Processes and Scheduling Algorithms

Multiprogramming

We use the term *process* for a computer program that is running on a system. It might not be currently using the CPU, but someone or something has started it. The OS maintains a data structure (sometimes called a *process control block*) that contains info about the process, such as its current working directory, which files it has open, its state, and more. Depending on the OS, there can be a number of possible states the a process is in. Many of the states are variations on these three:

- Running — The process is currently using the CPU.
- Ready — The process is ready to use the CPU, but it isn't using it, most likely because something else is.
- Blocked — The process doesn't want to use the CPU right now because it is waiting for something. That something might be I/O like user input or a file write. The process could also be waiting for a resource that another process is currently using.

On a modern OS, dozens, even hundreds, of processes may be active at the same time, even if there is only a single CPU. This is the idea of *multiprogramming*, that multiple processes can share a CPU. This can happen in a couple of ways:

1. A process that is currently using the CPU might block for I/O or a resource, which means it won't need the CPU again until the I/O or resource is ready. Therefore, the OS can swap that process out for another that is in the ready state.
2. The OS can give each process a small window of time to use the CPU. One the process's time is up, the OS will switch to a new process. Each process gets a little bit of time on the processor, and the OS quickly switches from process to process, giving each one the illusion that it has full access to the CPU.

In #2 above, the amount of time each process gets is called its *time slice* or *time quantum*. Time slices typically range from a few milliseconds to a few hundred milliseconds, depending on the process and the OS.

Context switches

The changeover from one process to another is called a *context switch*. A context switch essentially puts the currently running process into a state of suspended animation, while pulling the new process out of that state and putting it into the CPU.

The *context* in a context switch is the values of the CPU registers that the process is using. These register values are critically important to a running process. They serve as a workspace for whatever it is currently doing and they track important state, such as what instruction is currently running. When a new process uses the CPU, it will need to store its own info in those registers, which would destroy whatever values are there. To put the current process into suspended animation, we save those register values to memory, and when the process is restarted, we copy those saved values back into the registers.

Context switches are triggered by a few possibilities:

1. The currently running process decides to give up (or *yield*) the CPU. It can do this to be nice to other processes, because it's done using the CPU, or because it needs to block for I/O.
2. The currently running process's time slice has ended. Most systems have a hardware timer. When that timer goes off, the current process's time is up.

3. Some other event can happen, such as a higher priority process needing to run.

The exact details of how a context switch is done vary depending on the architecture of the system and on the OS. On many systems, the above will cause a trap operation that transfers control over to the OS. The current process's register values are saved in memory to its kernel stack or the process control block. The OS then chooses a new process to run, and the register values of that new process are moved from memory into the registers, and control is transferred over to the new process. Depending on the system, the cache may need to be flushed (cleared out) and work may need to be done related to the two processes' memories.

Two important questions come up: (1) How long do context switches take? (2) What is a reasonable value to use for the time slice? For #1, it depends on the system, but typical values range from around .1 microseconds to a few microseconds. For #2, values around 20 ms are common, but there are situations where the range can be more like 1 millisecond to a few hundred milliseconds.

There are problems both with long and short time slices. For a long time slice, suppose we have 10 processes on a system and each has a time slice of 500 ms. If we cycle through the 10 process round-robin, then each process has to wait $9 \cdot 500 = 4500$ ms in between when its time slice is over and when it next gets the CPU. A 4.5-second delay like this can make the system seem really laggy to users.

For a short time slice, suppose we have a time slice of .01 ms (which is 10 microseconds), and suppose context switches take 1 microsecond. Then, out of every 11 microseconds, 10 of them are used for real work and 1 is used to do the context switch. So 1/11 or about 9% of the system's time is spend doing context switches. If the time slice is 20 ms (20,000 microseconds), then that fraction becomes 1/20001, which is a much smaller percent. The take-home message here is that the overhead due to context switching starts to take up a noticeable percent of the system's time if we switch too often.

