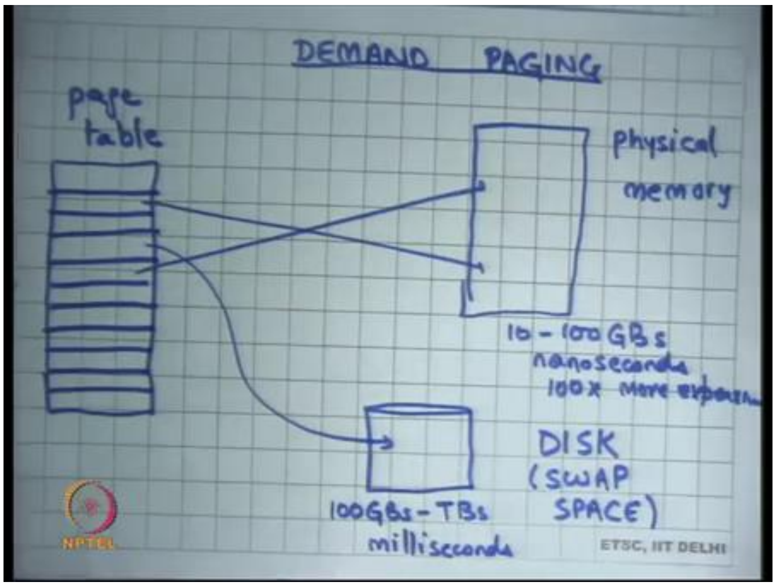
Operating Systems

Prof. Sorav Bansal
Department of Computer Science and Engineering
Indian Institute of Technology, Delhi

Lecture - 29
Demand Paging: Introduction to Page Replacement

Welcome to Operating Systems lecture 29 right.

(Refer Slide Time: 00:29)



So, we were discussing Demand Paging last time and we said that demand paging is a way for the OS to fool the process into believing that it has more memory then it actually is occupying in physical sense right. So, a process may have may see a very large virtual address space and recall that the virtual address space is implemented by a page table assuming you are using paging.

And so, it may see a very large virtual address space, but only some part of it may actually be present in the physical memory, other parts of this virtual address space may be present in the disk alright. And so, if this is the page table and these slots are the pages in some of these pages will have mappings to the physical memory and some of them will not have mappings to the physical memory and logically speaking they will have mappings to the disk alright.

The you know this area of a you also where you store the virtual memory pages in the disk is also call the swap space alright or the swap partition if you may have heard on different operating systems alright. But it is not necessary that these pages point to the swap space, swap space is basically extra space reserved in the disk.

Student: (Refer Time: 01:46).

To allow pages to get saved from physical memory. So, and a so on, but these pages may actually even be mapping into the file system right. For example, if you loaded a dot out file then some pages have not been loaded at load time and so, these pointers will actually will not be point into the swap space, they will be pointing to the file system space in the disk. In either case basically these pages from the hardware standpoint do not exist right.

And so, if a user ever executes an instruction that tries to an access a virtual address somewhere here, it will result in an exception and this exception is called a page fault right. So, page fault will occur and as we know if a an exception occurs control transfers to the operating system. And, the operating systems page fault handler will get to run; without demand paging the page fault handler would have probably just scaled the process.

But, if the operating system is implementing demand paging then the page fault handler should additionally check if the faulting address or the address on which the page fault was generated, if that actually has been points to the disk. So, internally the operating system is maintaining some data structure which basically says you know which pages are mapping to the disk and which pages are not present at all right. So, if this page is actually mapping to the disk then the operating system will load this page from the disk to the physical memory.

And, then create a mapping from the page table to the physical memory and restart the instruction right. Now, restarting the instruction needs to be safe as we had discussed last time and we you know typically; so, one way to do this is basically to rely on the hardware to provide certain guarantee.

So, one guarantee that is provided by the hardware is that if an instruction generates an exception and page fault is one example of an exception then before the exception handler is called the machine state will appear as though the instruction did not execute at all right. So, it is not like the instruction is partially executed. So, the page even though the instruction generated an exception in the middle of with execution the its a responsibility of the hardware to basically rollback all execution state of that instruction before it invokes the exception handler.

So, all the instructions previous to that instruction would have executed and that instruction will not have executed at all and of course, no instruction after that would have executed. So, now the page so that allows the operating system to just restart that instruction. And, how does the operating system restart the instruction? Just by putting the return address or the in the PC, in the tab frame to the instruction that faulted right.

And so, when you when the operating system does irate just goes back to the same instruction and that instruction gets run again. Because, the page table mapping has been created by the operating system this time this particular instruction will not fault right and so, it will continue.

And so, this is demand paging the idea. So, using this demand paging mechanism and operating system basically in some senses using the physical memory as a cache for the disk. So, you can imagine that this is you know one way to think about it is that the entire physical memory pages are actually on the disk, but the physical memory.

So, the entire virtual memory pages are actually on the disk. So, the physical memory is acting as a cache to that space right. So, it just that some pages have been loaded into physical memory and so, you know you basically.

So, most of the so, which ever pages have been loaded to the physical memory execute at full speed, any pages that have not been loaded into the physical memory or the cache have to take a page fault and you basically access the disk in that case. Let us understand some hardware characteristics of physical memory and disk to understand you know what are the tradeoffs involved in this caching setup.

The physical memory is you know one common technology is D RAM meta you today you will get 10 to 100 GBs of 100s of GBs of a DRAM. But, its more expensive its usually 100 times more expensive than disk ok, 100x more expensive roughly speaking.

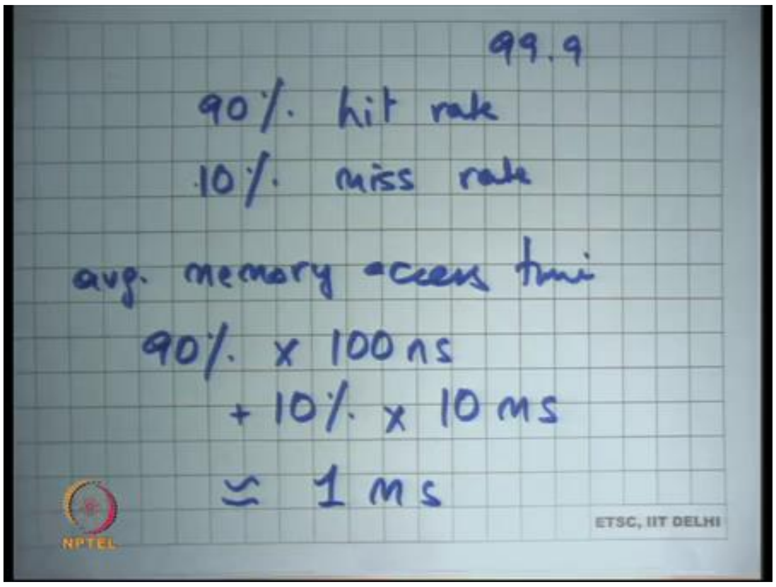
And, disk on the other hand you can get you know 100s of GBs to terabytes and so, but it is much cheaper.

