Hardware Support for Mutual Exclusion

Prof. Naga Kandasamy
ECE Department, Drexel University

We discuss major hardware-based approaches to achieving mutual exclusion.

Interrupt Disabling

In a uniprocessor machine, concurrent processes cannot be overlapped, that is, no two processes can run concurrently. Processes can only be interleaved. Moreover, a process will execute until its time quanta runs out or it is interrupted from the CPU. So, to guarantee mutual exclusion, one can simply prevent a process from being preempted from the CPU by disabling all interrupts. A process can enforce mutual exclusion in the following way.

```
while (1) {
   /* Disable interrupts */

   /* Critical section */

   /* Enable interrupts */

   /* Remainder of code */
}
```

The above approach will not work in a multiprocessor architecture, since if we have more than one processor, multiple processes may be executing at the same time. In this case, disabling the interrupts on one processor does not guarantee mutual exclusion.

Instruction Set Architecture (ISA) Support

Modern processor architectures provide *atomic* instructions as part of the ISA to support mutual exclusion in which an instruction, once started, cannot be interrupted. We will now discuss three commonly implemented instructions.

Atomic Compare and Swap

The definition of the atomic compare and swap function typically is as follows:

```
int atomicCAS(int *mutex, int compareVal, int newVal)
{
   int oldVal = *mutex;

   if (*mutex == compareVal)
      *mutex = newVal;

   return oldVal;
}
```

The above function takes as input the pointer to the mutex variable and compares the value stored at that location—call it *oldVal*—with *compareVal*. If the comparison succeeds, the mutex's value is replaced with *newVal*. The function returns *oldVal* back to the caller. Assume that *mutex* is a shared variable between multiple processes, which is initialized to 0. Each process in the system can use *atomicCAS* to achieve mutual exclusion.

```
int mutex = 0;
while (1) {
   while (atomicCAS(&mutex, 0, 1) != 0);

   /* Critical Section */

   mutex = 0;    /* Reset the mutex */

   /* Remainder of the code */
}
```

Test and Set Instruction

The functionality of the test and set instruction is defined as follows.

```
int testSet (int *mutex)
{
   if (*mutex == 0) {
      *mutex = 1;
      return 1;
   }

   return 0;
}
```

The instruction tests the value of *mutex* which is initialized to 0. If the value is 0, then the instruction sets *mutex* to 1 and returns true. Otherwise, the value is not changed and false is returned. Again, note that the entire *testSet* function is carried out atomically; if two processes simultaneously attempt to execute the instruction, only one succeeds. The code snippet below shows how

mutual exclusion can be implemented by a process within the function *foo()* using the *testSet* function. The shared variable *mutex* is first initialized to 0. If multiple processes try to execute the *testSet* instruction, then the only process that may enter the critical section is the one that finds *mutex* equal to 0. The other processes attempting to enter their critical section will go into a busy-waiting (also called spin-waiting) mode, by spinning on the *while* loop. When the process, currently in its critical section, leaves the critical section, it resets *mutex* to 0. Other processes are now free to compete for *mutex*. The winner is the process that happens to execute the *testSet* instruction next.

```
int mutex = 0;     /* This is the shared lock variable */
void foo(void)
{
   while (1) {
      while (testSet(&mutex) == 0);

      /* Critical section */

      mutex = 0;  /* Reset the mutex */

      /* Remainder of the code */
   }
}
```

Exchange Instruction

Like a *testSet*, the `exchange` instruction reads the value from the specified memory location into the specified register, but instead of writing a fixed constant into the memory location, it writes whatever value, was in the register to begin with. That is, it atomically exchanges or swaps the value in the memory location and the register. We can implement a lock as before by replacing the *testSet* with an *exchange* instruction as long as we use the values 0 and 1 and ensure that the value in the register is 1 before the swap instruction is executed; the lock has succeeded if the value left in the register by the swap instruction is 0. The functionality of the exchange instruction is defined as follows.

```
void exchange(int *mutex, int registerVal)
{
   int temp;
   temp = *mutex;
   *mutex = registerVal;
   registerVal = temp;
}
```

The following code shows how mutual exclusion is implemented with the *exchange* instruction.

```
int mutex = 0;     /* This is the shared lock variable */
void foo(void)
{
   int key = 1;
   while (1){
      while (key != 0)
         exchange (&mutex, key);

      /* Critical section */

      exchange (&mutex, key);  /* Reset the mutex */

      /* Remainder of the code */
   }
}
```

A shared variable *mutex* is initialized to 0. Each process uses a local variable called *key* that is initialized to 1. The first process that executes *exchange* finds *mutex* equal to 0 and sets *mutex* to 1, thereby preventing any other process from entering the critical section. When it leaves the critical section, it resets *mutex* to 0, allowing another process to gain access to the critical section. The Intel architecture supports an exchange instruction in the ISA called `XCHG`.

