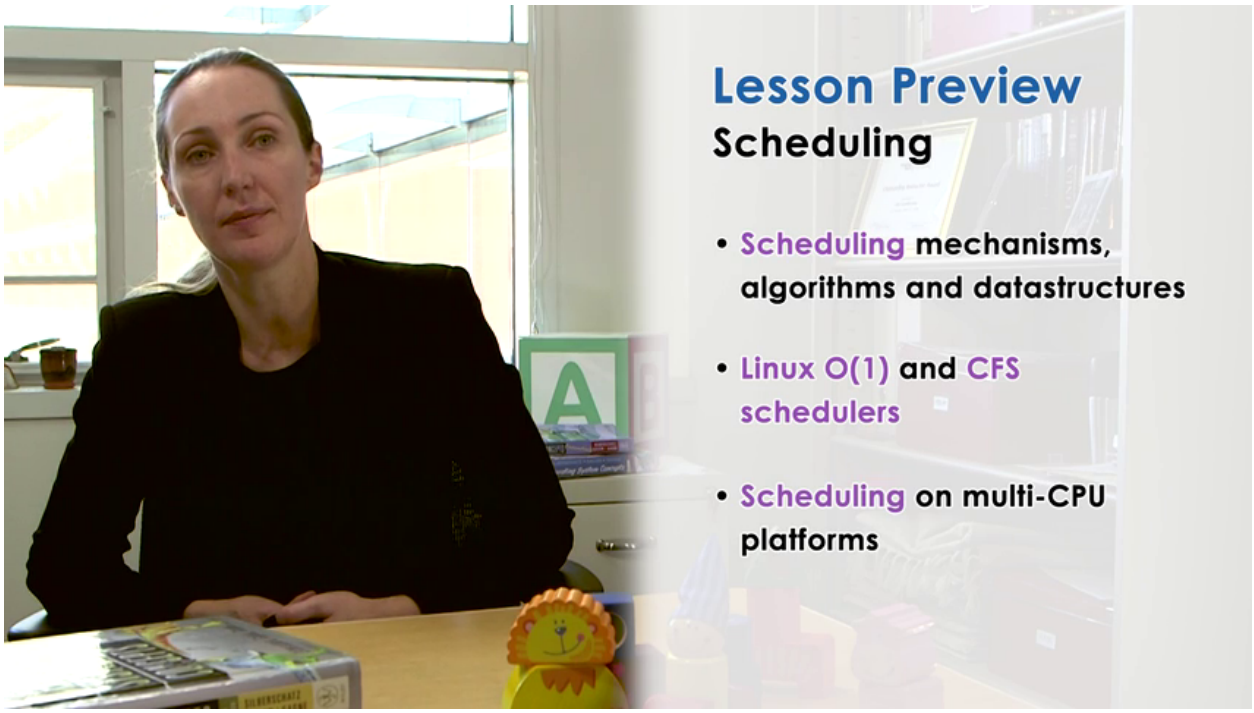
P3L1 - Scheduling

01 - Lesson Preview



Instructor Notes

- Fedorova, Alexandra, et. al.
- "Chip Multithreading Systems Need a New Operating System Scheduler".
- Proceedings of the 11th workshop on ACM SIGOPS European workshop (EW 11). ACM, New York, NY, 2004.F

Schedulers:

- O(1)
- CFS (Completely Fair Scheduler)

In the next set of lessons we will discuss in more detail some of the key resource mgmt components in operating systems. First we will look at how the OS manages CPU and how it decides how processes and their threads will get to execute on those CPUs. This is done by the scheduler. We will review some of the basic scheduling mechanisms, some of the scheduling algorithms and data structures that they use, and we will look in more detail at some of the scheduling **algorithms used in the Linux operating systems: the completely fair scheduler and the O(1) scheduler.** We will also look at certain aspects of scheduling that are common for multi cpu platforms. This includes multi core platforms and also platforms with hardware level multi threading. For this we will use the paper “Chip multi-threaded processors need a new operating system scheduler” in order to demonstrate some of the more advanced features that modern schedulers should incorporate.

02 - Visual Metaphor
Issue with scheduling described by toy shop metaphor. Assessment of complexity of orders.

FIFO - First In First Out
FCFS - First Come First Serve
SJF - Shortest Job First

In this lesson we will be talking at length about scheduling and OS schedulers. To continue with a visual metaphor, we'll try to visualize what are some of the issues when it comes to OS schedulers. Using our toy shop analogies we'll think of an OS scheduler and we'll compare it with a toy shop manager. Like an OS scheduler, the toy shop manager schedulers work in the toy shop. For a toy shop manager there are multiple ways how it can choose to schedule the toy shop orders and dispatch them to workers in the toy shop. First the toy shop manager can dispatch orders immediately as soon as they arrive in the toy shop or the manager may decide to dispatch the simple orders first. Or even conversely the manager can decide to dispatch the complex orders as soon as they arrive in the shop. The motivation for each of these choices is based on some of the high level goals of the manager and how he wants to manage the shop and utilize the resources in the shop. Let’s investigate these choices a little more.

If the manager wants low scheduling overhead, not to have to analyze too much what he needs to do when an order comes, he can choose to dispatch the orders immediately. The scheduling in this case is very simple. We can think of it as a FIFO. First order that comes in is the first one that will be served. If the manager is concerned with the total number of orders that get processed over a period of time, it would be good to **dispatch the simple orders as soon as they arrive**. They will be the ones that will be completing more quickly and that will maximize this particular metric. Obviously in order to do this, the manager has to spend a little bit more time whenever an order comes because

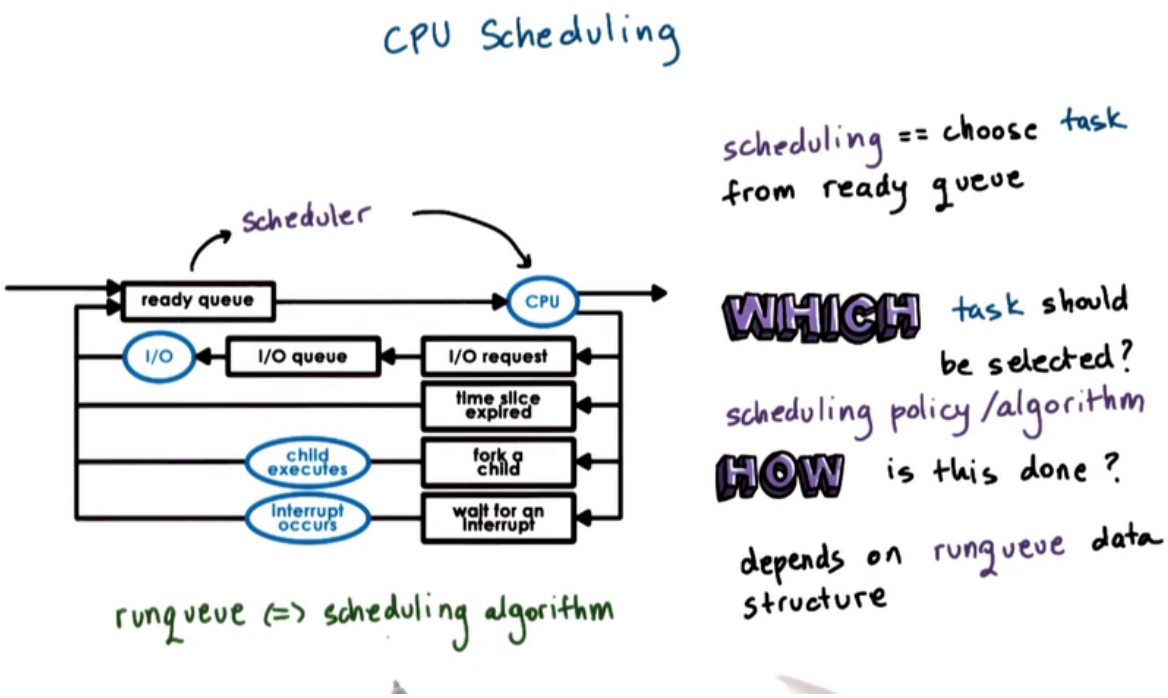
he needs to check what kind of orders that is , whether it's going to be a simple one or not and then decide what to do with it. So basically, **it will be important to the manager to be able to assess the complexity of the orders.**

On the other hand, on each of the workbenches in the shop there may be different kinds of tools and these simple orders, these may not really exercise of these tools. They may not require the use of all aspects of the workbenches. So the manager wishes to keep all of the resources that are available on the workbenches as busy as possible, he may choose to schedule complex orders as soon as they arrive, and then whenever simple ones come in, perhaps the workers can pause the processing of the complex order to get a simple one in and to finish it as soon as possible. In comparison, the scheduling options for toy shop manager are surprising similar to those of the OS scheduler. For an OS scheduler, these choices are analyzed in terms of tasks which is either a thread or process and these are being scheduled onto the CPU's that are managed by the OS scheduler. Like the workbenches in the case of the toy shop manager. For an OS scheduler we can choose a simple approach to schedule tasks in a first come, first serve manner. The benefit is that the scheduling is simple and that we don't spend a lot of overhead in the OS scheduler itself. Using a similar analogy to what the toy shop manager was thinking of when he chose to dispatch the simple orders first, an OS scheduler can assign and dispatch simple tasks first. The outcome of this kind of scheduling can be that the throughput of the system overall is maximized and there schedulers that actually follow this kind of algorithm and those are called Shortest Jobs First. Simple in terms of the task is equal to running time so shortest job first. Finally the scheduling logic behind assigning complex tasks first is similar to the case in the toy shop. Here the scheduler's goals are to maximize all aspects of the platform so to utilize well both the CPU's as well as any other devices memory or other resources that are present on that platform. As we move through this lesson we will discuss some of the options that need to be considered when designing algorithms such as these. And in general we will discuss the various aspects of the design and implementation of OS schedulers.

03 - Scheduling Overview

2

Task - a process or thread

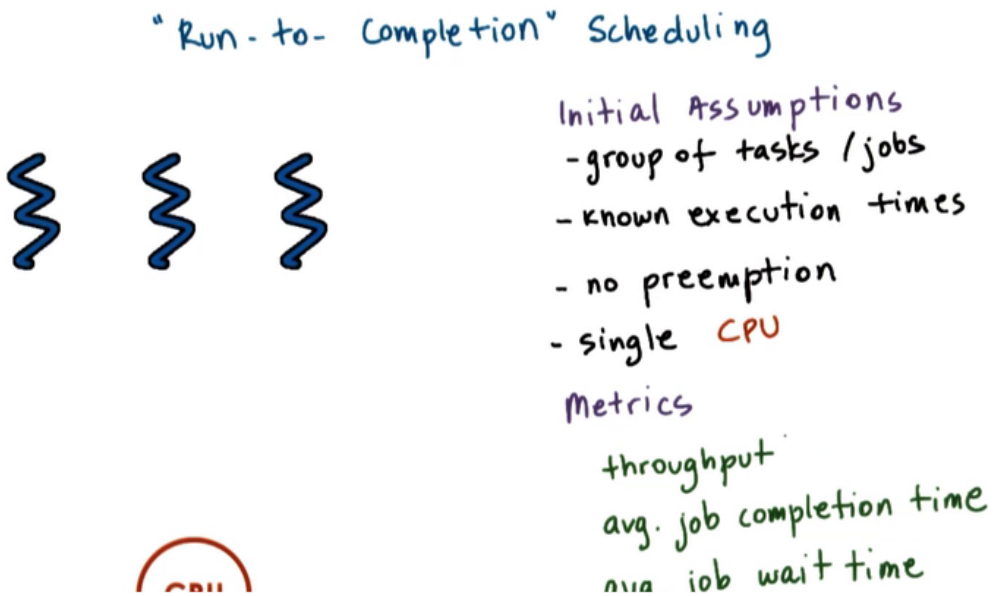


Different ***runqueues*** are used for different ***scheduling algorithms***.

We talked briefly about CPU scheduling in the introductory lesson on processes and process mgmt. We said that the CPU scheduler decides how and when processes access the shared CPU in the system. In this lesson we'll use the term task to interchangeably mean either processes or threads since the same kinds of mechanisms are valid in all contexts. The scheduler concerns the scheduling of both user level processes or threads as well as the kernel level threads. Recall this figure that we used when we talked originally about processes and process scheduling. The responsibility of the CPU scheduler is to choose one of the tasks in the ready queue a process or thread we'll just use the term task to refer to both and then to schedule that particular task onto the CPU. Threads may become ready so they may enter the ready queue after an IO operation they have been waiting on has completed or after they have been woken up from a wake interrupt for instance. A thread will enter the ready queue when it's created so when somebody has forked a new thread and also a thread that maybe was interrupted while executing on the CPU because the CPU has to be shared with other threads. That thread, it was ready and it was executing on the CPU so after it has been interrupted it continues to be ready so it will immediately go into the ready queue. So to schedule something the scheduler will have to look at all of the tasks in the ready queue and decide which is the one that able dispatch to run on the CPU. Whenever the CPU becomes idle, we have to run the scheduler. For instance if a thread that was executing on the CPU makes an IO request and now it has to wait in the IO queue for that particular device, the CPU is idle, what happens at that moment we run the OS scheduler. The goal is to pick another task to run on the CPU as soon as possible and not to keep the CPU idle for too long. Whenever a new task becomes ready,a task that was waiting on a IO operation or was waiting on an interrupt or some kind of signal or whenever a brand new task is created for the first time, for all of these reason again we have to run the scheduler. The goal is to check whether any of these tasks are of higher importance and therefore should interrupt the task that's currently executing on the CPU. A common way to how scheduler shares the CPU is to give each of the tasks in the system some amount of time in the CPU. So when a time

slice expires, when a running thread has used its time on the CPU, then that's another time when we need to run the scheduler so as to pick the next task to be scheduled on the CPU. Once the scheduler selects a thread to be scheduled, the thread is then dispatched onto the CPU. What really happens is that we perform a context switch to enter the context of the newly selected thread and then we have to enter user mode and set the program counter appropriately to point to the next instruction that needs to be executed from the newly selected thread. And then we are ready to go. The new thread starts executing on the CPU. So in summary, the objective of the OS scheduler is to choose the next task to run from the queue of threaded tasks in the system. The first question that comes to mind is what task should be selected? How do we know this? The answer to that will depend on the scheduling policy or the scheduling algorithm that is executed by the OS scheduler. We will discuss several such algorithms next. Another immediate question is how does the scheduler accomplish this? How does it perform whatever algorithm it needs to execute. The details of the implementation of the scheduler will very much depend on the run queue on the data structure that we use to implement this ready queue. That's called the Run Queue. The design of the run queue and the scheduling algorithm are tightly coupled. We will see that certain scheduling algorithms they demand a different type of run queue, different data structure, and also that the design of the run queue it may limit in certain ways what kind of scheduling algorithms we can support efficiently so the rest of the lecture will focus on reviewing different scheduling algorithms and run queue data structures.

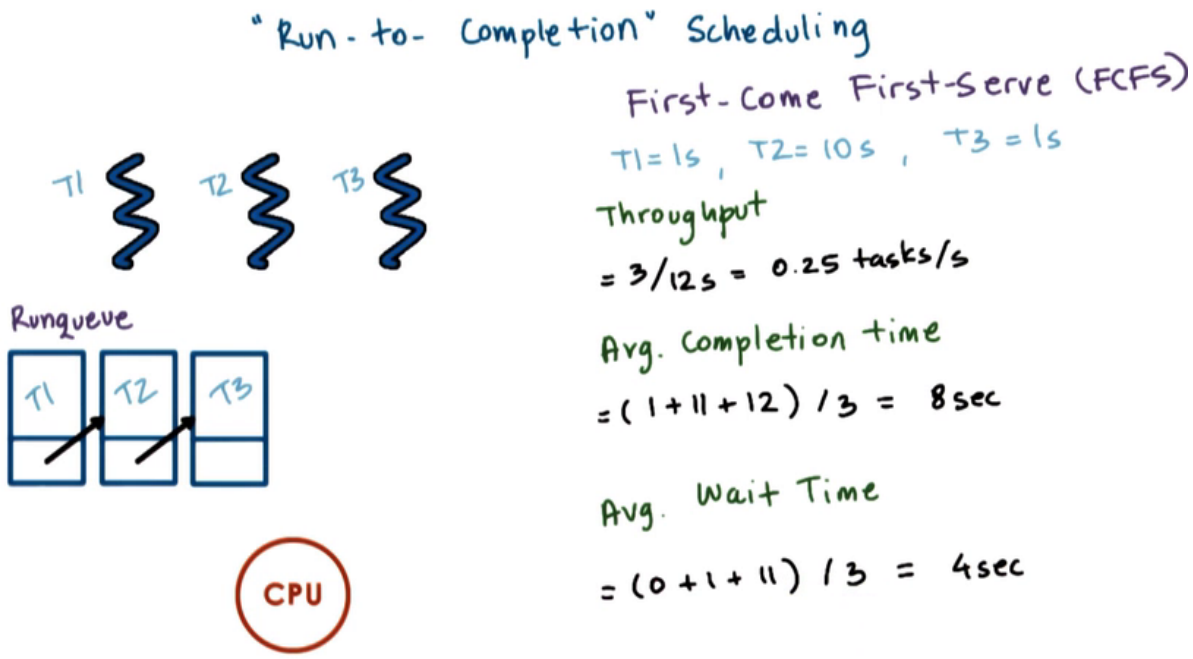
04 - Run To Completion Scheduling



For this first discussion of scheduling algorithms, I want to focus on what we call run to completion scheduling. This type of scheduling assumes that as soon as a task is assigned to a CPU, it will run until it finishes or until it completes.

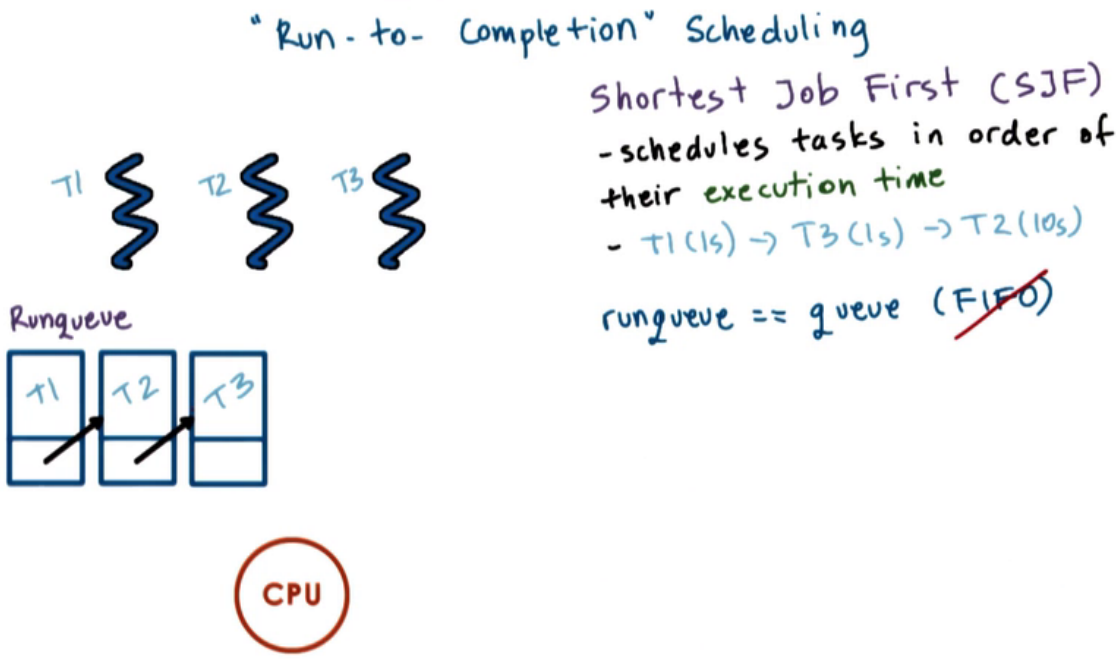
let us list some of the initial assumptions. We will consider that we have a group of tasks that we need to schedule. I'll refer to these also as threads and jobs and similar terms. Let me also start with assume that we will know exactly how much time these threads need to execute. So there will be no preemption in this system once starts running it will run to completion it will not be interrupted or preempted to start executing some other task and also to start with let us assume that we only have a single CPU. We will relax these requirements further as we go through this lesson. Now since we will be talking about different scheduling algorithms, it will be important for us to compare them so we'll want to think about some useful metrics. When it comes to comparing schedulers, some of the metrics that can give meaningful answers regarding those comparisons include throughput, the average time it took for tasks to complete, the avg time that tasks spent waiting before they were scheduled, overall CPU utilization. We will use some of these metrics to compare some of the algorithms that we will talk about.

The first and simplest algorithm is the first come first serve. In this algorithm tasks are scheduled on the CPU in the same order in which they arrive. Regardless of their execution time, of load in the system, or anything else. When a task completes, they will pick the next task that arrived in that order. Clearly a useful way to organize these tasks would be in a queue structure so that tasks can be picked up from a FIFO manner. Whenever a new task becomes ready it will be placed at the end of the queue and then whenever the scheduler needs to pick the next task to execute, it will pick from the front of the queue. To make this decision all the scheduler will need to know is the head of the queue structure and how to dequeue tasks from this queue. So basically for first come first serve scheduling, some FIFO like queue would be a great run queue data structure.

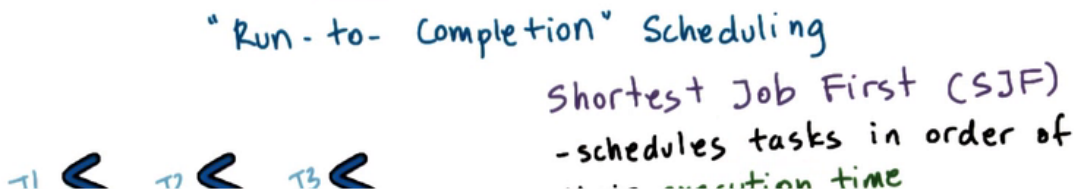


Now lets take a look at this area where these three tasks have the following executing times. T1 is 1 sec, T2 is 10s and T3 is also 1 sec. And let's assume they arrive in this order. So T1 followed by T2 followed by T3. So this is how they will be placed in the run queue. Now let's look at throughput as a metric for this kind of system.

