Concurrency and Threading: CPU Scheduling

Hello, and welcome to this lesson on CPU scheduling.

We will start by looking at the execution states of threads or processes. These can be either running, blocked waiting for resources, or ready to execute but waiting to get access to the CPU. Processes can also be swapped out.

This exploration of thread and process states will make it clear why operating systems have multiple schedulers, which work at different time scales.

We will then focus on the CPU scheduler. We will develop a general scheduler structure, which will allow us to describe a wide variety of different schedulers.

Before we enumerate different schedulers and scheduling algorithms, we explore those measures and criteria that distinguish good from bad schedulers.

Once we have those criteria in place, we discuss a number of specific scheduler, ranging from simple First-In-First-Out to sophisticated scheduling systems like MLFQ.

We will conclude with a brief discussion of CPU scheduling in multicore and multiprocessor systems.

As we explore CPU scheduling in general we will keep making references to how to implement a system scheduler in practice. At the end of this lesson you will have some simple measures to compare different scheduling algorithms, and you will be able to implement simple schedulers for an operating system if you have access to a thread dispatcher.

Let's begin.

State of a Thread/Process

Let's accompany a thread from start to termination. We do this by describing the states in which the thread finds itself.

The first state of course is "starting". If this is a process, the system needs to load the memory, allocate file descriptor tables, and so on.

Once the process or thread has been started, it is READY. This means that it has all the resources, and only needs the CPU to make progress.

Once it gets to the CPU, the thread is RUNNING.

Maybe sometimes later the thread is preempted, and is back to READY.

Sometime later it gets back on the CPU, and then something interesting happens. The thread gets blocked. This typically happens when the thread issues an I/O operation and yields the CPU or when it gets blocked on a lock or other synchronization primitive. As a result, the thread has yielded the CPU and is waiting on some queue until it is unblocked.

After a little while exactly this happens. The thread is unblocked and resumes execution. Of course it does not immediately return to running, as there may be somebody else using the CPU. Rather, the thread becomes READY, and waits for the CPU.

While the thread is waiting to get back to executing on the CPU, the system experiences a memory shortage. It addresses this by swapping out processes. Among others, the process to which this thread belongs is swapped out. The entire state of the process, including all the memory, is copied to secondary storage, and the thread is in the SUSPENDED but READY state.

After some time the memory pressure is reduced, and the process is swapped back in. The thread is back in the READY state, and after some time becomes RUNNING again.

It may get blocked again, and while it is blocked the process gets swapped out again. The thread now is in the SUSPENDED BLOCKED state.

A little bit later the process may be swapped back in, which moves the thread back to the BLOCKED state.

The process gets swapped out again, and the thread now finds itself once more in the SUSPENDED BLOCKED state.

In the meantime, the resource on which the threads was blocked, for example a lock, is released, and the thread is unblocked. Because it is in SUSPENDED state, the thread transitions to SUSPENDED but READY state.

When the process gets swapped back in, the thread moves to state READY, and eventually gets to execute on the CPU again.

While it does that, the thread exits, and so enters the TERMINATED state. As we follow the thread we see the transition graph of thread states take shape.

Schedulers

As we will see in a minute, most of these state transitions need to be carefully managed by the system, because they affect the resource availability in the system. For example, if we start too many processes at once, the system will run into memory shortages. Similarly, we can only have as many thread in the RUNNING state as we have CPUs in the system. These transitions are controlled by what we call "schedulers". Systems typically have three types of schedulers.

One such scheduler controls access to the system by controlling how many and which processes are allowed to actually start execution by transitioning from the STARTING to the READY state. This scheduler is called the long-term scheduler, or job-scheduler, or admission scheduler, the latter because it controls how many processes are admitted to run in the system. The long-term scheduler controls the degree of multiprogramming and must select a good process mix of I/O-bound and CPU-bound processes.

The second scheduler is the CPU scheduler. It allocates the CPU to one of the READY threads. It executes very often, at least every 100ms generally, and therefore has to be extremely fast. Because it runs so often, it is sometimes also called the short-term scheduler.

The third scheduler, is called the medium-term scheduler, or the memory scheduler, or informally just the swapper. It is called memory scheduler because it controls how many and which processes are allowed to stay in memory. If the system runs into memory shortage, the medium-term scheduler chooses a number of processes to swap out to reduce memory pressure.

Multiprogramming

In the following we will focus on CPU scheduling.

