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

Revisiting Attempt 0

```
1 boolean flag = false;

2 Thread.start { // P
3   // non-critical section
4   await !flag;
5   flag = true;
6   // critical section
7   flag = false;
8   // non-critical section
9 }
```

```
2 Thread.start { // Q
3   // non-critical section
4   await !flag;
5   flag = true;
6   // critical section
7   flag = false;
8   // non-critical section
9 }
```

- What was the problem with this?
- Can we introduce an atomic operations that can correct this? What would it have to do?

Revisiting Attempt 0

```
1  boolean flag = false;

2  Thread.start { // P
3    // non-critical section
4    atomic {
5      await !flag;
6      flag = true;
7    }
8    // critical section
9    flag = false;
10   // non-critical section
11 }
```

```
2  Thread.start { // Q
3    // non-critical section
4    atomic {
5      await !flag;
6      flag = true;
7    }
8    // critical section
9    flag = false;
10   // non-critical section
11 }
```

- Suppose we could ensure the atomicity of certain combinations of operations
- What can we say about mutual exclusion now? Draw the state diagram

How to Define Atomic Operations?

- Specific atomic operations are provided by hardware
- Eg.[1] CMPXCHG—Compare and Exchange

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

- Other instructions can be prefixed with lock too

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

Three Solutions

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

Three Solutions

- The solutions require that we pass arguments to methods that are to be modified
- Therefore we shall use a dummy class

```
class Ref {
  boolean value;
}
```

- Passing arguments by reference will be achieved simply by passing arguments of type `Ref`

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 }
```

Busy waiting

- All solutions seen up until now are inefficient given that they consume CPU time while they wait.
- It would be much better to suspend execution of a process that is trying to enter the critical region until it is possible to do so.
- This can be achieved using semaphores.