So, you can you know for the same so, byte cost per byte is 100 times more in physical memory then it is in disk. So, you can afford to have much larger disks then you can have physical memory. The more important trade off point is that a disk access takes milliseconds to access while a memory access takes nanoseconds to access.

So, there is a you know 10 to the power 5 or 10 to the power 6 performance difference between these things alright ok. So, you can imagine that if there is such a huge; so, basically a cache hit is only going to take 10 to 100 nanoseconds let us say, but a cache miss is going to take millisecond.

So, you know miss cost is much higher than hit cost and so for this scheme to be successful you would want that the hit rate is very high alright. If the hit rate is small then you would basically be always you know you will be executing the speed of the disk.

(Refer Slide Time: 07:17)



99.9

90% hit rate

10% miss rate

avg. memory access time

$90\% \times 100\,ns$

$+ 10\% \times 10\,ms$

$\simeq 1\,ms$

NPTEL
ETSC, IIT DELHI

So, let us take an example let us say you have a hit rate of 90 percent alright. So, 90 percent hit rate ok; so, let us say I had a 90 percent hit rate and 10 percent miss rate right. So, what is the what is my average memory access time? Average memory access time would be 90 percent into let say 100 nanoseconds plus 10 percent into let say 10 milliseconds right which is roughly equal to actually this cost is almost 0.

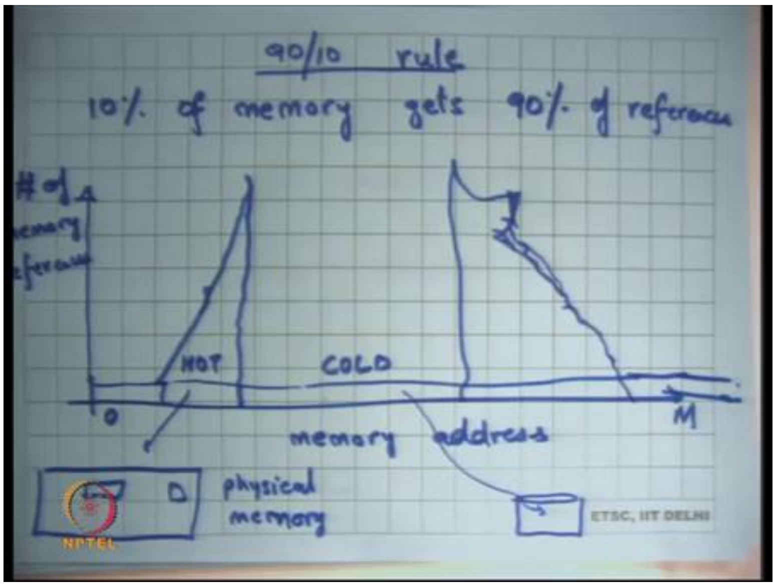
This is actually 10 percent into 10 millisecond that is roughly equal to 1 millisecond right. So, on average every memory access is going to take 1 millisecond of time right that is a very bad; I mean your actual physical memory was actually only 10 to 100 nanoseconds. And, now you know if you have; if you have a 90 percent hit rate you are basically executing at the speed of the disk, completely unacceptable right.

So, these kind of numbers are absolutely not acceptable, what you would; what you would be perhaps with is something like a 99.9 percent hit rate or something right. So, you know less than 0.1 percent miss rate and so, that will basically give you the illusion that yes you have the space of the disk, but the access time of memory alright. So, really the OS is trying to.

Student: Give.

Give the illusion that you have the space of the disk, which is you know 100 GBs to TBs, but the access time of memory right and fortunately in practice it is possible to do that.

(Refer Slide Time: 08:51)



And, it is possible to do that because of this 90 10 rule I mean also call the 90 10 rule which basically says that this is a high amount of locality in accesses by a typical programs. So, programs typically 10 to access the same memory locations over and over again. So, that is called temporal locality, also programs 10 to access locations close to the ones that they have accessed. So, if they access location x then quite likely that it will access either x plus 4 or x minus 4 so, that is called spatial locality right.

So, programs exhibit a lot of spatial and temporal locality and so and so caching the whole the whole basis of caching is that there needs to be some locality right and because and there is a huge amount of skew. So, basically the 90 10 rule says that roughly speaking typical programs 10 percent of memory gets 90 percent of the references.

So, if this 10 percent of memory is brought into the physical memory then you know you already have a very high hit rate. So, 10 percent of memory gets 90 percent of the references. So, if I were to just plot a graph which says you know let say this is the memory address 0 to let say whatever is maximum virtual address you have.

Then you know and you were to plot what memory access addresses are accessed how many times. So, on the y axis you are basically saying how many times was this memory address accessed throughout the life of the program. Then you know you have some small thing here which is saying that these memory access were access sometime once or twice or thrice or 10 times or whatever.

But there is some memory access says which are which have a huge number of accesses right. You can what are these memory addresses likely to be? Let say the code pointer right. So, if you are executing the program in a loop the same EIP or the same program counter is going to get accessed over and over again right. What is the stack pointer?

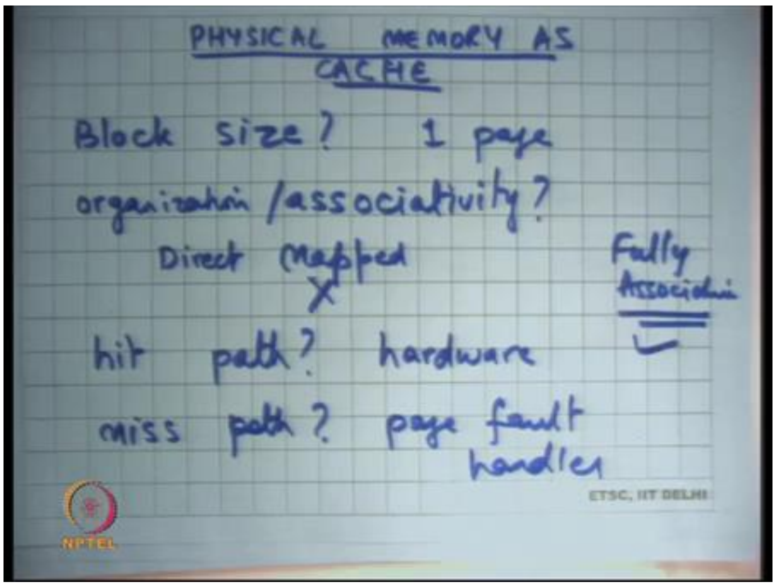
It is likely that the stack point is basically the stack pointer is moving in the same range. So, you basically accessing the same page over and over again right or even data. So, you know even the data structures of the heap, it’s quite likely that you know the root of let say you are maintaining a tree in your you know in your program; let say binary search tree or something.

