

-
- 2 problems. 6 pages (including this one).
 - Due Friday December 19 at 5:00 pm. Slip it under my office door or email it to me.
 - Your answer will be judged on accuracy, readability, and elegance.
 - Write neatly or type (font size 10 or more). Use only letter-size sheets with reasonable margins and spacing. Use only one side.
 - Hand in only your final answer. Keep your rough work.
-

NOTE:

- “ $\Box X$ holds” versus “ X satisfies invariance rule”
“ X is invariant”, or “ $\Box X$ holds”, means that X holds at every state of every execution, i.e., at every reachable state.
“ X satisfies invariance rule” means that X holds initially and, for any state (reachable or not) satisfying X and any event, executing the event in the state results in a state that satisfies X . So this implies that $\Box X$ holds, but the converse is not true.
- “ $P \rightsquigarrow Q$ holds” versus “ $P \rightsquigarrow Q$ satisfies leads-to rule”
“ $P \rightsquigarrow Q$ holds” means that for every complete execution, if a state s of the execution satisfies $P \wedge \neg Q$ then there is a later state in the execution that satisfies Q .
“ $P \rightsquigarrow Q$ satisfies leads-to rule” means that for any state s (reachable or not) satisfying $P \wedge \neg Q$, there is an event e subject to weak fairness such that e is enabled in s and its execution results in a state satisfying Q , and the execution of any other enabled event in s results in a state satisfying $P \vee Q$. So this implies that $P \rightsquigarrow Q$ and P unless Q hold, but the converse is not true. (Also note that because the event e can depend on the state s , this rule is a bit more general than the leads-to via event e rule.)

Problem 1

The distributed readers-writers problem is the readers-writers problem with the additional constraint that processes can interact only by passing messages over reliable (fifo) channels. On the next page is an attempted solution based on logical timestamps. Here are conventions followed in the program:

- For notational convenience, we allow an lc event to do multiple outputs to different systems.
- The type $0..N-1$ consists of integers $0, 1, \dots, N-1$.
- The `Msg` type is a list of message tuples. Each tuple is just like a record, e.g., $(REQw, \text{int } t)$ is a record with two fields, a constant field `REQw` and an integer field `t`.
- For any type X , the type `Sequence X` defines sequences of elements of X . For any variable q of type `Sequence X`, we have the following:
 - `empty(q)`: returns `true` if q is empty, otherwise returns `false`.
 - `remv(q)`: removes the message from head of q and returns it; it is undefined if q is empty.
 - `apnd(q, m)`: appends m to the tail of q .

Problem 1a Does system `XX` satisfy $\Box S_0$, where

$$S_0 : [\forall 0..N-1 j, k, j \neq k :: \neg (u[j].st = Ew \wedge u[k].st \in \{Ew, Er\})]$$

- If you answer yes, provide a list of predicates, say Y , such that (the conjunction of) the predicates satisfies the invariance rule and implies S_0 . **Just give Y and define any auxiliary variables used in Y .** Do not give any details of how Y satisfies these conditions.
- If you answer no, provide a counter-example execution.

Problem 1b Does system `XX` satisfy P_0 , where

$$P_0 : [\forall 0..N-1 i :: u[i].st \in \{Hr, Hw\} \rightsquigarrow u[i].st \in \{Er, Ew\}]$$

- If you answer yes, provide a total-ordering (F, \prec) where F is a function of the system variables (and any auxiliary variables you define) such that each of the following satisfies the leads-to rule:
 - $u[i].st \in \{Hr, Hw\} \wedge F = j \succ 0 \rightsquigarrow u[i].st \in \{Er, Ew\} \vee F \prec j$
 - $u[i].st \in \{Hr, Hw\} \wedge F = 0 \rightsquigarrow u[i].st \in \{Er, Ew\}$

Just give (F, \prec) and define any auxiliary variables used in F . Do not give the details of how (F, \prec) satisfies the conditions.

- If you answer no, provide a counter-example execution.

```

system-program XX( int N ) {
  type Msg = { (REQR, int t), (REQw, int t), (REL, int t), (ACK, int t) }; // messages
  type Msgq = Sequence Msg;
  for every i in 0..N-1 do { Process u[i] := StartProcess(X(i)) }
}

system-program X(int i) {
  type Status = (T,Hr,Hw,Er,Ew); // thinking, hungry-read, hungry-write, eating-read, eating-write
  Status st := T;
  int c := 0; // logical clock of u[i]
  int[N] req := 0; // req[j] is 0 or timestamp of last received req from u[j]
  boolean[N] wreq; // wreq[j] is true if last received req from u[j] was wreq
  int[N] ack := 0; // ack[j] is last received timestamp from u[j]
  Msgq[N] inq := ⟨ ⟩; // inq[j] is the channel from u[j] to u[i]

  xc-event snd(0..N-1 j, Msg m) { // append message m to tail of inq[j]
    ec true
    ac apnd(inq[j], m)
  }

  lc-event rec(0..N-1 j) { // receive message m from u[j]
    ec ¬empty(inq[j]) // message at head of inq[j]
    ac Msg m := remv(inq[j]);
    if m = (REQR, t) ∨ m = (REQw, t) then {
      req[j] := t;
      ack[j] := t;
      c := max(c,t)+1;
      u[j].snd(i, (ACK, c) );
      if m = (REQw, t) then wreq[j] := true else wreq[j] := false ;
    }
    else if m = (REL, t) then {
      req[j] := 0;
      ack[j] := t;
      c := max(c,t)+1
    }
    else if m = (ACK, t) then {
      ack[j] := t;
      c := max(c,t)+1
    }
  }

  lc-event bHr() { // become hungry-read
    ec st = T
    ac st := Hr;
    c := c+1;
    req[i] := c;
    wreq[i] := false;
    for j in 0..N-1, j≠i do { u[j].snd(i, (REQR, c) ) };
  }
}