Further, context switches are bad for cache performance. When a process starts up, it takes some time for the cache to be filled with frequently accessed items that the process uses. On a context switch, that cache is usually wiped out, replaced with things for the new process. If context switches are too frequent (i.e., we use too small of a time slice), then there is never much chance for the cache to become useful. By the time it's filled up with things the process needs, it's already time to switch to a new process.

Cooperative vs. preemptive multitasking

Under *cooperative* or *non-preemptive multitasking*, a process uses the CPU until it voluntarily gives up the CPU. The opposite is *noncooperative* or *preemptive multitasking*. In cooperative multitasking, the process can give up the CPU if it decides it wants to yield it to other processes, and it also gives it up any time it needs to make a system call. Early versions of MacOS and Windows used cooperative multitasking, and the scheduler for certain real-time processes (more on this a little later) in Linux is essentially cooperative.

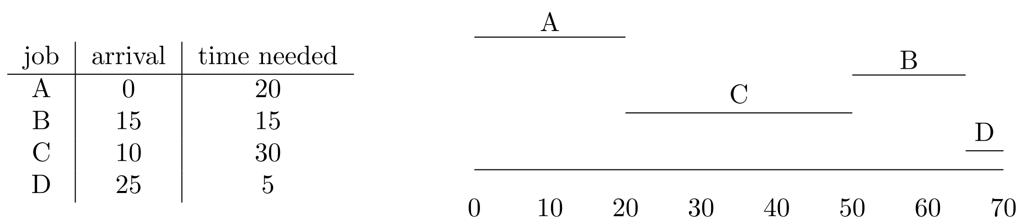
Cooperative scheduling is the easier of the two approaches to implement, but it has some obvious problems, namely that processes can hog the CPU by refusing to give it up voluntarily. Another problem is if a process accidentally gets caught in an infinite loop, then it can't give up the CPU. The only solution might be to restart the system.

Scheduling Algorithms

Above, we talked about switching from one process to another. Another big question is which process to switch to next. We'll see several examples below that all have benefits and drawbacks. Some of the scheduling algorithms we will look at apply to more than just process scheduling, and, indeed, people refer to the things being scheduled as jobs rather than processes when discussing these algorithms.

The first few scheduling algorithms are general algorithms that apply to many situations, though we will especially focus on their uses in batch systems. In batch systems, jobs generally don't have much, if any, user interaction. Often these jobs are things like running the payroll or accounts receivable for a company. For these jobs, we will often know how long they will take to run ahead of time.

First In, First Out (FIFO) This is the way lines work at store checkouts. It's first come, first served. The only rule is that when deciding which job goes next, the one that arrived the earliest is the one that goes next. Here we will consider the non-preemptive version of FIFO, in that once a job is running, it will run until it's done and won't be interrupted. If something better comes along, it doesn't matter. We will stick with what's running until it's done. Below we have an example of the algorithm. On the left is a table of when jobs arrive and how long they will need to run. On the right is a graph of when each is scheduled to run by FIFO.

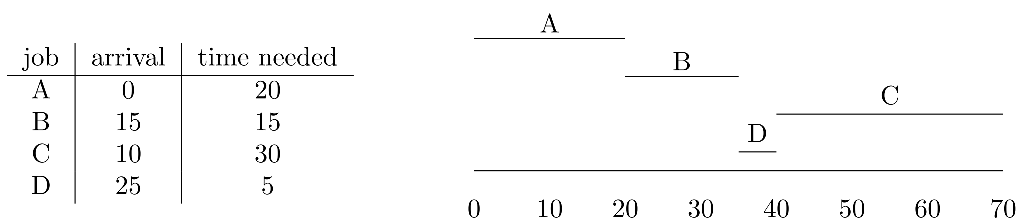


job	arrival	time needed
A	0	20
B	15	15
C	10	30
D	25	5

At the start (time 0), only A is available, so A is what runs. Since FIFO is non-preemptive, A will run until it's done. It finishes at time 20 and then we have to decide what to run next. Both B and C have arrived by that point, with C arriving earlier, so C runs. C finishes at time 50, and at that point, we have to decide between B and D. B arrived earlier, so B runs. Finally, when B is done, D runs, since it's the only thing left.

