10

Multiprocessor Scheduling (Advanced)

This chapter will introduce the basics of **multiprocessor scheduling**. As this topic is relatively advanced, it may be best to cover it *after* you have studied the topic of concurrency in some detail (i.e., the second major "easy piece" of the book).

After years of existence only in the high-end of the computing spectrum, **multiprocessor** systems are increasingly commonplace, and have found their way into desktop machines, laptops, and even mobile devices. The rise of the **multicore** processor, in which multiple CPU cores are packed onto a single chip, is the source of this proliferation; these chips have become popular as computer architects have had a difficult time making a single CPU much faster without using (way) too much power. And thus we all now have a few CPUs available to us, which is a good thing, right?

Of course, there are many difficulties that arise with the arrival of more than a single CPU. A primary one is that a typical application (i.e., some C program you wrote) only uses a single CPU; adding more CPUs does not make that single application run faster. To remedy this problem, you'll have to rewrite your application to run in **parallel**, perhaps using **threads** (as discussed in great detail in the second piece of this book). Multi-threaded applications can spread work across multiple CPUs and thus run faster when given more CPU resources.

ASIDE: ADVANCED CHAPTERS

Advanced chapters require material from a broad swath of the book to truly understand, while logically fitting into a section that is earlier than said set of prerequisite materials. For example, this chapter on multiprocessor scheduling makes much more sense if you've first read the middle piece on concurrency; however, it logically fits into the part of the book on virtualization (generally) and CPU scheduling (specifically). Thus, it is recommended such chapters be covered out of order; in this case, after the second piece of the book.

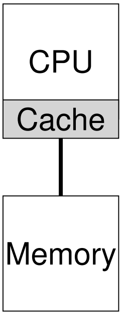


Figure 10.1: **Single CPU With Cache**

Beyond applications, a new problem that arises for the operating system is (not surprisingly!) that of **multiprocessor scheduling**. Thus far we've discussed a number of principles behind single-processor scheduling; how can we extend those ideas to work on multiple CPUs? What new problems must we overcome? And thus, our problem:

> CRUX: HOW TO SCHEDULE JOBS ON MULTIPLE CPUS
>
> How should the OS schedule jobs on multiple CPUs? What new problems arise? Do the same old techniques work, or are new ideas required?

10.1 Background: Multiprocessor Architecture

To understand the new issues surrounding multiprocessor scheduling, we have to understand a new and fundamental difference between single-CPU hardware and multi-CPU hardware. This difference centers around the use of hardware **caches** (e.g., Figure 10.1), and exactly how data is shared across multiple processors. We now discuss this issue further, at a high level. Details are available elsewhere [CSG99], in particular in an upper-level or perhaps graduate computer architecture course.

In a system with a single CPU, there are a hierarchy of **hardware caches** that in general help the processor run programs faster. Caches are small, fast memories that (in general) hold copies of *popular* data that is found in the main memory of the system. Main memory, in contrast, holds *all* of the data, but access to this larger memory is slower. By keeping frequently accessed data in a cache, the system can make the large, slow memory appear to be a fast one.

As an example, consider a program that issues an explicit load instruction to fetch a value from memory, and a simple system with only a single CPU; the CPU has a small cache (say 64 KB) and a large main memory.

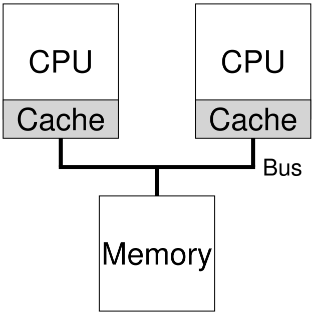


Figure 10.2: **Two CPUs With Caches Sharing Memory**

The first time a program issues this load, the data resides in main memory, and thus takes a long time to fetch (perhaps in the tens of nanoseconds, or even hundreds). The processor, anticipating that the data may be reused, puts a copy of the loaded data into the CPU cache. If the program later fetches this same data item again, the CPU first checks for it in the cache; if it finds it there, the data is fetched much more quickly (say, just a few nanoseconds), and thus the program runs faster.

