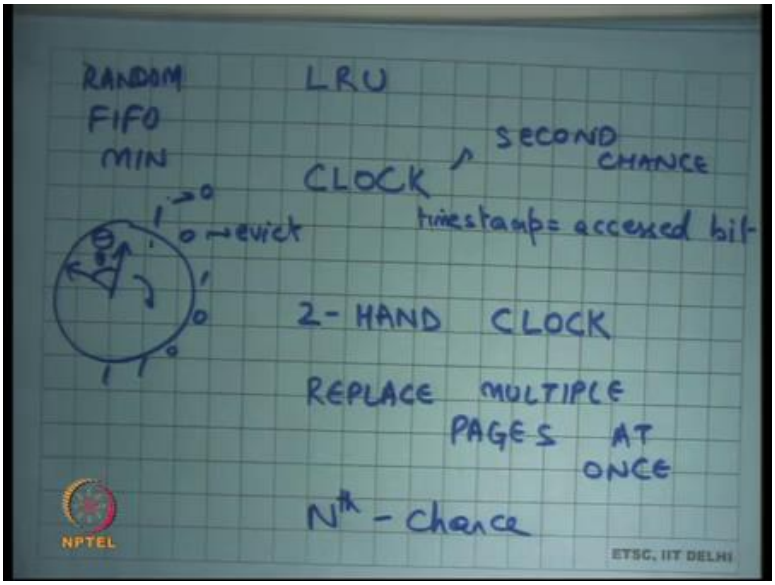
Operating Systems
Prof. Sorav Bansal
Department of Computer Science and Engineering
Indian Institute of Technology, Delhi

Lecture - 30
Page Replacement, Thrashing

Welcome to Operating Systems, lecture 30, right.

(Refer Slide Time: 00:31)



So, last time we were talking about Page Replacement algorithms and we looked at random, we looked at RANDOM, we looked at FIFO. We also looked at a hypothetical algorithm which we called MIN which is let us say the optimal, which assumed the knowledge of future and we said that assuming that past is equal to future LRU is the best approximation of MIN. So, if past approximates the future LRU approximates MIN.

But then we said LRU is too difficult to implement because you need to put a time stamp on every memory access and recall that memory accesses are on the hit path and they need to execute on you know very fast. So, putting a timestamp on every memory access is a very expensive operation. So, it is not practical in general.

So, LRU, so implementing perfect LRU is hard because you need to put a timestamp on every memory access and that needs to be on the hit path. And the you know, so putting a timestamp on every memory access is not possible instead clock is a 1-bit

approximation to LRU where the timestamp is just 1 bit which says whether it was accessed or not. So, there is just 1 bit and this bit is stored in the page table entry. Recall that there were these extra bits in the page table entry and so one of them is used on the (Refer Time: 01:52) disk hardware to represent the access bit.

And so, the hardware basically sets the access bit. The first time the page if the access bit was 0 and the page gets access then the access becomes bit becomes 1. If the access bit is already one and the page gets access nothing happens, right it just remains 1. So, that is a 1-bit approximation to LRU, and we said that you know that basically. So, we basically use FIFO except that we use this 1 bit to distinguish between pages that were access recently and that were not access recently.

Pages that were accessed recently are not replaced they are instead move to the other list for the first time. So, you basically examine these pages in FIFO order and if you see a page that has been accessed recently you clear its access bit indicating that it has been moved to the lower level, but if you see a page in the lower level in FIFO order then you basically evict it. The first page you see it, right.

So, that you know the one another way to think about it is that it is a clock and you keep a pointer to the last page that you accessed and you move the clock in one direction and these clocks have you know access bits 0, something like this. And, each time you come across a page whose access bit is set you set it to 0, and if you see something which is already 0 you evict it, right. And because you are storing the pointer, the next time you run the page replacement algorithm you going to start from the where you left off. So, it turns it looks like FIFO in that sense, right. Basically, you are just advancing the pointer from the from where it was last time.

What is the logic behind setting the one, setting the access bit from 1 to 0? Well, I mean basically the idea is that if a page was is 1, then I set it to 0, so that I want to observe whether it is you know I have seen it at once and now the next time I am more interested in seeing whether it was access in the last revolution or not, right. I do not want to carry over information from 10 revolutions behind, I am not interested. I am only interested in knowing whether it was accessed in the last revolution or not and to be able to do that I need to clear it. So, that it gets set again.

Student: So, cannot we traverse set all of them, set all 1s to 0?

Cannot we traverse all of them and set all of them to from 1 to 0? I mean this is this is in some sense doing that except that it is not a global operation, you are just going in a sequential way. I mean the net effect is similar, right. You are just going one by one and whichever you see as 1 you send it to the 0, and so each page is getting in equal time in to get accessed which is one revolution or roughly equal time you know on average. It is also metric anyways, right.

And, and, but of course, we said that you know if the number of pages is really large and the time that a page gets to prove that its actually being accessed recently maybe too large for your for what you need and so you may want to use a 2-hand clock. I am just say that the leading hand is going to clear the bit and the trailing hand is going to evict. So, the leading hand just clears the bits if it finds the 1 and the trailing hand evicts if it finds a 0. And so, for a page to not get evicted the page should have been accessed between the leading hand and the trailing hand, that is the only way the page will survive eviction. It will not be able to survive it is the past.

And so, you can tune that by using and so this angle is fixed between the leading edge and the trailing edge that is let us say, let us call that theta. So, this angle is fixed and depending on how much time you want to give to a page to really allow it to be called accessed recently you will choose theta. If you choose theta to it will too large you basically using you giving it the page more time, if you choose theta to be too small then you are giving page too less time. And you know extreme cases are theta is equal to 0 you are actually not giving the page any time at all and it is just FIFO at that point. And if you make theta is equal to 360 then you are basically back to 1-hand clock, right, alright.

So, there are some, so you know the clock algorithm is fairly successful in the sense that it is actually being used for many years to do page replacement and here are some more extensions to the clock algorithm that are usually used.

So, one is you know replace multiple pages at once. So, instead of replacing one page at a time; so, each time you get a page fault you may you actually just obliged to replace one page instead of replacing one page you may want to replace multiple pages at once; so, that you would save you know. So, firstly, if there are any 30 pages you can write all of them in a batch to the disk. And, so as we know that disk is dominated by the seek and

latency time, rotational latency it is better to write a bunch of pages together rather than one page at a time. Also, it saves; so, yeah, so I think it is basically about you know saving the number of writes that you can have.

