Synchronization

Coherency protocols guarantee that a reading processor (thread) sees the most current update to shared data.

Coherency protocols do not:
- make sure that only one thread accesses shared data or a shared hardware or software resource at a time
- force threads to start executing particular sections of code together

Critical sections order thread access to shared data

Barriers force threads to start executing particular sections of code together

Critical Sections

A critical section
- a sequence of code that only one thread can execute at a time
- provides mutual exclusion
  - a thread has exclusive access to the code & the data that it accesses
  - guarantees that only one thread can update the data at a time
- to execute a critical section, a thread
  - acquires a lock that guards it
  - executes its code
  - releases the lock

The effect is to synchronize or order the access of threads wrt their accessing shared data

Barriers

Barrier synchronization
- a barrier: point in a program which all threads must reach before any thread can cross
- threads reach the barrier & then wait until all other threads arrive
- all threads are released at once & begin executing code beyond the barrier
- example implementation of a barrier:
  - set a lock-protected counter to the number of processors
  - each thread (assuming 1/processor) decrements it
  - when the counter value becomes 0, all threads have crossed the barrier
- code that implements a barrier is a critical section
- useful for:
  - programs that execute in phases
  - synchronizing after a parallel loop

Locking

Locking facilitates access to a critical section.

Locking protocol:
- synchronization variable or lock
  - 0: lock is available
  - 1: lock is unavailable because another thread holds it
- a thread obtains the lock before it can enter a critical section
  - sets the lock to 1
  - thread releases the lock before it leaves the critical section
  - clears the lock

Acquiring a Lock

Atomic exchange instruction: swap a value in a register & a value in memory in one operation
- set the register to 1
- swap the register value & the lock value in memory
- new register value determines whether got the lock

AcquireLock:
```
li r3, #1    /* create lock value
swap r3, 0(r4) /* exchange register & lock
beqz r3, AcquireLock /* have to try again */
```
- also known as atomic read-modify-write a location in memory

Other examples
- test & set: tests the value in a memory location & sets it to 1
- fetch & increment/decrement: returns the value of a memory location + 1

Releasing a Lock

Store a 0 in the lock
Load-linked & Store Conditional

Performance problem with atomic read-modify-write:
- 2 memory operations in one
- must hold the bus until both operations complete

Pair of instructions appears atomic:
- avoids need for uninterruptible memory read & write
- load-locked & store-conditional
  - load-locked returns the original (lock) value in memory
  - if the contents of lock memory has not changed when the store-conditional is executed, the processor still has the lock
  - store-conditional returns a 1 if successful

```
GetLk:  li  R3, #1  /* create lock value
        li  R2, 0(R1) /* read lock variable
        ...  
        sc  R3, 0(R1) /* try to lock it
        beqz R3, GetLk /* cleared if sc failed
        ... (critical section)
```

Load-linked & Store Conditional

Implemented with special processor registers: lock-flag register & lock-address register
- load-locked sets lock-address register to lock’s memory address & lock-flag register to 1
- store-conditional returns lock-flag register value in the store register
- lock-flag register is cleared if the lock is written by another processor
- lock-flag register cleared if context switch or interrupt

Synchronization APIs

User-level software synchronization library routines constructed with atomic hardware primitives
- **spin locks**
  - **busywaiting** until obtain the lock
  - contention with atomic exchange causes invalidations (for the write) & coherency misses (for the rereads)
  - avoid if separate reading the lock & testing it
  - spinning done in the cache rather than over the bus

```
getLk:   li  R2, #1
spinLoop: li  R1, lockVariable
         blbs  R1, spinLoop
        sc  R2, lockVariable
        beqz R2, getLk
        st  R0, lockVariable
```

- **blocking locks**
  - block the thread after a certain number of spins

Synchronization Performance

An example overall synchronization/coherence strategy:
- design cache coherence protocol for little interprocessor contention for locks (the common case)
- add techniques to avoid performance loss if there is contention for a lock & still provide low latency if no contention

Have a race condition for acquiring a lock when it is unlocked
- \(O(p^2)\) bus transactions for \(p\) contending processors (write-invalidate)
- **exponential back-off** - software solution
  - each processor retries at a different time
  - successive retries done an exponentially increasing time later
- **queuing locks** - hardware solution
  - lock is passed from unlocking processor to waiting processor
  - also addresses fairness

Atomic Exchange in Practice

**Alpha**
- load-locked, store-conditional

**UltraSPARC**s (V9 architecture)
- several primitives
  - compare & swap, test & set, etc.

**Pentium Pro**
- compare & swap