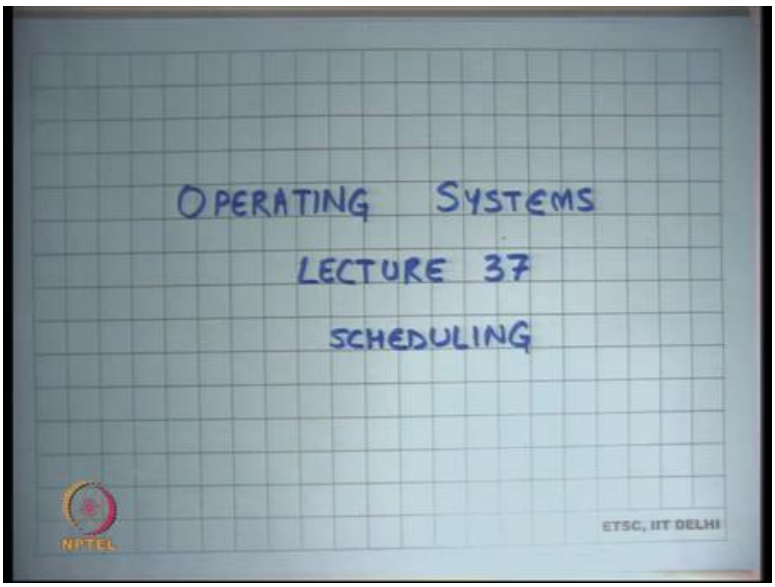
**Operating Systems**
**Prof. Sorav Bansal**
**Department of Computer Science and Engineering**
**Indian Institute of Technology, Delhi**

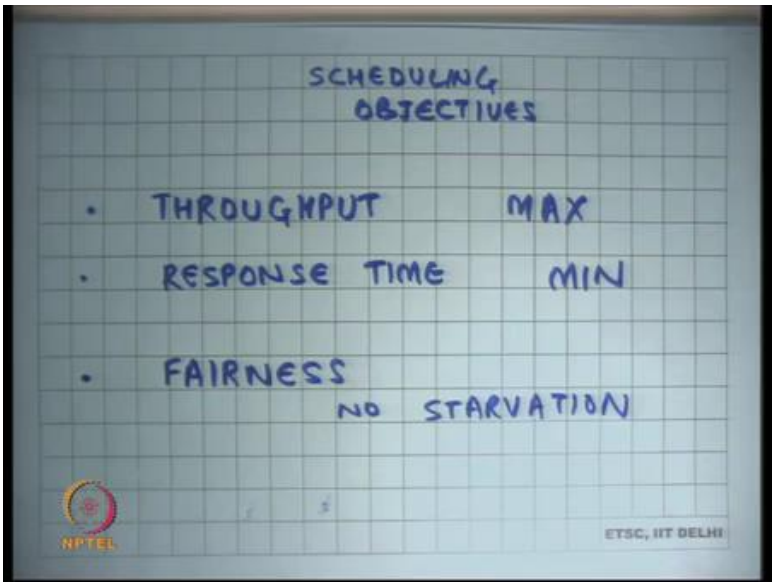
**Lecture – 37**
**Scheduling Policies**

(Refer Slide Time: 00:25)




Welcome to operating systems lecture-37 and today we going to talk about scheduling right. So, one of the central themes of an operating system is scheduling basically we have seen that an operating system runs many processes, handles many devices and maps processes to devices, and one of the central things that an operating system is really doing is scheduling right.

(Refer Slide Time: 00:47)




So, let us first understand what are the objectives of a scheduler? So, let us say scheduling objectives all right. So, let us say throughput, maximization, by this I mean that I want to act do the maximum possible work in the minimum possible time. So, there is a lot of work to be done, and I want to make sure that I finish as many jobs as possible in the minimum amount of time.

And so, I have these many resources and I want to judiciously use all these resources to do everything I can right. The other thing is minimize response time. This means that if as something needs to respond to something so if there is an into input output operation, then that response should be as fast as possible. For example, if the user presses a key, then he should see the key character getting printed on the screen as soon as possible right.

There should be some metric which says this is the response time, the response time is the time it took from between the pressing of the key and the display of the character on the monitor or you know the arrival of a packet, the arrival of a request on the over the network and the response or the reply over the network right.

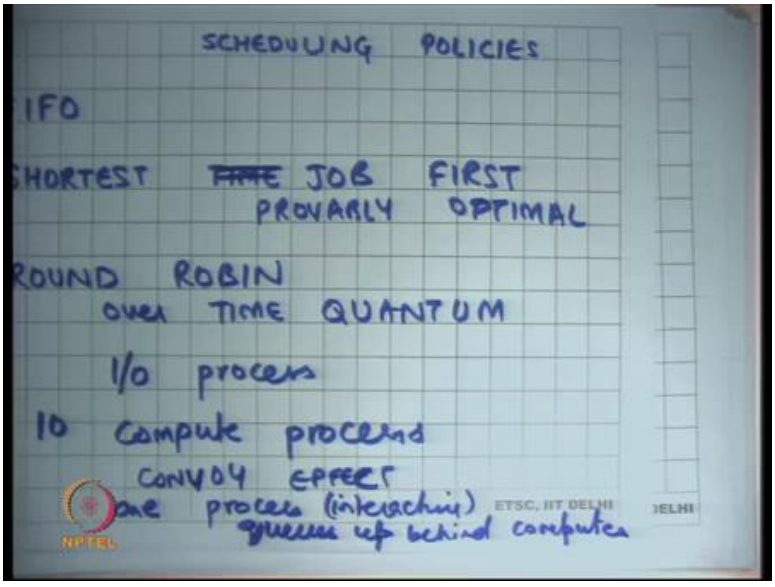
So, the user pressing a key and the character appearing on a monitor is a very; is a very easy constraint really, because humans perception time is very large you know. So, humans cannot perceive more than at more less than 1 millisecond granularity and computers are already as fast as nanoseconds, so that is not that big a deal. But if I talk

about network response times that becomes more of a more of an issue right, and it is not just network other IO devices also right.

And you are going to see that how you know these two requirements can conflict with each other. So, if you have maximum response, if you need maximum response time, it does not you know it may actually come at the expense of reduce throughput. And then you want some level of fairness right. So, in doing all this, you make you want to make sure that if there are many jobs that are running, there is some amount of fairness in the system which it should not be that if a job is giving you a lot of throughput, you just keep running that particular job and did not run any other job, so there should be some level of fairness.

In particular there should not be any starvation right. So, there should be no starvation. By starvation I mean there is some job that does not get to run at all or there is no guarantee on how frequently it gets to run right, so it can keep waiting for an unbounded amount of time that will be called starvation and this you know a scheduler should actually never allow something like starvation right.

(Refer Slide Time: 03:45)




So, those are some of our objectives then let us look at some types of scheduling policies all right. So, the simplest policy is first in first out or first come first out right that is your typical scheduling policy, you queue up the jobs in arrival order and you keep doing the jobs in the order that they arrived to you right. This is ok, but you know the process lens

can be actually very large you know, some process lens can be very small other process lens can be very large.