Let's assume that a thread, here in gray, starts.

The system immediately admits it, and the thread becomes ready.

After a while it gets hold of the CPU and is now RUNNING.

A little bit later it is blocked, say on some lock.

It the meantime, another thread, here in purple, starts becomes READY, (5) and gets hold of the CPU.

While this thread is running, another thread, here in green, starts, and becomes READY, waiting to get to the CPU, which is occupied by the purple thread. This is multiprogramming in practice: While the gray thread is blocked, the two other threads are either running on the CPU or waiting to get to the CPU.

Let's ignore the purple and the green thread for now.

At some point later the gray thread gets unblocked, and resumes execution. Resuming does not necessarily mean that it gets the CPU. Rather, the thread is READY, waiting for the CPU (10)

Once the thread gets to the CPU it becomes RUNNING, and it may go back and forth between running and READY for a while as it yields the CPU and resumes execution.

CPU Bursts vs. I/O Bursts

If we ignore swapping for now, the thread can find itself in two general modes, for lack of a better term.

The may have all the resources it needs, and only needs the CPU. As it does so, it is either RUNNING, or it is READY to get back onto the CPU.

We say that the thread is in a CPU burst.

The thread can also be blocked, waiting for resources. These periods of blocking time may be very length, in particular if the blocking is due to I/O think of a web server that waits for the next request to come in.

This time spent waiting for events such as IO to happen is called I/O burst.

In this lesson we focus exclusively on threads that are in a CPU burst, that is, threads that compete for the CPU. Access to the CPU is granted by the CPU scheduler.

CPU Bursts and Jobs

If we lay out the execution of the task in a time line, with the threads starting at one point in time and terminating sometime later, then we can draw the execution of the task as a sequence of CPU bursts, where the threads is ready or running followed by intervals where the thread is blocked.

These intervals alternate (we ignore swapping here) until the thread terminates.

As pointed out earlier, the CPU scheduler deals only with threads while they are in a CPU burst.

In the following, when we talk about scheduling algorithms we sometimes simplify things and use the term "Job" rather than talking about a "thread that is in a CPU burst", with the job starting when the thread enters a CPU burst, and the job terminating when the thread leaves the CPU burst.

The CPU scheduler therefore schedules a bunch of jobs. When the thread leaves the CPU burst and enters another CPU burst later, we treat this as a new job starting.

Scheduling Decisions

Let's have a closer look at what a CPU scheduler does.

We say that the scheduler makes scheduling decisions, which means that the scheduler decides who is going to use the CPU next.

Of interest are those threads that are either READY, RUNNING, or BLOCKED. The scheduler decides who gets to get access to the CPU at the following instants in time:

- A scheduling decision is made whenever a thread changes from running to blocked. The CPU of that thread is over, and the thread gives up the CPU. The CPU becomes idle, and a new thread can be scheduled.
- Second, the running thread terminates. Again, the CPU becomes idle, and another thread needs to be scheduled.

- Another case is when a thread becomes READY again after it has been BLOCKED. A new CPU burst starts for this thread. If this thread is important enough, the currently running thread may need to be preempted. for the newly READY thread to resume execution. (5)
- Finally, the current thread changes from RUNNING to READY for whatever reason, maybe preemption.

In this context, we distinguish two classes of schedulers, depending on when the scheduling decisions are made.

If the scheduler only makes decisions when the CPU becomes idle, that is, only in the first two points, then the scheduler is a non-preemptive scheduler. We say that it is non-preemptive because it will never interrupt, or preempt, a running thread. This for the simple reason that it only makes decisions when the CPU is idle, and there is nobody to preempt.

Schedulers that also make decisions while threads are running, and therefore can preempt the current thread, are called preemptive schedulers.

Structure of a Scheduling Subsystem

Let's look at the high-level structure of a scheduling subsystem of an operating system.

The tasks in the READY state are contained in the ready queue. The ready queue is not necessarily a queue, but a data structure that contains thread control blocks for the scheduler to use.

The scheduling subsystem consists of two parts.

The first part is the scheduler in the narrow sense. It uses an algorithm, which we call a "scheduling algorithm" to select the best thread control block. It then extracts the thread control block from the ready queue and hands it over to the Dispatcher, which saves the current thread and loads the new thread onto the CPU.

In this lesson we will focus on the scheduler and on scheduling algorithms to choose which thread to execute next.