The main selling point of FIFO is that it is probably the simplest algorithm to implement. But it has some drawbacks. Suppose job A arrives at time 0 and needs to run for 1000 ms, while B, C, D, and E arrive at time 1 and need only 5 ms each. They will all have to wait for A to finish before they can run, even though they are very short.

Shortest Job First (SJF) It's a simple idea: when choosing between available jobs, pick the one that needs to run for the least amount of time. We will only consider the non-preemptive version of this algorithm, so we assume that once something is running, there is no stopping it until it's done. Here is how SJF works on the same example we used for FIFO:



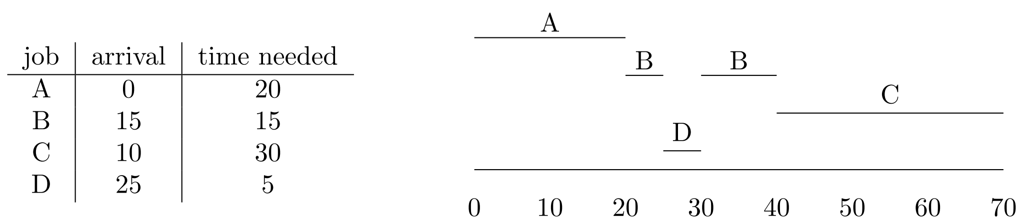
job	arrival	time needed
A	0	20
B	15	15
C	10	30
D	25	5

A is the only thing available at time 0, so it runs first. When A is done, we have to choose between B and C, which arrived while A was running. We choose B because it needs to run for 15 versus C which needs to run for 30. At time 35, B is done and we have to choose between C and D. We choose D since it needs to run for 5 versus 30 for C. Finally, once D is done, C runs since it's the only thing left.

Shortest job first is also pretty easy to implement, though it requires a little more work than FIFO since we have to sort jobs by how much time they need. SJF can actually be proved to be optimal for turnaround time (the time it takes between when a job arrives and when it finishes) in the case that all jobs arrive at the same time. However, it has a similar problem to FIFO if a long-running job arrives right before some short-running jobs. They will all have to wait for the long-running job to finish since we don't preempt running jobs. It also requires us to know how long jobs will be running for, and we often won't know that information.

Shortest Time to Completion First (STCF) This algorithm does allow running jobs to be preempted. The basic idea is whenever a new job arrives, we make a decision whether to keep running the

current job or switch to the new one. We compare how much time the current job has until it's done with how long the new job needs, and we go with whatever is shorter. When a job finishes, we look at how much time each other available job has left and go with the one that has the least amount left. Here is the same example as earlier, done with STCF:



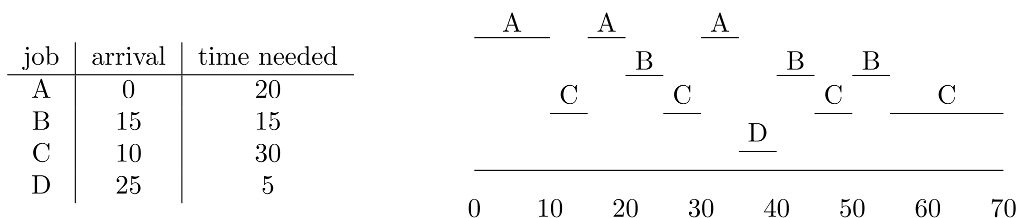
job	arrival	time needed
A	0	20
B	15	15
C	10	30
D	25	5

Here is a breakdown of all the decision points:

Time	Event	Action
0	Start	A is only one available, so it runs.
10	C arrives	A needs 10 more, C needs 30, so A keeps going.
15	B arrives	A needs 5 more, B needs 15, so A keeps going.
20	A finishes	B needs 15 and C needs 30, so B goes.
25	D arrives	B needs 10 more and D needs 5, so D goes.
30	D finishes	B needs 10 more and C needs 30, so B goes.
40	B finishes	C is the only one left, so it goes.