For example, if you run a compute bound process and the resource at your scheduling is the CPU, then the process compute bound process could probably take you know minutes and you do not you cannot afford to make other jobs wait for minutes right, so there is a there is an obvious problem with that.

Well, in the same vein there is something called shortest time completion first or shortest job first let us say. Here again instead of serving the jobs in the arrival order, you basically say here are all my jobs that I have come let us say I wait for some time and I collect some jobs, and I say what is the shortest job in this buffer and I picked that and as you do that and then I pick the next shortest job and so on all right.

So, with if I do that then you know arguably I will minimize the amount of response time for the you know the total average response time per job if I do that so this is you know provably optimal in terms of response time average response time per job, but once again it's too unfair to be actually practical right.

It is personally it you know you know the job even the shortest jobs could be very long in that which means that other people can have to wait. And secondly, if you know there is a lot of short jobs that are arriving all the time, then the lock job can just keep starving forever right, so these are all this is also not really very practical.

For something like a CPU, you could do something like round robin and define some kind of a time quantum right. So, recall that we said that there is some amount of there is a scheduling quantum and we define the scheduling quantum based on some matrix, one of the scheduling quantum that we sort of looked at was 10 milliseconds or 100 milliseconds.

How was the scheduling quantum decided? Firstly, the cost of doing a context switch should be very small with respect to the scheduling quantum, so that the overhead of scheduling is small number 1. And number 2, the scheduling quantum should be small enough such that so it cannot be too large if let us say the scheduling quantum was two seconds, then you know if I press a key and there was some job that is running currently,

then I will have to wait on average one second before I actually see that key to be displayed right.

So, whatever are my input output devices the response time should be proportional or roughly proportional to the scheduling quantum that you choose, right. So, if you are you know if you are worrying about human users, then human users need you know millisecond response time or 10 millisecond response times and so scheduling quantum should have some proportionality to the amount of response times that you are looking for or some relation with the amount of response times we are looking for.

As an interesting piece of fact, you know even in the 1970s the scheduling quantum used to be 10 milliseconds and even in today 2014 the scheduling quantum are still 10 milliseconds. Even though the computers have become really faster right, and then at that time the computers were 60 mega Hertz, today the computers are 2 giga Hertz so you know there is a 100x speed up roughly speaking. And so, what is so but the scheduling quantum have not changed, what is the reason.

Student: You are saying human response time.

Humans human response times have not changed right so good, all right all right, so that is round robin. But round robin completely ignores the type of the job, so you basically give all jobs completely equal you know it is completely fair and that may not be the best thing. For example, let us say there was an IO job IO process and there were as a compute process and let us say there were 10 compute processes all right.

So, now you are doing round robin between these compute processes and these IO and this one IO process and what you care about is not just your throughput, but also your response time right. Compute processes mean there is some you know long running computation that is going on. So, there is no IO in the compute process and so there is nothing like there is no notion of response time in the compute bound process, but for the IO bound process there is something called the response time.

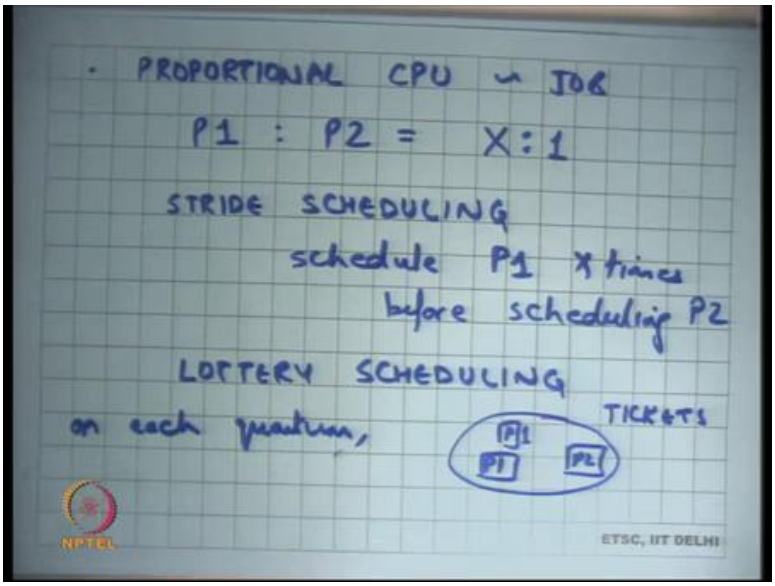
And so what you would want is that whenever the IO bound process is ready, you will want to run it first respective of how many compute bound processes are running at that point right, because if I press a key, let us say the IO bound process is your editor and it is you know it is currently waiting for a keyboard key press. And I press the key, now if I

just do pure round robin scheduling, then this IO bound processes process will have, may have to wait for 10 compute bound processes to finish or 10 scheduling quantum to elapse before I get to run right, so this is also called the convoy effect.

There one process interactive process queues up behind non-interactive processes or compute processes this is in computes, but compute processes, so that is not a good thing. And we will see how one can solve that. So, if I do a pure round robin scheduling, I can actually have a very poor response time. So, let us say I have a system that is running lots of compute bound jobs, let us say it is running simulations at the background that should not mean that if I press a key or if I login into the system, you know my the time it takes for the next prompt to show is really high right.

Even if there are lots of jobs running at the background, my system should appear interactive and there is the cost I pay for making the system interactive is really small right. So, it should both be doing the compute bound jobs and yet give me a very high level of interactivity and that is what I would like to have ok.

(Refer Slide Time: 10:32)




Also, round robin is just to fear you would probably want some level of proportional CPU depending on the job right. So, for example, some processes are more you know are more important than other processes, system level processes are more important than user level processes.

Let us say there is a process that is going to handle the disk, it is going to flush the disk, or it is responsible for doing something on which means lots of different user process, user level processes rely. And you want to say that the system level process should get more CPU than the user levels process on average, right.

So, you could say something like you know let us say process P1 is to process P2 is the you know the proportion should be X is to 1. And then you can just modify this round robin to basically do this X is to 1, proportional share scheduling. How can you do that? Well, one way to do that is what is called stride scheduling, where you schedule P1 x times before scheduling P2 all right, so that would not just a straightforward extension to round robin, you just add some proportionality to it and you basically say I am going to schedule this x times before I actually schedule.

The other way to do that is what is called lottery scheduling ok. And lottery scheduling basically is a probabilistic algorithm where you say that you know you have a pool of tickets, lottery tickets, and each ticket has a label which is let us say P1, P2, P1 and so on. And on each quantum, the see the OS draws one ticket at random from this pool.

And depending on whose ticket it is that is the CPU that gets to run right. So, it is a randomized algorithm it is a randomized version of straight scheduling, where you basically just pick up one random ticket from the pool, and just say that whoever comes is basically the one that gets to run. So, if I want to implement proportional shear scheduling, all I need to do is have x tickets for P1 and for everyone ticket of P2 and so on ok.

