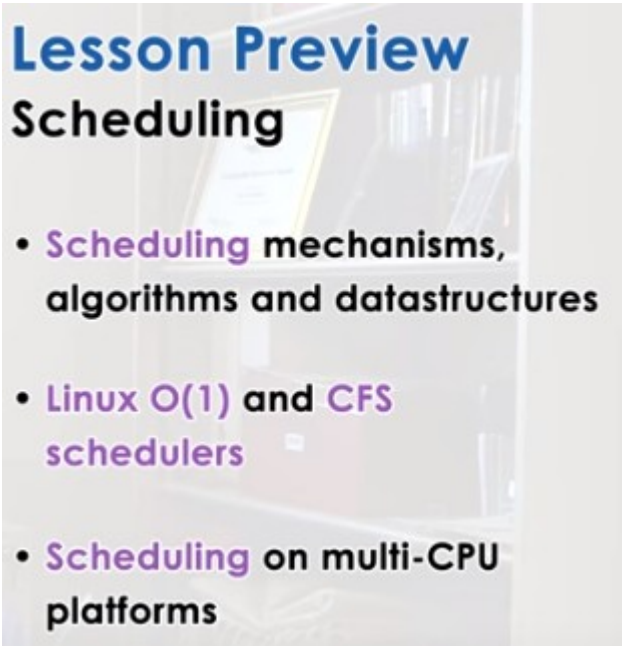
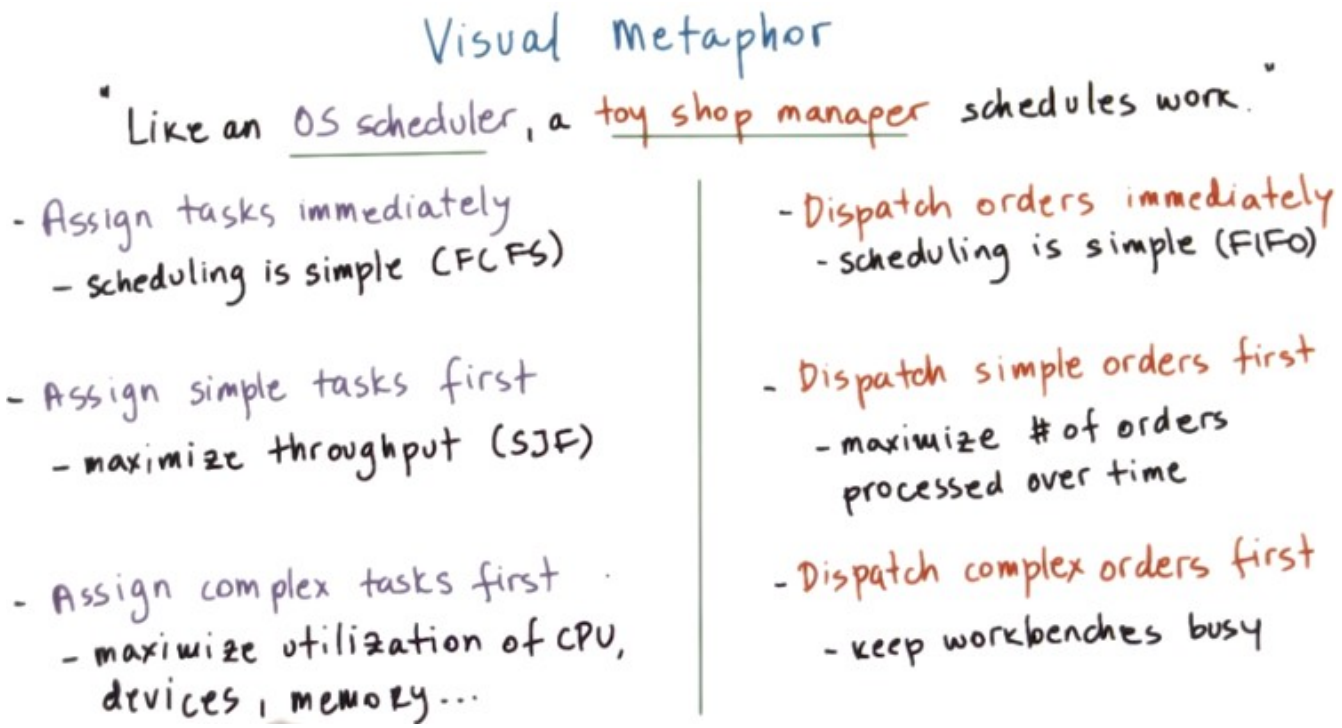
第二遍 review 時, 不僅要看圖片, 還要看紅字



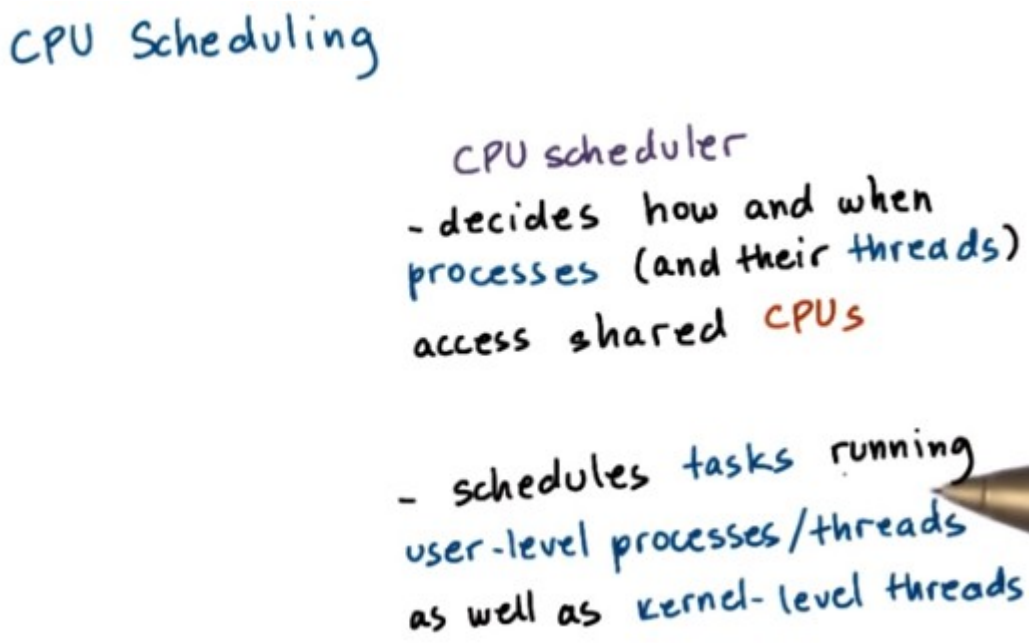
1. In the next set of lessons, we will discuss in more detail some of the key research management components in operating systems. First we will look at how the operating system manages CPUs, and how it decides how processes and their threads will get to execute on those CPUs. This is done by the scheduler. 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 system, 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 multicore platforms and also platforms with hardware level multithreading. For this we will use the paper chip multithreaded processors need a new operating system scheduler in order to demonstrate some of the more advanced features that modern scheduler should incorporate.



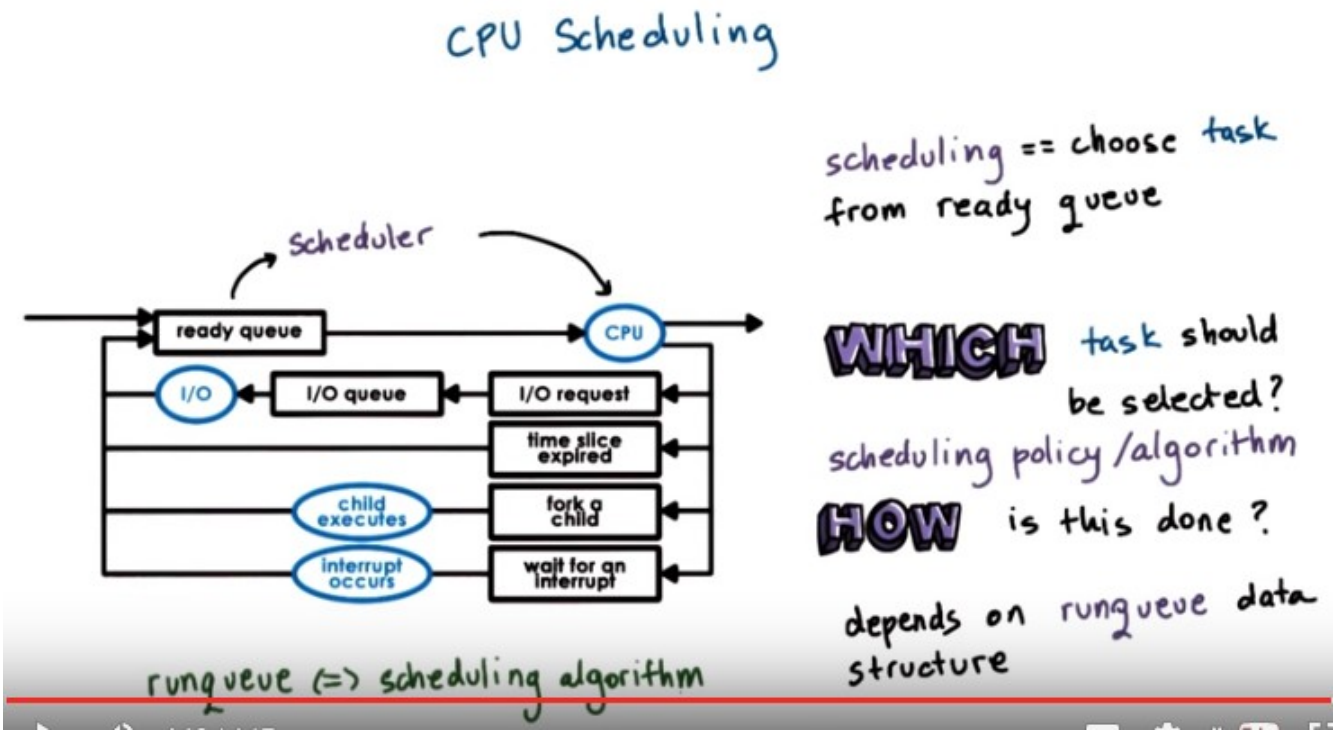
workbench: 工作臺
FIFO: first in first out
FCFS: first come first serve
SJF: shortest job first

2. In this lesson, we will be talking at length about scheduling and OS schedulers. So continuing with our visual metaphor, we'll try to visualize what are some of the issues when it comes to OS scheduling. 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 schedules work in the toy shop. For a toy shop manager, there are multiple ways how it can choose to schedule the toy shop workers and dispatch them two workers in the toy shop. First, the toy shop owner 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 order that it 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 for the manager to be able to assess the complexity of

the orders. On the other hand, on each of the work benches in the shop, there may be different kinds of tools, and these simple orders, these may not really exercise all of these tools, they may not require the use of all aspects of the work benches. So if the manager wishes to keep all of the resources that are available on the work benches 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 a toy shop manager are surprisingly similar to those of an OS scheduler. For an OS scheduler, these choices are analyzed in terms of tasks, which is either a thread or a process. And these are being scheduled onto the CPUs that are managed by the OS scheduler, like the work benches 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, can dispatch simple tasks first. The outcome of this kind of scheduling can be that the throughput of the system overall is maximized. And there are schedulers that actually follow this kind of algorithm and those are called shortest job first. So simple in terms of the task is equal to its running time, so shortest job first. And finally, the scheduling logic behind assigning complex task first is similar to the case in the toy shop. Here, the scheduler's calls are to maximize all aspects of the platform. So to utilize well both the CPUs 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 various aspects of the design and implementation of OS schedulers.

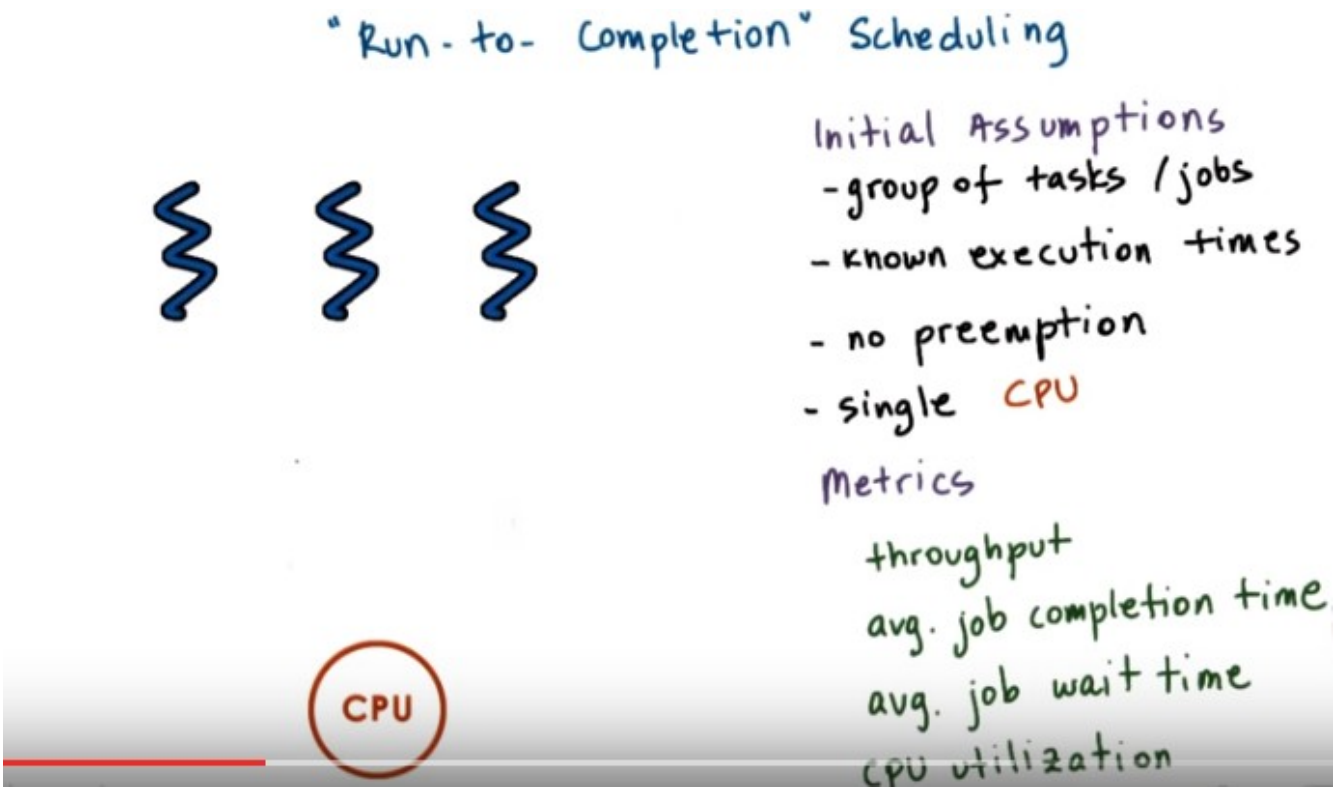


3. We talked briefly about CPU scheduling in the introductory lesson on processes and process management. We said then, the CPU scheduler decides how and when processes access the shared CPUs in the system. In this lesson, we'll use the term task to interchangeably mean either processes or threads, since the same kinds of mechanisms are valid in all context. The scheduler concerns the scheduling of both user-level processes or threads, as well as the kernel-level threads.



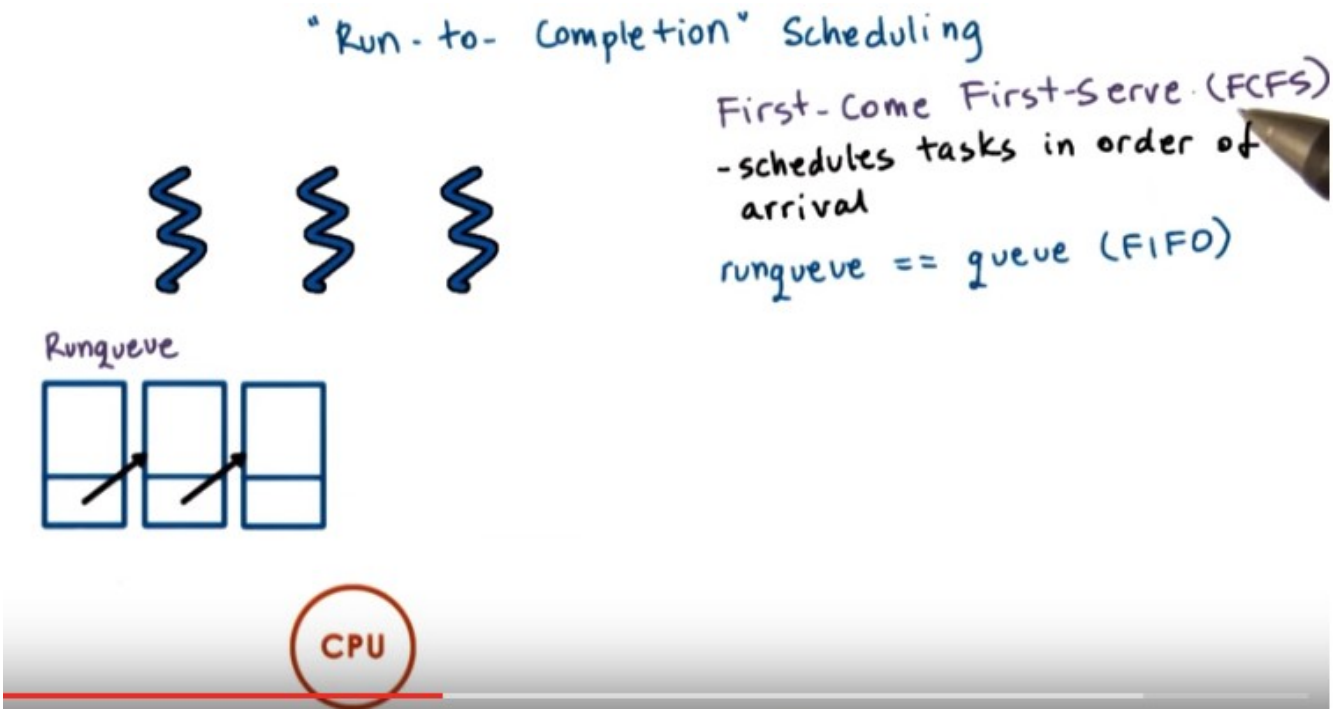
Recall this figure that we used when we talked originally about processes and process scheduling. The responsibility of the CPU scheduler is to choose one of the tasks in the ready queue, a process or a thread. We said we'll just use the term task to refer to both, and then to schedule that particular task onto the CPU. Threads may become ready so they may enter the ready queue after an I/O operation they have been waiting on has completed or after they have been woken up from a wait on an 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, 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 (上圖中有 ready queue) and decide which is the one that it will 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 I/O request and now it has to wait in the I/O queue for that particular device, the CPU's 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, so a task that was waiting on an I/O operation, or was waiting on an interrupt, or some kind of signal. Or whenever a brand new task was created for the first time. For all of these reasons, again, we need 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 how schedulers share the CPU is to give each of the tasks in the system some amount of time on the CPU. So, when a time slice expires, when a running thread has used up 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 we perform a context switch to enter the context of the newly selected thread. And then we have to enter user mode, then set the program counter appropriately to point to the next

instruction that needs to be executed from the newly selected thread. And then we're 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 ready tasks in the system. The first question that comes to mind is, which 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 runqueue on the data structure that we use to implement this ready queue. That's called the runqueue. The design of the runqueue and the scheduling algorithm are tightly coupled. We will see that certain scheduling algorithms, they demand a different type of runqueue, different data structure. And also that, the design of the runqueue, it may limit in certain ways what kinds of scheduling algorithms we can support efficiently. So, the rest of the lecture will focus on reviewing different scheduling algorithms and runqueue data structures.

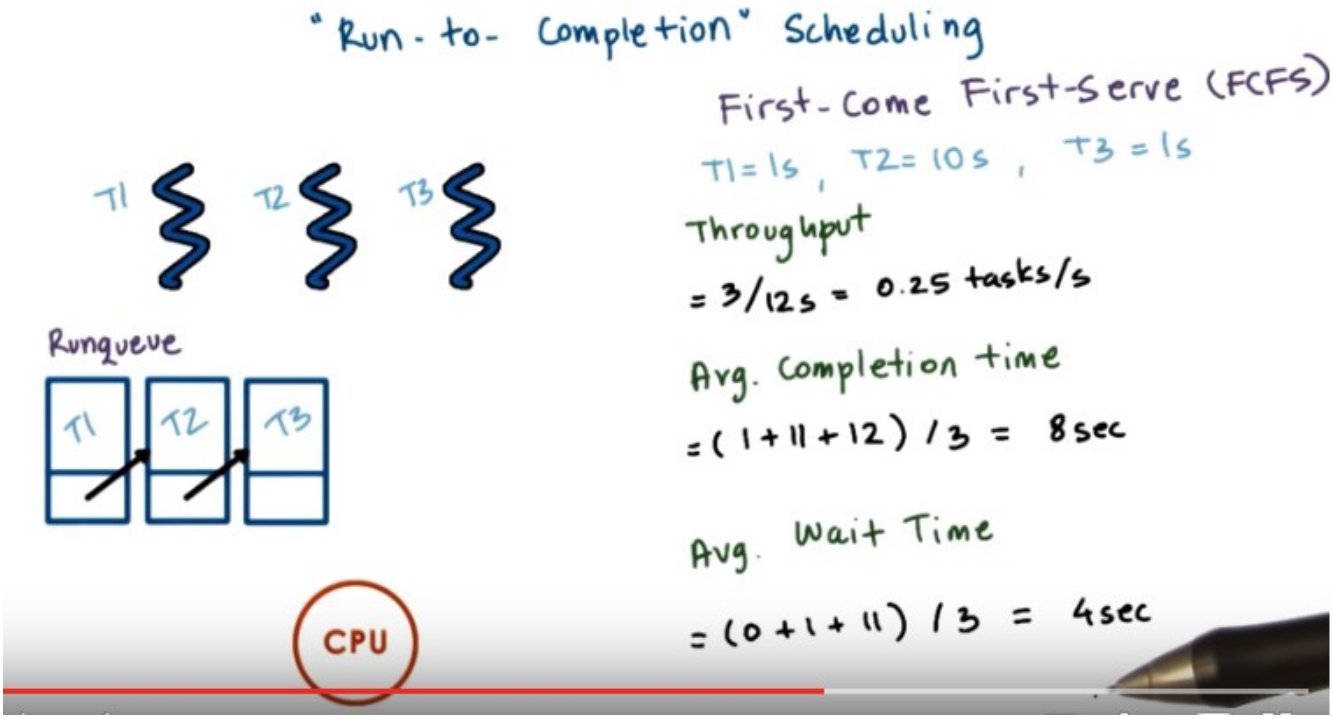


4. For this first discussion of scheduling algorithms I want to focus on what we call run-to-completion scheduling. This type of scheduling assumes that as soon as a task is assigned to a CPU, it will run until it finishes or until it completes. Let me list first some of the assumptions we will make. 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, to start with, assume that we will know exactly how much time these threads need to execute. So there will be no preemption in the system. Once a task 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 me 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 be able to compare them, so we're going to think about some useful metrics. When it comes to comparing schedulers, some of the metrics that can give meaningful

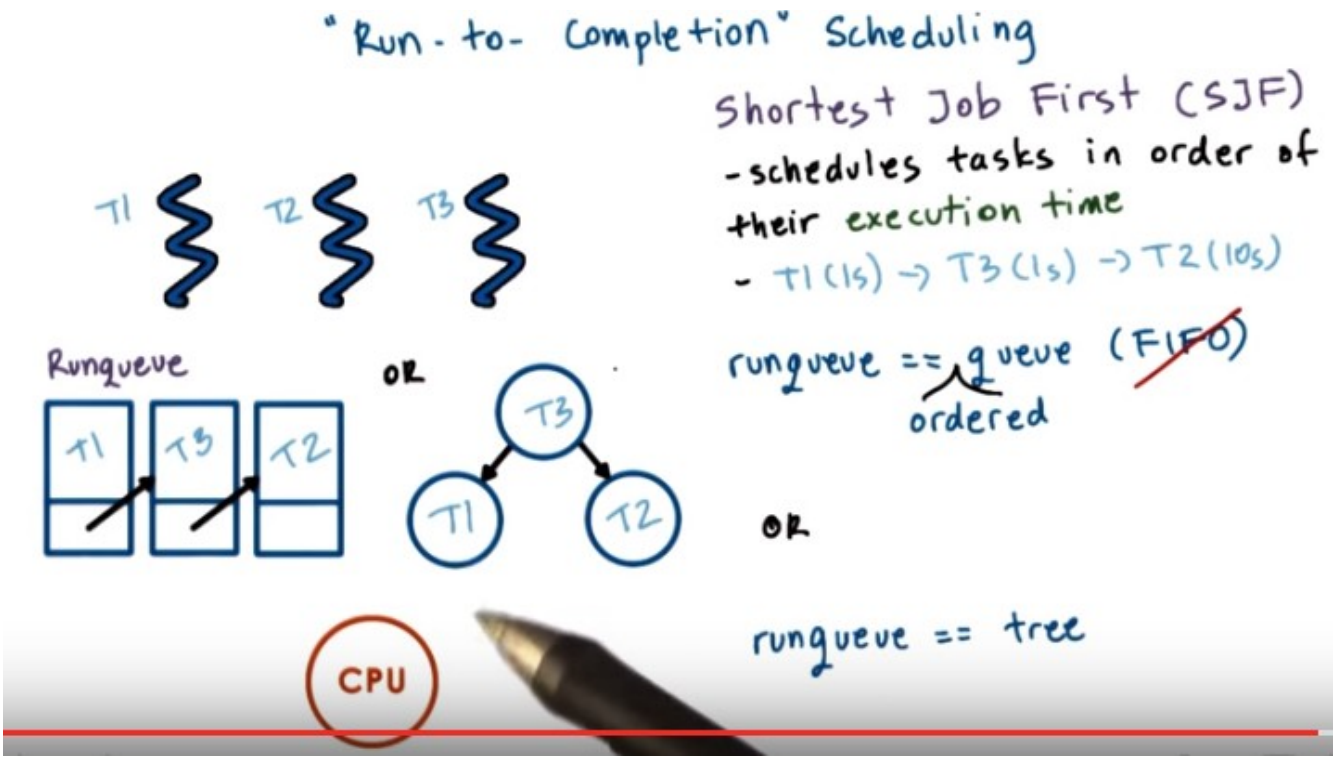
answers regarding those comparisons include throughput. The average time it took for tasks to complete, the average time the tasks spent waiting before they were scheduled, overall CPU utilization. We will use some of these metrics to compare some of the algorithms that we will talk about.



The first and the simplest algorithm we'll talk about is First-Come First-Serve. In this algorithm, tasks are scheduled on the cpu in the same order in which they arrive. Regarding of their execution time, of loading the system, or anything else. When a task completes, the schedule 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 it in 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 will be 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.



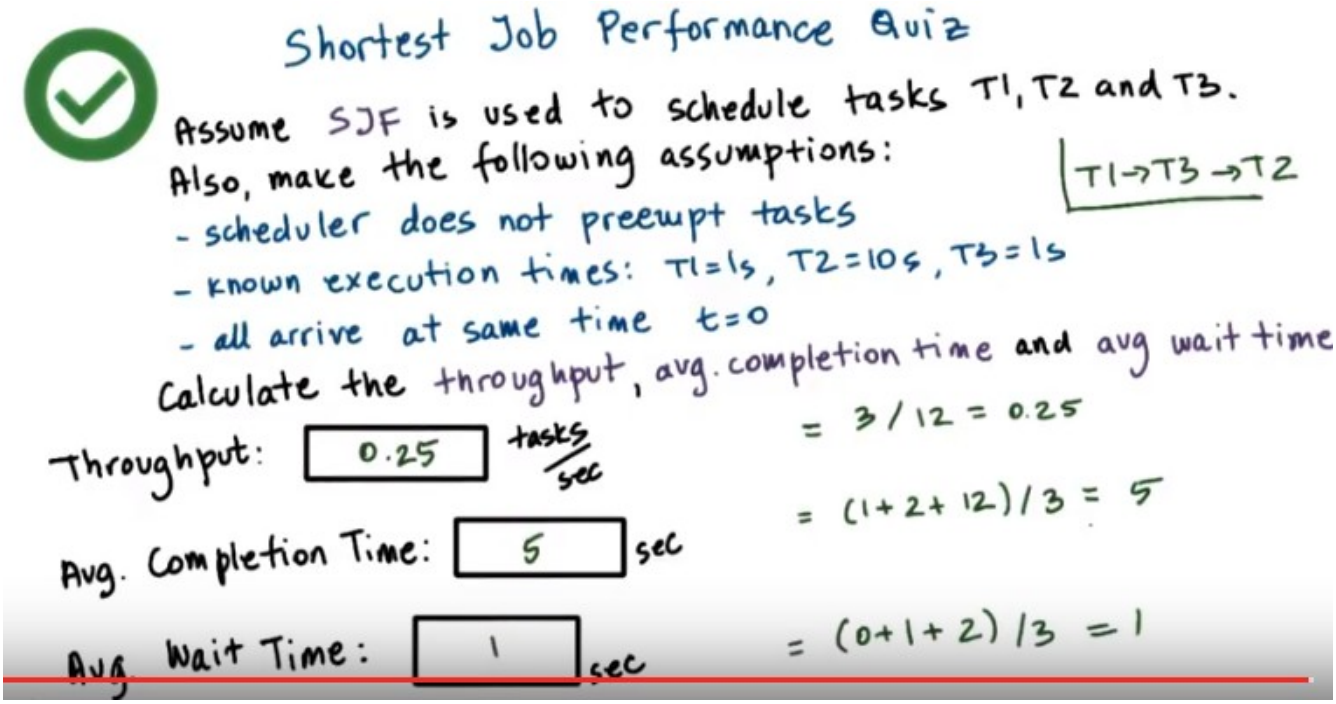
Now let's take a look at this area in which these three tasks T1 T2 and T3 have the following execution times. T1 is one second, T2 is ten seconds, and T3 is also one second. And let's assume they arrive in this order. So T1 followed by T2 followed by T3. So this is how they'll be placed in the runqueue. Now let's look at through put asymmetric for this kind of system that uses the first come first serve scheduler. We have three tasks. To execute them one after the other we will take total of 12 seconds, 1 plus 10 plus 1. So the scheduler on average achieves a quarter of a task per second. So 0.25 tasks per second. If we are interested in the average completion time of these tasks, the first task will complete in one second since it will start immediately. The second task, it will complete at time 11. It will have to wait one second for the first task to complete, and then it will execute for ten seconds. And then the third task it will complete at time T12, because it will have to wait for the 11 tasks for T1 and T2 to execute until it starts and executes for one second. So, if we compute the average completion time in the system we see that it is 8 seconds. If we're interested in the average wait time for the three tasks in the system, then the first task started immediately. The second task started a second later, because it had to wait for T1 to complete, and then the third task had to wait for 11 seconds before it started executing. So, the average wait time for these three tasks is four seconds. So we have our simple scheduling algorithm, however, probably we can do a little bit better in terms of the various metrics that you'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 and T3, the order in which we want to execute them would be T1 followed by 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. Now if we organize our run queue the same way as before, adding new tasks to the run queue will be easy, since it will just mean that we have to add a 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 really be a FIFO run queue anymore, since we will need to remove tasks from the queue in a very specific order based on execution time. One thing we can do is 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 a new task as short as it was before. We just need to know the head of the queue. Or, our run queue doesn't really have to be a queue. It can be some treelike data structure, in which the nodes in this tree are ordered based on their execution time. When inserting new nodes in the tree, new tasks in this 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 note in the stream (所以是 binary search tree? 答: 不是, 由後面知, 是 red-black tree), if the tree is ordered, the left most note will have the smallest execution time. So we a queue, a tree, this illustrates really that the run queue doesn't really have to be a queue. It can be other type of data structure, and we'll see that it often is, based on whatever is appropriate for the scheduling algorithm

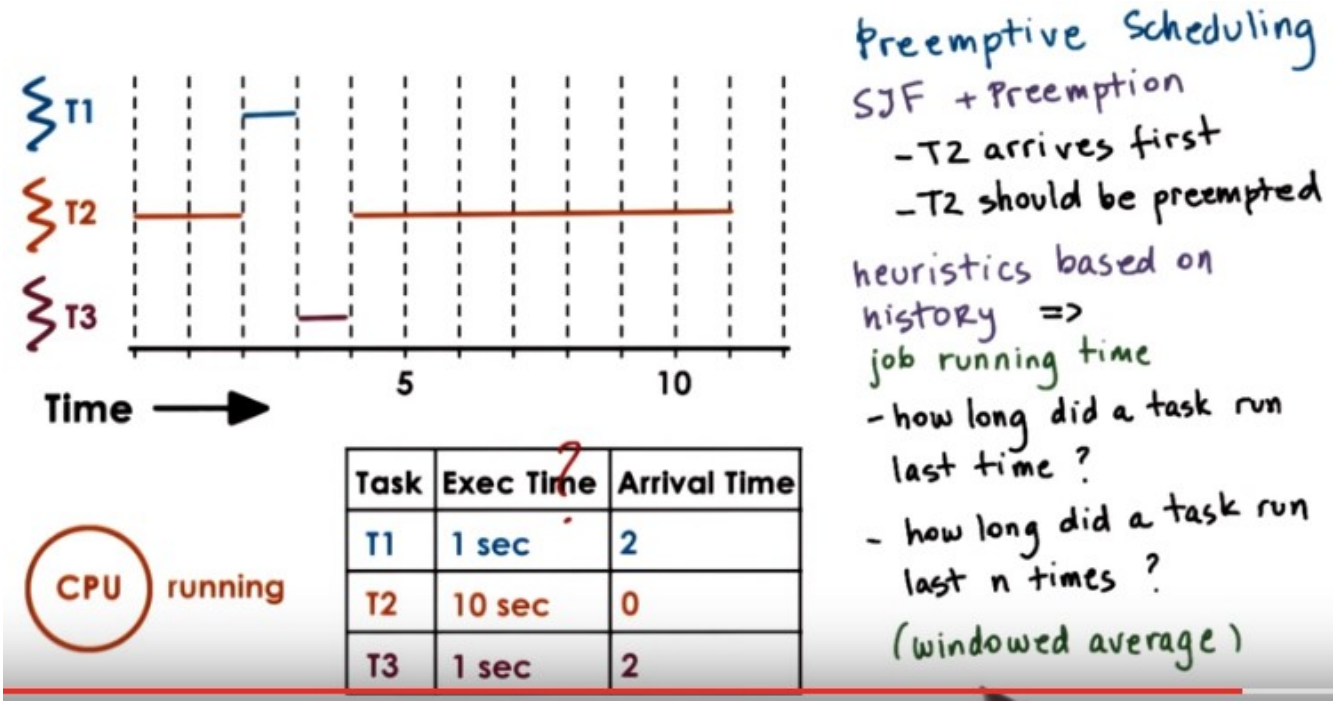
5. Let's do a quiz in which we analyze the performance metrics for shortest job first. To do this, let's assume first that shortest job first is used to schedule three tasks, T1, T2, and T3. Also, let's make the following assumptions. The scheduler will not preempt the task, so this will be the run to completion type of model that we discussed so far. We'll assume that we know the execution times of these tasks,

and we'll use the same values as we used before. So, T1 and T3 are 1 second and T2 is ten seconds. And, let's assume that they're all arrive at the same time. So, we will start analyzing this system from some time, T0, when all of these tasks are actually in the system already. Per this system, calculate the throughput, the average completion time, and the average wait time, using the shortest job first algorithm.