STCF is more complicated than the others, but not too much. It works pretty well. In the situation mentioned earlier where a long-running process arrives at the start and then a few short-running processes arrive right after, those short-running processes will not have to wait forever, like in FIFO and SJF. However, this can create a new problem. Suppose short-running processes keep arriving. Say A arrives at time 0 and needs 1000 milliseconds, and then every millisecond thereafter, a new process arrives that needs 5 ms. Then A will never get to run at all. This is an example of *starvation*, a topic that arises frequently in operating systems, where something is unable to get the resources it needs. There is no perfect scheduling algorithm, just algorithms with various strengths and weaknesses. You often will have to choose one whose strengths match your situation and whose weaknesses are manageable.

Round Robin This is an algorithm where jobs take turns running, with each getting a short time slice before being interrupted so that the next job can run. For instance, if jobs A, B, and C are available and the time slice is 5 milliseconds, then A will run for 5, then B runs for 5, then C runs for 5, then A runs again for 5, etc., where we keep cycling between the processes in order. There are a few details to worry about. Let's consider only a non-preemptive version, where we won't interrupt jobs in the middle of their time slice. To implement Round Robin, we can put all the processes into a queue. When a new process arrives, it gets put at the back of the queue. When the running process's time slice is up, it moves to the back of the queue, even behind any new processes that have arrived. Below is how the algorithm works on the example we've been using throughout this section.



job	arrival	time needed
A	0	20
B	15	15
C	10	30
D	25	5

The basic idea of Round Robin isn't too tricky, but it can be a bit of a pain to do an example like this by hand when processes keep arriving and finishing. It helps to keep track of the queue like below. In this table, the front of the queue is on the left, with the process at the front being the one that is currently running.

time	queue	comment
0	A	Only A is available
5	A	Only A is available
10	C A	C arrives and gets a turn
15	A B C	C arrives but it's A's turn next
20	B C A	
25	C A D B	D arrives
30	A D B C	A finishes
35	D B C	D finishes
40	B C	
45	C B	
50	B C	B finishes
55	C	Only C is left
60	C	
65	C	C finishes

Round Robin is one of the most fair algorithms in that all jobs get a chance to run. Starvation is not an issue. Round Robin is good for *response time*, which is the time it takes from when a process first arrives to when it first gets to use the CPU, but it is not good for *turnaround time*, the time it takes from when a process arrives to when it finishes, because there is so much sharing of the CPU.

Priorities

One of the key factors in scheduling is that certain processes are more important than others and should get priority. Many operating systems will prioritize things in the following order:

1. Real-time tasks — These are things that need to be done right away or else something bad happens. For instance, if there is an incoming phone call on a cell phone, that process needs to run as soon as possible or else the call will be missed. Real-time tasks are usually broken into two types: *soft real-time* and *hard real-time*. Soft real-time tasks are things like video streaming, where you want to have the stream displayed as soon as it comes in, but if a few seconds are lost, it's not the end of the world. Hard real-time is for tasks for which something bad would happen if a deadline is not met. For instance, many tasks in a self-driving car are hard-real time. Most desktop OSes only handle soft real-time tasks.

2. Interactive tasks — These are tasks where the user interacts regularly with the process, like a word processor or a web browser. These need relatively high priority or else the process will seem laggy and unresponsive.

3. CPU-bound tasks — These are things that don't have much user interaction and often require a lot of CPU time. Things like image and video processing or scientific calculations are CPU-bound tasks. These tasks can often wait to run until nothing more important is using the CPU. They also tend to run more efficiently if they can get larger chunks of CPU time where they aren't frequently interrupted. Frequent interruptions hurt cache performance since the process's data in the cache would be replaced with data from other processes.