Student: Sir, we got how will larger scheduling high some of the corner (Refer Time: 13:09).

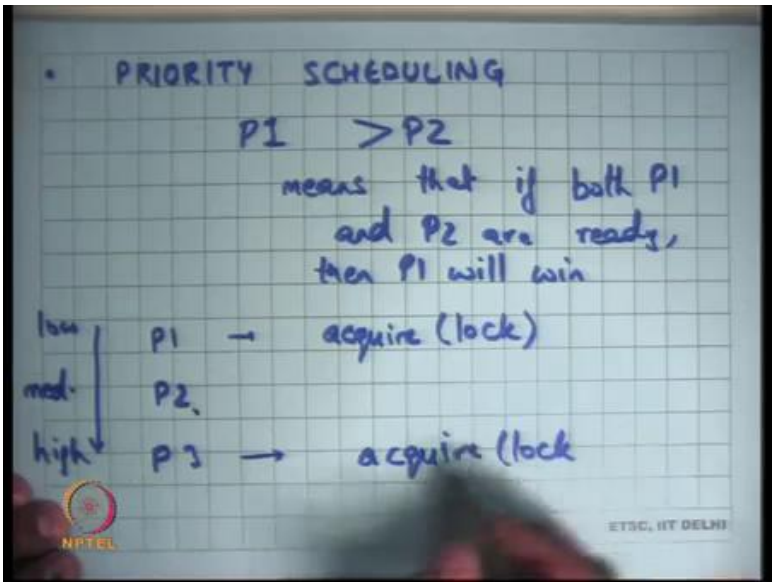
So.

Student: (Refer Time: 13:11) that its chance is not going to have.

Right, right, right. So, I am not solving the convoy effect right now, I am just talking about you know proportional share scheduling, and we are going to talk about what convoy effect very soon ok. The nice thing about these scheduling algorithms are they are tolerant to dynamic admission of processes and dynamic exit or the processes.

So, processes are coming in and going out, and that is completely right. So, if a process comes in for lottery for some for something like lottery scheduling, I will just add that many tickets to my pool, and you know it is a completely local express operation, and now you know it becomes proportional in that sense. A process exits, I remove the tickets for that particular process and that is it. Similarly, straight scheduling and round robin and etcetera ok.

(Refer Slide Time: 13:59)




But that is not, no, that does not solve our convoy effect, and then and so what we really need is some, some sort of priority scheduling all right, where the idea is that P1 has higher priority than P2 implies means that if both P1 and P2 are ready, then P1 will win all right.

So, in the proportional share, it was just an average thing you know basically a P1 will get x times more CPU than P2, but priority scheduling is a strict priority basically say is if P1 and P2 both are ready, then I will always give the CPU to P1 before I give the CPU to P2 right, so that is a strict priority scheduling.

So, some proper jobs have a higher priority than other jobs. Some examples of these are let us say interrupts or device if a if a device needs attention, I want to give it attention as soon as possible. So, interrupts you know we already know that interrupts have higher priority than other jobs.

So, let us say the some there is some job running, but somebody request an interrupt, the job gets preempted and the interrupts gets raised that is basically like that is basically saying that interrupts have higher priority or the device if the device needs attention the device will have higher priority than anything else all right. Or you could say you could solve the convoy effect by saying IO job bound has higher priority than compute bound job.

So, if there if the IO bound job is actually ready to run on the CPU which means there is some input that has been received from the keyboard. Now, it needs to be done on the CPU to actually do something, compute something and maybe display something on the monitor, then you know it should get higher priority.

So, even if there are hundred compute bound jobs, if there is IO bound job that is running, then it will get higher priority over the compute bound jobs all right. So, you could do something like that this.

Student: Sir, IO jobs should be a special case of device interrupts (Refer Time: 16:05).

Would not IO jobs be a special case of device interrupts, well, no not exactly, because a device interrupt may just mean that the device needs attention. And the device needs attention may mean that I just copy that character from there to my kernel buffer. And now you know I know that this character is meant for this particular process, so this process has become ready. And now that process becomes you know it is now the regular scheduling algorithm will pick up that process on the next scheduling quantum.

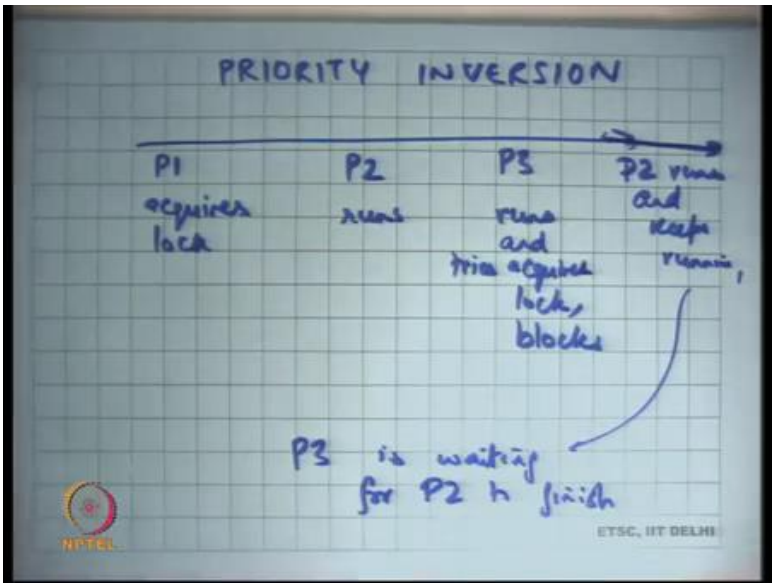
So, though both are examples of priority scheduling, but you know in different context. In one case, you are dealing with the device; in the other case you are dealing with the process that is handling that device all right. So, that is priority scheduling. You may also want priority scheduling for things like you know so some things are more much more important than other things, for example, real time operating systems.

So, you know if the if there is an operating system in the car, then somebody presses the break, then that should have the highest priority among all the other things like playing the music player etcetera right, so that is another thing example of a priority scheduler.

Now, you know when you do priority scheduling, you basically want strict priority strict priority is priority orders between these different jobs, but you know they are issues with that. So, let us say there were three processes P1, P2 and P3, and let us say the priority order had let us low priority to high priority, and let us say P2 is medium priority.

Now, let us say; let us say P1 acquires a lock or acquires any resource I am using with the lock as an example of a resource it requires a lock. And now P2 comes in and P2 gets to run. Then P3 comes in and P3 also tries to acquire the same lock or same resource that is it right.

(Refer Slide Time: 18:06)




So, the timeline is so this is my timeline. So, P1 acquires resource or lock; P2 runs P3 runs, so P3 is higher priority, so P3 comes in P3 starts running. And P3 it runs and tries to acquire the lock all right. But now because P the lock is already held by P1, P3 will have to block let us say and let us say it is a blocking lock, and so P3 blocks and tries to acquire a lock and blocks, and so P2 keeps running and keeps running.

