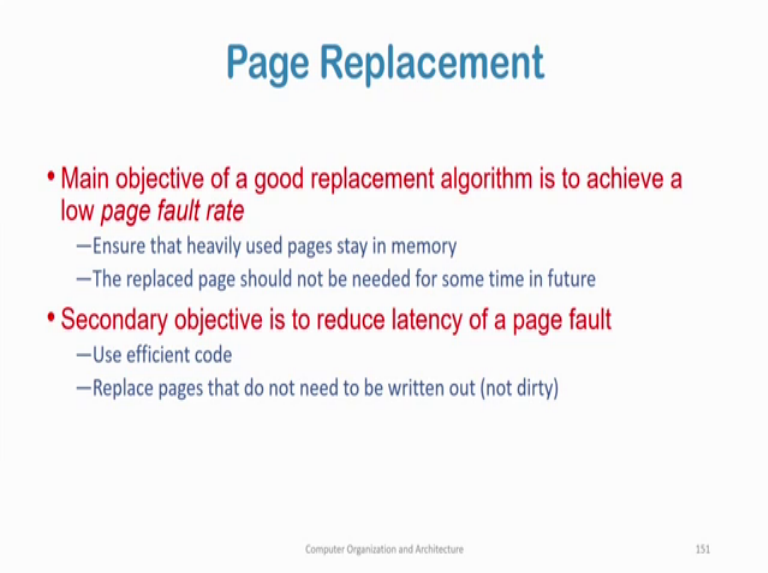
(Refer Slide Time: 55:29)



So, now we move to the next topic of discussion which is place replace page replacement. So, just to brush up, we discussed about the page replacement, but we will now discuss about algorithms just to brush up as to why we require page replacement. So, in paging if a page is not in physical memory, it has to be replaced though if in paging if the page is not in physical memory we have to find the page on the disk find a free frame in the physical memory and then bring the page into memory and put it in that free frame.

So, I need page replacement when memory is exhausted if there is free page in memory then use it. So, during replacement if I have a I do not need any replacement I just need to bring the page into a free page, if a free page in memory exists free page frame exists, there is no need of replacement if not, I have to select a victim frame and herein comes replacement, right the victim out to the disk read the desired page into the now free frame update page tables and restart the process. So, I had a page fault, I service the page fault and restarted the process just as in the previous example, the question is what to which victim to replace and how to replace what is the replacement mechanism and whom to replace.

(Refer Slide Time: 56:55)



So, the main objective of a good replacement algorithm is to achieve a low page fault rate. So, I need to achieve a low page fault rate. So, page fault should be very low as I as we saw that if the page fall rates are not very low the effective memory access times becomes will become very high. So, in that we could only in the previous example; we could only we could only control the page fault rate within 200 nanoseconds because the page fault rate was as low as 5.88 into 10 to the power minus 6.

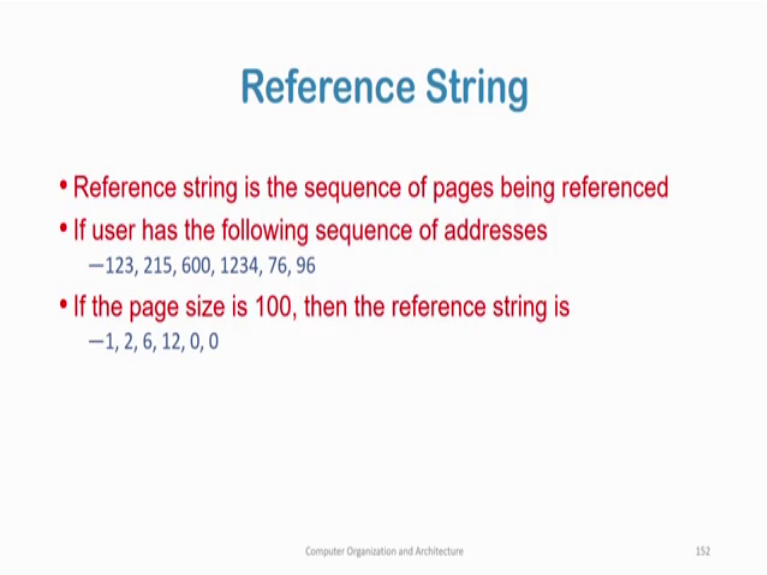
Now, a low page fault rate ensures that heavily used pages stay in memory how can we ensure a low page fault rate by ensuring that heavily used pages stay in memory I do not evict heavily used pages out of main memory and the why is this. So, because of the locality of reference again because if a page is being heavily used it is very probable that this page will be used again in future.

The replaced page should not be needed for some time the replaced page whom I replace it is more difficult to determine this. The replaced page should not be needed for some time in future this is also based on the locality of reference meaning that if there is a page which is being rarely used in the recent past which have rarely been used in the recent past it is likely not to be used again in the recent future as well.

The secondary objective of page replacement is to reduce latency of a page fault. So, how do I reduce latency of a page fault by using efficient code for handling a page fault and by replacing pages that do not need to be written out? So, I will try to write a page

which whose dirty bit is not on if that is. So, then I can just discard this page and the page fault can be quickly handled as we saw in the previous example although in there, it was not to exact numbers, but when it was it was not dirty when the when the page was not dirty I could I only had an overhead of 10 milliseconds; however, if the page was dirty then the replacement to and overhead of 20 milliseconds.

(Refer Slide Time: 59:28)

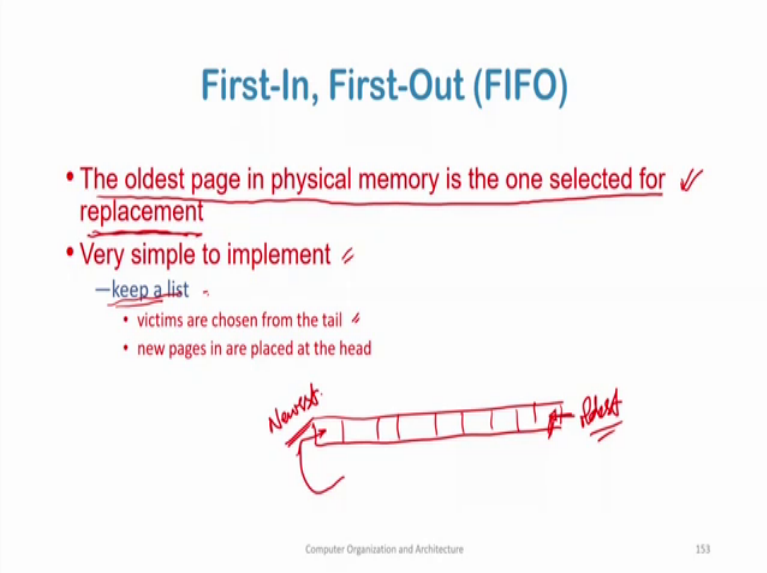


So, before going into specific algorithms, we will we will understand, what is called a reference string a reference string is a sequence of pages that are being referenced. So, for to understand the algorithms we will need to understand the algorithms based on a sequence of page accesses. And this sequence of page accesses of by a certain program say will be will be called its reference string.

