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

PROBLEM SET 5 SOLUTIONS

Problem 1. Dining Gourmands (30 points)

a)

```
MODULE Gourmands =
CONST N := 6
VAR utensil:  0 .. N-1 -> Mutex
% utensil[2k]   are knives
% utensil[2k+1] are forks

FUNC inc(n: 0 .. N-1) = n = N-1 => RET 0
                        [*] RET n+1

THREAD Gourmand(id: 0 .. N-1) =
VAR knife, fork: 0 .. N-1, eaten : Int := 0 |
  IF x // 2 = 0 =>
    knife := id;
    fork := inc(id);
  [*]
    fork := id;
    knife := inc(id);
  FI
  DO
    % P1
    utensil(knife).acq;
    % P2
    utensil(fork).acq;
    % P3
    eat();
    % P4
    << utensil(knife).rel; utensil(fork).rel >>
  OD

PROC eat() = %chomp!
```

b) Deadlocks cannot happen in this system. We show that at every point in time one of the 6 threads can make progress.

First observe that assignment of `fork` and `knife` variables is correct given the indices of gourmands. Next observe that if there exists a thread at point `P3` or point `P4` these threads can always make progress.

If a thread T is waiting at point `P2` for a fork, then the neighbor thread T' owning that fork must be at point P3 or P4, so T' can make progress. Finally, if a thread T is waiting at point P1 for a knife, then the thread T' owning that knife is at position P2, P3 or P4, so either T' can make progress or its other neighbor $T'' \neq T$ is at P3 or P4 and can thus make progress.

We can also observe that dependencies between threads are paths of length at most three containing neighboring threads and such paths cannot be cyclic because there is no resource that can cause the last thread wait on the first thread in the path.

Problem 2. Concurrent Buffers (30 points)

a)

```
MODULE BoundedBufferImpl[T] EXPORT Produce, Consume =
CONST N := 1024
TYPE BI = 0 .. N-1
VAR b : BI -> T
    last:  BI := 0
    m: Mutex      := m.new();
    cc: Condition := c.new();
    cp: Condition := c.new();

FUNC bufferFull()  -> Bool = RET (last=N)

FUNC bufferEmpty() -> Bool = RET (last=0)

PROC push(t) =
  b(last) := t;
  last + := 1;

PROC pop() -> T =
  VAR t := b(last - 1);
  last - := 1;
  RET t

PROC Produce(t) =
  m.acq;
  DO bufferFull() => cp.wait(m) OD;
  IF bufferEmpty() => cc.broadcast [*] SKIP FI;
  push(t);
  m.rel;

PROC Consume() -> T = VAR t |
  m.acq;
  DO bufferEmpty() => cc.wait(m) OD;
  IF bufferFull() => cp.broadcast [*] SKIP FI
  t := pop();
  m.rel;
  RET t;
```

Note that Produce and Consume invoke broadcast before the buffer is brought to the state where waiting threads can execute. There is no problem with this because according to Handout 17 semantics, broadcast (as well as signal) operations move the threads waiting on the condition variable into the queue for the lock m, but that lock still remains held by the procedure executing broadcast (or signal). It is therefore only after m.rel that procedures woken up can continue execution (if it happens to be the one that gets the lock).

b) We can prove that BoundedBufferImpl implements Buffer using an abstraction function. The abstraction function maps a state with k threads in BoundedBufferImpl into the state with k threads in Buffer.

The buffer content in BoundedBufferImpl is mapped to Buffer.b through function F:

```
FUNC F(b: (BI -> T), last: BI) -> SEQ T =
  IF (last=0) => RET {}
  [*] RET (\ n | n < last => RET b(last - 1 - n))
  FI
```

Let the labelling of program points in the implementation be the following:

```
PROC Produce(t) =
[PB]   m.acq;
       DO bufferFull() => cp.wait(m) OD;
[PCB]  IF bufferEmpty() => cc.broadcast [*] SKIP FI;
       push(t);
       m.rel;
[PCE]

PROC Consume() -> T = VAR t |
[CB]   m.acq;
       DO bufferEmpty() => cc.wait(m) OD;
[CCB]  IF bufferFull() => cp.broadcast [*] SKIP FI
       t := pop();
       m.rel;
[CCE]  RET t;
```

It is a property of the Mutex specification that two different threads cannot be simultaneously in any region enclosed by m.acq and m.rel statements. Hence the region between any two m.acq and m.rel statements define a critical region. Moreover, once a thread enters the critical region it can continue to make progress in that critical region and other threads are prevented from entering the critical region.