Caches are thus based on the notion of **locality**, of which there are two kinds: **temporal locality** and **spatial locality**. The idea behind temporal locality is that when a piece of data is accessed, it is likely to be accessed again in the near future; imagine variables or even instructions themselves being accessed over and over again in a loop. The idea behind spatial locality is that if a program accesses a data item at address x, it is likely to access data items near x as well; here, think of a program streaming through an array, or instructions being executed one after the other. Because locality of these types exist in many programs, hardware systems can make good guesses about which data to put in a cache and thus work well.

Now for the tricky part: what happens when you have multiple processors in a single system, with a single shared main memory, as we see in Figure 10.2?

As it turns out, caching with multiple CPUs is much more complicated. Imagine, for example, that a program running on CPU 1 reads a data item (with value D) at address A; because the data is not in the cache on CPU 1, the system fetches it from main memory, and gets the value D. The program then modifies the value at address A, just updating its cache with the new value D'; writing the data through all the way to main memory is slow, so the system will (usually) do that later. Then assume the OS decides to stop running the program and move it to CPU 2. The program then re-reads the value at address A; there is no such data

CPU 2's cache, and thus the system fetches the value from main memory, and gets the old value D instead of the correct value D'. Oops!

This general problem is called the problem of **cache coherence**, and there is a vast research literature that describes many different subtleties involved with solving the problem [SHW11]. Here, we will skip all of the nuance and make some major points; take a computer architecture class (or three) to learn more.

The basic solution is provided by the hardware: by monitoring memory accesses, hardware can ensure that basically the "right thing" happens and that the view of a single shared memory is preserved. One way to do this on a bus-based system (as described above) is to use an old technique known as **bus snooping** [G83]; each cache pays attention to memory updates by observing the bus that connects them to main memory. When a CPU then sees an update for a data item it holds in its cache, it will notice the change and either **invalidate** its copy (i.e., remove it from its own cache) or **update** it (i.e., put the new value into its cache too). Write-back caches, as hinted at above, make this more complicated (because the write to main memory isn't visible until later), but you can imagine how the basic scheme might work.

10.2 Don't Forget Synchronization

Given that the caches do all of this work to provide coherence, do programs (or the OS itself) have to worry about anything when they access shared data? The answer, unfortunately, is yes, and is documented in great detail in the second piece of this book on the topic of concurrency. While we won't get into the details here, we'll sketch/review some of the basic ideas here (assuming you're familiar with concurrency).

When accessing (and in particular, updating) shared data items or structures across CPUs, mutual exclusion primitives (such as locks) should likely be used to guarantee correctness (other approaches, such as building **lock-free** data structures, are complex and only used on occasion; see the chapter on deadlock in the piece on concurrency for details). For example, assume we have a shared queue being accessed on multiple CPUs concurrently. Without locks, adding or removing elements from the queue concurrently will not work as expected, even with the underlying coherence protocols; one needs locks to atomically update the data structure to its new state.

To make this more concrete, imagine this code sequence, which is used to remove an element from a shared linked list, as we see in Figure 10.3. Imagine if threads on two CPUs enter this routine at the same time. If Thread 1 executes the first line, it will have the current value of `head` stored in its `tmp` variable; if Thread 2 then executes the first line as well, it also will have the same value of `head` stored in its own private `tmp` variable (`tmp` is allocated on the stack, and thus each thread will have its own private storage for it). Thus, instead of each thread removing an element from the head of the list, each thread will try to remove the

```
1   typedef struct __Node_t {
2       int              value;
3       struct __Node_t *next;
4   } Node_t;
5
6   int List_Pop() {
7       Node_t *tmp = head;         // remember old head ...
8       int value   = head->value;  // ... and its value
9       head        = head->next;   // advance head to next pointer
10      free(tmp);                  // free old head
11      return value;               // return value at head
12  }
```

Figure 10.3: **Simple List Delete Code**

same head element, leading to all sorts of problems (such as an attempted double free of the head element at line 4, as well as potentially returning the same data value twice).