Then the root of the tree is likely to be very hot you are going to access that root each time you do anything on the tree or something right. So, in general there is a high amount of locality and you know the higher this bar the basically shows you know temporal locality because you know its same location is being access logarithms.

And, the spread of it can be you know called spatial locality because few of accessing this few are also accessing somewhere something close to it. So, it sorts of a sort of seems to have this kind of behavior. So, an operating system would be doing its job correctly with respect to demand paging, if these pages which are very hot are mapped into physical memory and these pages which are cold are mapped into addresses.

So, you can also call these addresses cold addresses and these addresses hot addresses right. And so, the hot addresses should be mapped into physical memory and the cold addresses should be mapped into disk. And, if you can do that then you basically have a system where you have the speed of physical memory and the size of the disk alright ok.

(Refer Slide Time: 12:11)



PHYSICAL MEMORY AS CACHE

Block size? 1 page

organization/associativity?

Direct Mapped ✗

Fully Associativ ✓

hit path? hardware

miss path? page fault handler

ETSC, IIT DELHI

So, we are saying that the physical memory as cache right. So, one way to think about this entire demand paging scenario is to think of the physical memory as a cache, as cache to the disk which is much bigger. So, clearly you know any time we will talk about a cache we basically know that the cache has to be smaller than the entire data set. And so, whenever you talk about a cache you have to worry about you know somethings.

For example: what is the block size? It by block size; that means, what is the unit of data which is cached where is the smallest unit of data that is cached. For an L1 cache for example, it will be your cache line which is you know 60 into 64 bytes depending on your architecture. But, for this what is the block size?

Student: 1 page.

1 page right. So, 1 page alright. How does one choose the right block size? Well, it should not, if the block size is really big then you may be getting too much extraneous data. So, let say in this figure you are accessing address x and your block size was let say 4 megabytes, then you are going to you know you are going to bring in x to x plus 4 megabytes into the memory. And, it is quite possible that you know all that data is not is very cold, it just that x and x some data round x was hot and everything around it was cold.

So, you are basically polluting your cache with cold data. So, if you choose a very large block size you have; you have; you have that problem that you are basically having cold data sharing. On the other hand, if the block size is really small then the firstly, you will heard spatial locality right.

So, let say x and x plus 4 are very likely to get access together, but because your block size is very small you do not get x plus 4 you just get x and so, for x plus 4 you take an another page fault right. So, that is one thing, but more importantly recall that a disk has a very long sort of access time right.

And, also the access time is basically dominated by the seek and the rotational latency right. So, it is better that if you are paid that price you get a large chunk of data from the disk at once rather than just getting 1 byte to get a larger chunk of data from the disk.

And so, you know there has to be some tradeoff between these two constraining forces and the 1 page seems to be a reasonable value and we will discuss more about this a later. The other thing is what is the organization of the cache? Right by organization I mean associativity right.

So, you know in your computer architecture course you must have studied, and your cache can be direct mapped, set associative or fully associative right. So, what you think makes more sense in this case? Should you have a direct mapped cache, or should you have a fully associative cache? . So, let us see you know what is a tradeoff? You know let say direct mapped versus on one side and fully associative on the other side right.

So, in direct mapped I basically know that here is a location for you know these set of addresses. And so, when you get a miss the one that you replace is the you know it is very clear whom you are going to replace. This is the you know there is a conflict between the addresses of the one that is present and the one that is access and the one that is present and gets replaced and the one that is being accessed gets to run right.

So, in other words you know replacement is very fast alright, but the price you pay is that you may get extra misses, conflict misses. Even though you could have adjusted both pages in your memory because, you are doing direct mapped you are getting more conflict misses right. On the other hand, fully associative has full freedom in choosing the page that needs to get erected right.

So, in this scenario where the miss cost is really high and maximizing your hit rate is the for prime importance and it cost of actually finding the page to replace is not going to be very high compared to the miss cost right. How many instructions will need to be get executed to find the page the to find the best candidate for replacement, its only some memory accesses after alright.

And, whatever those memory accesses are you can actually execute million memory accesses before you actually even reach close to the miss cost right. So, you know because the cost to actually find the replacement element; even in a fully associative cache is not very high compared to the miss cost.

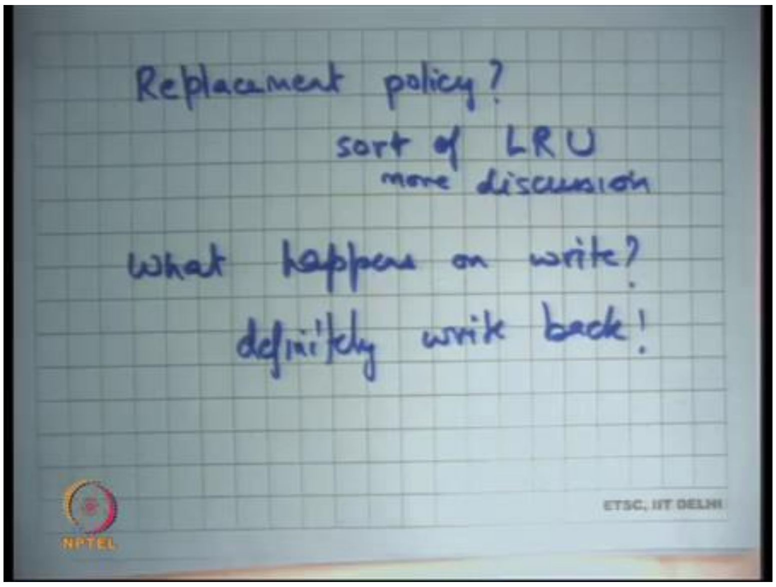
And, because hit rate is of prime importance, it is a fully associative cache makes make sense right as supposed as set associative or a direct mapped here alright. So, let say you know how does one; so, what is the hit path? Well, we want that the hit path should be released fast which means how does one how does the program, how does the system check whether the access as the hit.

Well, the a hit actually has no software involvement, it goes purely in hardware. So, an instruction executes, it has a memory address; the first thing you are do is you check the memory address, translate the memory address using the TLB right. If you hit from that you have just done the translation in 1 nano second; so, that is the; that is the fastest hit you can have right that is 1 nanosecond.

If you do not hit the TLB then you do the page table walking, page table walking will be 10 to 100 nanosecond. So, that is the second level hit in some sense. And, if you hit if you miss there then you do the disk access which is milliseconds alright. So, hit path is basically completely implemented in hardware and miss path will require software involvement because, the miss path will require the execution of the exception handler alright.

