

v4: typos fixed in hw7; no change in exam2.
v3: updated both hw7 and exam2.

Homework 7 (can be done in pairs)

The program on the next page defines n threads executing the black-white bakery algorithm. (Algorithm 2 in Taubenfeld's paper, *The Black-White Bakery Algorithm and related bounded-space, adaptive, local-spinning and FIFO algorithms*, <http://www.cs.tau.ac.il/~afek/gadi.pdf>.)

For each assertion below, prove or disprove whether the program satisfies the assertion (i and j range over $[1..n]$):

- $Inv A_0$, where
 $A_0 : \text{not } ((i \neq j) \text{ and } (t_i \text{ at } 12) \text{ and } (t_j \text{ at } 12))$
- $Inv A_1$, where
 $A_1 : \text{num}_i \text{ in } [0..n]$
- $P_1 : (t_j \text{ at } 2) \text{ leads-to } (t_j \text{ at } 12)$

If you disprove, provide a counter-example evolution.

If you prove, provide an assertional proof along the lines of our bakery algorithm proof, as follows:

1. Define relevant auxiliary variables and/or functions, including the (hypothetical) global request queue, passed, ahead-of, progress metric, etc.
2. Then give the "key" assertions, that is, the assertions which (in your opinion) capture the crux of the algorithm. Motivate/explain the key assertions (1-3 three sentences per assertion). Do not prove the key assertions yet.
3. Prove that the key assertions imply the desired properties $Inv A_0$, $Inv A_1$, and P_1 .
4. Now prove the key assertions. First, list any helper assertions that you need (for the Hoare-triples) of your proof. Prove the key assertion, i.e., give the Hoare-triples that establish it. Prove any non-obvious helper assertions.

The intention is that steps 1-3 convince the reader, and step 4 serves as an appendix.

Homework 7 continued

```
system BlackWhiteBakery(int n) {
  ia {n > 0}
  B ← 0; W ← 1; // black/white
  bit color ← W;
  boolean[1..n] choosing ← false;
  bit[1..n] mycolor;
  int[1..n] num ← 0;
  Thread*[1..n] t;
  for (i in [1..n]) ti ← startThread(loopOn(i));
  return mysid;

  function loopOn(int i) {
    bit[1..n] xc; int[1..n] xn; bit yc; // local variables
1  • choosingi ← true; // beginning of doorway
2  • mycolori ← color;
    for j in [1..n]
3    • [xcj, xnj] ← [mycolorj, numj];
4    • numi ← 1 + max(xnj:j in [1..n], xcj = mycolori);
5    • choosingi ← false; // end of doorway
    for (j in [1..n]) {
6      • await (not choosingj);
7      • if (mycolorj = mycolori)
8        • await ((numj = 0) or ([numj, j] ≥ [numi, i]) or (mycolorj ≠ mycolori));
9        else {
10         do { • [xcj, xnj] ← [mycolorj, numj];
11            • yc ← color;
12            } while (not ((xnj = 0) or (mycolori ≠ yc) or (xcj = mycolori)));
13          }
    }
    // critical section
12 • if (mycolori = black)
    color ← white;
    else color ← black;
13 • numi ← 0;
  }

  atomicity assumption {the “•”s are the only control points}
  // Every atomic step starts at a “•” and ends at the next “•” encountered.
  // For example, an execution of 10 ends either at 9 or at 12.

  progress assumption {weak fairness of every thread except at statement 1}
} // BlackWhiteBakery
```

Exam 2 (to be done individually)

The distributed timestamp-based lock implementation, `LockTsDist`, consists of component systems attached a a fifo channel. Each component system executes `LockTs` and exchanges messages with unbounded timestamps.

The next page shows a modified `LockTs`, referred to as `XLockTs`, that uses messages with modulo- N timestamps as follows.

- It has a variable status, indicating whether the system is thinking, hungry, or eating. This is used in the input assumptions to rule out invalid inputs and ensure that each system has at most one pending request. (Thus the service inverse is not needed.)
- It has a helper function `unMod(.)` which you have to define. It takes a modulo- N timestamp and returns an unbounded timestamp. When the system receives a message with a unbounded timestamp, say t , it replaces t by `unMod(mod(N , t))` and then processes the message as usual. (Thus the system can only access the modulo- N value of t .)

For each assertion below, prove or disprove whether the program satisfies the assertion (i and j range over ADDR):

- $Inv A_0$, where
 $A_0 : \text{set}(j: v[j].\text{status} = E).\text{size} \leq 1$
- $P_1 : (v[i].\text{status} = E \text{ leads-to } v[i].\text{status} = T) \Rightarrow (v[j].\text{status} = H \text{ leads-to } v[j].\text{status} = E)$

If you disprove, show that for any reasonable instantiation of `unMod(.)` and N , there is a counter-example evolution.

If you prove, proceed as in the sliding window protocol analysis (chapter 4):

1. Define function `unMod(.)`. In addition to the argument, it should depend on local variables.
2. Develop appropriate “correct-interpretation” predicates, i.e., upper and lower bounds that a received unbounded timestamp must satisfy to ensure that `unMod(.)` obtains the correct unbounded timestamp.
3. Obtain a lower bound on N and develop predicates, say D , that imply the correct-interpretation predicates and satisfies the invariance rule.

Exam 2 continued

```
system XLockTs(Set ADDR, ADDR aL, Sys* cL, int N) { // component system of LockTsDist
  ia {distinctNonNull(cL,mysid) and N ≥ ??}
  (types, helper functions, variables as in LockTs)

  T ← 1; H ← 2; E ← 3; // thinking, hungry, eating
  status ← T;
  return mysid;
  // end main

  input mysid.acq() {
    ia {status = T}
    mutex.acq();
    status ← H;
    ⟨as in LockTs⟩
    • condAcqRet.wait();
    while (not (reqQ.front() = [reqTs,aL] and lessReqQHts())) {
      condAcqRet.signal();
      • condAcqRet.wait();
    }
    status ← E;
    mutex.rel();
    return;
  }

  input mysid.rel() {
    ia {status = E}
    mutex.acq();
    status ← T;
    ⟨as in LockTs⟩
    mutex.rel();
    return;
  }

  function doRx() {
    while (true) {
      • Seq msg ← cL.rx();
      ia {msg in union(ReqMsg,AckMsg,RelMsg)}
      mutex.acq();
      if (xmsg[0] in set(REQ,ACK))
        msg[1] ← unMod(mod(N,msg[1]));
      ⟨process msg as in LockTs⟩
      mutex.rel();
    }
  }
}

atomicity assumption {as indicated by the three ‘•’s}

progress assumption {weak fairness of threads;
                    strong fairness of condAcqRet.wait}

} // end XLockTs
```
