2.10 Parallelism and instructions: synchronization

Parallel execution is easier when tasks are independent, but often they need to cooperate. Cooperation usually means some tasks are writing new values that others must read. To know when a task is finished writing so that it is safe for another to read, the tasks need to synchronize. If they don't synchronize, there is a danger of a *data race*, where the results of the program can change depending on how events happen to occur.

Data race: Two memory accesses form a data race if they are from different threads to same location, at least one is a write, and they occur one after another.

For example, recall the analogy of the eight reporters writing in COD Chapter 1 (Computer Abstractions and Technology). Suppose one reporter needs to read all the prior sections before writing a conclusion. Hence, he or she must know when the other reporters have finished their sections, so that there is no danger of sections being changed afterwards. That is, they had better synchronize the writing and reading of each section so that the conclusion will be consistent with what is printed in the prior sections.

In computing, synchronization mechanisms are typically built with user-level software routines that rely on hardware-supplied synchronization instructions. In this section, we focus on the implementation of *lock* and *unlock* synchronization operations. Lock and unlock can be used straightforwardly to create regions where only a single processor can operate, called a *mutual exclusion*, as well as to implement more complex synchronization mechanisms.

The critical 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. That is, nothing else can interpose itself between the read and the write of the memory location. Without such a capability, the cost of building basic synchronization primitives will be high and will increase unreasonably 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. In general, architects do not expect users to employ the basic hardware primitives, but instead expect system programmers will use the primitives 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 a basic synchronization primitive. One typical operation for building synchronization operations is the *atomic exchange* or *atomic swap*, which interchanges a value in a register for a value in memory.

To see how to use this to build a basic synchronization primitive, assume that we want to build a simple lock where the value 0 is used to indicate that the lock is free and 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 1, preventing any competing exchange in another processor from also retrieving a 0.

For example, consider two processors that each try to do the exchange simultaneously: this race is prevented, 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 primitive to implement synchronization is that the operation is atomic: the exchange is indivisible, and two simultaneous exchanges will be ordered by the hardware. It is impossible for two processors trying to set the synchronization variable in this manner to both think they have simultaneously set the variable.

Implementing a single atomic memory operation introduces some challenges in the design of the processor, since it requires both a memory read and a write in a single, uninterruptible instruction.

An alternative is to have a pair of instructions in which the second instruction returns a value showing whether the pair of instructions was executed as if the pair was 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 pair of instructions.

In LEGv8 this pair of instructions includes a special load called a *load exclusive register* (`LDXR`) and a special store called a *store exclusive register* (`STXR`). These instructions are used in sequence: if the contents of the memory location specified by the load exclusive are changed before the store exclusive to the same address occurs, then the store exclusive fails and does not write the value to memory. The store exclusive is defined to both store the value of a (presumably different) register in memory *and* to change the value of another register to a 0 if it succeeds and to a 1 if it fails. Thus, `STXR` specifies three registers: one to hold the address, one to indicate whether the atomic operation failed or succeeded, and one to hold the value to be stored in memory if it succeeded. Since the load exclusive returns the initial value, and the store exclusive returns 0 only if it succeeds, the following sequence implements an atomic exchange on the memory location specified by the contents of `X20`:

```
again: LDXR X10,[X20,#0]    // load exclusive
       STXR X23, X9, [X20]  // store exclusive
       CBNZ X9, again       // branch if store fails
       ADD  X23, XZR, X10   // put loaded value in X23
```

Any time a processor intervenes and modifies the value in memory between the `LDXR` and `STXR` instructions, the `STXR` returns 1 in `X9`, causing the code sequence to try again. At the end of this sequence, the contents of `X23` and the memory location specified by `X20` have been atomically exchanged.

Elaboration

Although it was presented for multiprocessor synchronization, atomic exchange is also useful for the operating system in dealing with multiple processes in a single processor. To make sure nothing interferes in a single processor, the store exclusive also fails if the processor does a context switch between the two instructions (see COD Chapter 5 (Large and Fast: Exploiting Memory Hierarchy)).

Elaboration

An advantage of the load/store exclusive mechanism is that it can be used to build other synchronization primitives, such as atomic compare and swap or atomic fetch-and-increment, which are used in some parallel programming models. These involve more instructions between the `LDXR` and the `STXR`, but not too many.

Since the store exclusive will fail after either another attempted store to the load exclusive address or any exception, care must be taken in choosing which 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 ***`STXR`*** *because of repeated page faults. In addition, the number of instructions between the load exclusive and the store exclusive should be small to minimize the probability that either an unrelated event or a competing processor causes the store exclusive to fail frequently.*

Elaboration

While the code above implemented an atomic exchange, the following code would more efficiently acquire a lock at the location in register ***`X20`****, where the value of 0 means the lock was free and 1 to mean lock was acquired:*

```
       ADDI  X11, XZR, #1    // copy locked value
again: LDXR  X10,[X20,#0]    // load exclusive to read lock
       CBNZ  X10, again      // check if it is 0 yet
       STXR  X11, X9, [X20]  // attempt to store new value
       BNEZ  X9, again       // branch if store fails
```

We release the lock just using a regular store to `write 0` *into the location:*

```
       STUR XZR, [X20,#0]    // free lock by writing 0
```

PARTICIPATION ACTIVITY 2.10.1: Check yourself: Synchronization.

When do you use primitives like load exclusive and store exclusive?

1) When cooperating threads of a parallel program need to synchronize to get proper behavior for reading and writing shared data.

- Yes
- No

2) When cooperating processes on a uniprocessor need to synchronize for reading and writing shared data.

- Yes
- No

Provide feedback on this section