So, for example, if user has the following sequence of if a user has the following sequence of memory accesses. So, he accesses memory 123, 215, 600, 1234, 76, 96; these are the memory references that he has done and if the page size is 100, then the reference string is 126, 1200; that means, for 123; that means, if the page size is 100. So, what is the page number of this page address it is modular 100. So, basically it is it is not modular 100, sorry, it is the it is page number 1 because 0 200 is in page number 0, 100, 101 to 200 will be in page number 1. Similarly 201 to 300 will be in page number 2. So, therefore, 215 will be in page number 2, 600 will be in page number 6.

So, so, basically I did a small mistake nothing 0 to 99 will be in page number 100 to 199 will be in page number sorry 0 to 99 will be in page number 0, 100 to 199 will be in page number 1, 200 to 299 will be in page number 2 likewise. So, that if we calculate in that way 600 comes in to page number 6. So, 1234 which means; it will come in to page number 12, 76 will come in to page number 0 and 96 will come in to page number 0.

(Refer Slide Time: 61:28)

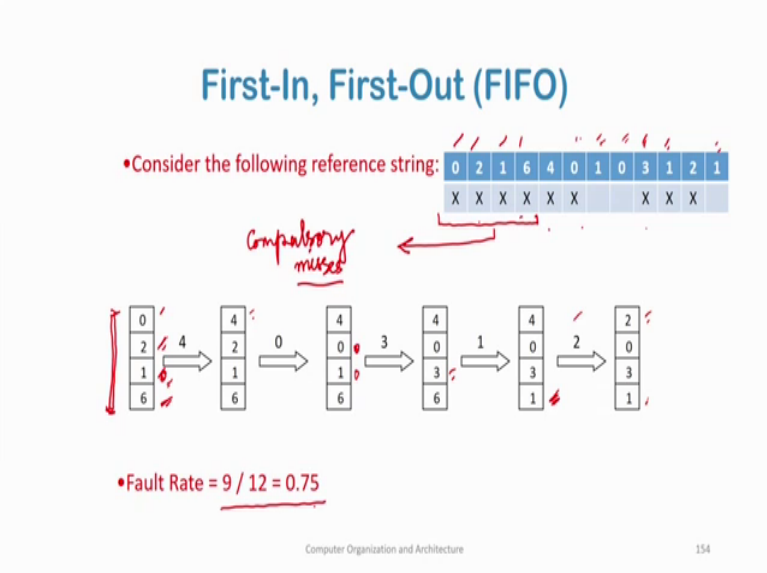


So, now the first page replacement algorithm this is called the first in first out replacement algorithm this is the oldest page this what does this scheme say the oldest page in physical memory is the one selected for replacement. So, the oldest page in physical memory is the one selected for replacement very simple scheme whichever has first come into the physical memory should be the first one to go out of the physical memory as well.

So, I will choose that page for replacement which came which was brought into the memory physical memory the earliest among all physical pages that we have in memory whoever was brought in the earliest the oldest one will be replaced. It is very similar to implement simple to implement what do we do we keep a list of victims and chosen from the tail. So, we keep a list we keep a list and this list is the references and this one is the first one that that had come this is the newest one. So, because this is the oldest I will choose this one to be replaced.

So, if this is the oldest this is the newest then from the tail of the list I will choose the oldest and replace it, new pages are placed at the head new pages are placed at the head.

(Refer Slide Time: 63:00)



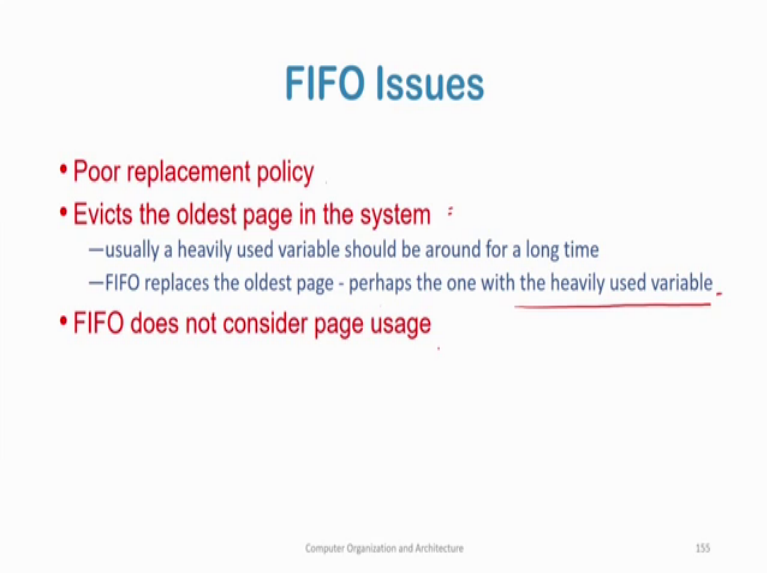
So, consider the reference string 2 0 1 6 4 2 0 1 6 4 0 1 0 3 1 2 1. Now here I have 4 page frames, let us say my physical memory only has 4 page frames available and these are the set of memory accesses. So, initially there is nothing in the physical memory. So, the first 4 misses are compulsory the first 4 misses are compulsory misses and. So, I bring in 0 2 1 6 0 2 1 6 missed and then what happens 4 comes in whom will 4 replaced 0 was the first one to 0 was the first one to come into memory. So, the 0 is the oldest. So, 0 is replaced and with 4.

Now, 0 is accessed again, now 0 is not there in the physical memory. So, 0 will again lead to another page fault and. So, whom will it replace 2 is now the oldest one who which came into memory. So, 0 replaces 2 then what happens one comes again one is referenced again this one does not result in a miss this one is already there in main memory. And this is not a miss, then 0 this also does not incur a miss 0 is already there in memory, it does not incur a miss then 3 is accessed when 3 is access 3 is not there in physical memory whom will it replace?