What is the miss path? Well, miss path is page fault right and this is much slower alright. And, let see what have what is the page replacement policy or a replacement policy, every cache must have a replacement policy right.

(Refer Slide Time: 18:37)



Replacement policy?

sort of LRU

more discussion

What happens on write?

definitely write back!

So, what is the replacement policy? Basically, you want to bring a page into the cache, and you have to figure out and your cache is already full. So, you have to figure out one page whom you are going to replace and as I said its very important to maximize hit rate. So, what should be the replacement policy? You know it should be some kind of fully associative policy requires more discussion, but sort of LRU; least recently used.

You have seen least recently used before in computer architecture or something alright. So, sort of LRU and we are going to discuss this more alright. Finally, what happens on a write? So, recall that a cache has needs to have a write policy. So, if you write something should it be a right through, which means it goes all the way you know on every write you go all the way to the bottom or should be a write back?

Student: Write, write, write.

Write definitely, write back right because if its write through then every write is going to access, we access a disk space; so, definitely write back ok. Now, alright so, let us see how does the operating system implement this. So, block size 1 page this is handled by the page table, associativity is fully associative, this is you know this is the page this is handled in the software by the page table handler.

So, this is wrong, and this is right, a hit path is hardware. Now, but in the hit path you also need to so, the hardware should also tell whether it was a hit or a miss right. So, the hardware is telling whether it is a hit or a miss using what?

Student: Page fault.

Using the a page fault. And how does it know whether it needs to generate the page fault? By looking at the page table entry and the.

Student: (Refer Time: 20:43).

Present bit in the page table entry. So, that is the way you need a fast way of figuring out whether it is a hit or a miss and the fast way is basically for the software to set the present bit and the hardware to read the present bit. So, basically the present bit in the page table entry is helping you to figure out whether it is a hit or a miss and you need to do it fast.

So, it is happening in hardware alright and miss path is the page fault handler. Replacement policy is implementing in this, but what happens on a write? Definitely write back, how does you know how does the hardware indicate that this you know this page is actually needs to be written back alright.

So, how does the how can the hardware indicate? So, basically if it is a write back policy the program should execute at full speed and it should just keep writing to these pages. And, at cache replacement time you will basically figure out whether this page needs to be written back to the disk or not.

One conservative policy is that whenever you replace a page you always write it back right, that is unnecessary because it is quite possible that the page was only read from and never written too. And so, the contents have not changed so, you do not need to write it back and.

Student: (Refer Time: 21:50).

So, you can save disk accesses and recall that disk saving disk access is a huge optimization. So, what is some other way of letting the for the hardware to let us know whether this page has been written to or not or modified or not?

Student: Maintain a dirty bit.

Maintain.

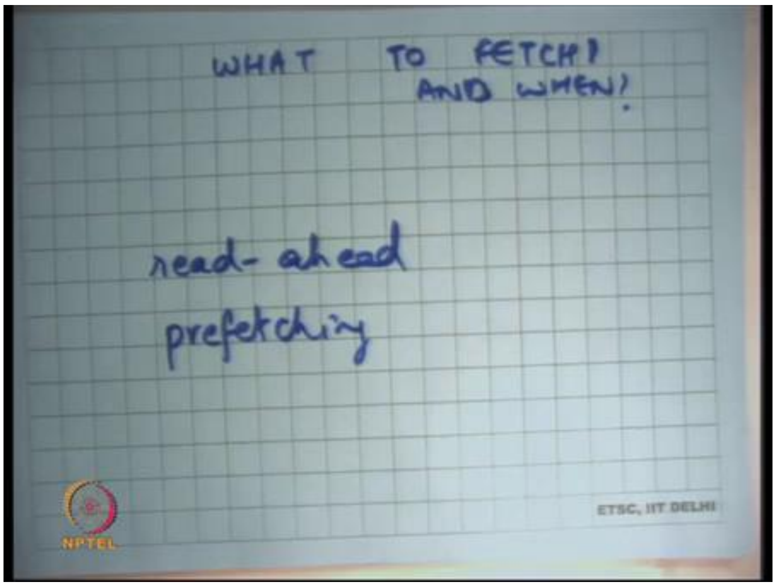
Student: (Refer Time: 22:06).

A dirty bit in the page table. So, one option is so, what is done on x86 is that the page table entry has another bit called the dirty bit. The operating system when it brings the page into the cache clears the dirty bit and if the page is written to the hardware sets the dirty bit. Notice that because the setting of the dirty bit is on the hit path it should be fast and its done by hardware ok.

At cache replacement time the replacement, the page fault handler will basically look at the dirty bit to figure out whether it need to write it back to disk or not write it back to disk. The dirty bit will be set by the hardware, it will be cleared initially by the software because when it brings the page in it will be cleared.

Because, at that time the page is clean and by clean, I mean that the contents of the page are identical to what they exist on the disk right. So, at the as soon as you bring the page into memory its clean, if you if the program ever writes to that page the dirty bit and the page table entry get set alright right.

(Refer Slide Time: 23:19)



WHAT TO FETCH? AND WHEN?

read-ahead

prefetching

So, now let us also look at what to fetch and when alright. So, for example, what should I fetch at load time? Right. So, this is the design question that an operating system designer has to figure out that you know at load time, if you basically say I want to run this process what pages should be fetched into physical memory a priori.

One option could have been that I just ask the user. So, you will provide an interface between the user and the operating system. For example, have an extra argument into your exist system call saying this is the list of pages that you should load apriori right. This is possible, has been tried, but the problem with this is number 1 you do not want to trust the user and the user may say you load all the pages.

And, you know you do not want to really satisfy his requires because that can lead to other process is having poor performance. Secondly, the user can user himself actually does not know and does not want to care about this right. When you write a program you do not care about what pages are hot and what are not, you basically want to basically say that you know here is my program you done it and you figure out what is hot and what is not and I will just manage things efficiently for me right.

So, asking the user is both sort of not very trustworthy and its it complicates the interfaces and makes puts more responsibility on the shoulders of the programmer than its actually really needed. The other thing is you could load some initial pages that load time, you can say that you know whatever is the first program counter that needs to with executed I will load one or two or three pages around that.

So, some kind of heuristic policy around that, then you load some pages which are for your stack and maybe some data pages based on some heuristic which in operating system is free to choose and different operating systems will have different variants of this. And, then as the program execute its going to access the memory and then implement demand paging.

Each time there is a page fault once again you have a choice, do you only fetch the page that was faulting that was faulted upon or do you fetch some pages around that also. So, you can say you will have something called read ahead which is basically for spatial locality. So, whenever you fetch a page you also fetch some pages around that page.