Some tasks can be combinations of interactive and CPU-bound. For instance, a programming language IDE is very interactive, but it also can do a lot of computation for things like autocomplete and syntax checking. Video games are another example.

Priorities in Linux Many operating systems assign priorities to processes and provide ways for those priorities to be changed. In Linux, there are 140 priority levels. Levels 0 to 99 are for real-time processes, and levels 100 to 139 are for normal processes. Level 0 has the highest priority and 139 the lowest. Levels 100 to 139 correspond to what are called *nice values*. Level 100 corresponds to nice value -20, Level 101 corresponds to -19, all the way up to Level 139, which corresponds to nice value 19. The default nice value

of a process is 0. Processes can increase their nice value (i.e., be nice to other processes), which has the effect of lowering their priority. Only privileged users (such as root or the OS kernel) can give a nice values in the negative range.

Priority-based scheduling in batch systems Some batch systems use a simple variation on FIFO that accounts for priorities. Each priority level has a queue. When choosing the next process to run, the OS picks the highest priority queue that has processes in it, and runs those processes in a FIFO style.

Multilevel Feedback Queue (MLFQ)

The MLFQ was introduced in a predecessor to Multics in 1962. It was used in Multics, and schedulers like it are used in Windows, macOS, and many variants of Unix. It has several queues (20 to 60 is typical), each with different priorities. The higher priority queues have shorter time slices than the lower priority queues. For instance, the highest priority queue might have a time slice of 10 ms, while the lower queues might have time slices of 200 ms. Here are the key rules to the MLFQ:

1. When picking a process to run, the OS picks the highest priority queue that contains processes and runs them round-robin.
2. When a process uses up all of its time in a particular queue, it drops down to the next lower queue.
3. Periodically, something needs to be done to boost the priorities of processes stuck in the lower queues.
4. New processes are inserted into the top queue.

To get a sense for how this works, picture two processes: a CPU-bound process and an interactive process. Both will start their lives in the highest priority queue. Let's say that queue has a time slice of 10 ms. When the CPU-bound process runs, it will likely spend all of its time slice using the CPU and by Rule #2 above, it will be dropped to the next queue down. When it gets a chance to run again, it will likely use up its entire time slice in that next queue and get dropped still further.

The interactive process, on the other hand, might only end up running for 5 ms before blocking for I/O (it's interactive, so it spends most of its time waiting for user input). So it won't use up its 10 ms time slice and it will get to stay in the top queue. A little later, after it wakes up from waiting for I/O, it will get to run again. Maybe this time, it goes 3 ms until blocking for I/O again. At this point, it has used up $5 + 3 = 8$ of its 10 ms time in the top queue, so it still gets to stay. It will only get booted once its combined running time in that queue reaches 10 ms.

So, the way the MLFQ is designed, since interactive processes spend so much time sleeping (i.e., not using the CPU because they are waiting for I/O), they tend to stay in the higher queues. CPU-bound processes, on the other hand, will fall to the lower-priority queues. However, that is not necessarily a bad thing for them. Those lower priority queues have longer time slices, which is good for CPU-bound threads, as they get to keep their caches warm for longer when they are running.

The higher level queues will be filled with threads that are mostly sleeping, so Rule #1 won't totally prevent things in the lower queues from running. However, processes that are in the system long enough, even interactive ones, may eventually use enough CPU time that they will fall into the lower queues. To prevent starvation and to make sure interactive processes that fall into those lower queues can get CPU time, periodically the system should boost processes back into higher queues. This is Rule #3 above. One way to do that is just to every so often throw all the processes back into the top queue. Another approach, used by Microsoft Windows, is to temporarily boost a process into a higher queue if the window associated with it gets moved to the foreground or if some I/O event related to it gets completed.

