Synchronization

Synchronization mechanisms are typically built with user-level software routines that rely on hardware-supplied synchronization instructions. For smaller multiprocessors or low-contention situations, the key hardware capability is an uninterruptible instruction or instruction sequence capable of atomically retrieving and changing a value. Software synchronization mechanisms are then constructed using this capability. For example, we will see how very efficient spin locks can be built using a simple hardware synchronization instruction and the coherence mechanism. In larger-scale multiprocessors or high-contention situations, synchronization can become a performance bottleneck, because contention introduces additional delays and because latency is potentially greater in such a multiprocessor.

We will see how contention can arise in implementing some common user-level synchronization operations and examine more powerful hardware-supported synchronization primitives that can reduce contention as well as latency. We begin by examining the basic hardware primitives, then construct several well-known synchronization routines with the primitives, and then turn to performance problems in larger multiprocessors and solutions for those problems.

Basic Hardware Primitives

The key ability we require to implement synchronization in a multiprocessor is a set of hardware primitives with the ability to atomically read and modify a memory location. Without such a capability, the cost of building basic synchronization primitives will be too high and will increase as the processor count increases. There are a number of alternative formulations of the basic hardware primitives, all of which provide the ability to atomically read and modify a location, together with some way to tell if the read and write were performed atomically. These hardware primitives are the basic building blocks that are used to build a wide variety of user-level synchronization operations, including things such as locks and barriers. In general, architects do not expect users to employ the basic hardware primitives, but instead expect that the primitives will be used by system programmers to build a synchronization library, a process that is often complex and tricky. Let's start with one such hardware primitive and show how it can be used to build some basic synchronization operations.

One typical operation for building synchronization operations is the ***atomic exchange***, which interchanges a value in a register for a value in memory. To see how to use this to build a basic synchronization operation, assume that we want to build a simple lock where the value 0 is used to indicate that

the lock is free and a 1 is used to indicate that the lock is unavailable. A processor tries to set the lock by doing an exchange of 1, which is in a register, with the memory address corresponding to the lock.The value returned from the exchange instruction is 1 if some other processor had already claimed access and 0 otherwise. In the latter case, the value is also changed to be 1, preventing any competing exchange from also retrieving a 0.

For example, consider two processors that each try to do the exchange simultaneously: This race is broken since exactly one of the processors will perform the exchange first, returning 0, and the second processor will return 1 when it does the exchange. The key to using the exchange (or swap) primitive to implement synchronization is that the operation is atomic: the exchange is indivisible and two simultaneous exchanges will be ordered by the write serialization mechanisms. It is impossible for two processors trying to set the synchronization variable in this manner to both think they have simultaneously set the variable. There are a number of other atomic primitives that can be used to implement synchronization. They all have the key property that they read and update a memory value in such a manner that we can tell whether or not the two operations executed atomically. One operation, present in many older multiprocessors, is ***test-and- set***, which tests a value and sets it if the value passes the test. For example, we could define an operation that tested for 0 and set the value to 1, which can be used in a fashion similar to how we used atomic exchange. Another atomic synchronization primitive is ***fetch-and-increment***: it returns the value of a memory location and atomically increments it. By using the value 0 to indicate that the synchronization variable is unclaimed, we can use fetch-and-increment, just as we used exchange. There are other uses of operations like fetch-and-increment.

An alternative is to have a pair of instructions where the second instruction returns a value from which it can be deduced whether the pair of instructions was executed as if the instructions were atomic. The pair of instructions is effectively atomic if it appears as if all other operations executed by any processor occurred before or after the pair. Thus, when an instruction pair is effectively atomic, no other processor can change the value between the instruction pair. The pair of instructions includes a special load called a ***load linked*** **or** ***loadlocked*** and a special store called a ***store conditional***. These instructions are used in sequence: If the contents of the memory location specified by the load linked are changed before the store conditional to the same address occurs, then the store conditional fails. If the processor does a context switch between the two instructions, then the store conditional also

fails. The store conditional is defined to return a value indicating whether or not the store was successful. Since the load linked returns the initial value and the store conditional returns 1 if it succeeds and 0 otherwise, the following sequence implements an atomic exchange on the memory location specified by the contents of R1:

```
try:    MOV R3,R4,R0 ;mov exchange value
        LL R2,0(R1) ;load linked
        SC R3,0(R1) ;store conditional
        BEQZ R3,try ;branch store fails
        MOV R4,R2 ;put load value in R4
```

At the end of this sequence the contents of R4 and the memory location specified by R1 have been atomically exchanged (ignoring any effect from delayed branches). Any time a processor intervenes and modifies the value in memory between the LL and SC instructions, the SC returns 0 in R3, causing the code sequence to try again.