Throughput - we have 3 tasks to execute one right after another, would take 12 seconds. So the scheduler on avg achieves .25 tasks per second. If we're interested in the avg compl3/etion time of these tasks the 1st task will complete in 1 sec. The second task will complete at time 11 seconds. The third will complete at time 12. So the average completion time would be 8 seconds. If we're interested in the avg wait time for the three tasks in the system then the first task started immediately, the second started a second later, and then the 3rd task had to wait for 11 seconds before it started executing. So the avg wait time for these 3 tasks was 4 seconds. So we have a simple scheduling algorithm however probably we can do a little better in terms of the various metrics that we're able to achieve with this algorithm when we try something else.



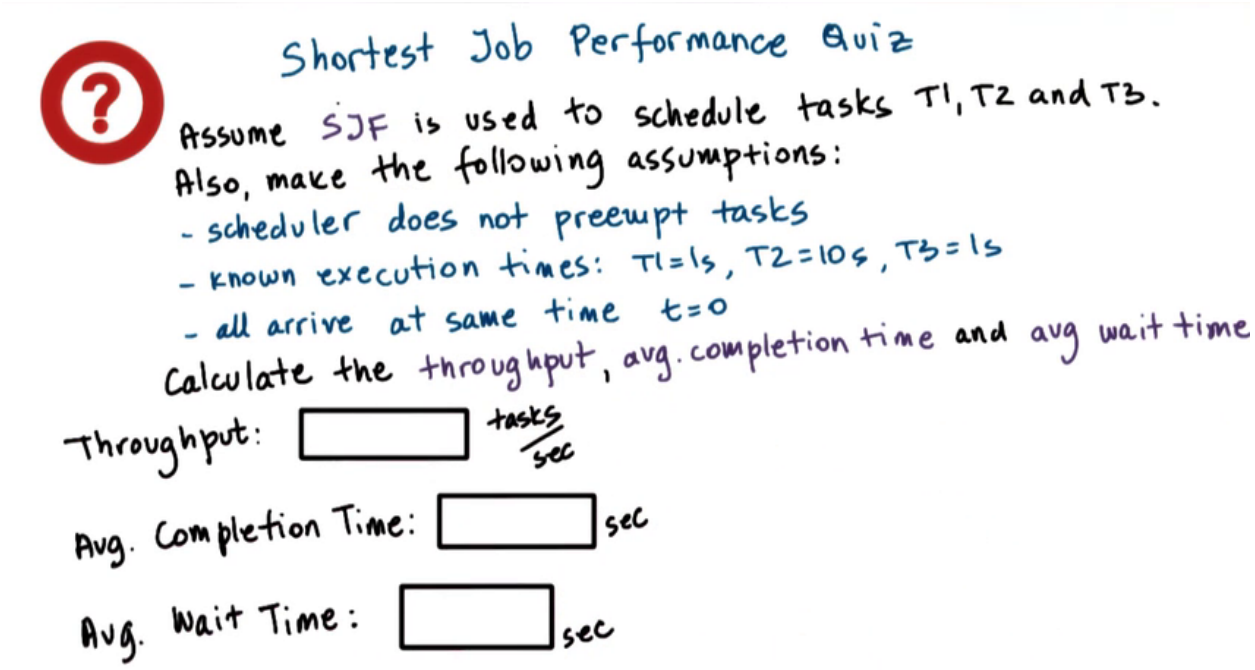
So we see that first come first serve is simple but the wait time for the tasks is poor even if there is just one long task in the system that has arrived ahead of some shorter tasks. So to deal with this we can look at another algorithm that's called Shortest Job First and the goal of this algorithm is to schedule the tasks in the order of their execution time. Given the previous example with tasks T1 T2 T3, the order in which we want to execute them would be T1 T3 and then T2 the longest task at the end. And for tasks that take the same amount of time to execute perhaps we break ties arbitrarily. If we organize our run queue the same was as before, adding new task to the run queue will be easy since it will just mean that we have to add that task at the tail of the queue. However, when we need to schedule a new task we'll need to traverse the entire queue until we find the one with the shortest execution time so run queue won't be a FIFO run queue anymore since we will need to remove tasks from the queue in a very specific order based on the execution time.



Rebalancing of ***tree*** could be done while the scheduler always evaluates the node on the left most, since it would be the node corresponding to the shortest job.

One thing that we can do is that we can maintain the run queue as an ordered queue so that tasks when they're inserted into the queue are placed in the queue in a specific order. It will make the insertion of tasks into the queue a little bit more complex, but it will keep the selection of the new task as short as it was before. We just need to know the head of the queue. Or our run queue doesn't have to be a queue. It can be some tree like data structure in which the nodes in this tree are ordered based on the execution time. When inserting new nodes in the tree, new tasks in the tree, the tree may need to be rebalanced. However, for the scheduler it will be easy since it will always have to select the left-most node in this tree. If the tree is ordered, the left most node will have the smallest execution time. So we have a queue, a tree, this illustrates that the run queue doesn't really have to be a queue. It can be another type of data structure. And we'll see that it often is based on what is appropriate for the scheduling algorithm.

05 - SJF Performance Quiz



Instructor Notes

Quiz Help

Throughput Formula:

- jobs_completed / time_to_complete_all_job

Avg. Completion Time Formula:

- time_to_complete_all_job / jobs_completed

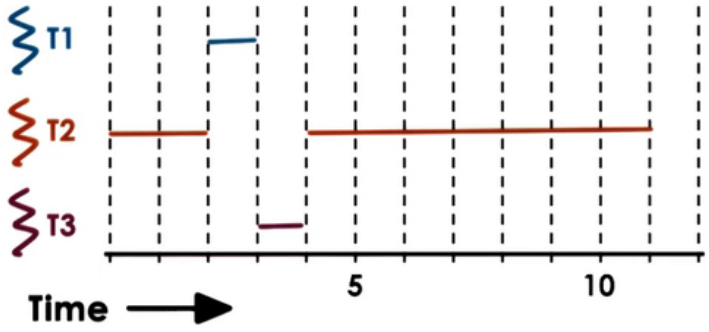
Avg. Wait Time Formula:

- (t1_wait_time + t2_wait_time + t3_wait_time) / jobs_completed

You do not have to include units in your answers. Also, for decimal answers, please round to the hundredths.

06 - SJF Performance Quiz

07 - Preemptive Scheduling SJF Preempt



Preemptive Scheduling
SJF + Preemption
- T2 arrives first
- T2 should be preempted
heuristics based on history =>
job running time
- how long did a task run

ERRATA:
T2 should execute until t=12

So far in this discussion we assumed that a task that's executing on the CPU cannot be interrupted or preempted. Let's now relax that requirement and start talking about preemptive scheduling. Scheduling in which the tasks don't just get the CPU and hog it until they're completed. So we'll consider preemption first in the context of the shortest job first algorithm. And for this we will also make another assumption, or we'll modify another assumption that we made earlier. Tasks don't have to arrive at the same time. So we're gonna look at a system that has 3 tasks. We know their execution time, so that assumption still holds. And we will assume that they arrive now at arbitrary times. In this particular case, T2 arrives first. When T2 arrives, it's the only task in the system. T1 and T3 haven't arrived yet so the scheduler will clearly schedule it and it will start executing. When the tasks T1 and T3 show up then T2 should be preempted. We're using shortest job first and T1 and T3 have shortest jobs compared to T2. The execution of the rest of the scenario will look like something as follows. Let's say at T2 when the tasks T1 and T3 show up, T1 is scheduled next. Once it completes, T3 is the next one that has the shortest running time so T3 will execute and once these two have completed, T2 can take the remaining of it's time to execute. So basically what would need to happen to achieve this kind of behavior is that whenever tasks enter the run queue like T1 and T3, the scheduler needs to be invoked so that the scheduler can inspect their execution times and then decide whether to preempt the currently running task (T2) and schedule one of the newly ready tasks. Now so far we talked as if we know the execution time of the task, but in principle it's hard to tell. It's really hard to claim that we know the execution time of a task. There are a number of factors. It depends on the inputs of the task, whether the data is present in the cache or not. which other tasks are running in the system. In principle, we have to use some kind of heuristics in order to guesstimate what the execution time of a task will be. When it comes to the execution time of the task, it's probably useful to consider what was the past execution time of that task for that job. In a sense, history is a good predictor of what will happen so we will use the past execution time to predict the future. For instance, we can think about how long a task ran the very last time it was executed or maybe we can take an average over a period of time or over a number of past runs of that particular task. We call this scenario in which we compute the averages over a period of the past a windowed average. So we compute some kind of metrics based on a window of values from the past and use that average for prediction of the behavior of the task in the future.

08 - Preemptive Scheduling Priority

ERRATA:
On timeline: T2 should go until time=11, T1 should execute from time 11-12

Shortest job considers the execution time of the tasks in order to decide how to schedule task and when to preempt a particular task. Another criteria for driving those decisions may be that tasks have different priorities. Tasks that have different priority levels, that is a pretty common situation. For instance we have several operating system level threads that run OS tasks that have higher priority than other threads that support maybe user level processes. Or even within a user level process, may be certain threads that are intended to run when there is user input. Such threads may have higher priority compared to other threads that just do some background processing or long running simulations. In such scenarios, the scheduler will have to be able to run the highest priority task next. So clearly it will have to support preemption. It will need to be able to stop a low priority task and preempt it so that the higher priority one can run next. So let's look now at the same example from before, except now we're going to use Priority Scheduling and we need to know the tasks priorities. In this particular example, the priorities P1 P2 and P3 are such that the first thread P1 has the lowest priority. Followed by the second thread P2 and finally the 3rd thread has the highest priority P3. Again we start with the execution of T2 since it's the only thread in the system. Now however when T1 and T3 become ready at this point of time when time is 2, we'll have a very different execution compared to the shortest job first with preemption scheduler. So when we look at the priorities we see that T3 has the highest priority. So when threads T1 and T3 are ready, T2 is preempted and the execution of T3 will start. When T3 completes, at that point, T2 starts running again and then T1 will have to wait until T2 completes to start. So T1 will not start until the 11 seconds and then it will complete at time 12. The entire schedule will complete at this time as well. In this example we were looking at this table, but in principle our scheduler if it needs to be priority based scheduler, it will somehow need to quickly be able to assess not just what are the runnable threads in the system that

are ready to execute but also what are their priorities and it will need to select the one that has the highest priority to execute next.

Different ***run queues*** for each priority level. Could use priority ordered tree structure.

We can achieve this by having multiple run queue structures. Different run queue structures for each priority level. And then have the scheduler select a task from the run queue that corresponds to the highest priority level. Other than having per priority queues, another option is to have some kind of ordered data structure like for instance the tree that we saw with the shortest job first however in this case with priority scheduling this tree would need to be ordered based on the priority of the task. One danger with priority based scheduling is starvation. We can have a situation in which a low priority task is basically infinitely stuck in a run queue just because there are constantly higher priority tasks that show up in some of the other parts of the run queue. One mechanism that we use to protect against starvation is so called **priority aging**. What that means is that the priority of the task isn't just a fixed priority. Instead it some kind of function of the actual priority plus one other factor and that is the time that the thread or task spent in the run queue. The idea is that the longer a task spends in a run queue, the higher the priority should become. So eventually, the task will become the highest priority task in the system and it will run. In this manner starvation will be prevented.

DANGER: ***starvation*** possible. A way to fix this is with ***priority aging*** which will change the priority of a task that has been sitting on a runqueue for a period of time.

09 - Preemptive Scheduling Quiz

Quiz Help

- You do not have to include units in your answer

- This scheduler preempts tasks
- Pay close attention to the ordering of priorities: P3 < P2 < P1
- In a scratch area, fill out the graph to find the answers

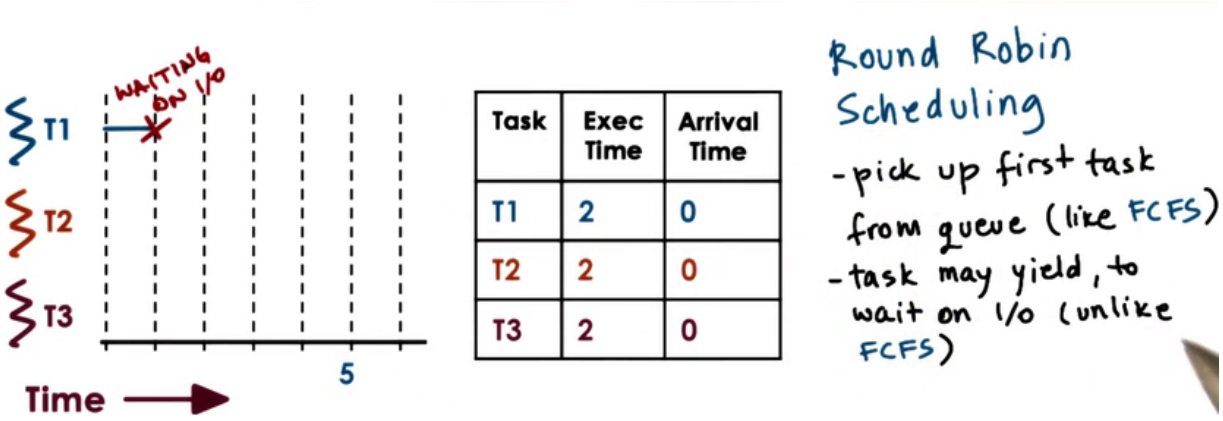
10 - Preemptive Scheduling Quiz

11 - Priority Inversion

An interesting phenomenon called priority inversion happens once we introduce priorities into the scheduling. Consider the following situation and we'll use the shortest job first scheduling to illustrate what happens. We have 3 threads. The first one has the highest priority. And P3 has the lowest priority. For this example, we left out the execution times just assume that they take some longer amount of time all these tasks and that the time graph continues into the future. Initially T3 is the only task in the system, so T3 executes and also acquired a lock. Now T2 arrives (has higher priority than T3) so T3 will be preempted and T2 gets to execute. Now at time 5 T1 arrives and has higher priority than T2 so T2 will be preempted. So T1 executes runs for 2 units of time and tries to acquire a lock held by T3. Unfortunately, T1 can't acquire the lock so T1 is put on a wait queue that is associated with the lock. We schedule the next highest priority task which is runnable and that would be T2 and T2 will run as long as it needs. And let's say in this case we lucked out T2 really only needed to execute for 2 more units at that point T2 completes, we schedule the next highest priority runnable task (the only runnable task in the system is T3) so T3 will get to execute and in fact T3 will execute for as long as it needs to until it releases the lock. When T3 releases the lock, T1 will become runnable and it has the highest priority thread so T1 will preempt T3 and T1 will continue its execution. It will acquire the lock and continue executing. So based on the priority in the system, we were expecting that T1 will complete before T2, and that T3 will be the last one to complete. However that's not what happend. In the actual order of execution was as follows. T2 the medium priority thread, followed by T3 the lowest priority thread and then finally the T1 the highest priority task in the system. The priorities of these task were inverted. This is what we call priority inversion. A solution to this problem would have been to temporary boost the priority of the mutex owner. What that means, is that at this particular point when the highest priority thread needs to acquire the lock that's owned by a lower priority thread, the priority of T3 is temporarily boosted to be basically at the same level as T1. Then T1 could not have proceeded just as before given that the lock is owned by somebody else,

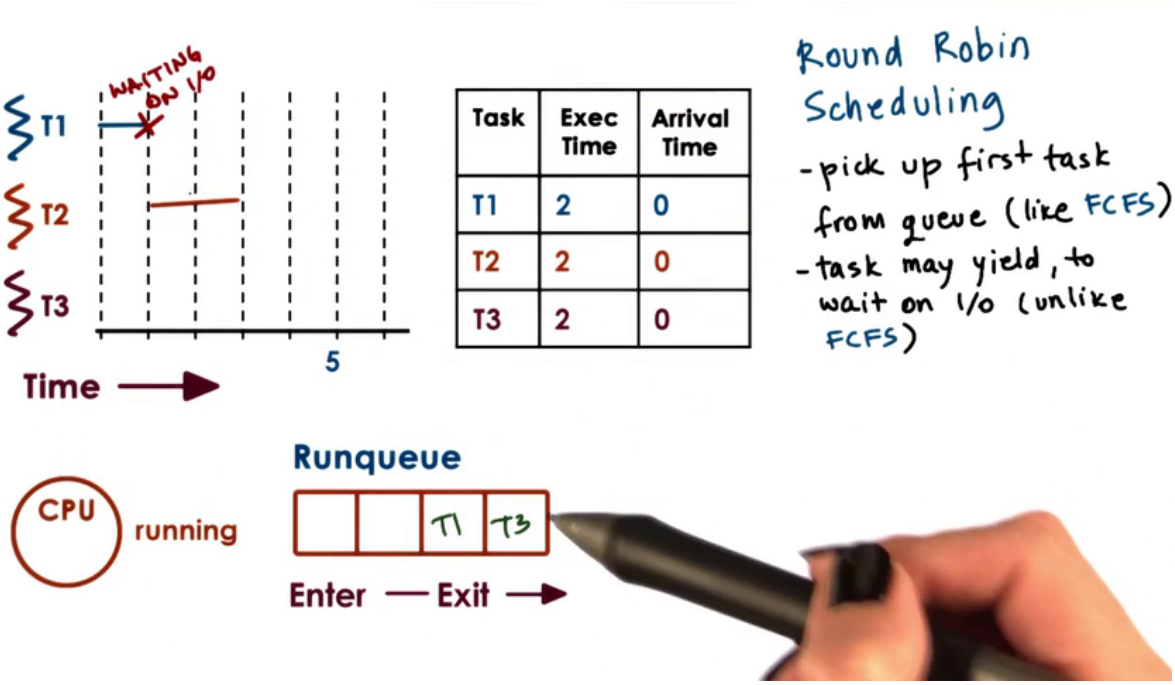
however instead of scheduling T2, we would have scheduled T3 because priority would have been boosted so it would have been higher than T2. So we wouldn't have to wait until T2 to complete before T1 can run again. This technique of boosting the priority of the mutex owner, this is why for instance it is useful to keep track of who is the current owner of the mutex. This is why we said we want to have this kind of information in the mutex data structure. And obviously if we are temporarily boosting the priority of the mutex owner, we should lower its priority once it releases the mutex. The only reason we were boosting its priority was so as to be able to schedule the highest priority thread to run as soon as possible. So we wanted to make sure that the mutex is released as soon as possible. This is a common technique that's currently present in many operating systems.

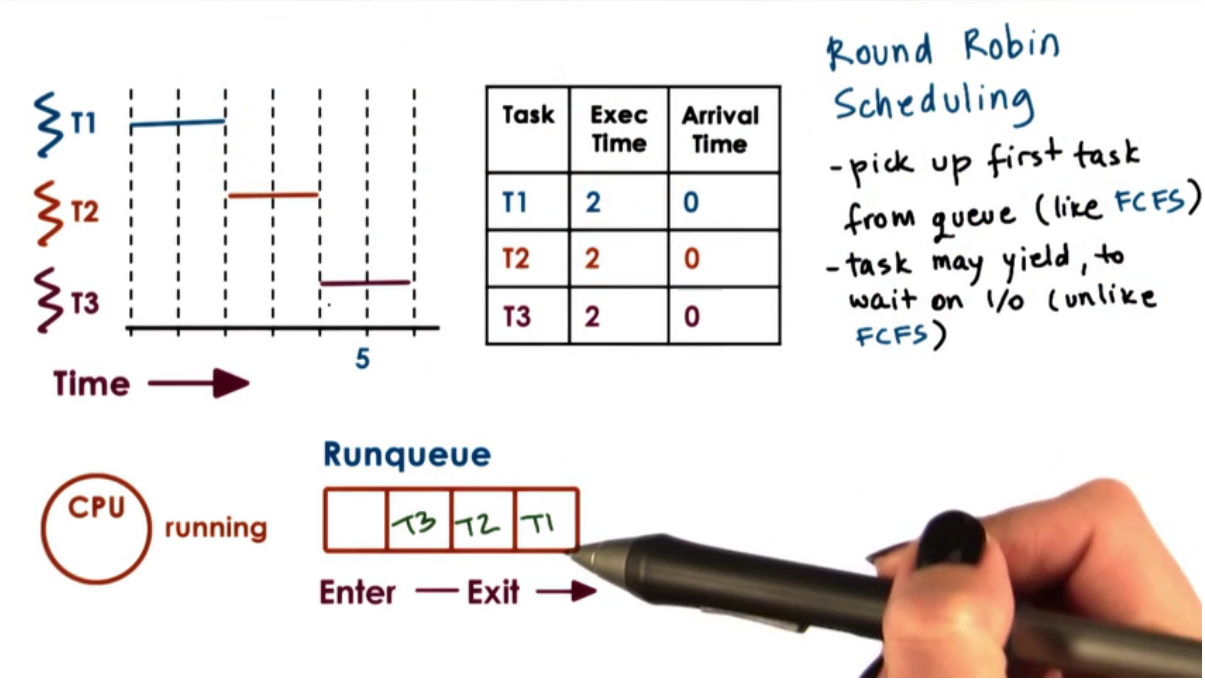
12 - Round Robin Scheduling



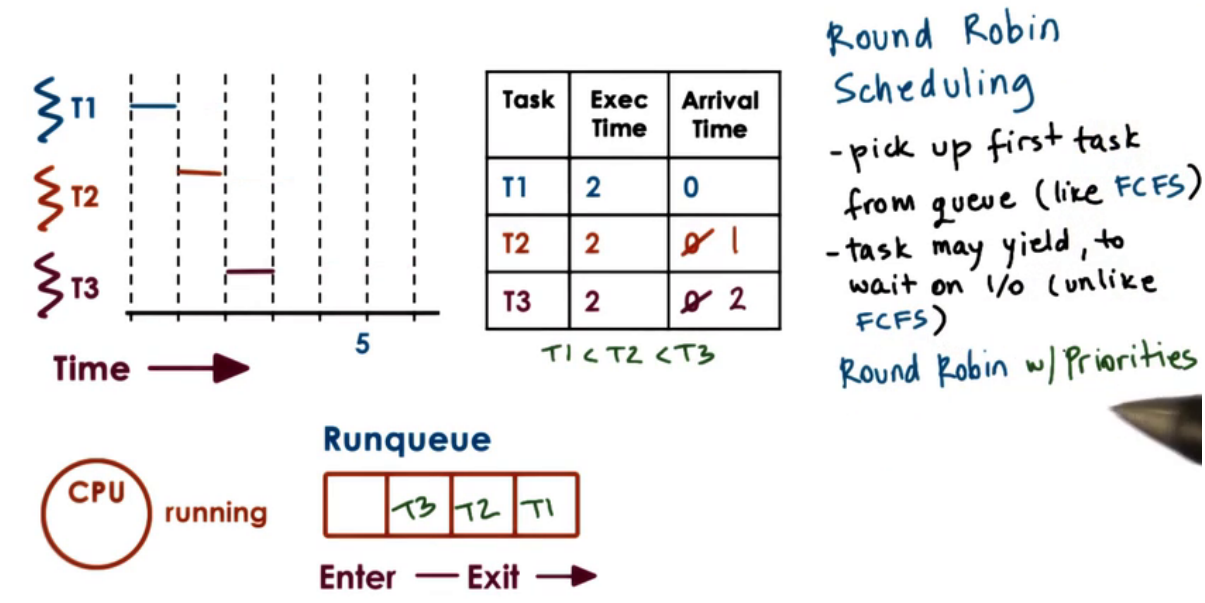
When it comes to running tasks that have the same priority. We have other options aside from the first come first server or shortest job first that we discussed so far. A popular option is so called Round Robin scheduling. So let's say we have the following scenario. Three tasks. With round robin scheduling we'll pick up the first task from the queue just like the first come first serve. So let's say we pick up T1 first. it's executing. After 1 time unit, the task stops because it waits on a IO operation. So it will yield the CPU and be blocked on that IO Operation. this is unlike what we saw in first come first serve where we were assuming that each of the tasks executes until it completes. If that's the case ,we'll schedule T2 and T3 will move to the front of the queue. Potentially, T1 will complete its IO operation it will be placed at the end of the queue behind T3. When T2 completes, T3 will be scheduled. When T3 completes, T1 will be picked from the queue. If T1 had not been waiting on IO, then the execution based on T1 T2 T3 the order in which they were placed on the queue would have

looked like this. Each of the tasks execute one by one in a round robin manner and the queue is traversed in a round robin manner one by one.