由第 13 段可知, completion time 即所有 task 結束的時刻加起來, 本例中(T1->T3->T2), T1 在 1s 結束的, T3 在 2s 結束的, T2 在 12s 結束的, 故為(1 + 2 + 12) / 3.

6. Let's look at the answer to this question. So first, given that we have a shortest job first algorithm and these are the execution times of the tasks, what is going to be the execution order of these tasks? Clearly they'll have to execute in this order. So T1, followed by T3 or the other way around, and then followed by T2. Now that we know the order of the tasks, we can actually compute these three metrics. To compute the throughput, again we have three tasks in the system, all of the three tasks get processed in 12 seconds. So 1 plus 10 plus 1. So we have a total throughput of 0.25 tasks per second. So nothing has really changed compared to the first come, first serve algorithm here. Now if we look at the average completion time, the first task T1, completes in 1 second. The second task T3, that one completes in another 1 second so in 2 seconds. And then the third task, T2, well that one will need another 10 seconds, so it will complete at time T12. So the average completion time is 1 plus 2 plus 12 over 3. That's five seconds. That's already way better than the eight seconds that we saw in the first come first serve algorithm. If we look at the average wait time, the first task, T1, it then waited, always started executing immediately. T3 had to wait one second, T2 had to only wait two seconds, since both T1 and T3 are short. So, the average wait time in this system is only one second. That is way better than the four seconds that first come first serve resulted in. So if we care about metrics such as these, clearly shortest job first would be a better algorithm than first come, first served.



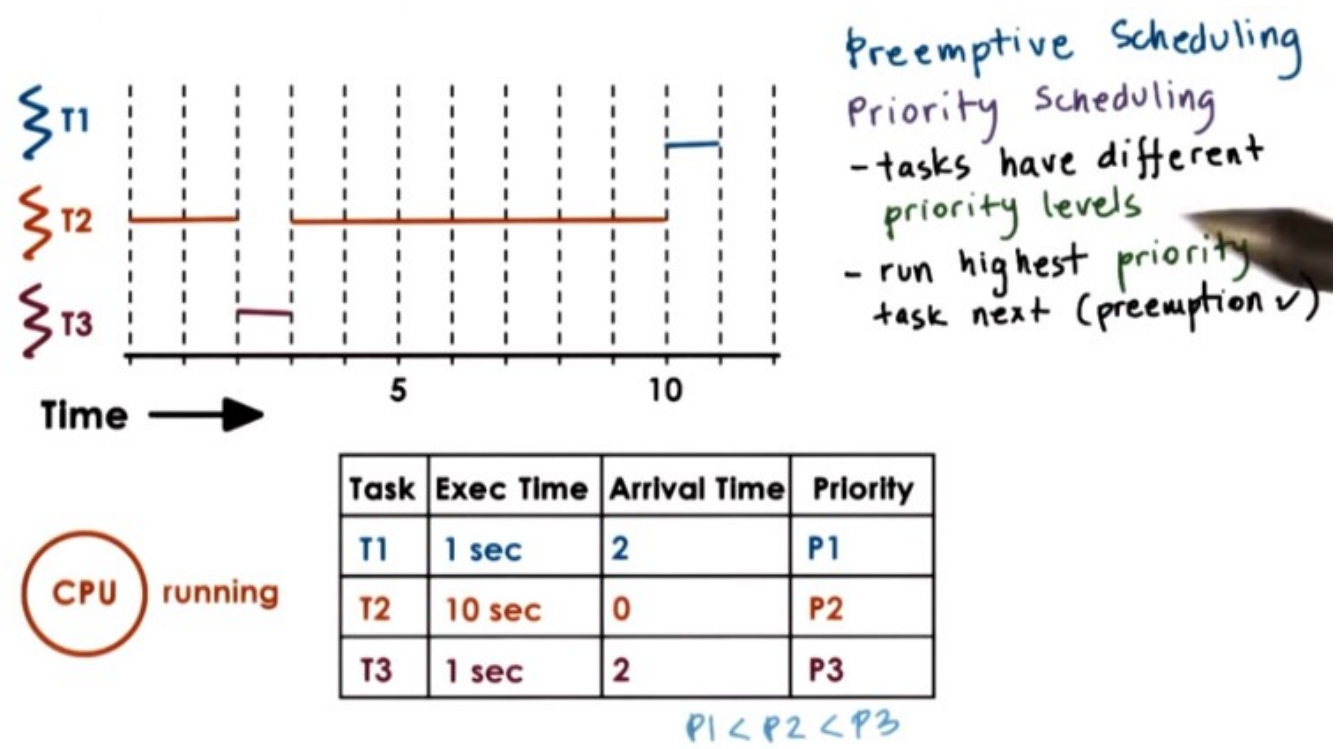
heuristics: 啟發式知識

From Wikipedia 中文: 搶占式多任務處理（Preemption）是計算機作業系統中，一種實現多任務處理（multi task）的方式，相對於協作式多任務處理而言。協作式環境下，下一個進程被調度的前提是當前進程主動放棄時間片；搶占式環境下，作業系統完全決定進程調度方案，作業系統可以剝奪耗時長的進程的時間片，提供給其它進程。

From Wikipedia: In computing, preemption is the act of temporarily interrupting a task being carried out by a computer system, without requiring its cooperation, and with the intention of resuming the task at a later time. Such changes of the executed task are known as context switches. It is normally carried out by a privileged task or part of the system known as a preemptive scheduler, which has the power to preempt, or interrupt, and later resume, other tasks in the system.

7. So far in our discussion, we assume that the task that's executing on the CPU cannot be interrupted, cannot be 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 will modify another assumption that we made earlier. Tasks don't all have to arrive at the same time. So we're going to look at the system that has three tasks. T1, T2, and T3. 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 something as follows. Let's say at T2 when the tasks T1 and T3 show up. T1 is the one that's scheduled next. Then once it completes, T3 is the next one that has the shortest running time. So T3 will execute. And once these two have completed, then T2 can take the remaining of its time to execute. So basically, what would need to happen in order for us to achieve this kind of behavior is

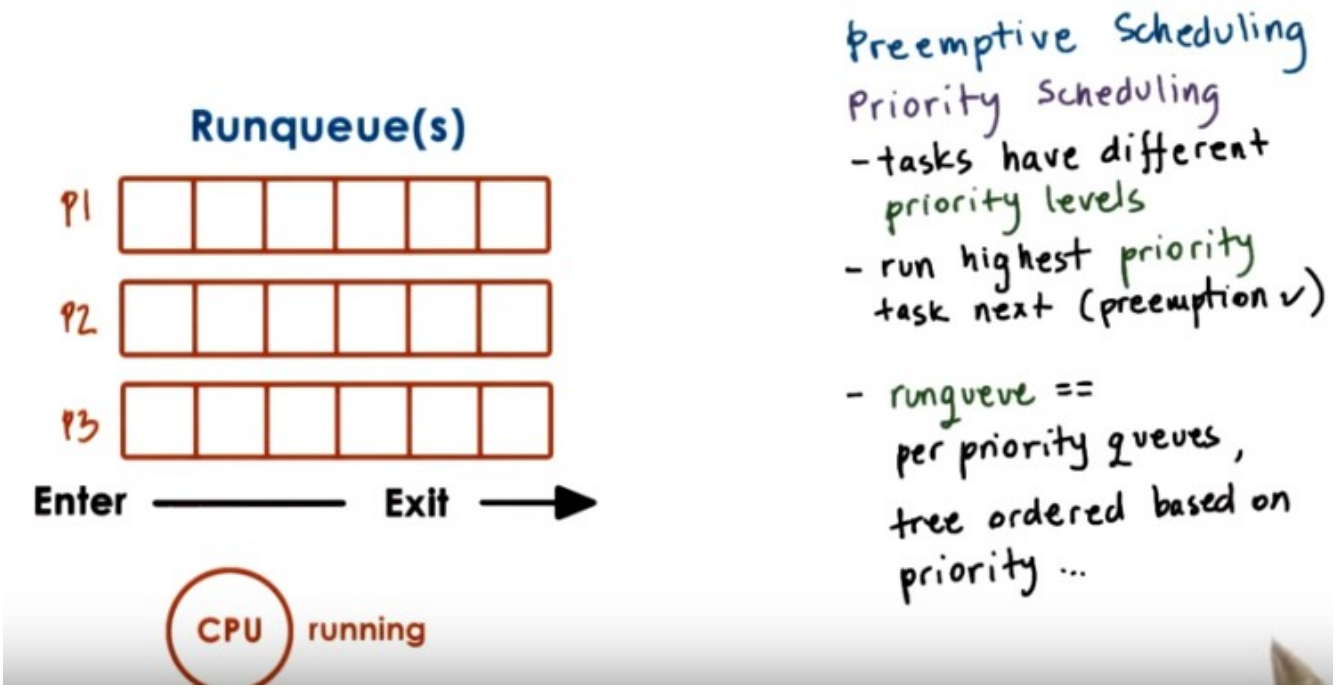
that, whenever tasks enter the run queue, like T1 and T3, the scheduler needs to be involved, so that the scheduler can inspect their execution times, and then decide whether to preempt the currently running task, task T2, and schedule one of the newly readied tasks. Now, so far we talked as if we know the execution time of a task. But it's principle, that's, that's hard to tell. It's really hard to claim that you know what is the execution time of a task. There are a number of factors that depend on the inputs of the task, on whether the data is present in the cache or not. Which other tasks are running in the system. So in principle, we have to use some kind of heuristics in order to estimate, or rather guesstimate what the execution time of a task will be. When it comes to the execution time, so the future execution time of the task, it's probably useful to consider, what was the past execution time of that task, or that job. So 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 metric based on a window of values from the past, and the window of historic values. And then use that average for prediction of the behavior of the task in the future.



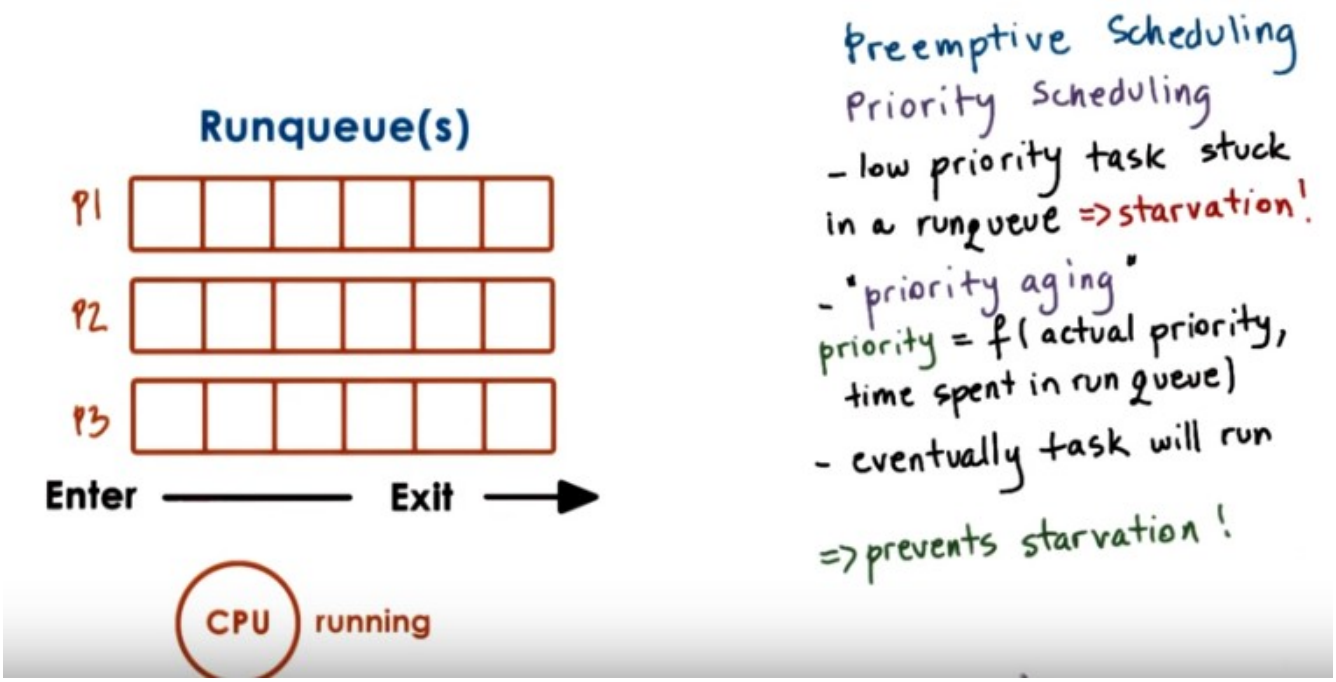
Task	Exec Time	Arrival Time	Priority
T1	1 sec	2	P1
T2	10 sec	0	P2
T3	1 sec	2	P3

8. Shorter's job considers the execution time of the tasks in order to decide how to schedule tasks, and when to preempt a particular task. Another criteria for driving those decisions may be that the tasks have different priority. Tasks that have different priority levels, that's a pretty common situation. For instance, we have certain 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, maybe 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 for 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 high priority one can run next. So, let's look at now at the same example from before, except now

we're going to use priority scheduling. And we need to know the task's priorities. And in this particular example, the priorities P1, P2, and P3 are such that the first thread, T1, has the lowest priority. Followed by the second thread, so it's Priority, P2. And then finally, the third thread, P3, 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 particular point in time, when time is doom, we'll have a very definite 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, P3. So when threads one and three become ready and they arrive at this particular moment, thread two is first going to be preempted and the execution of the thread three will start. So thread three has the highest priority. When thread three completes, at that point thread two has the lower priority. So we'll have to give the CPU to thread two. And now it's pretty unfortunate that for thread 1 we'll have to wait for all this time before it can execute. But a priority based scheduling is only going to look at the priorities of the threads. So thread 1 is not going to really start running until the 11th second in this time graph and then it will complete at time T12 as the entire schedule will complete at that time as well. In this example we were looking at this table, but in principal, our scheduler if it needs to be a priority based scheduler, it will somehow need to quickly be able to assess not just what are the runnable threads in this 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.

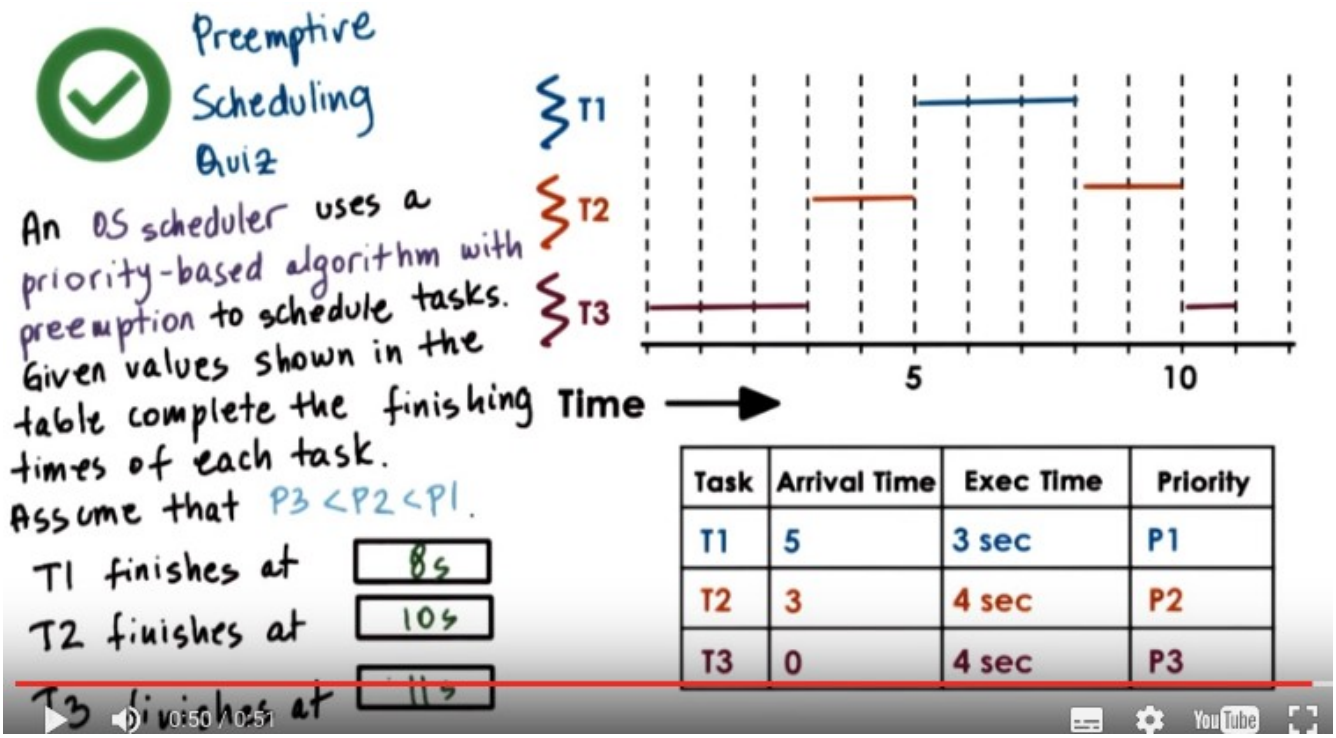


We can achieve this by having multiple run queue structures. Different run queue structures for each priority level. And then have the scheduler select a task from the runqueue that corresponds to the highest priority level. In this case that was the P3. Other than having per priority queues, another option would be to have some kind of ordered data structure. Like, for instance, the tree that resolve 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 tasks.

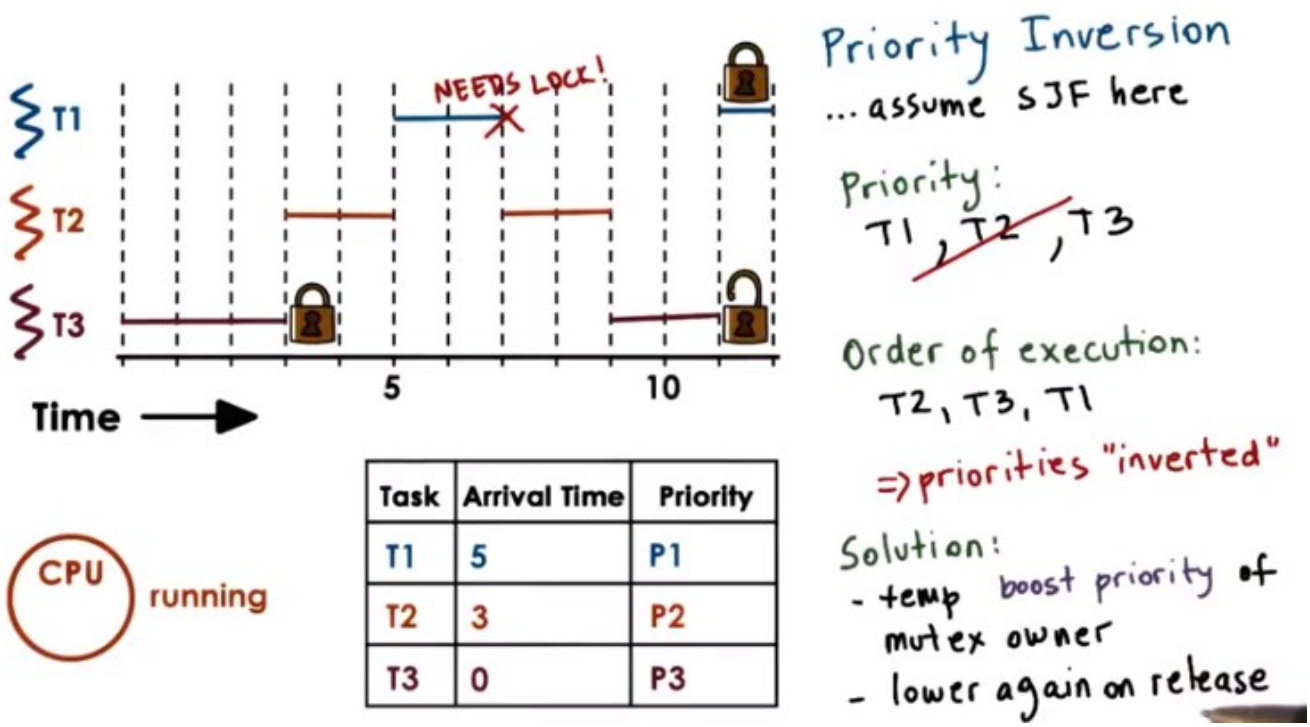


One danger with priority-based scheduling is starvation. We can have a situation in which a low priority tasks is basically infinitely stuck in a run queue just because they're constantly higher priority tasks that show up in some of the other parts of the run queue. One mechanism that we use to protect 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's some kind of function of the actual priority of the thread. Plus one other factor, and that is the time that the thread or the task spent (即 wait) in the run queue. The idea is that the longer a task spends in a run queue, the higher it priority should become. So eventually the task will become the highest priority task in the system, using this priority aging mechanism, and it will run. And in this manner, starvation will be prevented.

9. Let's take a quiz now in which we will compute some of the performance metrics for a preemptive scheduler. An OS scheduler uses a priority-based algorithm with preemption to schedule tasks. Given the values that are shown in this table, complete the finishing times for each of the three tasks. Also assume that the priorities are such that P3 is slower than P2 is, and then is slower than P1. Write the finish times of each of the tasks in the boxed areas provided here.



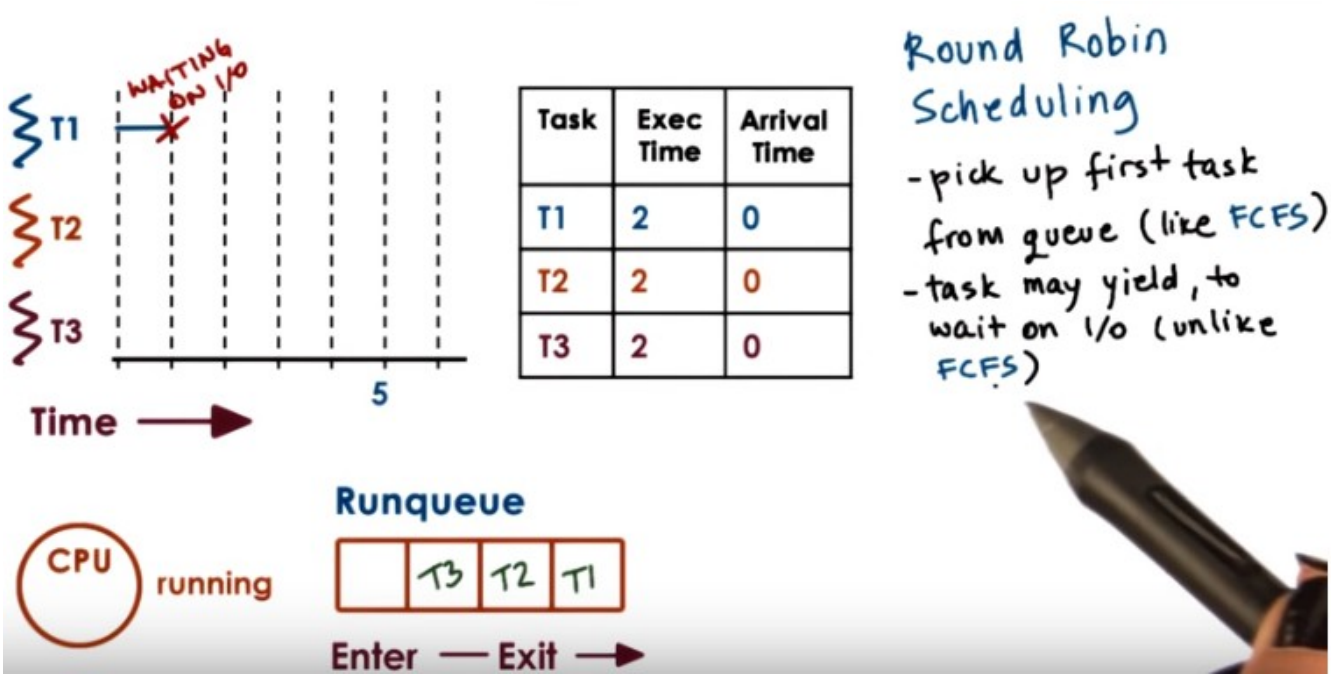
10. To answer this question, let's look at the execution scenario in this system. T3 is the first one to arrive in the system, and it will be the only task for a while. So, T3 will be the first one to be scheduled. At time, T3 it will be preempted, and task T2 will start executing, because it has higher priority than T3, at time T5. Task 1 will arrive, and that one will execute for three seconds. T1 will preempt the execution of T2, because T1 has higher priority than T2, according to this assumption. And then, we have two more seconds to finish the execution of T2, followed by the execution of the lowest priority task, T3. So, that one, we had one more second remaining. So, the finishing times of the three tasks according to this diagram are going to be 8 seconds, 10 seconds and 11 seconds, respectively.



T2 的第二段結束是因為 T2 自己跑完了.
右下角那個鎖是解開了的, 即 unlock 的意思. 右上角鎖那裡還有一小段 T1.
Pritority inversion 是 lock 造成的. 本來我們想的 priority 是 T1>T2>T3, 但最後變成了 T2>T3>T1.

11. 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 for scheduling to illustrate what happens. We have three threads (priority: P1 > P2 > P3), the first one has the highest priority. And P3 is the lowest priority, so T1 is the highest priority,` and T3 is the lowest one. For this example we have left out the execution time. Just assume they take some longer amount of time all these tasks and that the time graph (即 time line) continues into the future. Initially T3 is the only task in the system. So T3 executes, and lets say that T3 also acquired a lock. Now T2 arrives, T2 has higher priority than T3, so T3 will be preempted and T2 gets to execute. Now time 5, T1 arrives, it has higher priority than T2, so T2 will be preempted. And then T1 executes for two time units. And 假如 at that point, it needs to acquire a lock, and this happens to be the exact same lock that's already held by T3. Unfortunately that won't happen. T1 cannot acquire the lock so T1 at that point is put on a wait queue associated with the lock. We schedule the next highest priority task that's runable, that's going to be T2. And then T2 will execute as long as it needs. And let's say in this case, we're locked out. T2 really only needed to execute for two 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. In fact, T3 will execute for as long as it needs to, until it's releases the lock. Only after T3 re, release the lock will T1 become runnable and then once T1 becomes runnable again. It is the highest priority thread. So T1 will preempt T3 and T1 will continue it's execution. So 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 then T3, the lowest priority thread, will be the last one to complete. However, that's not what happened and the actual order of execution was as follows: T2, the medium priority thread; followed by T3, the lowest priority thread; followed by finally T1, the highest priority thread last in the system. So the priorities of these tasks were inverted. This is what we call Priority Inversion. A solution to this

problem would've been to temporarily boost the priority of the mutex owner. What that means is that, at this particular point when the highest priority thread needs to acquire a lock that's owned by a low 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 log is down by somebody else, however, instead of scheduling T2, we would have scheduled T3. Priority would have been temporarily boosted to that of T1, so it would have been higher than T2. And then T2 would have completed, and so at least we would have been able to go ahead and start executing T1 at this particular point. And then we would not have had to wait for T2 to complete for the medium priority threat to get in the middle. 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're temporarily boosting the priority of the mutex owner we should lower its priority once it release the mutex. The only reason why we were boosting its priority was so as to be able to schedule the highest priority threat to run as soon as possible. So 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.