So, that is called read ahead alright and that is basically that is going to help if you are likely to have a lot of spatial locality in your program. And, most operating systems will do some sort of read ahead somewhere right and when to do read ahead and then not to read do read ahead is also a bit of an art. And, you know many heuristics are present in real operating system to figure out this.

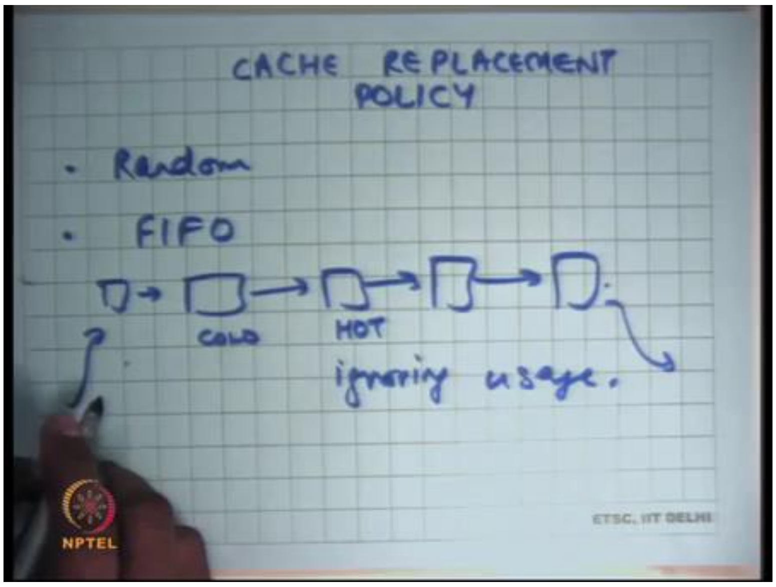
What is the tradeoff of doing too much read ahead? You may be polluting the cache with unnecessary data. And, what is the; what is the what is the disadvantage of doing too little read ahead? You may be encourage too many page faults unnecessarily right. So, that is basically also you know you can do what is called prefetching.

So, this is for temporal locality, if you basically say that every time you access page x as see as notice this pattern that every time you access page x you also access page y. So, if the operating system under the covers has been noticing this pattern in your program then it will basically say ok.

You know each time you take a page fault on page x let me also prefetch page y and this can happen for example, you know whenever you access this particular code region, this particular code always accesses this particular data structure right or there can be many such example; I am just giving you one example right. So, notice that you know the operating system is inferring all these performance decisions under the hood without the programmer having to know anything about it.

And, that makes the programmer you know really carefree about these things and that is a nice thing and it has worked for many years now and it has some limitations. However, and for example, in modern hardware like multi core you know this kind of inferences are harder and harder to make. And, all kinds of ideas have been proposed and on how to do this better with or without programmer involved it alright.

(Refer Slide Time: 27:13)



CACHE REPLACEMENT POLICY

- Random
- FIFO

COLD
HOT
ignoring usage.
NPTEL
ETSC, IIT DELHI

The other thing is cache replacement policy right. So, what page to evict? So, you have figured out that this is the page I want to bring in or this is the set of pages I want to bring in. So, if you have decided that these are the set of pages you want to bring in, what are the pages you are going to throw out or reject right?

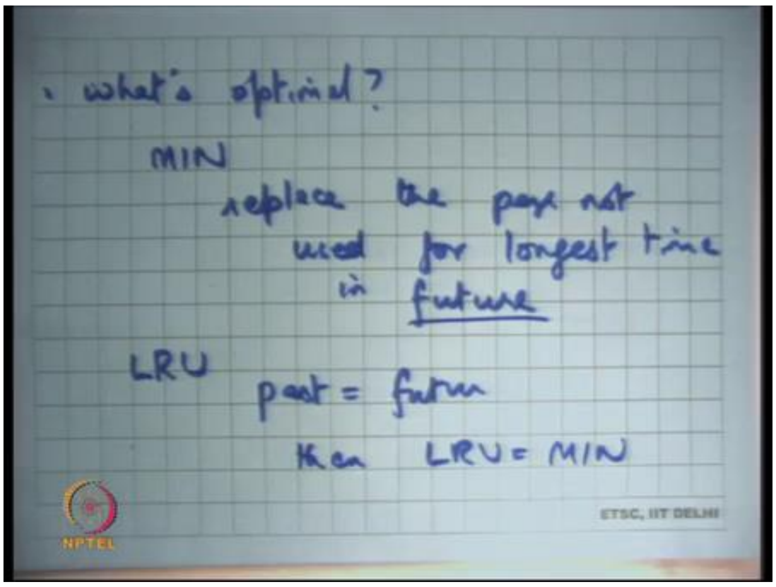
So, the first policy is random, just pick up any page or any set of pages in the thing and just in your cache and just eject them. The advantage of random is that it is very simple, you do not need to maintain any data structure.

The disadvantage is that you may actually be ejecting hot pages and that is not a good thing right. The other approach is FIFO whichever page has been brought in last you know earliest should be ejected. So, basically the way to implement it is basically have a cube in your memory, it's basically says this is you know every time you bring in a page you append to the cube. And, every if you want to eject you eject from the tale of the cube right.

Once again the nice thing about FIFO is that, it is very easy to implement; each time you bring a page in you add just to the data structure, adding to the data structure is cheap with respect to the full cost of the page fault handler, you are also making a disk access. So, you know just adding to the list is very easy, also you know removing from the list is very easy, it's very cheap. You know you just anyway ejecting the page you probably going to have a disk right or anyways taken a page fault; so, you know ejecting is easy.

So, but the disadvantage is that this organization completely ignores the usage. So, let say this page was hot and this one was cold right and you bringing in some page. So, because its FIFO the hot page gets removed and the cold page remains; so, you are ignoring usage alright. So, let us you know let us turn it around and let say what would have been the optimal.

(Refer Slide Time: 29:23)



- what's optimal?

MIN

replace the page not used for longest time in future

LRU

past = future

then LRU = MIN

ETSC, IIT DELHI

So, let say what is the optimal page replacement policy? Assuming I have an oracle who, can tell me you know who can give me every all the information I need; what would have been the optimal thing to do? So, the optimal thing would have been that if I am bringing a page in I want and if I have an oracle who can tell me what the future is going to look like; the I can ask the oracle what is the page that is likely to be used farthest in the future you know.

So, of all the pages that I have what will page that is going to likely to be used farthest in the future. So, that is the page I want to replace because all the pages, all the other pages will be used before that page. And so, you know you want hits on those pages and the one that is farthest that is the one you want to replace right.

