• Topics: synchronization wrap-up, need for sequential consistency, fences
Test-and-Test-and-Set

- lock:
  - test register, location
  - bnx register, lock
  - t&s register, location
  - bnx register, lock
  - CS
  - st location, #0
Load-Linked and Store Conditional

- LL-SC is an implementation of atomic read-modify-write with very high flexibility

- LL: read a value and update a table indicating you have read this address, then perform any amount of computation

- SC: attempt to store a result into the same memory location, the store will succeed only if the table indicates that no other process attempted a store since the local LL (success only if the operation was “effectively” atomic)

- SC implementations do not generate bus traffic if the SC fails – hence, more efficient than test&test&set
Spin Lock with Low Coherence Traffic

lockit:    LL     R2, 0(R1)    ; load linked, generates no coherence traffic
    BNEZ    R2, lockit ; not available, keep spinning
    DADDUI R2, R0, #1 ; put value 1 in R2
    SC      R2, 0(R1)   ; store-conditional succeeds if no one
                     ; updated the lock since the last LL
    BEQZ    R2, lockit ; confirm that SC succeeded, else keep trying

• If there are i processes waiting for the lock, how many bus transactions happen?
Spin Lock with Low Coherence Traffic

lockit:    LL   R2, 0(R1)    ; load linked, generates no coherence traffic
BNEZ    R2, lockit ; not available, keep spinning
DADDUI R2, R0, #1 ; put value 1 in R2
SC     R2, 0(R1)   ; store-conditional succeeds if no one
        ; updated the lock since the last LL
BEQZ    R2, lockit ; confirm that SC succeeded, else keep trying

• If there are $i$ processes waiting for the lock, how many
  bus transactions happen?
  1 write by the releaser + $i$ (or 1) read-miss requests +
  $i$ (or 1) responses + 1 write by acquirer + 0 (i-1 failed SCs) +
  i-1 (or 1) read-miss requests + i-1 (or 1) responses

(The $i/i-1$ read misses and responses can be reduced to 1)
Lock Vs. Optimistic Concurrency

lockit:   LL   R2, 0(R1)
         BNEZ   R2, lockit
         DADDUI R2, R0, #1
         SC   R2, 0(R1)
         BEQZ   R2, lockit

Critical Section
         ST   0(R1), #0

tryagain: LL   R2, 0(R1)
          DADDUI R2, R2, R3
          SC   R2, 0(R1)
          BEQZ   R2, tryagain

LL-SC is being used to figure out if we were able to acquire the lock without anyone interfering – we then enter the critical section

If the critical section only involves one memory location, the critical section can be captured within the LL-SC – instead of spinning on the lock acquire, you may now be spinning trying to atomically execute the CS
Coherence Vs. Consistency

• Recall that coherence guarantees (i) that a write will eventually be seen by other processors, and (ii) write serialization (all processors see writes to the same location in the same order)

• The consistency model defines the ordering of writes and reads to different memory locations – the hardware guarantees a certain consistency model and the programmer attempts to write correct programs with those assumptions
Example Programs

Initially, A = B = 0

P1
A = 1
if (B == 0)
critical section

P2
B = 1
if (A == 0)
critical section

Initially, Head = Data = 0

P1
Data = 2000
while (Head == 0)
Head = 1
{}

P2
... = Data

Initially, A = B = 0

P1
A = 1

P2
if (A == 1)
B = 1

P3
if (B == 1)
register = A
Sequential Consistency

We assume:
- Within a program, program order is preserved
- Each instruction executes atomically
- Instructions from different threads can be interleaved arbitrarily

Valid executions:
- abAcBCDdeE...
- ABCDEFabGc...
- abcAdBe...
- aAbBcCdDeE...
- .....
Problem 1

• What are possible outputs for the program below?

Assume x=y=0 at the start of the program

Thread 1
  x = 10
  y = x+y
  Print y

Thread 2
  y = 20
  x = y+x
Problem 1

• What are possible outputs for the program below?

Assume x=y=0 at the start of the program

Thread 1                                Thread 2
A     x = 10                               a    y=20
B     y = x+y b    x = y+x
C     Print y

Possible scenarios:  5 choose 2 = 10
ABCab   ABaCb   ABabC   AaBCb   AaBbC
  10 20 20 30 30
AabBC   aABCb   aABbC   aAbBC   abABC
  50 30 30 50 30
Sequential Consistency

- Programmers assume SC; makes it much easier to reason about program behavior

- Hardware innovations can disrupt the SC model

- For example, if we assume write buffers, or out-of-order execution, or if we drop ACKS in the coherence protocol, the previous programs yield unexpected outputs
Consistency Example - 1

- An ooo core will see no dependence between instructions dealing with A and instructions dealing with B; those operations can therefore be re-ordered; this is fine for a single thread, but not for multiple threads

```
Initially A = B = 0
P1                        P2
A ← 1                   B ← 1
...                      ...
if (B == 0)            if (A == 0)
Crit.Section       Crit.Section
```

The consistency model lets the programmer know what assumptions they can make about the hardware’s reordering capabilities
Initially, $A = B = 0$

P1

A = 1

if (A == 1)

B = 1

P2

if (B == 1)

register = A

P3

If a coherence invalidation didn’t require ACKs, we can’t confirm that everyone has seen the value of A.
Sequential Consistency

- A multiprocessor is sequentially consistent if the result of the execution is achievable by maintaining program order within a processor and interleaving accesses by different processors in an arbitrary fashion.

- Can implement sequential consistency by requiring the following: program order, write serialization, everyone has seen an update before a value is read – very intuitive for the programmer, but extremely slow.

- This is very slow... alternatives:
  - Add optimizations to the hardware (e.g., verify loads)
  - Offer a relaxed memory consistency model and fences
Relaxed Consistency Models

• We want an intuitive programming model (such as sequential consistency) and we want high performance

• We care about data races and re-ordering constraints for some parts of the program and not for others – hence, we will relax some of the constraints for sequential consistency for most of the program, but enforce them for specific portions of the code

• Fence instructions are special instructions that require all previous memory accesses to complete before proceeding (sequential consistency)
Fences

P1
{
    Region of code with no races
}
Fence
    Acquire_lock
Fence
{
    Racy code
}
Fence
    Release_lock
Fence

P2
{
    Region of code with no races
}
Fence
    Acquire_lock
Fence
{
    Racy code
}
Fence
    Release_lock
Fence