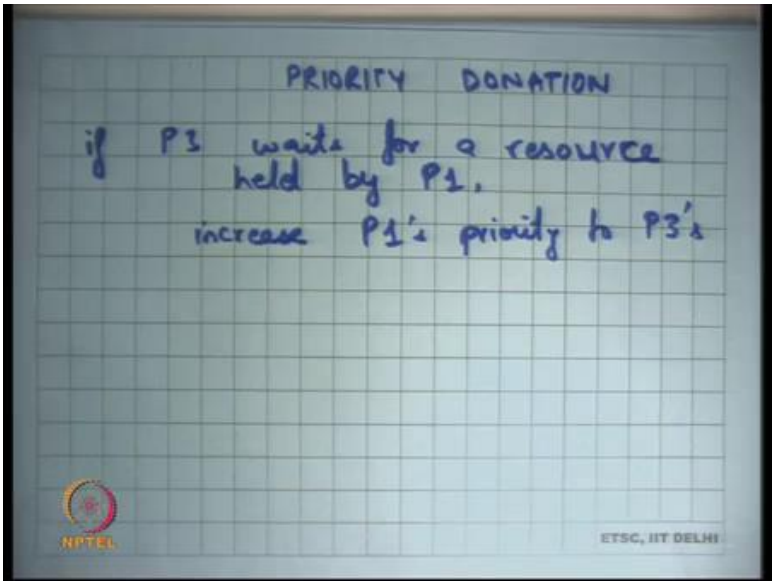
At this stage you have a situation, where P3 is waiting for P2 to finish right. And this violates the priority order. You always wanted that if P3 is ready to run, then P3 should get to run, but because P3 there is a dependency between P3 and the lower priority process, it is possible that P3 actually waits for a process that is lower priority then it, and this thing is called priority inversion right.

This problem is you know so this problem is basically once again the problem is that a higher priority process is waiting for a resource that is being held by a lower priority process, and but because the lower priority process never gets to run because there is a medium priority process that is already in the system.

And so, the medium priority process keeps getting to run, and so effectively what is happening is that the higher priority process is waiting for the medium priority process to finish, whereas that should not have happened. But the ideal thing to do here, so this is the this is what is called priority inversion. So, priorities have been inverted.

And so, the wait the ideal thing here would have been that the system if the system could figure out that the higher priority process is waiting for a resource that is being that is currently held by the lower priority process. And if that is the case then the priority of the lower priority process is bumped up to the priority of the waiting process, so that you know it gets higher priority than P2.

(Refer Slide Time: 20:29)



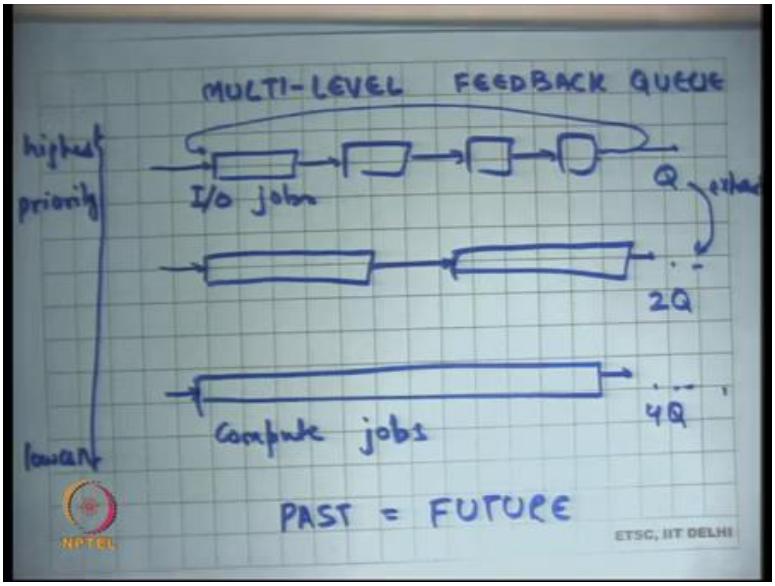

So, this problem can be solved by what is called priority donation. It says if P3 waits for a resource held by P1, increase P1's priority to P3's right. And so, you could solve this priority inversion problem by using priority donation. But but that is provided that this lock was something that was visible to the scheduler right.

So, I am assuming here that the scheduler or the operating system could figure out that P1 acquired a lock, and now P3 is waiting for a lock that is acquired by P1 that is only possible if the lock was a blocking lock or it was a lock that involved OS activity, for example, the lock involved a system call.

But if the OS has no idea about this dependency of between P3 and P1, then what is the worst thing that can happen that P1 holds a lock, P3 waits for P1 to complete, and P3 keeps waiting you know P3 let us say the lock was a spin lock, then P3 gets to run, and P3 just keeps running forever. And the lower priority low thread which is actually holding the lock does not get to run right.

So, that is the. So, you strict priorities can actually also result in deadlocks if the scheduler has no idea about the can as a fake to the dependencies or order resource dependencies between the different processes right.

(Refer Slide Time: 22:23)




So, using this priority idea what is used in operating systems is of what is popularly used in operating systems is what is called a multilevel feedback queue. So, what is this, you basically maintain multiple priority levels. So, this is highest priority, and let us say this is lowest priority.

And within each priority level you maintain the jobs in a round robin queue all right and so on. So, these are lower priority threads ok. You run the lower priority thread only if

so, you obey so by priority I mean you know the strict priority that we have discussed. So, you run a lower priority thread only if there is no higher priority thread available. So, you will run this only if all these other processes or jobs are not ready ok.

And so how and so basically what you would want to do is that all the IO jobs are at the highest priority, and all the compute jobs come at the lowest priority ok. So, basically what the system does is it organizes all these jobs into IO bound jobs and compute bound jobs, it gives highest priority to the IO bound jobs and serves all the jobs within the same priority in round robin order.

And then all the compute bound jobs are given lower priority than the IO bound jobs. And the and so if there is an IO bound job that is ready, then that will get served before there is a compute bound job that gets run. But then how does the system know what is an IO bound job and what is a compute bound job, number of?

Student: IOCs calls.

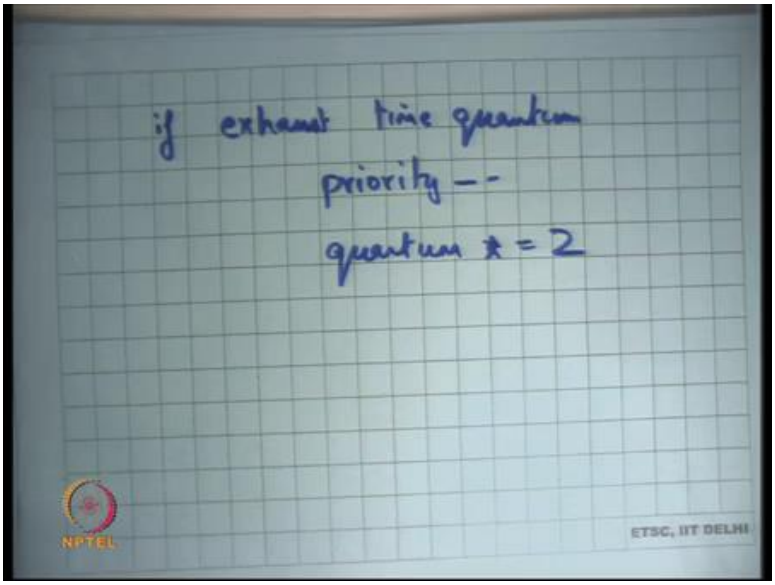
Number of IOCs calls ok. The number of IOCs calls, well, yes sort of. So, basically you just you know like as that everything else in the OS you will just use the past to approximate the future all right. So, basically say what has been the behavior of this application previously. And by IO, I basically mean something that actually blocks you know for whatever reason either IO or lock or something. And so, what you do is you start a job at the highest priority. And at each priority, you give lesser time quantum.