What is a Good Scheduler? Criteria

As we will be discussing and comparing a number of scheduling algorithms, it may be a good idea to start by thinking about what makes a good scheduler. There are a number of criteria that one may want to consider.

Criteria that are of interest to the user are, for example the response time, which is the interval from when a job is started until the first response. Of course, the response time cannot be shorter than the execution time of the job.

For really long jobs it is sometimes better to think in term of "normalized response time", which is the ratio of response time over the execution time. Another way to think about normalized response time is in terms of "slow-down". If a job experiences a normalized response time of 2.0, then this means that it ran half as fast as if it had run alone in the system.

Another important measure, although not immediately visible by the user, is the waiting time, which is the time the job is waiting in the queue instead of executing. In a simple system where jobs are either READY or RUNNING, the waiting time is the difference between the execution time and the response time.

For the system operator there are additional quality criteria.

For example, the CPU utilization, which is the percentage the CPU is busy, is an important measure. A low utilization may be due to the system not being used as intensely as planned or may be over-provisioned. Low utilization may also indicate performance problems, such as thrashing, which we learned about when we explored virtual memory.

Throughput is another important measure. System operators are interested in having high system throughput, as jobs completed per unit are easily monetized.

What are then the characteristics of a good scheduler? Any good scheduler should

- maximize CPU utilization and throughput.
- It should also minimize response time and waiting time.

Because we have always several jobs in the system, with potentially widely different characteristics, it is a bit tricky how those measures are to be optimized.

Should we design the system, and select the scheduling algorithms so as to optimize the minimum of all response times? Or should we minimize the maximum response time? (This would be important for some real-time systems.)

Or should we optimize the average values?

Or should we thrive to have small variances of these values, potentially at the cost of higher average values? This last criterion may sound a bit strange, but a video distribution network would prefer having video frames rendered at a regular rate rather than very much faster but very irregularly.

Scheduling Algorithms

Here is a list of scheduling algorithms that we will briefly visit. They are loosely ordered in increasing complexity. We will start with the simplest of all scheduling algorithm, by some called, no-scheduling-at-all, namely the first-come-first-serve algorithm, and we will wrap up with Multilevel Feedback Queue Scheduling and multiprocessor scheduling.

Common Structure of a Scheduler

How should we think of as a scheduling system?

Let's focus on the ready queue. It is central to the scheduling system.

On one end new jobs arrive whenever threads enter a CPU burst and on the other the jobs leave the ready queue to execute on the CPU.

Between those two ends, many schedulers have something that we may want to call a "priority list" of jobs. In practice this is a container of thread control blocks that keeps them sorted in some fashion that is convenient for the scheduler.

We said earlier that the scheduler picks the best job to execute next. We are therefore tempted to think of the scheduler rummaging through the priority list every time it needs to hand a new job to the dispatcher. Scheduling systems are typically not realized that way.

Rather, the Scheduler makes sure that the priority list is kept organized in a way that makes it easy for the scheduler to pick the best job and hand it over to the dispatcher.

As a result, the scheduler make decisions when it queues new jobs into the priority list, then it just picks the first job in the list and hands it over to the dispatcher.

This all sounds awfully abstract. Let's make it real by looking at a few scheduling algorithms.

First-Come-First-Served (FCFS/FIFO)

The first scheduling algorithm is the FCFS algorithms, also called FIFO.

The implementation is trivial, was we maintain a linked list of jobs, of of thread control blocks.

New jobs are appended at the end when they come in.

And the next job to execute is removed from the head of the queue.

The main advantage of this algorithm is that it is very simple.

We get reminded of the disadvantages of this algorithm every time we find ourselves at the supermarket waiting, with one bottle of milk in our shopping cart, behind somebody who does their weekly shopping. Both average and worst-case waiting times can be very large with this algorithm.

In computing systems this is even worse because threads often go back and forth between CPU bursts and access to some resources. As they do so, they tend to get stuck behind a long job as they wait for the CPU and again as they wait behind for the resource. They stay there for a long time as they loop. Other jobs get caught as well, and a so-called "convoy" of job builds up behind the long job.

Shortest-Job-First

An algorithm that performs much better in terms of waiting times is the SJF algorithm.

This algorithm is quite simple as well. Whenever the CPU is idle, the scheduler picks that job that has the shortest execution time. This is like sorting the shopping carts by the number of items in them.

Internally, we keep a data structure that sorts the jobs by their length, with the longest jobs on one end and the shortest on the other.