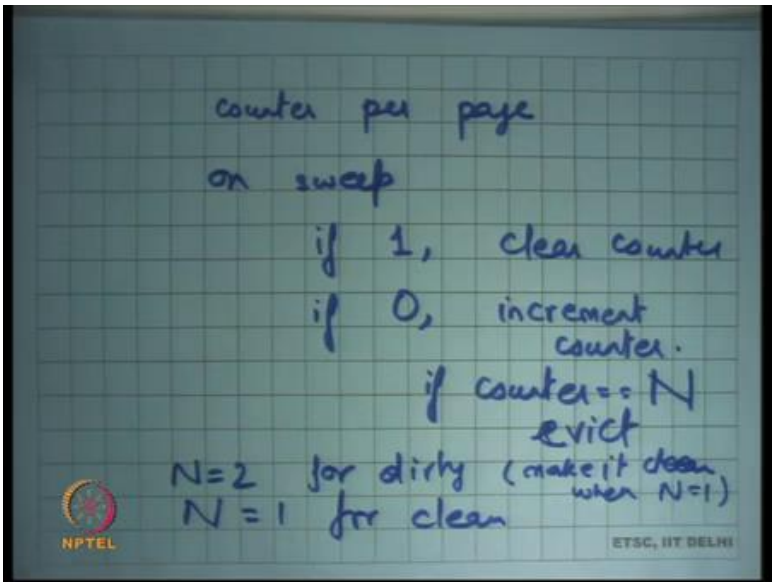
Another sort of straightforward extension to the clock algorithm is what is called the Nth chance algorithm. So, one way to think about the clock algorithm is it is also called a second chance algorithm where pages that have been accessed recently are given a second chance, right.

So, you basically process all your pages in a FIFO order, but if you see that a page has been accessed recently you give it a second chance which means you set it to 0 and you give it another chance to basically prove itself once again, right. And, if it is able to prove it is again which means it gets access recently you given give yet another chance, but if it is not able to get access recently then it gets evicted. So, it is a second chance algorithm.

The Nth chance algorithm is a just a straightforward extension you say instead of giving it two chances give it N chances. So, the idea is that each time you see, so you go along the clock you see one bit, you set the counter, so we maintain a counter with every page which says how many times you have seen this page to be not accessed, right; so, how many rounds have you seen this page to be not accessed

And each time you see a 1 bit in that in that page you clear the counter to 0. Basically, means it was just accessed recently to the counter 0. If you say see it as 0 then you increment the counter, you do not replace it immediately you increment the counter by 1, right. And when the counter reaches N which is where N is the you know Nth chance of then you replace it, right. So, let me just write the Nth chance algorithm to make it clearer.

(Refer Slide Time: 08:29)



Counter per page on sweep by the clock hand, if you see a 1 then clear counter, if you see a 0 then increment N, increment counter, right. Also, if counter equals 1 replace or evict, alright. So, in other words the clock algorithm as we seen at the second chance algorithm is just Nth chance with N is equal to 1. The first time you see it as a 0, you increment the counter and the counter becomes equal to 1 and you replace it. Did I say N 1, it should be N.

So, the clock algorithm is the Nth chance with N equal to 1, but if you want to give a page more chances to basically prove itself then you know you increment. The more, the higher you have the larger N, the Nth the more revolutions you have to make before you actually evict a page or to actually see the page as not accept for at least you know N revolutions before you actually evicted. So, giving a page more chances in that sense; also, the higher the N the closer you are to LRU, right.

So, basically let us say N was equal to 1000 basically saying that this page has not been accessed the 1000 times and so this is the best candidate to get evicted, right. On the other hand, if there is a page that was accessed in the last 1000 revolutions before you know will get priority over, so you are getting finer grained. You are distinguishing between a page that was accessed a 1000 times and a page that was accessed that was not accessed across a 1000 revolutions and that was not the page that was not accessed across 900 revolutions let us say, right.

So, the higher the N the more fine grained you are getting. You are distinguishing between pages that have been not been accessed a 1000 time for a 1000 revolutions and a page that has not been accessed for a 900 revolutions. So, the higher the N the closer you get into LRU. The downside is the more work you have to do to find a page to replace, right, ok, alright.

So, you know if this is, so the Nth chance algorithm with a large N is one way to approximate LRU, but in general clock algorithm works just as well as LRU. LRU is anyways an approximation to the to the what is what is optimal and clock algorithm is approximation to LRU and the in general it works good enough. So, you do not need to do these Nth chance Nth chance or kind of kind of things or you do not need to have a very large N to basically have good hit ratios, right. The whole point of doing all this is basically to have good hit ratios without having too much overhead on your hit path and also on the miss path.

One common one common thing that is done is treat dirty pages preferentially over clean pages. So, when you are doing replacement you can imagine that if there if you make a dirty page you will have to do more work because dirty page if eviction requires a right to the disk, a clean page eviction does not require a right to the disk. So, keeping this in mind you may want to give more priority to your dirty pages. So, you may say what is the dirty page let me give it more priority over your clean page. So, if I have a choice between dirty and a clean page, I will probably pick the clean page to evict, alright.

So, you know one common way one common thing that is that is sometimes done is use user Nth chance algorithm with N = 2 for dirty pages and N = 1 for clean pages, right. So, evict the evict the clean page the first time you see it not accessed evict the dirty page the second time you see it not accessed, alright.

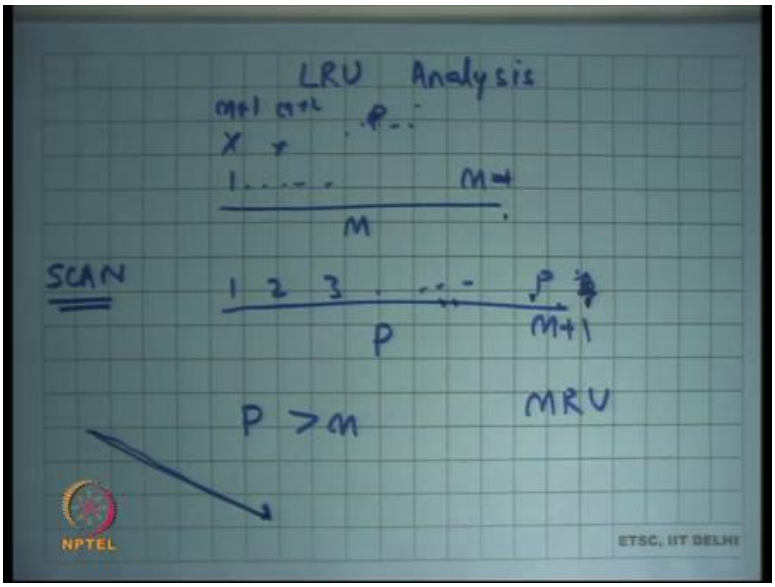
Also, in this scenario you make it clean when N = 1, right. So, let us say you went through that page and you figured out that it has not been access in the last revolution. So, you do not evict it immediately, but you also say that you know let us just write it back disk, so that by the time I you know hopefully by the time I come to it next it becomes clean.