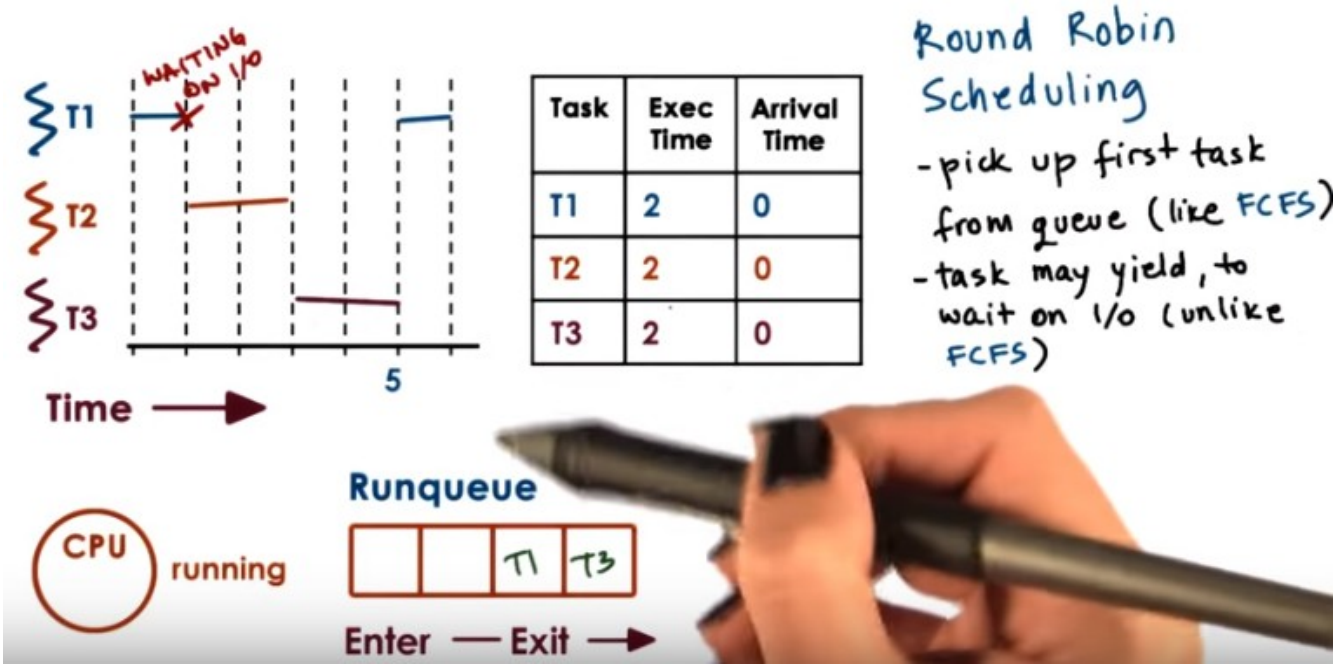


From wiki:
Round-robin (RR) is one of the algorithms employed by process and network schedulers in computing. As the term is generally used, time slices are assigned to each process in equal portions and in circular order, handling all processes without priority (also known as cyclic executive). Round-robin scheduling is simple, easy to implement, and starvation-free. Round-robin scheduling can also be applied to other scheduling problems, such as data packet scheduling in computer networks. It is an operating system concept.

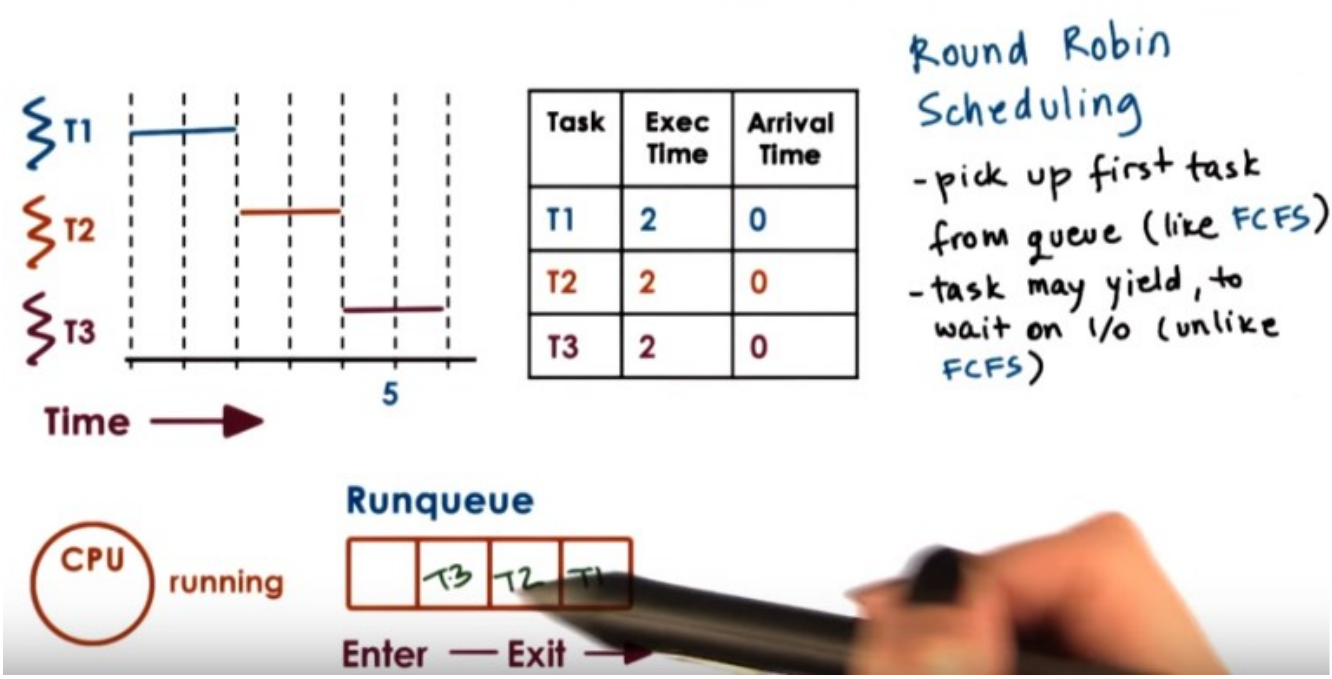
The name of the algorithm comes from the round-robin principle known from other fields, where each person takes an equal share of something in turn.

12. Now, when it comes to running tasks that have the same priority, we have other options in addition to the first-come first-serve or shortest job first that we discussed so far. A popular option is so-called a

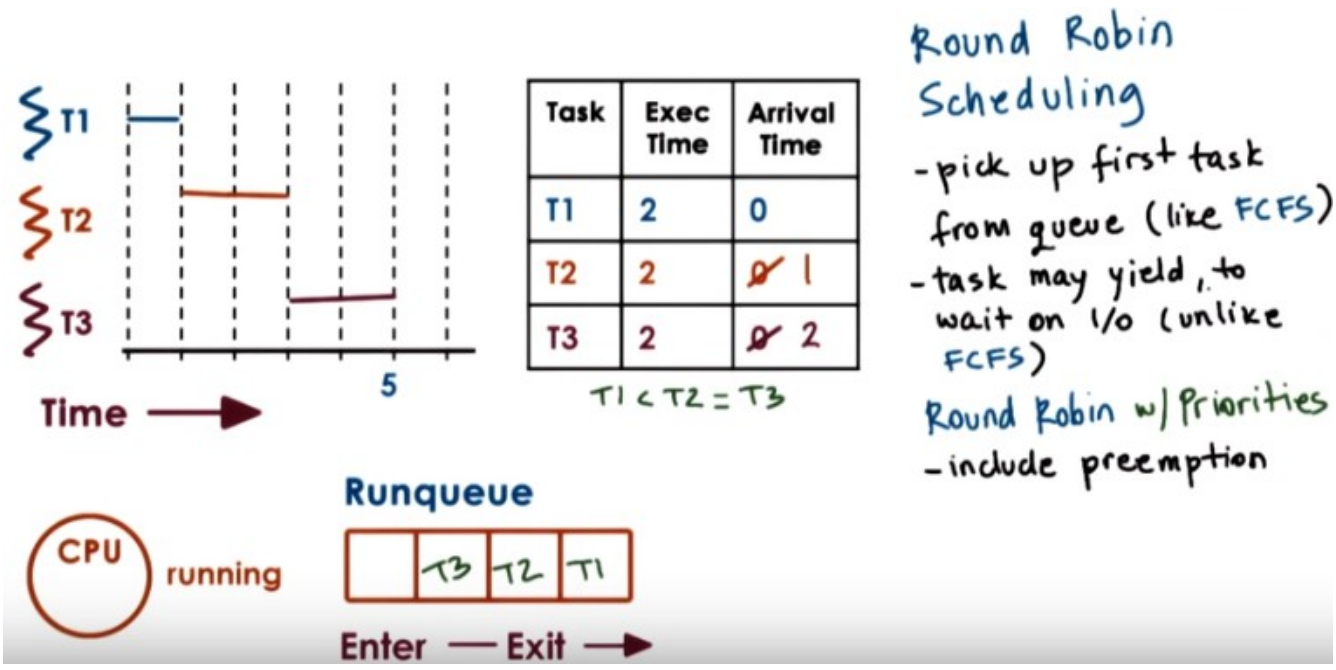
round robin scheduling. So let's say we have the following scenario. We have three tasks, they all show up at the same time, and they're all in the queue. With round robin scheduling, we'll pick up the first task from the queue just like with the first-come, first-serve scheduling. Now, let's say we pick up T1, that's the first in the queue, so T1 starts executing. Now, if they're executing for one time unit, the task stopped because it now needs to wait on an I/O operation. So it will yield the CPU and be blocked on that I/O operation. 然後 T1 被放在 queue 最後(即最左). 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. T3 will move to the front of the queue. Now, potentially, T1 will complete its I/O operation and will be placed at the end of the queue behind T3. Then when T2 completes, we will schedule T3. The execution time here we assume is only two time units. And then when that one completes, we'll finally pick up T1 from the queue and complete T1.



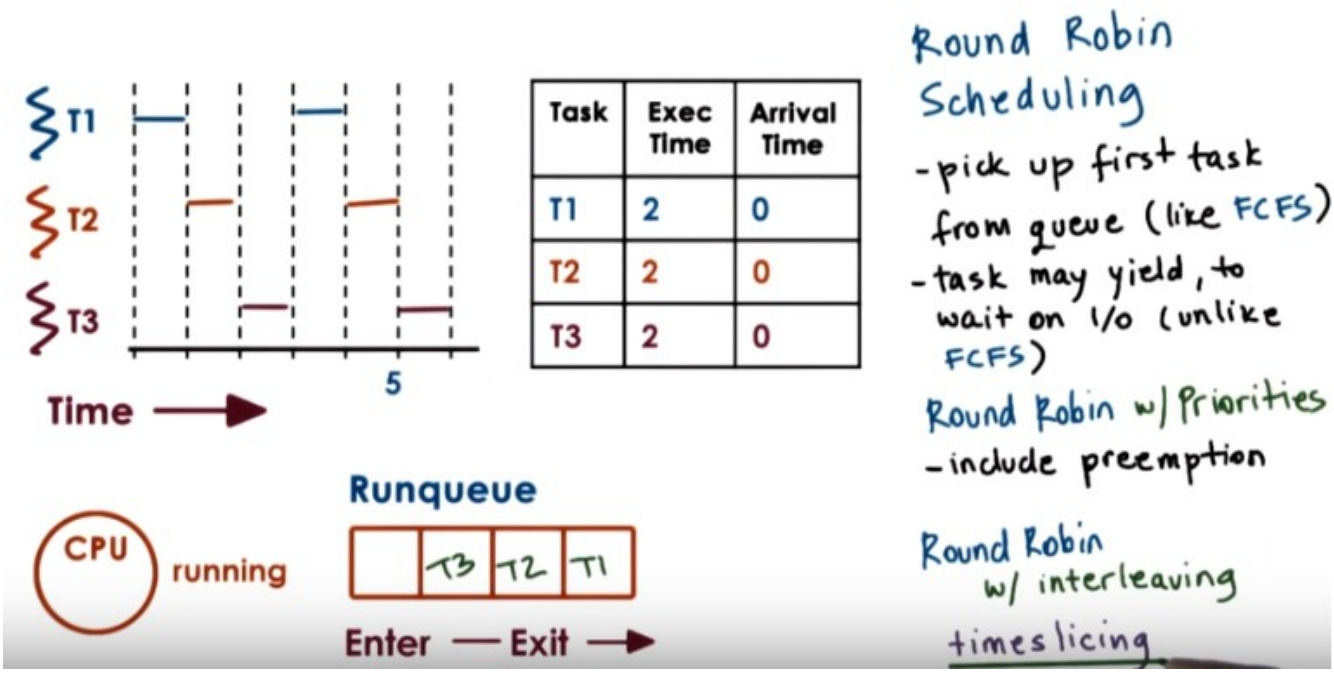
If T1 had not been waiting on I/O, then the execution based on T1, T2, T3, the order in which they were placed in the queue, would've looked something like this. So each of their tasks executes one by one in a round robin manner, and the queue is traversed in a round robin manner one by one.



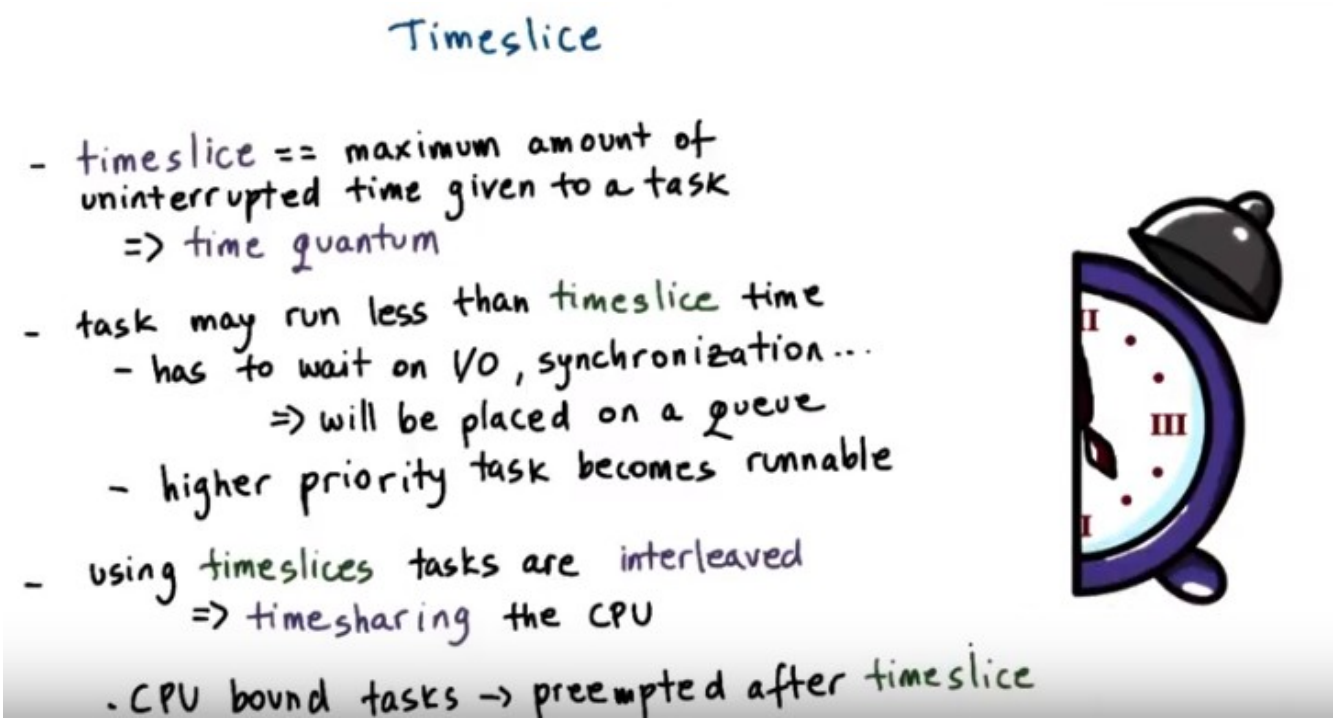
上圖右邊的幾點即 round robin scheduling 要尊守的.
注意上圖中的 priority: T1 < T2 = T3

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 a little later, and that their priorities are as follows, T1's

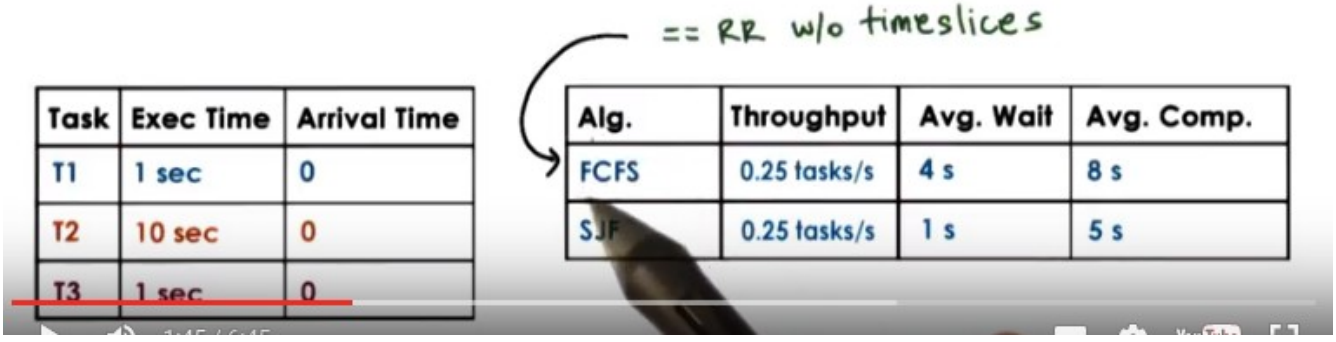
priority is the lowest and T3's is the highest. So in that case, what happens is that when a higher priority task arrives, the lower priority task will be preempted. If T2 and T3, however, have the same priorities, then the scheduler will just go round robin between them until they complete. So basically, in order to incorporate priorities, we have to include preemption, but otherwise the tasks will be scheduled from the queue like in first-come first-serve. 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 on the queue.



A further modification that makes sense for round robin is not to 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 tasks a time slice of one time unit. And then, after a time unit, we will interrupt them, we will preempt them, and we will schedule the next task in the queue in a round robin manner. So we will cycle through them, T1->T2->T3, and then again. We will focus our discussion next on timeslicing to better explain this mechanism.

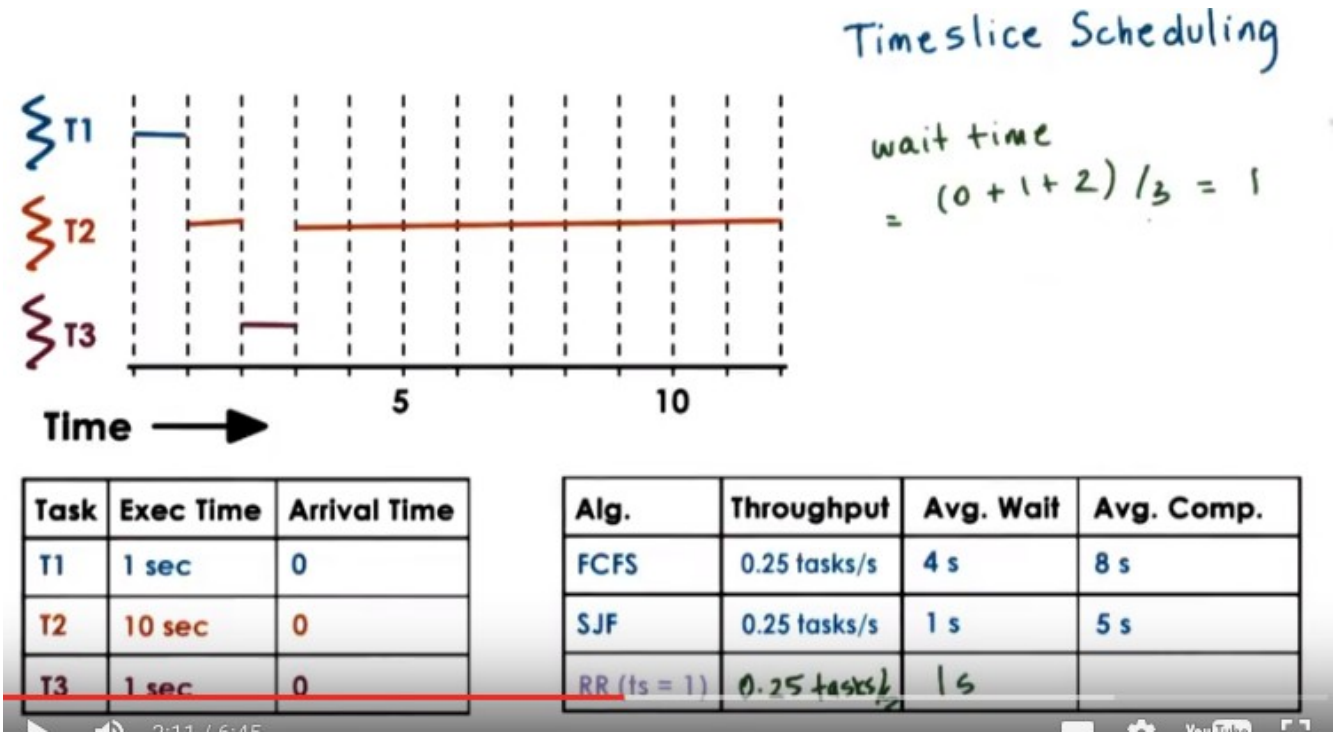


13. We mentioned timeslices very briefly in the introductory lesson on processes. To define it more specifically, a timeslice is the maximum amount of uninterrupted time that can be given to a task. It is also referred to as a time quantum. The timeslice determines the maximum amount, that means that a task can actually run a less amount of time than what the timeslice specifies. For instance the task may need to wait on an I/O operation or to synchronize with some other tasks in the system, on a mute tag that's locked. In that case the task will be placed on a queue, will 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 that the lower priority task will run less amount of time than the timeslice. Regardless of what exactly happens, the use of timeslices allows us to achieve for the tasks to be interleaved. They will be timesharing the CPU. For I/O bound tasks, this is not so critical since they're constantly releasing the CPU to wait on some I/O event. But for CPU bound tasks, timeslices is the only event that we can achieve time-sharing. They will be preempted after some amount of time as specified by the timeslice. And we will schedule the next CPU bound task.



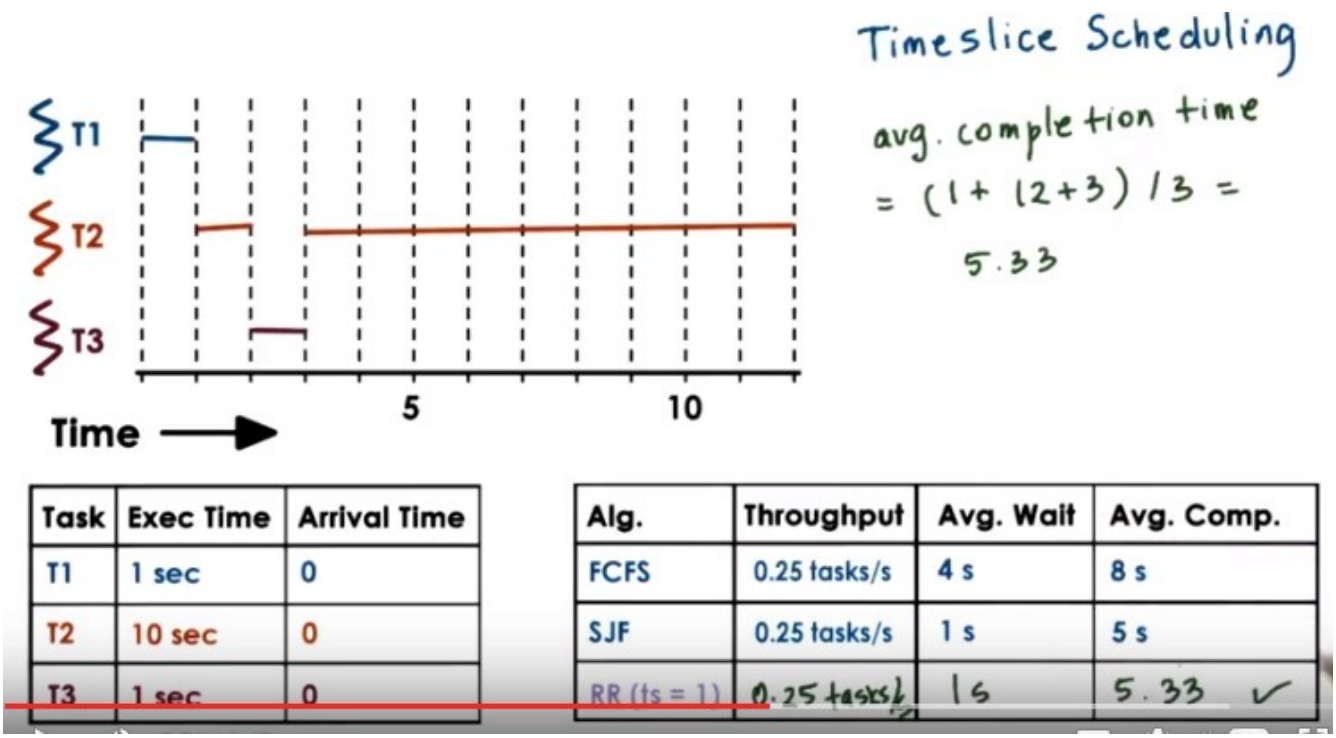
上圖右表中的結果是前面算過的.

Let's look at some examples now, consider for an instance the simple first-come-first-serve and shortest job first scheduler that we saw earlier in this lesson, they both had a same mix of task with same arrival times but led to different metrics. And, note that the metrics that we computed for first-come-first-serve also apply to a round-robin scheduler that doesn't use timeslices. Given that these tasks don't perform any I/O, they just execute for some fixed, specified amount of time. Then, round-robin would have scheduled them one and after another the way they showed up in the queue. And, that would have been identical to first-come-first-serve.



由視頻知, 上圖中的 wait time 不算 T2 第二段等別人的時間. 另一個例子見第 15 段.

Now let's calculate the metrics for a round-robin scheduler with timeslices, and let's say we'll first look at a timeslice of 1. The execution of these tasks will look as follows. T1 will execute for 1 second, that happens to be exactly the time that T1 requires, so T1 will complete. Then, with timeslicing, we would have preempted the execution of T1 anyways. We will execute T2. Now, T2 needs more time to execute, 10 seconds. So, it will actually be preempted. T3 will run for 1 second. At the time when we're about to preempt it, the T3 will anyways complete, and the only runnable task in the system is T2, and we will execute T2. So T2 will run for the remainder of the time. Now if we look at some of the metrics for throughput, we'll see we have the exact same thing. It's still takes us 12 seconds to complete these three tasks, so the throughput will again be the same as in the previous two systems. Looking at the wait time, we have a wait time of, 0, 1, and 2 for each of the tasks respectively. So wait time is 1 second.

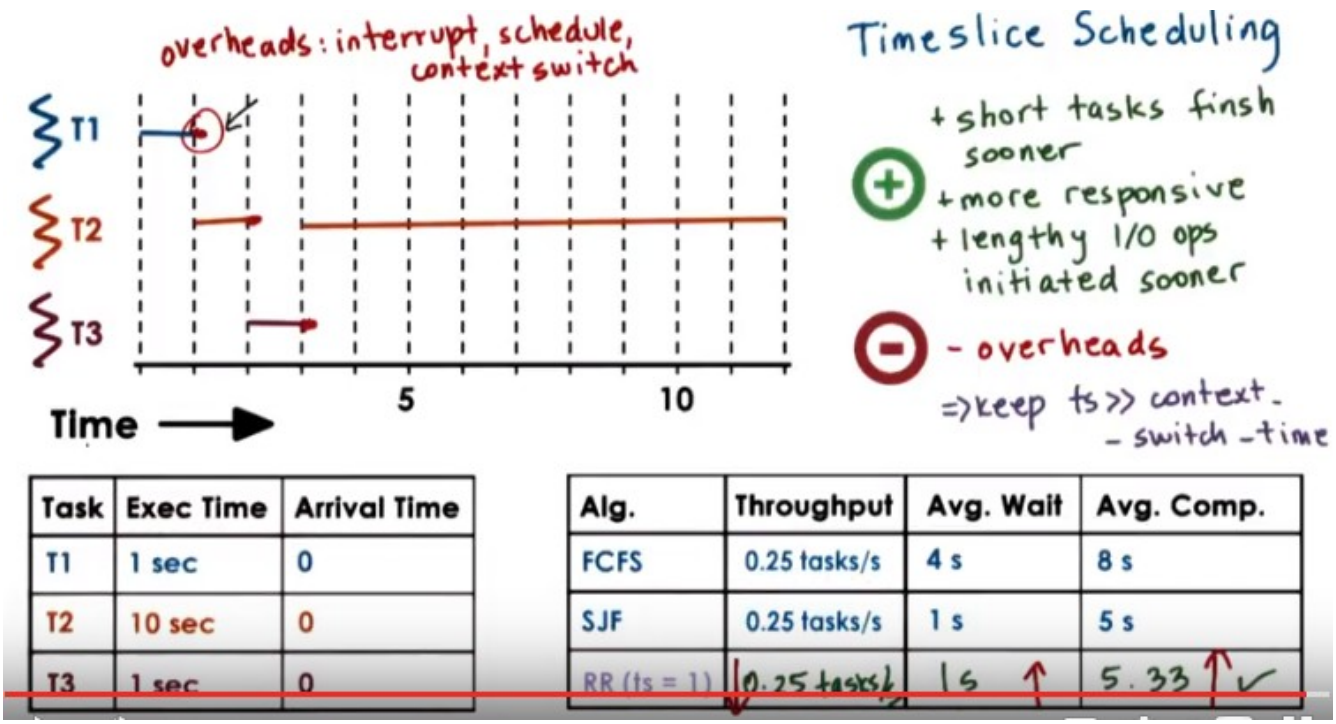


Task	Exec Time	Arrival Time
T1	1 sec	0
T2	10 sec	0
T3	1 sec	0

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 ✓

由視頻知, 上圖中 avg completion time 的算法是: T1 在 1s 結束的, T2 在 12s 結束的, T3 在 3s 結束的, 所以 avg completion time = (1 + 12 + 3) / 3.

If we look at the average completion time, the tasks complete at 1, at 12, and at 3 seconds respectively. So the average completion time is 5.33 seconds. So without even knowing what are the execution times of these tasks, with a timeslice of 1, we were able to achieve a schedule that's really close to this best one that we saw before, the SJF (shortest job-first) one. This is good. We keep some of the simplicity that we had in first-come-first-serve. We don't need to worry about the going out, what is the execution time of the task. And yet we're able to achieve good wait time and good completion time for all of the tasks.