The solution, of course, is to make such routines correct via **locking**. In this case, allocating a simple mutex (e.g., `pthread_mutex_t m;`) and then adding a `lock(&m)` at the beginning of the routine and an `unlock(&m)` at the end will solve the problem, ensuring that the code will execute as desired. Unfortunately, as we will see, such an approach is not without problems, in particular with regards to performance. Specifically, as the number of CPUs grows, access to a synchronized shared data structure becomes quite slow.

10.3 One Final Issue: Cache Affinity

One final issue arises in building a multiprocessor cache scheduler, known as **cache affinity** [TTG95]. This notion is simple: a process, when run on a particular CPU, builds up a fair bit of state in the caches (and TLBs) of the CPU. The next time the process runs, it is often advantageous to run it on the same CPU, as it will run faster if some of its state is already present in the caches on that CPU. If, instead, one runs a process on a different CPU each time, the performance of the process will be worse, as it will have to reload the state each time it runs (note it will run correctly on a different CPU thanks to the cache coherence protocols of the hardware). Thus, a multiprocessor scheduler should consider cache affinity when making its scheduling decisions, perhaps preferring to keep a process on the same CPU if at all possible.

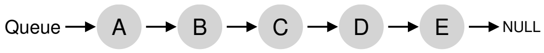
10.4 Single-Queue Scheduling

With this background in place, we now discuss how to build a scheduler for a multiprocessor system. The most basic approach is to simply reuse the basic framework for single processor scheduling, by putting all jobs that need to be scheduled into a single queue; we call this **single-queue multiprocessor scheduling** or **SQMS** for short. This approach has the advantage of simplicity; it does not require much work to take an existing policy that picks the best job to run next and adapt it to work on more than one CPU (where it might pick the best two jobs to run, if there are two CPUs, for example).

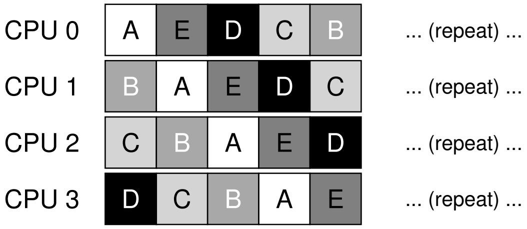
However, SQMS has obvious shortcomings. The first problem is a lack of **scalability**. To ensure the scheduler works correctly on multiple CPUs, the developers will have inserted some form of **locking** into the code, as described above. Locks ensure that when SQMS code accesses the single queue (say, to find the next job to run), the proper outcome arises.

Locks, unfortunately, can greatly reduce performance, particularly as the number of CPUs in the systems grows [A91]. As contention for such a single lock increases, the system spends more and more time in lock overhead and less time doing the work the system should be doing (note: it would be great to include a real measurement of this in here someday).

The second main problem with SQMS is cache affinity. For example, let us assume we have five jobs to run (A, B, C, D, E) and four processors. Our scheduling queue thus looks like this:



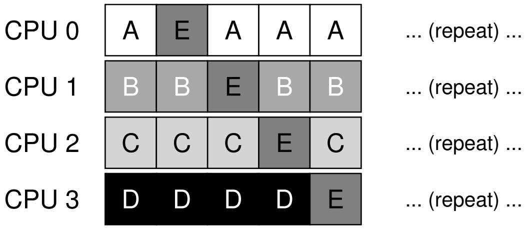
Over time, assuming each job runs for a time slice and then another job is chosen, here is a possible job schedule across CPUs:



CPU 0	A	E	D	C	B	... (repeat) ...
CPU 1	B	A	E	D	C	... (repeat) ...
CPU 2	C	B	A	E	D	... (repeat) ...
CPU 3	D	C	B	A	E	... (repeat) ...

Because each CPU simply picks the next job to run from the globally-shared queue, each job ends up bouncing around from CPU to CPU, thus doing exactly the opposite of what would make sense from the standpoint of cache affinity.

