Computer Architecture III Study Guide Synchronization

Maximillien Courchesne-Mackie (mcour010@uottawa.ca)

I. Synchronization Primitives

In a shared system, physical resources can be accesses by several different threads or processes at any given time. However, control can only be given to one agent at once, while the others wait. This is the problem we solve with synchronization.

A. Locks

The most basic synchronization problem is that of locking. For example, consider two threads A and B which both want to increment a shared variable S. Given that they run on a parallel multiprocessor system, each thread will execute independently. The expected outcome of running both threads would be to see S incremented by 2. Let's say thread A loads S from memory, thread B then also loads S from memory. Then both threads increment S and store it back into memory. The result is the shared variable S only being incremented by 1 because both threads accessed the data from memory at the same time.

To get the intended result, there needs to be a way to "lock" the segments of code such that only one process can execute at a time when dealing with shared resources. Several threads competing for the same resource must wait until the resource is marked as unlocked before marking it as locked again and using it. The part of the code which is executed by a thread under a lock is called a *critical section*. The method of locking described above has some pitfalls. Multiple threads can deadlock if they all lock a resource at the same time and adding lock commands to code makes it much more convoluted. For these reasons, locking is usually implemented at the hardware level with atomic RMW (read-modify-write) instructions, dedicated bus lines or synchronization registers.

B. Barriers

A barrier is a synchronization protocol which involves a group of threads all reaching a certain point in execution before proceeding with the rest of their code. Each thread must hit this "barrier" before being able to continue. This is implemented using a shared variable (called the barrier count) which gets incremented by each process as it hits the barrier. Once the barrier count is equal to the amount of processes in the group, execution continues.

C. Producer/Consumer Synchronization

Another synchronization protocol allows one thread to communicate with another thread, usually signalling that it has reached a certain point of execution. This can be done by storing a flag in memory which the signalling thread (the producer) changes the value to alert the reading thread (the consumer). Seeing as only the producer has the right to write to the flag, the code doesn't need a critical section. However, this solution relies on the memory consistency model.

II. Hardware Synchronization

Locks and barriers can be implemented in hardware using dedicated bus lines and flip flops. However, there are pitfalls to implementing a synchronization protocol in hardware. These include poor scalability (more resources are needed which increase the bandwidth strain on the network), limited flexibility (if there are not enough physical resources to fulfil the synchronization needs of the threads, certain processes must block until resources become available) and complexity (multi-threaded processors require synchronization resources for each thread of each core).

III. Software Synchronization

Software-based synchronization is implemented using a different class of memory instructions known as RMW. RMW instructions allow reliable and complex implementations of locks through shared memory.

A. Test_and_set

Test_and_set (T&S) is a RMW instruction which solves the locking problem. It is a simple command which probes a location in memory, tries to set it to 1 and returns its value. If it returns a 0, the lock is successful and the program can continue executing its critical section. If it returns a 1 then another process has an active lock.

If T&S is executed in memory, it bypasses all caches and the memory controller updates the store. If it is executed in cache, an invalidation cache coherence protocol must be used to ensure other processes obey the lock. Executing the lock in memory requires a lot of traffic across the network, which is why it is normally done in cache. To reduce the cache-related traffic, programs an exponential back-off technique if they cannot get a lock. This means that the delay until the next T&S attempt increases exponentially.

A common technique to keep traffic at a minimum is to use a test_and_test&set lock. This involves a loop with regular loads hitting the shared variable. If it returns 0, then a regular T&S is attempted to gain a lock. If this fails, the regular load loop is resumed.

B. SWAP

The SWAP RMW instruction is similar to T&S but instead of writing a value to the shared memory, the contents are swapped atomically with the register value.

C. Compare and Swap (CAS)

This is similar to SWAP, but in this case the SWAP is only done if a condition is met. CAS is often used for atomic insertions and removals from queues.

D. Fetch_and_op (F&OP)

This operation returns the value from memory and immediately applies an operation to it. The most common form is F&ADD.

E. Non-RMW Instructions

The complexity of RMW instructions make them poorly suited for RISC-based machines. However, a similar can be achieved by separating a T&S instruction into two different atomic instructions (verified by hardware). This is known as *load lock* (LL) and *store conditional* (SC). First, an LL instruction is used to load a value of shared memory into a register. Then, an SC instruction will update the memory only if the value was not accessed since the LL.

IV. Waiting Methods

When multiple threads attempt to get a lock on the same resource at once, one will inevitably have to wait for the other(s) to finish. A waiting thread can do one of two things.

A. Busy Waiting

A busy waiting thread will continuously test the lock to see if it is available. This consumes a lot of bandwidth and can sometimes cause the thread to have its priority lowered by the system.

B. Blocking

A blocking thread will be de-scheduled and placed in a waiting queue for its turn with the lock. This frees the processor to work with other threads.