So, let us say there is a time quantum q here then you will give 2 Q here, and let us say 4 Q here. And you give it a quantum queue to run. If it exhausts the time quantum which means it actually does not block within the time quantum that you gave it, let us say you gave it 10 milliseconds to run and it did not added any exhausted the 10 milliseconds without blocking. If it is an IO bound job it is like it, it is likely that it will block before the 10 milliseconds require expire right. For example, it will go and say I want to read the next character from the keyboard.

So, if it is an IO bound job, it will not exhaust it is a time quantum. If it exhausts the time quantum it basically means that this jobs average length is longer than the time quantum that is given to it. And so, what you do is if you exhaust, then you move it to the lower priority queue right.

So, you start it from the highest priority queue and give it at one time quantum queue, if the job expires at time quantum, then you move it to a lower priority and also increase it is time quantum right. If it does not exhaust a time quantum which means it blocks before it wanted it could expire time quantum you leave it where it was ok. So, this basically ensures.

(Refer Slide Time: 26:29)




And, you basically as you move the so the logic is if exhaust time quantum, then priority--, and quantum is multiplied by 2 right. So, you also multiply the quantum by 2 to make sure that you know the scheduling overhead is even less. So, it basically says that it seems like this job is there to you know take large chunks of CPU.

And so why to keep interrupting it if it needs a large chunk of CPU, you not just decrease it is priority, but also give it higher amount of CPU. So, when it gets scheduled, it will actually get a larger chunk of CPU whenever it gets scheduled right.

And you maintain some levels to do this. So, you are, also it is not an unbounded list. So, you have a maximum bound on how much time quantum you going to have for a job ok. So, so question is that if there is a compute bound process that is running, and there was an interrupt that occurred in the middle of this compute bound process, then you know the interrupt will gets served, and want that serving of the interrupt caused problems with the calculation of this quantum is that your question?

Student: No sir. If the (Refer Time: 27:42) occurring between and then some other process which was using a network being scheduled in it is process.

No. So, hold on. An interrupt will not, so and a network interrupt will not cause a scheduling behavior all right. Let us just; let us just work in this model that device interrupts do not cause scheduling changes right. Device interrupts are meant for device attentions. For example, a device interrupt has occurred let us say a network card created a interrupt, all it means is that the device needs some attention maybe it wants that I should copy some buffer from there to here, and then I will go resume the process that was already running that is all ok.

And the only where time that you do a scheduling switch is either when the process actually voluntarily yields the CPU or if there was a time and interrupt which caused the preemptive schedule switch ok. So, but what is going to do is that any process that is very IO bound. So, let us take an example let us say you are running you know gcc in your on your system and you are running an editor like vi.

So, vi is a compute bound pro is an IO bound process and gcc is a compute bound process. And so, let us say gc these are both long relatively long running processes. So, eventually what will happen is that gcc will come will have a lower priority, but a larger time quantum, and vi will have a higher priority and a smaller time quantum and that is exactly what you wanted right. vi does not need that much CPU, but it needs very highest attention spans; gcc needs lots of t CPU, it does not care about response times all right.

So, with this with that you can basically have some level of a you know you can have good levels of throughput and yet have acceptable levels of a response times. But let us say you know if I have something like this, then is not it possible that all the lower priority threads keep starving.

So, let us say there are lots of IO bound jobs that are waiting in the queue, then the compute bound job will actually never get to run right that is a so this is an this is a nice idea and principle, but you know one needs to do some modifications to make sure that there is no starvation possible. Anytime you have a strict priority system, starvation is always possible for the lower priorities processes. So, the way to deal with this is that

you basically if there is some process you basically in keep increasing the priority of a process at some rate with time.

So, if a process has not been scheduled so for example, one way to implement this is on every schedule on every scheduling quantum if you do not use the process, implicitly you increase the probability of that process by some constant number what. So, all the processes that did not get chosen on that quantum, their process, their priority will get increased. And so eventually a process will get reach the highest priority and will get to run all right, so that way you deal with starvation.

Student: (Refer Time: 30:28) will be decreased right.

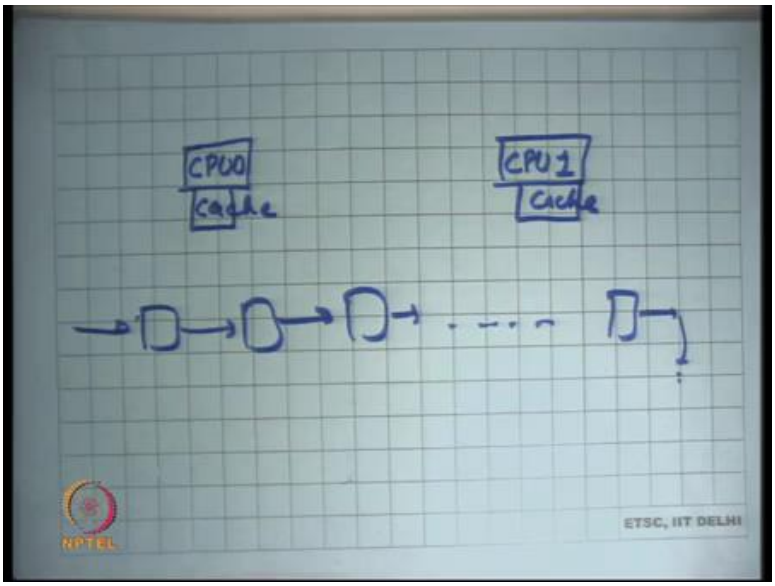
But it is quantum will be decreased, yes, all right ok. Also if you know if you have decided that a job is compute bound, so let us say it has moved down, so it has you know it is priority has been decreased, it is quantum has been increased, but you know let us say there is a process that was compute bound and then it suddenly became IO bound.

So, you know we are using the this assumption that past is equal to future, but past is not always equal to future. And so, the system should be able to adapt to such changes, and so there should be a way for the process to move up. So, we have discussed a way to for the process to move down in the priority hierarchy, there should always also be a way for the process to move up.

