Problem Set 5

Instructions: There are 3 problems in this problem set; please turn in each problem on a separate sheet of paper with your name at the top, or you will run the risk of your solutions being lost. Also give the amount of time you spend on each problem.

Problem 1 (Mutual Exclusion)

One day Ben Bitdiddle stumbles upon an arcane implementation of Mutex:

TYPE Choosing = MutexImpl -> Bool
   Number = MutexImpl -> Int

VAR choosing := {}
   number := {}

CLASS MutexImpl EXPORT register, acquire, release =

APROC register() =
   choosing := choosing{self -> False};
   number := number{self -> 0}

PROC acquire() =
VAR others: SET MutexImpl := number.dom - {self} | ~choosing!self => HAVOC [\*]
   %%% begin ‘doorway region’
   choosing := choosing{self -> True}
   number := number{self -> 1 + {j: MutexImpl | choosing!j | number(j)}.max};
   choosing := choosing{self -> False};
   %%% end ‘doorway region’
   DO others.size > 0 =>
      VAR other: MutexImpl := others.choose | ~choosing!self => HAVOC [\*]
         DO choosing(other) = True => SKIP OD;
         DO number(other) # 0 \ %%% lexicographic comparison
            (number(self), self) >= (number(other), other) => SKIP OD;
            others - := other
      OD;

PROC release() =
   ~choosing!self => HAVOC [\*]
   number := number{self -> 0}
END MutexImpl

Ben decides to augment the state of his MutexImpl with a program counter:

TYPE Choosing = MutexImplPC -> Bool
    Number = MutexImplPC -> Int

VAR choosing := {}
    number := {}
    pc: Int := 0

CLASS MutexImplPC EXPORT acquire, release =

APROC register() =
    choosing := choosing{self -> False};
    number := number{self -> 0}

PROC acquire() =
    VAR others: SET MutexImplPC := number.dom - {self},
        other: MutexImplPC |
        ~choosing!number => HAVOC [*]
    DO
        << pc = 0 => choosing := choosing{self -> True}; pc := 1 []
        pc = 1 =>
            number := number{self -> 1 + {j: Int | | number(j)}.max};
            pc := 2 []
        pc = 2 => choosing := choosing{self -> 0}; pc := 3 []
        pc = 3 => others.size > 0 => other := others.choose; pc := 4
            [*] pc := 999 []
        pc = 4 => choosing(other) = True => pc := 4;
            [*] pc := 5 []
        pc = 5 => number(other) # 0 /
            (number(self), self) >= (number(other), other) => pc := 5
            [*] pc := 6 []
        pc = 6 => others - := other; pc := 3
        >>
    OD

PROC release() =
    ~choosing!number => HAVOC [*]
    number := number{self -> 0};

END MutexImplPC

You may assume that all threads call register before any calls to acquire or release.

(a) Ben proposes the following invariant: “suppose process one is the critical region, and process two is in acquire (but not in the doorway), or in the critical region. Then, (number(1), 1) < (number(2), 2).” Carefully state Ben’s invariant in SPEC, and prove it correct.
(b) Using the invariant and a suitable abstraction function, prove that MutexImplPC implements Mutex.

Problem 2 (Mutual Exclusion)$^1$

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 1. 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.

![Figure 1: Six feasting gourmands.](image)

(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 3

Consider the problem of implementing a FIFO buffer for efficiency and performance. Instead of strict FIFO semantics (see Buffer, Handout 17, p. 15), we want to allow the actions Produce and Consume to execute concurrently. Then, we want to enforce the following, “loose FIFO” rule:

if the action Produce(y) begins after Produce(x) ends, and if a later call to Consume returns x, then there exists a call Consume -> y in the traces of the buffer, such that the action Consume -> y did not end before the action Consume -> x began.$^2$

---

$^1$From Ward, Computation Structures.

$^2$Assume that x and y are globally unique.
(a) Write a specification \textbf{FastBuffer} that implements the semantics described above. You may specify a queue with no space constraints.

(b) Now write an efficient implementation of \textbf{FastBuffer}. Design your implementation to address the following issues:

- Suppose \texttt{Produce} and \texttt{Consume} are called many times on your queue, with many calls to each overlapping. Your implementation should allow as many calls to overlap as possible, while maintaining the loose FIFO semantics.
- Your implementation must use at most \( N \) slots in a hashtable, sequence, set, or function domain (take \( N \) to be some globally-defined constant). Hint: one way to do this is with a “cyclic” queue that maintains two pointers to the queue state.

(c) Rigorously prove that your implementation implements your specification.