If T1 has not been blocked by I/O operation, the order of operation will follow the pattern in the runqueue.



We can further generalize round robin scheduling to include priority. In this case, when higher priority task arrives, the lower priority task will be preempted.

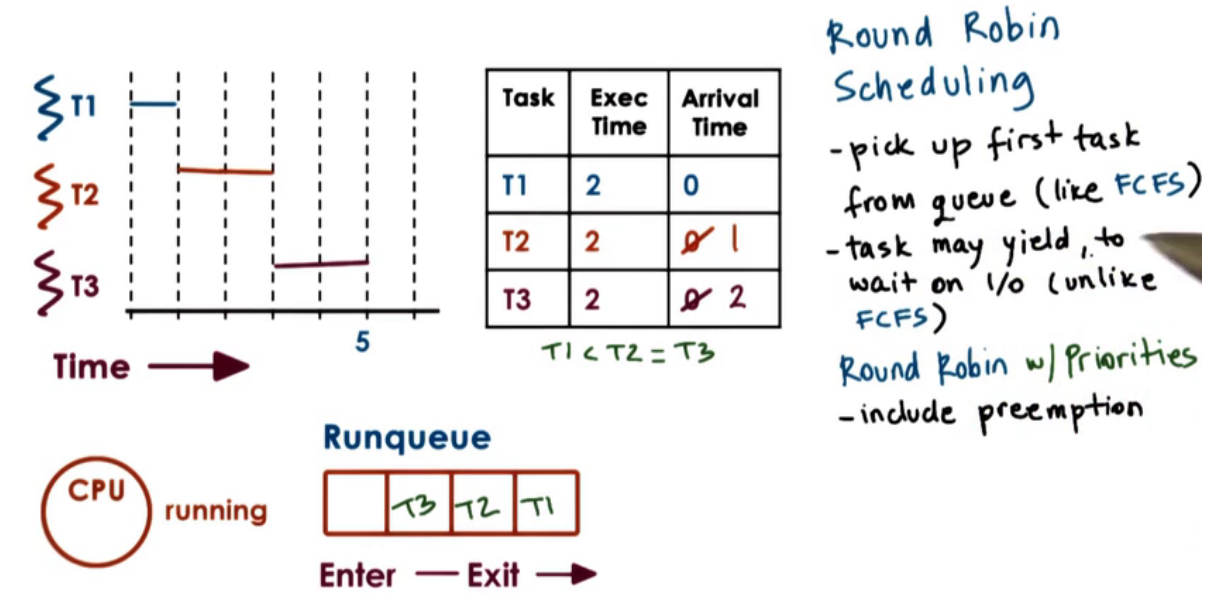
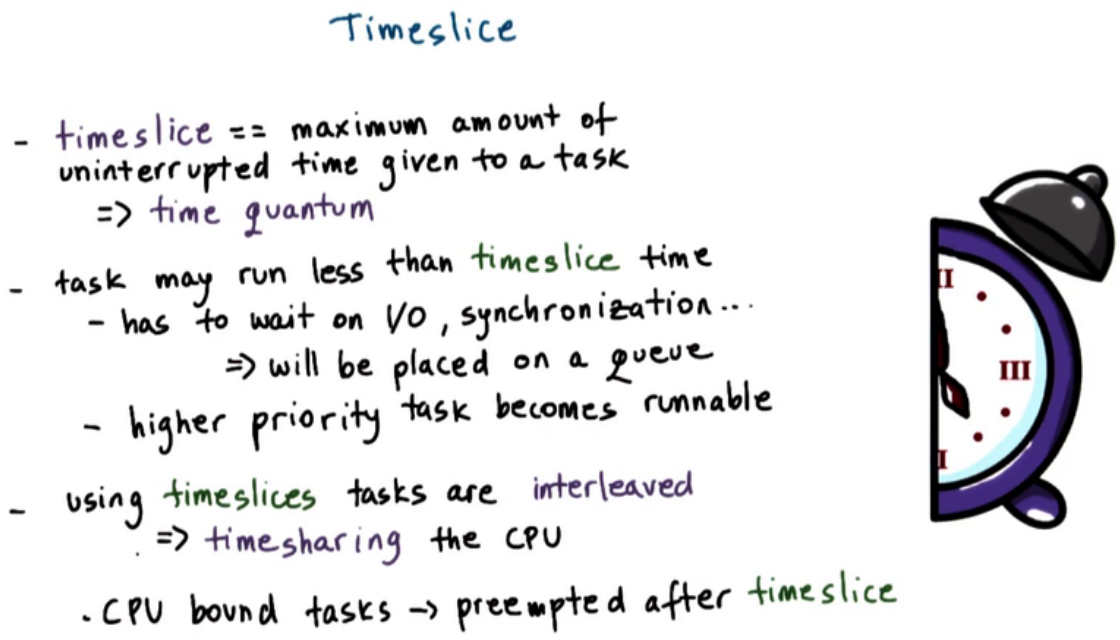


Image Errata: Above T1 is not shown with it's last second of execution which should take place after T3 completes.
If T2 and T3 has the same priority, the scheduler will run the round robin until the task is finished.

Further modification made for Round robin scheduling is not waiting tasks yield explicitly, instead to interrupt them by mix the tasks that are in the system at the same time. This mechanism is called time-slicing. For example, we can give each task a time slice of one time unit.

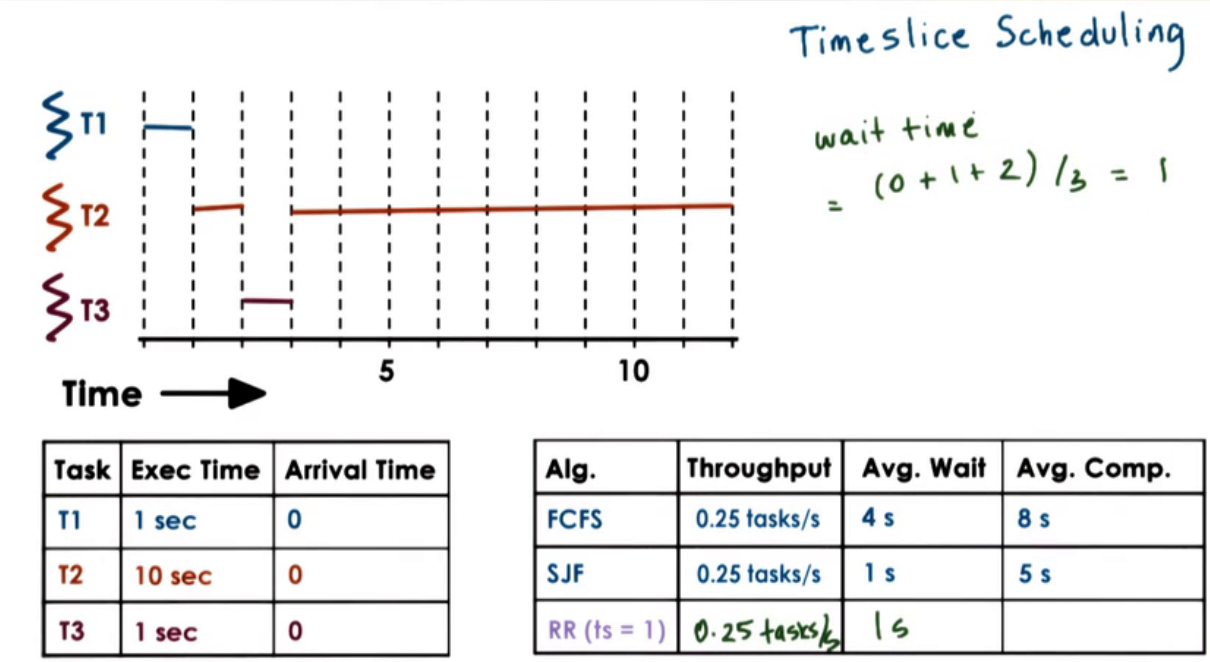
We can further generalize round robin to include priorities as well. Let's assume that the tasks don't arrive at the same time. T2 and T3 arrive later and that their priorities are as follows. In that case, what happens is that when a higher priority task arrives, the lower task will be preempted. If T2 and T3 have the same priorities, the scheduler will round robin between them until they complete. So basically in order to incorporate priorities we have to include preemption. But otherwise, the tasks will be scheduled from the queue like in FCFS so the first task from the queue however we will release the requirement that they have to control the CPU, that they have to execute on the CPU until they complete, instead they may explicitly yield. And we will just round robin among the tasks in the queue. A further modification that makes sense for round robin is to not wait for the tasks to yield explicitly but instead to interrupt them so as to mix in all of the tasks that are in the system at the same time. We call such mechanism time slicing. So let's say we can give each of the task a time slice of 1 time unit. And then after time unit, we will interrupt them, preempt them, and we will schedule the next task in the queue in round robin manner.

13 - Timesharing and Time-slices



We mentioned time slice very briefly in the introductory lesson processes. To define it more specifically, a time slice is the maximum amount of uninterrupted time that can be given to a task. It is

also referred to as a time quantum. The time slice determines the maximum amount. That means a task can actually run less amount of time than what the time slice specifies. For example, the task may have to wait on an IO operation or to synchronize with some other tasks in the system, on a mutex that's locked. In that case, the task will be placed on a queue and no longer be running on the CPU. the task will run less amount of time once it's placed on the queue. The scheduler will pick the next task to execute. Also if we have a priority based scheduling, a higher priority task will preempt a lower priority one which means the lower priority task will run less amount of time than the time slice. Regardless of what exactly happens, the use of time slices allows us to achieve for the tasks to be interleaved, they will be time-sharing the CPU. For IO bound tasks, this is not so critical, since they are constantly releasing the CPU to wait on some IO event. But for CPU bound tasks, time slice is the only event how we can achieve time sharing. They will be preempted after some amount of time specified by the time slice and we will schedule the next CPU bound task.

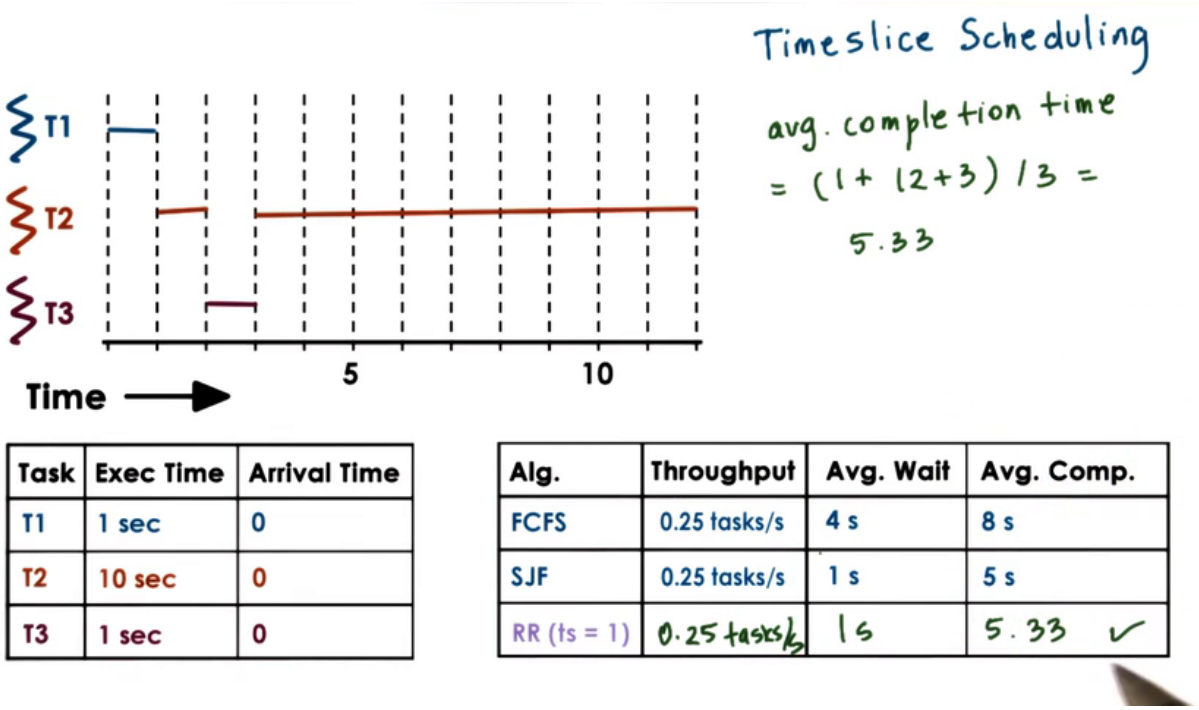


Task	Exec Time	Arrival Time
T1	1 sec	0
T2	10 sec	0
T3	1 sec	0

Alg.	Throughput	Avg. Wait	Avg. Comp.
FCFS	0.25 tasks/s	4 s	8 s
SJF	0.25 tasks/s	1 s	5 s
RR (ts = 1)	0.25 tasks/s	1 s	

Let's look at some examples now. Consider for instance the simple first come first serve and shortest job first schedulers that we saw earlier in this lesson. They both had a same mix of tasks with same arrival times but led to different metrics. And note that the metrics that we computed for FC FS also applied to a round robin scheduler that doesn't use time slices. Given that these tasks don't perform any IO, they just execute for some fixed specified amount of time, Round robin would have scheduled them one after another the way they showed up in the queue and that would have been identical to FCFS. Now let's compute the metrics for round robin scheduler with time slices. And let's say we'll first look at the time slice of 1. The execution of these task will look as follows. T1 will execute for 1 second. T1 completes. T2 will start but will be preempted after 1 second and T3 will run for a second. T3 will complete and T2 will continue running. If we look at some of the metrics, we'll have the same

numbers for throughput. Wait time of 1 s is the same.

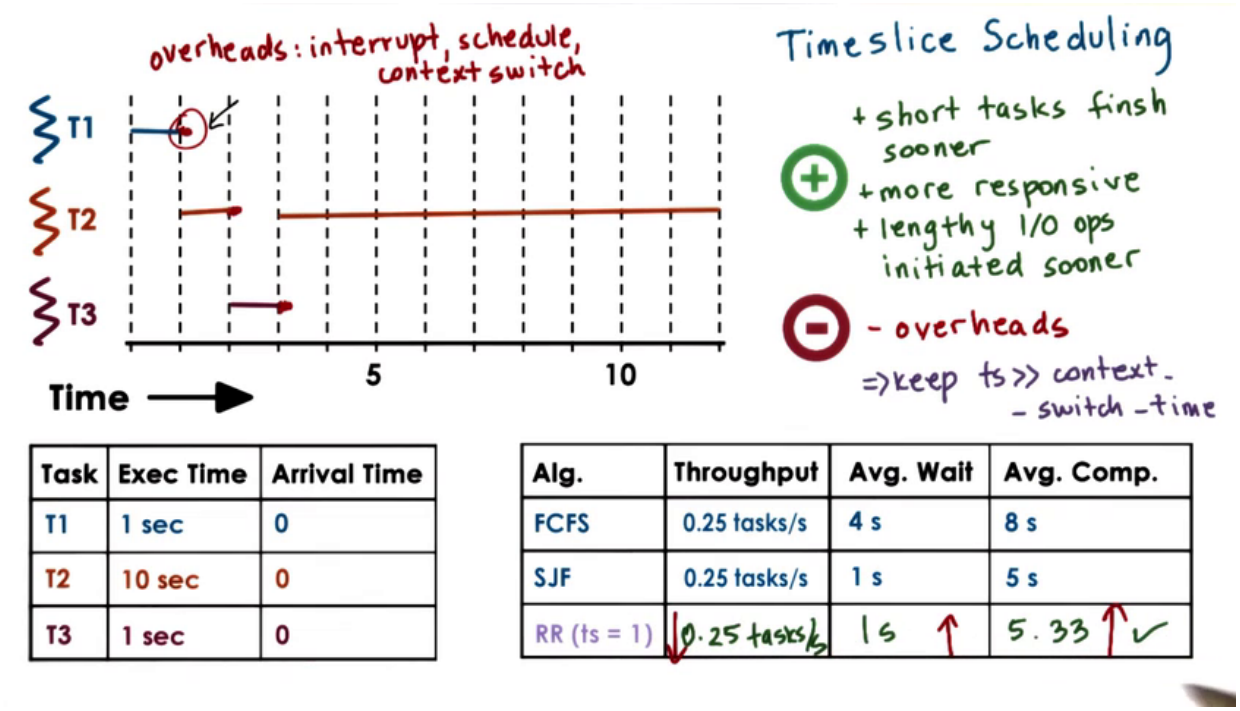


Task	Exec Time	Arrival Time
T1	1 sec	0
T2	10 sec	0
T3	1 sec	0

Alg.	Throughput	Avg. Wait	Avg. Comp.
FCFS	0.25 tasks/s	4 s	8 s
SJF	0.25 tasks/s	1 s	5 s
RR (ts = 1)	0.25 tasks/s	1 s	5.33 ✓

If we look at the average completion time, the tasks complete at 5.33 seconds. So without even knowing the execution times of these tasks, with a time slice of 1 we were able to achieve a schedule that's really close to this best one we saw before, the shortest job first one. This is good. We keep some of the simplicity that we had from FCFS. We don't need to worry about figuring out what is the execution time of the tasks and yet we were able to achieve good wait time and good completion time for all of the tasks. So some of the benefits of using the time slice method particularly when the time slice is relatively short like in this case, is that we end up with a situation where the short tasks finish sooner and we're also able to achieve a schedule that is more responsive and any IO operations can be executed and initiated as soon as possible. So for instance, consider if T2 is a task that the users interact with. It will be able to start as soon as possible...only 1 second into the execution the users will see that it is running and yet it will be pre-empted to squeeze in T3 at this point. If T2 needs to initiate any IO operations those will be initiated during this interval that it's running at this point. That would not have been the case with the shortest job first scheduler because T2 would have only been scheduled after all of the shorter jobs completed. The downside is that we exaggerated so far a little bit in that we have these tasks immediately starting their execution after the previous one was interrupted. However, there's some real overhead, we have to interrupt the running task, we have to run the scheduler, in order to pick which task to run next, and then we actually need to perform the context switch when we're scheduling from the context of 1 task to another. All of these times are pure overhead. There is no useful application processing that's done during those intervals. Note that these overheads, so if we have a time slice of one, these timeouts for the timer will appear during the execution of T2 except that at that point there are no other tasks in the system, we're not going to be scheduling or context switching and that's the dominant part of the overhead. And exactly how these ticks are handled, we're not going to discuss further in this class. The dominant sources of these overheads will impact the total execution of this timeline and increasing the time will cause the throughput to go down. Each of the tasks will also start just a little bit later. So the wait time will

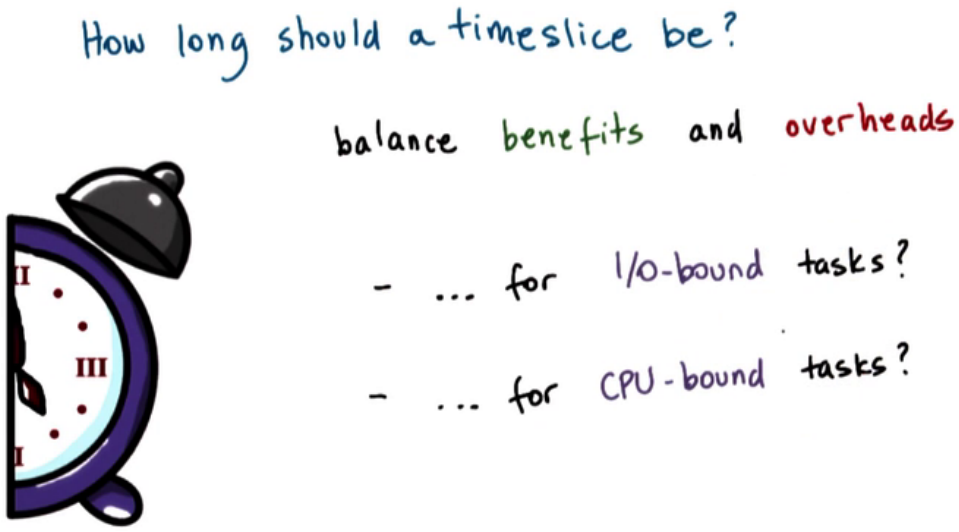
actually increase a little bit and the completion time for each of the tasks will be delayed by a little bit. So the average completion time will increase as well. The exact impact on these metrics will depend on the length of the time slice and how it relates to the actual time to perform these context switching and scheduling actions. So as long as the time slice values are significantly larger than the context switch time, we should be able to minimize these overheads. In general, we should consider both the nature of the tasks as well as the overheads in the system when determining meaningful values for the time slice.



