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

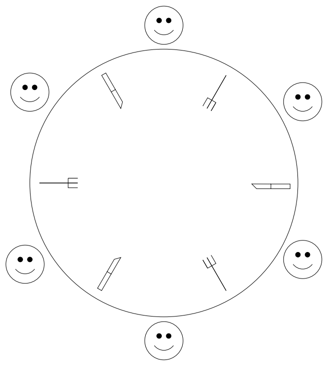
PROBLEM SET 5

Assigned: March 8, 2006 Due: March 15, 2006

Problem 1: Dining Gourmands[1] (30 points)

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: Concurrent Buffers (30 points)

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

a) Write an implementation of `Buffer` where the 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 your system always makes progress if there are calls to both `Produce` and `Consume` initiated.

Problem 3: Consistent Matrices (40 points)

Your company is developing a new system with a core data structure that is a matrix of integers with rows and columns. This matrix has the invariant that the numbers in each column must sum to a given N.

As the application runs, this column will be concurrently accessed by multiple threads that will periodically update the matrix. Each update will be coded to assume that the invariant holds when it starts (if the thread observes values that are inconsistent with this assumption, it may crash catastrophically, which is not acceptable in this system). During each update, the updating thread may write values into the matrix that temporarily cause the invariant to be violated. However, by the time the update completes, it is the responsibility of the thread to restore the invariant.

You realize that you will have to come up with some synchronization mechanism to ensure that each update executes atomically. You decide to use a lock-based mechanism that assigns each element of the matrix to a mutual exclusion lock. Each update then uses these locks to ensure that it executes atomically with respect to all of the other updates.

You are considering the following four alternatives:

- One lock for the entire matrix.
- One lock for each column in the matrix.
- One lock for each row in the matrix.
- One lock for each matrix element.

a) What factors should you consider when determining which lock policy to use?

b) For each alternative, describe a usage scenario for which that lock alternative is clearly superior to the other three scenarios. Explain your reasoning.