Lottery scheduling and stride scheduling

Lottery scheduling In some systems, you simply want to allocate CPU time so that each process gets to use the CPU for a certain percentage of the time. For instance, maybe we want A to get 50% of the time, B to get 30%, and C to get 20%. An effective way to do this is to hold a lottery. This approach is fairly easy to implement and is useful in other contexts besides scheduling.

In this example, we pick a random integer from 1 to 100. If it comes out in the range from 1 to 50, then A gets to run. if it is in the range from 51 to 80, then B gets to run. Otherwise, C gets to run. While it's possible that the random numbers could continually come out greater than 80 and cause C to run too often, in reality that is very unlikely to happen according to the law of large numbers (from probability). It says that in the long run, it is very likely that things will come out close to 50-30-20.

Stride scheduling This solves the same problem as lottery scheduling but without using randomness. Let's take the same 50-30-20 example as above. We pick a large number (the exact value isn't too important). Let's go with 1200. We divide that number by each of 50, 30, and 20 to get values called *strides*. The strides for A, B, and C come out to $1200/50 = 24$, $1200/30 = 40$, and $1200/20 = 60$.

In stride scheduling, we keep track of how much time each process spends using the CPU, but the amounts are not real times. Each time a process uses the CPU, we increase a usage variable by the process's stride amount. For instance, when A runs in the example above, its usage is incremented by `usage = usage + 24`. When B runs, its usage is incremented by `usage = usage + 40`. When B runs, its stride is larger. We can think of this either as time passing faster for B than A or as B's usage of the CPU as being more expensive than A's.

When choosing a process to run, we always pick the one with the least usage. We give each process an equal length time slice, and at the end of that, we update its uses by using the formula `usage = usage + stride`. Advancing by the stride amount means that lower priority processes (that are supposed to get a smaller share of the CPU) will advance their usage more quickly, making them less likely to run.

Here is how this would run with the three processes above. The table below shows how the usage values evolve over time and what process will run. Initially, we assume all three processes start with usage 0. We'll break ties alphabetically.

time	A	B	C	who runs next
0	0	0	0	A
1	24	0	0	B
2	24	40	0	C
3	24	40	60	A
4	48	40	60	B
5	48	80	60	A
6	72	80	60	C
7	72	80	120	A
8	96	80	120	B
9	96	120	120	A

Notice that A runs more often than B, who in turn runs more often than C. In fact, A ran 5 times, B ran 3 times, and C ran 2 times, matching the desired 50-30-20 breakdown. The percentages won't always match this perfectly, but in the long run they will always be very close to 50-30-20. Stride scheduling is used in the current Linux scheduler, as we will see below.

Scheduling in Linux

As mentioned earlier, Linux treats real-time processes separately from ordinary processes. Those real-time processes always get priority over others. Real-time processes can either be run with a FIFO system or

round-robin, with the FIFO ones getting priority. Those FIFO processes can't be preempted except by a higher priority FIFO process.

For ordinary threads, in early versions of Linux a fairly simple scheduler was used. It was replaced around 2003 by something called the O(1) scheduler that was very fast at picking new threads to run, but turned out to have issues with interactive threads. In 2007, the main scheduler become the *Completely Fair Scheduler* (CFS), which is still the main one today. CFS is a stride scheduler. CPU usage for each process is tracked by a variable called `vruntime` (the `v` is for virtual). Below are some highlights of the scheduler. The full scheduler is over 10,000 lines of code, as the scheduler needs to deal with all sorts of real-world problems.

