Overview

Here, we'll explore how to schedule jobs on multiple CPUs.

This section should help us answer the following questions:

- How do we schedule jobs on multiple CPUs?
- What new issues emerge?
- How should a multi-queue multiprocessor scheduler manage load balancing to better reach its scheduling goals?

Introduction

This chapter will cover **multiprocessor scheduling** basics.

Multiprocessor systems are present in desktops, laptops, and even mobile devices. This growth is due to the difficulty of making a single CPU faster without using excessive power.

Of course, having more than one CPU creates several issue. A typical application (i.e., a C program you wrote) only uses one CPU; adding more CPUs does not make it run faster.

To solve this, you'll need to redesign your software in **parallel**, potentially utilizing **threads**. When given more CPU resources, multi-threaded apps can split work over numerous CPUs and run faster.

So, **multiprocessor scheduling** becomes a new concern for the operating system.

We've explored some single-processor scheduling principles; how might we apply those ideas to multiple CPUs?

Multiprocessor Architecture: Caches

Let's explore the difference between single- and multiprocessor hardware which is mainly concerned with the use of **hardware caches** and how data is shared across processors.

A single CPU system has a hierarchy of **hardware caches** that let it run applications faster.

Caches are small, quick memories that hold copies of data found in the system's main memory.

The main memory, however, holds all data, but it is slow to access this larger memory.

Data that is frequently accessed can be cached to make a slow, large memory appear faster.

Imagine that your program issues an explicit load instruction to retrieve a value from memory, and that you have only one CPU with a small cache and a large system memory.

Because the data is stored in main memory, the first load takes a long time (perhaps in the tens or hundreds of nanoseconds).

In anticipation of reusing the data, the processor places a copy of the loaded data in the CPU cache.

Later, if the program needs this data item again, it checks the cache for it first; if it is present in the cache, the data is fetched much faster (for instance, in just a few nanoseconds), and so the program runs more quickly.

Caches are based on the idea of temporal(time) and spatial(space) locality.

Imagine variables or even instructions being accessed repeatedly in a loop.

The idea behind spatial locality is that if a program reads a data item at address x, it is likely to access data objects nearby x as well.

Multiprocessor Architecture: Example

Now, what happens when several processors share a single main memory? Caching with many CPUs is much more difficult.

Consider a program on CPU 1 reading a data item (with value D) at address A.

- It retrieves the value D from main memory because the data is not in the cache on CPU 1.
- The program then updates its cache with the new value D′ at address A.

Writing data to main memory is slow, so the system does it later. Assume the OS then decides to stop the program and move it to CPU 2.

- The program then reads the value at address A again.
- CPU 2's cache does not contain this data, so the system fetches it from main memory, which returns the old value D instead of D′.

This problem is called **cache coherence**. Hardware provides the basic solution. By monitoring memory accesses, hardware can verify that the "right thing" happens and the view of a single shared memory is maintained.

On a bus-based system one method is to use bus snooping, where each cache monitors the bus that connects them to main memory. When a CPU notices an update for a data item in its cache, it will either invalidate (i.e. remove it from its own cache) or update its copy (i.e., put the new value into its cache too).

Write-back caches complicate things (since the write to main memory isn't apparent until later), but you can see the general idea.

Synchronization

Do programs (or the OS) have to worry about accessing shared data when the caches do all the work to provide coherence?

Sadly, the answer is yes. We won't go into details here, but we'll review some basic notions. Other methods, including building **lock-free** data structures, are complex and only used on occasion.

Assume we have a shared queue accessed by multiple CPUs.

When there are no locks, adding or removing elements from the queue concurrently will not work as expected, even when there are coherence protocols in place. Locks are required to atomically update the data structure.

Imagine this code sequence to the left, which removes an element from a shared linked list, and two CPU threads entering this procedure at the same time.

1. Thread 1 will have the current value of `head` in its `tmp` variable if it performs the first line.
2. Thread 2 will have the same value of head in its own private `tmp` variable if it performs the first line (`tmp` is allocated on the stack, and thus each thread will have its own private storage for it)

Instead of deleting an element from the list's head, each thread will try to remove the same element, causing chaos. The remedy is to make these routines correct via **locking**.

We can address this issue by allocating a mutex (e.g., `pthread_mutex_tm;`) and using `lock(&m)` before the routine and `unlock(&m)` after it.

This strategy has disadvantages, especially in terms of performance. Access to a synchronized shared data structure gets quite slow as the number of CPUs grows.

Cache Affinity

Cache affinity is a major issue in multiprocessor cache scheduling.

A process running on a given CPU accumulates state in the caches (and TLBs) of the CPU. The process will run faster if some of its state is already in the caches on that CPU.

Instead, running a process on a different CPU each time slows it down since it has to reload its state (note it will run correctly on a different CPU thanks to the cache coherence protocols of the hardware).

So, a multiprocessor scheduler should consider cache affinity when scheduling, if possible keeping a process on the same CPU.