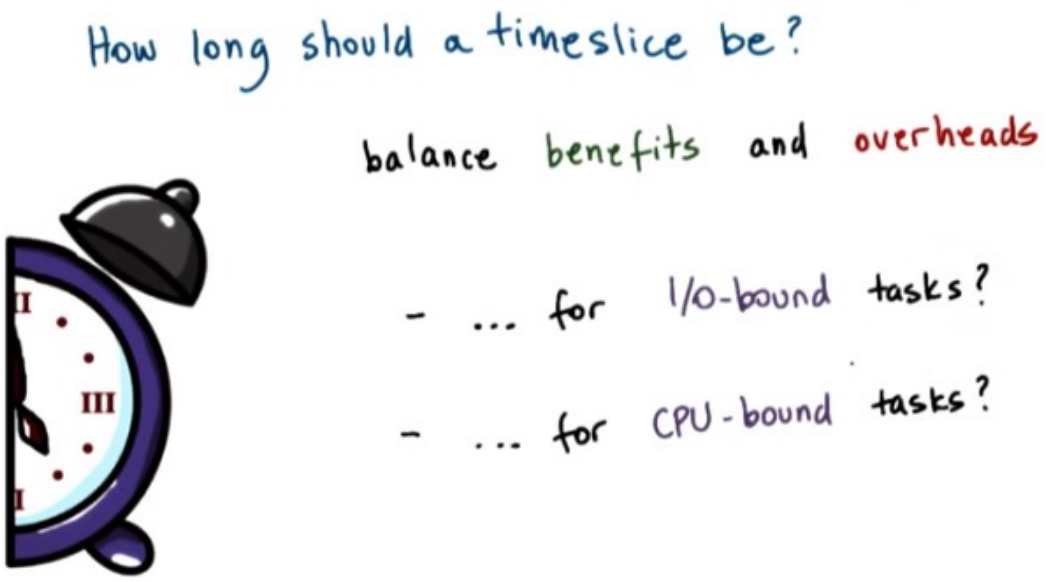
Task	Exec Time	Arrival Time
T1	1 sec	0
T2	10 sec	0
T3	1 sec	0

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

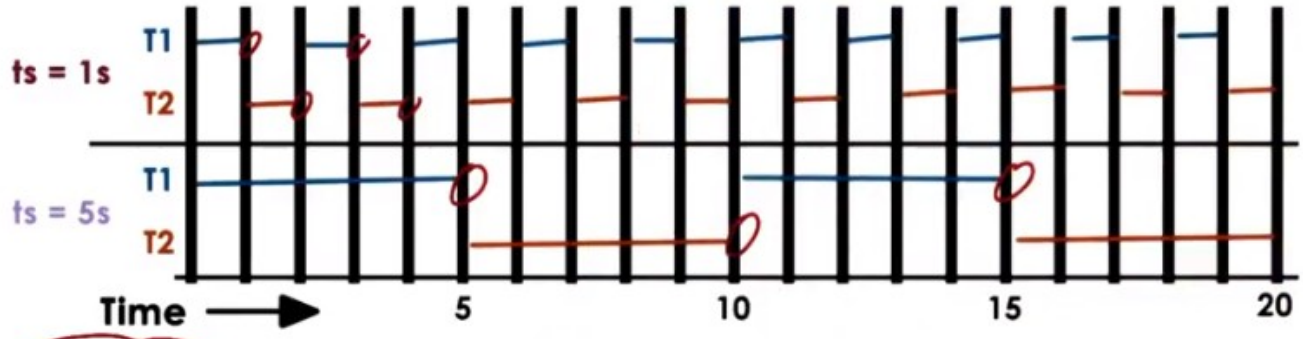
Keep ts 中的 ts 即 time slice

Some of the benefits of this timeslice-based method, particularly when the timeslice is relatively short like in this case, is that we end up with a situation where the short tasks finish sooner. So T3 was able to complete much sooner than in the first come, first serve case. And we're also able to achieve a schedule of that is more responsive and any I/O 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 preempted to squeeze in T3 at this point. If T2 needs to initiate any I/O 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, and because T2 would have only been scheduled after all of the shorter jobs complete, so T1 and T3 in that case. The downside is that we exaggerated so far a little in that we had these tasks immediately starting their execution after the previous one was interrupted. However, there's some real overheads. We have to interrupt a 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 which when we're scheduling from the context of task of one task to another. All of these times are pure overhead. There's no useful application processing that's done during those intervals. Know that these overheads, so if we have a timeslice of one these set time-outs for the timer will appear during the execution of task T2 except at that point since there are no other runable tasks in the system we're really not going to be scheduling or contact switching. And that's the dominant part of the overhead. And exactly how these sticks are handled, we're not really 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 timeslice and how it relates to the actual time to perform these context switching and scheduling actions, so as long as the timeslice value 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 timeslice.



14. We saw in the previous morsel that the use of timeslices 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 timeslice. We will see that to answer this question, how long should a timeslice be, the balance between these two differs if we're considering I/O-bound tasks, so tasks that mostly perform I/O operations, versus CPU-bound tasks, so tasks, these are tasks that are mostly executing on the CPU and don't do any I/O or do very little I/O. We will look at these two different scenarios next.



CPU bound tasks
- 2 tasks, exec. time = 10s
- ctx switch time = 0.1s

Alg.	Throughput	Avg. Wait	Avg. Comp.
RR (ts = 1)	0.091 tasks/s	0.55 s	20.85 s
RR (ts = 5)	0.098 tasks/s	3.05 s	17.75 s

Instructor Notes: Full Calculations + ERRATA (for avg. wait time)

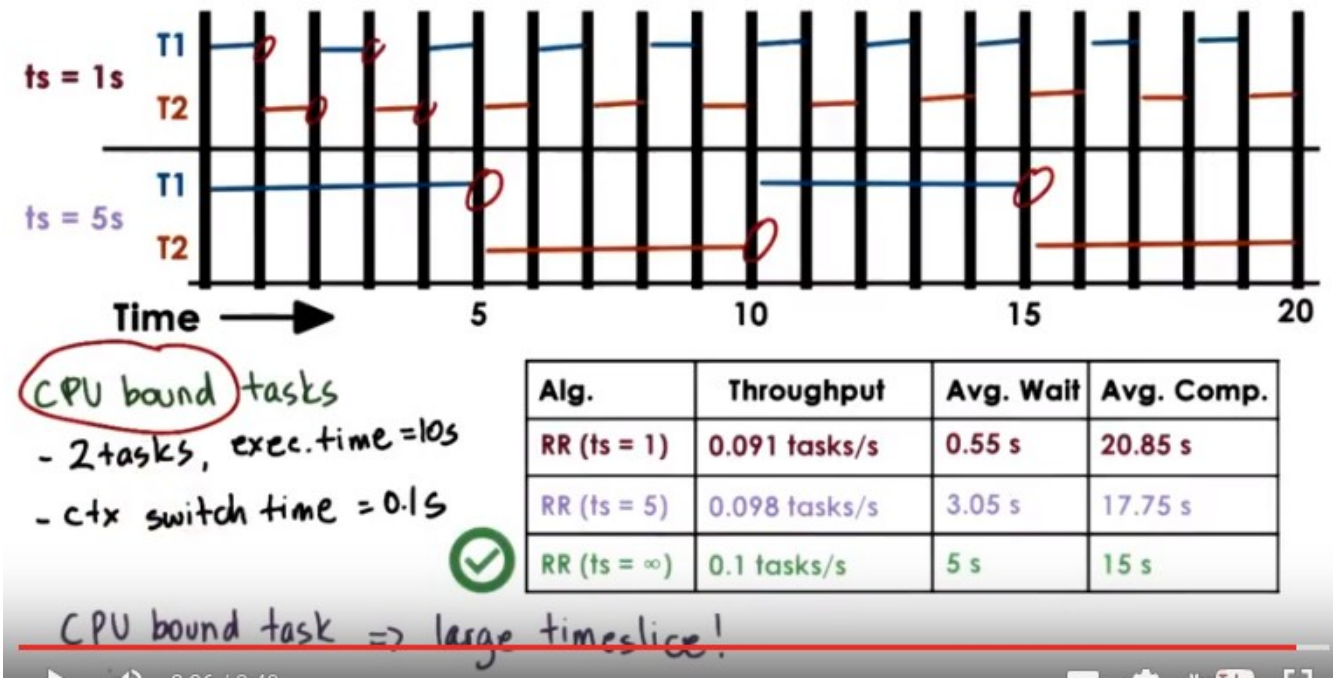
Timeslice = 1 second
throughput = 2 / (10 + 10 + 19*0.1) = 0.091 tasks/second

avg. wait time = (0 + (1+0.1)) / 2 = 0.55 seconds ← 注意算 wait time 時每個 task 只算一次
avg. comp. time = 21.35 seconds , 理由: ((19 + 18*0.1) + (20 + 19*0.1)) / 2 = 21.35 (先看下面紅字)
(注意上圖中的 20.85 是錯的)

Timeslice = 5 seconds
throughput = 2 / (10 + 10 + 3*0.1) = 0.098 tasks/second. ← 注意分母就是下面的 T2 總結束的時間. 下面 I/O bound 的情況也是這樣.
avg. wait time = (0 + (5+0.1)) / 2 = 3.05 seconds
avg. comp. time = 17.75 seconds 理由見下:
T1 總結束的時間: 15 + 2*0.1, 注意是 2*0.1, 而不是 3*0.1, 是因為 T1 第二段結束後, 就不用加入它帶的那個 ctx switch time 了
T2 總結束的時間: 20 + 3*0.1
avg. comp. Time =((15 + 2*0.1) + (20 + 3*0.1)) / 2 = 17.75

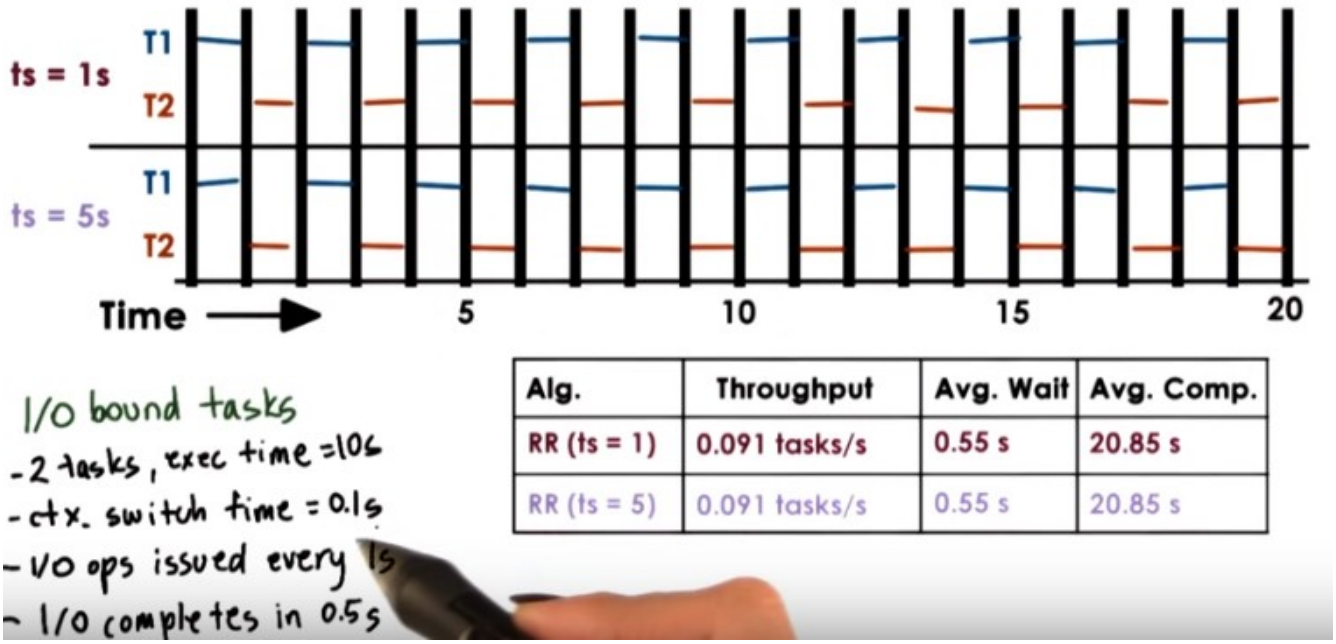
Timeslice = ∞ ← Just remember: 此時可忽略 ctx switch time. 可能是因為 10 >> 0.1.
throughput = 2 / (10 + 10) = 0.1 tasks/second
avg. wait time = (0 + (10)) / 2 = 5 seconds
avg. comp. time = (10 + 20)/2 = 15 seconds

15. Let's first look at an example in which we will consider two CPU bound tasks. So, these are tasks that are mostly just running on the CPU and don't perform any I/O. Let's assume their execution time is ten seconds, and in this system let's assume that the time to context switch from one context to another takes just 0.1 seconds. For this scenario now let's take a look at what will happen when we consider two different timeslice values, timeslice of one second, and timeslice of five seconds. With a timeslice of one second, the execution of these two tasks will look something as follows, and let's assume that the thicker vertical bars are trying to capture this context, which overhead in this case. For the timeslice value of five seconds, the schedule will look as follows. In the context switch overhead is only paid at this particular points. That's in contrast with having to context switch at every single one of these vertical bars here. Now if we compute the metrics for throughput, average wait time, and average completion time for these tasks, we will obtain the following results. To complete the throughput, we calculate the total time that it took to execute these tasks, and divide it by two, by the number of tasks. To complete the average wait time, we look at the start time of each of the tasks, and divide that by two, the number of tasks. And to complete the average completion time, we'll look at when each of the two tasks completed, and then we average that. And the detailed calculations for both of these are in the instructors notes. Looking at these metrics, we see that for throughput, we are better off choosing a higher timeslice value. For completion time, also, we're better off choosing a higher timeslice value. And then for the average wait time of the task, we are actually better off choosing a lower timeslice 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 (即不管過程中 wait 了多久, 只要完成得早就好. 由後面知, 應該是這麼個意思: CPU bound tasks 不關心 avg wait time, 而只關心 avg comp time, which makes sense), and overall when all of the tasks submitted to the system complete. So this really means that CPU bound tasks we're really better off with choosing a larger timeslice. This is the winning combination for us.



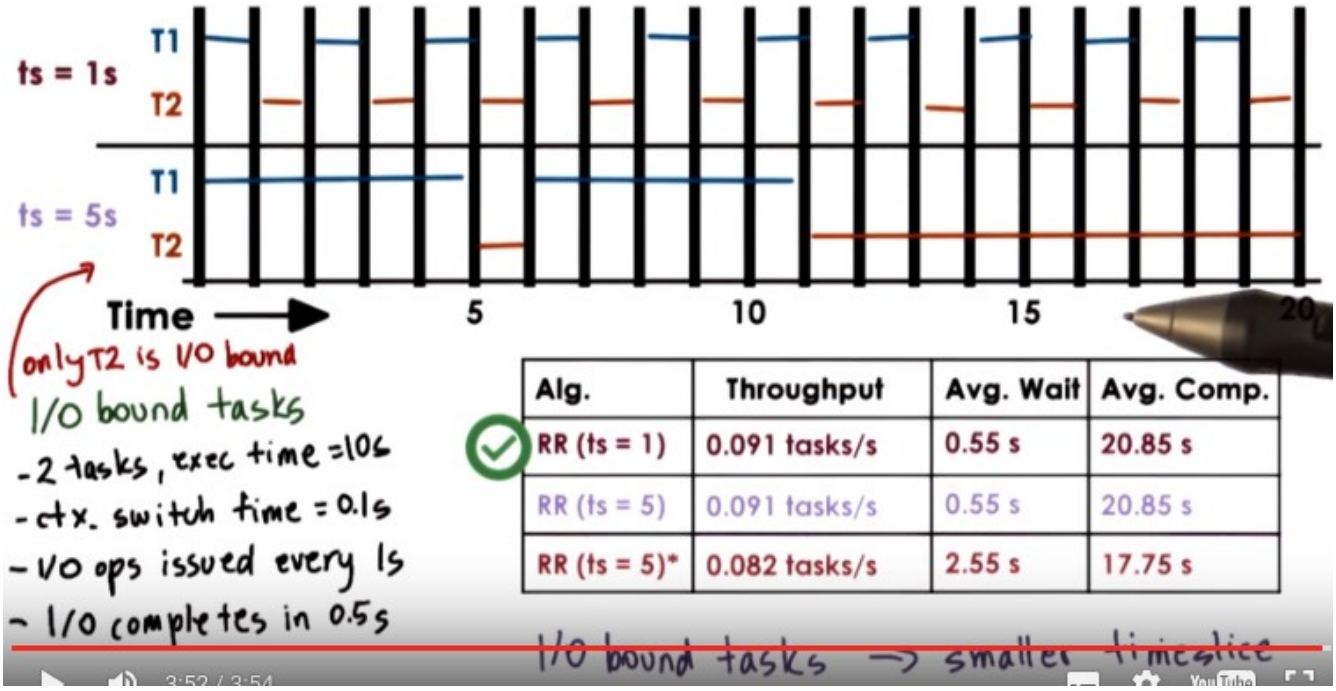
Alg.	Throughput	Avg. Wait	Avg. Comp.
RR (ts = 1)	0.091 tasks/s	0.55 s	20.85 s
RR (ts = 5)	0.098 tasks/s	3.05 s	17.75 s
RR (ts = ∞)	0.1 tasks/s	5 s	15 s

In fact, for CPU bound tasks, if we didn't know timeslicing at all, so like the timeslice value is infinite, we'd end up with absolutely the best performance in terms of the throughput and the average completion time, so the metrics that we care for when we run CPU bound tasks. Yes, of course, the average wait time of the task will be worse in that case, but we don't care about that. It's CPU bound task. We won't notice that. In summary, a CPU bound task prefers a larger timeslice.



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

16. Now, let's try to see what will happen if we're considering two I/O bound tasks. And again, we'll think of two tasks that have execution time of ten seconds, and in a system that has a context switch overhead of 0.1 second. And let's also assume that the nature in which these I/O calls are performed is that a task issues an I/O operation every one second. And also let's assume that every single one of those I/O operations complete in exactly half a second. If we look at the timeline, it looks identical as what we saw in the case for the CPU-bound jobs with a time slice of one second. This makes sense, because exactly after one second, the tasks are, in this case, not exactly preempted. They actually end up issuing an I/O operation, so yielding voluntarily, regardless of the fact that the time slice is 1. So if we look at the performance metric for this case, they will be identical to the previous case in the case of the CPU bound tasks(在 T2 運行的同時, T1 就在等 I/O, 所以 I/O 不佔用 execution time, T1 等 I/O 的時間被 T2 屏蔽了). Now if we look at the second case that has a time slice value of five seconds, we see that the timelines, the schedules of the two tasks, are identical to the case when we had a much smaller time slice value of one second. Similarly, obviously, if we compute the matrix, they will be identical, it's for two identical schedules. The reason for this is that at this particular moment, we're not exactly time slicing. We're not interrupting the tasks in either one of these cases. The I/O operation, again, is issued every one second. So regardless of the fact that in this scenario, the time slice value is much longer, we still end up issuing an I/O operation at the end of the first second. And therefore, the CPU is at this point released from T1, T1 yields, and T2 is free to be scheduled and to start executing. So one conclusion that you can make from this is that for I/O bound tasks, the value of the time slice is not really relevant (corrected in the followings).



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
RR (ts = 5)*	0.082 tasks/s	2.55 s	17.75 s

Instructor Notes: Full Calculation + ERRATA

for Timeslice = 1sec (前面已算過)

avg. comp. Time = (21.9 + 20.8) / 2 = 21.35 , 注意上圖中的 20.85 是錯的

Timeslice = 5 second*

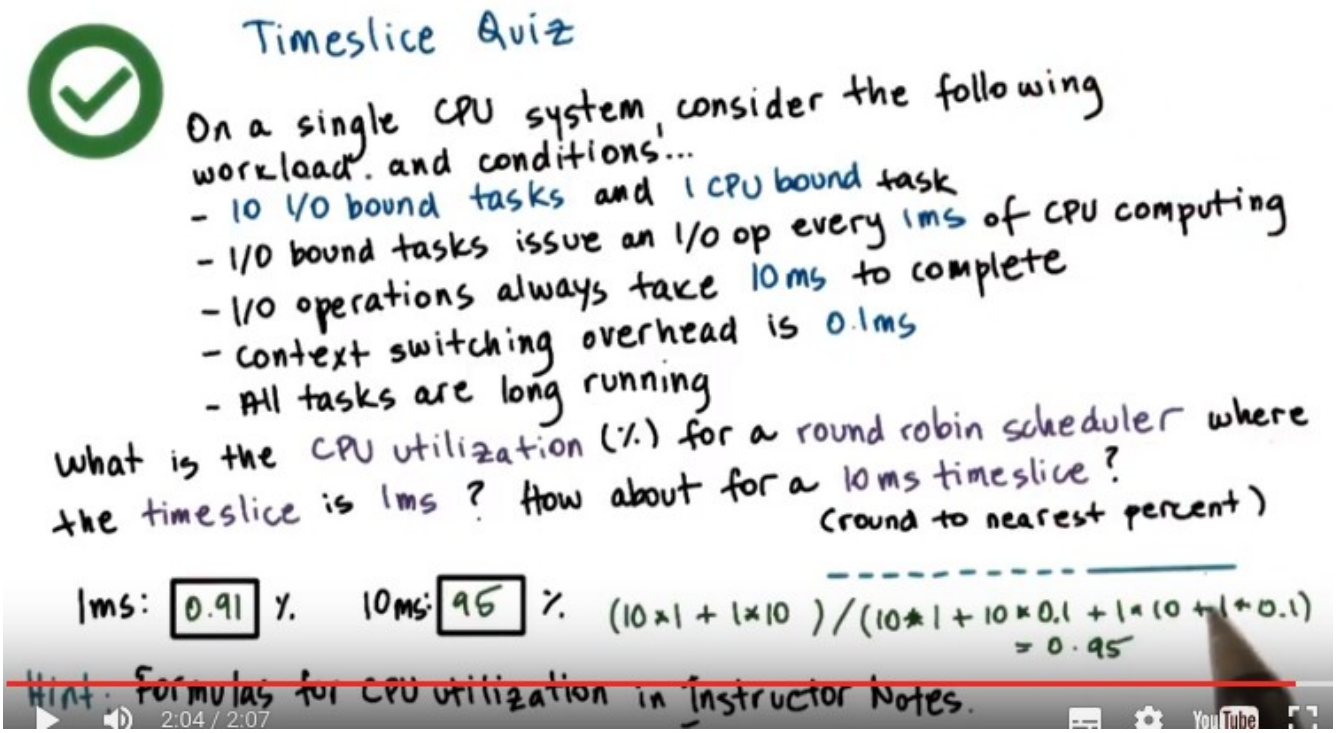
throughput = 2 / 24.3 = 0.082 tasks/second 24.3 理由見下
avg. wait time = 5.1 / 2 = 2.55 seconds
avg. comp. time = (11.2 + 24.3) / 2 = 17.75 seconds
T1 總結束的時間: 11 + 2*0.1 = 11.2, 注意 T2 第一段的等 I/O 不佔用時間, T2 第一段等 I/O 的時間被 T1 第二段屏蔽了 , 原因見前
T2 總結束的時間:
T1 兩段加 T2 第一段: 11 + 3*0.1
T2 第二段 9 + 8*0.5, 因為 T2 第二段中, T2 每 1s 就要等 0.5s 的 I/O, 此時沒人屏蔽, 所以 I/O 要佔用時間了
(11 + 3*0.1) + (9 + 8*0.5) = 24.3

Well, let's not draw that conclusion that fast. Let's look first at a scenario where only T2 is an I/O bound task. T1 is a CPU bound task, as what we saw in the previous morsel. In that case, the execution for the two tasks T1 and T2 when the timeslice is 1 will look the same. The one difference is that in the event of T1, we preempted after one second, where as in the case of T2, the I/O bound task, after one second, it voluntarily yields since it has to go and wait for an I/O operation. In the case of five seconds, the execution of T1 and T2 will look something as follows. T1 will run for five seconds, and at that point, its time slice will expire, so it will be preempted. T2 will be scheduled, and as an I/O bound task, T2 will yield after one second because of reading on an I/O operation. At that point, T1 will be scheduled again. Now, at this point, T1 is actually complete, so T2 is the only runnable task in the system, and that's why we have this illustrated in this manner. If we work out the performance metrics for this last case, the numbers will look as follows. And again, for all of these, the calculations are posted in the instructor notes. 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, then, if you look at the calculations, there is a huge variance between the completion time of T1, which is at 11(主要是因為 T1 完成得早), and then T2, which is way later. We see from this that for I/O bound tasks, using a smaller time slice is better. The I/O bound task with a smaller time slice has a chance to run more quickly, to issue an I/O request, or to respond to a user. And with a smaller time slice, we're able to keep both the CPU as well as the I/O devices busy, which makes, obviously, the system operator quite happy. MO: timeslice 越短, 等 I/O 的時間越容易被別人屏蔽.

17. Let's summarize quickly on our question: how long should a timeslice be? CPU bound tasks prefer longer timeslices. The longer the timeslice, the fewer context switches we'll have to perform, so that basically limits the context switching overheads that the scheduling will introduce. To perform useful work, the useful application processing to as slow as possible. And as a result, both the CPU utilization, as well as the system throughput as metrics will be as high as possible. On the other hand, I/O bound tasks prefer shorter timeslices. This allows I/O bound tasks to issue I/O 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 use, the system is responsive and that it responds to its commands and displace output.

18. Let's take a quiz that looks at a problem involving timeslices and their values. On a single CPU system, consider the following workload and conditions. We have ten I/O bound tasks in the system and one CPU bound task. The I/O bound tasks issue one I/O operation every one milliseconds of CPU compute time. I/O operations always take 10 milliseconds to complete. The context switching overhead is 0.1 millisecond. And at the end also all tasks are long running. So we're not reporting any kind of execution time for the tasks. Given this, answer first, what is the CPU utilization for a round robin scheduler with a timeslice of one millisecond. How about for a 10 millisecond timeslice?

Provide the two answers here, and round them up to the nearest percent value. The formula for computing CPU utilization is provided in the Instructor's Notes.



From sample final 第 1 題:
算 CPU utilization 和 I/O utilization 時, 最後一節的 ctx switch 要算在內
算 throughput, avg wait time, avg comp time 時, 最後一節的 ctx switch 不算在內

實測表明, 以上 1ms 的答案寫錯了, 應該是 91%, 而不是 0.91%

1 / (1 + 0.1) = 91%, 理由見下面講的

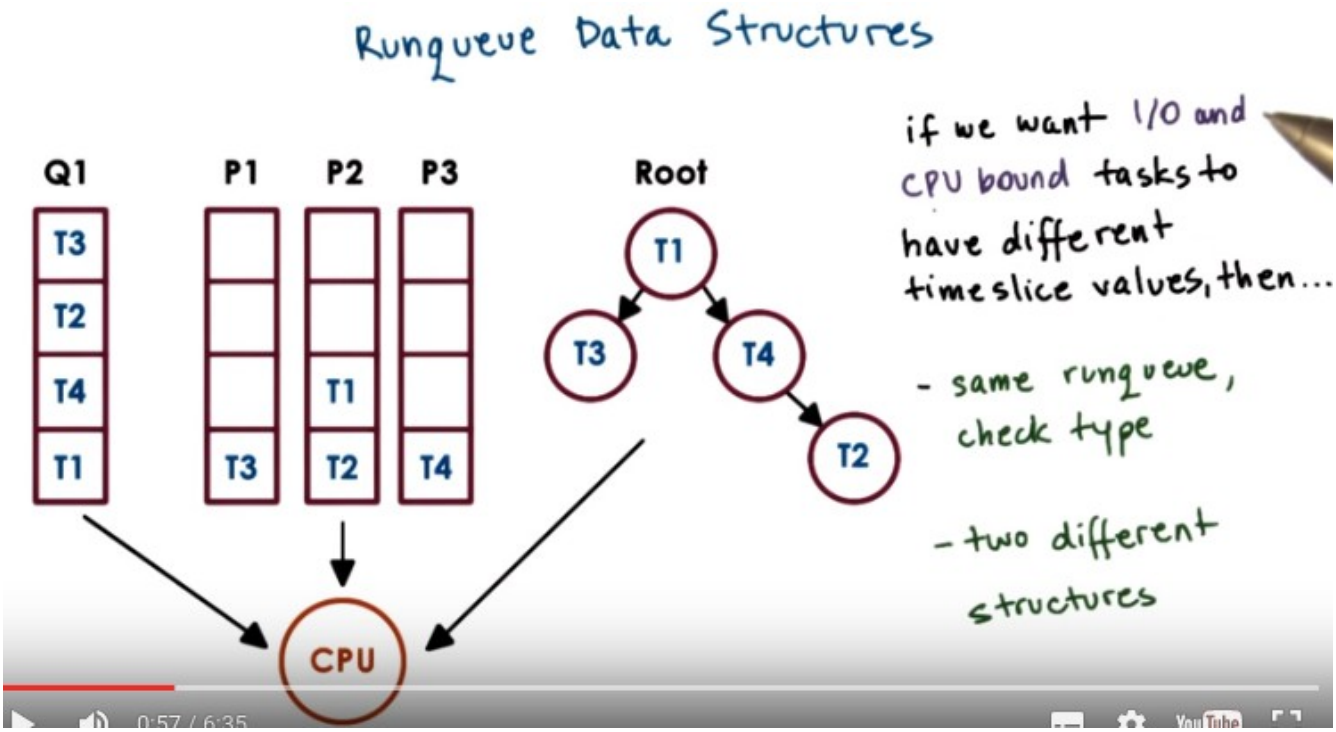
Instructor Notes
Quiz Help

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:

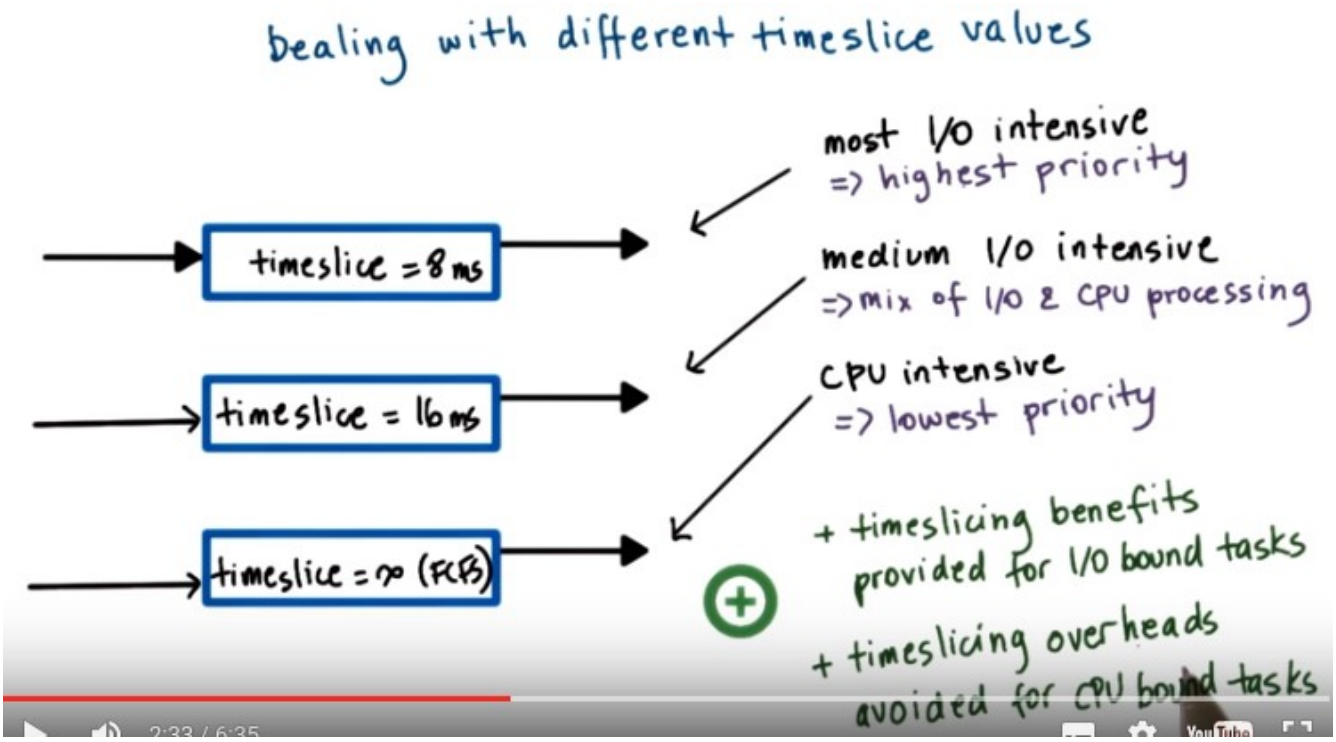
Determine a consistent, recurring interval
In the interval, each task should be given an opportunity to run
During that interval, how much time is spent computing? This is the cpu_running_time
During that interval, how much time is spent context switching? This is the context_switching_overheads
Calculate!

