Unit 7: Programming exercises
Vijay Saraswat

Day before Thanksgiving fun!

Many of the following programs are based on presentations in [?].

1 Successive attempts at mutual exclusion

We are interested in developing different solutions for the mutual exclusion problem. We shall develop implementations for the following interface:

```
interface Lock[T] {
  def lock(t:T);
  def unlock(t:T);
}
```

We want solutions for the mutual exclusion problem which satisfy the following properties:

Mutual Exclusion Only one activity may be in the critical section (have executed its `lock` operation but not its `unlock` operation.

Freedom from Deadlock If *some* activity is trying to enter the critical region, then at least one of them will succeed in doing so.

Freedom from Starvation An activity trying to enter its critical region will eventually succeed.

1.1 First attempt

```
type Two=Int{0 <= self <=1}
class Lock1 implements Lock[Two] {
  private var turn:Int=1;

  public def lock(i:Two) {
   await turn == i;
  }

  public def unlock(i:Two) {
    turn=1-i;
  }
}
```

1.2 Second attempt

```
class Lock2 implements Lock[Two] {
  private val want = Rail.makeVar[Boolean](2, (Int)=>false);

  public def lock(i:Two) {
   await !want(1-i);
   want(i)=true;
  }

  public def unlock(i:Two) {
    want(i)=false;
  }
}
```

1.3 Third attempt

```
class Lock3 implements Lock[Two] {
  private val want = Rail.makeVar[Boolean](2, (Int)=>false);

  public def lock(i:Two) {
   want(i)=true;
   await !want(1-i);
  }

  public def unlock(i:Two) {
    want(i)=false;
  }
}
```

1.4 Fourth attempt

```
class Lock4 implements Lock[Two] {
  private val want = Rail.makeVar[Boolean](2, (Int)=>false);

  public def lock(i:Two) {
   want(i)=true;
   while (want(1-i)) {
     want(i)=false;
     want(i)=true;
   }
  }

  public def unlock(i:Two) {
    want(i)=false;
  }
}
```

1.5 Dekkler's algorithm

```
class Dekker implements Lock[Two] {
  private val want = Rail.makeVar[Boolean](2, (Int)=>false);
  private var turn:Int=1;

  public def lock(i:Two) {
   want(i)=true;
   while (want(1-i)) {
     if (turn==1-i) {
       want(i)=false;
       await turn==i;
       want(i)=true;
     }
   }
  }

  public def unlock(i:Two) {
    turn = 1-i;
    want(i)=false;
  }
}
```

For comparison, here is Peterson:

```
class Peterson implements Lock[Two] {
  private val flag = Rail.makeVar[Boolean](2, (Int)=>false);
  private var victim:int=0;

  public def lock(i:Two) {
    val j = 1 - i;
    flag(i)=true;
    await (flag(i) && victim==i);
  }

  public def unlock(i:Two) {
    flag(i)=false;
  }
}
```

2 Dining philosophers

2.1 First attempt

```
type Five=Int{0 <= self <=4}
class Philosopher {
  private val fork = Rail.makeVar[Semaphore](5, (Int)=new Semaphore(1));

  public def getForks(i:Five) {
    fork(i).p();
    fork(i+1).p();
  }

  public def giveForks(i:Five) {
    fork(i).v();
    fork(i+1).v();
  }

}
```

2.2 Second attempt

```
type Five=Int{0 <= self <=4}
class Philosopher {
  private val fork = Rail.makeVar[Semaphore](5, (Int)=new Semaphore(1));
  private val room = new Semaphore(4);
  public def getForks(i:Five) {
    room.p();
    fork(i).p();
    fork(i+1).p();
  }

  public def giveForks(i:Five) {
    fork(i).v();
    fork(i+1).v();
    room.v();
  }

}
```