Massachusetts Institute of Technology
Department of Electrical Engineering and Computer Science

6.826 – Principles of Computer Systems

Handout 27 October 13, 1993

# Problem Set #5

**Due date: October 20, 1993**

**Problem 1** (20 points)

For each of the following systems, indicate whether or not it can deadlock, and explain briefly why or why not.

Note: each example involves two variables declared with type `Mutex`. We assume here that this type provides methods `Acquire` and `Release` that can be called on a variable of the type, with each variable treated as a separate instance of the `Mutex` specification given in Handout 20.

**(A)** (5 points)

```
VAR m1: Mutex
    m2: Mutex
    x: Int := 0

  THREAD P () =
     m1.Acquire();
     m2.Acquire();
     x := x + 1;
     m1.Release();
     m2.Release()

  THREAD Q () =
     m1.Acquire();
     m2.Acquire();
     x := x + 1;
     m1.Release();
     m2.Release()
```

**(B)** (5 points)

```
VAR m1: Mutex
    m2: Mutex
    x: Int := 0

THREAD P () =
   m1.Acquire();
   m2.Acquire();
   x := x + 1;
   m2.Release();
   m1.Release()

THREAD Q () =
   m1.Acquire();
   m2.Acquire();
   x := x + 1;
   m1.Release();
   m2.Release()
```

**(C)** (5 points)

```
VAR m1: Mutex
    m2: Mutex
    x: Int := 0

THREAD P () =
   m1.Acquire();
   m2.Acquire();
   x := x + 1;
   m2.Release();
   m1.Release()

THREAD Q () =
   m2.Acquire();
   m1.Acquire();
   x := x + 1;
   m1.Release();
   m2.Release()
```

**(D)** (5 points)

```
VAR m1: Mutex
    m2: Mutex
    x: Int := 0

THREAD P () =
   m1.Acquire();
   m2.Acquire();
   x := x + 1;
   m2.Release();
   m1.Release()

THREAD Q () =
   m2.Acquire();
   m1.Acquire();
   x := x + 1;
   m2.Release();
   m1.Release()
```

**Problem 2** (20 points)

**(A)** (10 points)
Consider a system in which the data is organized in a tree, with a mutex associated with each tree node. Assume that each thread that accesses the tree does so according to the following rules:

- It starts by locking an arbitrary node that is an ancestor of all the nodes it wants to access.
- The data stored in a node is accessed only if the thread holds the lock on the node.
- After locking its starting node, a thread locks a node only if it holds a lock on some ancestor of the node.
- A thread does not lock a node if it holds a lock on any non-ancestor of the node (e.g., a descendant or a sibling).

Prove that this system cannot deadlock.

**(B)** (10 points)
Suppose that the data is organized as a directed acyclic graph (DAG) instead of a tree. How, if at all, must the rules in part (A) change to ensure that threads do not deadlock? Explain briefly.

**Problem 3** (35 points)

Here is the spec for a simple clock that can be incremented and read. The idea is that `Delay` will space out increments of the clock variable `t` appropriately, and the clients, possibly more than one, will call `Read`. Note that `Read` is not atomic; it returns some value of `t` between the values at the time it is invoked and the time it returns.

```
MODULE Clock

VAR t: Int

THREAD Inc() = DO << t := t + 1; Delay() >> OD

PROC Read() -> Int = VAR t0: Int |
       << t0 := t >>;
       << VAR t1: Int | t0 <= t1 /\ t1 <= t => RET t1 >>
END Clock
```

Here is an implementation of `Clock` given atomic read and write of single `Words`, assuming that two `Words` are required to represent `t`, that a single `Word` variable is read or written atomically (the standard Spec semantics), and that `Word` has a method

```
"&" := FUNC (hi: Word, lo: Word) -> Int = RET hi * ClockImpl.base + lo
```

In other words, you can write `hi & lo` for `hi * base + lo`. It works by incrementing `lo` and propagating a carry to `h1` and `h2` if `lo` has reached `base`.

```
MODULE ClockImpl

VAR base := ...             % constant
VAR lo: Word := 0,
    h1: Word := 0,
    h2: Word := 0

THREAD Inc() =
    DO VAR newLo, newHi: Word |
        newLo := lo + 1; newHi := h1;
        BEGIN newLo = base => newLo := 0;
                              newHi := newHi + 1
              [*] SKIP END;
        h2 := newHi; lo := newLo; h1 := newHi;
        Delay()
    OD

PROC Read() -> Int =
    VAR th1, tlo, th2, tt: Mem.Word |
```

```
        th1 := h1; tlo := lo; th2 := h2;
        tt := th2 & ( th1 = th2 => tlo [*] 0 ); RET tt

    END ClockImpl
```

**(A)** (10 points)

Use the results of Handout 21 on implementing atomic actions to add `<<...>>` brackets to `ClockImpl` in such a way that you get the largest possible atomic commands, so that part (C) of the problem is less work. Insert labels for all the possible PC values in both `Clock` and the modified `ClockImpl`.

**(B)** (5 points)

Write an abstraction function from `ClockImpl` to `Clock`. You might want to add a history variable to make this convenient. Don't forget that the thread PCs are part of the state.

**(C)** (15 points)

Give a proof by the simulation method described in class that `ClockImpl` implements `Clock`, assuming that the external transitions are the invocation and return from `Read`, and that there's only one reader at a time. Be sure to state the invariants explicitly; it's convenient to write invariants of the form "`$pc = x ==> P`" as "`INVARIANT P`" at the label x.

**(D)** (5 points)

Suppose there are two threads calling `Read` concurrently. Does `ClockImpl` still implement `Clock`? Why or why not?

**(E)** (5 points)

[optional] Suppose we make `Clock1` from `Clock` by changing `Clock.Read` to the simpler

```
PROC Read() -> Int = << RET t >>
```

Does `ClockImpl` implement `Clock1` (still with one reader)? Why or why not? What happens with two readers?