19. In the case when the time slice is one millisecond, every one millisecond either we will be preempting the CPU bound task or the I/O bound tasks on their own will be stopping because they need to perform an I operation every one millisecond. So that means for every one millisecond of useful work, we have total of one millisecond of the useful work, plus 0.1 millisecond of the context switching overhead. So, total useful CPU utilization is 91% (I/O 應該都被屏蔽了, 因為別外還有 9 個 I/O bound task 和 1 個 CPU bound task). For the case when we have a round robin scheduler with a 10 millisecond timeslice, as we're going through the 10 I/O bound tasks, every single one of them will run just for one millisecond and then we will have to stop because of issuing an I/O request, so we'll have them context switch in that case. So we will perform for the I/O bound tasks, 以下# #之間說的即(10*1 + 10*0.1) # 10 times 1 millisecond of useful work, and 10 times 1 millisecond plus 10 times 0.1 millisecond for the context switch #. So this is the total, amount of total work that has been performed. And then finally the CPU bound tasks will be scheduled and that 以下# #之間說的即(1*10 + 1*0.1) # one will run for full 10 milliseconds because the timeslice value is 10 milliseconds. So that will complete 10 milliseconds of useful work and then it will complete total of 10 milliseconds plus a context switch time of a total time #. If we compute this, this comes out too close to 95%. So, what this example shows us is that from the CPU's perspective, from the CPU utilization perspective, having a large timeslice that favors the CPU bound task is better. We didn't ask the question of what is the I/O utilization for both of these cases. Likely if we work out the math for that case, we will see that from the perspective of the I/O device, it is better to have a smaller timeslice. And that will be because the I/O device really cannot do anything during this entire period, when the CPU bound tasks is running for 10 milliseconds with the long time slice.



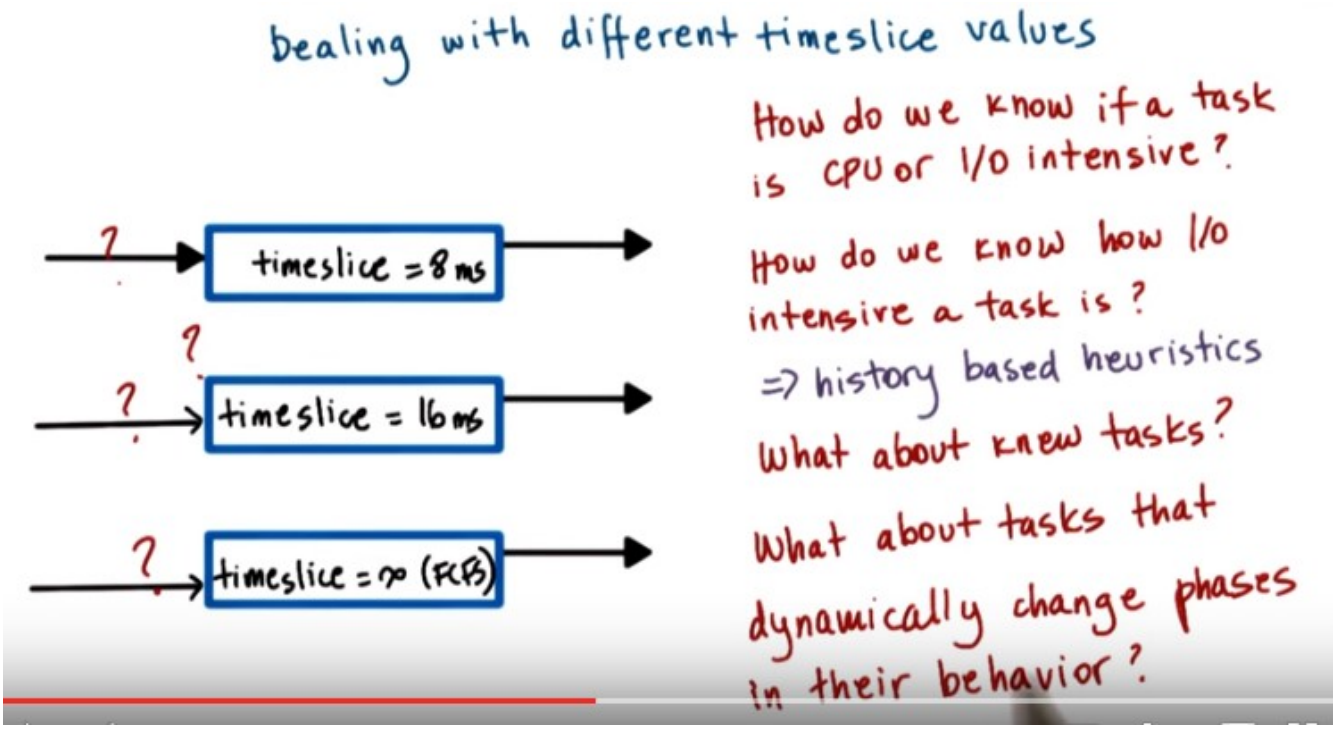
20. We said earlier that the runqueue is only logically a queue. It could also be represented as multiple queues, like when dealing with different priorities, or it could be a tree, or some other type of data structure. Regardless of the data structure, what is important is that it should be easy for the scheduler to find the next thread to run, given the scheduling criteria. If we want the I/O and CPU bound tasks in the system to have different timeslice values, we have two options. The first option is to maintain a

single runqueue 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 I/O and CPU bound tasks into two different data structures, two different runqueues, and then with each runqueue associate a different kind of policy that's suitable for CPU versus for I/O bound tasks.

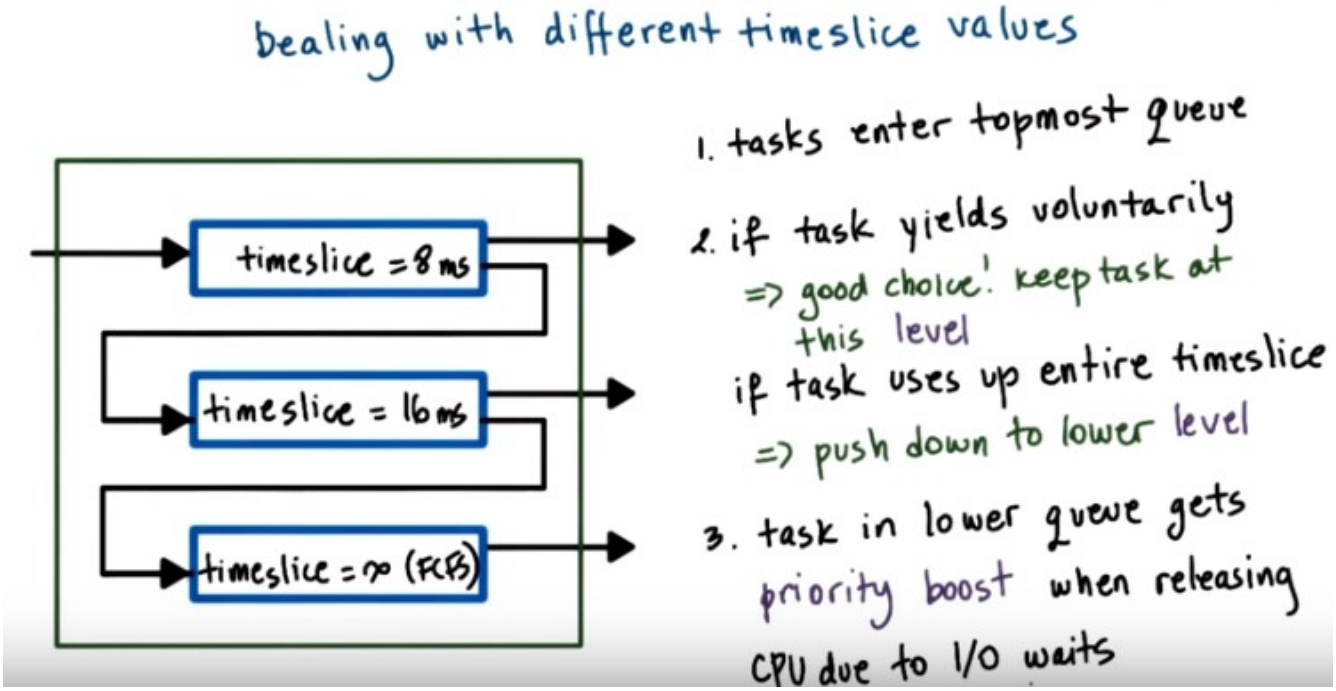


上圖簡記為: I/O, small timeslice, high priority (ISH).
注意跟 21 段的 I/O, high timeslice, high priority, (IHH)對比.

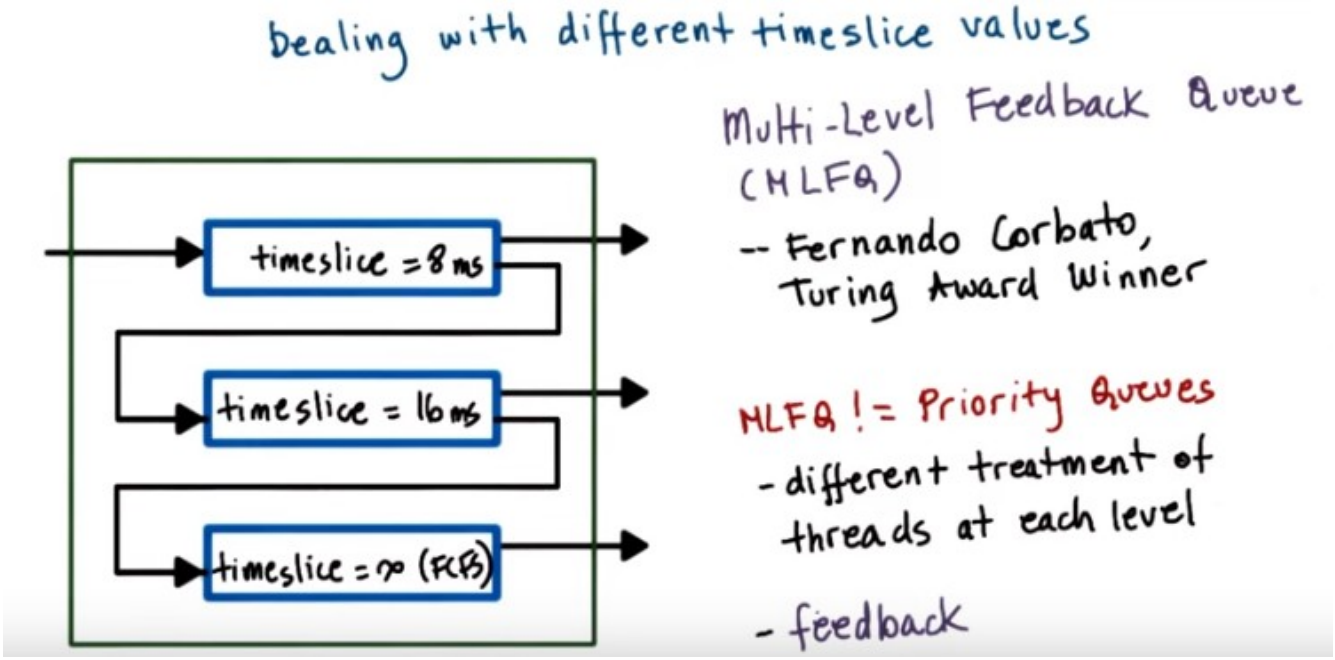
One solution for this, is this type of data structure that we will explain now. It is a multi queue data structure, that internally it has multiple separate queues. Let's say in the first round of queue we'll place the most I/O intensive tasks, and we will assign with this round queue a timeslice of 8 milliseconds. Let's say for the tasks that are of medium I/O intensity, so they have a mix of I/O and CPU processing, we have a separate queue in this multi-queue data structure. And here we assign with this queue, a timeslice of 16 milliseconds. And then for all of our CPU intensive tasks, we'll use another queue in the multi-queue data structure. And here we'll associate with this timeslice, basically that's infinite. So this will be like the first come, first serve policy. >From the scheduler's perspective, the I/O intensive tasks have the highest priority, which means that the scheduler will always check on, on this queue first. The queue that's associated with the I/O Intensive Tasks. And the CPU-bound tasks will be treated as tasks with lowest priority. So, this queue will be the last one to be checked when trying to figure out what is the next task to be scheduled. So, depending on the type of task that we have, we place it in the appropriate queue. And on the other side, the scheduler selects which task to run next. Based on highest priority, medium, and then lowest. So in this way we both provide the time slicing benefits for those tasks that benefit for the I/O bound tasks, and avoid the time slicing overhead for the CPU bound tasks.



But how do we know if a task is CPU or I/O intensive? How do we know how I/O intensive is the task? Now we can use for that some history based heuristics, like slide the task run 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, or what about tasks that have dynamic changes in their behavior?

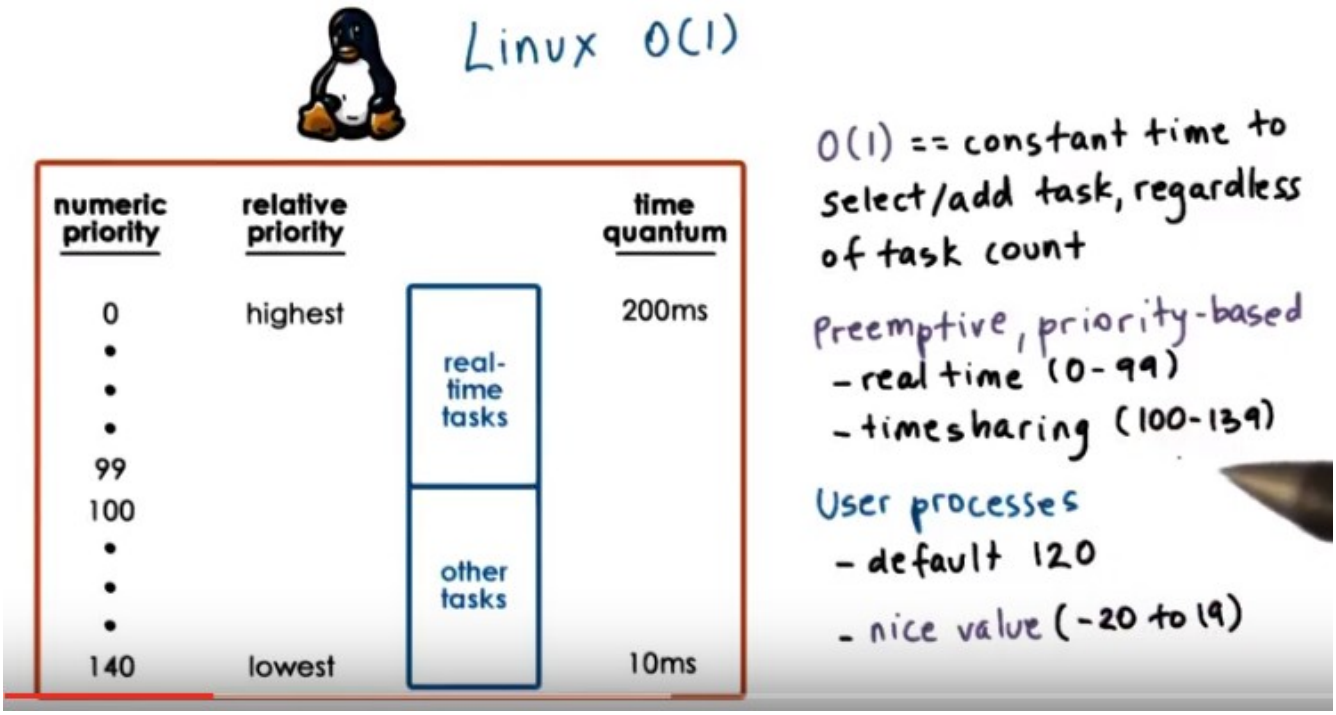


To deal with those problems, we will treat these three queues not as three separate runqueues, but as one single multi-queue data structure. This is how this data structure will be used. When a task first enters the system, so a newly created task will enter it in the topmost queue. The one that has the lowest timeslice associated with it. It's like we're expecting that it's the most demanding task. When it comes to these scheduling operations that it will need to be context which most often. If the task stops executing before these 8 milliseconds, so whether it yields voluntarily or it stops, because it needs to wait an I/O operation. That means we made a good choice. The task is indeed fairly I/O interactive, and so we want to keep the task in this level. So next time around, when it becomes runnable, after that I/O operation completes, it will be placed in this exact same queue. However, if the task ends up using up its entire timeslice. That means that it was more CPU intensive than we originally thought. So we will push it down the next level. It will be preempted from over here (最上面那個 queue), but then the next time it needs to run, it will actually be pushed into this queue (中間那個 queue), so it will be scheduled from this queue (中間那個 queue). If the task ends up getting preempted when it was scheduled from this queue (中間那個 queue), so it used up its entire 16 millisecond time slice. That means that it's even more CPU intensive. So in that case it will even get pushed down to the bottom queue. So we basically have a mechanism to push the task down these levels based on its historic information. Although we didn't know if a task is I/O CPU intensive to start with. We made an assumption, and then we were able to correct it. So we assume that it's I/O intensive. And we were able to correct and push it down these levels, down to the lowest most level, in case it's CPU intensive. Now if a task that's in one of the lower queues all of a sudden starts getting repeatedly releasing in the CPU earlier, then whatever the timeslice specifies, because it is waiting on our operation. There will be a hintto the scheduler to say, oh, well, this task is more I/O intensive than I originally thought. And it can push it up at one of the queues that are on the higher levels.

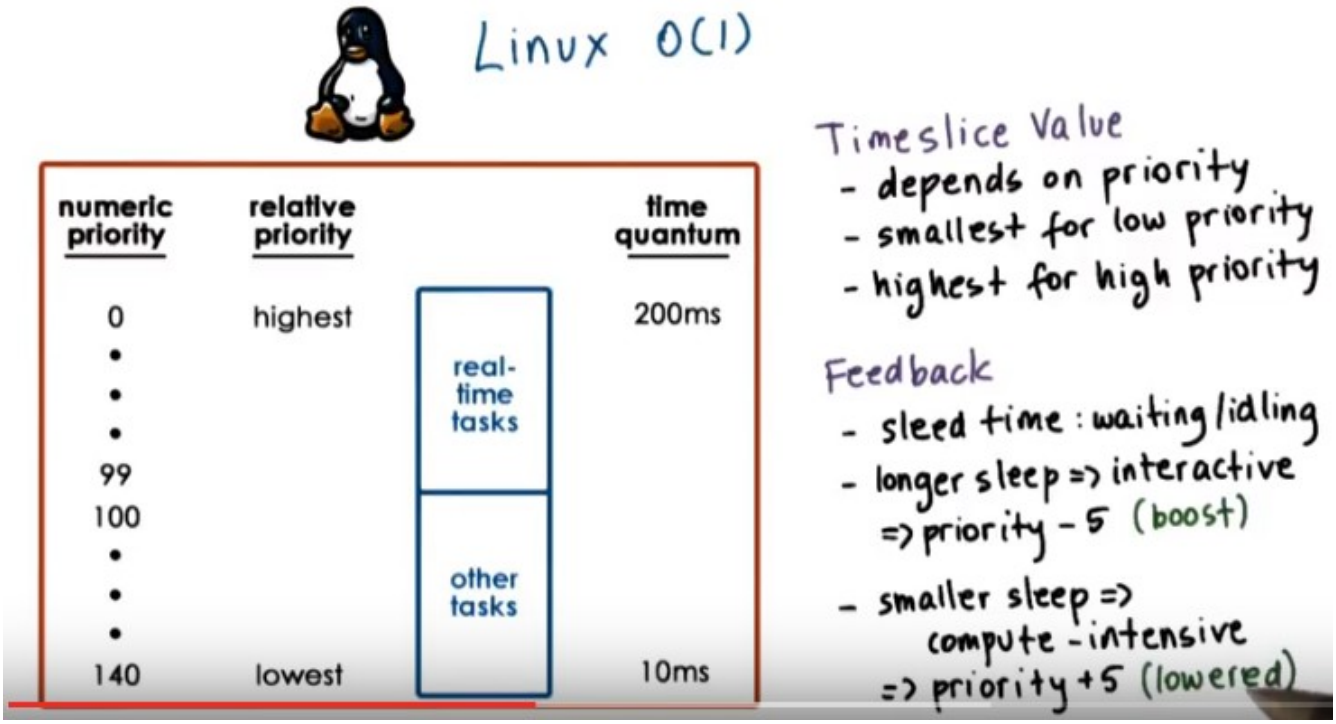


上圖中的 != 即 不等於

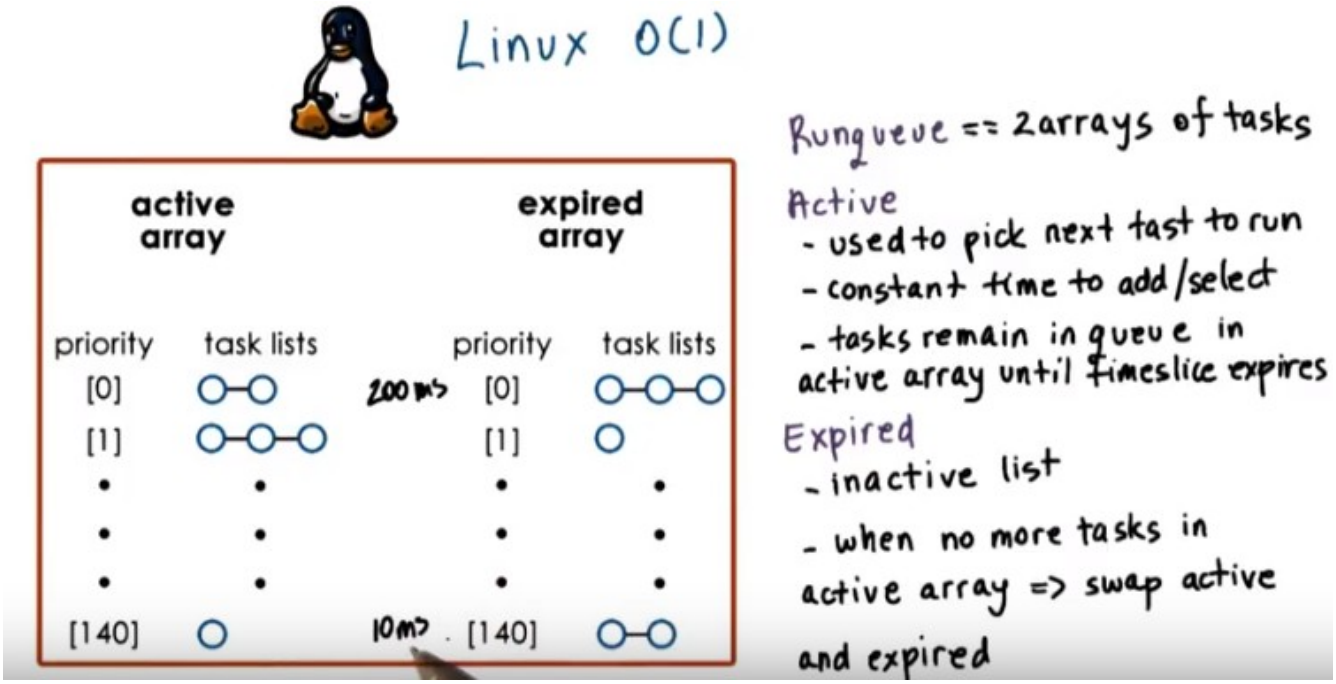
This resulting data structure is called the multi-level feedback queue. And for the design of this data structure, along with other work on time sharing system, Ferdando Corbato received the Turing Award, which is like the highest award in computer science. It's the equivalent of the Nobel Prize for computing. I want to make sure you don'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 the different levels that are part of this data structure. More uniquely, however, this data structure incorporates this feedback mechanism, that allows us over time to adjust which one of these levels will be place a task, and when we're trying to figure out what is the best time sharing schedule for the subtask in the system. The Linux, so called O(1) scheduler, that we will talk about next, that uses some of the mechanism borrowed from this data structure as well. And we won't describe the Solari scheduling mechanism. But I just want to mention that that's pretty much a multi-level feedback queue with 60 levels. So 60 subqueues. And also some fancy feedback rules that determine how and when a thread gets pushed up and down these different levels



21. Let's now look at couple of concrete examples of schedulers that are part of an actual operating system. First, we will look at the so called O(1) scheduler in Linux (O(1)念為 O of one). The O(1) scheduler receives it's name because it is 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 total of 140 priority levels, with zero being the highest and then 139 the lowest. These priority levels are organized into two different classes, the tasks, the priority levels from zero to 99 fall into a class of real time tasks, and then all others fall into a so called time sharing class. All user processes have one of the time sharing priority levels. Their default priority is 120 but it can be adjusted with a so called nice value. There's a system call that can be used to do this. And the nice values can be between negative 20 and 19, so as to span the entire set of time sharing priorities.

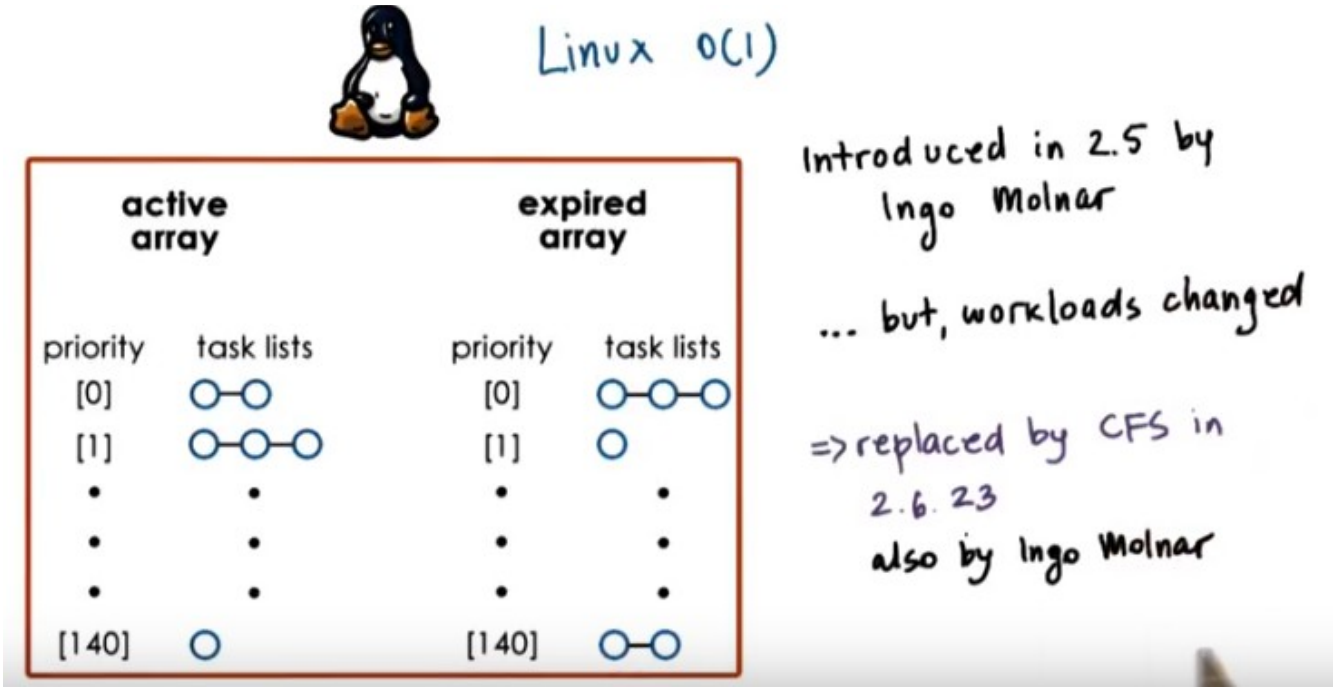


The O(1) scheduler borrows from the multilevel feedback queue scheduler, in that it associates different timeslice values with different priority levels. And it also uses some kind of feedback from how the tasks behaved in the past, to determine how to adjust their priority levels in the future. It differs, however, in how it assigns the timeslice values to priorities and how it uses the feedback. It assigns timeslice values based on the priority level of the task, similar to what we saw in the multilevel feedback queues in the scheduling. However, it assigns smallest timeslice values to the low priority, CPU bound tasks and it assigns high timeslice values to the more interactive high priority tasks (即 I/O, high timeslice, high priority, (IHH), 注意跟第 20 節的 I/O, small timeslice, high priority (ISH)對比). The feedback it uses for the time sharing tasks is based on the time that the task spends sleeping, the time that it was waiting for something or idling. Longer sleep times indicate that the task is interactive, it's spent more time waiting, for instance, on user input or in some type of events. Therefore, when longer sleeps are detected, we need to increase the priority (可由 IHH 推出) of the task and we do that by actually subtracting five, in particular from the priority level of the task. In this way, we're essentially boosting the priority, so next time around, this interactive task will execute bit higher priority. Smaller sleep times are indicative of the fact that the task is compute intensive. Therefore, we want to lower its priority and we do that by incrementing it by adding the number five to it up to a maximum, and essentially, the task next time around will execute in a lower priority class.