It will replace the oldest one because 0 2 1 6 was the order 0 and 2 has already been replaced. So, now, the oldest is one now the oldest is to 1. So, 3 replaces one 3 replaces 1.

Now, 1 is accessed again 1 is currently not there in physical memory 3 just replaced 1 accessed again and then a one incurs a miss again whom does 1 replace will see 1 replaces 6? So, 1 replaces 6, now 2 is accessed 2 is accessed 2 is currently not there in physical memory. So, 2 replaces 4, 2 replaces 4 and because 4 is now the oldest and then 1 is now there again in physical memory. So, what is the fault rate? So, I had 12 the different string is of length 12. So, I have 12 mem references out of 12 references nine resulted in a fault 1 2 3 4 5 6 7 8 9; 9 resulted in a fault. So, the fault rate is 9 by 12 or 0.75.

(Refer Slide Time: 66:05)

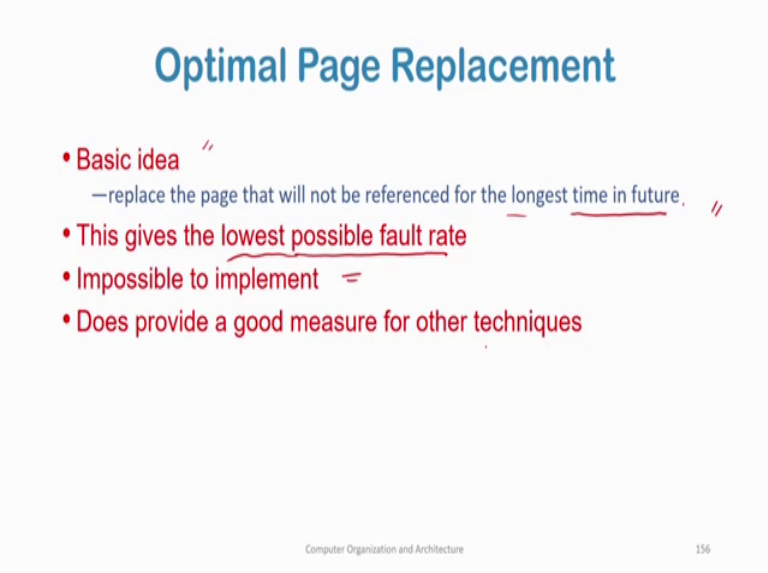


So, FIFO although is very simple, I find out the oldest page in memory and replace it, this is the algorithm, it is a poor replacement policy why it evicts the oldest page in the system and it does not take care whether this page is being heavily used currently or not, it does sees who has been brought at the earliest and it will replace that it does not care as to which if this page is being frequently used.

So, usually of a heavily used variable in a page should be around for a long time. So, we should try to keep a heavily used page for a long time, but FIFO is unaware as to how heavily your page is being used and may easily evicted. So, FIFO replaces the oldest page perhaps the one with the heavily used variable.

So, FIFO basically does not consider page usage.

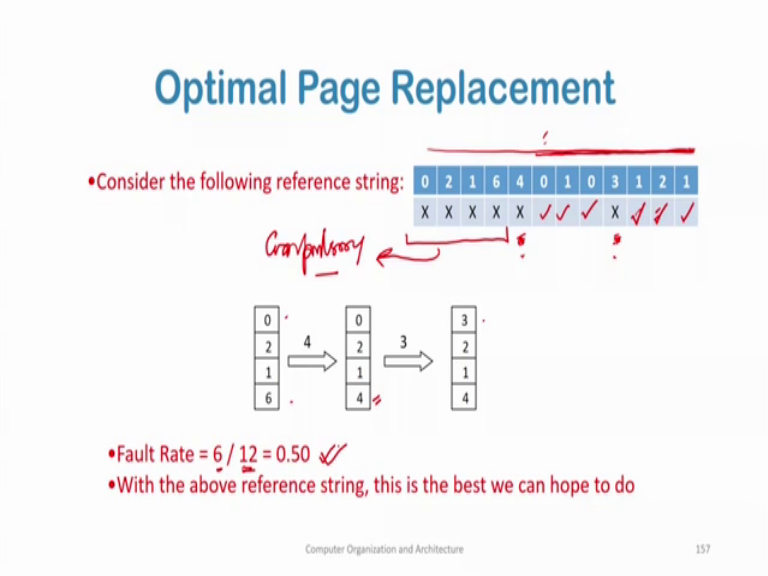
(Refer Slide Time: 67:04)



Before proceeding to the next actual algorithm we will we will go into an hypothetical actual algorithm which actually can cannot exist in practice, but is used as a measure of comparison for all other algorithms.

So, this one is called the optimal page replacement algorithm what is the basic idea of the algorithm, I replace the page that will not be reference for the longest time in future this is where is the why it is impractical, I want to replace that page which will not be reference for the longest time in future. Now I cannot see the future and therefore, this algorithm cannot be implemented in practice because I am saying the future this algorithm will give the lowest possible fault rate page fault rate, this will give the lowest possible page fault rate. However, it is impossible to implement, it does provide a good measure for other techniques.

(Refer Slide Time: 68:00)



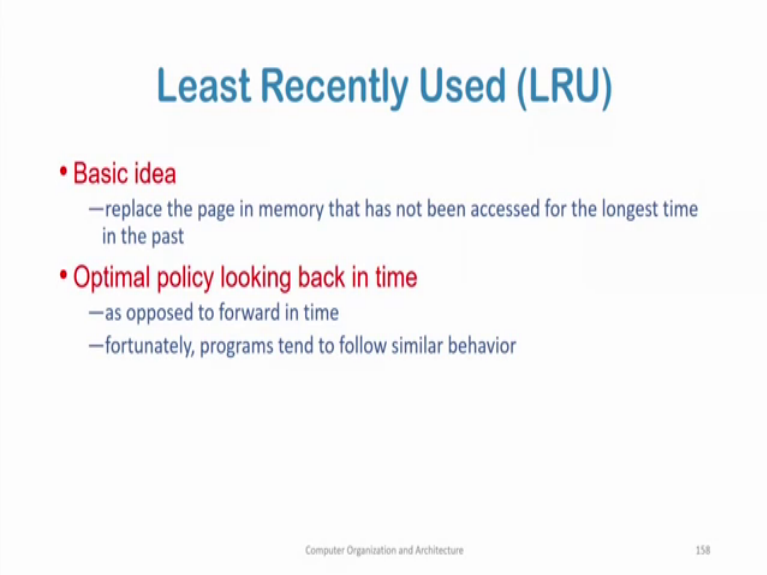
So, if we take the same set of 12 reference strings what happens is that this first 4 will still incur a miss because these are compulsory this is compulsory this is compulsory. Now the 4 is 4 is not there in memory. So, it will find out whom to replace it will find out that one which will not be used for the in for the longest time in future now out of this 12 reference 6 is never used in future.