For example, you could say for thread blocks without exhausting its time quantum then you know you move it up with policy that you want to use right. And you can choose different policies. So, in general when you have chosen, we have chosen these policies you will favor recency over frequency. So, you will look at the most recent behavior to decide whether it is IO bound or compute bound.

You know you would not so just like a caching, caching placement algorithms in general you know you know recency is a much better metric to predict the future than frequency. So, we do not look at you know what is its behavior over the history of its execution, we just look at its behavior over the last time quantum or last five for time quantum something and that is good enough all right. So, that is you know a very simple notion of scheduling.

(Refer Slide Time: 32:00)




But actually scheduling is a relatively complex topic and you know subject of much research over so many years and still not completely mastered. So, let us see let us say these are two CPUs and there are lots of jobs that need to run. So, the question is how should one choose which job to run on which CPU and when? So, we have answered when to run which job, so you know there is some idea we have there.

We are going to do it in round robin order, we are going to have some priorities, and we are going to give higher priority to IO bound jobs and we are going to give lower priority to lower bound jobs. But is does not matter which CPU I schedule them on? Well, actually it turns out it matters because CPUs have caches ok.

And so it is much more beneficial to schedule the same process on the same CPU repeatedly, because what will happen is the working set, the memory set of that particular process will in the respective cache will get warmed up with that working set and so you will have much fewer cache misses. And you know saving the number of cache misses is a significant optimization for any compute bound process ok, especially on modern systems.

So, then there is something that is implemented in operating systems today is what is called affinity scheduling. So, you run some sort of you know logic to basically try and ensure that the same process gets to run on the same CPU if it gets rescheduled right. If it is not a strict affinity, but it is a high probability affinity scheduling. Which means that a

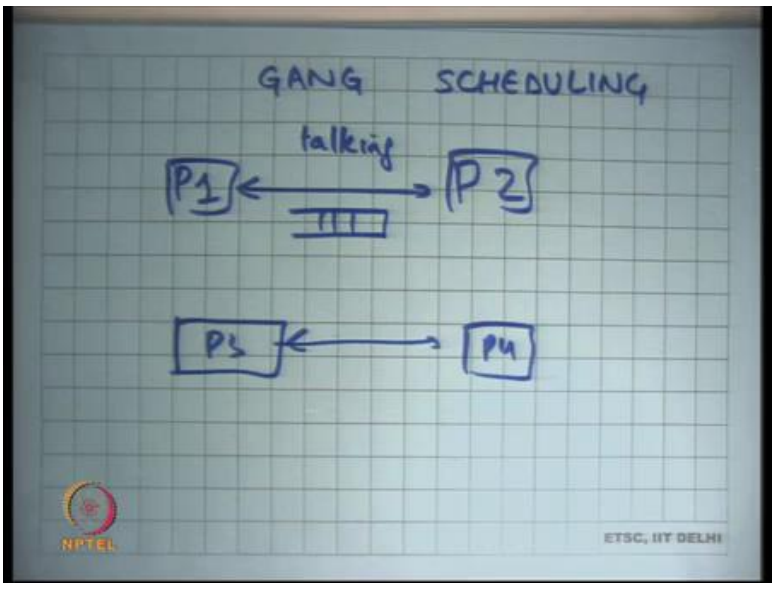
process has an affinity to a particular CPU, so whenever it gets it is scheduled, it will you know one CPU will be preferred over the other CPU and in scheduling it.

This can be implemented in multiple ways. You could have separate queues for each schedule each CPU, and you know so that these processes only get two schedule get two schedule on these CPUs if this CPU and these processes also get to schedule on all these CPUs.

Notice that this cache caching also has a significant bearing on the time quantum. Your time quantum should be large enough such that the process that gets to run has enough time to warm up its cache and then run on a warm cache. If most of the time on in that time quantum is just (Refer Time: 34:43) warming up the cache, and then you get scheduled out that is a very wasteful system right.

So, you should have enough time in your scheduling quantum to warm up the cache and then run on a warm cache for 90 percent of the time. So, let say 10 percent of time is taken to warm up the cache, and then 90 percent of the time is actually spent in running on a warm cache all right. So, you can have much better cache utilization if you are doing affinity scheduling.

(Refer Slide Time: 35:10)




There is also something called gang scheduling. So, if there are processes P1 and P2 that communicate with each other, let us say they have a producer consumer queue between

each other. Let us say one P1 is the network thread and P2 is the server thread, and they could talk to each other because they have a queue in the middle, and they will just keep exchanging information between each other right. So, there is a let us take an example, let us say there is a producer consumer queue or any other thing, so they are communicating talking to each other right.

And now there is P3 and P 4 that are also talking to each other. So, it will be a much better scheduling policy to basically say that P1 and P2 will get scheduled together. So, they will scheduled get scheduled as a gang, so that is called gang scheduling. So, you know P1 and P2 will get scheduled together. So, when they run, they are actually you know simultaneously, he produces he consume and consumes and so the buffer does not get full, there is no waiting and spinning and so it becomes very fast right.

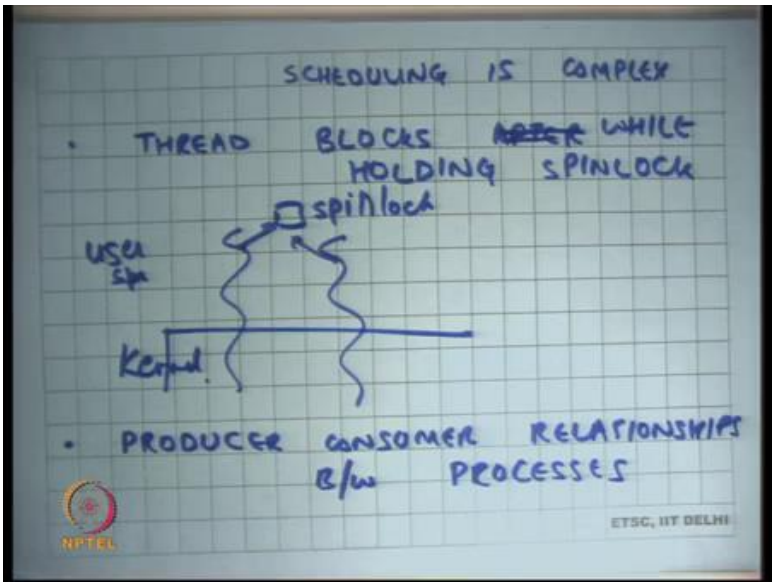
As opposed to let us say I did not schedule P1 and P2 together. So, P1 gets to run P1 fills up the producer queue, and then spins for some time just wasting or blocks which basically means there is extra overhead, then P2 gets to run, he empties the queue, then he is again spins to wait on an empty queue or blocks, and these are all wasteful operations as opposed to you know P1 and P2 running simultaneously.

So, it can actually significantly increase throughput for to communicating processes, so gang scheduling. So, the question is can the operating system always figure out that these are community communicating processes right. So, let us say; let us say there was there were two processes that were using a producer consumer queue that producer consumer queue was or there were two threads there were using a producer consumer queue when the producer consumer queue was completely implemented in user space.