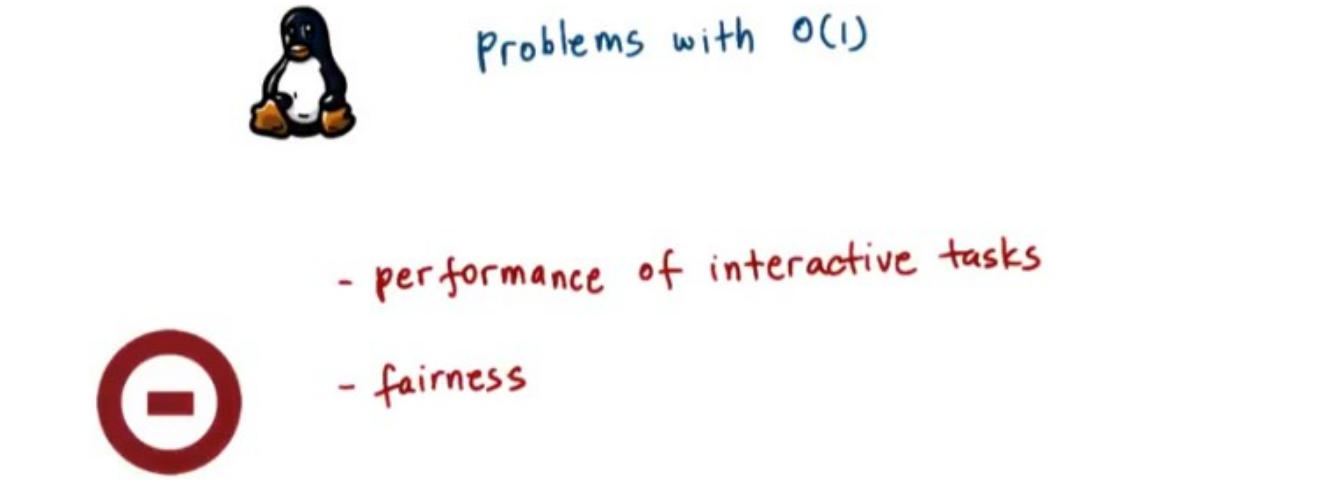


The runqueue in the O(1) scheduler is organized as two arrays of task queues. Each array element points to the first runnable task at the corresponding priority level. These two arrays are called Active and Expired. The active list is the primary one that the scheduler uses to pick the next task to run. It takes constant time to add a task since you simply need to index into this array based on the priority level of the task and then follow the pointer to the 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 (以下的意思是 scheduler 在一個二进製數序列中, 找出為 1 的二进製數的位置). So, if the sequence of bits corresponds to the priority levels and a bit value of one 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 in to this array and select the first task from the runqueue that's associated with that level. If tasks yield the CPU to wait on an event or are preempted due to higher priority task becoming runnable. The time they spent on the CPU is subtracted from the total amount of time, and if it is less than the timeslice, they're still placed on the corresponding queue in the active list. Only after a task consumes its entire timeslice will it be removed from the active list and placed on the appropriate queue in the expired array. The expired array that contains the tasks that are not currently active in the sense that the scheduler will not select tasks from the expired array as long as there are still tasks on any of the queues in the active array. When there are no more tasks left in the active array, at that point, the pointers of these two list 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 low timeslices, and the high priority tasks are given high timeslices (以下與其說 是在講 我們要將其弄成 IHH 的原因, 還不如說 是在講 'IHH 的性質' 跟 'active array - expired array 這個結構的性質' 是 consistent 的. 以下意思是說, high timeslice 在 active array - expired array 這個結構中 先天 就俱備 high priority 的性質, 因為 timeslice 不用完, 就不會出 active array, low timeslice 的就得等它). As long as the high priority tasks have any time left in their timeslice, they will keep getting scheduled, they will remain in the 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

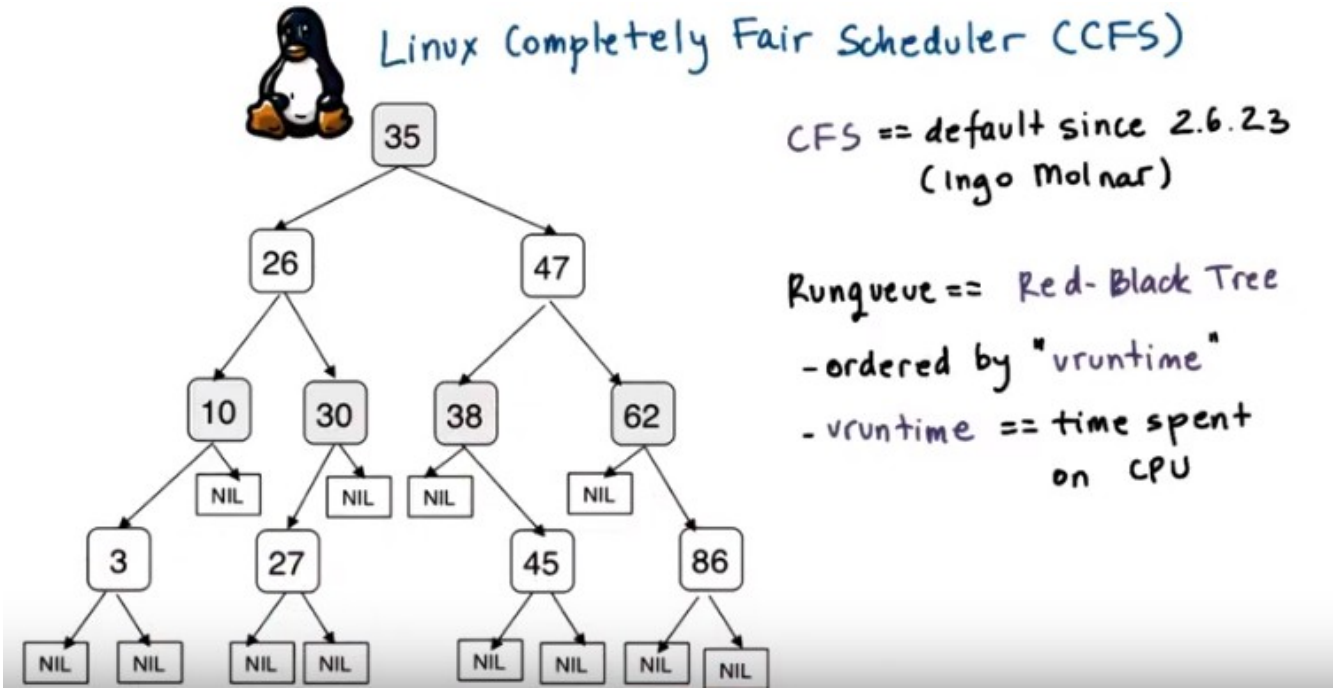
to have a low timeslice value so that, yes they will get a chance to run, however they won't disrupt the higher priority tasks, they won't delay them by too much. Also note that the fact that we have these two arrays also serves like an aging mechanism so these high priority tasks will ultimately consume their timeslice 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) scheduler 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) scheduler really affected the performance of interactive tasks significantly (22 段和 24 段有對這點更詳細點的解釋, 總結起來說, 就是因為 IHH 的原因, iteractive task 最先被执行, timeslice 用完了後, 就被放入了 expired array 中去了, 然後就要等"all of the low priority tasks consumed their entire time quantum (from 24 段)") (interactive 應該就是 I/O interactive 中的那個 interactive 的意思, 前面多次用過 interactive 這詞). And as the work loads changed as typical applications in the Linux environment were becoming more time sensitive, think Skype, movie streaming, gaming. The jitter (神經過敏,戰戰兢兢) 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 (即 CFS), and the CFS scheduler became the default scheduler starting in the Linus 2.6.23 kernel. Ironically, both of these scheduler's are developed by the same person. You should note that both the O(1) and the CFS scheduler are part of the standard Linux distribution. This one (指 CFS) is the default, however, if you wish, you can switch and choose the Linux O(1) scheduler to execute your tasks.

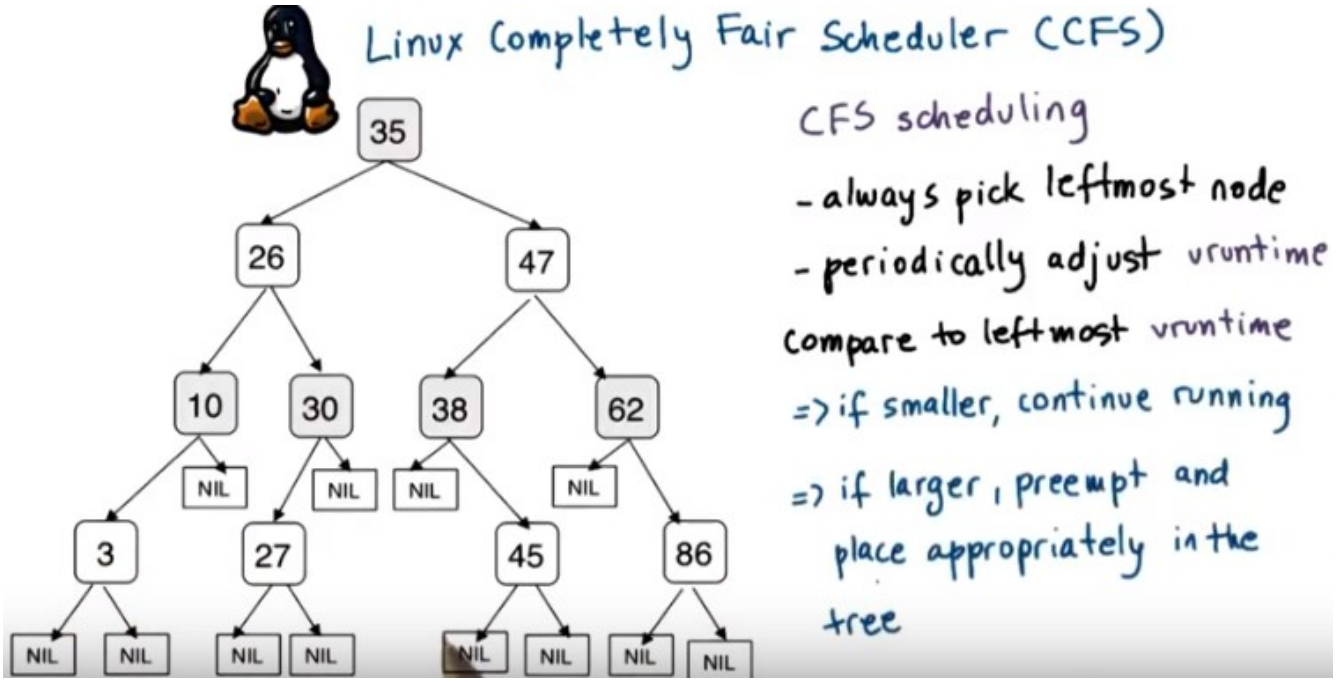


22. As we said, one problem with the O(1) scheduler in Linux is that once tasks are placed on the expired list, they wouldn't be scheduled until all remaining tasks from the active list have a chance to execute for whatever their timeslice amount of time is. As a result, the performance of interactive tasks is affected. There is 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.

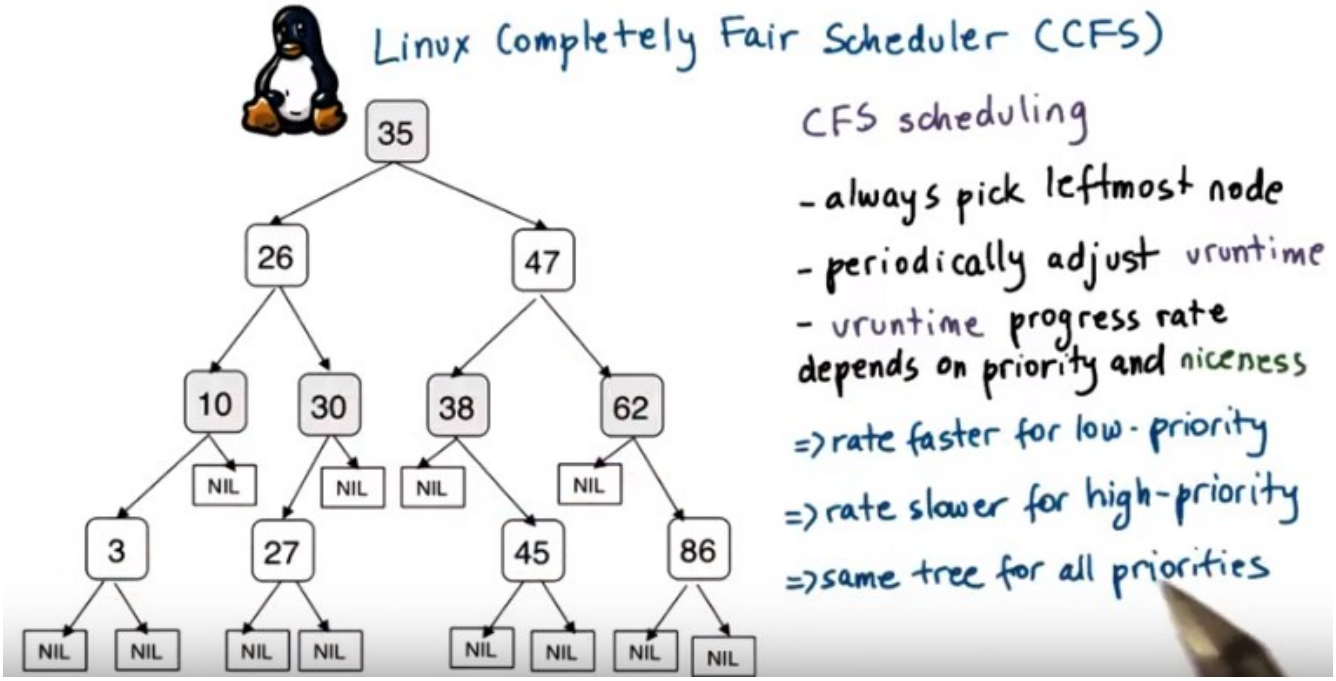


As we said, Ingo Molnar proposed the completely fair scheduler, CFS, to address the problems with the O of 1 scheduler. And CFS is the default scheduler in Linux since the 2.6.23 kernel. It's the default

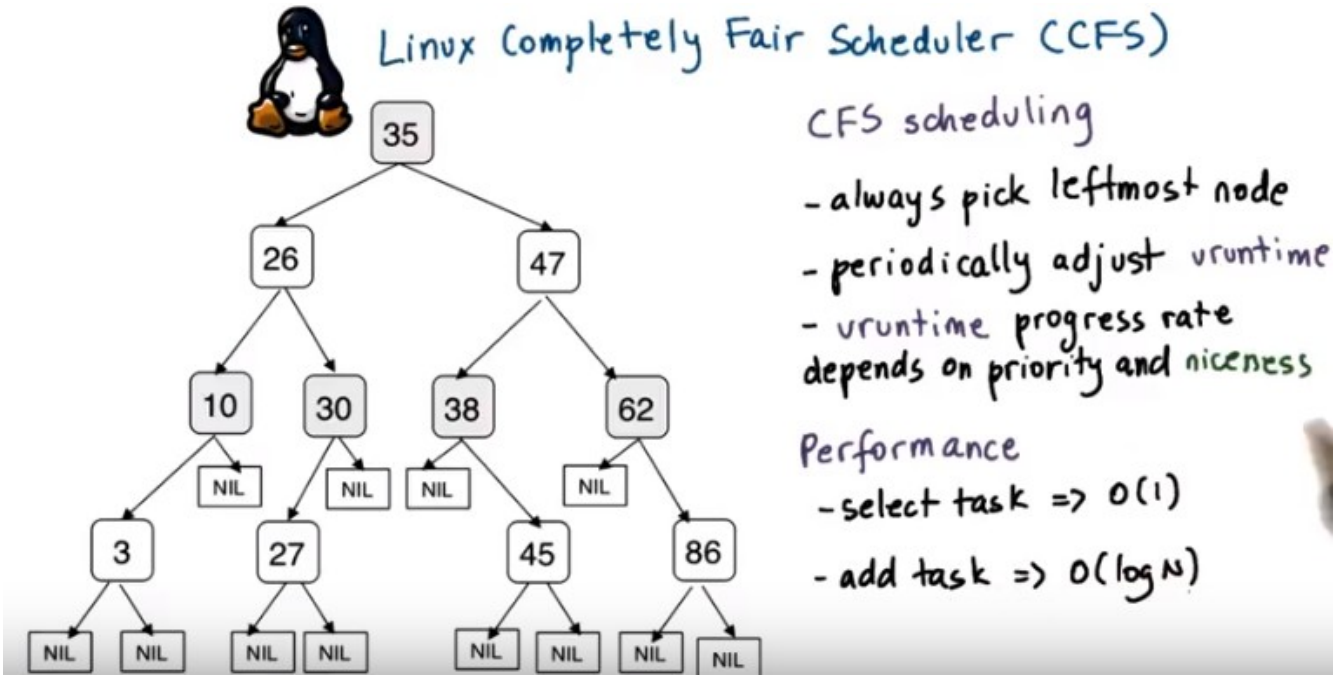
scheduler for all of the non-real time tasks. The real time tasks are scheduled by a real time scheduler. The main idea behind the Completely Fair Scheduler is that it uses a so-called a Red-Black Tree as a Runqueue structure. Red-black trees belong to this family of dynamic tree structures that have 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 of 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 that 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 left most 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 virtual 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 (the running task) will be preempted, and it will be placed in the appropriate location in the tree. Obviously, the task that's corresponding to the leftmost node in the tree will be the one that will be selected to run next.

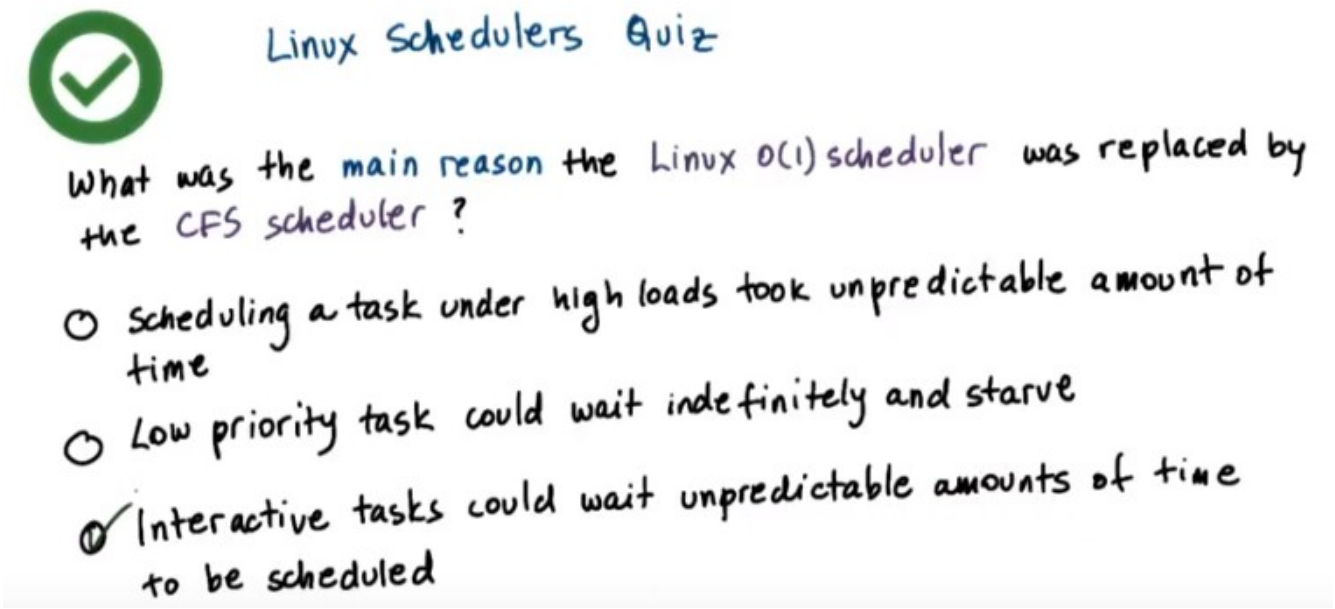


To account for differences in the task priorities or in their niceness value. CFS changes the effective rate at which the task's virtual time progresses. For lower priority tasks, time passes more quickly. Their virtual run time value progresses faster. And therefore, they will likely lose their CPU more quickly, because their virtual run time will increase, compared to other tasks in the system. On the contrary, for high priority tasks, time passes more slowly. Their virtual runtime value 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 run queue structure for all of the priority levels, unlike what we saw with some of the other scheduler examples.

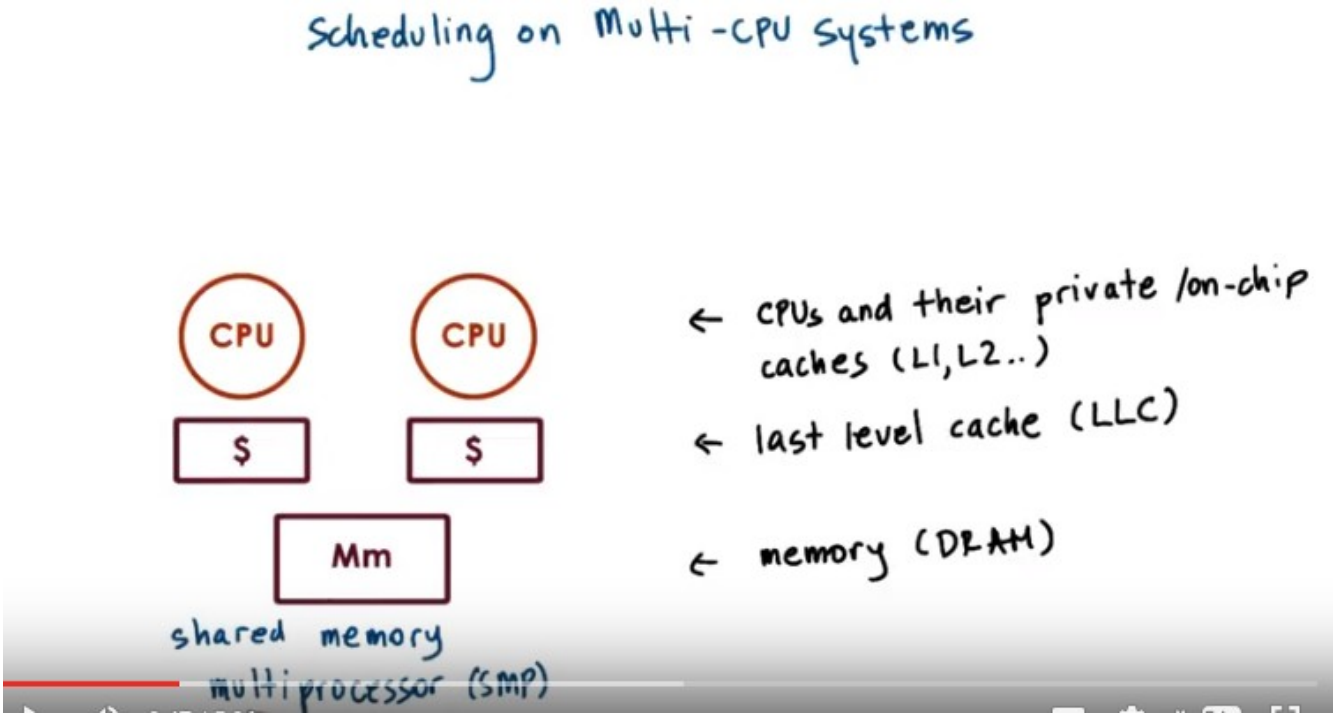


In summary, selecting a task from this run queue to execute takes O of 1 time. Takes constant amount of time since it's typically just a matter of selecting the leftmost node in the tree. At the same time, adding a task to the run queue takes logarithmic time relative to the total number of tasks in the system. Given the typical levels of load in current system, this log of n time is acceptable. However, as the computer 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. As a review, I would like to ask a question about the two Linux schedulers that we just discussed. What was the main reason the Linux O of 1 scheduler was replaced by the CFS scheduler? Was it because scheduling task under high loads took unpredictable amount of time? Low priority tasks could wait indefinitely and starve? Or because interactive tasks could wait unpredictable amounts of time to be scheduled to run? Select the appropriate answer.

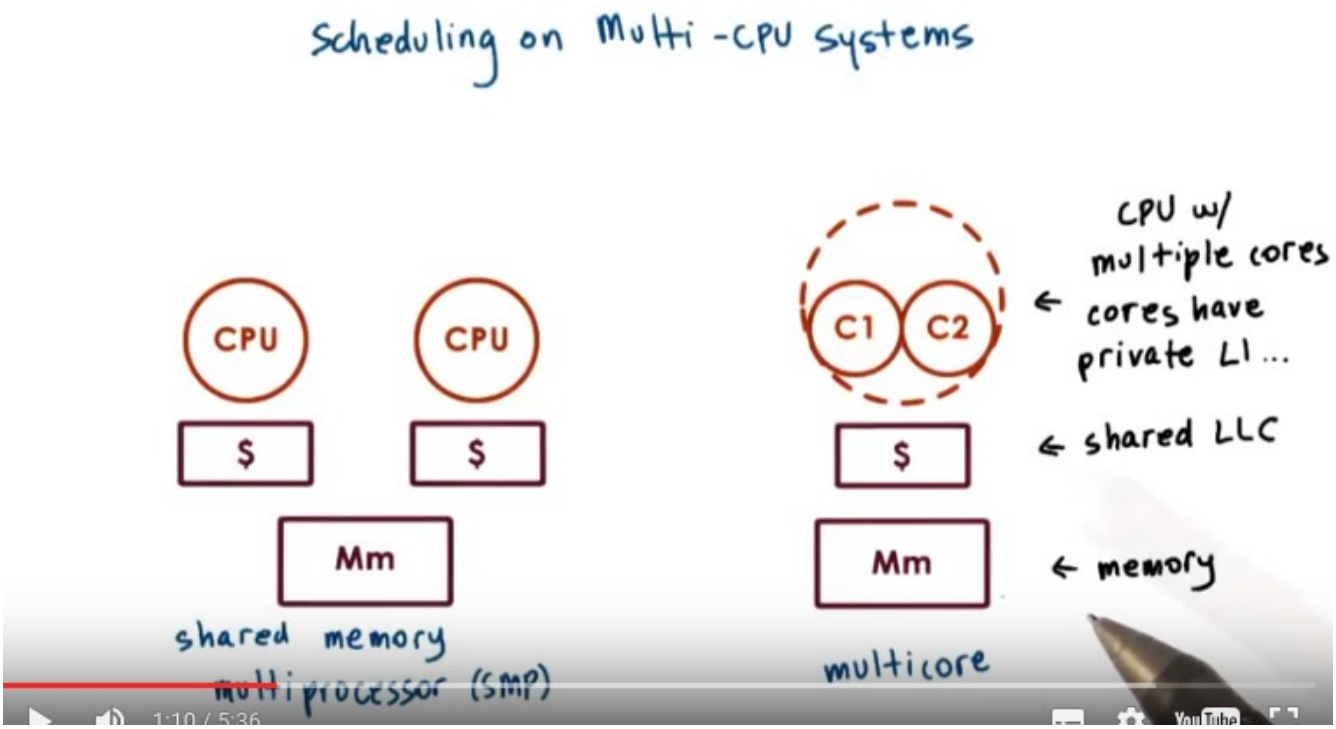


24. Let's take a look at each of the choices that are given. The first statement is not correct. The Linux O of 1 scheduler was O of 1 because it took constant amount of time to select and schedule a task regardless of the load. The second statement is sort of correct in the sense that as long as there were continuously arriving higher priority tasks, it was possible for the low priority tasks to keep waiting an unpredictable amount of time and possibly indefinitely and therefore, starve. But this was really not the main reason why the scheduler was replaced. The final choice was the main reason. Recall that we said that the common work, workloads were becoming much more and more interactive and were demanding high predictability. In the O(1) scheduler with the active and expired list, once the 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 time Linus Torvalds resisted integrating a scheduler that would address the needs of the small interactive tasks in the Linux kernel. His rational 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 offline scheduler. However, as the general purpose work loads began to change, then a general purpose operating system like Linux, have to really incorporate a scheduler that would address the needs of those general purpose workloads and CFS was really meeting those needs.

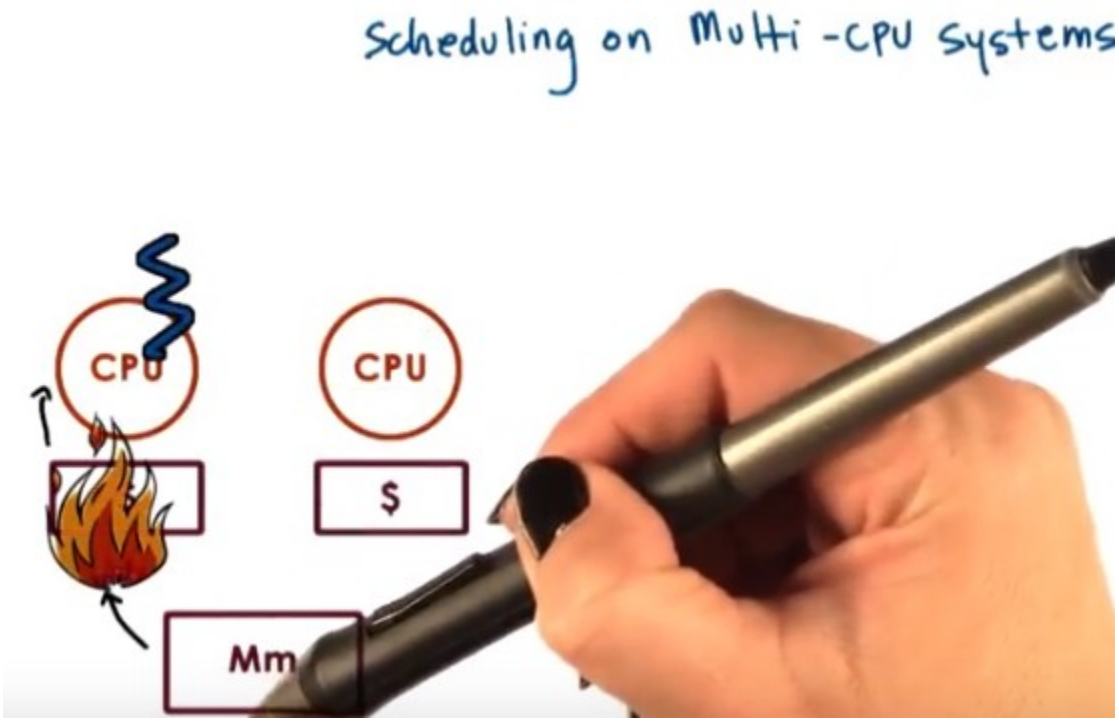


SMP: shared memory multiprocessor (後面的視頻中也會出現多次)

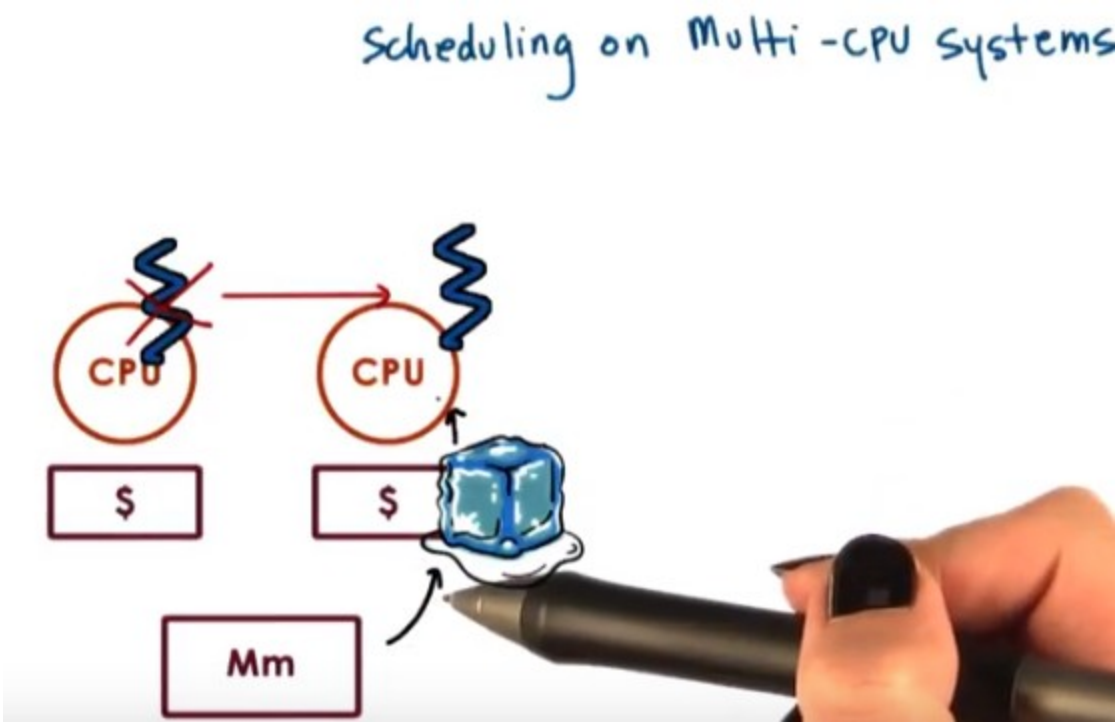
25. Let's now look at scheduling on multi-CPU systems. However, before we start talking about scheduling, let's look a little bit at some architecture detail. First we will look at shared memory multiprocessors. And then we will also take a look at how this compares to multi-core architectures. In a shared memory multiprocessors, or SMPs, there are multiple CPUs. Each of them have their maybe own private caches, like L1 and L2. There are last level caches that may or may not be shared among the CPUs. And there is a system memory, DRAM, that is shared across all of the CPUs. Here we show just one memory component, but it is possible that there would be multiple memory components. But the point is that all of the memory in the system is shared among all of the CPUs.