So, therefore, this 4; this 4 will replace 6 here previously it replaced 0 in the in the FIFO algorithm 4 replace 0, but here 4 replaces 6 because it can see the future and it sees that 6 you will not be used for the longest time in future. So, it replace 4 replaces 6 then 0 is there in memory. So, it is a hit one is a hit 0 is again a hit because these are there in the page frames in physical memory 3 is again a miss whom 3 will replace 0 because out of this 12, it is never being used in the future. So, 3 replaces 0 and this one is a miss again 1 2 and 1. So, 1 2 and one are again hits.

So, in this algorithm we see that we have an addition to the 4 compulsory misses we only have 2 more um. So, we only have 2 more page faults I possibly have been talking of page faults as cache misses please bear with me then that was a mistake. So, these are all page faults not cache misses. So, I in addition to these 4 page faults I have I this one is a page fault and this one is a page fault. So, I have 6 page faults. So, the page fault rate is 6 out of 12 which is 0.050.

So, so, previously in FIFO it was 0 point in FIFO the page fault it was 0.75 and then optimal page replacement give me a fault rate of 0.5. So, with the above reference string this is the best we can hope to do.

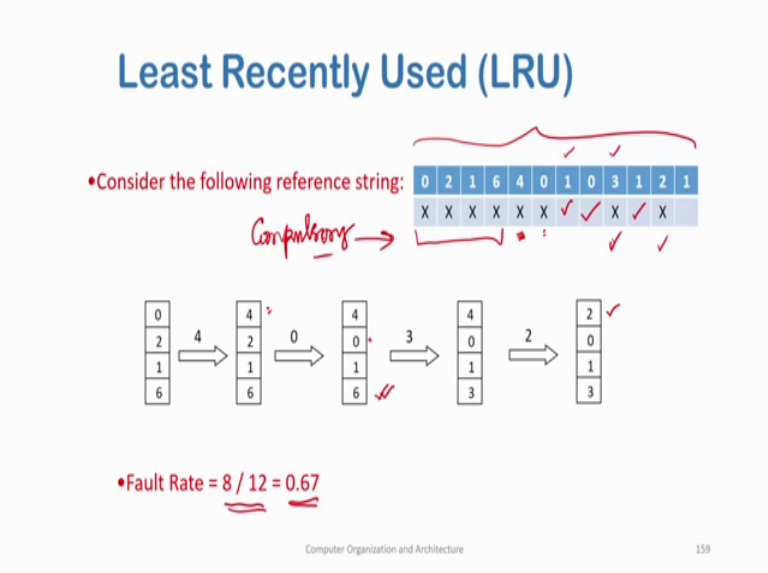
(Refer Slide Time: 70:29)



Then we come to the next algorithm as it is so, it is a very popular algorithm; however, due to the problems with this implementation various approximations of the algorithm is used. So, we first in a learn; what the algorithm is the basic idea is that you replace that page in memory that has not been accessed for the longest time in the past. So, this is practical because you can see the past you cannot see the future.

So, although it may it is algorithm is costly, it is possible to look into the past and therefore, this can give it is an optimal policy looking back in time. So, if I do not have the option of looking forward in time which I cannot have this is the best possible alternative that I have as opposed to forward in time, it looks back in time and finds fortunately programs tend to follow a similar behavior. So, looking back in time is almost always very beneficial, because it allows you to utilize locality of reference.

(Refer Slide Time: 71:40)

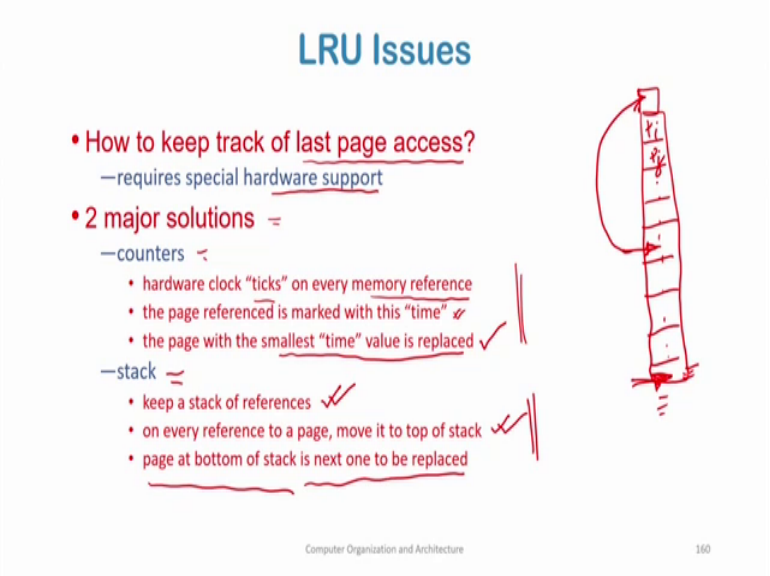


Now, we take the same reference string again same set of 12 references on this the first 4 are again compulsory, these are compulsory, these are compulsory and then the fourth one replaces whom the fourth one replaces. So, all become because it will replace 0 the 4 will replace 0 because all have been used once and the least recently used is 0 because 0 2 1 6 the least recently used is 0.

So, 0 is replaced and. So, by 4 and then 0 is accessed again. So, this will result in a miss. So, who will replace; obviously, 2 will replace because that is the least recently used. So, after this one here 2 replaces sorry 0 replaces 2 and then the subsequent one is a hit this one is a hit, again 0 is a hit and then 3 3 is replaced by whom 3 is replaced by the least recently used. So, 3 is replaced by 6 because 6 is least recently used among all the page frames that we have currently in memory 6 is the least recently used and therefore, 3 replaces 6.