So, the OS has absolutely no knowledge about this sharing behavior. It kicks you know they can they are they are some methods to figure out that there is some sharing, but you know those are two long winded to basically be really practical. If for example, you know these processes were communicating using a pipe, using an operating system pipe, then yes you know the operating system as full knowledge that P1 and P2 are communicating with each other.

He is writing to the pipe and he is reading those pipe and so the operating system can actually schedule them together. And there is of gang scheduling and get much better.

(Refer Slide Time: 37:37)




But in general, you know I am going to give you some examples to show that scheduling is actually complex all right. So, one example is thread blocks after holding spinlock or while holding instead of saying after that is a while holding spin lock. So, let us say there are you know there are 10 threads in the system, and they are all wanting to operate on some shared data and so they need to have a common lock right.

And so there is one thread that hold that takes a spin lock, and before it could actually leave the spin lock it either blocks which means that actually makes a you know it blocks voluntarily which means it makes an IO call let us say it makes a disk operation or there was a page fault right. So, page fault will also cause the thread to block right.

What will happen if there is a thread which is running and there is a page fault you basically just say that this thread is no longer ready you wait for the disk to bring the page into memory and you will let other processes run in the meanwhile that is what that is what we going to do. So, even if there is a page fault or if the process calls a system call which says disk read or any other IO call or if there was a preemption, there is a timer interrupt while you are holding the spin lock. All these cases are examples, where you are holding the spin lock and you have been de scheduled you know you are no longer running on the CPU.

And so, what will happen is all these other 9 threads will get to run, and they will just probably spin on the spin lock for the entire duration of the time quantum. So, they will

exhaust. So, spin lock is an example of synchronization which does not involve the operating system right, just like a producer consumer user left space producer consumer queue has absolutely no visibility for the operating system.

Spin lock is an example of a synchronization primitive that has no visibility for the operating system. So, the operating system has no idea that there is this something called spin lock spin lock is completely existing in the user space. Now, this thread actually takes a spin lock and gets de scheduled, all the other 9 threads just keep spinning on the spin lock, absolutely doing no useful work, but the operating system schedules them one after another and wasting a lot of CPU cycles all right.

What would have been a better way to do this? Well, if the operating firstly if the operating system knew that there is something called a spin lock that would have been the better thing to do. What does it mean for the programmer, so you know and there are other ways to do the to deal with these problems and they those ways to deal with this are have to do with operating system design?

So, in the Unix semantics as we have discussed you know this is possible and this is a very bad thing that can happen. And the only way the programmer can safeguard herself from you know not having this kind of a problem is to make sure that your critical if you are only protecting critical sections which are very small using spin locks all right.

Also, you make sure that within in those critical sections, you are not making any blocking calls. So, for example, you are not making any discrete or any other system calls that can take a long time. Also ensure try to make sure that you are not accessing too much memory, so that the probability of a page fault in that critical section is small all right.

So, these are some safeguards that you can take from to ensure that you know with Unix semantics of an operating system you basically do not have these kinds of problems where you know all threads wait for the spin lock. What could have been you know I am constantly using the word Unix semantics, what could have been the other semantics?

So, in the semantics that we have discussed so far, we are saying that the operating system has full power. It can take away the CPU from the operating system anytime it likes right. The other way to do it is you can tell the operating the process I want to take

the CPU back from you and the process has some idea of what are the resources it is holding. For example, it is holding spin lock and it can release those spin locks or you know finish up the it is critical section before it releases those spin locks and then saying here I am.

So, instead of a preemptive snatching of the resource from the process, it can be a more civilized you know requesting of the resource. You basically tell ask the process you know I need the resource back. So, you will say give me a second, I will just clean up a things, so that you know you have more best space throughput and then you know he gets at to you right.

Of course, you know this more civilized thing has of the problem that the process is not trusted. So, you know if you have asked him you know over the guarantee that he will actually obey you. So, in which case you know you also have some fallback mechanism. So, you have basically you know after you requested him, you wait for some time to you know, and if he does not behave as the good citizen you actually snatch it from him you know. So, these are ways to try to prevent this problem I am going to discuss some operating system models that actually do this kind of thing and it is very useful.

Student: You know (Refer Time: 42:35) spin lock is required by a CPU is not by a thread. So, as a how will it be possible that different different processes have been scheduled and all of them keep waiting for (Refer Time: 42:45).

So, question is you know spin lock is required by a CPU not by a thread. Well, that is not true, you know the spin lock that we discussed in XV6 was a per CPU spin lock, because you know we also disable interrupts, so that make meant there was no context switching. So, there was the spin lock was also asserted with the CPU, but in general spin locks have no need not be per CPU they can be per thread.

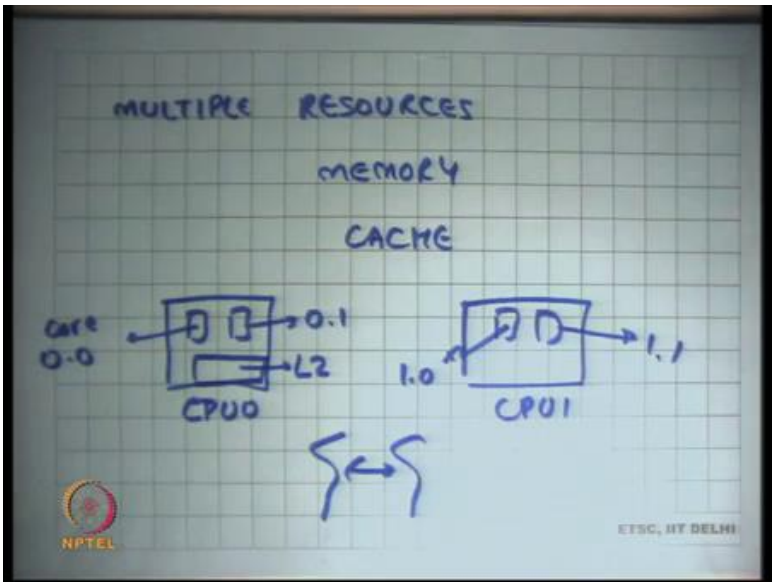
So, here is an example let us say you know there are threads, and these are this is kernel space, and this is user space, and they are basically having a spin lock here right. And so and this thread can call acquire on this and this thread can call acquire on this and the kernel has no idea that there is a spin lock that is going on. This acquire has nothing to do with clearing of interrupts or anything this. These are spin locks for threads all right.

Just like threads you know spin locks there can be producer consumer relationships. So, spin locks is just one example, but you know there can be any types of synchronization between processes right. For example, there is a producer thread and there is a consumer thread, and you know the producer thread has not actually filled up anything and it get switched out. And the consumer thread gets to run and lots of the consumer threads.

So, all the consumer threads will just and let us say the way you are implemented the producer consumer queue is completely at the user level, and these threads spin if you consumer thread spins if it finds the queue to be empty, and the producer threads spins if it finds a queue to be full. And so, the operating system or the scheduler has no idea that there is a producer consumer relationship and it will keep scheduling the consumer threads only for them to spin and do really no useful work all right.

