Massachusetts Institute of Technology
Department of Electrical Engineering and Computer Science

6.826 – Principles of Computer Systems

Handout ?? November 10, 1993

Solutions to Problem Set #5

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

There is a total ordering on the acquisition of mutexes. Mutex `m1` is always acquired before mutex `m2` by both threads; therefore, it is impossible for the two threads to hold one mutex each and want to acquire the other. As a result, the system cannot deadlock.

(B) (5 points)

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

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

Similar to part A, the two mutexes are acquired in the same order, so no deadlock can be possible. The order of mutex release does not matter.

(C) (5 points)

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

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

Deadlock is possible. The mutexes are not acquired in the same order by the two threads. If each thread executes its first line before the other executes its second, then each holds the mutex that the other desires, so they will wait for each other forever.

(D) (5 points)

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

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

Similar to part C, deadlock is possible if the first lines of both threads are executed before the second line of either thread. Each thread is waiting for the mutex held by the other thread. Again, the order of mutex release does not matter in this case.

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.

The system cannot deadlock. We are to prove this by contradiction.

Suppose this system can deadlock. There must exist a situation where n threads, $t_1, t_2, \ldots, t_n$ are waiting for each other in a ring. Suppose t_i has lock l_i which t_{i-1} $(i \geq 2)$ is waiting for, and t_1 has l_1 which t_n is waiting for. Also, let node(l_i) denotes the node of l_i in the tree. For i<n, t_i has l_i and is waiting for l_{i+1}; therefore, node(l_i) is an ancestor of node(l_{i+1}). By transitivity of the ancestor relation in a tree, node(l_i) is an ancestor of node(l_n). However, since t_n is waiting for l_1 while holding l_n, node (l_n) is also the ancestor of node (l_1). This is an contradiction.[1]

Another method to prove that the system cannot deadlock is to give a partial ordering of all the locks. It can be done by assigning a unique number to each lock on the node. The number is assigned in a monotonically increasing fashion as we search the tree nodes by "walking in a left-to-right breath first search manner". Then the rules stated in the problem require lock acquisitions to be done according to the partial ordering imposed on the locks. Therefore, no deadlock is possible.

Comments: Note that the definition of ancestor includes the node itself, as well as its strict ancestors. For this problem, let us assume a thread never tries to lock a node it already has locked. With this definition of ancestor, the root node can also be locked and accessed.[2]

(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.

Intuitively, a directed acyclic graph (DAG) does not contain any cycle, we may, therefore, argue that deadlock is impossible on a DAG using a similar set of rules used in part (A). However, there are two particularly tricky issues to be dealt with for a DAG before the rules in part (A) are to be workable without causing deadlock on a DAG.

First, a node in a DAG, unlike a node in a tree, may have more than one direct ancestor; as a result, it may not be possible to find a node that is an ancestor of **all** the nodes a thread wants to access. For

[1] Proof written by C.S. Wang

[2] Pointed out by Dawson Engler and Deborah Wallach

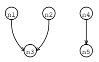

Figure 1: A Simple DAG with a Node Having Two Direct Ancestors and Disconnected Components.

example, in figure 1, node n_3 has two direct ancestors n_1 and n_2. If a thread wants to access data in n_2 and n_3, then the thread cannot find a starting node which is an ancestor of all the nodes the thread wants to access. (Rule 1 of part (A) fails to be executed.)

Second, a DAG may not be connected, that is, it may contain more than one connected components. For example, in figure 1, if thread t_1 locks n_1 and n_5 while thread t_2 locks n_3 and n_4 and if t_1 tries to lock n_3 and t_2 tries to lock n_5, then a deadlock can occur even with the set of rules used in part (A).

As a result, we first want to do a topological sort on each connected component of the DAG to impose a partial ordering on the nodes within each DAG component. Now a node n_2 is an "ancestor" of node n_3 if it is lower down in the topological sorted list of nodes. This solves the first problem mentioned above because there is a topological sort picks an ordering between any two direct ancestors of a node in the connected component of the DAG. For example, a topological sorted list for the left component of figure 1 can be (n_1, n_2, n_3) or (n_2, n_1, n_3).