And you know this, right back to the disk can be done in an asynchronous fashion in a batch fashion, so lots of pages, lots of dirty pages can be collected together and written

back. So, if a page has not been see seen access in the last revolution which is you know at N = 1 for the dirty page, you can just put it in your queue to be written back to the disk and you know because it is likely to get replaced in future, there is you know there is a some indication to use there. So, you make it clean at N = 1 and you replace it at N = 2, right. For clean pages you can just replace it at N = 1.

So, just one example of Nth chance of algorithm used and to potentially treat dirty pages over clean pages, alright. So, let us just understand where LRU works and where LRU does not work, right.

(Refer Slide Time: 13:39)



So, LRU works very well basically LRU captures the recency how recently you have access the page and in general our workloads are you know have a lot of similarity between past and future and so recency works.

But here is an example; here is a common example of a pattern where LRU does not work. So, let us say you have a, you have some physical memory M and you are accessing a set of pages P such that P > M. Now, let us say you going let us say accessing 1, 2, 3 …. P and you were just accessing these pages in let us say repeatedly in a cyclic manner 1, 2, 3, 4, 5, 6, till P and then you come back to 1 and so on, right.

Now, in this case let us see what happens what with LRU. So, you accessed 1, 2, 3, 4, 5, 6, till N, all of them are misses in the first iteration and so, but all of them get into M. So,

your N basically becomes 1 …. M - 1, 1 …. M. Then you access M + 1 and which is the page that you replace?

Student: 1.

1, right. So, you replace 1 and you put M + 1 here and you replace 2 and put M + 2 here and so on, right, till let us say till let us say P, hope you have some. And then you access 1 again. The problem is that you know 1 the page that you replaced is also the page that you are going to access. So, here is a case where past is not equal to future. The page that has access least recently is the page that will be accessed most closes in future, right.

In this sense, in this case instead of replacing 1 it would have been in fact better to replace the page M because M is the page that is going to be access latest in future, right. So, so basically you know instead of LRU perhaps MRU most recently used which is you know completely counter intuitive would have worked in this in this in this example workload. Just pick the page that was most recently used that is a page that is you know going to be accessed further in the future and so that is going to have some hit rate. LRU in this workload will have completely 0 hit rate, you know all the pages will be all the accesses will be misses.

And this kind of a pattern is actually quite common for example, you know process going through an array of just scanning its array linearly or going through its data structure like a tree or whatever. So, basically any linear scan and repeated is quite a common scenario in real workloads and that is why we have to worry about it ok. So, what happens with a scan?

So, this kind of workload is also called a scan and notice that the worst performance of the LRU seems to perform the worst if P was equal to just 1 > M. So, let us say P = M + 1, so basically for the M + 1st page you replace the first page whereas, the next page you want to access this 1. So, you basically replacing the page that is most likely to be access next, right.

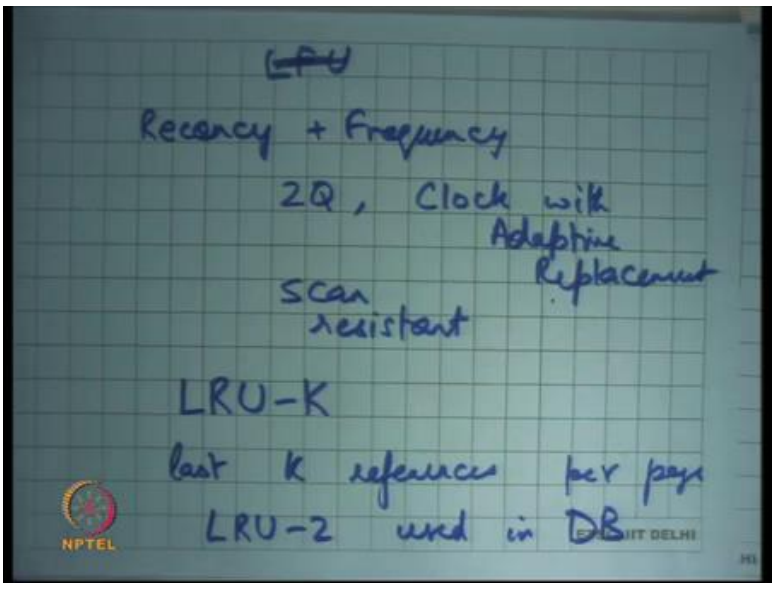
And so this kind of pattern is called a scan. And the problem with scan is that let us say typically your operating system will have multiple processes running or multiple threads running or at least multiple access patterns. So, there is somebody who is accessing it in a linear scan and there are other parts of the program that are just accessing it in a very

differential way; that means, they are always accessing the same locations again and again, but one scan can actually pollute the cash by bringing in lots of code pages into your cash.

So, all your hot pages get flushed out and the cold pages because of the scan get place in your cache and so that causes a lot of misses. Ideally, it would have been nice if your algorithm could figure out that hey this is a scan and so I do not need to do any caching for these pages, right. Anyways these are going to be misses, and rather I should focus on doing caching, I should focus on retaining my hot pages. These, all these P pages are anyways cold pages, so let them have misses. These hot pages should not get polluted because of the cold pages.

And the reason is this problem is occurring is because LRU was only looking at recency, it is only looking at what was the least recently used throwing it out, it is not looking at frequency at all. So, if there are some hot page it is getting access to 100 times and there is a cold page that was just accessed once recently, the page the cold page gets preference over with the hot page just because it was access more recently. So, ideally it should we should have some way of figuring out, some way of combining frequency also with recency, alright. So, you know there is there is an algorithm called least frequently used.