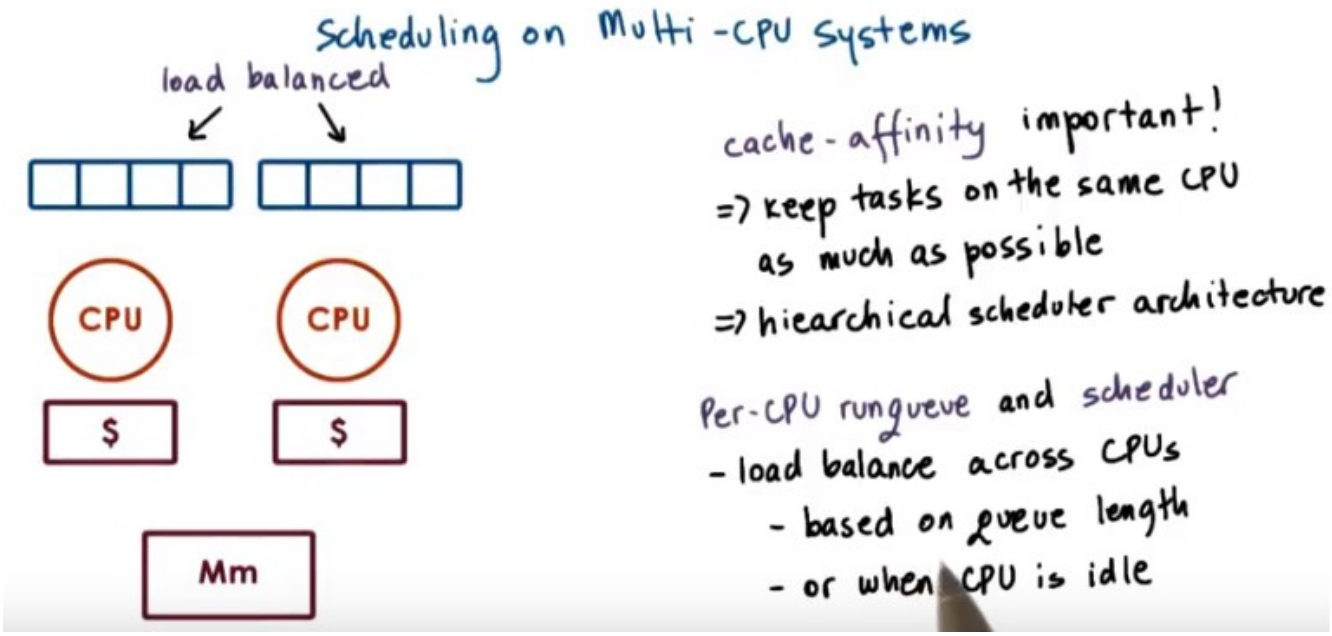
In the current multicore world, each of these CPUs can have multiple internal cores, so multiple CPUs internally. Each of those cores will have private caches, and then overall the entire multicore CPU will have some shared last level cache. And then again, there will be some shared system memory. Here in this picture, we have a CPU with two 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 CPUs. Whereas on the server and platforms, it's more common to have CPUs that have six or eight cores and to also have multiple such CPUs, so we'll have multiple multicore CPUs. As far as the operating system is concerned, it sees all of these CPUs as well as the cores in the CPU as entities onto which it can schedule all execution context, so threads. All of these are, as far as the operating system is concerned, possible CPUs for it can schedule some of its workload. So to make our discussion more concrete, we will first start talking about scheduling on multi-CPU systems in the context of SMP systems, and a lot of these things will apply to the multicore world because again, the scheduler just sees the cores as CPUs. And we'll make some comments that are more exclusively applied to multicores towards the end of this lesson.



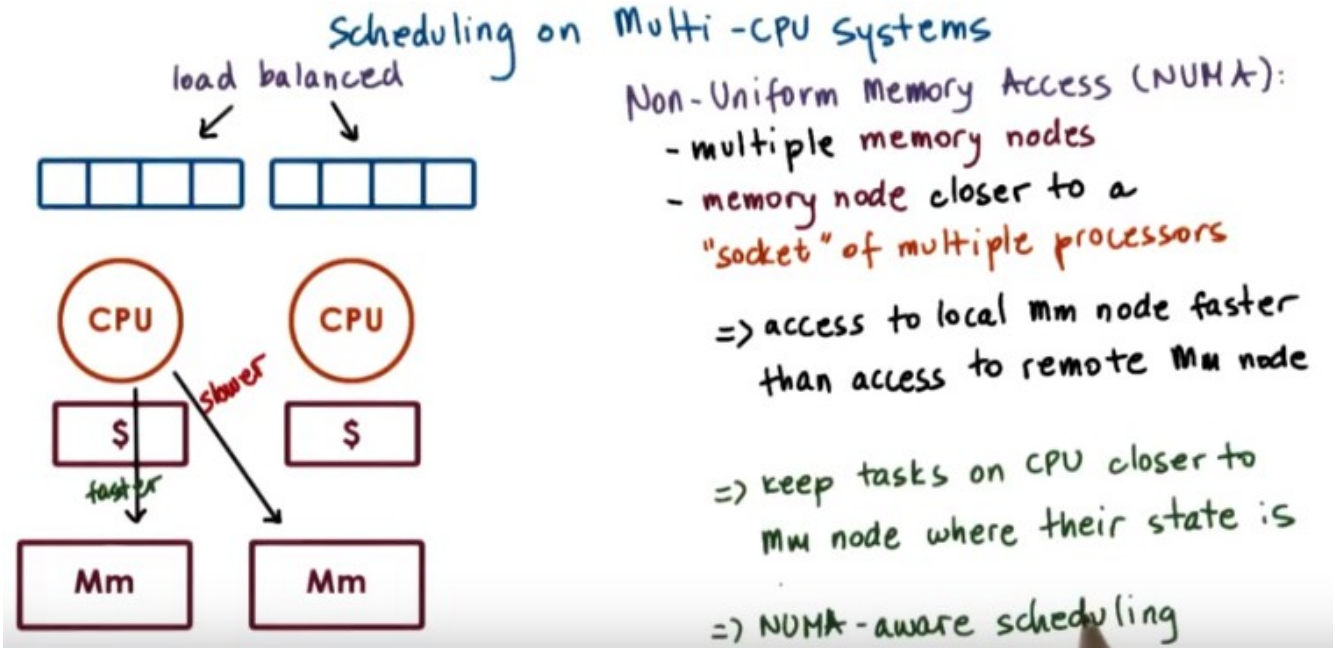
We said in our earlier lectures that the performance of threads and processes is highly dependent on whether the state that the thread needs is in the cache or in memory. Let's say a thread was executing on CPU 1 first. Over time this thread was slightly able to bring a lot of the state that it needs both into the last level of cache that's associated with this CPU, as well as in the private caches that are available on the CPU. And 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 its state will be there so the thread will operate with a cold cache. We'll have to bring all of the state back in and that will affect performance.

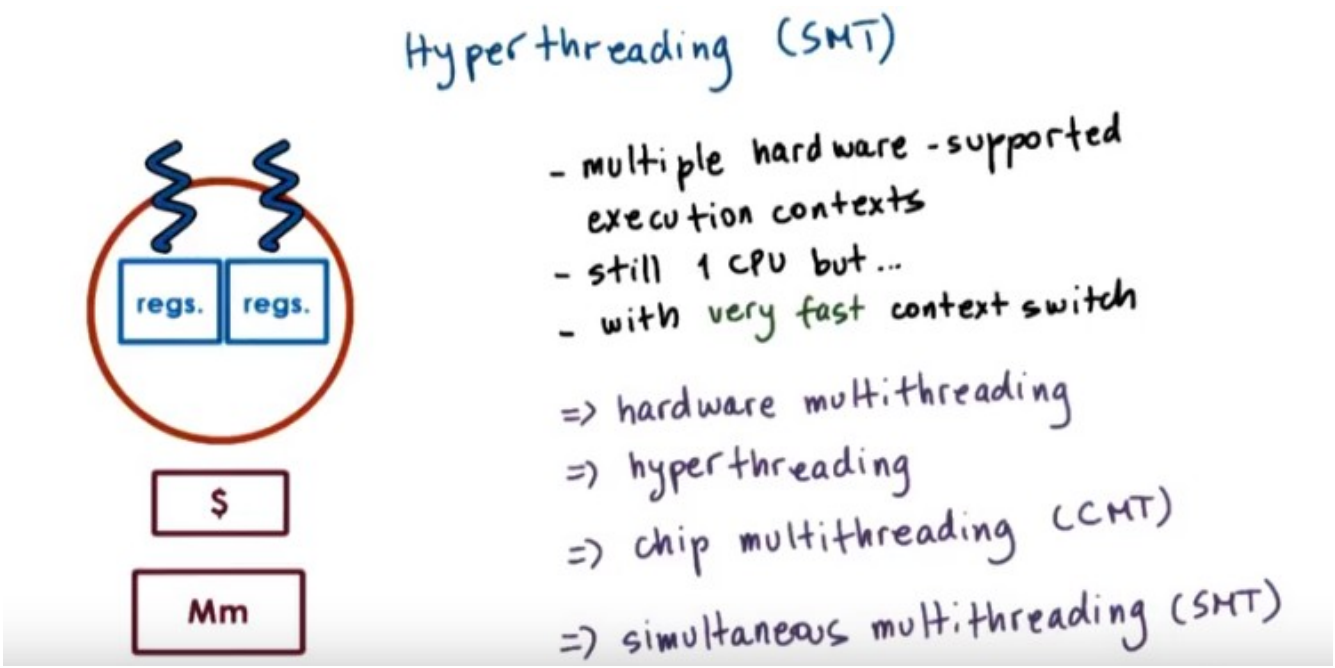


Therefore, what we want to achieve with a scheduling on multi-CPU systems is to try to schedule the thread back on the same CPU where it executed before because it is more likely that its cache will be hot. We call this cache affinity and that is clearly important. To achieve this, we basically want 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 tasks among CPUs. And then a per-CPU scheduler with a per CPU runqueue repeatedly schedules those tasks on a given CPU as much as possible. To balance the load across the different CPUs and their per-CPU runqueue, 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 tasks across them. Or potentially when a CPU is idle, it can at that point start looking at the other CPUs and try to get some more work from them.



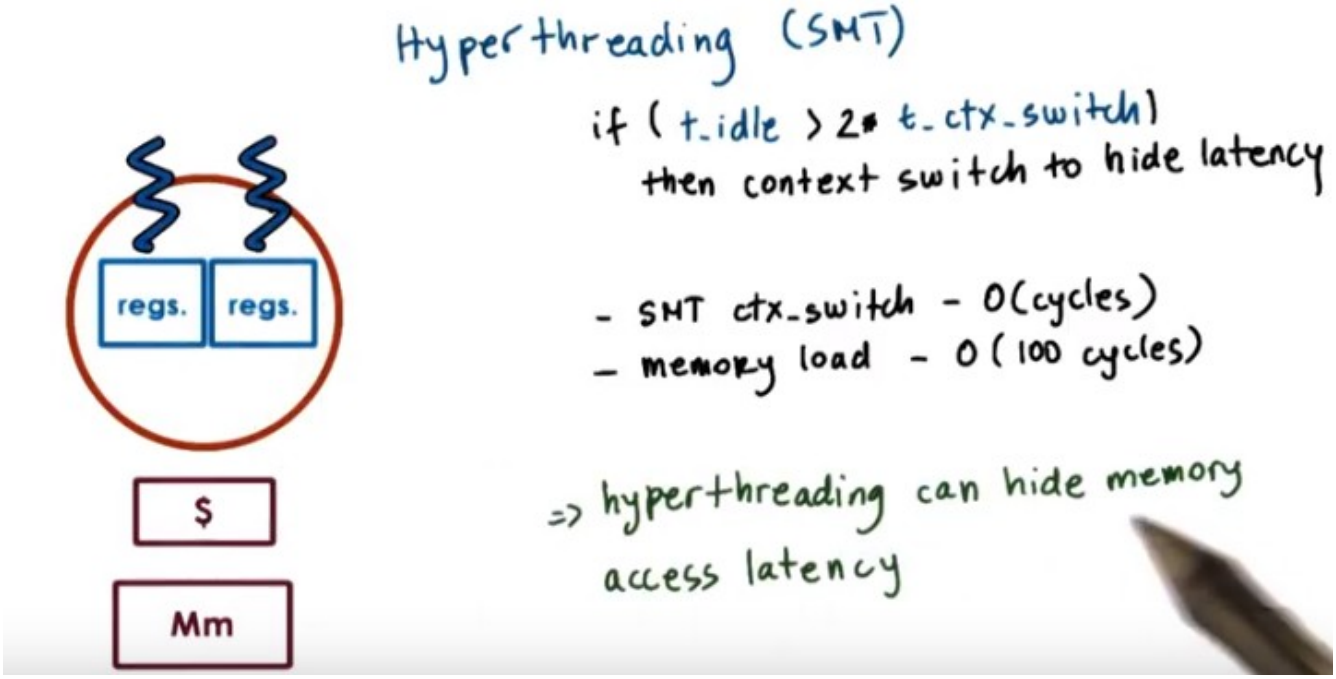
上圖左下的兩個詞為 faster 和 slower

In addition to having multiple processors, it is possible to also have multiple memory nodes. The CPUs and the memory nodes will be interconnected via some type of interconnect. For instance, on modern Intel platforms, there is a interconnect that's called QuickPath Interconnect, or QPI. One way in which these memory nodes can be configured is that a memory node can be technically connected to some subset of the CPU, so for instance, to a socket (此 socket 不是 network 裡面的 socket) that has multiple processors. If that is the case, then the access from that set of CPUs to the memory node will be faster versus from that particular processor to a memory node that's associated with another set of CPUs. Both types of accesses will be made possible because of the interconnect that's connecting all of these components. However, they will take 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 that tasks are bound to those CPUs that are closer to the memory node where the state of those tasks is. We call this type of scheduling NUMA-aware scheduling.

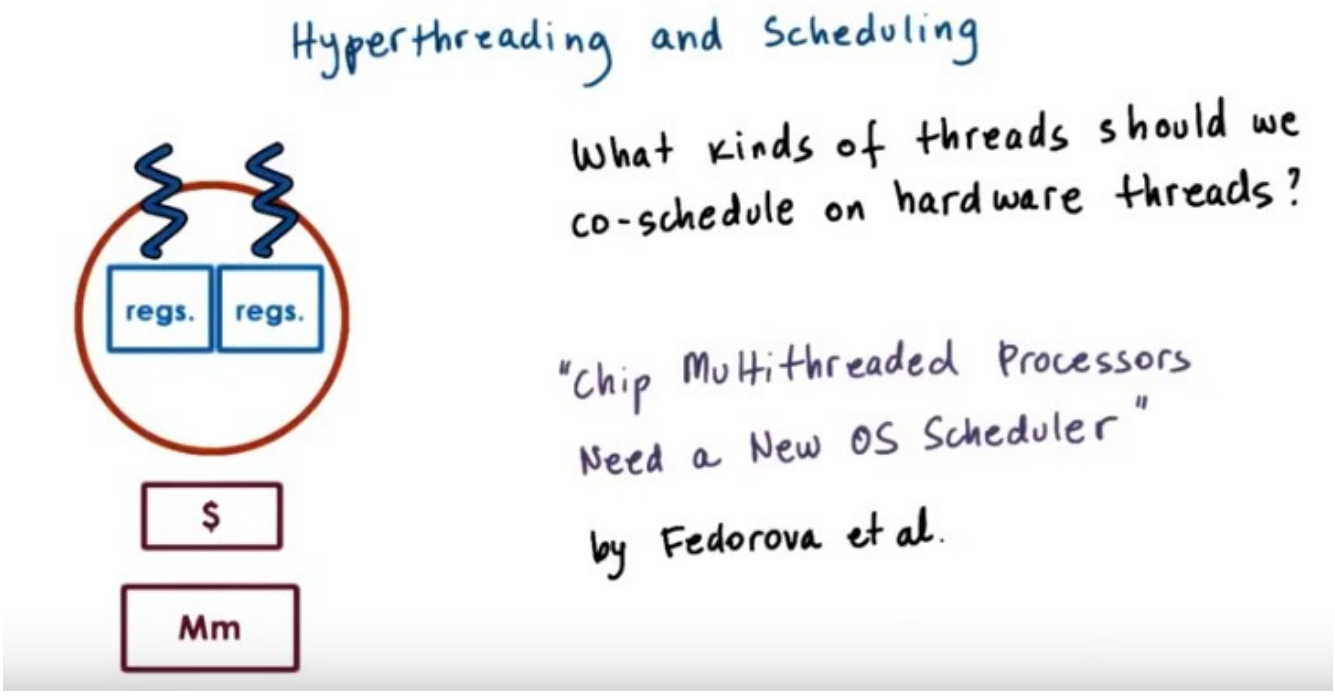


上圖中的幾個=>右邊的都是 hyperthreading 的幾個不同的名字

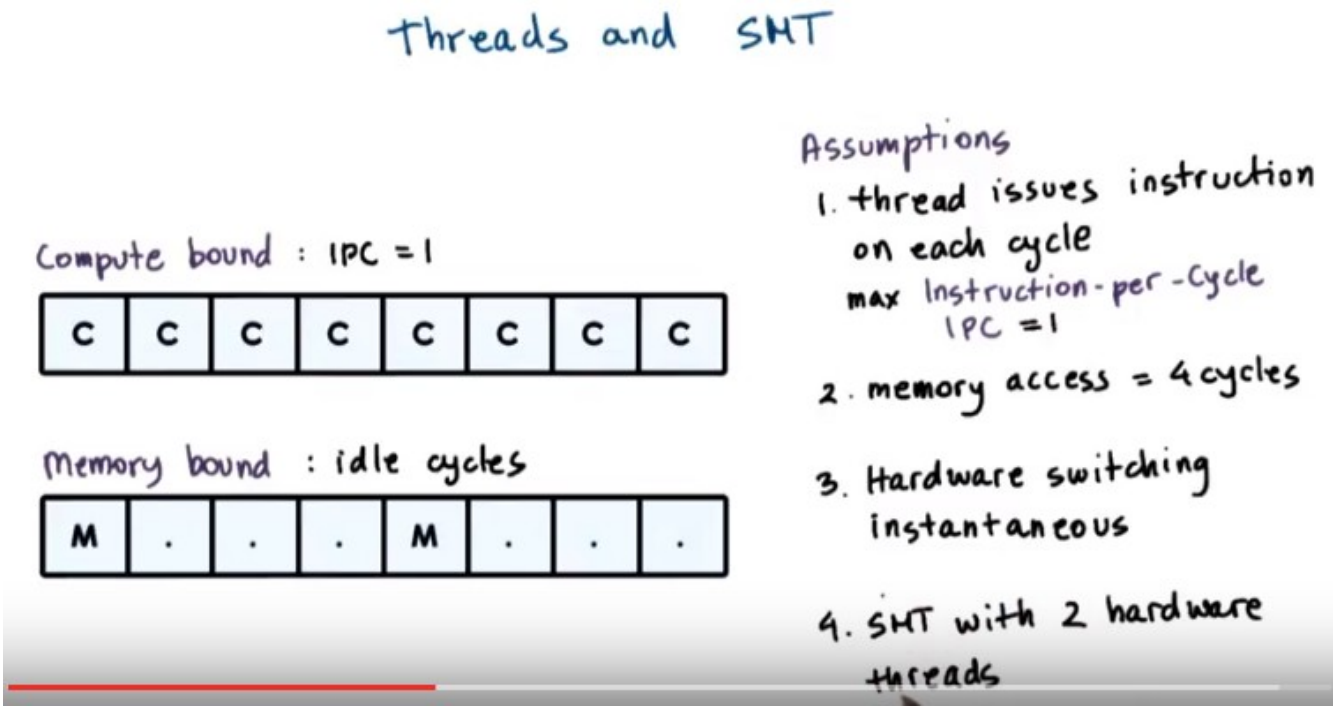
26. 以下就是講單 CPU 如何多線程. The reason why we have to context switch among threads is because the CPU has one set of registers to describe the active execution context, the thread that's currently running on the CPU. These include the stack pointer and program counter, in particular. Over time, however, hardware architects have recognized that they can do certain things to help hide some of the overheads associated with contact switching. One way this has been achieved, is to have CPUs that have multiple set of registers, that each set of register can describe the context of a separate thread, of a separate execution entity. One term that's used to refer to this is hyper threading. So, hyper threading refers to multiple hardware-supported execution contexts, so hyper threads. There's still just one CPU, so on this CPU only one of these threads can execute at a particular moment of time. However, the context switching between these threads is very fast. And just basically the CPU needs to switch from using this set of registers to another set of registers. Nothing has to be saved or restored. This mechanism is really referred to by multiple names. So in addition to hyperthreading a common term is also to refer to this is hardware multithreading, or chip multithreading or simultaneous multithreading, SMTs. So, we will used basically these two terms, hyperthreading and SMTs (即將 SMTs 作為 hyperthreading 的簡稱, SMTs 這個名字只能靠死記: smart), more dominantly than the others. Hardware today frequently supports two hardware threads. However, there are multiple higher end server designs to support up to eight 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's some trade-offs associated with this as always. If it is enabled, as far as the operating system is concerned, each of these hardware contexts appears to the operating system's scheduler as a separate context, a separate virtual CPU, onto which, which it can schedule threads given that it can load the registers with the thread context concurrently. So for instance in this figure the scheduler will think that it has two CPUs 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 it, the time that a thread is idling, that a thread has to wait on something to happen is 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 a context switch among the two hardware threads is in the order of cycles. And the time to perform a memory access operation, a memory log, remains in the order of hundreds of cycles, so it's much greater. So given this, then it means that it does make sense to context switch to the other hardware thread. And in that way, this technology hyperthreading will help us even hide the memory access latency that threads are experiencing.

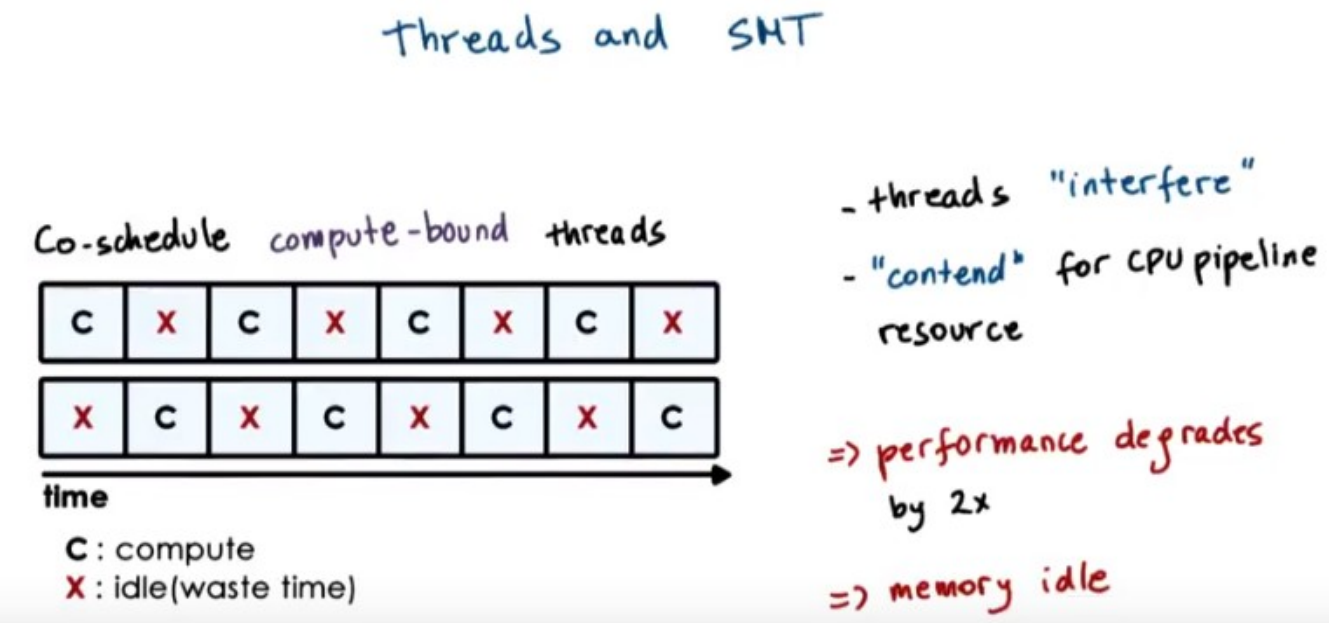


Hyperthreading does have implication and scheduling, in that it raises some other requirements when we're trying to decide what kinds of threads should we co-schedule on the hardware threads in the CPU. We will discuss this question in the context of the paper Chip Multithreaded Processors Need a New Operating System Scheduler by Sasha Fedorova and others.

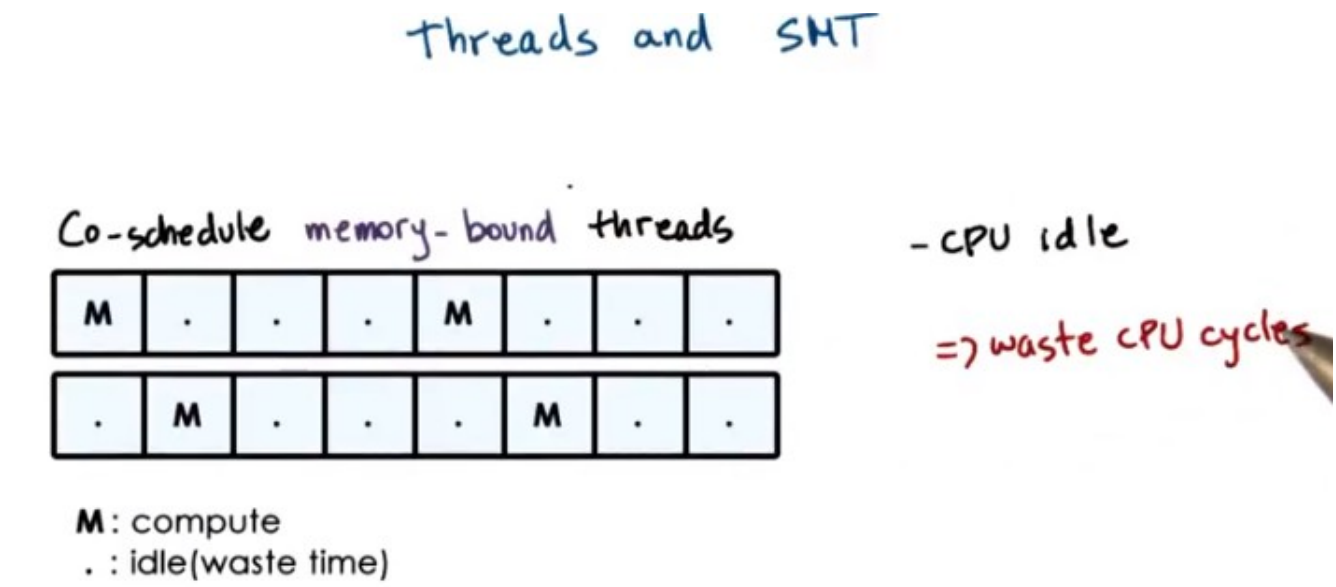


上圖中 3. Hardware switching instantaneous 的意思見下面藍字.

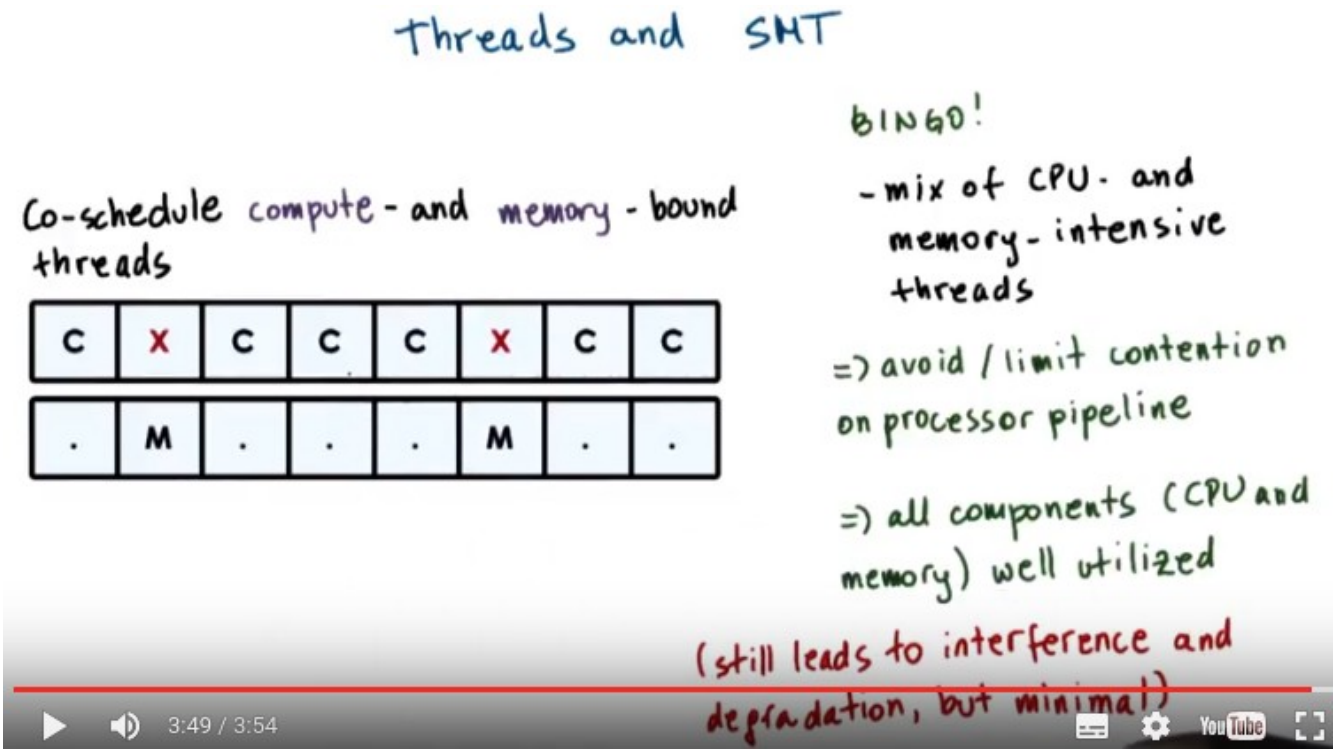
27. To understand what's required from a scheduler in a simultaneous multi threading system, let's make some assumptions first. Since we will base our discussions on Federova's paper, we will use the same assumptions that she has made, and the figures that we will use to illustrate those assumptions will be reproductions from her paper and her presentation. The first assumption that 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 would be instructions per cycle equal one. Given that we have just one CPU, we cannot have a IPC that is greater than one. The second assumption that we will make is that a memory access takes four 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 also assume that the time it takes to contact 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 two hardware threads.