Single-Queue Scheduling

With this background, we can describe how to design a multiprocessor scheduler. The easiest approach is to reuse the basic structure for single processor scheduling by putting all jobs into a single queue, which we call **single-queue multiprocessor scheduling**, or **SQMS**.

It is easy to adapt an existing policy that picks the best job to run next to work on several CPUs. But SQMS has significant flaws.

Scalability

The first issue is **scalability**.

The developers will have used some type of locking in the code to ensure the scheduler operates correctly on multiple CPUs.

Locks make sure that once the SQMS code accesses the single queue (for example, to find the next job to run), the intended result occurs.

Locks, however, can also significantly slow down performance, especially as the number of CPUs increases. As competition for a single lock grows, the system spends more time in lock overhead and less time executing its job.

Cache Affinity

The second major SQMS flaw is **cache affinity**.

Suppose we have five jobs (A-E) and four processors. Here's our scheduling queue:

Here is an example job schedule across CPUs, assuming each job runs for a time slice before being replaced:

Because each CPU simply takes the next job from the globally shared queue, each job ends up bouncing from CPU to CPU, which is contrary to cache affinity.

Most SQMS schedulers contain an affinity mechanism to try to keep processes on the same CPU if possible. Affinity for some jobs may be given, but others may be moved to balance load.

Consider the following five jobs:

Jobs A through D are not shifted between processors, only job E, keeping affinity for most. You may then move a different job the next time, establishing some affinity fairness.

But implementing such a plan can be difficult. SQMS has both strengths and weaknesses. It is simple to implement given an existing single-CPU scheduler with only one queue.

Due to synchronization overheads, it does not scale well and does not easily maintain cache affinity.

Multi-Queue Scheduling

Due of the issues with single-queue schedulers, some systems use multiple queues, one per CPU. This is called **Multi-Queue Multiprocessor Scheduling** (or **MQMS**).

The scheduling framework in **MQMS** consists of multiple scheduling queues. Each queue will likely use a different scheduling technique, like round robin.

When a job is entered into the system, it is placed on exactly one scheduling queue according to some rule (e.g., random, or picking one with fewer jobs than others).

Then, each component is scheduled primarily independently, so there are no concerns with information sharing and synchronization.

Assume we have a system with only two CPUs (CPU 0 and CPU 1), and jobs A, B, C, and D join the system. Since each CPU has its own scheduling queue, the OS must select where to put each job.

It might do this:

Depending on the queue scheduling policy, each CPU now has two options for running jobs. For round robin, the machine might generate a schedule like this:

MQMS should be more scalable than SQMS. As the number of CPUs increases, so should the number of queues, so should lock and cache contention.

MQMS also supports cache affinity; jobs stay on the same CPU, allowing them to reuse cached items.

But, you may notice a new issue with the multi-queue approach: load imbalance.

Assume the identical setup (4 jobs, 2 CPUs), but one of the jobs (say C) completes. Now we have these scheduling queues:

This is what happens when we apply our round-robin policy to each queue:

A gets double the CPU as B and D, which isn't intended. Worse, imagine A and C are done, leaving only B and D. Here are the two scheduling queues and their timelines:

Sadly, CPU 0 is idle. As a result, our CPU consumption timeline looks bleak.

So, how should a multiqueue multiprocessor scheduler handle load imbalance to achieve better scheduling?

Handling Load Imbalance

The answer to this is to move jobs around, a technique which we call migration. True load balancing can be achieved by moving jobs between CPUs. Let's look at some examples to help clarify. Again, one CPU is idle while the other is busy.

In this situation, the OS should simply transfer one of B or D to CPU 0. This single work migration results in an even load and everyone is pleased. It gets more tricky, like in our previous example. A was left alone on CPU 0, while B and D alternated on CPU 1:

A single migration does not address the issue. What would you do? The answer is constant employment migration. One solution is to keep switching jobs. A is alone on CPU 0, while B and D alternate on CPU 1. After a few time slices, B joins A on CPU 0, while D gets some alone time on CPU 1. So load is balanced:

There are, of course, many additional options. However, how should the system decide to perform such a migration?

Work stealing is one basic method.

When work-stealing takes place, a queue that is low on jobs occasionally peeks into another queue, to determine how full it is.

It's not uncommon for the source to "steal" one or more jobs from a target queue to balance the load. Of course, such a strategy creates friction.

Overlooking other queues causes significant overhead and scaling issues, which was the whole point of introducing multiple queue scheduling in the first place!

Conversely, if you don't glance at other queues often, you risk significant load imbalances. Finding the proper threshold is a dark art in system policy design.

Summary

We have explored several ways to scheduling multiprocessors.

- Constructing a single-queue system (SQMS) is straightforward, but scaling to many processors and cache affinity is tough.
- The MQMS technique scales better and manages cache affinity effectively, but it is more complex.
- Whatever you do, constructing a general purpose scheduler is still a difficult undertaking, because small code changes might result in big differences in behavior.