To handle this problem, most SQMS schedulers include some kind of affinity mechanism to try to make it more likely that process will continue to run on the same CPU if possible. Specifically, one might provide affinity for some jobs, but move others around to balance load. For example, imagine the same five jobs scheduled as follows:



CPU 0	A	E	A	A	A	... (repeat) ...
CPU 1	B	B	E	B	B	... (repeat) ...
CPU 2	C	C	C	E	C	... (repeat) ...
CPU 3	D	D	D	D	E	... (repeat) ...

In this arrangement, jobs A through D are not moved across processors, with only job E **migrating** from CPU to CPU, thus preserving affinity for most. You could then decide to migrate a different job the next time through, thus achieving some kind of affinity fairness as well. Implementing such a scheme, however, can be complex.

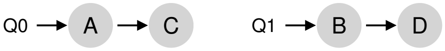
Thus, we can see the SQMS approach has its strengths and weaknesses. It is straightforward to implement given an existing single-CPU scheduler, which by definition has only a single queue. However, it does not scale well (due to synchronization overheads), and it does not readily preserve cache affinity.

10.5 Multi-Queue Scheduling

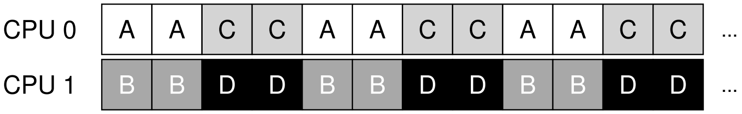
Because of the problems caused in single-queue schedulers, some systems opt for multiple queues, e.g., one per CPU. We call this approach **multi-queue multiprocessor scheduling** (or **MQMS**).

In MQMS, our basic scheduling framework consists of multiple scheduling queues. Each queue will likely follow a particular scheduling discipline, such as round robin, though of course any algorithm can be used. When a job enters the system, it is placed on exactly one scheduling queue, according to some heuristic (e.g., random, or picking one with fewer jobs than others). Then it is scheduled essentially independently, thus avoiding the problems of information sharing and synchronization found in the single-queue approach.

For example, assume we have a system where there are just two CPUs (labeled CPU 0 and CPU 1), and some number of jobs enter the system: A, B, C, and D for example. Given that each CPU has a scheduling queue now, the OS has to decide into which queue to place each job. It might do something like this:



Depending on the queue scheduling policy, each CPU now has two jobs to choose from when deciding what should run. For example, with **round robin**, the system might produce a schedule that looks like this:

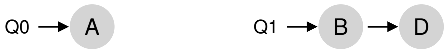


CPU 0	A	A	C	C	A	A	C	C	A	A	C	C	...
CPU 1	B	B	D	D	B	B	D	D	B	B	D	D	...

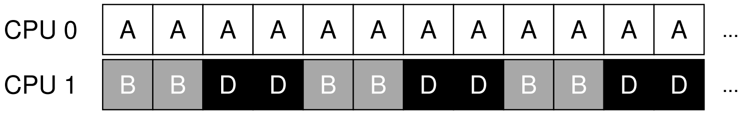
MQMS has a distinct advantage of SQMS in that it should be inherently more scalable. As the number of CPUs grows, so too does the number of queues, and thus lock and cache contention should not become a central problem. In addition, MQMS intrinsically provides cache affinity;

jobs stay on the same CPU and thus reap the advantage of reusing cached contents therein.

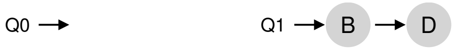
But, if you've been paying attention, you might see that we have a new problem, which is fundamental in the multi-queue based approach: **load imbalance**. Let's assume we have the same set up as above (four jobs, two CPUs), but then one of the jobs (say C) finishes. We now have the following scheduling queues:



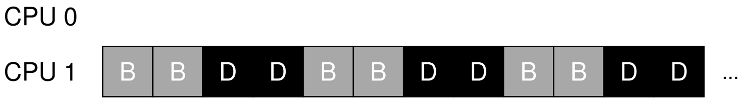
If we then run our round-robin policy on each queue of the system, we will see this resulting schedule:



As you can see from this diagram, A gets twice as much CPU as B and D, which is not the desired outcome. Even worse, let's imagine that both A and C finish, leaving just jobs B and D in the system. The scheduling queues will look like this:



As a result, CPU 0 will be left idle! *(insert dramatic and sinister music here)* And hence our CPU usage timeline looks sad:



So what should a poor multi-queue multiprocessor scheduler do? How can we overcome the insidious problem of load imbalance and defeat the evil forces of ... the Decepticons[1]? How do we stop asking questions that are hardly relevant to this otherwise wonderful book?