And then I have a 1 1 is a hit and then I have 2 2 is a miss and who does 2 replace 2 replaces 4 because 4 is the least recently used. So, using a least recently used scheme the page fault rate is 8 out of 12 which is 0.67. Therefore, it is not as good as optimal page replacement algorithm because we cannot see the future, but it is not as bad as FIFO it is the best that we can do looking into the past.

(Refer Slide Time: 73:48)



Now, as we told LRU is difficult to implement in practice how do we keep track of the last page access, this is the problem require special hardware support. So, I will 2 I need to find out corresponding to each page; how recently, it was used in the past and for that we need to have we need to have special hardware support. So, there are 2 major solutions to this problem one is a counter based solution.

So, it uses hardware clock ticks on every memory reference on every memory reference I have a hardware clock tick. So, reference one tick is one reference two; so, out of these 12 reference; tick 1, tick, 2 up to tick 12. So, 12th reference is a tick number 12. So, the page referenced is marked with time. So, each page when I have accessed because this stick is progressing globally over all memory references when a particular page is accessed I will mark the time at which it was accessed.

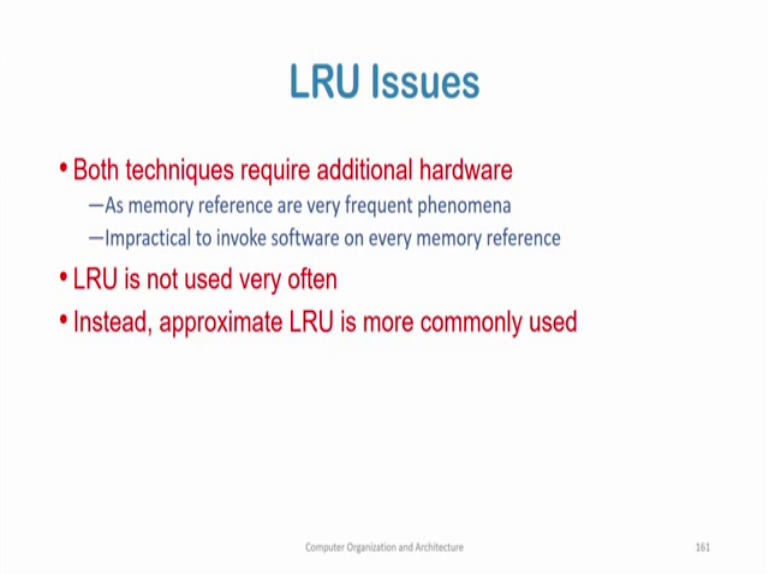
So, in the future, I will be able to know when in the past was this page accessed because I am marking this page with the tick number when it is accessed. So, in future I will be able to access the stick number of this page to understand how back in the future was this page referenced. So, the page with the smallest time value is replaced the other way to implement this LRU is with the use of a stack we keep a stack of references on every reference to a page we move we move it to the top of the stack. So, we keep a stack of references the. So, these are the memory references. So, pages p 1, p 2, p 3 and p 1 p j pi pj something these are the stack of references and in every reference. So, if I reference

this page in. So, this page is referenced then I take this page and move it to the top of the stack ok.

So, this will require possibly a link repre in implementation of the stack to handle it efficiently. So, on every whatever it is it is costly, but we will see these are the solutions then these solutions also these exact solutions for LRU also do not suffice. So, people have devised different approximation algorithms, but we need to see the solutions first. So, we saw the counter based solution the other one is the stack based solution, we keep a stack of references on every reference to a page we move this page to the top of the stack the page at the bottom.

Therefore, because at every reference, if the page is referenced, I am moving to the top of the stack which one is the page at the bottom the page at the bottom is the least recently used page the page at the bottom of the stack is the next to be replaced, because this is the least recently used page and this page will be selected for replacement.

(Refer Slide Time: 77:06)

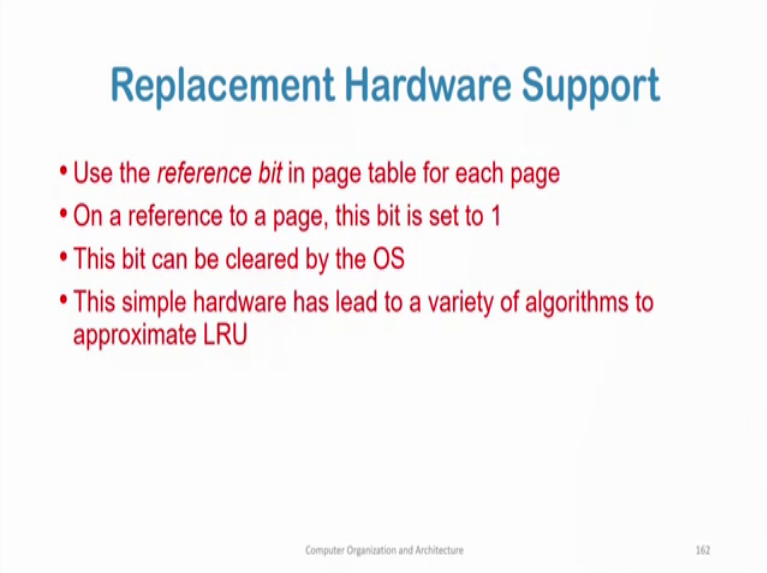


Now, both these techniques will require additional hardware support both the counter base technique which keeps count which for each page I need to keep the value of the tick when it was referenced. So, this one as well as the stack implementation will require additional hardware support a memory reference is very is a very frequent phenomenon. So, at every memory reference, I cannot go back to the OS I cannot interrupt the OS and update the value of the tick for a counter based implementation or I cannot update the

stack when the stack is implemented as a software when the stack is implemented in software why because this will be very costly because memory access is very common, and because it is very common, it is very frequent. So, it is impractical to invoke the software on every memory reference.

So, what software meaning here is the OS I have to invoke the OS to find out who referenced this to basically update the tick value corresponding to a page. So, I cannot do. So, in software because memory references are common and software will be costly the overhead will be very high. So, this is why LRU is not often used and instead, we approximate LRU we use an approximate LRU.

(Refer Slide Time: 78:37)



So, the first way to the way to approximate LRU is the use of the reference bit which we have which we have already studied earlier that reference bit tells me that within a given epoch of time whether I have referenced a page or not so corresponding to. So, I have in the page table in the page table corresponding to each page I have a reference bit for each page, right.