Whenever a new job arrives it is inserted into this data structure. We notice that this is not as simple as in the case of FIFO, because we need to insert the job into the correct location. This can be done in O(log n) time.

The selection of the the job to run next remains simple. The first job in the sorted data structure is selected.

One great advantage of this algorithm is that it provably minimizes the average waiting time of all jobs in the queue. In a minute we will actually prove this.

The disadvantages are twofold. First, whenever we sort jobs according to some criteria, we run the risk of starving some jobs. In this case, long jobs always find themselves at the end of the queue. Everybody cuts in in front of them. We will see later that there are simple techniques to address this.

A more serious problem is that we are ordering the jobs in order of their length, which in an OS would be the length of the next CPU burst of the thread. The keyword here is "next", and the problem is that the system has no way to know how long the next burst will be. The SJF algorithms would therefore have to be able to foresee the future. In practice, all it can do is to predict the length of the next CPU burst length from past behavior.

SJF Minimizes Average Waiting Time

We said that we were going to prove that SJF minimizes that average waiting time. By this we mean that there is no other algorithm that pick jobs better than SJF.

We will prove this by contradiction. Let's say somebody walks up to you and claims that they have an algorithm, call it Algorithm X, that generates schedules with waiting times shorter than SJF.

You put that Algorithm X to the proof by looking at a schedule generated by X for some set of jobs. If the schedule looks the same as the schedule generated by SJF, then X is just another name for SJF. For Algorithm X to be different from SJF, there has to be at last one schedule for some set of jobs that is different than the SJF schedule. For that schedule to be different, there has to be at least one pair of adjacent jobs in the schedule that are not in SJF order.

In this figure we see such a case: time flows from left to right, and Plong, which is longer than P short, is executed before Pshort. This is therefore not a SJF schedule.

We now do a simple transformation by taking the two jobs and switching their execution order. The waiting time of Pshort decreases by much, while that of Plong increases by little. The waiting time of all the other jobs does not change. Overall, the sum of all waiting times has decreased. This contradicts the statement that Algorithm X always generates the schedule that minimizes the waiting time. Moreover, we can do this transformation for all schedules that are not SJF. Only SJF schedules cannot be improved further. Therefore, SJF is optimal.

Here we see an example how an arbitrary schedule is transformed into a SJF, minimizing the sum of all the waiting times along the way.

Fixed-Priority Scheduler

Another, very popular scheduler in practical systems, is the Fixed Priority scheduler.

Whenever the CPU is idle, we pick that thread with the highest priority.

This is realized using a similar sorted data structure as before.

Here we see how a job is added, and how the first job in the sorted data structure is removed.

The priority of the thread can be, (5), the class of the thread or of the process, or the criticalness of the thread, or others. An important detail here is that the priority is not a function of the job, but rather a function of the thread the job belongs to. Every job in the thread has the same priority. This is different than SJF, where the length of jobs, and therefore their priority, can change for the same thread. This sounds innocuous now, but will come in handy in a minute.

One problem with fixed priority schedulers is their propensity to cause starvation. Low priority threads will always be served last.

This solution is easy to fix, however. We periodically increase the priority of whoever is in the ready queue. This is called “aging”.

Let’s see how this works. This job is at the bottom of the queue. (10)

Periodically, the scheduler bumps the priority of whoever is in the queue at that given point in time. The job gets a bump as well.

After some time, the job gets another bump, and then some more. After more of these bumps, the job is either still in the queue, and is therefore in need of more bumps, or it has reached a high-enough priority to be picked next. (14)

Fixed Priority Scheduling (implementation)

We promised earlier that the priority being a function of the thread rather than of the job would simplify the scheduler implementation.

Conceptually, we have this sorted data structure, which we now draw from top to bottom, with high priority jobs at the bottom and low priority jobs at the top.

When the scheduler adds a new job, it does this by comparing priorities of the new job with those of the jobs in the queue.

For every new job that becomes ready, the scheduler has to insert the thread control block into this sorted data structure. This takes about log N time where N is the number of jobs in the queue.

In practice, in a fixed-priority system, threads have a finite number of different priorities, maybe 8, or 128, or at most 256. (5)

This allows us to greatly simplify the data structure.

We can implement the sorted data structure as an array of FIFO queues. All threads of priority 3 are queued up in FIFO queue #3, for example.

Maybe there is a minor transformation needed, where we map the priority to a particular queue, typically when we don't want to manage as many queues as we have priorities, but this is very simple.