[1] Little known fact is that the home planet of Cybertron was destroyed by bad CPU scheduling decisions. And now let that be the first and last reference to Transformers in this book, for which we sincerely apologize.

CRUX: HOW TO DEAL WITH LOAD IMBALANCE

How should a multi-queue multiprocessor scheduler handle load imbalance, so as to better achieve its desired scheduling goals?

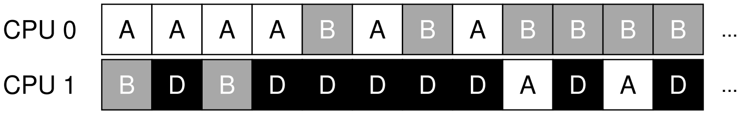
The obvious answer to this query is to move jobs around, a technique which we (once again) refer to as **migration**. By migrating a job from one CPU to another, true load balance can be achieved.

Let's look at a couple of examples to add some clarity. Once again, we have a situation where one CPU is idle and the other has some jobs.

In this case, the desired migration is easy to understand: the OS should simply move one of B or D to CPU 0. The result of this single job migration is evenly balanced load and everyone is happy.

A more tricky case arises in our earlier example, where A was left alone on CPU 0 and B and D were alternating on CPU 1:

In this case, a single migration does not solve the problem. What would you do in this case? The answer, alas, is continuous migration of one or more jobs. One possible solution is to keep switching jobs, as we see in the following timeline. In the figure, first A is alone on CPU 0, and B and D alternate on CPU 1. After a few time slices, B is moved to compete with A on CPU 0, while D enjoys a few time slices alone on CPU 1. And thus load is balanced:



CPU 0	A	A	A	A	B	A	B	A	B	B	B	B	...
CPU 1	B	D	B	D	D	D	D	D	A	D	A	D	...

Of course, many other possible migration patterns exist. But now for the tricky part: how should the system decide to enact such a migration?

One basic approach is to use a technique known as **work stealing** [FLR98]. With a work-stealing approach, a (source) queue that is low on jobs will occasionally peek at another (target) queue, to see how full it is. If the target queue is (notably) more full than the source queue, the source will "steal" one or more jobs from the target to help balance load.

Of course, there is a natural tension in such an approach. If you look around at other queues too often, you will suffer from high overhead and have trouble scaling, which was the entire purpose of implementing the multiple queue scheduling in the first place! If, on the other hand, you don't look at other queues very often, you are in danger of suffering from severe load imbalances. Finding the right threshold remains, as is common in system policy design, a black art.

10.6 Linux Multiprocessor Schedulers

Interestingly, in the Linux community, no common solution has approached to building a multiprocessor scheduler. Over time, three different schedulers arose: the O(1) scheduler, the Completely Fair Scheduler (CFS), and the BF Scheduler (BFS)[2]. See Meehean's dissertation for an excellent overview of the strengths and weaknesses of said schedulers [M11]; here we just summarize a few of the basics.

Both O(1) and CFS use multiple queues, whereas BFS uses a single queue, showing that both approaches can be successful. Of course, there are many other details which separate these schedulers. For example, the O(1) scheduler is a priority-based scheduler (similar to the MLFQ discussed before), changing a process's priority over time and then scheduling those with highest priority in order to meet various scheduling objectives; interactivity is a particular focus. CFS, in contrast, is a deterministic proportional-share approach (more like Stride scheduling, as discussed earlier). BFS, the only single-queue approach among the three, is also proportional-share, but based on a more complicated scheme known as Earliest Eligible Virtual Deadline First (EEVDF) [SA96]. Read more about these modern algorithms on your own; you should be able to understand how they work now!