So, let say let us call this policy the minimum policy which is basically replace the page not used for longest time in future right. Of course, I am assuming that there is some oracle that is telling me what the futures going to look like, and you know this is impractical. I do not know what the future is going to look like which pages are going to get access, depends on what paths the program takes right. And, I cannot predict as an operating system what paths the program is going to take.

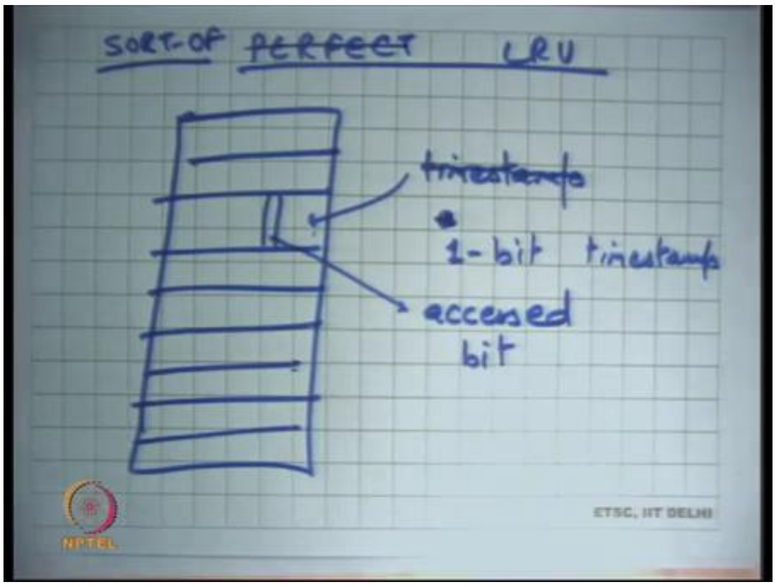
I can guess them, I can ask the user for them, but these are all sort of just heuristics or and asking the user is actually very error prone; so, you do not want to do that. So, LRU is an approximation to so, assuming; so, one common principle design principle is that if you do not know the future assume that future will look something similar to your past.

So, what is just happened in the past, the future is going to look something similar to that alright. So, if I were to if I were to sort of make this assumption that past is equal to future then instead of saying replace the page not used for longest time in future, say replace the page not used for longest time in past.

So, whatever is the page that is not been used for the longest time in the past that is the page you will replace right. So, just replace past with future, if past is equal to future then LRU is equal to MIN, if past is roughly equal to future LRU is roughly equal to MIN alright; so, that is a idea ok.

So, we looked at how random is implemented, we looked at how FIFO is implemented; clearly you cannot implement MIN, it is a completely impractical algorithm. But you can implement LRU and let us take let us look at what it takes to implement the LRU alright.

(Refer Slide Time: 32:01)



So, one way of implementing perfect LRU is that let say you have a page table or a virtual memory space whatever you want to call it. And, these are the pages, each time you access a page you also store a timestamp with when it was accessed.

Each time you access a page you know record the timestamp at when it was accessed and then when you want to replace you go through all your pages and look for the page with the smallest timestamp alright. Now, who should put the timestamp on the page?

Student: Hardware.

Hardware right because it is a it has to be on the hit path. So, if I cannot I do not on the hit path there should not be any software involvement, I want the hit path to be as fast as possible. So, the timestamp has to be put by hardware and so, that does not sound very practical right. I mean what kind of timestamp should be put by the hardware and how big the timestamp should be.

And, each time if I have to put a timestamp then there is extra overhead. So, recall that you know each time I put a time stamp I have to make a memory write in some sense to put the timestamp and that is not; that is not great. Also, at the cache replacement time I have to go do a global sort on the timestamps and then figure out which is the least right.

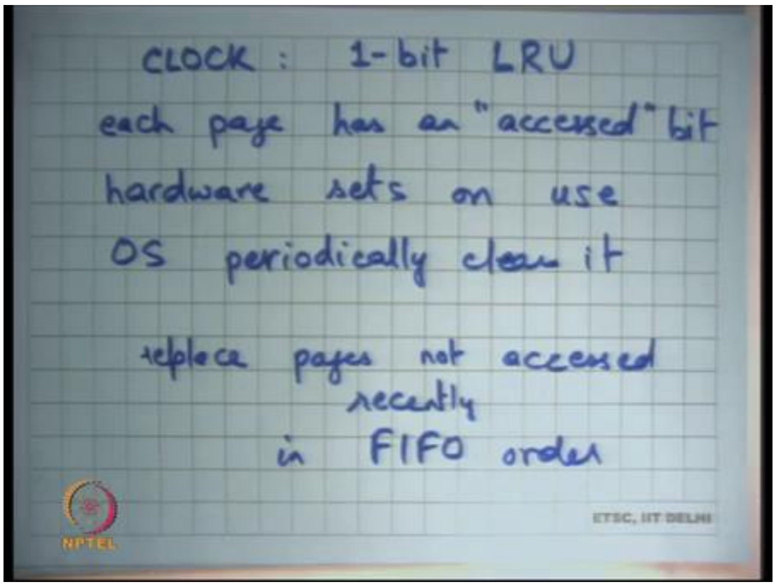
The global sort may not be that much of a problem given that anyways there is a disk access involved, but you know on the hit path having to put this timestamp is not practical. So, what is done is actually not perfect LRU, but you know let say sort of LRU.

So, sort of LRU basically says that let us say instead of putting a full-time stamp on your page, on your axis let us approximate the time stamp by some number of bits alright. So, let say I have a 1-bit timestamp. So, instead of a timestamp that is likely to be you know very long timestamp you know I just have a one-bit timestamp and that timestamp is going to say whether this page has been accessed recently or not right.

So, each let us so, basically the idea is that there is just 1 bit inside this page table entry. If this bit is and this bit is call the access bit which says whether this page has been accessed or not right. When you load the page initially you clear this bit, you set it to 0 and then when it gets accessed the hardware will set this bit to 1. If this bit is already 1, then it gets accessed, then it remains 1 it does not change right.

So, it is a 1-bit approximation to a to a full timestamp, basically saying whether it was accessed at all since a last time it was loaded or last time it was checked right. So, let see what a how this works.

(Refer Slide Time: 35:19)



CLOCK : 1-bit LRU

each page has an "accessed" bit

hardware sets on use

OS periodically clear it

replace pages not accessed recently in FIFO order

So, this algorithm is also called the clock algorithm, which is 1 bit, now can be thought of as a 1-bit LRU ok. Each page has an accessed bit or a reference bit right; hardware sets this one use and OS periodically clears it alright. So, the idea is that the operating system will periodically look through all its pages and the pages which have the accessed bit set it will clear those that bit.

