Synchronization

Introduction

1. Providing efficient synchronization among cores and threads working together on the same program

Synchronization Example

1. 2 threads, counting occurrences of letters
   - Thread A: Counting in first half of document
   - Thread B: Counting in second half of document
2. Need to ensure concurrent accesses to the same memory location occur in serial
   - "Atomic" or "critical" sections

```
LW L,0(R1)
LW R, Count[L]
ADDI R, R, 1
SW R, Count[L]
```

Synchronization Example Lock

1. Atomic sections use locks (mutual exclusion)
   - Locks enforce mutual exclusion, but not a particular ordering

```
LW L,0(R1)
lock CountLock[L]
LW R, Count[L]
ADDI R, R, 1
SW R, Count[L]
unlock CountLock[L]
```

Lock Variable Quiz

1. Consider the following code:

```
lock(CountLock[L]);
count[L]++;
unlock(CountLock[L]);
```

2. What is CountLock[L]?
   - Just another location in shared memory (yes)
   - A location in a special synchronization memory
   - A special variable without a memory address

Lock Synchronization

1. The following code sample doesn't actually prevent synchronous access:

```
typedef int mutex_type;
void lock_init(mutex_type& lockvar) {
    lockvar = 0;
}

void lock(mutex_type& lockvar) {
    // this read-update-write must be done as one hardware instruction
    while(lockvar == 1);
    lockvar = 1;
```

```
}

void unlock(mutex_type& lockvar) {
    lockvar = 0;
}
```

Implementing Lock

1. Lamport's Bakery algorithm can achieve synchronization with just loads and stores, but is complicated
   - Expensive and slow
2. Special atomic read/write instructions

Atomic Instructions Quiz

1. To implement locks easily, we need:
   - A load
   - A store
   - An instruction that both reads and writes memory (yes)
   - An instruction that does not access memory

Atomic Instructions

1. Atomic exchange
   - EXCH R1, 78(R2)
     - Swaps the contents of R1 with the contents of R2
   - Writes all the time, even while the lock is busy

```
R1 = 1;
while(R1 == 1) {
    EXCH R1, lockvar;
}
```

2. Test-and-write
   - TSTSW R1, 78(R2)
   - Only does the write if the lock has the unlock state

```
if(mem[78+R2] == 0) {
    mem[78_R2] = R1;
    R1 = 1;
}
else {
    R1 = 0;
}
```

Test and Set Quiz

1. The following code for TSET R1, Addr is equivalent to...

```
if(mem[addr] == 0) {
    mem[addr] = 1;
    R1 = 1;
} else {
    R1 = 0;
}

lock(mutex_type& lockvar) {
    R1 = 0;
```

```
    while(R1 == 0) {
        TSET R1, lockvar
    }
}
```

Atomic Instructions 2

1. Test-and-write solves the problem of continuously writing to the variable, but is a strange instruction (neither a load nor a store)
2. A load linked/store conditional behaves more similar to our typical instructions

Load Linked Store Conditional

1. Atomic read/write in same instruction is bad for pipelining
   - Test-and-write takes multiple cycles to read/write memory
     - Requires multiple memory stages, complicates the implementation
   - Instead, split into two separate instructions
2. Load linked
   - Behaves just like a normal load
   - Save address in special link register
3. Store conditional
   - Check if address is same as in link register
     - If so, do a normal SW and return 1
     - Else, return 0
4. The LL/SC combination behaves like a single atomic instruction

How is LL SC Atomic?

1. LL/SC relies on coherence
   - LL R1, lockvar
   - SC R2, lockvar
   - If the value in the link register changed, we know somebody else had the lock and we can't store to that location
2. Simple critical sections don't need locks anymore
   - Use LL/SC directly

```
TRY: LL R1, var
R1 = R1 + 1;
SC R1, var
if(R1 == 0)
    goto TRY;
```

LL SC Lock Quiz

1. Consider the following code section:

```
void lock(mutex_type& lockvar) {
    trylock:
        MOV R1, 1
        LL R2, lockvar
        SC R1, lockvar
}
```

2. Which of the following implements the lock correctly?
   - BEQZ R1, trylock
   - BEQZ R2, trylock

- BNEZ R2, trylock; BEQZ LINK, trylock
- BNEZ R2, trylock; BEQZ R1, trylock (correct)
- BNEZ R2, trylock; BNEZ LINK, trylock

3. Can't access the link register directly, only implicitly through registers

Locks and Performance

1. Exchange locks have bad performance due to continually invalidating/moving data between caches due to busy waiting
   - Power hungry
   - Slows down useful work in other threads due to bus contention

Test and Atomic OP Lock

1. Instead of the following:

```
R1 = 1;
while(R1 == 1) {
    EXCH R1, lockvar;
}
```

2. Use normal loads while lockvar is busy
   - Spins on the local value

```
R1 = 1;
while(R1 == 1) {
    while(lockvar == 1);
    EXCH R1, lockvar;
}
```

Test and Atomic OP Quiz

1. The following code implements a lock using LL/SC instructions:

```
void lock(mutex_type& lockvar) {
    trylock:
        MOV R1, 1
        LL R2, lockvar
        SC R1, lockvar
        BNEZ R2, trylock
        BEQZ R1, trylock
}
```

2. How can these be rearranged to use test-and-atomic-op instructions?

```
void lock(mutex_type& lockvar) {
    trylock:
        MOV R1, 1
        LL R2, lockvar
        BNEZ R2, trylock
        SC R1, lockvar
        BEQZ R1, trylock
}
```

Unlock Quiz

1. How do we implement unlock?

- SW 0, lockvar (yes)
- LL, see if 1, SC
- We need additional atomic instructions

Barrier Synchronization

1. Occurs when there's is a parallel section where several threads are working independently and we want to wait for each thread to finish
   - All must arrive before any can leave
2. Two variables
   - Counter (count arrived threads)
   - Flag (set when counter == N)

Simple Barrier Implementation

1. Consider the following barrier implementation:

```
lock(counterLock);
    if(count == 0) {
        release = 0;
    }
    count++;
unlock(counterLock);
if(count == total) {
    count = 0;
    release = 1;
} else {
    spin(release == 1); // wait for release to be 1
}
```

Simple Barrier Implementation Doesn't Work

1. The above implementation of a barrier does not work
   - Can't be used more than once because the count = 0; and release = 1; instructions aren't completed atomically
     - Can result in deadlock

Simple Barrier Quiz

1. Consider the case where two threads synchronize on a barrier, then the second thread ends
2. Does a simple barrier work in this case?
   - Yes: The simple barrier only breaks down when trying to reuse it

Reusable Barrier

1. Reusable barrier can be implemented as follows:

```
localSense = !localSense;
lock(counterLock);
    count++;
    if(count == total) {
        count = 0;
        release = localSense;
    }
unlock(counterLock);
spin(release == localSense);
```

2. Using the boolean localSense prevents a thread from racing through the next barrier (flipping instead of reinitializing)

Conclusion

1. Locks and barriers are needed to coordinate activities among multiple threads
   - Atomic instructions are needed to implement locks efficiently
2. Synchronization can be messed up when a processor reorders loads and stores