10.7 Summary

We have seen various approaches to multiprocessor scheduling. The single-queue approach (SQMS) is rather straightforward to build and balances load well but inherently has difficulty with scaling to many processors and cache affinity. The multiple-queue approach (MQMS) scales better and handles cache affinity well, but has trouble with load imbalance and is more complicated. Whichever approach you take, there is no simple answer: building a general purpose scheduler remains a daunting task, as small code changes can lead to large behavioral differences. Only undertake such an exercise if you know exactly what you are doing, or, at least, are getting paid a large amount of money to do so.

[2] Look up what BF stands for on your own; be forewarned, it is not for the faint of heart.

References

[A90] "The Performance of Spin Lock Alternatives for Shared-Memory Multiprocessors" by Thomas E. Anderson. IEEE TPDS Volume 1:1, January 1990. *A classic paper on how different locking alternatives do and don't scale. By Tom Anderson, very well known researcher in both systems and networking. And author of a very fine OS textbook, we must say.*

[B+10] "An Analysis of Linux Scalability to Many Cores Abstract" by Silas Boyd-Wickizer, Austin T. Clements, Yandong Mao, Aleksey Pesterev, M. Frans Kaashoek, Robert Morris, Nickolai Zeldovich. OSDI '10, Vancouver, Canada, October 2010. *A terrific modern paper on the difficulties of scaling Linux to many cores.*

[CSG99] "Parallel Computer Architecture: A Hardware/Software Approach" by David E. Culler, Jaswinder Pal Singh, and Anoop Gupta. Morgan Kaufmann, 1999. *A treasure filled with details about parallel machines and algorithms. As Mark Hill humorously observes on the jacket, the book contains more information than most research papers.*

[FLR98] "The Implementation of the Cilk-5 Multithreaded Language" by Matteo Frigo, Charles E. Leiserson, Keith Randall. PLDI '98, Montreal, Canada, June 1998. *Cilk is a lightweight language and runtime for writing parallel programs, and an excellent example of the work-stealing paradigm.*

[G83] "Using Cache Memory To Reduce Processor-Memory Traffic" by James R. Goodman. ISCA '83, Stockholm, Sweden, June 1983. *The pioneering paper on how to use bus snooping, i.e., paying attention to requests you see on the bus, to build a cache coherence protocol. Goodman's research over many years at Wisconsin is full of cleverness, this being but one example.*

[M11] "Towards Transparent CPU Scheduling" by Joseph T. Meehean. Doctoral Dissertation at University of Wisconsin—Madison, 2011. *A dissertation that covers a lot of the details of how modern Linux multiprocessor scheduling works. Pretty awesome! But, as co-advisors of Joe's, we may be a bit biased here.*

[SHW11] "A Primer on Memory Consistency and Cache Coherence" by Daniel J. Sorin, Mark D. Hill, and David A. Wood. Synthesis Lectures in Computer Architecture. Morgan and Claypool Publishers, May 2011. *A definitive overview of memory consistency and multiprocessor caching. Required reading for anyone who likes to know way too much about a given topic.*

[SA96] "Earliest Eligible Virtual Deadline First: A Flexible and Accurate Mechanism for Proportional Share Resource Allocation" by Ion Stoica and Hussein Abdel-Wahab. Technical Report TR-95-22, Old Dominion University, 1996. *A tech report on this cool scheduling idea, from Ion Stoica, now a professor at U.C. Berkeley and world expert in networking, distributed systems, and many other things.*

[TTG95] "Evaluating the Performance of Cache-Affinity Scheduling in Shared-Memory Multiprocessors" by Josep Torrellas, Andrew Tucker, Anoop Gupta. Journal of Parallel and Distributed Computing, Volume 24:2, February 1995. *This is not the first paper on the topic, but it has citations to earlier work, and is a more readable and practical paper than some of the earlier queuing-based analysis papers.*

Homework (Simulation)

In this homework, we'll use `multi.py` to simulate a multi-processor CPU scheduler, and learn about some of its details. Read the related README for more information about the simulator and its options.