Then we arbitrarily order the connected components and require a client to complete acquiring all the locks it wants to access in one connected component before it can start to acquire any locks in the next connected component in the "component-order list". This is equivalent to making a connected DAG out of all the components by adding links from every node in connected component to every node in the next connected component in the "component-order list". This resulting DAG is a single connected component containing no cycles and having a partial order imposed on all the nodes in the DAG, so no deadlock can occur. For example, in figure 1, the right component may be followed by the left component in the "component-order list". The resulting abstract DAG can then have a topological sorted list of nodes (n_4, n_5, n_2, n_1, n_3).

Problem 3 (35 points)

(A) (10 points)

```
THREAD Inc() = DO [I0] << t := t + 1 ; Delay() >> OD [I9]

PROC Read() -> Int =[R1]
    VAR t0: Int |
```

```
            << t0 := t >>
        ;[R2] << VAR t1: Int | t0 <= t1 /\ t1 <= t => RET t1 >>
         [R4]

THREAD Inc() =[i0]
    DO VAR newLo, newHi: Word |
            << newLo := lo + 1 >>
        ;[i1] << newHi := h1 >>
        ;[i2] << BEGIN newLo = base => newLo := 0; newHi := newHi + 1
                     [*] SKIP
                  END >>
        ;[i3] << h2 := newHi >>
        ;[i4] << lo := newLo >>
        ;[i5] << h1 := newHi; Delay() >>
    OD

PROC Read() -> Int =[r1]
    VAR th1, tlo, th2, tt, tStart: Mem.Word |
            << th1 := h1; tStart := t >>
        ;[r2] << tlo := lo >>
        ;[r3] << th2 := h2;
                  tt := th2 & ( th1 = th2 => tlo [*] 0 );
                  RET tt >>
         [r4]
```

Note that PC `[I9]` is never reached.

(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.*

Add the history variable `tStart` to `Read` and `tStart:=t` before `r2` as shown above.

The abstraction function is then:

`t`	=	`h2 & ( h1 = h2 => lo [*] 0 )`	
`t0`	=	`tStart`	
`t1`	=	`tt`	
`Inc.$pc`	=	`I0`	always
`Read.$pc`	=	`R1`	if at `r1`
		`R2`	if at `r2` or `r3`
		`R4`	if at `r4`

(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`.

```
% Invariants
%    h1 =  h2 at i0
%    h1 <= h2
%    th1 <= h1 at r2, r3, r4
```

etc.

(D) (5 points)

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

Yes. Since the two readers have disjoint local state and none of the actions of the reader affect the global state, the same invariants and the same simulation argument apply to the local state of the second reader.

(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?*

With one reader thread `T`, yes, because there's no way to tell whether or not `t` is beyond the result of `Read` (which is in `T.$a` at `[R4]`). Proving this with an abstraction function is not so easy, however, because there's no transition of `ClockImpl` that corresponds to the one transition of `Clock1.Read.`

With two readers `T1` and `T2` it depends on exactly how we interpret the interface. Strictly speaking, the `Ti.$a` variables are part of the interface, and `Clock1` requires the property that if `T1` returns before `T2`, `T1.$a<=T2.$a`, but neither `Clock` nor `ClockImpl` has this property. So from this point of view `ClockImpl` does not implement `Clock1`.

However, no Spec program can find this out, because there's no way for a program to see `$a` without making another transition, which could be delayed longer in `T1` than in `T2`. So from the viewpoint of what a program can observe, `ClockImpl` does implement `Clock1`, even with multiple readers.

Note that there is an important distinction between the "implements" relation (defined in terms of sets of sequences of actions) and the existence of a conventional abstraction function. The mere fact that a simple abstraction function doesn't exist doesn't mean that the `ClockImpl` can't implement `Clock1`.