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

```
global boolean flag = false;
```

```
thread { //
  // non-critical section
  await !flag;
  flag = true;
  // critical section
  flag = false;
  // non-critical section
}
```

```
thread {
  // non-critical section
  await !flag;
  flag = true;
  // critical section
  flag = false;
  // non-critical section
}
```

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

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;

  public Ref(boolean initValue) {
    value=initValue;
  }
}
```

- 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:

```
global Ref shared = new Ref(false);

thread {
  // non-critical section
  await !TestAndSet(shared));
  // critical section
  shared.value = false;
  // non-critical section
}
```

```
thread {
  // non-critical section
  await !TestAndSet(shared);
  // critical section
  shared.value = false;
  // non-critical section
}
```

Exchange

Note: we assume `Ref` stores integers

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

Revisiting our example

```
global Ref shared = new Ref(0);
```

```
thread {
  local = new Ref(1);
  // non-critical section
  do
     Exchange(shared,local)
  while (local.value == 1);
  // critical section
  Exchange(shared,local);
  // non-critical section
}
```

```
thread {
  local = new Ref(1);
  // non-critical section
  do
     Exchange(shared,local)
  while (local.value == 1);
  // critical section
  Exchange(shared,local);
  // non-critical section
}
```

Exchange

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

Revisiting our example

```
global Ref shared = new Ref(0);

thread {
  local = new Ref(1);
  while (true) {
    do
      Exchange(shared,local)
    while (local.value == 1);
    Exchange(shared,local);
  }
}
```

```
thread {
  local = new Ref(1);
  while (true) {
    do
      Exchange(shared,local)
    while (local.value == 1);
    Exchange(shared,local);
  }
}
```

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) {
  local = ref.value;
  ref.value = ref.value + x;
  return local;
}
```

Revisiting our example

```
global Ref ticket = new Ref(0);
global Ref turn   = new Ref(0);

thread {
  int myTurn;
  // non-critical section
  myTurn = FetchAndAdd(ticket, 1);
  await (turn.value == myTurn.value);
  // critical section
  FetchAndAdd(turn, 1);
  // non-critical section
}
```

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 monitors
- Monitors, moreover, provide a high-level construct for synchronization.