(Refer Slide Time: 18:29)



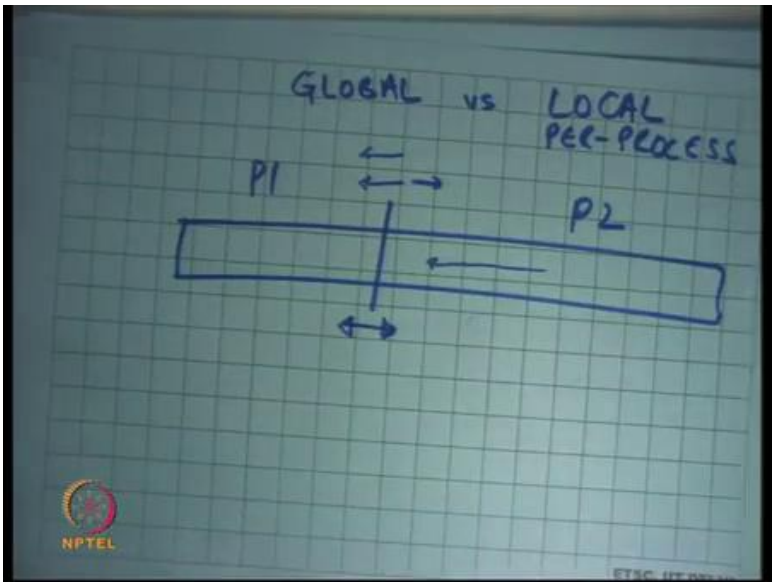
But this is this is also very impractical algorithm, you basically say what is the least frequently used page, but the problem with this is that it has a long memory. You know a page that was accessed a lot of times one hour back may just keep polluting your cache which is not which is not known typically your work typical workloads will perform badly in LFU.

So, typically you combine recency and frequency in some way and so LRU is not used in explained form and they are there are examples of algorithms which extend either LRU. So, there is an algorithm called 2Q, there is an algorithm called clock with adaptive replacement and so on which basically dos and there are many others. So, most operating systems will implement some variant of LRU that is scan resistant, right. So, because scans are common workload patterns and any good algorithm should also be scan resistant and LRU is not, ok, alright.

So, one algorithm that I will that I just to give you a flavor is called LRU-K. Here the idea is that you do not evict the page that was least recently used, you for each page you track the last K references, alright and then you evict the page which has the least timestamp for the Kth reference in the past.

So, in LRU we were just looking at the last reference for the last accesses, instead of the last timestamp with the last access you look for the last time stamp for the last Kth access and that is the one new replace. So, you know LRU-K is basically giving some preference to frequency and, but the LRU-K is also expensive to implement in the context of an operating system, but LRU-2 is used in databases, alright. Just, ok, alright.

(Refer Slide Time: 20:55)



So, now let us talk about how something like a how a cache replace in algorithm is implemented and there are two choices global versus local or per process. Should I do replacement for all the processes as considered as one pool or should I do replacement for each process separately? Right. There are two options I can consider the memory used by all the processes as one global pool and so in one page process page faults I will consider this whole pool as one replacement pool and I can take page from some other process to serve this process and other cases I have separate pools for process, right.

So, so clearly if you use global then you have more optimality overall. If you have per process quotas of memory then it is possible that one process is is under utilizing its memory and another process is over committing its memory and so you know you have artificial barriers and that is causing extra page faults and you actually need to. On the other hand, global replacement has its problem that one process can actually run away with all the memory causing all of the process to become perform very poorly, right.

So, a typical security attack or you know performance attack on a machine could be that you respond a process and it just touches a lot of memory, right. And, just keeps touching a lot of memory and so it just bringing in a lot of memories all other processes is much less memory than actually available on the system, right. And so, you know the operating system has to balance optimality or efficiency with some level of fearness ok.

So, in in practice you basically use, so in practice there are two things that are typically used, one is you know either they use completely global replacement. So, you know just global replacement assuming you have a very large memory pool and your caching parameters are correct; you know it is very hard for one process to actually pollute the cache of other processes.

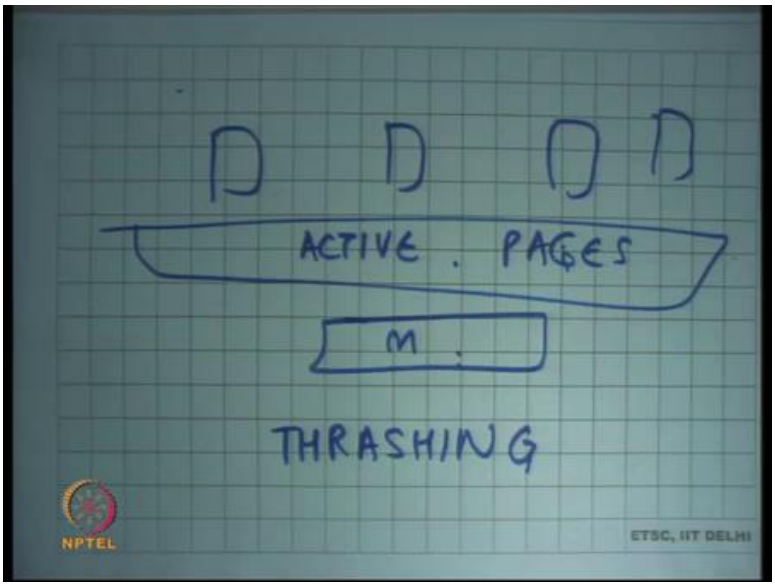
The other way to do it is basically have per process pools. So, you have per process quotas, let us say you know I have a quota for let us say this is my complete memory and I have some quota for process P1 and I have another quota for process P2. And then I basically if process P2 is taking lots of page faults then slowly I will move this barrier in one direction or another.

So, if I find out that P2 is taking lots of page faults and P1 is under utilizing its memory, then slowly maybe on the order of minutes I will basically figure out and not try to move this quota allocation one direction or the other. The important note there is that the movement of this allocation boundary is much slower than the boundary, than the actual page replacement rate, right. So, that gives you both some level of fearness and yet not too much inefficiency, right. So, it gives you fairness because P2 gets, P1 gets isolated from P2 to some extent.

At the same time if the operating system figures out that P2 actually needs a lot of memory and P1 does not need that much memory then there is some adjustment at a time level granularity to make to account for that, alright, ok, good. So, this is you know so we have discussed demand paging and we have said that you know pages are brought as in when they are required and this page replacement algorithm going on.