Questions

1. To start things off, let's learn how to use the simulator to study how to build an effective multi-processor scheduler. The first simulation will run just one job, which has a run-time of 30, and a working-set size of 200. Run this job (called job 'a' here) on one simulated CPU as follows: `./multi.py -n 1 -L a:30:200`. How long will it take to complete? Turn on the `-c` flag to see a final answer, and the `-t` flag to see a tick-by-tick trace of the job and how it is scheduled.
2. Now increase the cache size so as to make the job's working set (size=200) fit into the cache (which, by default, is size=100); for example, run `./multi.py -n 1 -L a:30:200 -M 300`. Can you predict how fast the job will run once it fits in cache? (hint: remember the key parameter of the warm_rate, which is set by the -r flag) Check your answer by running with the solve flag (`-c`) enabled.
3. One cool thing about `multi.py` is that you can see more detail about what is going on with different tracing flags. Run the same simulation as above, but this time with time_left tracing enabled (`-T`). This flag shows both the job that was scheduled on a CPU at each time step, as well as how much run-time that job has left after each tick has run. What do you notice about how that second column decreases?
4. Now add one more bit of tracing, to show the status of each CPU cache for each job, with the `-C` flag. For each job, each cache will either show a blank space (if the cache is cold for that job) or a 'w' (if the cache is warm for that job). At what point does the cache become warm for job 'a' in this simple example? What happens as you change the warmup_time parameter (`-w`) to lower or higher values than the default?
5. At this point, you should have a good idea of how the simulator works for a single job running on a single CPU. But hey, isn't this a multi-processor CPU scheduling chapter? Oh yeah! So let's start working with multiple jobs. Specifically, let's run the following three jobs on a two-CPU system (i.e., type `./multi.py -n 2 -L a:100:100,b:100:50,c:100:50`) Can you predict how long this will take, given a round-robin centralized scheduler? Use `-c` to see if you were right, and then dive down into details with `-t` to see a step-by-step and then `-C` to see whether caches got warmed effectively for these jobs. What do you notice?
6. Now we'll apply some explicit controls to study **cache affinity**, as described in the chapter. To do this, you'll need the `-A` flag. This flag can be used to limit which CPUs the scheduler can place a particular job upon. In this case, let's use it to place jobs 'b' and 'c' on CPU 1, while restricting 'a' to CPU 0. This magic is accomplished by typing this `./multi.py -n 2 -L a:100:100,b:100:50, c:100:50 -A a:0,b:1,c:1` ; don't forget to turn on various tracing options to see what is really happening! Can you

predict how fast this version will run? Why does it do better? Will other combinations of ’a’, ’b’, and ’c’ onto the two processors run faster or slower?

7. One interesting aspect of caching multiprocessors is the opportunity for better-than-expected speed up of jobs when using multiple CPUs (and their caches) as compared to running jobs on a single processor. Specifically, when you run on N CPUs, sometimes you can speed up by more than a factor of N, a situation entitled **super-linear speedup**. To experiment with this, use the job description here (`-L a:100:100,b:100:100,c:100:100`) with a small cache (`-M 50`) to create three jobs. Run this on systems with 1, 2, and 3 CPUs (`-n 1`, `-n 2`, `-n 3`). Now, do the same, but with a larger per-CPU cache of size 100. What do you notice about performance as the number of CPUs scales? Use `-c` to confirm your guesses, and other tracing flags to dive even deeper.
8. One other aspect of the simulator worth studying is the per-CPU scheduling option, the `-p` flag. Run with two CPUs again, and this three job configuration (`-L a:100:100,b:100:50,c:100:50`). How does this option do, as opposed to the hand-controlled affinity limits you put in place above? How does performance change as you alter the ’peek interval’ (`-P`) to lower or higher values? How does this per-CPU approach work as the number of CPUs scales?
9. Finally, feel free to just generate random workloads and see if you can predict their performance on different numbers of processors, cache sizes, and scheduling options. If you do this, you’ll soon be a **multi-processor scheduling master**, which is a pretty awesome thing to be. Good luck!