By examining the Condition specification, we observe that broadcast (as well as signal) does not release the lock at all. On the other hand, wait releases the lock temporarily, but reacquires it by the time it returns. From this we conclude that program points between PCB and PCE and between CCB and CCE form two critical regions. Next, we observe that the critical regions protect the buffer in the appropriate way:

1. Buffer is not accessed from outside the critical regions. In this case, buffer data structure is implemented in BoundedBufferImpl with variables b and last. The only place where these variables are accessed is push and pop, both of which are called only from within the critical regions.
2. The code in the critical regions only reads and writes variables that cannot be accessed by another thread outside the critical regions. In this case, the only variables accessed are b,last, as well as the local variable p that is inaccessible from outside the thread.

These two conditions are sufficient to guarantee that we may transform the critical regions into atomic regions without increasing the set of external traces. The result is the following.

```
PROC Produce(t) =
[PB]   m.acq;
       DO bufferFull() => cp.wait(m) OD;
[PCB]  << IF bufferEmpty() => cc.broadcast [*] SKIP FI;
       push(t);
       m.rel; >>
[PCE]

PROC Consume() -> T = VAR t |
[CB]   m.acq;
       DO bufferEmpty() => cc.wait(m) OD;
[CCB]  << IF bufferFull() => cp.broadcast [*] SKIP FI
       t := pop();
       m.rel; >>
[CCE]  RET t;
```

We can now essentially ignore the code in `Produce`, except for the `push` procedure call, and we can ignore the code in `Consume`, except for the `pop` procedure call, because the other parts of the code do not change the image of the state under the function `F`. Function `F` specifies the mapping of the buffer variables. We just need to specify the mapping of program counters. This mapping has the same form for every thread. Let the labelling of program points in the specification be the following:

```
MODULE Buffer[T] EXPORT Produce, Consume =
VAR b : SEQ T := {}
APROC Produce(t) = [P1] << b := {t} + b; >> [P2]
APROC Consume() -> T = VAR t | [C1] << b # {} => t := b.head; b := b.tail; RET t >> [C2]
END Buffer
```

We map the state at `[PB]` in `Produce` to `[P1]`. All program points between `[PB]` and `[PCB]` as well as `[PCB]` itself are also mapped to `[P1]`. Implementation states with program counter `PCE` are mapped to states with program counter `[P2]`. We proceed similarly for mapping of program points in `Consume`: points from `[CB]` up to and including `[CCB]` are mapped to `[C1]` whereas point `CCE` and the point after it are mapped to `[C2]`.

For the simulation relation, we need to show that the initial state of the buffer data structure is mapped to the empty sequence, and that for every external step in the implementation there exists a corresponding step in the specification. The base case is easy because initially

$$\texttt{last} = 0$$

so `F` returns `{}`.

For the initial and final actions for the invocation of `Produce` and `Consume` in the implementation, there exist corresponding initial and final actions in the specification.

For the atomic action from `[PCB]` to `[PCE]` in `BoundedBufferImpl` the corresponding action is the invocation of the body of `Buffer.Produce`. Let the initial state in `BoundedBufferImpl` have values

$$\texttt{last}, \texttt{b}$$

After executing `Produce` the concrete state components have values

$$\texttt{last+1}, \texttt{b\{last->t\}}$$

So we need to show that

```
{t}+F(b,last) = F(b{last->t},last+1)
```
(1)

Observe that that if the test for `bufferFull` is true, then the loop exits without releasing the lock, so in those cases the `bufferFull` test can be thought of as belonging to the atomic region from `[PCB]` to `[PCE]`. So `~bufferFull()` holds, which means that it is *not* the case that

$$\texttt{last} = \texttt{N}$$

Therefore, `last=k` where $0 \leq \texttt{k} \leq \texttt{N} - 1$.