Task	Exec Time	Arrival Time
T1	1 sec	0
T2	10 sec	0
T3	1 sec	0

Alg.	Throughput	Avg. Wait	Avg. Comp.
FCFS	0.25 tasks/s	4 s	8 s
SJF	0.25 tasks/s	1 s	5 s
RR (ts = 1)	0.25 tasks/s ↓	1 s ↑	5.33 ↑ ✓

14 - How Long Should a Timeslice Be



We saw in the previous, the use of time slice delivers certain benefits. We're able to start the execution of tasks sooner and therefore we are able to achieve an overall schedule of the task that's more responsive. But that came with certain overheads. So the balance between these two is going to imply something about the length of the time slice. We will see that to answer this question, how long should a time slice be? The balance between these two differs if we're considering IO bound tasks so tasks that mostly perform IO operations vs CPU bound tasks so tasks that are mostly executing on the CPU and don't do any IO. Do very little IO. We will look at these two scenarios next.

15 - CPU Bound Timeslice Length

Let's first look at an example in which we will consider two CPU bound tasks, so these are mostly running on the CPU and don't perform any IO. Let's assume their execution time is 10 seconds and in this system let's assume the time to context switch takes 0.1 seconds. For this scenario, let's take a look at what will happen when we consider 2 different time slice values. Time slice of 1 second and time slice 5 seconds. With a time slice of 1 second ,the execution of these two tasks will look something as follows. Let's assume that the thicker vertical bars are trying to capture this context switch overhead in this case. For the time slice value of 5 seconds, the schedule will look as follows. In the context switch overhead, it's only paid at these particular points. That's in contrast with having to context switch at every single one of these vertical bars here. If we compute the metrics throughput, avg wait time, and avg completion time for these tasks we will obtain the following results. To complete the throughput, we calculate the total time to execute these tasks and divide by 2. To complete the avg wait time, we look at the start time of each of the tasks and divide by 2. To complete the avg completion time we look at when each of the two tasks completed and then we average that. The detailed calculation for both of these are in the instructors notes. Looking at these metrics, for throughput we are better off choosing a higher time slice value. For completion time, we are better using higher time slice values, but for avg wait, we are better with a lower time slice value. However these are cpu bound tasks. We don't really care about their responsiveness and the user is not necessarily going to perceive when they started. the user really cares about when they complete and overall when all of the tasks submitteed to the system complete. So this really means for CPU bound tasks, we're really better off choosing a larger time slice. This is the winning combination. For CPU bound tasks, if we did no time slicing at all, we'd end up with absolutely the best performance in terms of throughput and the avg completion time. So the metrics we care for when we run CPU bound tasks. Yes, the avg wait time will be worse in that case, but we don't care about that because it's CPU bound. We won't notice that. So in summary, a CPU bound task prefers a larger time slice.

Winning combination

Full Calculations

- Timeslice = 1 second
    - throughput = 2 / (10 + 10 + 19*0.1) = 0.091 tasks/second
    - avg. wait time = (0 + (1+0.1)) / 2 = 0.55 seconds
    - avg. comp. time = 20.85 seconds
- Timeslice = 5 seconds
    - throughput = 2 / (10 + 10 + 3*0.1) = 0.098 tasks/second
    - avg. wait time = (0 + (5+0.1)) / 2 = 2.55 seconds
    - avg. comp. time = 17.75 seconds
- Timeslice = ∞
    - throughput = 2 / (10 + 10) = 0.1 tasks/second
    - avg. wait time = (0 + (10)) / 2 = 5 seconds
    - avg. comp. time = (10 + 20)/2 = 15 seconds

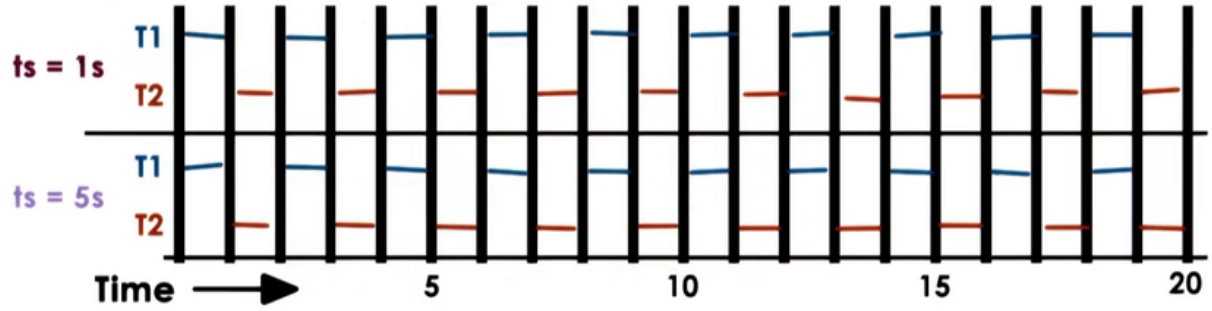
Errata: Some of the calculations are not correct.

- Timeslice = 1 second
    - avg. comp. time = ((19+18*0.1)+(20*19*0.1))/2 = 21.35 seconds
- Timeslice = 5 seconds
    - avg. wait time = (0 + (5+0.1)) / 2 = 2.55 seconds

For timeslice inf, it can include the context switch time as well, but I do not think it makes big difference.

- Timeslice = ∞
    - throughput = 2 / (10 + 10.1) = 0.0995 tasks/second
    - avg. wait time = (0 + (10.1)) / 2 = 5.05 seconds
    - avg. comp. time = (10 + 20.1)/2 = 15.05 seconds

16 - IO Bound Timeslice Length



I/O bound tasks
- 2 tasks, exec time = 10s
- ctx. switch time = 0.1s
- I/O ops issued every 1s
- I/O completes in 0.5s

Alg.	Throughput	Avg. Wait	Avg. Comp.
RR (ts = 1)	0.091 tasks/s	0.55 s	20.85 s
RR (ts = 5)	0.091 tasks/s	0.55 s	20.85 s

Now lets try to see what will happen if we're considering two io bound tasks and again we'll think of two tasks that have execution time of 10 seconds and in system that has a context switch overhead of 0.1 seconds. And let's also assume that the nature in which these IO calls are performed is that a task issues an IO operation every 1 second and let's also assume that every one of those IO operations complete in exactly half a second, If we look at the time line, it looks identical as what we saw in the case for the CPU bound jobs with a time slice of 1 second. This makes sense because exactly after 1 second the tasks are in this case not exactly preempted, they actually end issuing an IO operation so yielding voluntarily regardless of the fact that the time slice is 1. So if we look at the performance metrics for this case, they will look identical to the case of the CPU bound tasks. Now if we look at the second case where the timeslice value of 5 seconds, we see that the timeline..the schedules of the two tasks are identical to the case where we had a much smaller time slice value of 1 second. Similarly, if we compute the metrics. The reason for this is because at this particular moment, we aren't exactly time slicing. We aren't interrupting the task in either one of these cases. The IO operation again is issued every 1 second so regardless of the fact that in this scenario the time slice value is much longer, we still issue the IO operation at the end of the 1st second, and therefore the CPU is at this point released from T1, T1 yields, and T2 is free to start executing. So one conclusion is that for IO bound tasks, the value of the time slice is not really relevant. Well, let's not draw that conclusion that fast. Let's look first at a scenario, where only T2 is IO bound. T1 is a CPU bound task as we saw in the previous case. In that case the execution for the two tasks T1 and T2 when the time slice is 1. The one difference is in the event that T1 we preempted after 1 second. Whereas in the case of T2, the IO bound task after 1 second, it voluntarily yield since it has to go and wait for an IO. In the case of 5 seconds, the execution of T1 and T2 will look as follows. T1 will run for 5 seconds, and at that point it's time slice will expire. So it will be preempted. T2 will be scheduled and as an IO bound task it will yield after 1 second. At that point T1 will be scheduled again. T1 is actually going to complete and then T2 completes. If we work out the performance metrics for the last case, the numbers will look as follows. We see that both with respect to throughput and the average wait time, the use of a smaller time slice results in better performance. The only reason why in this case the average completion time is so low is because there was a huge

variance between the completion time of T1 and T2. We see from this, that for IO bound tasks, using a smaller time slice is better. The IO Bound task with a smaller time slice has a chance to run more quickly, to issue an IO request, to respond to a user, and with a smaller time slice we're able to keep both the CPU as well as the IO devices busy which makes obviously the system operator quite happy.

Full Calculation

- Timeslice = 5 second*
    - throughput = 2 / 24.3 = 0.082 tasks/second
    - avg. wait time = 5.1 / 2 = 2.55 seconds
    - avg. comp. time = (11.2 + 24.3) / 2 = 17.75 seconds

17 - Summarizing Timeslice Length

Let's summarize quickly how long should time slice be?

CPU bound tasks prefer longer time slices. The longer the time slice, the fewer context switches that need to be performed. So that basically limits the context switching overhead that the scheduling will introduce.

Prior to perform useful work, the useful application processing to as low as possible and as a result both the CPU utilization and system throughput will be as high as possible.

On the other hand IO bound tasks prefer shorter time slices. This allows IO bound tasks to issue IO operations as soon as possible and as a result we achieve both higher CPU and device utilization as well as the performance that the user perceives that the system is responsive and that it responds to its commands and displays output.

18 - Timeslice Quiz

Quiz Help

CPU Utilization Formula:

- [cpu_running_time / (cpu_running_time + context_switching_overheads)] * 100
- The cpu_running_time and context_switching_overheads **should be calculated over a consistent, recurring interval**

Helpful Steps to Solve:

1. Determine a consistent, recurring interval
    - In the interval, each task should be given an opportunity to run
2. During that interval, how much time is spent computing? This is the **cpu_running_time**
3. During that interval, how much time is spent context switching? This is the **context_switching_overheads**
4. Calculate!

19 - Timeslice Quiz

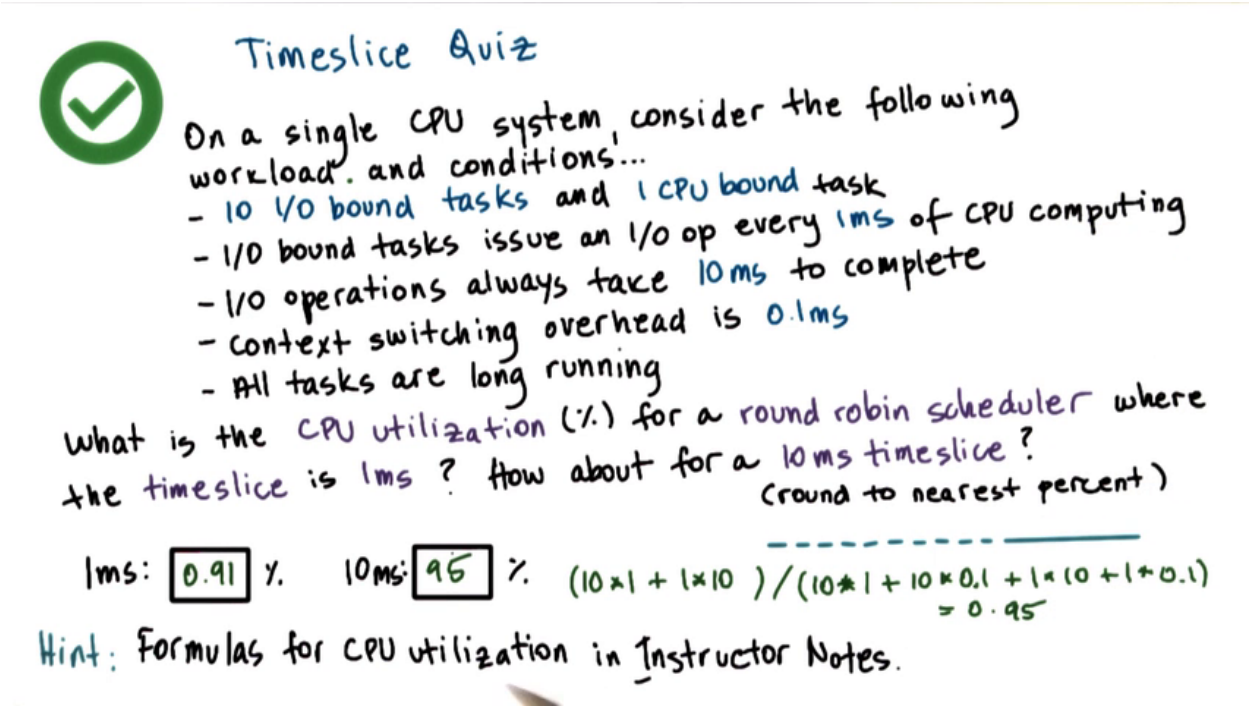


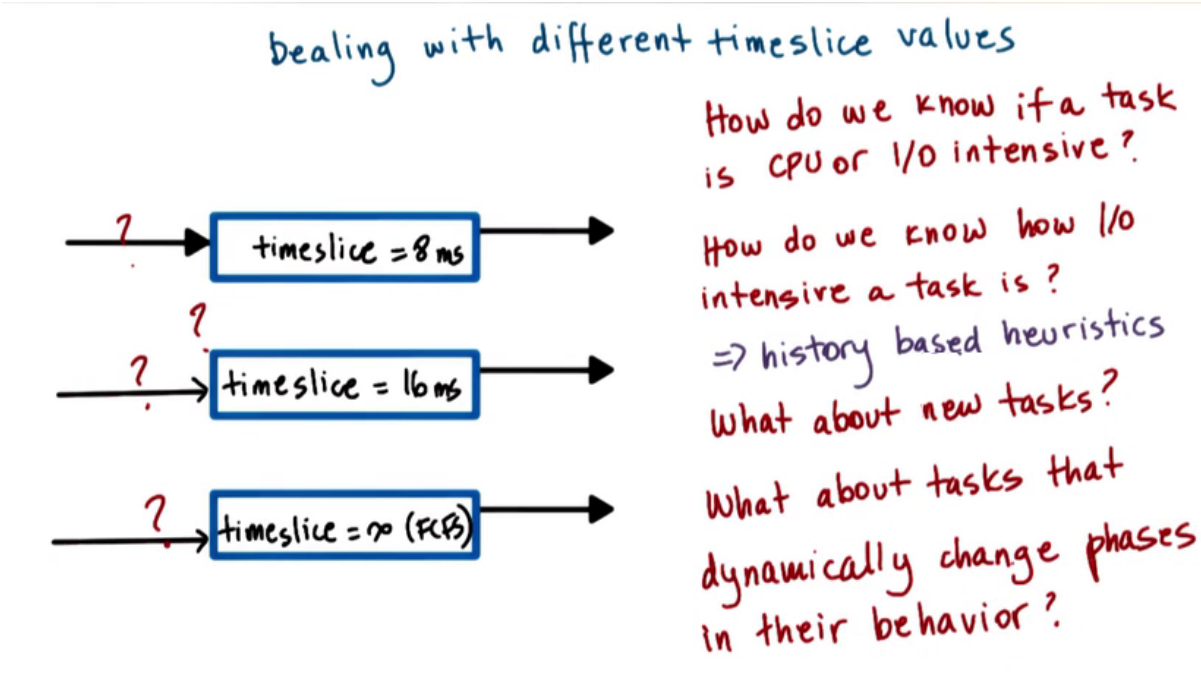
Image Errata: The left box should be 91%, not 0.91%.

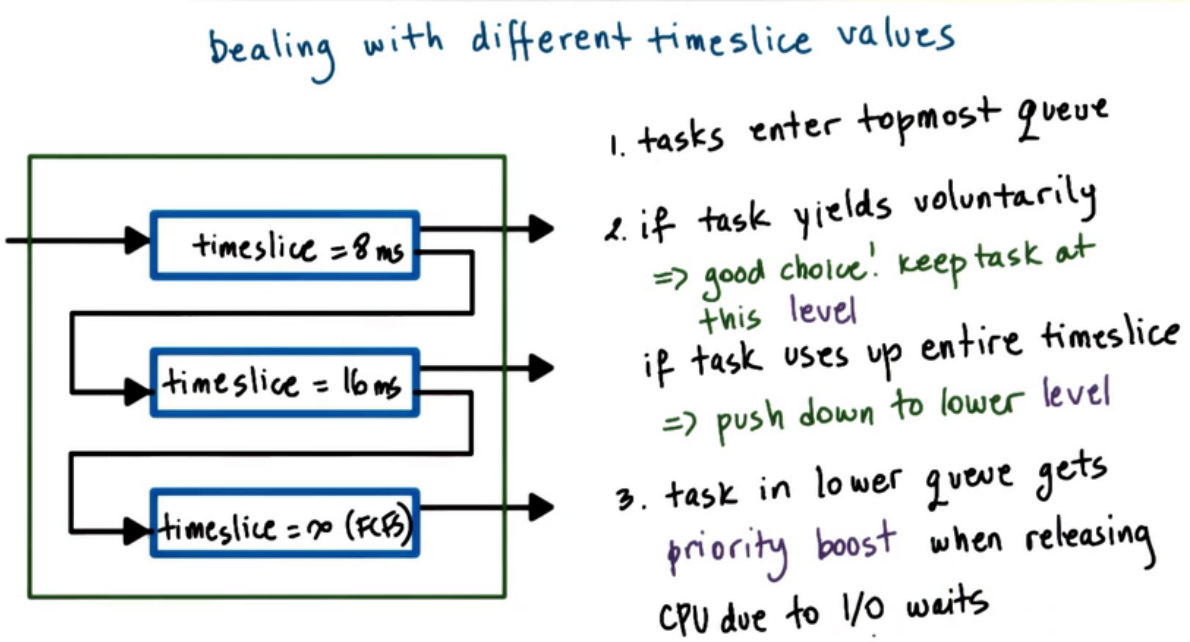
20 - Runqueue Data Structure

We said earlier that the run queue is only logically a queue. It could also be represented by multiple queues like when we are dealing with different priorities or it could be a tree or some other type of data structure. Regardless of the data structure, it should be easy for the scheduler to find the next thread to run given the scheduling criteria. If we want the IO and CPU tasks in the system to have different time slice values, we have 2 options. The first option is to maintain a single run queue structure but to make it easy for the scheduler to figure out easy what type of task is being scheduled so that it can apply the different policy. Another option is to completely separate IO and CPU bound tasks into two different data structures, two different run queues, and then each run queue associate a different kind of policy that's suitable for CPU vs. IO bound tasks. One solution for this is this type of data structure that we'll explain now.

Is a multi queue data structure that internally it has multiple separate queues. Let's say in the 1st run queue we'll place the most IO intense tasks, and we'll assign this run queue a time slice of 8 ms. Let's say for tasks that have medium IO, we have a separate queue in this multi queue data structure and here we'll assign with this queue a time slice of 16ms. And then for all of our CPU intensive tasks we'll use another queue and here we'll associate with the time slice that's infinite. It will be like first come first serve policy. From the scheduler's perspective, the IO intensive tasks have the highest priorities which means the scheduler will always check on this queue first. And the CPU bound tasks will be treated with tasks at lowest priorities so this queue will be the last one checked when trying to figure out what is the next task that needs to be scheduled. So depending on the type of task that we have, we place it on the appropriate queue and on the other side the scheduler selects which tasks to run next based on highest priority, medium, and then lowest. So in this way, we both provide both the time slicing benefits for those tasks that benefit for the IO bound tasks and avoid the time slicing overhead for the CPU bound tasks. How do we know if a task is CPU or IO intensive. How do we know how IO intensive is a task. Now we can use for that some history based heuristics like let this task and then decide what to do with it. Sort of like what we explained with the shortest job first algorithm but what about new tasks, what about tasks that have dynamic changes in their behavior. To deal with those problems, we will treat these three queues not as 3 separate run queues but as one single multi queue data structure. This is how this data structure will be used. When a task first enters the system, we'll enter it into the top most queue. The one with the lowest time slice. The most demanding task when it comes to the scheduling operations. It will need to be context switched most often. If the task stops executing before these 8ms, if it yields voluntarily, or it stops because it needs to wait on an IO operation that means we made a good choice. The task is indeed fairly IO interactive so we want to keep the task in this level. So next time around, after that IO operation completes, it will be placed in this exact same queue. However, if the task uses the entire time slice, then that means that it was actually more cpu intensive than we originally thought so we will push it down 1 level. It will be preempted from top level, but the next run it will be scheduled from the second queue. If the task gets preempted from the second queue after using it's entire 16 ms time slice, that

means it's even more cpu intensive so it will get pushed down to the bottom queue. So we basically have a mechanism to push the task down these levels based on it's historic information. Although we didn't know if a task was IO or CPU intensive to start with, we made an assumption and then we were able to correct for it. We assumed it was IO intensive and we were able to correct and push it down to the lowest most level in case it was CPU intensive. Now if a task that's in the lower queue and all of a sudden starts getting repeatedly releasing the CPU earlier than whatever the time slice specified because it was waiting on IO operation, that will be a hint to the scheduler to say oh well this task is more IO intensive than I originally thought and it can push it up one of the queues on one of the higher levels. This resulting data structure is called the **multi-level feedback queue a**nd for the design of this data structure along with other work on time sharing system Fernando Corbato received the Turing Award which is the highest award for Computer Science. We shouldn't trivialize the contribution of this data structure and say that it's equivalent to priority queues. First of all there are different scheduling policies that are associated with each of these different levels that are part of this data structures. More uniquely however, this data structure incorporates this feedback mechanism that allows us over time to adjust which one of these levels will we place a task and when we're trying to figure out what is the best time sharing schedule for the subtasks in the system. The Linux so called O of one scheduler we will talk about next uses some of the mechanisms borrowed from this data structure as well and we won't describe the solaris scheduling mechanism, but I just want to mention that it's pretty much a multi level feedback queue with 60 levels and also some things feedback rules that determine how and when a thread gets pushed up and down these different levels.