If a thread arrives, we compute the index q of the FIFO queue, and insert the thread at the end of the selected FIFO queue. The cost of this is for all practical purposes constant.

Round-Robin

Let's look at another scheduler, called Round-Robing Scheduler, which is very popular in time-sharing systems. RR uses the FIFO scheduler as its fundamental structure. We have a FIFO queue, where jobs are added at the end, and removed from the front.

We achieve the RR behavior by defining a TIME QUANTUM, which is the maximum amount of time a job may be running on the CPU. If at the end of the time quantum the job is not done, then it is removed from the CPU, we call this PREEMPTED, and returned to the end of the FIFO queue.

Let's observe how this works. A job starts and is added to the queue. It slowly makes its way to the head of the queue as other jobs complete.

Once it is at the head of the queue it is loaded onto the CPU.

Now a timer starts ticking.

If the job terminates before the end of the time quantum when the timer expires, it yields the CPU and terminates. No surprise here.

If, on the other hand, the job is still executing when the timer goes off, it goes back to the end of the FIFO queue.

This closes the round-robin cycle.

Fundamentally, RR is FIFO with preemption after a time quantum.

As we said earlier, it is a popular method to provide time sharing. Everybody gets a slide of the CPU, and as long as we don't have too many competing threads, everybody makes progress at an acceptable rate.

What is tricky about RR is the choice of the time quantum. If we pick the quantum too long, we basically have FIFO. This is not a problem if all the jobs are short, but waiting times can become very large if some of the jobs are long.

When the time quantum is very short, we approximate what is called processor sharing, where RR cycles through the jobs so quickly that each job thinks it owns a share of the CPU.

Small time quanta are not easy to implement because the cause a lot of context switching overhead, and it makes no sense to context switch rather than doing actual computation.

Multilevel Feedback Queue Scheduling

RR schedulers are used in modern operating systems in a very sophisticated fashion as part of so-called multilevel-feedback-queue-schedulers.

Let's look at how these schedulers work, at least conceptually.

Let's start with the fixed-priority scheduler that we know from earlier.

We remember that the scheduler is implemented as an array of FIFO queues, one for each priority level.

Let's now add a time quantum to each of these FIFO queues, which turns each of them into a RR queues. Each of the RR queues is given a different quantum size. (5)

The highest priority queue in this example has size 2ms, the next lower-priority queue 4ms, and so on.

The lowest priority queue has quantum infinity, which makes the queue a FIFO queue.

As a result, high priority threads are handled by the RR queue that has a 2ms quantum. Lower priority threads are served by RR queues that are served with lower priority, but have longer threads. Really-low priority threads are served by the lowest-priority queue. Once a thread in this queue gets to execute, it can do so until it completes without being preempted by the end-of-quantum timer. Note, however, that all running threads get preempted whenever a thread with a higher priority becomes ready.

Just like in the case of basic fixed-priority scheduling, we use aging to prevent starvation. Periodically, all threads get a priority boost. As the priority of a thread increases, the thread is removed from the current RR queue and is added to the next-higher queue. In this way, even the lowest-priority threads eventually get to execute. Whenever a thread gets to run on the CPU, the standard RR mechanism kicks in: An end-of-quantum timer is set, and if the thread yields the CPU before the timer fires, it keeps its priority and gets queued up in the same queue the next time it becomes ready. If, on the other hand, the thread is still running when the timer fires, the thread is preempted. In a basic RR scheduler the thread would be queued up on the back of the RR queue.

In the MLF scheduler, the thread is penalized for over-running the quantum by being "demoted". Instead of being queued up at the end of the same RR queue, it is queued up at the end of the next-lower-priority queue. Now it has a lower priority, but when it acquires the CPU again, it can run for a longer time quantum before being preempted by the End-of-Quantum timer.

Let's see how this works in practice.

A thread arrives, and is enqueued in the 8ms RR queue. Other high-priority threads are running, and so this thread does not get served.

After some time the periodic aging mechanism wakes up, and increases the priority of our thread such that it gets promoted to the next higher-priority queue. It stays put there because still-higher priority threads are running.

After some time the periodic ager wakes up again, and this time the thread is pushed into the highest-priority queue.

When it reaches the front of this queue, it finally gets to run on the CPU. When it starts, the end-of-quantum timer starts.

After two 2ms the timer fires, and the thread is still running.

