Consistency Models for Multiprocessor systems

Prof. Naga Kandasamy
ECE Department, Drexel University

The following notes were derived from:

- A. S. Tanenbaum, *Distributed Operating Systems*, Prentice Hall, Upper Saddle River, NJ, 1995, pp. 315-333.
- J. Hennessy and D. Patterson, *Computer Architecture: A Quantitative Approach*, 5^{th} Edition, Morgan Kaufmann, 2011.

A *consistency model* is a contract between the software and memory, stating that if the software agrees to obey certain rules when operating on shared variables, the memory promises to work correctly. If not, the correctness of memory operations is no longer guaranteed. We will now discuss consistency models for a multi-processor system.

Strict Consistency

The most stringent consistency model is called *strict consistency*, defined by the following condition: Any read to a memory location x returns the value stored by the most recent write operation to x. Though uniprocessors have traditionally observed strict consistency, the matter is more complicated in a shared-memory system. Let us denote a processor in a multi-processor system as P_i, and let $W(x)\ a$ and $R(y)\ b$ mean that a write to memory location x with the value a and a read from y returning b have been performed. The initial value of all variables in the following diagrams is assumed to be 0.

In the scenario sketched out below, processor P_1 does a write to location x, storing the value 1. Later, P_2 reads x and sees 1. This behavior is correct for a strictly consistent memory.

P_1	W(x) 1	
P_2		R(x) 1

In the following scenario, P_2 does a read after the write by P_1 (possibly a nanosecond after it), and gets 0. A subsequent read gives 1. Such behavior is incorrect for strictly consistent memory.

P_1	W(x) 1		
P_2		R(x) 0	R(x) 1

In summary, when memory is strictly consistent, all writes are instantaneously visible to all processors. If a memory location is changed, all subsequent reads from that location see the new value, no matter how soon after the change the reads are done.

Weak Consistency

Though strict consistency can be achieved for small-scale systems using hardware-based cache coherence protocols such as the write invalidate, it is impossible to achieve for large multi-processor systems. Also, overhead of the write-invalidate protocol can be excessive for large systems.

A better solution would be to first encapsulate the shared variables within a *critical section*, and let a processor finish its critical section and then make sure that the final results are sent everywhere, without worrying if all intermediate results had also been propagated to all memories in order, or even at all. This can be done using a synchronization variable used to synchronize memory. When a synchronization completes, all writes done on that processor are propagated outward, and all writes done on other processors are brought in. This is called a *weak consistency* model and has the following properties:

1. Accesses to the synchronization variables are strictly consistent. This says that when a synchronization variable is accessed, this fact is broadcast to other processors, and no other synchronization variable can be accessed in any other processor until this one is finished.
2. No access to a synchronization variable is allowed to be performed until all previous writes have completed everywhere. This means that when a processor accesses a synchronization variable, it forces all writes that are in progress or partially completed or completed at some memories, but not others to complete everywhere. When a synchronization access is done, all previous writes are guaranteed to be done as well. Also, by doing a synchronization after updating shared data, a processor can force the new values out to all other memories.
3. No data access (read or write) is allowed to be performed until all previous accesses to synchronization variables have been performed. When ordinary variables are accessed, for reading or for writing, all previous synchronizations have been performed. By synchronizing before reading shared data, a processor can be sure of getting the most recent values.

The following shows processor P_1 performing two writes to a shared variable, and then synchronizes (indicated by S). If $P2$ and $P3$ do not synchronize before reading the shared variable, no guarantees are given about what they see.

P_1	W(x) 1	W(x) 2	S			
P_2				R(x) 1	R(x) 2	S
P_3					R(x) 1	S

In the following, P_3 has synchronized, meaning that its memory has been brought up to date. When it reads x, it must, therefore, get the value 2. Getting 1 is not permitted.

P_1	W(x) 1	W(x) 2	S			
P_2				R(x) 1	R(x) 2	S
P_3				S	R(x) 2	

Release Consistency

Weak consistency has the problem that when a synchronization variable is accessed, the memory does not know whether this is being done because the processor is finished writing the shared

variables or about to start reading them. So, it must take the actions required in both cases, that is, making sure that all locally initiated writes have been propagated to other processors, as well as gathering all writes from other processors. If the memory could tell the difference between entering a critical region and leaving one, a more efficient implementation might be possible. Two kinds of synchronization operations are needed.

- *Acquire* accesses are used to tell the memory that a critical region is about to be entered. Typically, a shared memory block has associated with it a lock variable that must be acquired before read or write operations can be performed.
- *Release* accesses tell the memory that the processor has just exited the critical region. The lock variable must be unlocked before exiting the critical section.

Lock variables do not have to apply to all of memory. Instead they are typically used to guard specific shared variables, in which case, only those variables are kept consistent. The shared variables that are kept consistent are said to be protected. The contract between memory and software says that when a processor acquires the lock, the local copies of the corresponding protected variables will be brought up to date to be consistent with the remote ones if needed. When the lock is released, the changes to the protected variables are propagated to other processors.

The following shows a valid sequence of events for release consistency. Processor P_1 acquires the lock L, changes the shared variable x twice and then releases the lock. Processor P_2 acquires the lock and reads x. It is guaranteed to get the value of x at the time of the release, namely 2. If P_2 had tried to acquire the lock before P_1's release, the acquire would have been delayed until the release had occurred. Since P_3 does not do an acquire before reading x, the memory has no obligation to give it the current value, so returning 1 is valid.

P_1	Lock(L) 1	W(x) 1	W(x) 2	Unlock(L)			
P_2					Lock(L)	R(x) 2	Unlock(L)
P_3						R(x) 1	

Implementing the Release Consistency Model

The key part of implementing release consistency is ensuring that when multiple processors compete for the lock, only one should be able to acquire it. The others must wait for the lock to be released before trying to acquire it again. This can be achieved in a number of ways:

1. *No caching of the protected and lock variables.* In this model, the shared variables are never cached in local memory. A processor, dedicated to executing the operating system (OS), maintains all the shared variables and the corresponding locks. When processors want access to the protected variables, they attempt to gain access to the lock via an atomic operation (to be discussed later). The atomicity property guarantees that if multiple processors attempt this operation at the same time, the OS will allow only one to succeed.
2. *Selective cache flushing.* When a processor acquires the lock, the corresponding protected variables are now cached in its local memory and read/write operations can now be performed locally. When the lock is released, the protected variables are flushed (or written back) to the dedicated processor executing the OS.

3. *Hardware-based cache coherence.* The lock variables can be locally cached on individual processors, and consistency is achieved using the write-invalidate protocol.

Let us now take a closer look at hardware support for implementing the release consistency model. 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.

Atomic Exchange Operation

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 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 from also retrieving a 0.

The behavior of the exchange instruction is defined as follows. It is designed to atomically exchanges or swaps the value in the memory location and the register. Note however that the code snippet below cannot be executed atomically since it comprises of ordinary machine instructions. We will soon discuss how the exchange operation can be implemented in an atomic fashion using special instructions.

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

The following code shows how mutual exclusion is implemented using `exchange`.

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

Now we return to the question of how to implement `exchange` in atomic fashion. 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.

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

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
```

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.

We now briefly discuss other basic synchronization operations typically used in processors: compare and swap, and test and set.

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 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 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)
```

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 */
   }
}
```

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.