It would have been much better if there was a way for the operating system to know this or better the operating system would tell and indicate it is intentions to the operator to the process that I want to take the CPU away from you, or I want to schedule you, and get it is feedback on you know how much time you know when I should do that, what is the right way time to do that all right.

(Refer Slide Time: 44:55)




Also, there are multiple resources. So, I am talking about CPU, but they are actually multiple resources like memory all right. So, if I you know if I have a high priority process let us say vi and gcc example, and I am giving lot of CPU I am giving vi higher

priority over gcc. But that is of no use if I am not giving vi also the amount of memory that it needs right. If the higher priority process is not getting the same amount of memory that it needs, then the first thing it will do when it gets scheduled is block by taking a page fault.

So, priority does not need to be obeyed only on the on one resource with this CPU, it also needs to be obeyed or there some needs to be some reflection of it on the other resources, in this case the physical memory right. So, for example, if there is a higher priority process perhaps it you know it is working set should be in memory as opposed to the lower priority process.

Of course, once again there is an issue of you know one process should not get be able to run away with all the memory. And so there has to be some the some safeguards in scheduling the memory. So, there is not just scheduling. So, they are scheduling in time of CPU, but there is also scheduling in space of the memory and these to need to work in tandem with each other that makes things a little difficult.

And we have also talked about cache scheduling. So, cache is basically caches in even scarce resource then memory. So, you know for best performance you would want that the cache is properly scheduled, affinity scheduling being one example of a good way of in scheduling a cache.

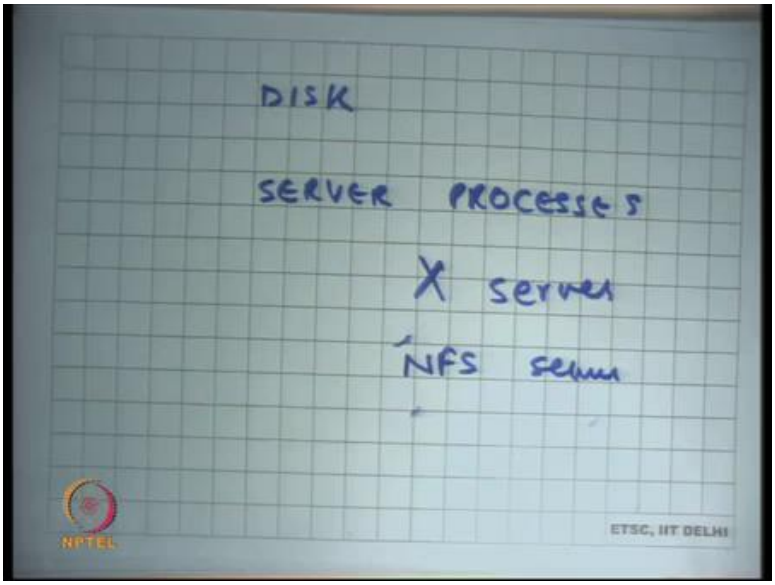
And there you know similarly if there is a producer consumer relationship for example, let us say you know you have CPUs. So, these days you get multi core CPUs. So, let us say there are two CPUs which have two cores each. So, this is CPU 0, and this is CPU 1, and this is core 0 dot 1 and this is core 0 dot 1 this is 1 dot 0, and 1 dot 1.

If there are two processes that have a producer consumer relationship between them then it is better to schedule both of them on the same CPU together right. So, that way you are efficiently using the cache, because each CPU here will have L 2 cache that they can share between themselves as opposed to going to the main memory to implement to do this producer consumer relationship right.

So, there is a very complex thing here that you know you first need to figure out that who are having the producer consumer relationships, then you to figure out you know what

are the parts of the CPU which CPUs have shared caches, and then schedule those processes in tandem in a gang scheduling way to do that all right.

(Refer Slide Time: 47:30)




Then you know similarly this disk. So, for example, you know there are page faults from multiple processes, there is a page fault from the vi process and there is a page fault from the gcc process the disk schedule has no idea which has which requires has higher priority and which should get scheduled first. And you know either I you know either I do not care about it in which case I am not actually going strict priorities.

There are other parts of the system that have no idea about your priority system or I you know flow this information down in which case I can severely impact my throughput right. So, if I do; if I do strict priority scheduling at the disk level, then you know it is I cannot and can no longer follow the elevators algorithm and maximize my throughput.

Also, there are server processes like for example, the X severs right. So, X server runs as a separate process and all the other processes talk to the X server to you know make their requires for displaying or rendering their screens. And so, the X server has no notion of priority or you know it is very you know it will require a more interfaces to ensure that the X server also has some notion of priority.

But ideally if you know one process a higher priority over the other process, his request should be given more priority over the other processor priority. So, I do not care about

gcg screen so much, but I definitely care about vi screen for example and so you know how is this kind of a thing done. From a schedulers perspective he is blind to you know what is up he just sees that the X server wants to access the display device, but the X server internally has no idea that you know which required which process has higher priority or not.

One could build it in, but you know that makes life complex there are multiple things that are happening here. In another example is an NFS server right. So, NFS server is a network file system, you could have a separate server and the multiple processes that are accessing this NFS server. But the NFS server needs to be either made aware of the priorities or you are not actually following strict priorities ok, so good so that is it for scheduling.

I am going to discuss one very interesting example of non-linear effects of scheduling. So, scheduling in general is a problem if the resources are relatively scarce right. So, if you have lots of resources let us say CPU is plentiful and you do not care about it, there is no problem right, scheduling is not an issue, but the moment resources become scarce it becomes an issue.

If you are running a web server that is handling a 100 or 1000 requests per day the performance of a scheduler will actually not matter at all. But if you are running a web server that you want to operate at peak capacity let us say hundreds of thousands of requests per second, then you know the way you schedule your request and the way you schedule your resources becomes very important right.

You are you know you have you are bottlenecked by your caches, you are bottlenecked by your disk throughput, you are bottlenecked by the CPU and so the an efficient scheduling and inefficient scheduling there can be a large gap in performance between the two.

In fact, the gap can be so large that at high throughput I am going to show you know we are going to discuss a an example of an where a poor scheduling can actually lead to zero throughput at very high loads ok. So, there is a non-linear effect, it is not necessary whereas, a good scheduler can actually you know saturate very gracefully in presence of high load right.

So, overall, I mean scheduling was a very important topic in the early days of computers 1970s, where this would be large, shared computers that are very low computing power and lots of people sharing them. Over the years the computers became more and more powerful and more and more distributed. We had personal computers and each and so fewer users and more power and so scheduling was less of a problem is less of a problem on a desktop or laptop PCs.

But you know we are moving back to you know let us say what is called cloud computing, where you have a centralized computer or centralized data center and being shared by lots of people. And once again to maximize the utilization of your resources, so and so once again you know to operate your centralized system which is shared by lots of different processes scheduling becomes again important in the modern world as well good. Let us continue the discussion next time.