And in general if a process takes a page fault it has to wait for a disk request and while the disk is actually serving this process you will probably run another process and so you know the process that the cost of page faults actually gets hidden and the page faults in some sense become free, right. But, let us see what happens if it memory gets overcommitted, right. So, let us say the sum of all the memory that is required or all the active memory that required all these processes is greater than the physical memory that you actually have, right.

(Refer Slide Time: 25:07)



So, let us say you know there is a processes running, and you have physical memory, but you know the totals amount of memory is smaller than the amount of physical memory that you have. This is greater than this. So, this is let us say the active set, active pages of all these processes and this is the physical memory, and this is greater, alright.

So, what is likely to happen is processes are likely to take lots of page faults. So, let us say this process takes a page fault it is likely that it will go to the disk and in and it will replace one of these pages and these pages, and so one of the pages that are actually resident has going to be replaced and because all these pages are active most likely you will replace an active page.

And so, because you replace an active page very soon you will get another page fault because of that replacement, and so eventually what will happen is all these processes will just keep taking page faults, right. So, one-page process takes page faults and the other processes page, that process takes a page faults it replaces this processes page or its own pages and so you are basically spending a lot of time on page faults.

And so, what is happening at this point is that the system is spending a lot of time taking page faults and reading and writing pages from the disk and not actually executing useful instructions. The program was written to execute useful instructions, but these instructions are just page faulting and the and for no fault of theirs, it is actually the operating system that was trying to play tricks under the carpet and providing a very

fancy revision of a large address space with x performance latency of a of physical memory.

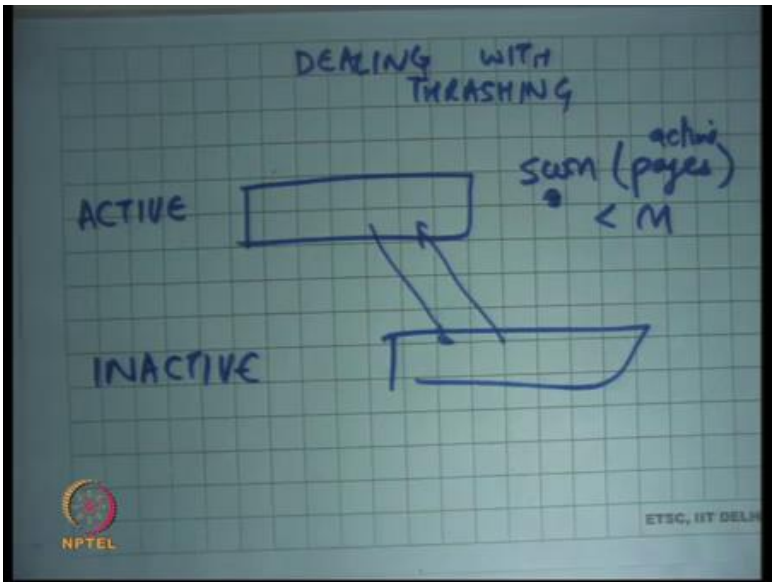
But what is what a at this point what has happened is you have an address space, alright, but the access latency is not a physical memory, the access latency now becomes of the disk, right. So, each memory access now becomes disk latency access at this point and so this situation is called thrashing, right. So, at this point the operating system is thrashing or your computer system is actually thrashing which means most of the time is actually being spent on reading and writing data to and from the disk and not actually executing useful instructions.

And you may have seen this in real life, in practice if you spawn too many processes you may have seen that is the sometimes the computer becomes really slow depending on you know if you are if your memory is not big enough to handle that many processes. And you know your let us say your flickering light for your hard disk access is continuously flickering which means the hard disk is continuously being read and written to, but the computer system as a whole seems to be very slow and really at that point your system is thrashing, right ok. So, and so typically what do you do to resolve the situation in practice?

Student: (Refer Time: 27:23).

You close some processes, right. And so, why do you expect that if you close some processes things become better? Because you close some processes then that you know they will reduce the set of active pages. So, some processes are not active, so their pages have been removed from the process and now the set of active pages fits in the physical memory and so at this point you can now start executing from physical at physical memory speed back again, alright, ok. So, an operating system automatically also can try at least to do the same thing. So, how does an operating system deal with thrashing?

(Refer Slide Time: 28:33)



So, let us say dealing with thrashing. Well, if just one process is causing all the thrashing then there is nothing the operating system can do about it. That one process is accessing so much memory or so many active pages that it cannot fit in the physical memory and there is nothing the operating system can do about it. The maximum it can do is you know schedule that process less often, so that other processes are not affected or maybe even kill that process, right. So, those are the only two options just one processes causing all the problem.

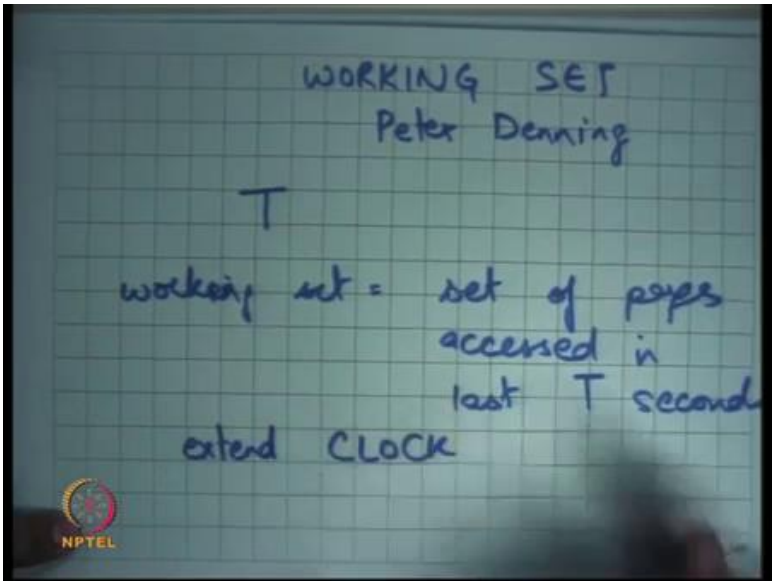
The more common case is that the sum total of all the running processes is causing all the problems, and so in that case the operating system can do something like that. It can figure out which processes using how much memory and then schedule the processes in groups. So, have two groups, one is active processes and the other inactive processes. And so at a time you will have only some set of processes in the active processes and make sure that the sum of all the memory active memory usage of those processes is smaller or comfortably smaller than the size of the physical memory, right.