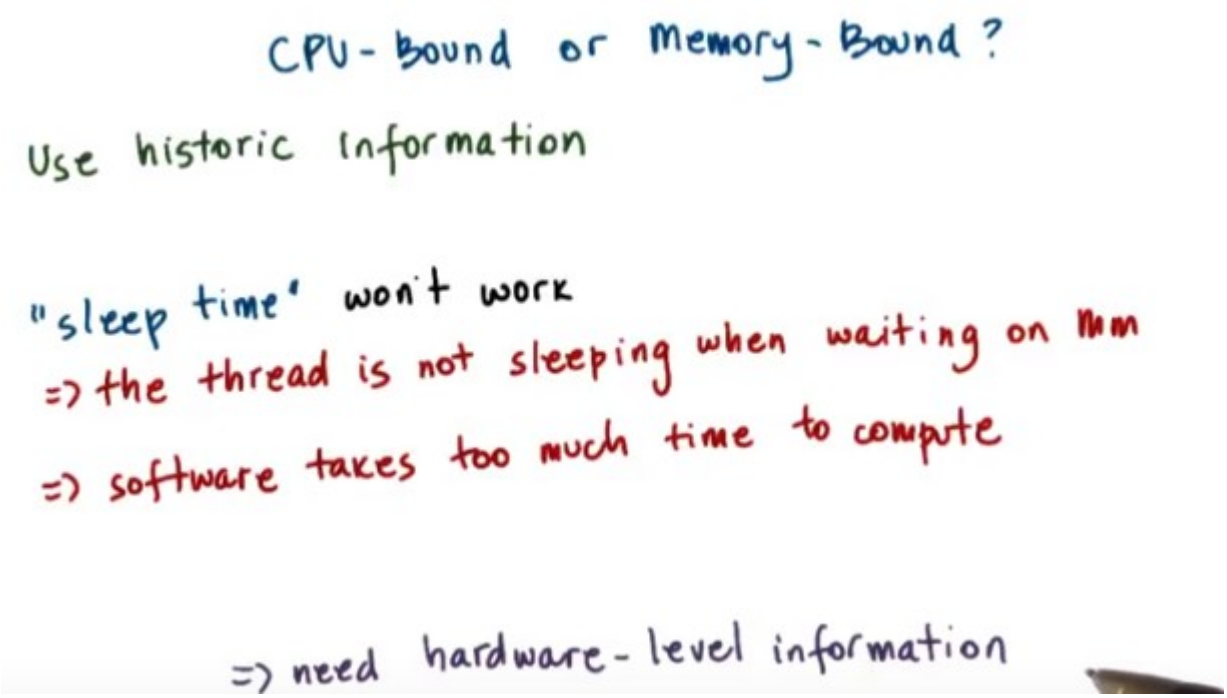
let's take a look first at what would happen if we chose to co-schedule on the two hardware contacts. Two threads that are both compute-bound, so compute intensive, or CPU bound. What that means is both of the threads are ready to issue a CPU instruction on every single cycle. However given that there is only one CPU pipeline, so one CPU fetch decode issue ALU logic, only one of them can execute at any given point of time. As a result these two threads will interfere with each other. They will be contending for the CPU pipeline resources. And best case, every one of them will basically spend one cycle idling while the other thread issues its instruction. 這兩個 threads 不一定要像圖中那樣嚴格地平分交替, 也可以第一個 thread 連續佔用 CPU 兩個 cycle, 第二個 thread 再佔用 CPU 一個 cycle. 重點是這兩個 threads 不能同時佔用 CPU, 一個 thread 佔用 CPU 時, 另一個 thread 就不能運行(而這個 thread 也沒有甚麼 I/O 甚麼的需要等的). As a result, for each of the threads, its 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, they're idle. There's nothing that's scheduled that performs any kinds of memory accesses. Well that's not good either.



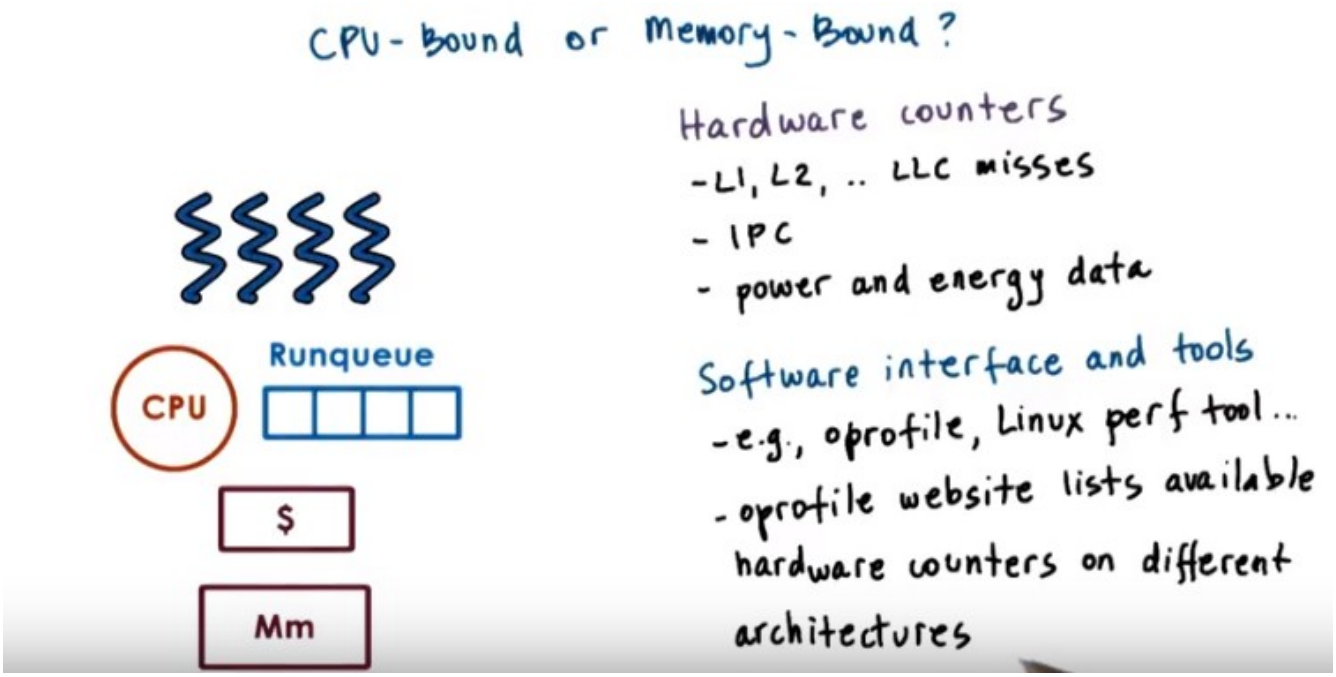
Well another option is to co-schedule two memory-bound threads. In this case we see however, that we end up with some idle cycles because both of the threads end up issuing co-memory operation. And then they need to wait four cycles until it returns. Therefore, we have two of the cycles that are unused. So, the strategy then to co-schedule memory bound threads leads to wasted CPU cycles.



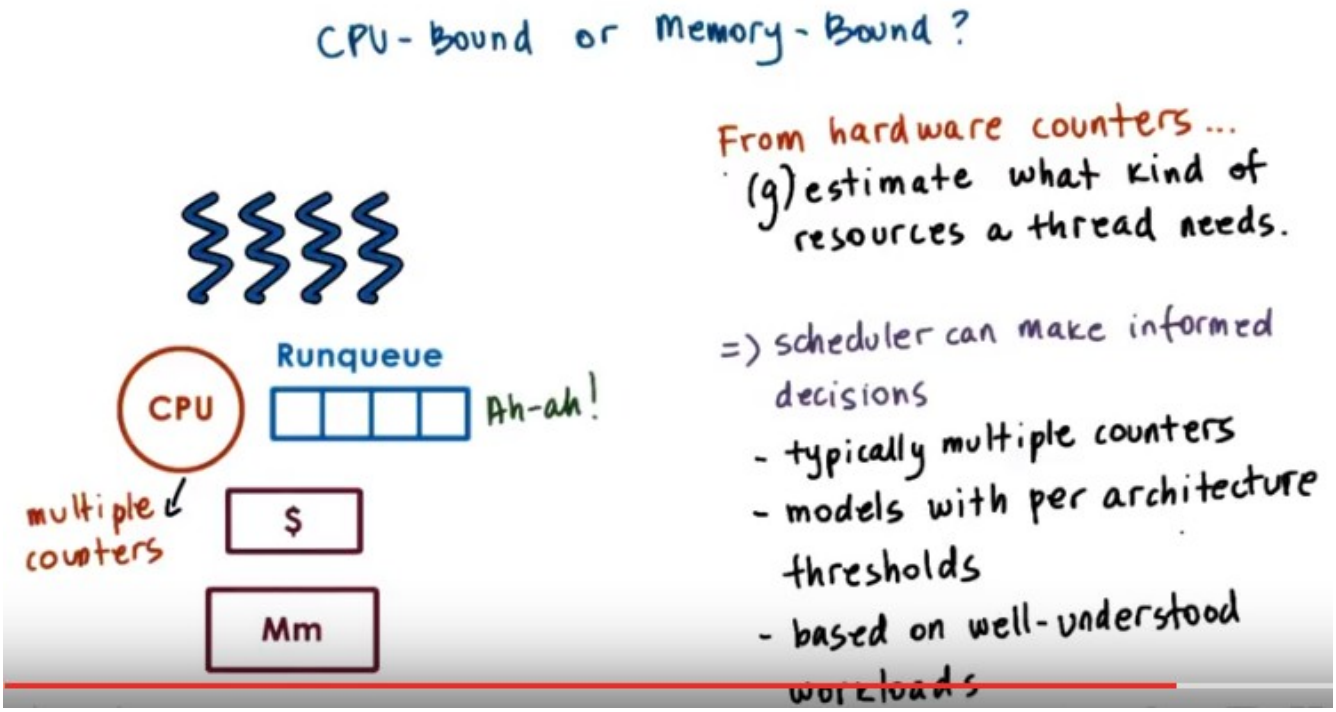
So then our final option is to consider mixing some CPU and memory intensive threads, and then if we see, we end up with the desired schedule. It's a bingo. We end up fully utilizing each processor cycle, and then, whenever there is a thread that needs to perform a memory reference, we context switch to that thread in hardware. The thread issues the memory reference, and then we context switch back to the CP-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 on the processor pipeline. And then also, all of the components, both the CPU and the memory will be well utilized. Note that this still will lead to some level of degradation due to the interference between these two threads. For instance, the compute bound thread can only execute three out of every four cycles, compared to when it ran alone. However, this level of the degradation will be minimal given the properties of the particular system



28. While the previous example gave us a good idea what type of scheduler 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 versus a memory bound. To answer this question we will use historic information. We will look at the thread's past behavior. And this is similar to what we said was useful when we were trying to determine whether a thread is interactive or I/O bound versus CPU bound. However, previously we used sleep time for this type of differentiation of I/O versus 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 in the processor pipeline and not on 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 a 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.



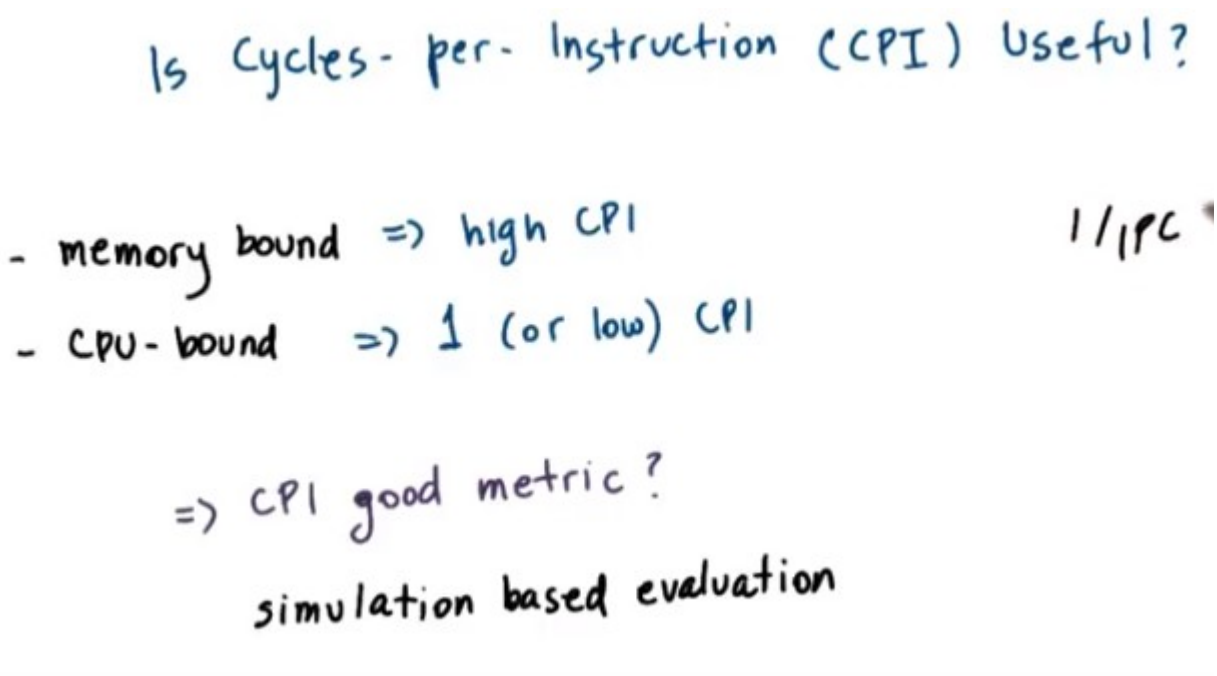
Fortunately, modern hardware has lots of so called hardware counters that get updated as the processor is executing and keep information about various aspects of the execution. These include information about the cash usage for instance such as the L1, L2, or the last level cash misses or information about the number of instructions that were retired. So that we can compute the IPC or in your platforms there's also information regarding 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, oprofiler 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 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 decision?



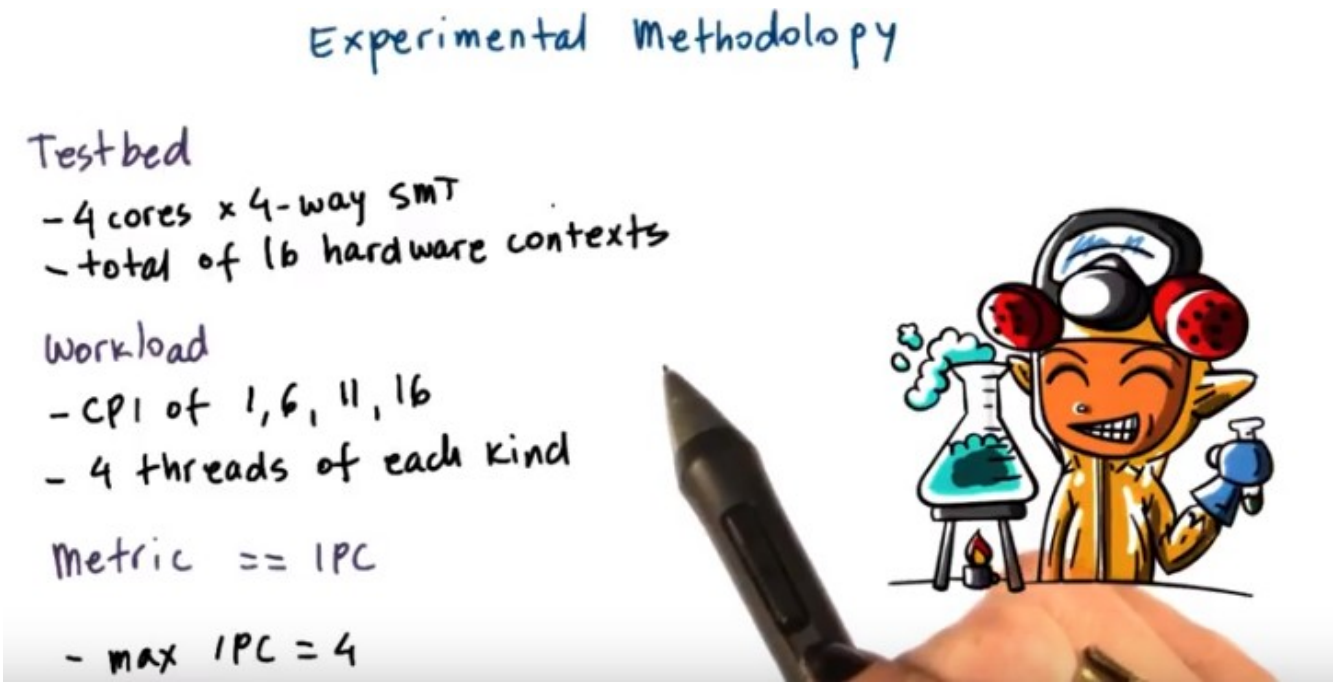
(g)estimate 就是 estimate 或 guesstimate

Many practical as well as research 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 scheduler can look at a counter like the last level cash misses and using this counter, a scheduler can decide that a thread is memory bound, so its footprint doesn't fit in the cash. Or the same counter can also tell the scheduler that something changed in the execution of the threat so that now it's executing with some different data in a different phase of its execution, and it's running with the cold cash. What this tells us is that, 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 thread's 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 a specific hardware platform and that have been trained using some well understood workloads. So we ran a workload that we know is memory intensive and we made some observations regarding 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 much more advanced research problems, which are a little bit out of the scope of this particular introductory course. However, I really wanted to make sure that you're 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 not just regarding CPU scheduling.



29. As a more concrete example, Fedorava speculates that a useful counter to use to detect the thread CPU-ness versus memory-ness is cycles-per-instruction. She observes that memory bound threads take a lot of cycles to complete an instruction therefore it has a high CPI. Where is the CPU-bound thread, it will complete an instruction every cycle or near that and therefore, it will have a CPI of 1, or a low CPI. So this speculates then, it would be useful to gather this kind of information, this counter about the cycles per instruction. And use that as a metric in scheduling threats on hyper-threaded platforms. Given that there isn't exactly a CPI counter on the processors that Fedorova uses in her work, and computing something like 1/IPC would require software computations so that wouldn't be acceptable. For that reason Fedorapa uses a simulator, that supposedly the CPU does have a CPI counter. And then she looks at a better 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.



To explore this question she simulates a system that has four cores where every one of the cores is four way multi threaded. So, there's a total of 16 hardware contexts in her experimental test bed. Now, she wants to bury 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 one, six, 11 and 16. Clearly the thread with the CPI of one will be the most CPU intensive, and then the threatd with a CPI of 16 will be the most, memoring threads. And the overall work load mix has four 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 in tact of such potentially different scheduling positions, she uses a metric, the instructions per cycle (IPC, 注意不是 CPI). Given that the system has four cores in total, the maximum IPC that could be achieved is going to be four. So, four instructions per cycle will be the best case scenario for where every single one of the cores complete one instruction in each cycle.

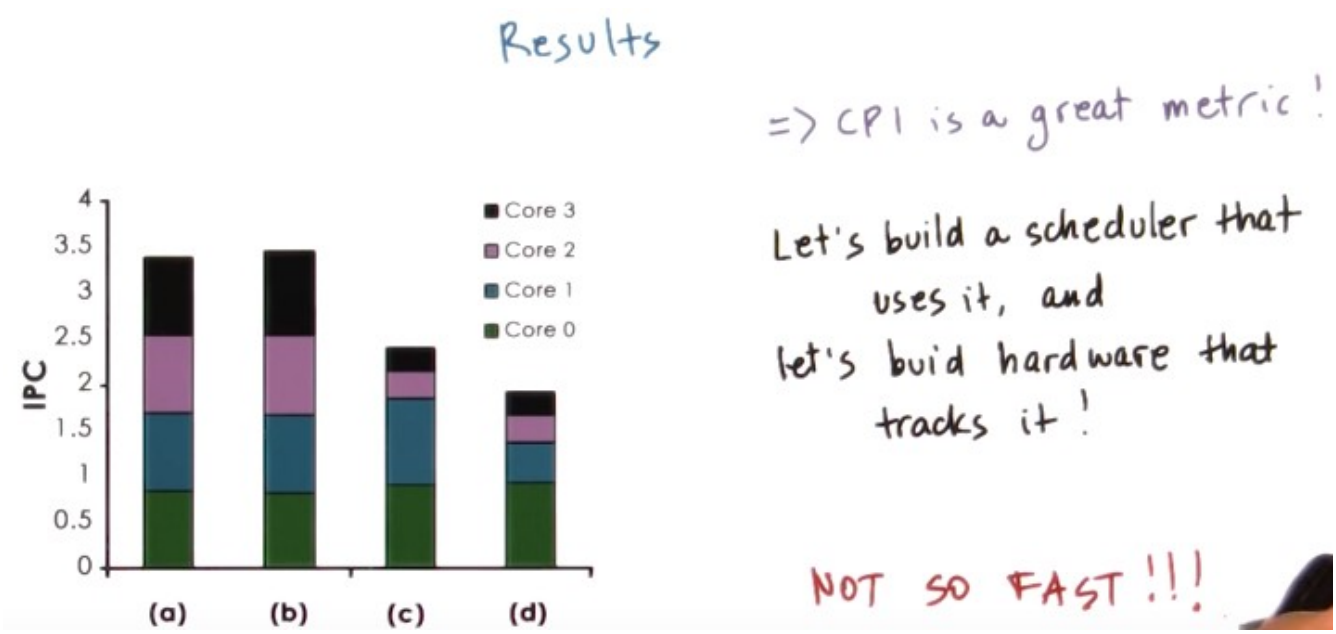
Actual Experiments

	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

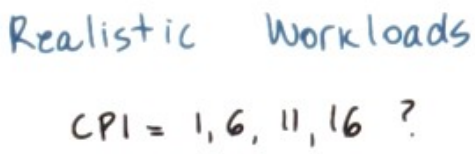
And then she conducts several experiments as shown in this figure. In every one of the experiments she manually and statically changes how the workload is distributed across the course. So in the first experiment on core one, the four hardware threads will be assigned threads that have, software threads that have a CPI of one, six, 11, and 16. The four hardware threads on Core 2 will be assigned software threads and tasks that have CPI of one, six, 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 then 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, they have a CPI of 1. Where as on Core 3, all of the tasks are memory intensive, because they have a CPI of 16. And then the second and the third round of the experiments falls somewhere between these two extremes. So what she's trying to do, she's trying to make some static decisions that a scheduler would have made. And in doing that she's trying to understand whether it even makes sense to build a scheduler that will use CPI as a metric.

30. Instead of a typical quiz, I would like for you to do a self check of your analytical skills. Here is a diagram that's summarizing the performance results that Fedorova gathered from running the experiments that we showed before. What do you think these results tell us about the use of a metric-like cycles per instruction for scheduling? Again, this is not really a quiz. This is more of a self check of your analytical skills. Try answering this question and then see our summary of the results of using CPI as a scheduling metric in the next video.

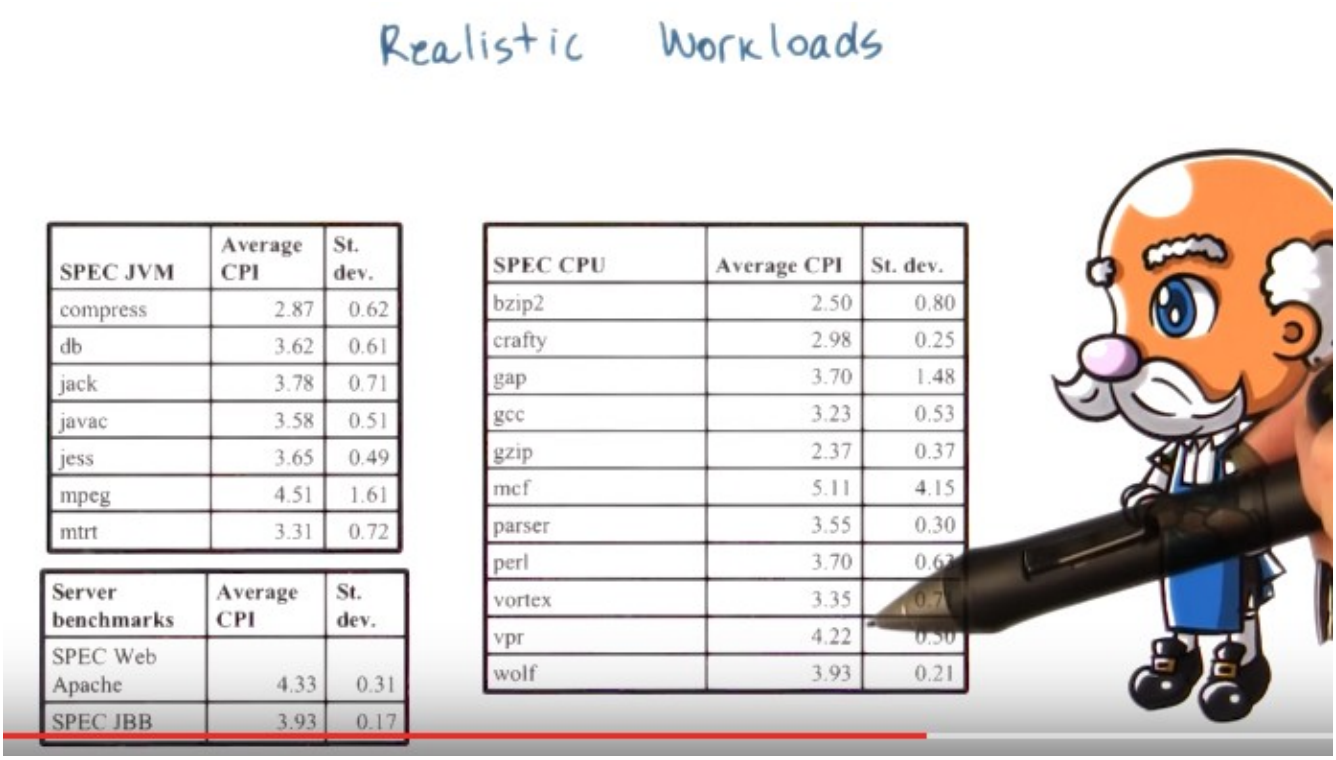
31. Here are the conclusions that we can draw from these results. If we look at the cases for a and b, if we remember from the table that we saw with the actual experiments, in these cases we had a fairly nice mix of tasks with different CPI values on each of the cores. So in these cases, the processor pipeline was well utilized and we're obtaining a high IPC (即同一個 core 中的幾個 task 有不同的 CPI). It is not the maximum IPC of 4, so the maximum performance that one can obtain on the processor that they will be simulating, but it's fairly high. If we'll look at the cases for c and d, in these experiments each of the cores was assigned tasks that have much more similar CPI values (即同一個 core 中的幾個 task 有相近的 CPI). In fact, in the case of d, every single one of the cores 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 for the case a and b. The reason for that is that on some cores, like on core zero in particular, but also on core one, there is a lot of contention for the CPU pipeline. So these (core 0) are the cores that mostly had tasks with a low CPI, so mostly the compute intensive tasks were here. On the other hand, cores two and three, they contribute very little to the aggregate IPC metrics. So basically, they really execute only very few instructions per cycle or a few, small percentage of an instruction per cycle rather, given that a maximum is one. The reason for that is that they mostly have 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 a running tasks that have mixed CPI value is a good thing, that that will lead to an overall high performance of the system.



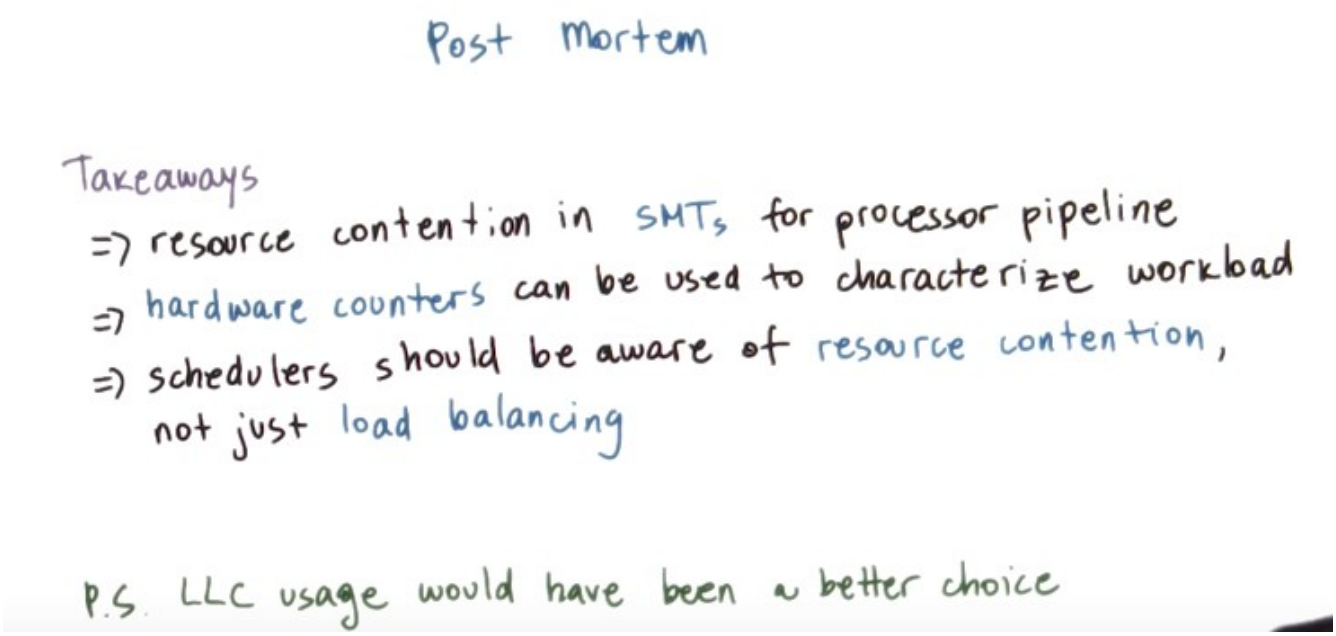
So the conclusion from these results is that CPI's a great metric. And therefore, we should go ahead and build hardware that has a CPI hardware counter and tracks this value so that we can then go ahead and also build operating system schedulers that use this value in order to schedule the right workload mix.



Not so fast. In our discussions so far in the experimental analysis, we used a workloads that had CPI values of one, six, 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 that we used in the synthetic workload?

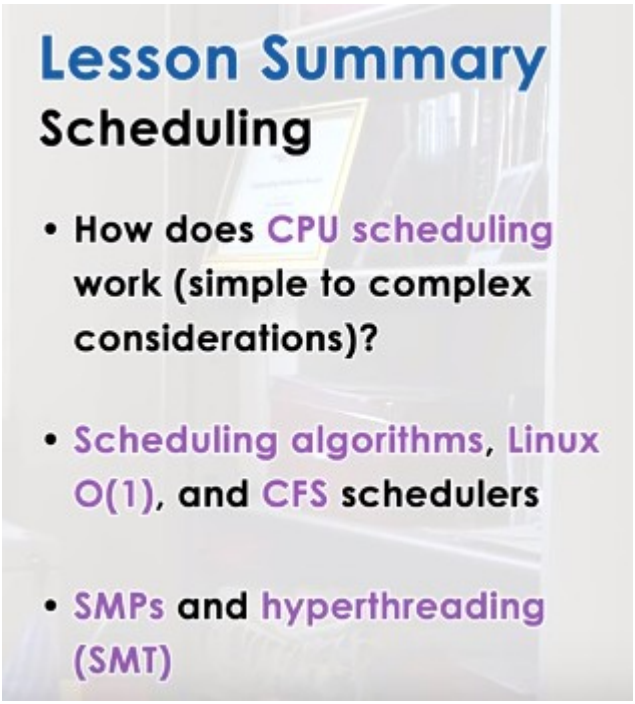


To answer this, Fedorova profiled a number of applications from several benchmark suites. And these benchmark suites are widely recognized in industry and in academia as well that they include workloads that are representative of real world, relevant applications. And let's look at the CPI values for all of these benchmarks. We see that they're all sort of cluttered together. They are (not) in such distinct CPI values like one, six, 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 cycles for instruction as a scheduling metric for hyperthread of platforms, in practice, real workloads don't have behavior that exhibit significant differences in their CPI value, and therefore CPI really won't be a useful metric.



post mortem : 事後析誤, 死後的, 尸體檢查
LLC: last level cache

So I showed you a paper about something that doesn't work. There's still some very important takeaways from this paper. First, you learn about SMTs and some of the resource contention issues there, specifically regarding the processor pipeline as a resource. Next, you learn 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 operating system level resource management. In addition, you learn that it is important to design schedulers that will also think about 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 I/O device. So these principles generalize to other types of resources, not just to the processor pipeline in hardware multithreaded platforms. And by the way, in Fedorova's follow-on 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, multithreaded 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 look in any additional papers that really further explore this issue (即 LLC), not in this course at least.



32. In summary, you should now know how scheduling works in an operating system. We discussed several scheduling algorithms and the corresponding runqueue data structures that they use. We described two of the schedulers that are default in the Linux Kernel, the Completely Fair Scheduler and its predecessor, the Linux Cell One Scheduler. And also, we discuss some of the issues that come up in scheduling when considering multiple CPU platforms. This we said includes platforms with multiple CPUs that's you memory, multi-core platforms as well as platforms with hardware level multithreading.

33. As the final quiz, please tell us what you learned in this lesson. Also, we would love to hear your feedback on how we might improve this lesson in the future.