MLFQ = Multi-Level Feedback Queue
Linux uses O(1) Scheduler, Solaris uses MLFQ with 60 levels (subqueues) with fancy feedback rules.

21 - Linux O1 Scheduler

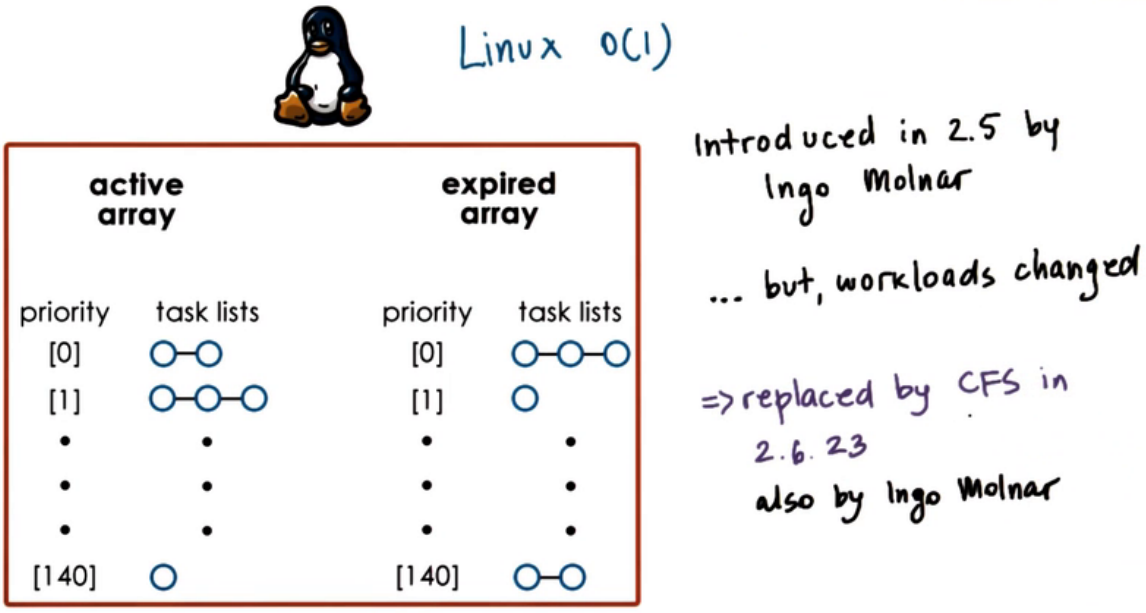
Image Errata: Range of values in image does not correspond with what is written to the right. 0 - 140 should be 0 - 139 for the above and next three images.

Let's now look at concrete examples of schedulers that are part of an actual operating system. First we will look at the O(1) (pronounced "O of 1") scheduler in linux. The O(1) scheduler receives its name because it's able to perform task management operations such as selecting a task from the run queue or adding a task to it in constant time. Regardless of the total number of active tasks in the system. It's a preemptive and priority based scheduler which has a total of 140 priority levels. With 0 being the highest. 139 is the lowest. The priority levels can be grouped into two classes. The task with priority levels from 0 to 99 falls into a class of real time tasks. All others fall into a time sharing class. All user processes have one of the time sharing priority levels. The default priority is 120 but it can be adjusted with a so called nice value. The nice values range from -20 to 19. The O(1) scheduler borrows from the multi-level feedback queue scheduler in that it associates different time slice values with different priority levels and it also uses some kind of feedback from how the task behaves in the past to determine how to adjust their priority levels in the future. It differs however in how it assigns the time slice values to priorities and how it uses the feedback. It assigns time slice values based on the priority level of the task similar to what we saw in the multilevel feedback queue scheduling however it assigns small time slice values to the low priority cpu bound task and it assigns high time slice values to the more interactive high priority tasks. The feedback it uses for the time-sharing tasks is based on the time that task spent sleeping. The time that it was waiting for something or idling. Longer sleep times indicate that the task is interactive. it spent more time waiting on user input or some type of event. therefore when longer sleeps have been detected, we need to increase the priority of the task and we do that by actually subtracting 5 from that particular priority level of the task. We are essentially boosting the priority so next time around this interactive task will execute with higher priority. Smaller sleep times are indicative of the fact that the task is compute intensive, so we want to lower its priority by incrementing it by 5. So next time the task will execute in

a lower priority class.

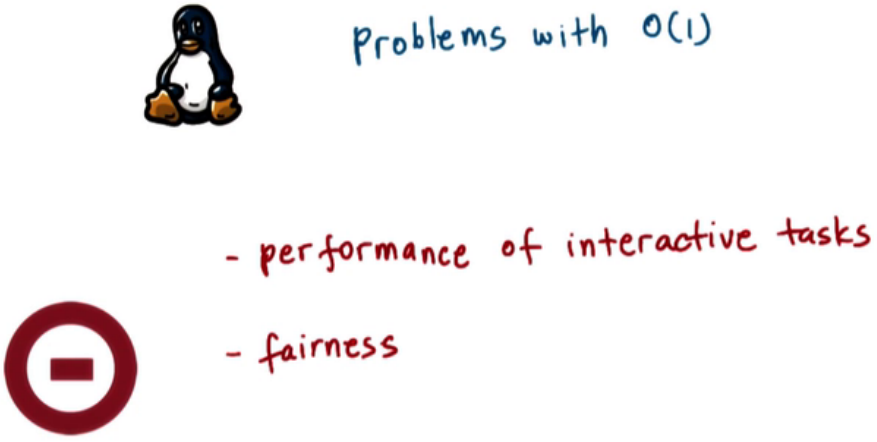
The run queue in the o(1) scheduler is organized as 2 arrays of task queues. Each array element points to the first runnable task at the corresponding priority level. These two arrays are called active and expired. The active list is the primary one that the scheduler uses to pick the next task to run. It takes constant time to add a task since you simply need to index into this array based on this priority level of the task and then follow the pointer to end of the task list to enqueue the task there. It takes constant time to select a task because the scheduler relies on certain instructions that return the

position of the first set bit in a sequence of bits. So if the sequence of bits corresponds to the priority levels and a bit value of 1 indicates that there are tasks at that priority level, then it will take a constant amount of time to run those instructions to detect what is the first priority level that has certain tasks on it. And then once that position is known, it also takes a constant time to index into this array and select the first task from the run queue that's associated with that level. If tasks yield the cpu to wait on an event or are preempted due to higher priority tasks becoming runnable the time spent on the CPU is subtracted from the total amount of time and if it's less than the time slice they are still placed on the corresponding queue in the active list. Only after a task consumes its entire time slice will it be removed from the active list and placed on the appropriate queue in the expired array. The expired array, that contains the task that are not currently active in the sense that the scheduler will not select tasks from the expired arrathrify as long as there's still tasks on any of the queues on the active array. When there are no more tasks on the active array, at that point the pointers of these two lists will be swapped and the expired array will become the new active one and vice versa. The active array will start holding all of the tasks that are removed from the active array and are becoming inactive. This also explains the rationale why in the O(1) scheduler the low priority tasks are given the low time slices and the high priority tasks are given high time slices. As long as the high priority tasks have any time left in their time slice, they will keep getting scheduled. They will remain in one of the queues in the active array. Once they get placed on the expired array they will not be scheduled and therefore we want the low priority tasks to have a low time slice value so that yes they will have a chance to run, but they won't disrupt the higher priority tasks, they won't delay them by too much. Also know that the fact that we have these 2 arrays also serves like an aging mechanism. So these high priority tasks will ultimately consume their time slice, be placed on the expired array, and ultimately the low priority tasks will get a chance to run for their small time amount. The O(1) was introduced in the linux kernel 2.5 by Ingo Molnar. In spite of its really nice property of being able to operate in constant time, the O(1) really affected the performance of interactive tasks significantly. As the workloads changed, as typical applications are becoming more time sensitive (skype, movie streaming, gaming) the jitter that was introduced by the O(1) scheduler was becoming unacceptable. For that reason, the O(1) scheduler was replaced with the completely fair scheduler. The CFS became the default scheduler in kernel 2.6.23. Ironically, both of these schedulers are developed by the same person. Both the O(1) and CFS scheduler are part of the standard linux distribution. CFS is the default. But you can switch to O(1) scheduler to execute your tasks.



Jitter caused by O(1) scheduler was unacceptable, replaced with CFS (Completely Fair Scheduler).

22 - Linux CFS Scheduler

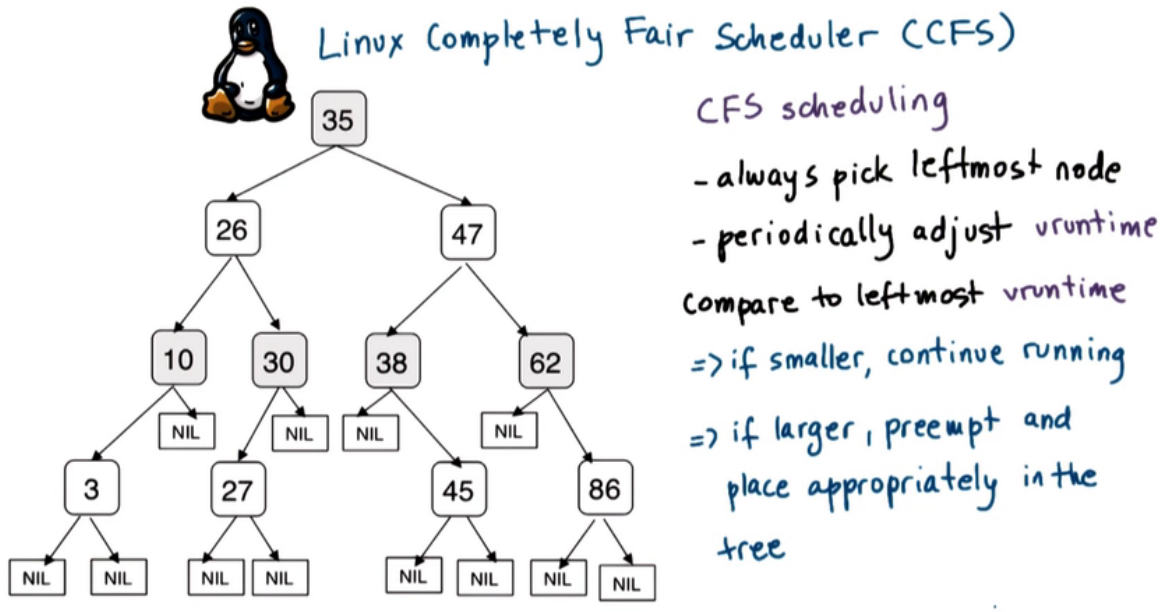


Fairness - in a given time interval, all of the tasks should be able to run for an amount of time that is proportional to their priority.

CFS == default since 2.6.23 (Ingo Molnar)

Runqueue == Red-Black Tree
- ordered by "vruntime"
- vruntime == time spent on CPU

As we said, one problem with the O(1) scheduler in Linux is that once tasks are placed on the expired list, they won't be scheduled until all remaining tasks from the active list have a chance to execute for whatever their time slice amount of time is. As a result the performance of interactive tasks is affected. There's a lot of jitter. In addition, the scheduler in general doesn't make any fairness guarantees. There are multiple formal definitions of fairness but intuitively you can think of it that in a given time interval, all of the tasks should be able to run for an amount of time that is proportional to their priority. And for the O(1) scheduler, it's really hard to make any claims that it makes some kind of fairness guarantees.

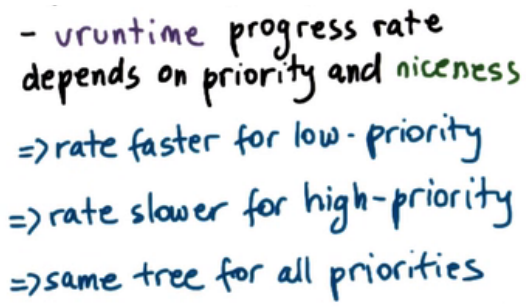


Instructor Notes

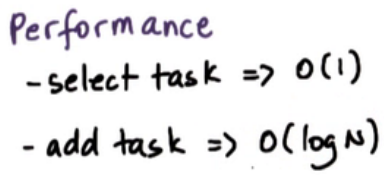
If you need more information about the red-black tree data structure, then see this red-black tree explanation.

As we said, Ingo Molnar proposed the completely fair scheduler (CFS) to address the problems with the O(1) scheduler and CFS is the default scheduler in Linux since the 2.6.23 kernel. It's the default scheduler for all of the non real-time tasks, the real time tasks are scheduled by a real-time scheduler. The main idea behind the CFS is that it uses a so called red-black tree as a run queue structure. Red-black tree belong to this family of dynamic tree structures that have a property that as nodes are added or removed from the tree, the tree will self balance itself so that all the paths from the root to the leaves of the tree are approximately the same size. You can look at the instructor notes for a link for more information about this type of data structure. Tasks are ordered in the tree based on the amount of time they spend running on the CPU and that's called virtual runtime. CFS tracks this virtual runtime in a nanosecond granularity. As we can see in this figure, each of the internal nodes in the tree corresponds to a task and the nodes to the left of the task correspond to those tasks which had less time on the CPU, they had spent less virtual time and therefore they need to be scheduled sooner. The children to the right of a node are those that have consumed more virtual time, more cpu time, and therefore they don't have to be scheduled as quickly as the other ones. The leaves in the

tree really don't play any role in the scheduler. The CFS scheduling algorithm can be summarized as follows. CFS always schedules the task which had the least amount of time on the CPU so that typically would be the leftmost node in the tree.

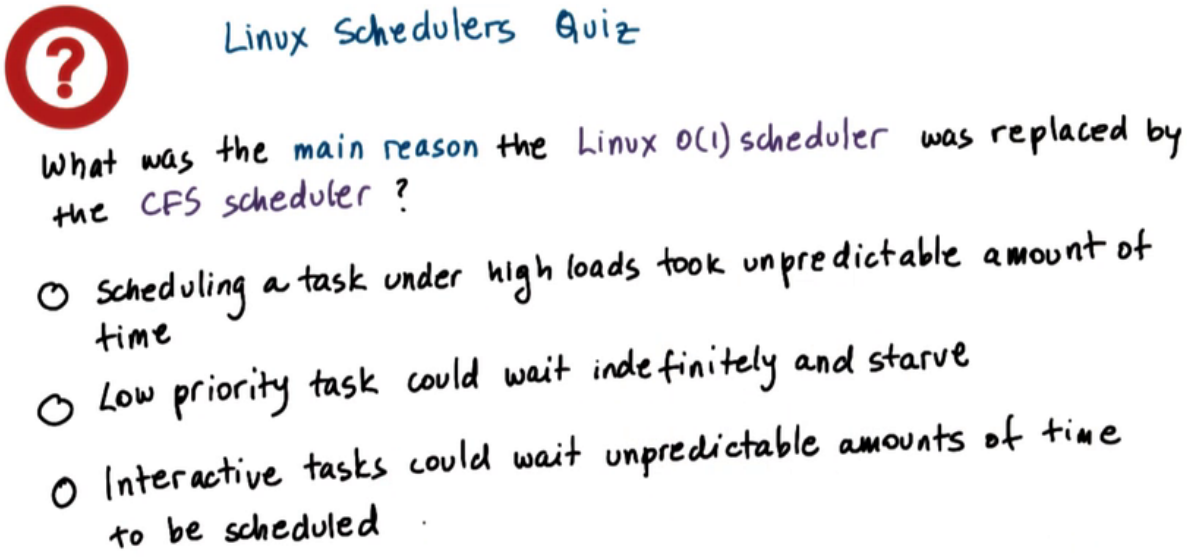


Periodically, CFS will increment the vruntime of the task that's currently executing on the CPU and at that point, it will compare the runtime of the currently running task to the vruntime of the leftmost task in the tree. If the currently running task has a smaller vruntime compared to the one that's in the leftmost node in the tree, the currently running task will continue executing. Otherwise, it will be preempted and it will be placed in the appropriate location in the tree. Obviously the task that corresponds to the left most node in the tree will be selected to be run next. To account for differences in the task priorities or in their niceness value, CFS changes the effective rate at which the tasks virtualtime progresses. For lower priority tasks, time passes more quickly at their virtual runtime value progresses faster and therefore they will likely lose their cpu more quickly because their virtual runtime will increase compared to other tasks in the system. On the contrary, for high priority tasks, time passes more slowly. Virtual runtime values will progress at a much slower rate and therefore they will get to execute on the CPU longer. You should take note of the fact that CFS uses really one runqueue structure for all of the priority levels unlike what we saw with some of the other scheduler examples.

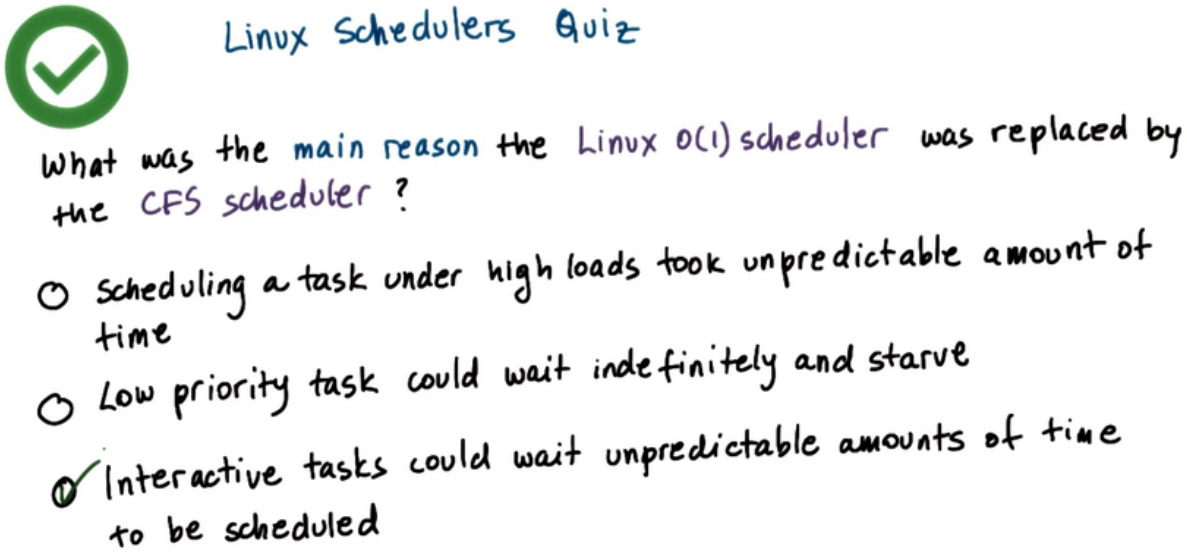


In summary, selecting a task from this run queue to execute takes O(1) time, takes constant amount of time since it's typically just a matter of selecting the leftmost node in the tree. At the same time, adding a task to the run queue takes logarithmic time relative to the total number of tasks in the system. Given the typical levels of load in current systems, this log(n) time is acceptable however as the computed capacity of the nodes continues to increase and systems are able to support more and more tasks, it is possible that at some point the CFS scheduler will be replaced by something else that will be able to perform better when it comes to this second performance criteria.

23 - Linux Schedulers Quiz



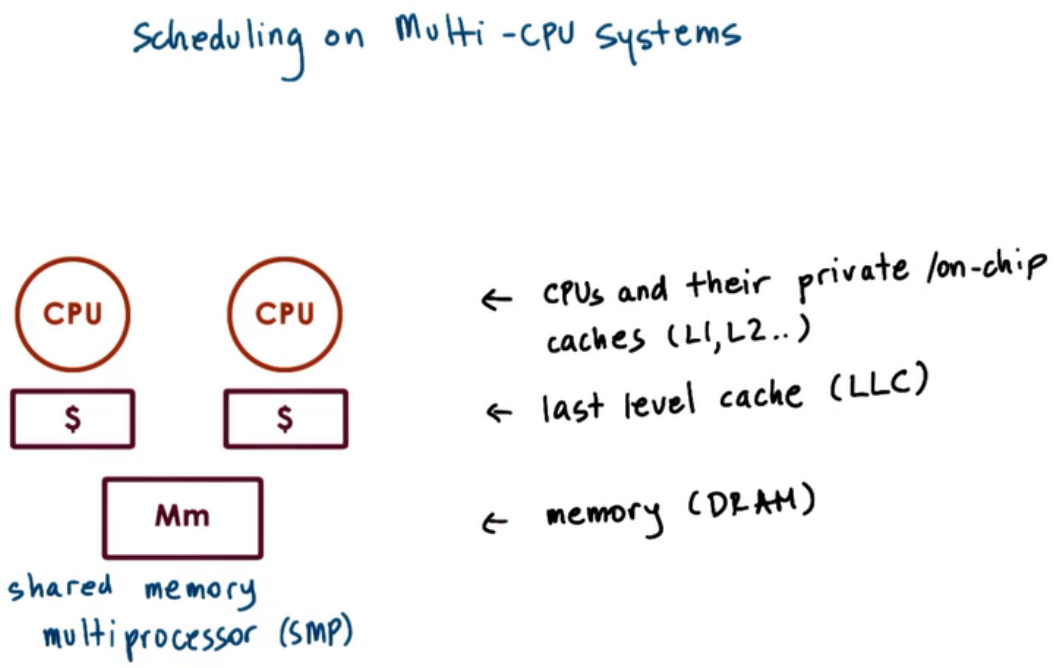
24 - Linux Schedulers Quiz



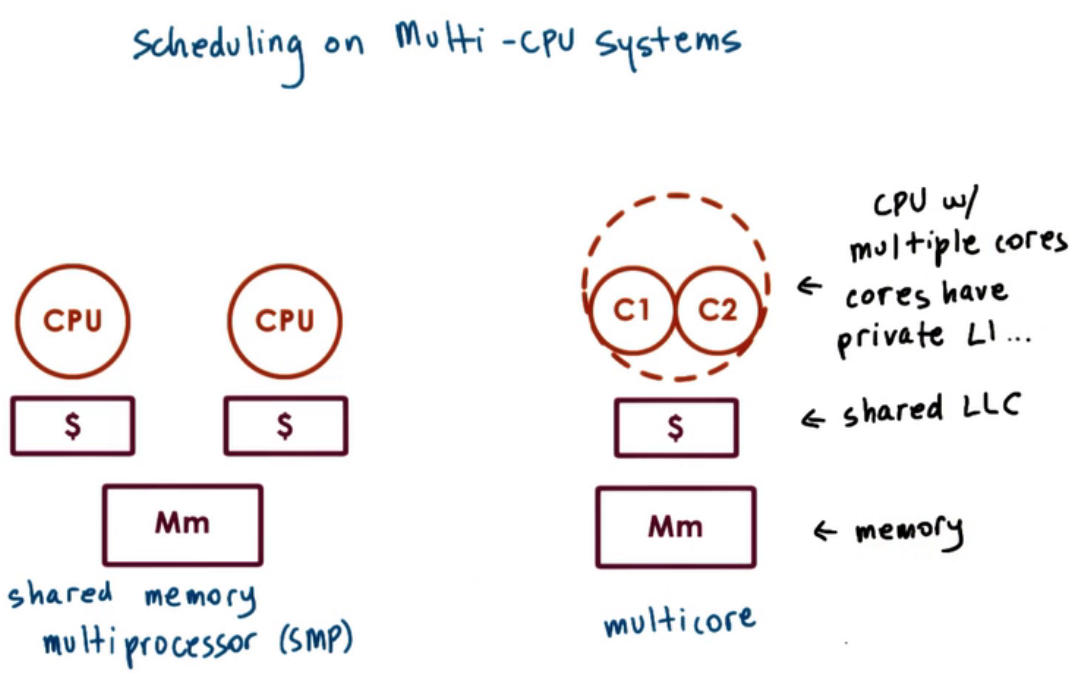
As a review, I would like to ask a question about the 2 linux schedulers that we just discussed. What was the main reason the Linux O(1) scheduler was replaced by the CFS scheduler. Was it because scheduling of tasks under high loads took unpredictable amount of time, lower priority task could wait indefinitely and starve, or because interactive tasks could wait unpredictable amounts of time to be scheduled. Select the appropriate answer. Let's take a look at the choices. The first is not correct. The linux O(1) scheduler was O(1) because it took a constant amount of time to select and schedule a task regardless of the load. The 2nd statement is sort of correct in the sense that as long as there

were continuously arriving higher priority tasks, it was possible for the low priority tasks to continue waiting an unpredictable amount of time and possibly indefinitely and therefore starve, but this was not the main reason the scheduler was replaced. The final choice was the main reason. Common workloads were becoming much more interactive and demanding high predictability. In the O(1) scheduler, with the active and expired lists once a task was moved to the expired list, it had to wait there until all of the low priority tasks consumed their entire time quantum. For a very long Linus Torvald resisted integrating a scheduler that would address the needs of these more interactive tasks in the linux kernel. His rationale was that linux was supposed to be a general purpose operating system and should not necessarily be addressing any of the needs of some more real time or more interactive tasks and therefore he liked the simplicity of the O(1) scheduler however as the general purpose workloads began to change, then a general purpose operating system like Linux, had to incorporate a scheduler that would address the needs of those general purpose workloads and CFS was really meeting those needs.