Now, consider `{t}+F(b,last) = F(b{last->t},last+1)`. Expanding the definition of `F`, we have that

```
F(b{last->t},last+1)
= (0..last).rev * b{last->t}
= last * b{last->t} + (0..last-1).rev * b{last->t}
= last * b{last->t} + (0..last-1).rev * b
= {t} + F(b,last)
```

which completes the proof of this part of the simulation relation. You should include mentions of the other parts in your solution too, but they're routine.

Showing that the atomic action of `BoundedBufferImpl.Consume` corresponds to the body of `Buffer.Consume` in our implementation is analogous to the previous proof for `BoundedBufferImpl.Produce` so we omit it.

For all steps other than procedure initiations, terminations, atomic action from `[PCB]` to `[PCE]` and atomic action from `[CCB]` to `[CCE]`, the abstract state does not change. With these actions we can associate empty sequence of transitions in the specification since they have no external labels.

This finishes the sketch of the proof that `BoundedBufferImpl` implements `Buffer`.

c) We require the assumption that the scheduler has the following property: it never produces an infinite trace with the property that: an action inside `Produce` or `Consume` remains enabled infinitely many times, but never executes. This assumption can be stated in a variety of ways, but it was not necessary to make it explicit in your solution.

From the assumption it follows that if one of the `Produce` actions is at point `[PCB]` or a subsequent point, then this `Produce` action will complete. Similarly if some `Consume` action is at point `[CCB]` or later, then that action will eventually complete. Also, if a `Produce` procedure is at a point before `[PCB]` and not waiting on `cp` then at some later point it will either go through the loop and terminate or it will end up waiting at `cp`. Similarly `Consume` action either terminates or ends up waiting on `cc`.

To show that some action eventually completes, assume the opposite: after some time, `Produce` and `Consume` stop completing even though both `Produce` and `Consume` keep initiating. This means that both `Produce` and `Consume` actions follow the path that keeps them within the `DO` loop, otherwise they would terminate. Since the procedures remain in their loops, the state of the buffer does not change. Since new actions keep being initiated and (by fairness) entering the loop, it means that both `bufferFull()` and `bufferEmpty()` return `true`, even though the buffer does not change at all. This is clearly impossible since the same buffer of positive capacity cannot be both full and empty. We arrived at a contradiction, which means that the assumption that no action completes is wrong.

Problem 3. Consistent Matrices (40 points)

a) The two major factors to consider are *performance* and *maintainability*: the system, as a whole, will have certain throughput requirements; and it has to be *correct* (both initially and in its maintenance phase). Not surprisingly, these factors often happen to be in direct conflict. In particular, fewer locks guarding more data mean that it's easier to prove that the implementation has *safety* and *liveness* guarantees, but the implementation will also admit less concurrency. It turns out that safety usually won't be the issue, but liveness can be trickier to guarantee. More locks enable more concurrency (and thus higher throughput, but probably not better latency), at the cost of increased engineering effort required for creating correct programs.

Another factor to consider is the amount of time needed to acquire a lock. It might be always slow in absolute terms; it might be slow in contention-free situations; or it might be relatively fast. If it is slow, alternatives which minimize the number of locks acquired are favorable.

We consider each alternative.

- One lock for matrix: This is easiest to implement, and it serializes access to the matrix. If *acquiring locks is always expensive*, and *each update updates many entries all across the matrix at once*, then this alternative is best; it is especially so if coding time is at a premium, because you just don't need to worry about deadlock.
- One lock for each column: This alternative is best if clients tend to *update a large number of entries in a column at once*, and if more concurrency is needed than in the One Big Lock case. Note that a suitable deadlock-free locking policy is still relatively easy to formulate: we just need to always lock columns starting, say, from the lowest-numbered column. Maintaining the invariant is also easy with this policy; it just needs to be ensured before any lock is released. Updating multiple entries in one column is also quite nice here, because we just need to update the sum once.
- One lock for each row: This alternative is best if clients tend to *update a large number of entries in a row at once*, and if more concurrency is needed than in the One Big Lock case. Greater engineering effort may be required than in one-lock-per-column in order to maintain the invariant, so the need for concurrency should sufficiently outweigh the engineering cost. Deadlock can be avoided by using a strategy similar to that described in the one-lock-per-column case.
- One lock for each element: In principle, this would permit the most concurrency, in exchange for implementation overhead and lock acquisition time. This is best when writes occur in *random-access patterns* all over the matrix, and if *locking overhead is low.*