Massachusetts Institute of Technology
Department of Electrical Engineering and Computer Science
6.826 Principles of Computer Systems

PROBLEM SET 5

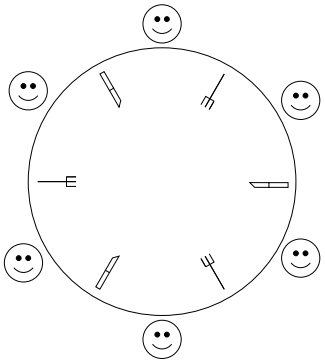
Issued: Thursday, March 7, 2002 **Due: Thursday, March 14, 2002**

There are three problems in this problem set. Please turn in each problem on a separate sheet of paper and write your name on every sheet.

Problem 1. Dining Gourmands[1]

An interesting variant of the dining philosophers problem has recently been discovered. Six gourmands are seated around a table with a large hunk of roast beef in the middle. Forks and knives are arranged as shown in figure. Each gourmand obeys the following algorithm:

1. Grab the closest knife,
2. grab the closest fork,
3. carve and devour a piece of beef,
4. replace the knife and fork.



Six feasting gourmands.

a) Write SPEC code that models this problem.

b) Can deadlock ever occur in this situation? If so, provide an example trace of your SPEC code that results in deadlock. If not, argue that your code will never result in a deadlock.

Problem 2. FIFO Queues

Consider `Buffer` specification on on page 16 of Handout 17.

```
MODULE Buffer[T] EXPORT Produce, Consume =
VAR b : SEQ T := {}
APROC Produce(t) = << b + := {t} >>
APROC Consume() -> T = VAR t | << b # {} => t := b.head; b := b.tail; RET t >>
END Buffer
```

a) Write an implementation of `Buffer` so that buffer has finite length and is represented by the array `b`:

```
MODULE BufferImpl[T] EXPORT Produce, Consume =
CONST N := 1024
TYPE BI = 0 .. N-1
VAR b : BI -> T
```

[1] From Ward, *Computation Structures*

Make sure that you properly synchronize `Produce` and `Consume` actions so that `BufferImpl` implements `Buffer`. Also make sure that your implementation does not deadlock: if there are both `Consume` and `Produce` requests pending then the system should always make progress.

b) Prove that your `BufferImpl` implements `Buffer`.

c) Prove that system always makes progress if there are calls to both `Produce` and `Consume` initiated.

Problem 3. Locks and Data Structures

The following code is an idealization of a linked data structure starting at the node `first`.

```
TYPE Nodes = Int
CONST null:Nodes := 0
VAR first: Nodes;
    next: Nodes -> Nodes
    m: Nodes -> Mutex

PROC advance(n) =
  t = next(n); m(n).rel; m(t).acq; RET t
PROC traverse(n: Nodes,i: Int) =
  m(n).acq;
  DO (~(n=null) /\ (i > 0)) =>
    n := advance(n);
    i := i - 1;
  OD;
  RET n
PROC deleteOne(i: Int) =
    VAR n0 := traverse(first,i) | ~(n0=null) =>
    n1 := next(n0);
    ~(n1=null) =>
       m(n1).acq;
       next(n0) := next(n1);
       m(n1).rel;
       m(n0).rel;
```

a) Draw an initial graph with three nodes reachable from `first` such that subsequent concurrent executions of `deleteOne` procedure can cause system to deadlock. Explain why the deadlock occurs.

b) Draw an initial graph with three nodes reachable from `first` which guarantees that that there can be no deadlocks in subsequent concurrent executions of `deleteOne`. Explain why deadlocks cannot occur.

c) Write in Spec an invariant on `first` and `next` that guarantees that deadlocks cannot occur.

d) Assume that the invariant from c) holds. The intended semantics of `deleteOne` is to implement a removal of one element from a set of nodes reachable from `first`. Does the following property (P) hold:

(P): When k instances of `deleteOne` procedure finish concurrent execution on a long list with i arguments of procedures less than half of the list length (such that all guards that test for `null` succeed) the result is always removal of k elements from the list.

If (P) holds, argue informally why it holds. If (P) does not hold show an example why it does not hold and show how to modify the code so that it holds. Check whether deadlocks can occur in the new implementation.