So, basically you know have set of active processes and have a set of inactive processes and keep moving processes from active to inactive, right. And make sure that sum of sum of pages active pages inside of the active processes is less than physical memory or comfortably less than physical memory.

The scheduler only schedules the active processes at a time. The inactive processes remain in the sleeping or suspended state or even in the ready state they are not actually brought up and started to run. So, it is a scheduler that is playing you know playing in consonance with your virtual memory subsystem to figure out you know which processes should be run at this point, so that you will the system does not start thrashing at this point ok.

So, the question is how do I figure out what is the size of the active pages of a process, right. So, notice I am not just saying that how much memory is allocated in the process that is not important. What is important is what is a set of active pages in a process which pages that have been being used by this process on a in an active way, right.

(Refer Slide Time: 30:59)



So, there is a way, so there is a term called a working set, used to describe the active set of a process and this was this term was or this method was proposed by Peter Denning, alright. And informally, the working set of a process is a collection of pages that have been that are actively being used by the process, alright, ok. And let us see how the working set is computed.

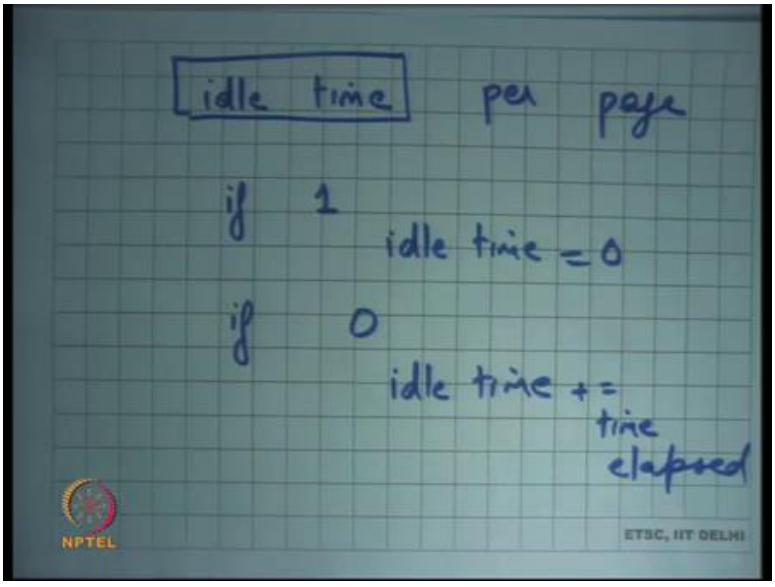
So, firstly, if you if you can compute the working set of every process then it is very easy, I basically you know make sure that the sum of working sets of the active processes is less than physical memory and the scheduler basically does this does this intelligently, ok. Also notice that the working set of processes may change over time. So, right now

the process is accessing let us say 5 MB of, has a working set of 5 MB, sometime later let us say a few minutes later it becomes 5 KB. So, I should figure that out I should and I should account for that in my scheduled algorithm, right.

So, now, let us first say see how a working set can be computed, alright. So, the working set is based on a parameter called T and working set is basically set of pages accessed in last T seconds, alright. So, if I define the working set as a set of pages that have been in the last T seconds, you know that is the one way of defining the set of active pages or process.

Notice that here again, I am using recency as a criteria to distinguish between active and non-active, right. Just like in LRU, I was using recency to distinguish between something that needs to be replaced or not. I am not using frequency at all and that is also primarily because workloads typically behave in a past is equal future manner and so recency is more accurate, guess then frequency, right. So, how do I compute the set of pages that have been accessed in the last T seconds? Well, one way to do this is basically extend the clock algorithm, alright.

(Refer Slide Time: 33:19)



With each page keep a field called idle time per page, ok. And in the clock algorithm, if you see a one then set idle time to 0 of that page, alright and if you see a 0 then increment the idle time with the time elapsed since the last time you saw it, ok. And so that way you can keep you keep track of in a proximate manner not completely precise,

but in an approximate manner what how long this page has been idle, right. And once you have an idle time per page your working set is basically all the pages whose idle time is.

Student: Less.

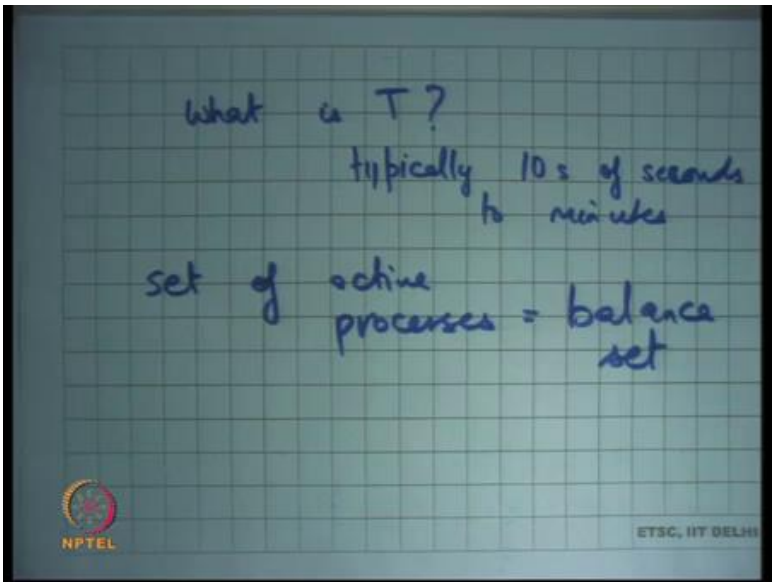
Is less than T, ok. So, you can compute the working set in an efficient way, and you can even use it. Yes, question.

Student: Sir, did I have to keep a record of the time elapsed for each of the page in each revolution?

So, keeping the time recording the time elapsed is you know you can have another field for example, idle time and last time it was set, right. And so you just have to make one computation to figure out you know how much time has elapsed, you know the current time, you know the last time you set it and you basically know how much time it is been set or you can use some approximation of this. In any case, recall that this movement of the clock hand is happening on the miss path, right and so this extra computation is not a big deal because you know it is dominated by the page fault handler and the disk access and all that, this extra computation is actually not too expensive.

In this competition of the working set no way have we increased the time or increase the overhead on the hit path that is the most important thing, right, ok. So, you know, so that is the working set and using the working set you can decide which processes to keep in the active set and that way you can prevent thrashing, alright. Some difficult questions for the working set idea. Well, firstly, how long how big should T be?