And, then it will again periodically look for it and then if it finds that the accessed bit has been set between the last time it looked at it, then basically means that it was accessed in this region. So, it basically means it was accessed recently, something that was not accessed in this time period was not accessed recently. So, it is a one-bit approximation to LRU in the sense that you are differentiating between a page that was accessed recently whose access bit is 1 and a page that was not accessed recently whose access bit is 0.

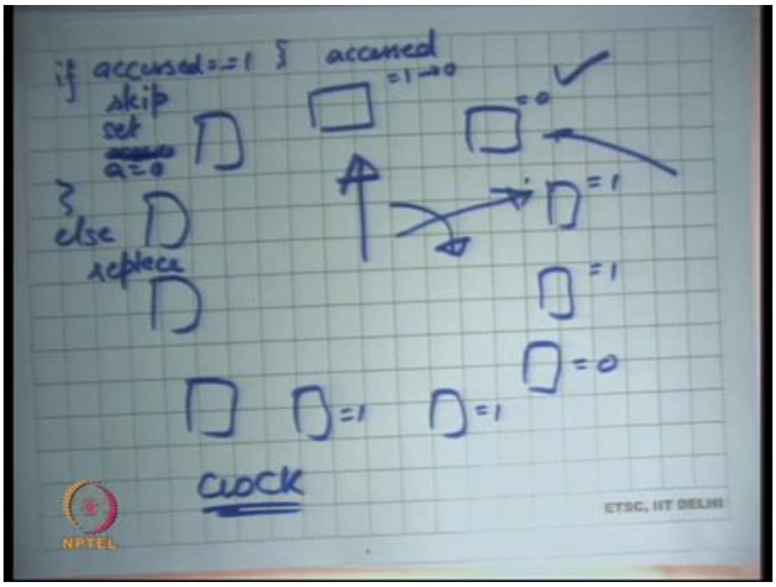
And, the notion of recently is basically the time between two checks by an operating system alright. So, instead of doing a full timestamp and sorting on the timestamp you just basically divide the pages into two categories accessed recently and not accessed recently.

And, you replace the page which has not accessed recently, and you do not replace the page that has been accessed recently alright ok. So, the way it works is that replace pages not accessed recently in FIFO order alright. So, FIFO had some merit to it right, FIFO basically said that whoever comes first goes first right.

And, it has some merit because it is quite likely that your programmed first is looking at this data structure and so, it loaded with lot of pages and now you are looking at this. So, FIFO does captures some part of recency in its (Refer Time: 37:41).

It is the recency of the first access to the page and the nice thing about FIFO was that it was very easy to implement right. LRU was much finer grained in figuring out who accessed what recently, but the problem with the LRU is that it is too expensive to implement. So, clock is a 1-bit approximation to LRU that does 1 bit of differentiation between what is accessed recently or what is not and, for everything else and so for pages in the same category it just use a FIFO alright.

(Refer Slide Time: 38:17)



So, you know purely implementation standpoint one way to think about the clock algorithm is that you arrange all the pages in a clock in a circle. And, your hand that is your clock hand that just moves let say clockwise alright. Let say you want to replace a page; you just move the hand till you find a page that has accessed bit equal to 0. So, each of these pages as an accessed bit equal to 1 0 1 0 1 and so on let say.

And so, the clock hand is going to so, let say I want to replace the page I will move the clock hand. Any page that I find that it has been its accessed bit is 1 I will skip it, I will not just skip it I will also change it to 0. Basically, means that I have seen it in this my in one revolution, I have seen it and I saw it has accessed. And so, I clear it to 0 to basically see that by the time I reach it next has it been converted to 1 or not.

The moment I see a page that has not been accessed since a last time I cleared it which means its accessed bit is still 0 I will take that for replacement right. So, this algorithm I mean this is this algorithm is basically doing what I said before its differentiating between pages that have been accessed recently. And, now here recently means accessed in the last clock in the last revolution of this clock hand, in the last one full revolution of this clock hand right.

So, its distinguishing page between pages that have been accessed recently and pages that have not been accessed recently for the pages that have not been accessed recently, it is replacing them in a FIFO order alright. So, basically why is it a FIFO order? Anytime you pick a page to replace; so, let us say you pick this page to replace you will add the new page at this location and move the clock hand next right.

So, the page has just been the new page will get added just before the clock hand right. So, the pages the all the not accessed pages are replaced in FIFO order because, of that. The clock hand just sweeps through all the pages, the algorithm may if it finds the page that has bit equal to 1, it marks into 0 and skips it. If it finds a page that has bit equal to 0 it replaces it alright and it puts the new page at that position and the clock hand just points to the next page after that. So, it is pushed at the tale of the tube in some sense.

Student: And, the pages which are not sweep, they are not set to 0.

The pages that are not sweeped are not set to 0, but they will be set to 0 the next time the clock moves in that direction right. So, even the pages that have been set, that have been accessed are being examined in FIFO order right. There is there is an order in which they have basically being examined. So, we basically have two categories; recently accessed, not recently accessed and you are processing both of them in FIFO order.

And, one way to think about it is basically you arrange the pages in a clock, and you let the clock hand rotate. And, you use this algorithm that if accessed is equal to 1 skip set accessed is equal to 0, set a is equal to 0 let just say that alright else replace. So, each time you need to replace a page you just move the clock hand and you basically figure out which page to replace alright.

When you replace you would you know should you set a is equal to 1 or should you set a is equal to 0? Both are possibilities, but typically you would set a is equal to 0; we will let it remain as it is and see where the page program accessed or not.

Likely, to that program is going to access at after all you know that is the reason it being brought in if its it should be, but let say you are doing read ahead then you know you brought in some extra pages. And so, you have set the access bit to 0 and so, and if let say the read ahead pages were not the read ahead pages were not access.

So, then they will get replace in the next revolution alright. So, this is how it works. So, what does it mean to say that the hand is sweeping really fast? Now, it just going; so, each time you know by saying that the hand is sweeping really fast I am saying, if there was a page fault and I wanted to bring a page in I had to go through a lot of pages before I could find something that could be replaced. What does that mean?

Student: (Refer Time: 42:52) the pages are used.

Student: Pages are being (Refer Time: 42:54).

That basically means that if the pages are really being used, many pages have being used in other words.

Student: (Refer Time: 42:59).

The hot memory footprint of the program is very large, it is probably larger than what the memory can actually support. In other words, you need to increase the size of a physical memory right. So, if the physical memory is smaller than the size of the hot memory footprint, also call the working set of the program then you have the you will see the that the hand is has to move a lot of huge in a very fast right.

Let say I am doing a completely synchronous replacement, basically it may happen that each time I want to replace a page all the pages have been accessed before I get to I get page before I basically see a page.

So, I have to do a one full revolution before I actually get to do a replace of page. So, for every page that I replace I basically do one full revolution; so, that is you know that is really bad. You are basically doing a full global operation going through all the pages and also means that you are basically having a lot of you know you would likely to have a lot of misses.