Implementing Atomic Instructions in the ISA

We return to the question of how to implement the above-described operations in atomic fashion, at the ISA level. Consider the *exchange* operation for example. Implementing a single atomic memory operation introduces some challenges, since it requires both a memory read and a write in a single, uninterruptible instruction. This requirement complicates the implementation of coherence, since the hardware cannot allow any other operations between the read and the write, and yet must not deadlock.

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 *load locked* 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 1 if it was successful and a 0 otherwise. Since the load linked returns the initial value and the store conditional returns 1 only if it succeeds, the following sequence implements an atomic exchange on the memory location specified by the contents of `R1`:

```
try:    MOV  R3, R4      ; 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. Anytime 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.

The `LL` and `SC` instructions work very much like their simple counterparts load and store. The `LL` instruction, in addition to doing a simple load, has the side effect of setting a user transparent bit called the load link bit or LLbit. The LLbit forms a breakable link between the `LL` instruction and a subsequent `SC` instruction. The `SC` performs a simple store if and only if the LLbit is set when the store is executed. If the LLbit is not set, then the store will fail to execute. The success or failure of the `SC` is indicated in the target register of the store after the execution of the instruction. The target register is loaded with 1 in case of a successful store or it is loaded with 0 if the store was unsuccessful. The LLbit is reset upon occurrence of any event that even has potential to modify the lock-variable (like semaphore or counter) while the sequence of code between `LL` and `SC` is being executed. The most obvious case where the link will be broken is when an invalidate occurs to the cache line which was the subject of the load. In this case, some other processor successfully completed a store to that shared line.

In general, the link will be broken if following events occur while the sequence of code between `LL` and `SC` is being executed:

1. External update to the cache line containing the lock variable.
2. External invalidate to the cache line containing the lock variable.
3. Any snooped signal invalidating cache the line containing the lock variable.
4. Upon return from an exception.

More information on how the `LL` and `SC` instructions are implemented can be found in the following document: `https://www.cs.auckland.ac.nz/courses/compsci313s2c/resources/MIPSLLSC.pdf`.

The `LL/SC` pair can be used to build other synchronization primitives such as an atomic *fetch-and-increment*.

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

Finally, 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 succeeds. Spin locks are used when programmers expect the lock to be held for

a very short amount of time and when they want the process of locking to be low latency when the lock is available.[1] The simplest implementation, which we would use if there were no cache coherence, would be to keep the lock variables in memory. A processor could continually try to acquire the lock using an atomic operation, say the atomic exchange that was just discussed, 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:

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

If the multiprocessor supports cache coherence, we can cache the locks using the coherence mechanism to maintain the lock value coherently. Caching locks has two advantages:

1. It allows an implementation where the process of spinning (that is, 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.

2. There is often locality in lock accesses; 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.

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 (note that 0 is unlocked and 1 is locked):

```
lockit: LW   R2, 0(R1)  ;load lock
        BNEZ R2, lockit ;not available; spin
        ADDI R2, R0, #1 ;load locked value
        XCHG R2, 0(R1)  ;swap
        BNEZ R2, lockit ;branch if lock wasn't 0
```

[1] Because spin locks tie up the processor, waiting in a loop for the lock to become free, they are inappropriate in some circumstances.

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.

The following assembly code shows how the `testSet` behavior can be implemented in an atomic fashion using the `LL` and `SC` instructions. Here the unlocked mutex has its LSB set to 0 whereas the locked mutex has its LSB set to 1. Initially, the mutex is unlocked.

```
; Load mutex into R2; set LLbit in the cache block having 0(R1)
Loop:     LL    R2, 0(R1)
          ORI   R3, R2, #1     ; Test mutex
          BEQ   R3, R2, Loop   ; Mutex is set. Try again.
          NOP                  ; Delay slot

          ; Store new mutex value to 0(R1) predicated on the LLbit
          ; Cache lines on other processors are invalidated if
          ; SC is successful
          SC    R3, 0(R1)
          BEQ   R3, #0, Loop   ; SC failed. Try again
          NOP

          ; Critical section


          ; Restore the mutex
          ; Invalidate cache blocks on other processors
          SW   R2, 0(R1)
```

Advantages and Disadvantages of Using Special Instructions

Using special machine instructions to enforce mutual exclusion has the following advantages:

- It is applicable to any number of processes on either a single processor or multi-processor systems sharing main memory.
- It can be used to support multiple critical sections, where each critical section can be protected by its own `mutex` variable.

There are some disadvantages:

- When a process is waiting for access to a critical section, it continues to spin on the while loop, and therefore, consumes processor cycles doing nothing.
- When a process exits the critical section, the selection of the next process to enter the critical section is arbitrary, and depends on the scheduler. So, some process could be denied access to the critical section indefinitely.