(Refer Slide Time: 35:31)



What is T? Right. Well, typically 10s of seconds to minutes, right. So, you use a relatively coarse grained value of T to figure out what are the set of active pages, you being a little conservative in the sense and set of active pages you being a little conservative to make sure that you your system is actually completely divide of trashing. So, you have completely preventing trashing by having a large parameter for T, alright

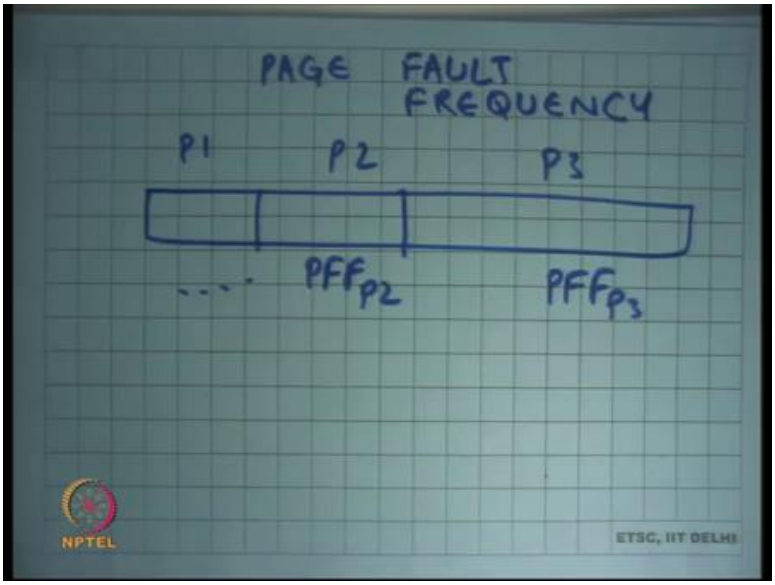
Also, you need to change handle changes in your working set. So, as the working set is changing you need to figure that out on the on dynamically and based on that change the set of active processes. Also, know this set of active processes is called the balance set balance set and so the operating system makes sure that the sum total of the working sets of the balance set of processes is less than the precise of physical memory, it is just a term called balance set, alright.

And, also you know when you are doing this kind of thing you have to worry about what if processes share memory. So, let us say there are two processes that share memory then you need to worry about that as well, right. For example, one thing to do maybe that these processes are considered as one unit, so they either together go into the balance set or they together go out of the balance set, right.

Because otherwise you know you have to you have to account for, otherwise its more inefficient because shared pages are likely to be; so, it is better to account for shared pages you may want to do that if this number of shared page is very large if the number

of shared page is not very large you may want to have other ways to figure out you know what the working sets are ok, alright.

(Refer Slide Time: 37:47)



Another way of preventing trashing is using the page fault frequency. So, one way of preventing thrashing was using the working set approach another way of preventing thrashing is look at the page fault frequency of every process. Here, basically you know you have let us say this is your physical memory you have per process allocations of physical memory, alright and you basically maintain the page fault frequency of each process.

So, let us say this is the page fault frequency of process P3 and PFF of P2 and so on, right. And if you figure out that one process has an extremely high page fault frequency and another process has an extremely low page fault frequency then you can adjust these boundaries based on that, right. So, as opposed to the working set which has which also has a tunable parameter called T, page fault frequency will also have some tunable parameters saying you know what is the threshold where you are going to move things and the.

Also, if you figure out that the sum of all the memory pools, does not fit in your main memory then you can deactivate some processes ok, alright, ok. So, this is an alternate idea to working sets to be able to manage your virtual memory and to prevent thrashing.

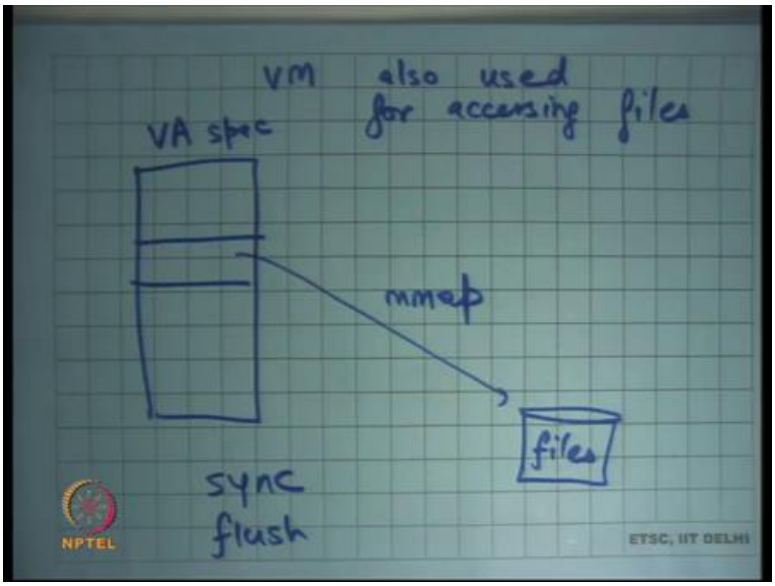
In general, thrashing was more of a problem. So, this thrashing was a big problem in the early days of shared computers. So, you know let us say in the late 70s or 80s when we are used to have large shared computers like mainframes and lots of people accessing these mainframes, then at that point you know you had to worry about thrashing and these kind of things to basically make sure that all the users get some level of fearness and also the maximum amount of performance that you can get out of a shared computer.

With the advent of personal computers this became less of a problem. Number 1, you could you know memory becomes became cheaper and so if you see thrashing the best way one way to do it is basically you will just add more memory and so you get rid of much of this much of these problems and the does not need to worry about it as much as it had to worry about it in shared computers. So, memory has become cheaper.

The other thing is the user has more control over a system. So, the user himself can manually by hand just switch off processes, but on a shared computer no nobody wants to stop his process because of the other process, right, but when you have a personal computer you know the user is doing more of this management itself. So, in other words basically you know in a personal computer the balance set it can be managed by hand by the user, alright, ok, good.

So, that is that is all I have for the virtual memory subsystem. And, next time I will start discussing about storage devices and what are the different kinds of storage devices we have. And, then and the kind of data structures we use on storage devices to implement abstractions like file systems, ok.