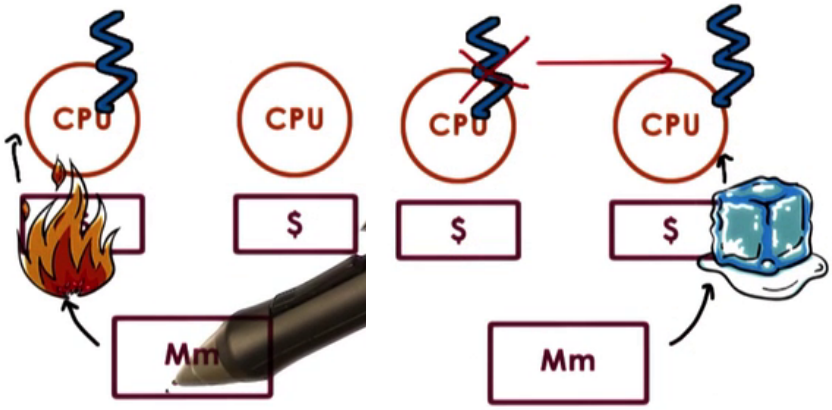
25 - Scheduling on Multiprocessors



Let's look at scheduling on multi cpu systems. However, before we start talking about scheduling let's look at some architecture details. First we will look at shared memory multiprocessors and then we'll take a look at how this compares to multi-core architectures. In a shared memory multiprocessors (SMP), there are multiple CPU's, each of them have their maybe own private caches like L1 and L2, their last level caches that may or may not be shared among the CPU's and there is a system memory DRAM that is shared across all of the CPU's. Here we show just one memory component but it's possible that there are multiple memory components. The point is that all of the memory in the system is shared among all of the cpu's.

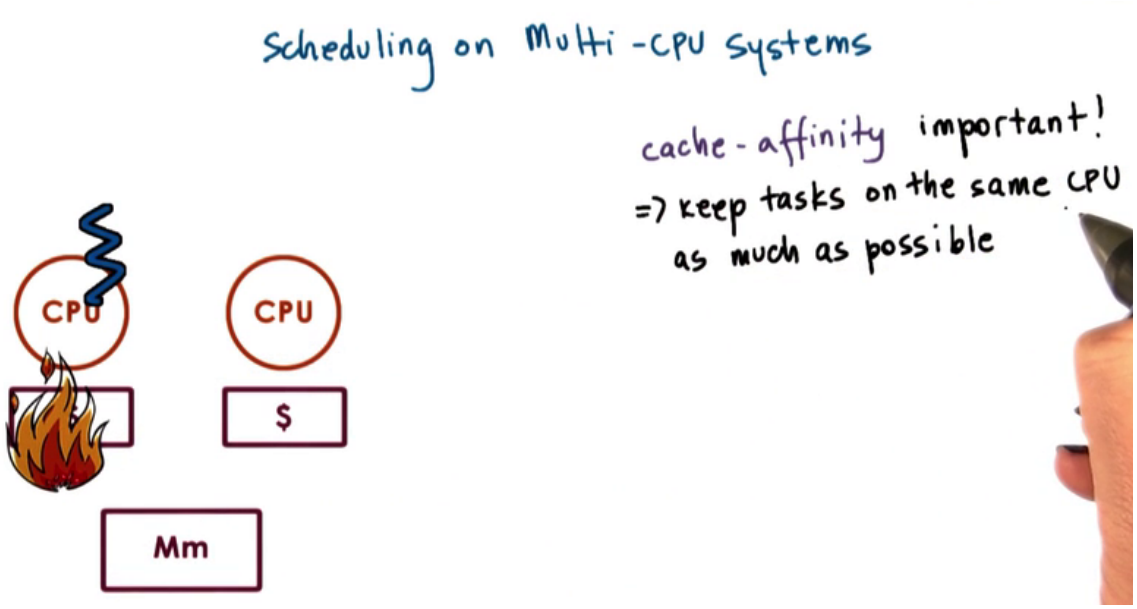


In the current multicore world, each of these cpu's can have multiple internal cores. Each of the cores will have private caches and overall the entire multicore cpu will have some shared last level cache and then again there will be some shared system memory. Here in this picture, we have a CPU with 2 cores so that's a dual core processor and this is more common for client devices like laptops for instance or even cell phones today can have two CPU's whereas on the servers, it's more common to have 6 or 8 cores and to have multiple such CPU's. As far as the OS is concerned it sees all of the CPU's as well as the cores in a CPU as entities onto which it can schedule execution contexts (threads). All of these are (as far as the OS is concerned) possible CPU's where it can schedule some of its workload. To make our discussion more concrete. We will first talk about scheduling on multi-cpu system in the context of SMP systems and a lot of these things will apply to the multi-core world b/c again the scheduler just sees the cores as CPU's. And we'll make some comments that more exclusively apply to multi core towards the end this lesson.

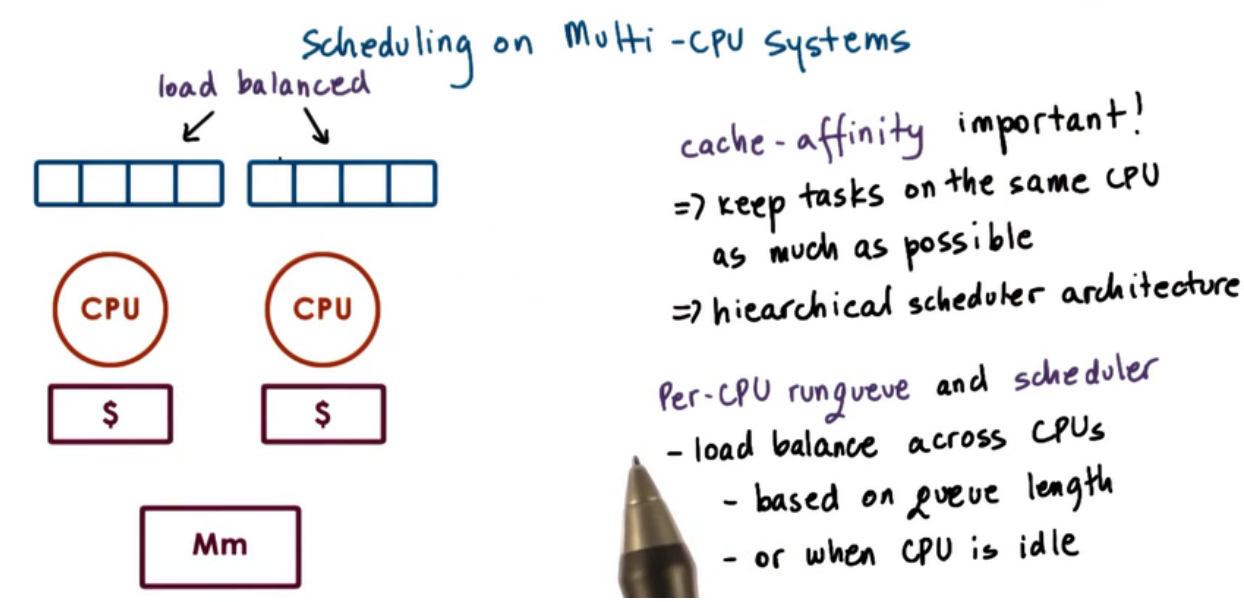


We said in our earlier lectures that the performance of threads and processes is highly dependent on whether the state that the thread needs is in the cache or in memory. Let's say a thread was

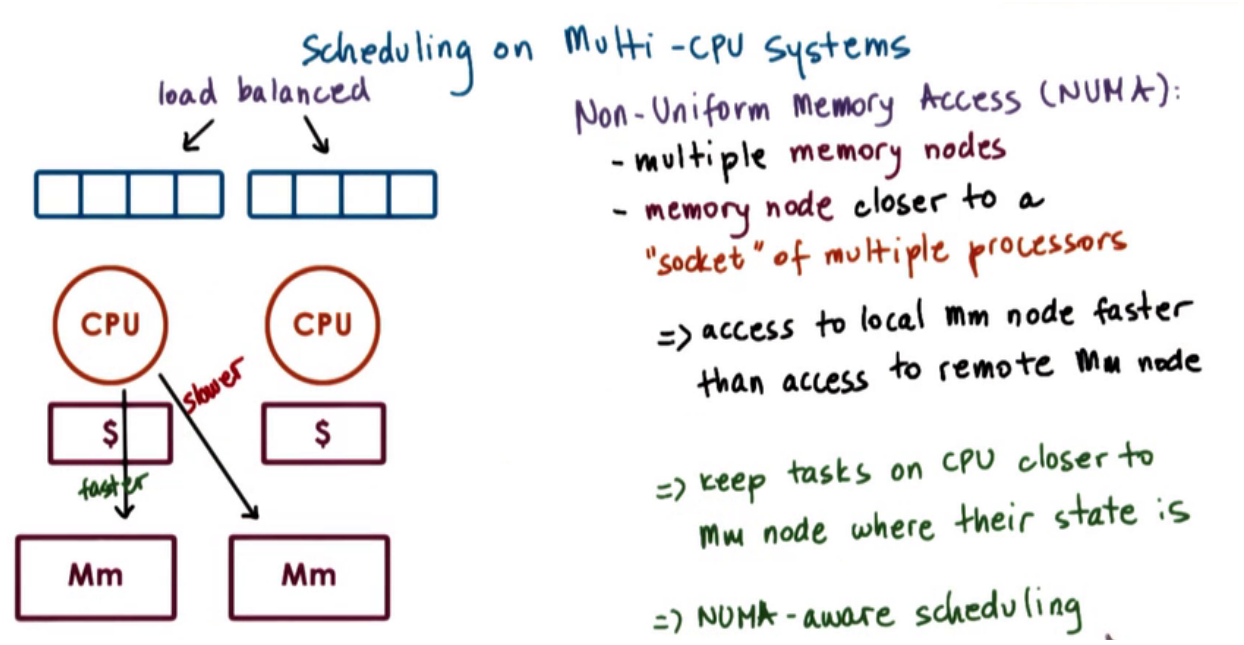
executing on cpu 1 first. Over time, this thread was likely to bring a lot of the state that it needs both into the last level cache that's associated with this cpu as well as in the private caches that is available on the CPU. In this case, when the caches are hot, this helps with the performance. Now the next time around, if the thread is scheduled to execute on the other cpu, none of it's state will be there, so the thread will operate with a cold cache, will have to bring all of the state back in, and that will affect performance.



Therefore what we want to achieve with a scheduling on multi-cpu systems is to try to schedule the thread back on the same cpu where it executed before because it is more likely that it's cache will be hot. We call this cache affinity. And that is clearly important. To achieve this, we basically want the scheduler to keep a task on the same cpu as much as possible. To achieve this, we can maintain a hierarchical scheduler architecture where at the top level, a load balancing component divides the task amongst the cpu's and then a per cpu scheduler and then a per cpu scheduler with a per cpu run queue repeatedly schedules those tasks on a given cpu as much as possible.



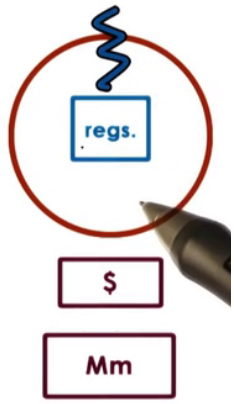
To balance the load across the different cpu's and their per cpu run queues, the top level entity in the scheduler can look at information such as the length of each of these queues to decide how to balance task across them. Or potentially when a cpu is idle it can at that point start looking at the other cpu's and try to get some more work from them.

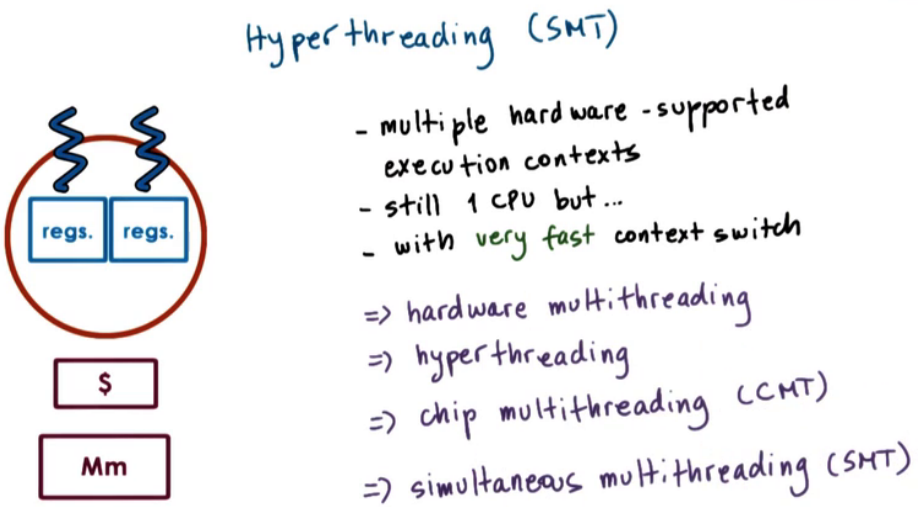


In addition to having multiple processors it is possible to also have multiple memory nodes. The cpu's and the memory nodes will be interconnected via some type of interconnect for instance on modern intel platforms there is an interconnect quick pipe interconnect or QPI. One way these memory nodes can be configured is that a memory node can be technically connected to some subset of the CPU's. for instance to a socket that has multiple processors. If that is the case, then the access from that set

of cpu's to the memory node will be faster vs from that particular processor to a memory node that's associated with another set of CPU's. Both types of accesses will be made possible because of the interconnect that's connecting all of these components however they will take a different amount of time. We call these types of platforms non-uniform memory access platforms or NUMA platforms. So then clearly from a scheduling perspective, what would make sense is for the scheduler to divide tasks in such a way so that tasks are bound to those cpu's that are closer to the memory node where the state of those tasks is. We call this type of scheduling NUMA aware scheduling.

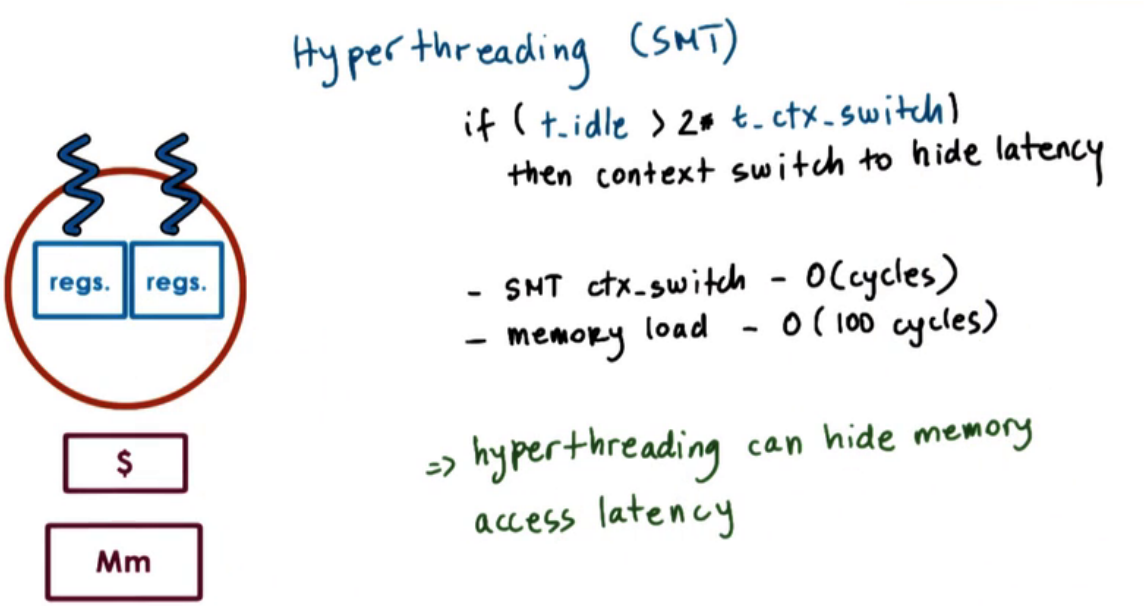
26 - Hyperthreading



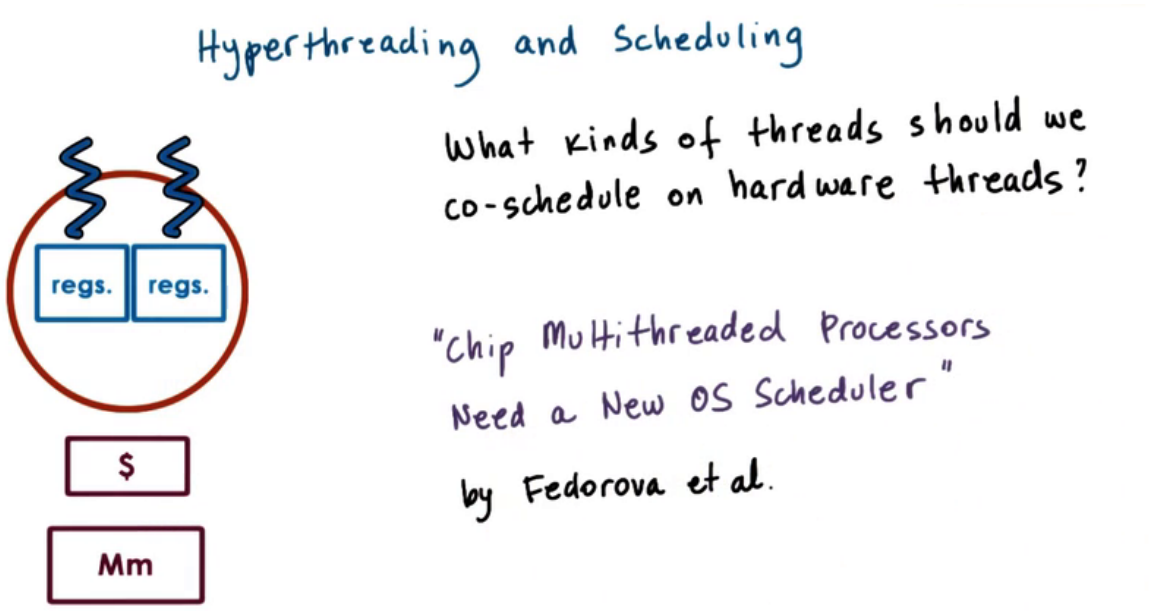


The reason why we have to context switch among threads is because the CPU has 1 set of registers to describe the active execution context. the thread that's currently running on the cpu. These include the stack pointer and program counter in particular. Over time however hardware architects have recognized that they can do certain things to help hide some of the overheads associated with context switching. One way this has been achieved is to have cpu's that have multiple sets of registers that each set of registers can describe the context of a separate thread, a separate execution entity. One term that's used to refer to this is hyper threading. Hyper threading refers to multiple hardware supported execution contexts (hyper threads). There's still just one CPU. On this CPU, only one of these threads can execute at a particular moment in time however the context switching between these threads is very fast. And just basically the CPU just needs to switch from

using this set of register to another set of registers. Nothing has to be saved or restored. This mechanism is really referred to by multiple names. In addition to hyper threading, a common term is also to refer to this as hardware multithreading or chip multithreading or simultaneous multi threading (SMT). So we will use basically these two terms more dominantly than the others. Hardware today frequently supports two hardware threads however there are multiple higher end server designs that support up to 8 hardware threads and one of the features of today's hardware is that you can enable or disable this hardware multithreading at boot time given that there is some trade offs associated with this as always.If it is enabled as far as the OS is concerned, each of these hardware contexts appears to the OS scheduler as a separate context, a separate virtual cpu onto which it can schedule threads given that it can load the registers with the thread contexts concurrently. For instance, in this figure the scheduler will think it has two cpu's and it will load these registers with the context of these two threads. So one of the decisions that the scheduler will need to make is which two threads to schedule at the same time to run on these hardware contexts.