```

```

lc-event bHw() { // become hungry-write
  ec st = T
  ac st := Hw;
  c := c+1;
  req[i] := c;
  wreq[i] := true;
  for j in 0..N-1, j≠i do { u[j].snd(i, (REQw, c) ) };
}

lc-event bE() { // become eating-read
  ec st = Hr ∧
  [∀0..N-1 j, j≠i :: req[i]<ack[j] ∧
  ( req[j]=0 ∨ req[i]<req[j] ∨ (req[i]=req[j] ∧ i<j) ∨ ¬ wreq[j] )
  ]
  ac st := Er
}

lc-event bE() { // become eating-write
  ec st = Hw ∧
  [∀0..N-1 j, j≠i :: req[i]<ack[j] ∧
  ( req[j]=0 ∨ req[i]<req[j] ∨ (req[i]=req[j] ∧ i<j) )
  ]
  ac st := Ew
}

lc-event bT() { // become thinking
  ec st = Er ∨ st = Ew
  ac st := T;
  c := c+1;
  req[i] := 0
  for j in 0..N-1, j≠i do { u[j].snd(i, (REL, c) ) };
}

progress-assumptions {
  Wfair(rec(j), bEr(), bEw, bT()) // bHr(), bHw() has no fairness
}
}

```

Problem 2

The generalized mutual exclusion problem [EWD 625] involves N processes, each placed at a vertex of an undirected graph. Each process cycles between thinking, hungry, and eating. The goal is to ensure that adjacent processes do not eat simultaneously, and that every hungry process eventually eats.

An attempted solution [EWD 625] is described on the next page. It associates with each edge a (binary) semaphore and imposes a total ordering on the edges. A hungry process eats only after acquiring (via P operations) the semaphores of its incident edges in increasing order. When a process stops eating, it releases (via V operations) all the semaphores of its incident edges (in any order).

The program uses the following conventions:

- The graph has N vertices, identified $0, \dots, N - 1$, and M edges, identified $0, \dots, M - 1$. N and M are greater than zero.
- $\text{edgs}(i)$, for any vertex i , denotes the set of edges incident on i .
- $\text{opp}(i, x)$, for vertex i and edge x in $\text{edgs}(i)$, denotes the vertex at the other end of x .
- $\text{adj}(i)$, for vertex i and edge x in $\text{edgs}(i)$, denotes the set of vertices adjacent to i , i.e., $\text{adj}(i) = \{\text{opp}(i, j) : j \in \text{edgs}(i)\}$.
- The underlying platform ensures atomicity of the semaphore operations, P and V . It does not ensure atomicity of anything else.

```
system-program X(N, M, edgs) { // N,M,edgs define an undirected finite graph
// initialization and shared variables
Semaphore[M] lck := 1 ; // lck[x] is 1 iff edge x is free
for i in 0..N-1 do Process u[i] := StartProcess(code(i)) ;
// end initialization

function code(int i) {
  forever do {
    1: "noncritical section";
    2: for p in edgs(i) in increasing order do { // entry code, p local to u[i]
    3:   P(lck[p])
      };
    4: no-op; // "critical section"
    5: for p in edgs(i) do { // exit code, p local to u[i]
    6:   V(lck[p]);
      }
  }
} // end code(i)

progress assumptions { Wfair(*) }
}
```

Problem 2a Does $X()$ satisfy $\square S_0$, where

$S_0 : [\forall 0..N-1 j, k, k \in \text{adj}(j) :: \neg (u[j] \text{ at } 4 \wedge u[k] \text{ at } 4)]$

- If you answer yes, provide a list of predicates, say Y , such that (the conjunction of) the predicates satisfies the invariance rule and implies S_0 . **Just give Y and define any auxiliary variables used in Y . Do not give any details of how Y satisfies these conditions.**
- If you answer no, provide a counter-example execution.

Problem 2b Does system X satisfy

$P_0 : [\forall 0..N-1 i :: u[i] \text{ at } 2 \rightsquigarrow u[i] \text{ at } 4]$

- If you answer yes, provide a total-ordering (F, \prec) where F is a function of the system variables (and any auxiliary variables you define) such that each of P_1 and P_2 given below satisfies the leads-to rule:

$P_1 : u[i].st = H \wedge F = j \succ 0 \rightsquigarrow u[i].st = E \vee F \prec j$

$P_2 : u[i].st = H \wedge F = 0 \rightsquigarrow u[i].st = E$

Just give (F, \prec) and any auxiliary variables used in F . Do not give the details of how (F, \prec) satisfies the conditions.

- If you answer no, provide a counter-example execution.

Problem 2c Repeat problem 2b but now assume that every semaphore obeys the additional (safety) property that waiting processes are served in fifo order.

Problem 2d Repeat problem 2b with the following changes:

- Assume every semaphore P operation has strong fairness.
- If you answer yes, provide a total-ordering (F, \prec) such that the P_3 satisfies the leads-to rule and P_4 holds (instead of P_1 and P_2):

$P_3 : u[i].st = H \wedge F = j \rightsquigarrow u[i].st = E \vee F \succ j$

P_4 Strong fairness for semaphore P operation implies that F cannot increase without bound.