An advantage of the load linked/store conditional mechanism is that it can be used to build other synchronization primitives. For example, here is an atomic fetch-and-increment:

```
try:    LL R2,0(R1) ;load linked
        DADDUI R3,R2,#1 ;increment
        SC R3,0(R1) ;store conditional
        BEQZ R3,try ;branch store fails
```

These instructions are typically implemented by keeping track of the address specified in the LL instruction in a register, often called the *link register*. If an interrupt occurs, or if the cache block matching the address in the link register is invalidated (for example, by another SC), the link register is cleared. The SC instruction simply checks that its address matches that in the link register; if so, the SC succeeds; otherwise, it fails. Since the store conditional will fail after either another attempted store to the load linked address or any exception, care must be taken in choosing what instructions are inserted between the two instructions. In particular, only register-register instructions can safely be permitted; otherwise, it is possible to create deadlock situations where the processor can never complete the SC. In addition, the number of instructions between the load linked and the store conditional should be small to minimize the probability that either an unrelated event or a competing processor causes the store conditional to fail frequently.

Implementing Locks Using Coherence

Once we have an atomic operation, we can use the coherence mechanisms of a multiprocessor to implement *spin locks*: locks that a processor continuously tries to acquire, spinning around a loop until it suceeds. Spin locks are used when a programmer expects the lock to be held for a very short amount of time and when she wants the process of locking to be low latency when the lock is available. Because spin locks tie up the processor, waiting in a loop for the lock to become free, they are inappropriate in some circumstances. The simplest implementation, which we would use if there were no cache coherence, would keep the lock variables in memory. A processor could continually try to acquire the lock using an atomic operation, say exchange, and test whether the exchange returned the lock as free. To release the lock, the processor simply stores the value 0 to the lock. Here is the code sequence to lock a spin lock whose address is in R1 using an atomic exchange:

```
        DADDUI R2,R0,#1
lockit: EXCH R2,0(R1) ;atomic exchange
        BNEZ R2,lockit ;already locked?
```

If our multiprocessor supports cache coherence, we can cache the locks using the coherence mechanism to maintain the lock value coherently. Caching locks has two advantages. First, it allows an implementation where the process of "spinning" (trying to test and acquire the lock in a tight loop) could be done on a local cached copy rather than requiring a global memory access on each attempt to acquire the lock. The second advantage comes from the observation that there is often locality in lock accesses: that is, the processor that used the lock last will use it again in the near future. In such cases, the lock value may reside in the cache of that processor, greatly reducing the time to acquire the lock. Obtaining the first advantage—being able to spin on a local cached copy rather than generating a memory request for each attempt to acquire the lock—requires a change in our simple spin procedure. Each attempt to exchange in the loop directly above requires a write operation. If multiple processors are attempting to get the lock, each will generate the write. Most of these writes will lead to write misses, since each processor is trying to obtain the lock variable in an exclusive state.

Thus we should modify our spin-lock procedure so that it spins by doing reads on a local copy of the lock until it successfully sees that the lock is available. Then it attempts to acquire the lock by doing a swap operation. A processor first reads the lock variable to test its state. A processor keeps reading and testing until the value of the read indicates that the lock is

unlocked. The processor then races against all other processes that were similarly "spin waiting" to see who can lock the variable first. All processes use a swap instruction that reads the old value and stores a 1 into the lock variable. The single winner will see the 0, and the losers will see a 1 that was placed there by the winner. (The losers will continue to set the variable to the locked value, but that doesn't matter.) The winning processor executes the code after the lock and, when finished, stores a 0 into the lock variable to release the lock, which starts the race all over again. Here is the code to perform this spin lock (remember that 0 is unlocked and 1 is locked):

```
lockit:  LD R2,0(R1) ;load of lock
         BNEZ R2,lockit ;not available-spin
         DADDUI R2,R0,#1 ;load locked value
         EXCH R2,0(R1) ;swap
         BNEZ R2,lockit ;branch if lock wasn't 0
```

Let's examine how this "spin-lock" scheme uses the cache-coherence mechanisms. Once the processor with the lock stores a 0 into the lock, all other caches are invalidated and must fetch the new value to update their copy of the lock. One such cache gets the copy of the unlocked value (0) first and performs the swap. When the cache miss of other processors is satisfied, they find that the variable is already locked, so they must return to testing and spinning.