If the miss rates are very high and shows that you are you need to increase your memory size. On the other hand, if the hand is sweeping really slow it basically means that your memory is larger than your working set size or the set size of the hot memory regions in your code.

Student: Sir.

So, the question is that.

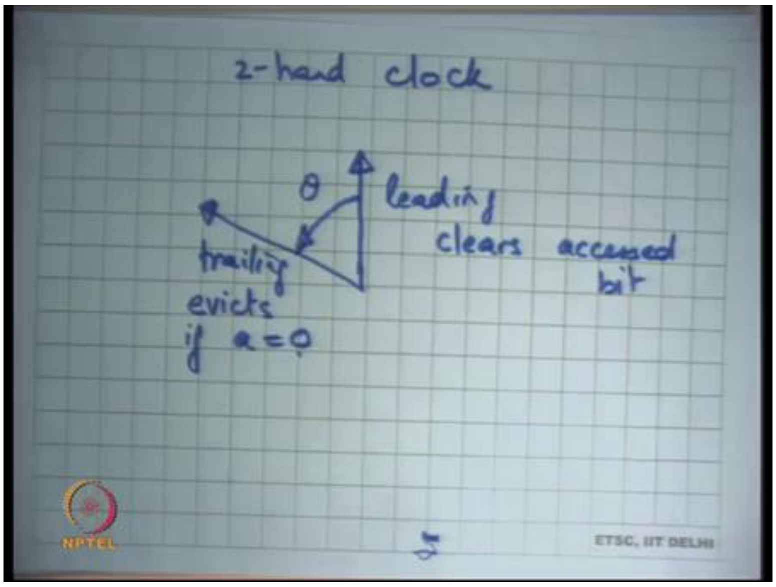
Student: So, all the (Refer Time: 44:13).

If I have to make a full revolution that basically means that all these pages were hits before I actually find something that I actually get to replace right. So, does not it mean that because all these pages were hits, does not that mean that the miss rate is small. Well, no not really right because all these pages are hits, but there are other pages that are also getting accessed basically means that all these pages are.

So, all the pages in your memory are hot and yet you have basically bringing in you know you are taking misses on other pages that also might be hot right. So, you basically taking misses and you your working set size seems to be bigger than your physical memory; so, your likely taking a lot of capacity misses.

It because all these pages are hot; so, you are finding it difficult which page to replace, that clearly shows that you know your miss rates are likely to be high ok. Another sort of modification to this is what is called a 2-hand clock.

(Refer Slide Time: 45:17)



So, if your memory size is really big then you do not you may not want to make a full revolution on your entire memory on every page fault. And so, what is done is basically instead of having 1 hand, you have 2 hands. One is called the leading edge and other is called the trailing edge. The leading edge clears accessed bit and the trailing edge evicts, if accessed bit is equal to 0 right.

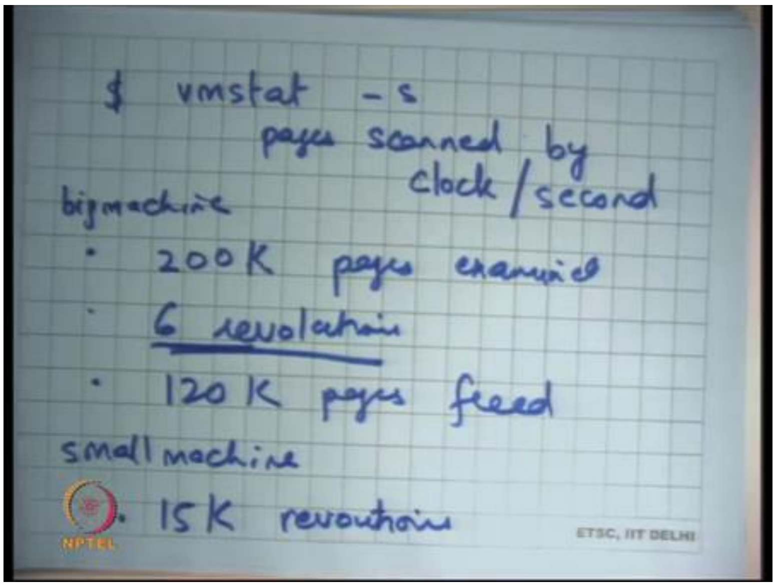
So, what have you done here? You are basically set that the leading edge clears the accessed bit and the trailing edge evicts, if access is equal to 0. If you are saying that the page will get evicted, if it gets accessed in this interval. In the 1 hand clock I was saying that the page will get evicted, if it was accessed in one revolution.

Here I am saying the page will get evicted, if it gets accessed in this interval and both of these; so, this angle is fixed. Let say this angle is theta. So, the theta is fixed. So, whenever that leading edge moves the trailing edge moves along with it right and so, a page get replaced if it was not accessed within this theta interval.

So, instead of theta being 360 as it was in the case of 1 hand clock, now theta is something smaller and you can choose theta depending on what your memory size is and what is the maximum overhead you want to have on your page fault. So, and so this basically make sure that you know you will examine only a that many number of pages before you actually find something to replace alright. If the angle is too small what does it mean? Let say theta is equal to 0, it basically means it is a FIFO because, the leading edge clears the accessed bit and trailing edge is just evicts it immediately.

So, basically means you did not care about the accessed bit at alright; on the other hand, if theta is equal to 360 you are back to 1 hand clock alright.

(Refer Slide Time: 47:37)



```
$ vmstat -s
        pages scanned by
              clock / second
```

big machine
- 200K pages examined
- <u>6 revolutions</u>
- 120 K pages freed

small machine
- 15 K revolutions

ETSC, IIT DELHI

So, you know just as an example let say you know, I had a machine may find you know if you there is a command called vm stat on Sun OS. And, which allows you to check how many pages were scanned by the clock algorithm on operating system. So, clock algorithm is a commonly used algorithm in inside operating systems.

And, you can check what are the statistics of your clock algorithm on your operating system which will by using this command on some operating system, that tells you what are the pages scanned by the clock per second. So, for example, you know here is some example data. So, on a big machine big machine means lots of memory this slightly whole data, but let say you know roughly 200,000 page is examined. 6 revolutions of clock hand and 120 K pages freed alright, roughly speaking.

So, you know the 6 revolution basically shows that the clock hand is moving relatively moderately, its memory pressure is not that much. On the other hand, if you have a small machine you know here is another piece of data you could have something like let say 15,000 revolutions per second.

This basically shows that your memory is basically too small and you basically need to increase your memory to reduce your memory pressure and so, that your clock hand moves slowly alright.

So, let us stop here and continue this discussion next time.