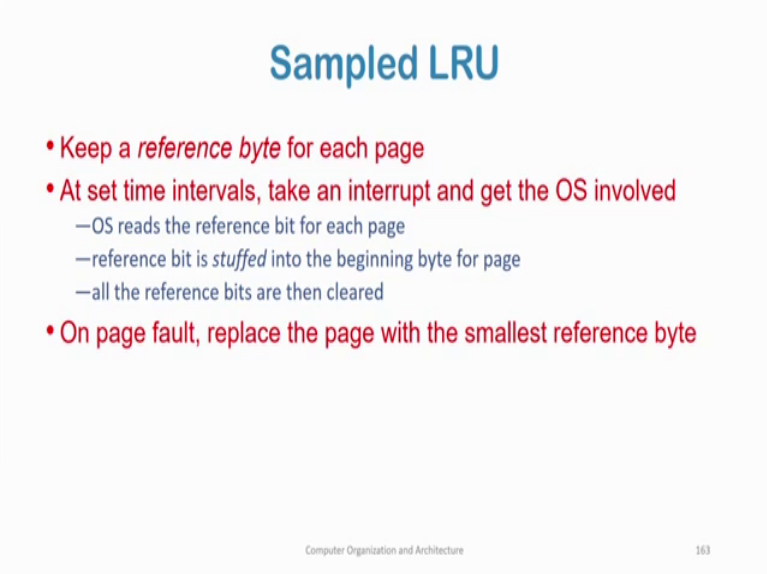
So, that reference bit tells me whether in the last epic of time this page was replaced and at the beginning of each epoch a frame a stipulated interval of time at the beginning of the interval, what I do is I basically 0 down all the differences and again find out, whether, it has it was referenced in the current epoch, I use a reference between the page table corresponding to each page and you know; what do I do this reference bit is set to

1, if this page is referenced in each epoch and at the end of the epoch I 0 down all the reference bits I go to the OS I 0 down all the reference bits and at a given time all the I try to find the page whose reference between 0.

So, in the absence of the reference bit I take the FIFO order. So, in the FIFO order I try to find that page. So, I find out what I find out pages whose reference bit is 0 if the reference bit 0, it means that it was not referenced in the current epoch of time. So, it is not being heavily used it is not being heavily used I am approximating it is not exact, but I am approximating that it is not being heavily used because in the current epoch of time it was not referenced. And I replace that page ok.

This bit can then be cleared by the OS this simple hardware has led to a variety of algorithms. So, the first technique is that of approximate LRU is that I have a reference bit and that reference bit is there for each page in the page table and whenever a page is accessed within a given epoch of time, I set the reference bit at the end of the epoch I 0 down all the reference bit corresponding to all pages. And then at any given time when I need a page to be replaced, I try to find out a page whose reference bit is 0 this will mean that in the current epoch it was not reference and possibly this is not this pages not being heavily used and therefore, it is a good candidate to be replaced.

(Refer Slide Time: 81:24)

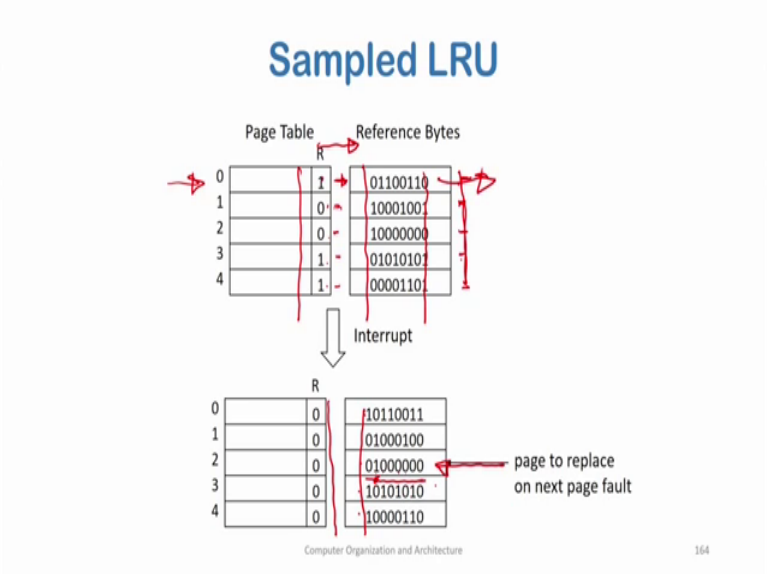


Then we come to sampled LRU. So, using the reference bit we generate a reference byte for each page in this in this technique. So, how do we get this reference byte at set time

intervals which is this epoch, I was talking about I take and interrupt and get the OS involved what do I do the OS reads the reference bit of each page and the reference bit is stuffed into the beginning of the byte for page. So, at the beginning byte of the page I stuff this I stuff this bit all reference bits are then cleared.

So, on a page fault I replace the page with the smallest reference byte.

(Refer Slide Time: 82:12)



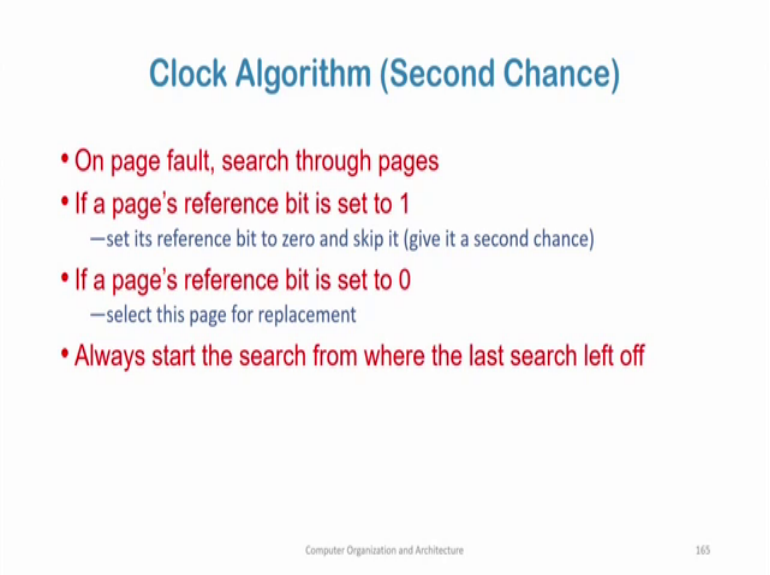
So, how do I will take an example? So, this is let us say the first byte of each page this one these ones are the fur this one is page one page one page 2 page 3 page 4 page 5. So, these are the first bytes and this is kept for reference bits. So, at a given point in time, let us say this is the value of the bytes and in this epoch the reference bit values are this; that means, in page 4 page 0, sorry, this is page 0 for page 0 in this particular epoch this page was accessed in this current time interval this page was accessed page 1 was not accessed, page 2 was not accessed, page 3 was accessed and page 4 was accessed.