- The scheduler always chooses to run the process with the smallest `vruntime` value. That process can be chosen very quickly since things are stored in an efficient data structure called a red-black tree. When a process finishes its time slice, it is reinserted into the tree with its now higher `vruntime` value.
- Only runnable processes are able to be chosen. If a thread is sleeping (like if it is waiting for I/O), then it is not part of the red-black tree. Tasks that sleep a lot (i.e. interactive tasks) will naturally have a lower `vruntime` than CPU-bound processes since interactive tasks spend a lot less time using the CPU. Thus, when they need to run, they will be able to. It is possible that a process that sleeps a ton will have a `vruntime` that is so much lower than everything else that when it finally runs, it will hog the CPU and starve other processes. To prevent this, when a process wakes up from sleeping, its `vruntime` is updated so that it is no lower than the minimum of all the `vruntime` values of processes in the system.
- A process is able to spawn new mini-processes called *threads* to do some of its work. CFS schedules each of those threads, but all the threads are grouped together as far as `vruntime` is concerned so that a process can't use threads to get an unfair share of the CPU.
- If there are less than 8 processes, the time slice each gets is given by 48 ms divided by the number of processes. For instance, if there are 3 processes, then each gets a 16 ms time slice. If there are 8 or more processes, then each gets a 6 ms time slice. Time slices less than 6 ms would mean context switches would be happening too much, which is inefficient. When each process runs, it would get that 6 ms (or whatever) time slice, and priorities are handled by certain processes getting more turns at the CPU according to the stride scheduling rules.
- CFS wakes up every millisecond to see if a decision needs to be made. For instance, a process that just woke up (maybe due to a keypress) might get to preempt the currently running process.
- The proportion of time each process gets to use the CPU is determined by its nice value (priority). Here is how it works: Each nice value corresponds to a weight. Nice value 0 is given weight 1024. The weight for each nice value is roughly 1.25 times the weight for the next nice value. The formula for the weights is approximately $1024/1.25^n$, where n is the nice value. The weights for nice values from -2 to 2 are 1586, 1277, 1024, 820, 655.

  A process's share of the CPU is given by dividing its weight by the sum of the weights of all the processes. For example, suppose we have processes A, B, and C with nice values -2, 0, and 2, respectively. Then A will get $1586/(1586+1024+655) \approx 48.6\%$ of the CPU time, B will get $1024/(1586+1024+655) \approx 31.3\%$ of the CPU time, and C will get $655/(1586+1024+655) \approx 20.1\%$ of the CPU time.

A little about multicore scheduling

Most CPUs nowadays have multiple cores. A quad-core CPU can actually be running four processes simultaneously. This makes scheduling more tricky. Here are a few highlights of issues important in scheduling on a system with multiple cores.

- Affinity — It's a good idea to try to schedule things on the same processor they used the last time they got to run. The reason for this is some of their data might still be in that CPU's cache, which can provide a speedup.
- Interactive vs CPU-intensive tasks — If a system has a dozen interactive tasks that spend most of their time sleeping and two long running CPU-intensive tasks, it makes sense to schedule all the interactive tasks on one CPU and give an entire CPU to the other two tasks. That way, the caches for the CPU-intensive tasks can stay relevant for a long time. The interactive tasks on the one CPU spend most of their time sleeping and don't really need much CPU time. Keeping them separate from the CPU-intensive threads saves continuous interruptions. It's just like with people — if you're interrupted while trying to do something, it takes some time to get your focus back. With a computer, getting the focus back means getting the cache back to being relevant.
- Load balancing — With multiple processor cores, it can happen that one processor ends up very busy and another not so much. Load balancing involves detecting if a processor is not busy enough and moving processes over to it.
- Single queue versus multiqueue scheduling — A single queue scheduler has a queue of waiting processes, and it hands them out to different cores. A multiqueue scheduler has separate queues for each core. This is a little like lines at a grocery store. Some places have a single line and the person at the front of the line goes to whatever register opens up first. This is like single queue scheduling. Multiqueue scheduling would be like where each register has its own line.

  There are pros and cons to each approach. It's straightforward to adapt ordinary scheduling algorithms to work with single queue scheduling, but this approach involves something called *locks* (more on these in another section of notes), which introduces a significant overhead. With multiqueue schedulers, the big problem is load balancing.