(Refer Slide Time: 41:09)



One last thing I would like to say is that the virtual memory subsystem is also used for accessing files in modern in modern operating systems, right. So, how? Well, in your in a virtual address space you can; so, let us say this is the VA space, you can map regions of a VA space to point to files, right. And so, for example, on UNIX you can do this using the m map system call, right.

And once you do that the program the programmer can now just treat this region as any other memory and do pointer dereferences on it, right. So, read and write can be done directly here and under the covers the operating system is managing reads and writes to the disk, right. Notice that this is an alternate way of accessing files from the one that we have discussed so far. So far, we are discussed ways of open, read, write, close, right.

Here I have mapped the file into my address space and then just use pointers or just use my program how just consider those start space like any other memory region, right. For example, I could manage a tree data structure in this space, right. And so, my program does not need to worry about that it is a disk that is that is that is getting written to at the back end.

To make it efficient of course, the operating system is caching pages in physical memory, right. So, it is not like every pointer dereferences going all the way to the disk, they are being handled by page faults and pages are being brought into physical memory so that most of your operations are fast, but they are also flushing demons and page

replacement demons are running at the background to basically make sure that the file contains are updated ok.

The only the only difference between using this virtual memory subsystem for other space and with using files is that for files the user always also needs some guarantees. So, for example, if I write something to this address space that has been m mapped, what is my how do I know whether this has been written to disk or not, right. And I may need some guarantees over that. In case of virtual memory, I did not care about any guarantee I did not care if the thing was written to swap or not because when the system reboots the swap is anyways cleared, the swap spaces anyways cleared, but for the file I need some guarantees that it has to be written to the to the disk.

So, one way to do that is basically using you know providing system calls like you know flush or sync. So, you basically if the; in general when you write there is no guarantee that the contents will be reflected on disk, but if you want to have a guarantee then you can use an explicit system call to basically say that the contents should be flushed to disk and they will be flushed to disk in some dashed way, so that it becomes efficient also, right.

What are the advantages of doing file accesses using m map over doing file accesses using read or write? Well, one advantages I do not pay the cost of the system call. If I need to access the file, I just make a pointer dereference and that is it, right. I did not have to do a system call which involves a trap, some kernel code getting to run and so on. In this case, I can just dereference the pointer and assuming that the page is already mapped in my virtual address space, the operation will be logically done, not physically done, physically done later, but logically it is done.

Student: Sir. Sir, but if you just look it into, if we just read the file into a character array and then we perform all the computation on that array and later write it back to the disk. So, would it be the same thing?

If we read the file into a character array and write to the character array and then write it all back to the disk won't it be the same thing? Yes, I mean this is a way of the operating system providing you that functionality under the covers that you know there you have to do it explicitly. So, you have to read the whole file into your character array you. So, for example, if there are multiple processes who want to access the same file, so there is,

there needs to be some way of the file system is also giving you some sharing between multiple processes, but if you read into an a character array then there is no sharing.

For example, you know if there is a there is one file that is opened by multiple processes and so one process write to that file the other process can immediately read from that file because it is a shared address space, it is a shared file system, but file is the same. The obstruction is that it is being feared by multiple processes, but if you read a new character array then there is no sharing for example, right.

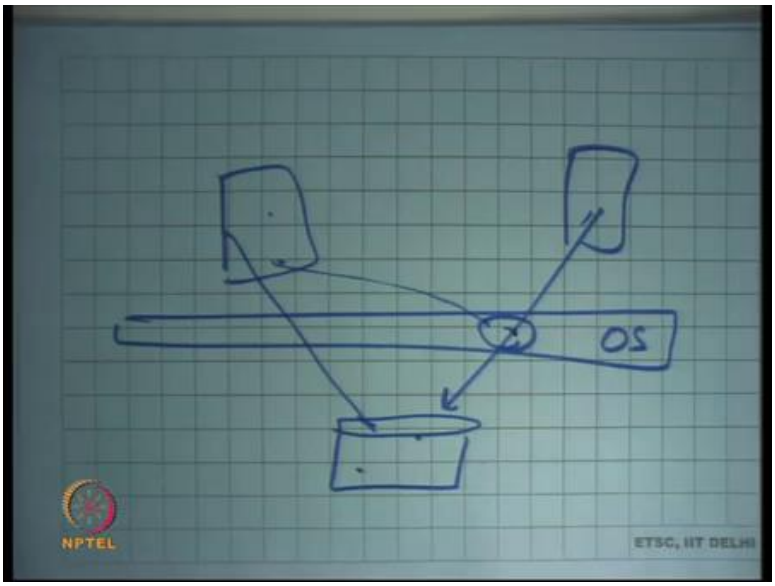
Also you know reading from a character array to into a character array requires a programmer, effort and programmer, understanding of what it means to you know when to write it back and optimization, it is better to rely on the operating system to do these things for you, right in a memory mapped environment.

So, you are saving the cost of a system call number one, but is that that important? Well, you know firstly, its if you are going to take a disk access then yes this cost of system call is not important, but if you are going to be if you were going to get served from the buffer cache then you know the cost of system saving the cost of system call is good.

The other thing is you know recently we have seen lots of different types of storage devices apart from magnetic disks that we have talked about there are things like flash memory and other sort of persistent memory, technologies, that are giving you very fast sort of persistent memory. They have their own problems, or you know quirks, but, but in any case, they you know they are providing alternate ways of providing secondary storage.

And, so accessing that if the latency of accessing that kind of secondary storage is very small then the cost of actually, doing the system call becomes large. So, let us say let us say hypothetically speaking that this was as fast as memory, right. In that case, using the read write system call to access the disk is very wasteful. You would want to access it with the same interface as you access memory, which is just pointed dereferences, right and so m map is a nice, elegant way of being able to do that.

(Refer Slide Time: 47:45)



So, we were going to discuss this in the file system discussion, but let us say this is the disk, the disk is the operating system sits on top of the disk, right. So, no process can touch the disk directly. So, if there is a process, if there are two processes that are accessing the disk the operating system is maintaining synchronization between them firstly. And so, if this one changes something it changes both in the buffer cache and in the disk, and so later if somebody reads read the latest value from the disk, ok.

So, the obstruction is that of a shared file. The operating system is responsible for doing synchronization which means that you know there should not be any race conditions on accessing access of the buffer cache, there should not be any race conditions on access to the disk, but at the same time the abstraction is that of a common file its not you know the obstruction is not of two independent files, ok.

Let us stop here. And, start our discussion on file systems next lecture.