Then what happens I shift this bit sorry I shift this bit from here to the MSB this one at this place and I discard the I discard these I discard these ones I throw them off and I bring these bits into the MSB. So, after this I 0 down all the references. So, what I have done I have taken the reference bit see that these reference bits these reference bits have now become the MSBs here have now become the MSBs here 1 1 0 1 0 0 1 1 1 0 0 1 1. So, these have now become the reference and these have come into the MSBs.

Now, which one will be replaced the one with the smallest reference byte will then be replaced. So, this one is the one with the smallest reference byte; what does this tell me that this value basically the numerical value of this byte the numerical value of this byte. This is the smallest numerical value of this byte it tells me that it was replaced it was used this page was referenced in the previous memory.

This is the least recently this is the least frequently this is the least frequently being used it was only used among the last 1, 2, 3, 4, 5, 6, 7, 8 epochs; among the last 8 epochs, it was only used once and how recently, it was used in the in the current, but one. That means, the previous frame in the previous to previous epoch, it was used once, but all others have a higher value and therefore, this one is the least recently used it becomes the least recently used and is therefore, replaced.

(Refer Slide Time: 84:48)

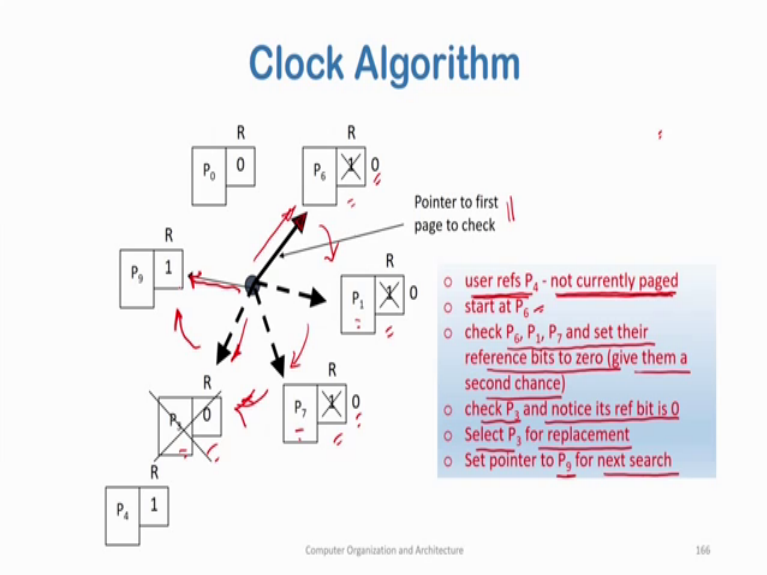


Then we come to the clock replacement algorithm now we come to the clock algorithm or the second chance place page replacement algorithm this is another approximation of the LRU technique. So, here what happens on a page fault I search through pages if the pages reference bit is one if the reference bit is one; that means, it was reference in the current epoch I set its reference bit to 0 and skip it. So, if the if the reference bit is one, I do not replace this page one thing to note here is that in the absence of this reference bit I use I am using a p 4 replacement algorithm.

Now, after seeing this among all these bits suppose I get a 0 I get a 0 reference bit. So, among all these bits which have 0 reference bit, I will use p 4 taking who whoever has among all those pages for which the reference bit is 0, I will replace that page which came at the earliest.

Now, in the second chance page replacement algorithm if the next page that time, I am I am searching that the next page that I am looking for a has a reference bit of 1, I do not replace that page I give it a second chance it, I set its reference bit to 0 and skip it, give it a second chance, if the pages reference bit is set to 0 if the pages reference bit is 0 if it is already 0 then I replace the page. So, I always start the search from where I left off in the second chance page replacement algorithm as like other algorithms I am searching for pages to be replaced.

(Refer Slide Time: 86:47)



Now, let us say; the last time I replace the page frame was at page number six. So, according in to the last slide as we saw my current search will start from page number 6, here my current search will start here. So, let us say the pointer to the first page to check. So, this gives me the pointer to the first page to be checked. So, user references p 4 user references p for page number 4 which is currently not in the in the main memory.

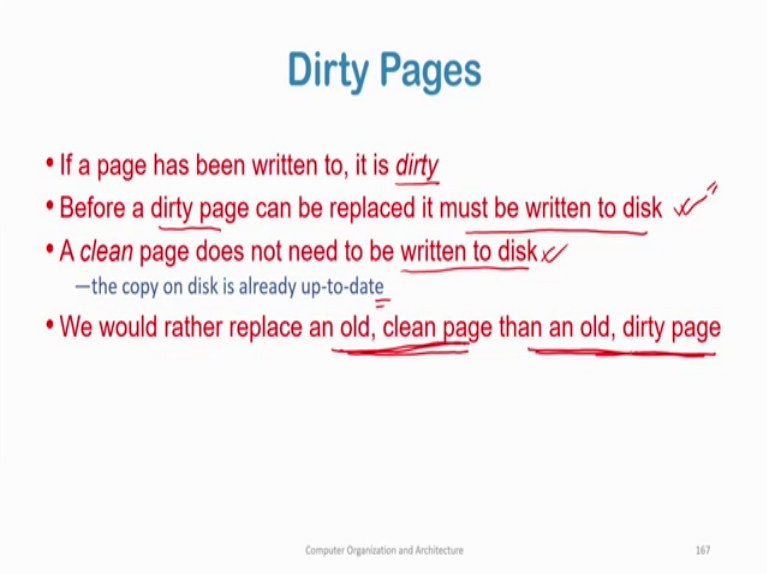
So, I start at page 6 p 6 has the reference bit. So, I set to 0 and go I give it a second chance because if the reference bit was 0 I would immediately use it for replacement because the reference bit is 1 I give it a second chance. So, what do I set this reference

bit 1 to 0 and go to the next one I come here, I get this reference bit 1. So, again I go to p 1, then I go to p 1 p 6, then p 1 and p 7 I find the reference bit to be one for all these 3 I change it to 0 and move on when I come to when I come to p 3, I see that the reference bit is already 0; so I therefore, 3 p 3 for replacement.