The thread is preempted and demoted. It gets queued up at the end of the 4ms RR queue. If it gets preempted next time as well, it gets demoted again, this time to the 8ms queue, and so on. MLF schedulers do very well in general-purpose computers, because they tend to boost the priority of interactive processes over processes that compute-bound. Threads of compute-bound processes tend to have longer CPU burst, and therefore tend to get preempted when they are in high-priority queues. As a result, they end up in lower-priority queues. This is fine because compute-bound processes are often not interactive. Interactive workloads on the other hand typically have short CPU bursts. As a result, they don't get demoted from high-priority queues. Because they, like everybody else, get promoted periodically, they tend to run at higher priority. This is great for the user, who gets a more responsive interaction.

There are some workloads where basic MLF scheduling fails, for example for some multimedia applications or for games. In such applications the interactive workload may require compute-intensive rendering or AI computations, and therefore tends to have long CPU bursts. This causes demotions, and parts of these applications then run at low priorities. In practice this can be addressed by tweaking basic MLF scheduling to handle these situations correctly.

Dynamic Multiprocessor Scheduling

Since when multiprocessors and then multicore systems became popular, the scheduling for such systems became important as well. In this lesson we will briefly touch how this affects the scheduler design.

Let's start with a simple single-processor system, which of course has a single scheduler. For now we don't care what scheduling algorithm is being used.

This scheduler can be easily extended to work with multiple processors by changing how the the dispatcher interacts with the scheduler.

For example, whenever one of the CPUs becomes idle, this is signaled to the scheduler, which hands over to the dispatcher the next thread to dispatch to the CPU.

The system has a single scheduler, which we call "global scheduler", which serves all the processors. This is sometimes also called "dynamic multiprocessor scheduling" because the scheduler decide dynamically where a thread is going to execute next, depending on which CPU will become idle first.

The main disadvantage of this type of scheduling, in addition to widely varying response times, is this dynamic behavior, which causes difficulties for the threads, in particular when it comes to cache behavior. Since each CPU has its own cache, it is important for a preempted thread to return to the same CPU, in particular if it was preempted for only a short time. If it gets dispatched to a different CPU, it looses the cache, and memory references need to go to memory again. The same holds for the TLB. In some cases this can become particularly bad when we have multiple threads of the same process that are executed on different processors.

Static Multiprocessor Scheduling

It is therefore much better to have for each processor, a separate scheduler.

We call these schedulers "local schedulers" because they each handle the workload on a single processor.

New threads are pass through a thread partitioning module, which assigns the thread to one of the CPUs.

The thread assignment in this way partitions the set of threads, and each partition executes on a different processor.

This is also called "static" multiprocessor scheduling because the thread assignment to processors is static. Once a thread has been assigned to a processor, it typically does not execute on a different processor. This form of static scheduling has a number of advantages, chief among which is that threads can hold on to their cache. Also, response times are generally less variable and therefore more predictable.

The job of the task partitioning module is not easy, as it has to balance the workload across the processors. It has to do so while keeping threads of the same process when possible on the same processor in order to minimize memory access overhead. When the threads of a process need to be run in parallel, however, these threads must be allocated to different processors again to achieve parallelism. Many other constraints have to be considered as well, making multiprocessor schedulers tricky to design and to analyze.

Summary: CPU Scheduling

We have come to the end of this lesson on CPU scheduling.

We have explored together how threads progress through the system from the moment they are started until they terminate.

We learned about the three schedulers that typically shepherd the threads through their execution. We focused on the CPU scheduler, which handles the running thread and all the READY threads.

We learned about a generic scheduler framework that allows us to describe a very large class of schedulers.

Then we explored a number of different scheduling algorithms, in particular FIFO, SJF, FP, RR, and MLFQ scheduling.

In the end we learned about some of the difficulties in scheduling in multiprocessor or multicore systems. We distinguished between dynamic and static multiprocessor scheduling and compared the two approaches.

With what you have learned in this lesson, you are able to assess scheduling algorithms and make an informed decision when selecting such an algorithm for your system. You will also be able to implement or modify simple scheduling algorithms in a simple operating system. Note that we have simplified the description of the algorithms presented in this lesson. In practice, this algorithms may be implemented differently, in particular the MLF scheduler. Conceptually they remain very similar to what we describe in this lesson, however.

I truly hope that you enjoyed this lesson on CPU scheduling and that you will be able to put this knowledge into practice when you build your own scheduling subsystem.

Thank you for watching.