

# 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
- ▶ In this class we take a look at some examples

# Revisiting Attempt 0

```
1 global boolean flag = false;

2 thread { //                                2 thread {
3   // non-critical section                  3   // non-critical section
4   await !flag;                             4   await !flag;
5   flag = true;                             5   flag = true;
6   // critical section                      6   // critical section
7   flag = false;                           7   flag = false;
8   // non-critical section                  8   // non-critical section
9 }                                           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 global boolean flag = false;

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

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

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

```
3 thread {  
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 {  
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 global Ref shared = new Ref();  
2 shared.value = 0;
```

```
3 thread {  
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 {  
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 global Ref ticket = new Ref();  
2 global Ref turn   = new Ref();  
3 ticket.value = 0;  
4 turn.value   = 0;  
5  
6 thread {  
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.