Now, let us say if all page frames had the reference bit of one if this would have a reference bit of one then it would go to p 3 p nine p 0 and then come back to p 6. So, when it comes back to p 6 it is surely will get a reference bit of 0 because it itself because the algorithm has set the reference bit to 0. So, only when all reference bits of all pages are one will, it choose um a possibly frequently used page otherwise it will go and choose a page whose reference bit is 0 ok.

So, now so, what happens we start at page 6, we check page 6; page 1, page 7 and set their reference bit to 0 that is give them a second chance and then check page 3 and notice that its reference bit is 0, I select page 3 for replacement and set pointer to p 9 for the next search for the next search, I will start from where I left off in the previous search. So, I replace this one in the in the previous search. So, I go to the next page frame which is p 9 in the current in the next search, I will point to start mine excerpt with p 9 for the next replacement.

(Refer Slide Time: 89:24)

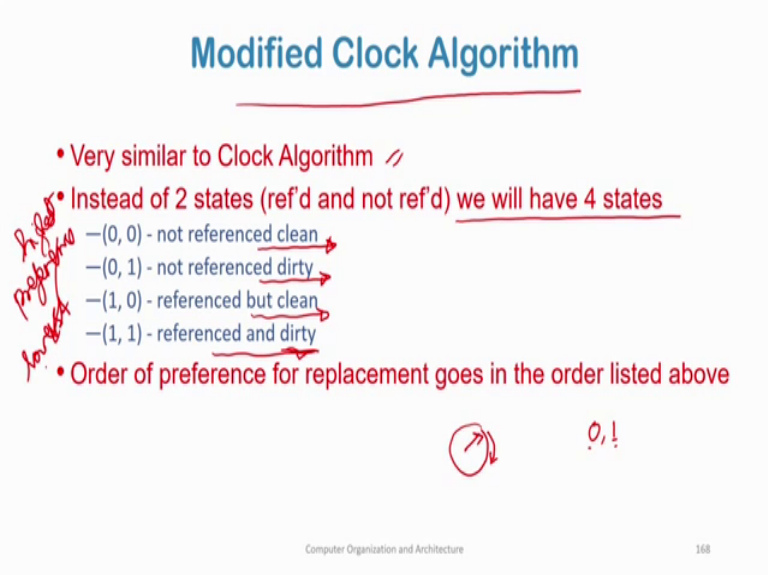


Now, in the last algorithm that we will study we use dirty pages. So, I all I along with the reference bit I also use the dirty bit that that we had studied earlier. So, if a page has been

written to it is dirty before a dirty page can be replaced, it must be written to the disk. So, this is this is what I do not we want. So, reference a page can be reference, but a reference can be a write or a read, if it is only read if the page was referenced only for a read still the replacement is less costly than if the reference bit is 1 and then the page is also dirty. So, in a replacement I have to write this page first back into the disk and then bring in a new page.

So, a clean page does not need to be written back. So, this is the advantage of the dirty bit is not set. So, the copy on disk is already up to date. So, I do not need to write back you would rather replace an old clean page; that means, yes than an old dirty page. So, I want to choose an old clean page than an old dirty page.

(Refer Slide Time: 90:36)



So, this is the basis of the modified clock algorithm. So, the modified second chance algorithm.

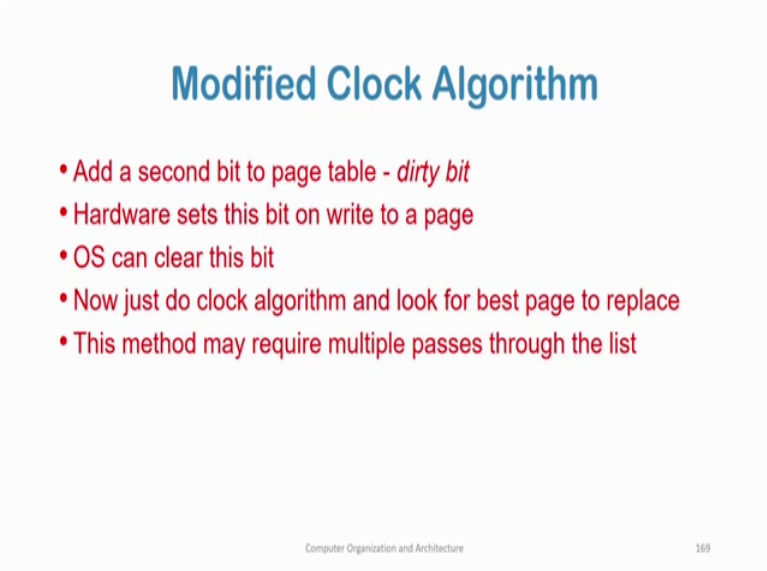
So, here it is very similar to the clock algorithm I similarly use a circular queue and keep the pointer to the last search and then start the search from there. So, you know; however, instead of 2 states. So, previously I already I only had 2 states for each page either the reference bit is 0 or 1, these are the 2 states of each page.

Now, I have 4 states and the 4 state is 0 0; that means, it is the reference bit is 0 as well as the dirty bit is 0 0 1 the ref not reference. That means, it is not reference, it is an old

page, but it is dirty; that means, that even if it is not replaced even if it is not referenced in the recent past if I replace if I choose this page I will have to replace this page if it is one 0 it is referenced, but it is clean if; that means, it is used in the in the recent times in recent times it is being used this page is you being used heavily possibly, but it is being it is being read and it I do not have to write it and if both are one; that means, it is both being heavily used and the page is also dirty.

So, so, the order of preference for replacement goes down the order. So, this one is the highest preference highest preference and this is the lowest preference.

(Refer Slide Time: 92:12)



So, add a second bits; how do I implement it add a second bit to the page table. So, we have the reference bit as well as the dirty bit the hardware sets this bit on write to a page the OS can clear this bit at the end of each epoch let us say. Now, just do clock algorithm and look for the best page to replace it is the same, I want to do the best page to be replaced, but now this method will require main require multiple passes through the list.

So, what will happen I will now first find try to find out none. Firstly, I will go through one round, I will setting the 0 reference bit to 0s, then I if all the ref, I do not find any if I do not find, I will try to find a page whose both bits are clean in the first round, if I do not find suppose for all of whom for which the reference bit is one I will set that to 0, then again in the next page, I will try to find someone with 0 0, if I do not get I will go on

searching. So, I may require multiple passes over this over a particular set of page frames to find the page frames that I want to replace.

With this, we come to the end of this lecture.