To answer that question, let's remind ourselves of what we talked about when we talked about the context switching time. We said that if the time that a thread is idling, that a thread has to wait on something to happen, if it's greater than twice the time to perform a context switch , then it makes sense to actually do the context switch in order to hide this waiting, this idling, latency. In SMT systems, the time to perform the context switch among the two hardware threads is on the order of cycles and the time to perform a memory access operation, a memory load remains on the order of 100's of cycles, so it's much greater. So given this, it means that it does make sense to context switch to the other hardware thread and in that way, this technology, hyper threading will help us even hide the memory access latency that threads are experiencing. Hyper threading does have implication on scheduling in that it raises some other requirements when we're trying to decide what kinds of threads should we co schedule on the hardware threads on a cpu. We will discuss this question in the context of the paper Chip multithreaded processors need a new OS scheduler by Sasha Fedorova et al.

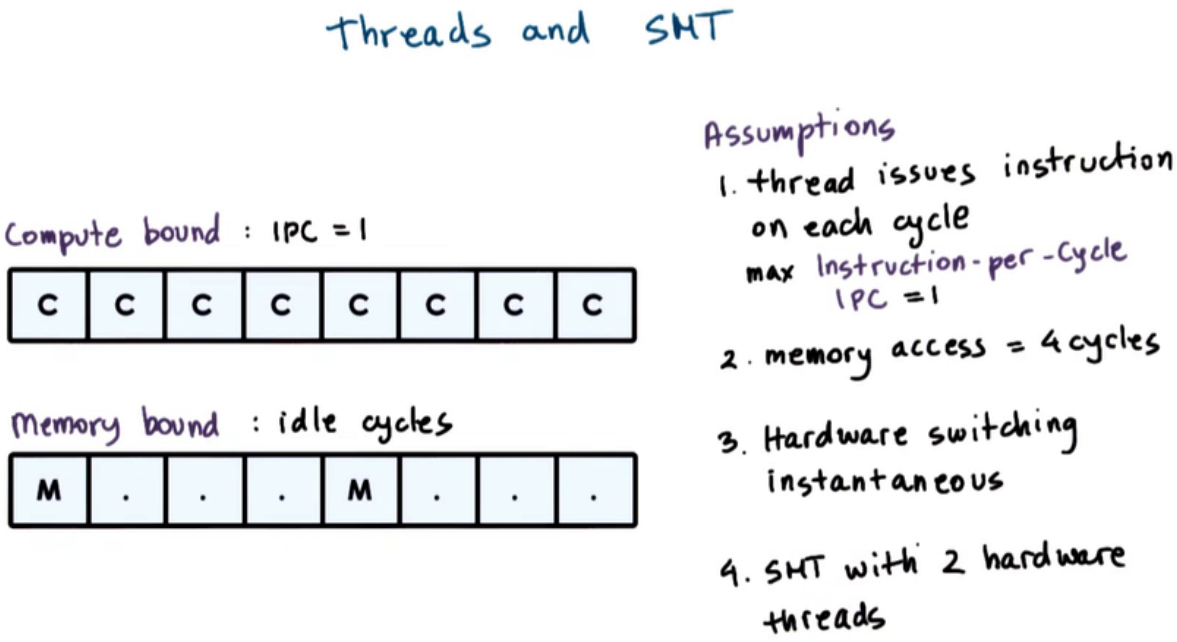


Instructor Notes

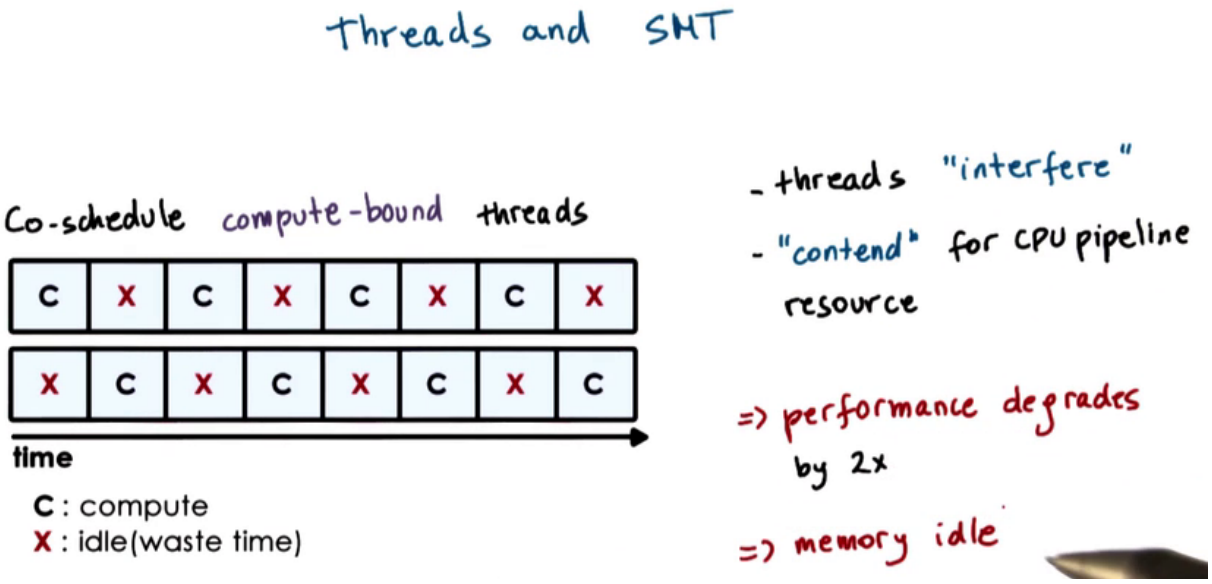
- Fedorova, Alexandra, et. al.
- "Chip Multithreading Systems Need a New Operating System Scheduler".
- *Proceedings of the 11th workshop on ACM SIGOPS European workshop (EW 11). ACM*, New York, NY, 2004.

27 - Scheduling for Hyperthreading Platforms

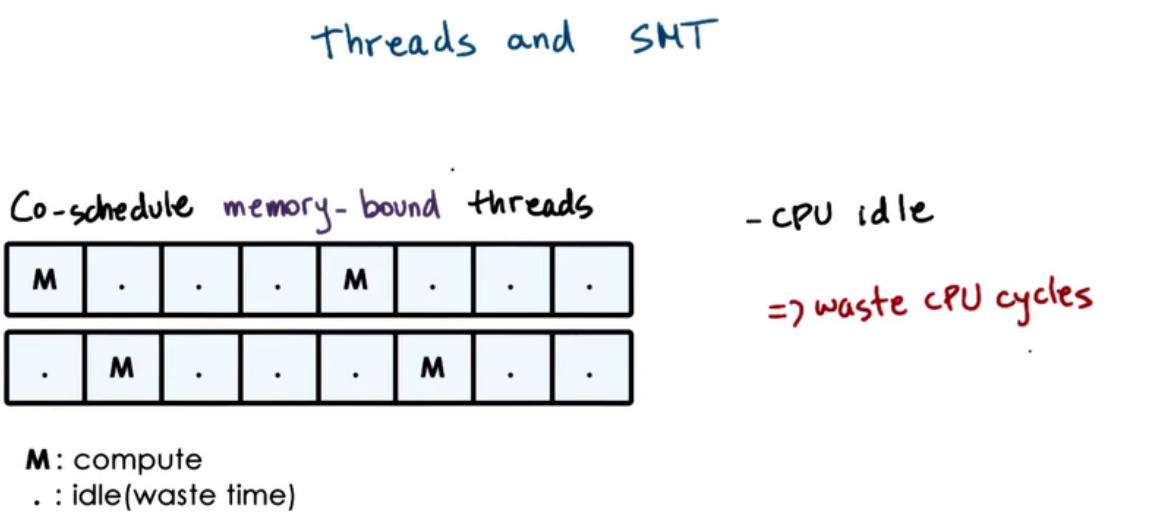
Assumptions made in Fedorova's paper.



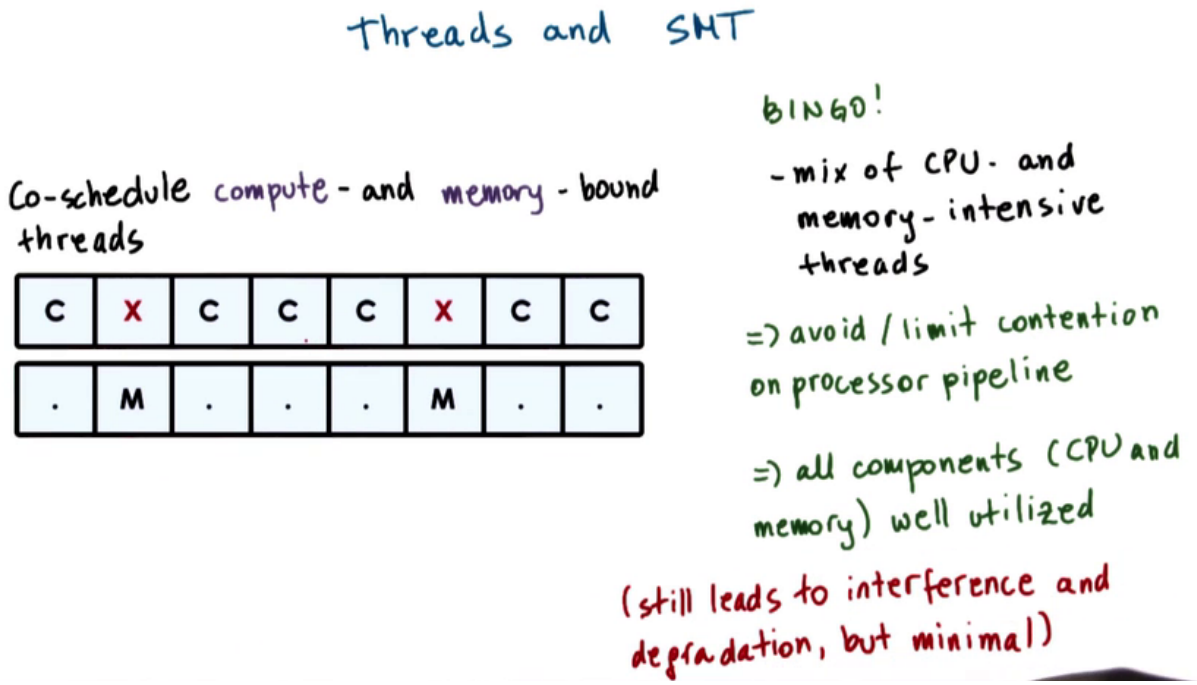
To understand what's required from a scheduler in SMT system, let's make some assumptions first. Since we will base our discussion on F's paper, we'll use the same assumptions that she has made and the figures we'll use to illustrate those assumptions will reproductions from her paper and her presentation. The first assumption we will make is that a thread can issue an instruction on every single cycle. So that means that a CPU bound thread, a thread that just issues instructions that need to run on the cpu, will be able to achieve a maximum metric in terms of instructions per cycle and that will be instructions per cycle = 1 given that we have just 1 cpu, we cannot have a IPC that is greater than 1. The second assumption that we will make is that a memory access will take 4 cycles. What this means is that a memory bound thread will experience some idle cycles while it's waiting for the memory access to return. We will assume that the time it takes to context switch among the different hardware threads is instantaneous. So we won't take any overheads over that into consideration. And also let's to start with for the sake of our discussion let's assume that we have a SMT with 2 hardware threads.



Let's take a look first at what would happen if we chose to co schedule on the 2 hardware contexts two threads that are both compute bound, compute intensive or cpu bound. What that means is that both of the threads are ready to issue a CPU instruction on every single cycle however given that there is only 1 cpu pipeline, so 1 cpu fetch, decode, issue alu logic, only 1 of them can execute at any given point in time. As a result, these 2 threads will interfere with each other, they will be contending with the CPU pipeline resources and best case every one of them will basically spend 1 cycle idling while the other thread issues it's instruction. As a result, for each of the threads, it's performance will degrade by a factor of two. Furthermore looking at the entire platform, we will notice that in this particular case, our memory component, the memory controller is idle. There is nothing that's scheduled that performs any kinds of memory accesses. That's not good either.



Another option is to co schedule 2 memory bound threads. In this case, we see however that we end up with some idle cycles because both of the threads end up issuing a memory operation and then they need to wait 4 cycles until it returns. therefore we have 2 of the cycles that are unused. So the strategy to co schedule memory bound threads leads to wasted cpu cycles.



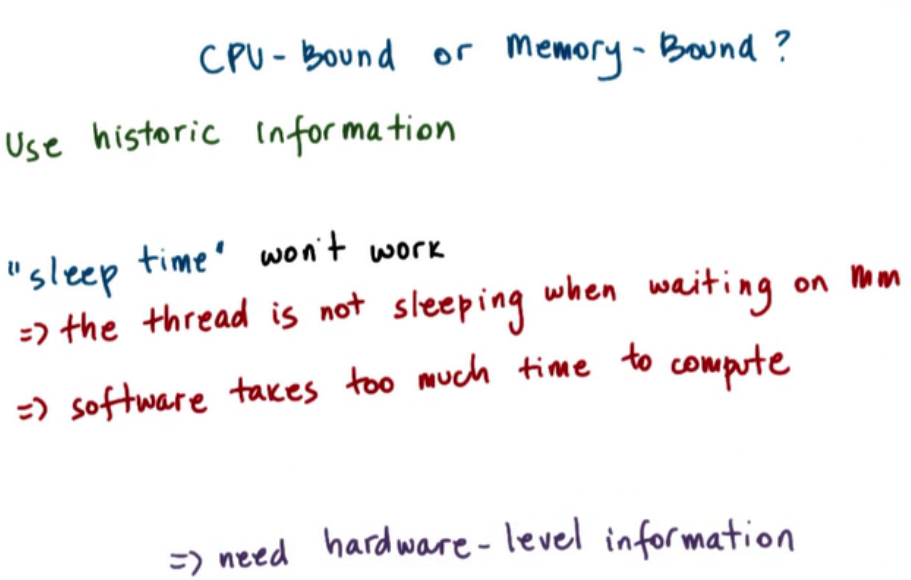
BINGO!

The final option is to consider mixing some cpu and memory intensive threads and then if we see we end up with a desired schedule. We end up fully utilizing each processor's cycle and then whenever there is a thread that needs to perform a memory reference we context switch to that thread in hardware, the thread issues the memory reference and then we context switch back to the CPU intensive thread until the memory reference completes. Scheduling a mix of cpu and memory intensive threads allows us to avoid or at least limit the contention the processor pipeline and then

also all of the components both the cpu and the memory will be well utilized. Note that this will lead to some level of degradation due to the interference between these 2 threads, for instance the compute bound thread can only execute 3 out of every 4 cycles compared to when it ran alone, however this level of degradation will be minimal given the properties of a particular system.

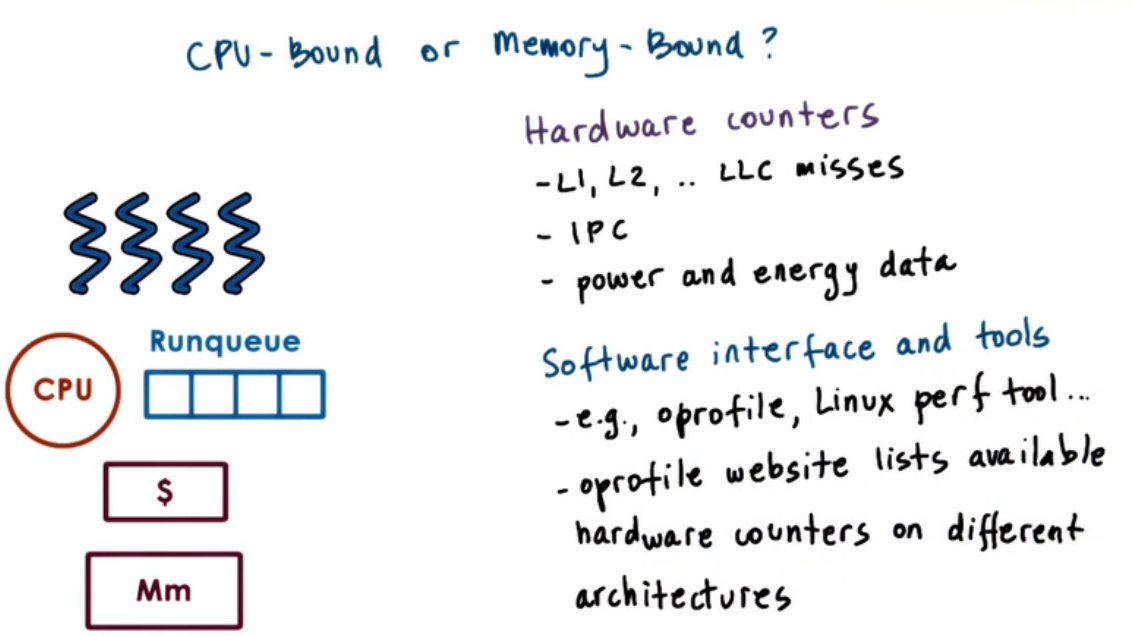
28 - CPU Bound or Memory Bound

How do we know if a thread is CPU-Bound or Memory-Bound?

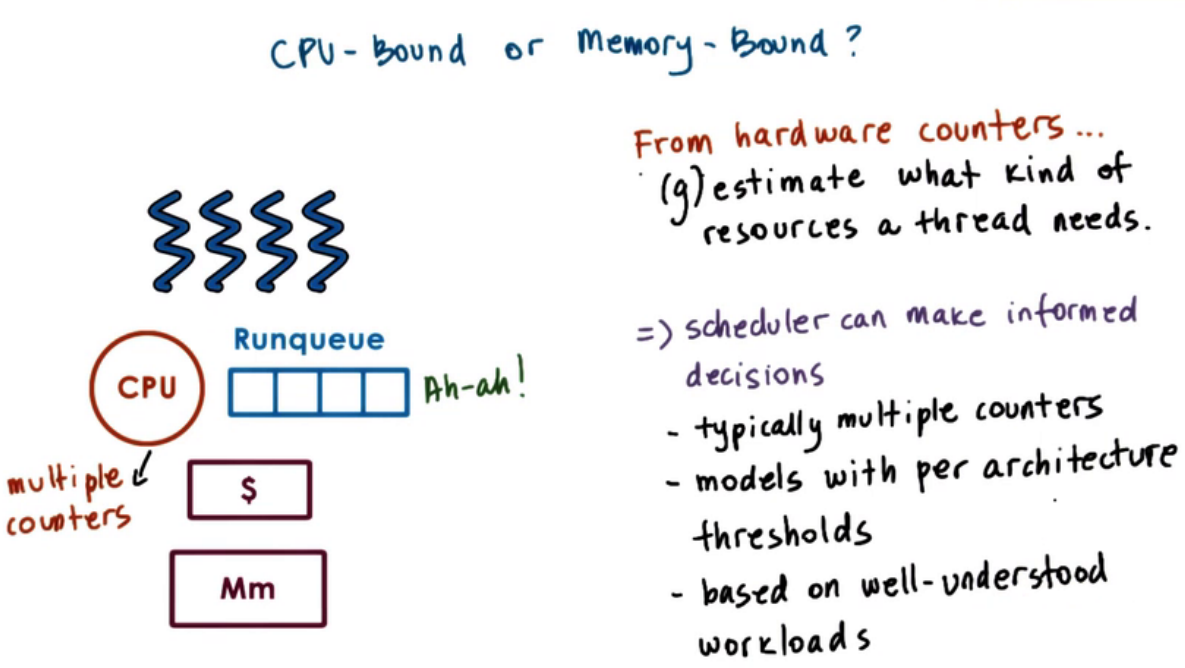


While the previous example gave us a good idea what type of schedule to use, that the scheduler should mix cpu and memory bound tasks. the question that is open at this point is how do we know if a thread is cpu bound or memory bound. To answer this question we will use historic information, we will look at the threads' past behavior and this is similar to what we said was useful when we trying to determine whether a thread is interactive or IO bound vs CPU bound. However, previously we used sleep time for this type of differentiation of IO vs CPU bound and that won't work in this case. First of all, the thread is not really sleeping when it's waiting on a memory reference. The thread is active and it's just waiting in some stage of the processor pipeline and not in some type of software queue. Second, to keep track of the sleep time, we were using some software methods and that's not acceptable. We cannot execute in software some computations to decide whether the thread is cpu bound or memory bound given the fact that the context switch takes order of cycles. So the decision what to execute should be very very fast. Therefore we somehow need some kind of hardware support, some information from the hardware in order to be able to answer this question.

Need hardware-level information in order to answer this question.



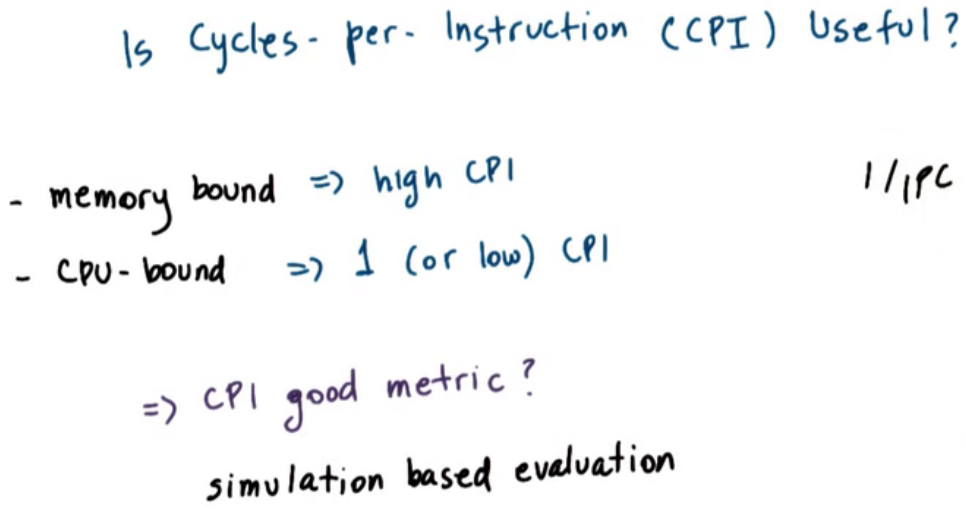
Fortunately, modern hardware has lots of so called hardware counters that get updated as the processor is executing and keep information about various aspects of the execution. These include information about the cache usage for instance such as the L1 L2 or last level cache misses or information about the number of instructions that were retired so that we can compute the IPC or on newer platforms there's also information on the power or energy usage of the CPU or particular components of the system. There are a number of interfaces and tools that can be used to access these hardware counters via software. For instance the oprofile or the linux perf tool are available in linux and one of the things that's useful is that if you look at the oprofile website it actually has a list of all of the hardware counters that are available for different for architectures because the hardware counters are not uniform on every single platform. So then how can hardware counters help a scheduler make any kinds of scheduling decisions?



