Complex Atomic Operations

CS511

Complex Atomic Operations

- Its not easy to solve the MEP using atomic load and store, as we have seen
- This difficulty disappears if we allow more complicated atomic operations
- Note: Also known as read-modify-write (RMW) operations
- In this class we take a look at some examples

Four Solutions

- We'll see four solutions using complex atomic statements
  - Compare and set
  - Test and set
  - Exchange
  - Fetch and add
- These are all equivalent

Compare And Swap

Other Complex Atomic Actions

Compare and Set (CAS)

```
atomic boolean compare_and_set(register,expected_value,new_value)
    boolean updated = false;
    if (register.read() == expected_value) {
      register.set(new_value);
      updated=true;
    }
    return updated;
}
```

- Atomically update register with `new_value` if its current value is `expected_value`
- Hardware support (Eg.[1] CMPXCHG—Compare and Exch.)

Description

Compares the value in the AL, AX, EAX, or RAX register with the first operand (destination operand). If the two values are equal, the second operand (source operand) is loaded into the destination operand. Otherwise, the destination operand is loaded into the AL, AX, EAX or RAX register. RAX register is available only in 64-bit mode.

This instruction can be used with a LOCK prefix to allow the instruction to be executed atomically. To simplify the interface to the processor's bus, the destination operand receives a write cycle without regard to the result of th

[1] www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-instruction-set-reference-ma pdf

Compare and Set - Turnstile Example Revisited

```
3  import java.util.concurrent.atomic.*
4
5  AtomicInteger c = new AtomicInteger(0)
6
7  P = Thread.start {
8        Integer i
9        50.times {
10           do {
11               i = c.get()
12           } while (!c.compareAndSet(i,i+1))
13       }
14   }
15
16 Q = Thread.start {
17       Integer i
18       50.times {
19           do {
20               i = c.get()
21           } while (!c.compareAndSet(i,i+1))
22       }
23   }
24
25 P.join()
26 Q.join()
27 println c
```

Compare and Set - Turnstile Example Revisited

```
3  import java.util.concurrent.atomic.*
4
5  AtomicBoolean mutex = new AtomicBoolean(false)
6  c=0
7
8  P = Thread.start {
9      50.times {
10         while (!mutex.compareAndSet(false,true)) {}
11         c++
12         mutex.set(false)
13     }}
14
15 Q = Thread.start {
16     50.times {
17         while (!mutex.compareAndSet(false,true)) {}
18         c++
19         mutex.set(false)
20     }}
21
22 P.join()
23 Q.join()
24 println c
```

Mutex+Absence of Livelock

Busy Waiting vs Blocking

What should one do if one can't get a lock?

- Keep trying
  - "spin" or "busy-wait"
  - Makes sense on multiprocessors if delays are short
  - Example: Peterson, Dekker, Bakery, CompareAndSet
- Give up the processor
  - Good if delays are long (eg. consumer waiting to dequeue from an empty shared buffer)
  - Always good on uniprocessor but also makes sense for multiprocessors
  - Upcoming examples: semaphores, monitors

Compare And Swap

Other Complex Atomic Actions

Test and Set

```
atomic boolean TestAndSet(ref) {
  result = ref.value; // reads the value before it changes it
  ref.value = true;   // changes the value to true
  return result;      // returns the previously read value
}
```

Revisiting our example:

```
1  Ref shared = new Ref();
2  shared.value = false;
```

```
3  Thread.start { //P
4    while (true) {
5      // non-critical section
6      await !TestAndSet(shared));
7      // critical section
8      shared.value = false;
9      // non-critical section
10   }
11 }
```

```
3  Thread.start { //Q
4    while (true) {
5      // non-critical section
6      await !TestAndSet(shared);
7      // critical section
8      shared.value = false;
9      // non-critical section
10   }
11 }
```

Exchange

```
atomic void Exchange(sref, lref) {
    temp       = sref.value;
    sref.value = lref.value;
    lref.value = temp;
}
```

Revisiting our example

```
1  Ref shared = new Ref();
2  shared.value = 0;
```

```
3  Thread.start { // P
4    local = new Ref();
5    local.value = 1;
6    while (true) {
7      // non-critical section
8      do
9         Exchange(shared,local)
10     while (local.value == 1);
11     // critical section
12     Exchange(shared,local);
13     // non-critical section
14   }
15 }
```

```
3  Thread.start { //Q
4    local = new Ref();
5    local.value = 1;
6    while (true) {
7      // non-critical section
8      do
9         Exchange(shared,local)
10     while (local.value == 1);
11     // critical section
12     Exchange(shared,local);
13     // non-critical section
14   }
15 }
```

Problem

- Previous solutions do not guarantee serving in the order in which they arrive
- Can we use an atomic operation that allows us to guarantee the order?

Fetch and Add

```
atomic int FetchAndAdd(ref, x) {
  temp = ref.value;
  ref.value = ref.value + x;
  return temp;
}
```

Revisiting our example

```
1  Ref ticket = new Ref();
2  Ref turn   = new Ref();
3  ticket.value = 0;
4  turn.value   = 0;
5
6  Thread.start { //P
7    int myTurn;
8    // non-critical section
9    myTurn = FetchAndAdd(ticket, 1);
10   await (turn.value == myTurn.value);
11   // critical section
12   FetchAndAdd(turn, 1);
13   // non-critical section
14 }
```