Scribe Notes

Lecture #8

Jared Hall

3/6/12

Let's remember for a moment why we're here. We tried to use a spin-lock over multiple threads to increment a variable up to some large number. The single processor implementation took O(n) time when incrementing to n like we would expect. The spin-lock implementation *should* take the same amount of time, but it didn't. There is an exponential increase in time required to increment a variable using multiple threads and spin-locks. The question is: why is this?

Intuitively, there is no reason for this slow-down in performance. There could be some constant level of slow-down due to the spin lock using an if-statement and a couple reads and writes, but that doesn't explain the exponential slow-down we've witnessed. Somewhat randomly, let's examine the caching mechanism implemented by multiprocessor machines. In most cases, all of the processors have individual caches, but share a single bus that interfaces with other caches and memory farther away from the processors. In this scenario, when multiple processors want to do operations on the same 'logical' data(the same variable in the code) they need to have individual copies of the variable in their respective caches. An example would be two processors both executing an addition instruction on the same variable. In the code, the variable is the same, but the processors need their own copy to operate on. The problem is keeping these cached variables consistent with other caches and the main memory.

A single processor would simply store the variable in its cache, and read/write to it freely. When there are multiple caches, cache consistency is a problem. If the same logical variable is in two or more physical locations, the physical locations must contain the same data. What is happening in the multiprocessor incrementing program is the same logical variable having different values in different caches after being operated on by different processors. This incurs a penalty imposed by the hardware. The hardware has to go through all of the caches and update the variable. This variable could be the lock variable itself or the variable being incremented, it doesn't matter. The same result will occur, forcing a slow-down as the caches have to maintain their consistency. Needing to keep the cache consistent among processors essentially serializes parallel processing.

There are various solutions to this problem. One is to use the Test-and-Test-and-Set lock. This algorithm is slightly better, because it doesn't try to write to memory so viciously as a normal Test-and-Set, but it isn't much better than the normal spin-lock using Test-and-Set.

Another solution is to try and introduce an exponentially growing delay every time a call to Test-and-Set fails. For instance, after the first spin fails, you wait 1ms before trying again. Then you wait 2ms, then 4ms, then 8ms, until finally you get the lock. This helps to considerably speed up the lock, but still leaves us with a worse than linear time complexity for the lock. The reason it works is because there is less cache validation needed as less requests for the lock are made.

The final solution is to use a queue that wakes up threads when it is their turn to run. In an Anderson Queue Lock, a large array of keys is maintained by the lock master. Every thread that requests a lock, receives a location in the queue, which is their 'key'. The key is simply a variable that is either true or false. When it is true, the thread can run, and when it is false, the thread should be blocked. You can imagine the queue lock as a bunch of single locks all maintained in a row. Each thread gets one of the locks in the row. Only one lock may be activated at any given time, so that mutual exclusion is maintained. The reason that this speeds up the algorithm is that there is no cache contention for the lock anymore. Every thread has its own lock that is maintained by the 'master'. When the master says your lock is active, you can run your code.