Many practical as well as researched based scheduling techniques rely on the use of hardware counters to understand something about the requirements of the threads in terms of the kinds of resources that they need, CPUs or memory. So then the scheduler can use that information to pick a good mix of the threads that are available in the run queue to schedule on the system so that all of the components of the system are well utilized or that so that the threads don't interfere with one another, or whatever other scheduling policy needs to be achieved. For instance, a schedulers can look at a counter like the last level cache misses, and using this counter a scheduler can decide that a thread is memory-bound, so its footprint doesn't fit in the cache. Or the same counter can also tell the scheduler that something changed in the execution of the thread so that now it is executing with some different data in a different phase of its execution and its running with a cold cache.

What this tell us is that one counter can tell us different things about a thread. So given that there isn't a unique way to interpret the information provided from hardware counters, we really sort of guesstimate what it is that they're telling us about the threads resource use. That doesn't mean that the use of hardware counters is not good, in fact schedulers can use hardware counters to make informed decisions regarding the workload mix that they need to select. They would typically use some combination of the counters that are available on the CPU, not just one in order to build a more accurate picture of the threads resource needs and they would also rely on some models that have been built for specific hardware platform and that have been trained using some well understood work loads. So, we ran a workload that we know is memory intensive and we made some observations about the values of those counters and therefore we now know how to interpret them for other types of workloads. These types of techniques really fall into more much more advanced research problems which are a little bit out of the scope of this particular introductory course, i really wanted to make sure you are aware of the existence of these hardware counters and how they can be used and how they can be really useful when it comes to resource management in general and not just when it comes to CPU scheduling.

29 - Scheduling with Hardware Counters



As a more concrete example, Fedorova speculates that a useful counter to use to detect a thread's CPU-ness vs memory-ness is cycles per instruction. She observes that a memory bound thread takes a lot of cycles to complete the instruction therefore it has a high CPI. Whereas a CPU bound thread, it will complete an instruction every cycle or near that and therefore have a CPI of one or a low CPI. She speculates that it would be useful to gather this type of information this counter about the cycles per instruction and use that as a metric in scheduling threads on hyperthreaded platforms. Given that there isn't an exact CPI counter on the processors that Fedorova uses in her work and computing something like 1/IPC would require software computation so that wouldn't be acceptable. For that reason, Fedorova uses a simulator that supposedly the CPU does have a CPI counter and then she looks at whether a scheduler can take that information and make good decisions. Her hope is that if she can demonstrate that CPI is a useful metric, then hardware engineers will add this particular type of counter in future architectures.

Experimental Methodology

Testbed
- 4 cores x 4-way SMT
- total of 16 hardware contexts

Workload
- CPI of 1, 6, 11, 16
- 4 threads of each kind

Metric == IPC
- max IPC = 4

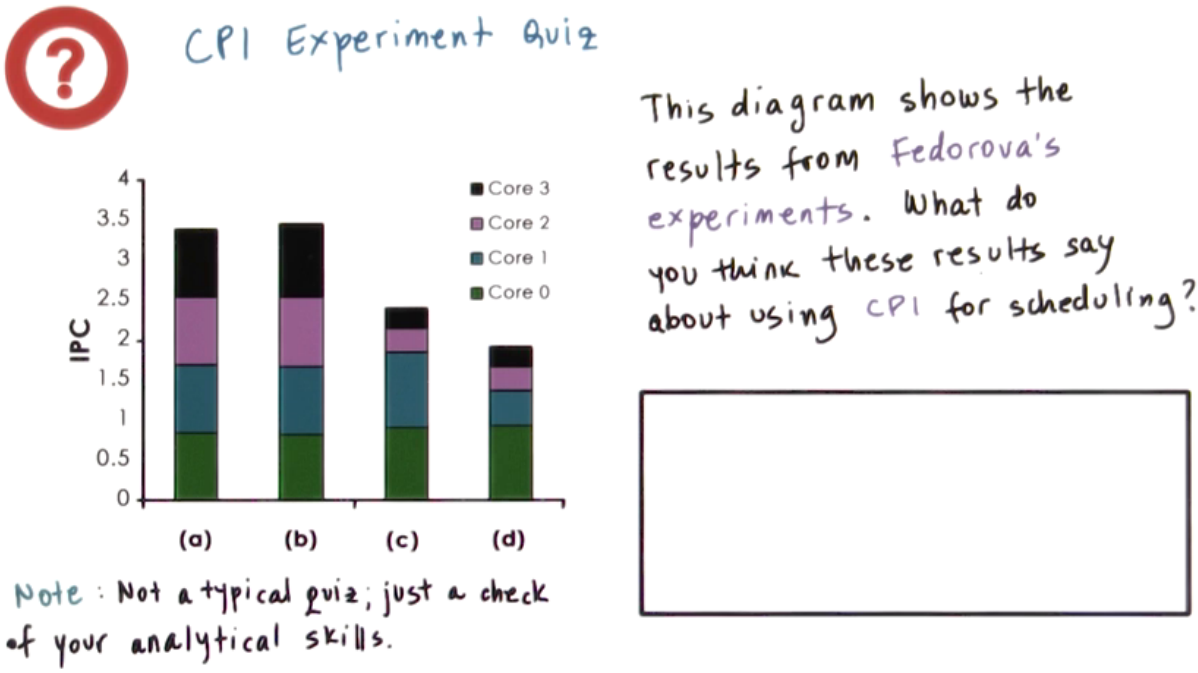
To explore this question she simulates a system that has 4 cores where every one of the cores is 4 way multithreaded (SMT) so there's a total of 16 hardware contexts in her experimental test bed. Now she wants to vary the threads that get assigned to these hardware contexts based on their CPI so she creates a synthetic workload where her threads have a CPI of 1, 6,11, and 16. Clearly the threads with a low CPI will be the most CPU intensive. The threads with the CPI of 16 will be the most memory intensive threads. The overall workload mix has 4 threads of each kind. And then what she wants to evaluate is what is the overall performance when a specific mix of threads gets assigned to each of these 16 hardware contexts. To understand the performance impact of such potentially different scheduling decisions, she uses as a metric the instructions per cycle. given that the system has 4 cores in total, the maximum IPC that can be achieved is 4. 4 instructions per cycle is the best case scenario where every single one of the cores completes one instruction in each cycle.

Actual Experiments

	Core 0	Core 1	Core 2	Core 3
(a)	1, 6, 11, 16	1, 6, 11, 16	1, 6, 11, 16	1, 6, 11, 16
(b)	1, 6, 6, 6	1, 6, 11, 11	1, 11, 11, 16	1, 16, 16, 16
(c)	1, 1, 6, 6	1, 1, 6, 6	11, 11, 11, 11	16, 16, 16, 16
(d)	1, 1, 1, 1	6, 6, 6, 6	11, 11, 11, 11	16, 16, 16, 16

And then she conducts several experiments as show in this figure. In every one of the experiments she manually and statically changes how the workload is distributed across the cores. So in the first experiment on core 1, the 4 hardware threads will be assigned threads that have software threads that have a CPI of 1, 6, 11, and 16. The 4 hardware threads on core 2 will be assigned software threads or tasks that have a CPI of 1, 6, 11, and 16. And so forth. In the first experiment, every one of the cores runs identical mix where each hardware thread runs a task with a different CPI and in the last experiment each of the cores runs a very different kinds of mix where on Core 0, all of the tasks are CPU intensive (CPI of 1) whereas on Core 3 all of the tasks are memory intensive because they have a CPI of 16. The second and the third round falls somewhere between these two extremes. What she's trying to do is trying to make some static decisions that a scheduler would have made and in doing that she's trying to understand whether it makes sense to make a scheduler that will use CPI as a metric.

30 - CPI Experiment Quiz

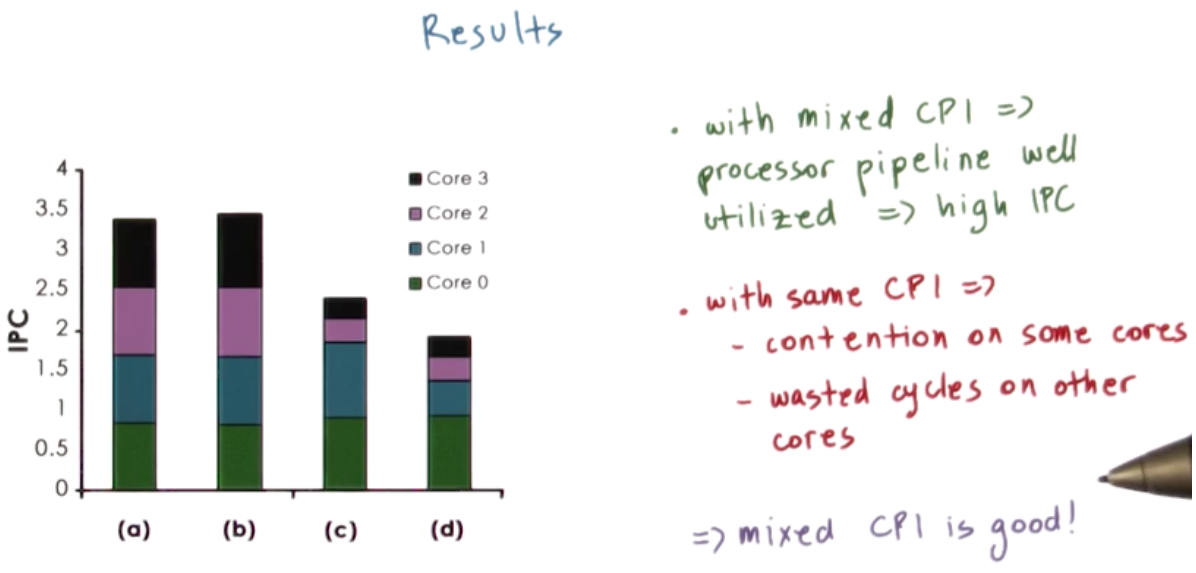


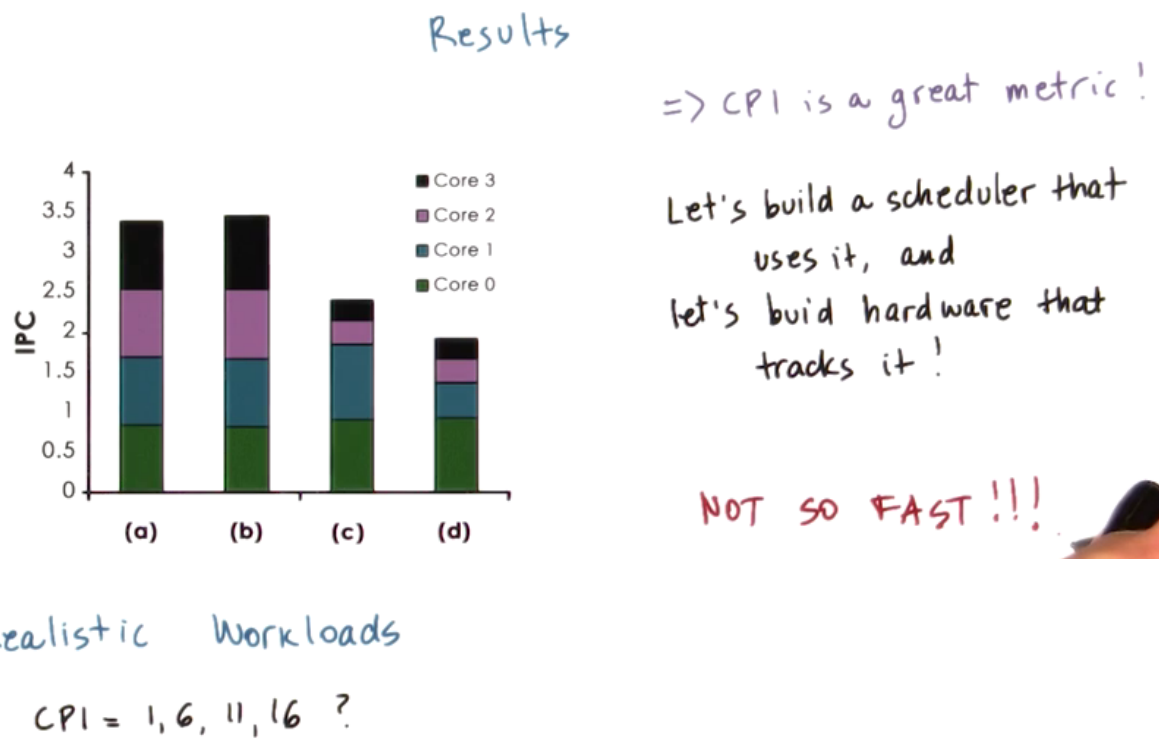
Quiz Help

For analyzing the graph, it might be helpful to use the full experiment chart (Added below).

	Core 0	Core 1	Core 2	Core 3
(a)	1, 6, 11, 16	1, 6, 11, 16	1, 6, 11, 16	1, 6, 11, 16
(b)	1, 6, 6, 6	1, 6, 11, 11	1, 11, 11, 16	1, 16, 16, 16
(c)	1, 1, 6, 6	1, 1, 6, 6	11, 11, 11, 11	16, 16, 16, 16
(d)	1, 1, 1, 1	6, 6, 6, 6	11, 11, 11, 11	16, 16, 16, 16

31 - CPI Experiment Results





Here are the conclusions that we can draw from these results. If we look at the cases for a and b, if we remember from that table that we saw with the actual experiments, in these cases we had a fairly nice mix of tasks with different CPI values on each of the cores. In these cases, the processor pipeline was well utilized and we obtained a high IPC. It is not the max IPC of 4, so the maximum performance that one can obtain on the processor that Fedorova was simulating. If we look at the cases for C and D, in these experiments each of the cores was assigned tasks that had much more similar CPI values. in fact in the case of D, every single one of the cores that ran tasks that had the exact same CPI value. So if we take a look at these results, we see that the total IPC is much lower than what we saw in the case A and B. The reason for that is that in some cores, like in Core 0 in particular and also in Core 1, there is a lot of contention for the CPU pipeline. These are the cores that mostly had tasks with a low CPI so mostly the compute intensive task were here. On the other hand, on cores 2 and 3, they contributed very little to the aggregate IPC metrics so basically they really executed only very instructions per cycle, small percentage of an instruction per cycle given that the max is one. The reason for that is that we mostly had memory intensive tasks on these two cores and that leads to wasted cycles on them. So by running this experiment Fedorova confirmed her hypothesis that running tasks that have mixed CPI values is a good thing. That will lead to an overall high performance of the system. So the conclusion from these results is that CPI is a great metric and therefore we should go ahead and build hardware that has a CPI hardware counter and tracks this value so that we can go ahead and build OS schedulers that use this value in order to schedule the right workload mix.

Realistic Workloads

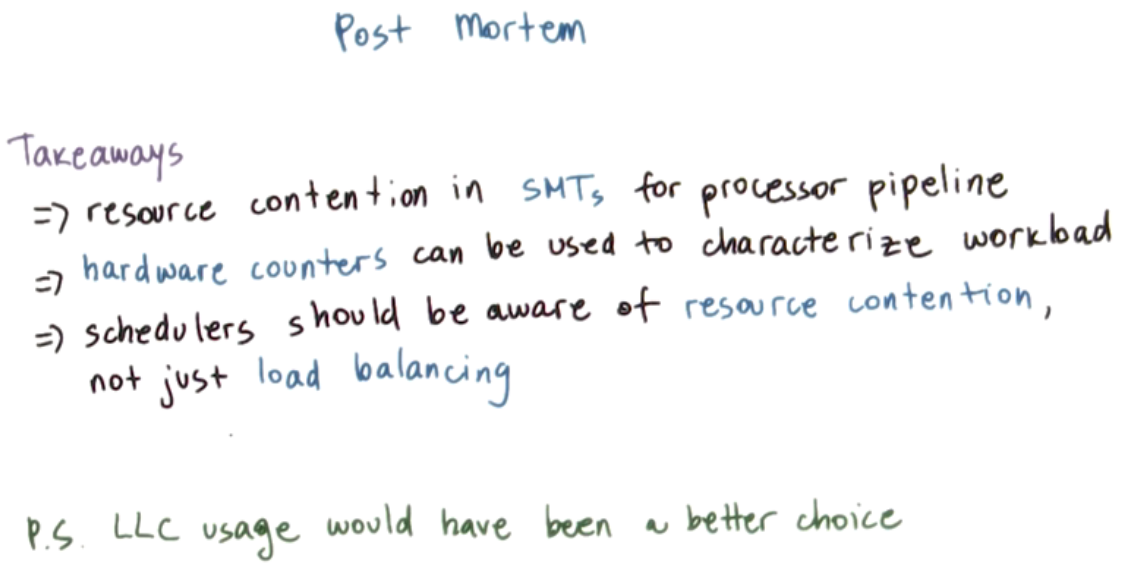
SPEC JVM	Average CPI	St. dev.
compress	2.87	0.62
db	3.62	0.61
jack	3.78	0.71
javac	3.58	0.51
jess	3.65	0.49
mpeg	4.51	1.61
mtrt	3.31	0.72

Server benchmarks	Average CPI	St. dev.
SPEC Web Apache	4.33	0.31
SPEC JBB	3.93	0.17

SPEC CPU	Average CPI	St. dev.
bzip2	2.50	0.80
crafty	2.98	0.25
gap	3.70	1.48
gcc	3.23	0.53
gzip	2.37	0.37
mcf	5.11	4.15
parser	3.55	0.30
perl	3.70	0.63
vortex	3.35	0.71
vpr	4.22	0.50
wolf	3.93	0.21

Not so fast. In our discussions so far, in the experimental analysis, we used workload that had CPI values of 1, 6, 11, and 16. And the results showed that if we have such a hypothetical workload that has such distinct and widely spread out CPI values, the scheduler can be effective. But the question is do realistic workloads have CPI values that exhibit anything we used in this synthetic workload. To answer this, Fedorova profiled a number of applications from several benchmark suites. These benchmark suites are widely recognized in the industry and academia as well that they include workloads that are representative of real world relevant applications. Let's look at the CPI values for all of these benchmarks. We see that they're all sort of cluttered together. They aren't such distinct CPI values such like 1, 6, 11, and 16 as what she used in her experimental analysis. What this tells us is that although in theory, it seems like a great idea to use CPI as a scheduling metric for hyperthreaded platforms, in practice, real workloads don't have behavior that exhibit significant differences in the CPI value and therefore CPI really won't be a useful metric.

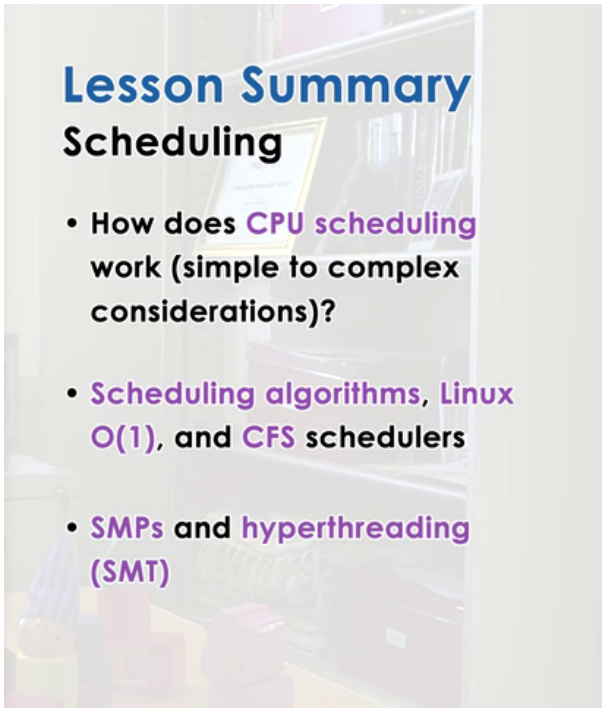
The paper shows us something that doesn't work!



So I showed you a paper that doesn't work. There's still some very important takeaways from this paper. First to learn about SMT's and some of the resource contention issues there, specifically regarding the processor pipeline as a resource. Next, you learned how to think about the use of hardware counters to establish some kind of characteristics about the workload to understand it better so that you can better inform the OS level resource management. In addition you learned that it's important to design schedulers that will also think about the resource contention, not just about load balancing. For instance a scheduler should think about choosing a set of tasks that are not going to cause a resource contention with respect to the processor pipeline or the hardware cache or the memory controller or some type of IO device so these principles generalize to other types of resources, not just to the processor pipeline in hardware multi threaded platforms. And btw, in Fedorova's work as well as several other efforts, it's been established that particularly important contributor to performance degradation when you're running multiple tasks on a single hardware multi threaded or multi core platform is the use of the cache resource, in particular the last level cache. So what that has told us is to for instance keep track of how a set of threads is using the cache as a resource and pick a mix that doesn't cause contention on the last level cache usage. And this is just for your information. We're not going to be looking at any additional papers that further explains this issue. Not in this course.

LLC = Last Level Cache

32 - Lesson Summary



In summary, you should now know how scheduling works in an operating system. We discussed several scheduling algorithms and the corresponding run queue data structure that they use. We described two of the schedulers that are default in the linux kernel the completely fair scheduler and it's predecessor the linux O(1) scheduler. And also we discussed some of the issues that come up in scheduling when considering multiple cpu platforms. This we said includes platforms with multiple cpu's that share memory, multi core platforms, as well as platforms with hardware level multi threading.