Fall 2015 **EECS 338 Assignment 3 Solns**

G. Ozsoyoglu *Concurrent Programming with Monitors; 140 points*

(100 pts) **1. Priority-based Searchers/Inserters/Deleters Problem without starvation: Monitor-based solution.** Three types of processes, namely, *searchers, inserters*, and *deleters* share access to a singly linked list L, and perform search, insert, or delete operations, respectively. The list L does not have duplicate values.

a) *Searchers* merely search the list L, and report success (i.e., item searched is in L) or no-success (i.e., item searched is not in L) to a log file. Hence they can execute concurrently with each other.

b) *Inserters* add new items to the end of the list L, and report success (i.e., item is not in L, and successfully inserted into L) or no-success (i.e., item is already in L, and no insertion takes place) to a log file. Insertions must be mutually exclusive to preclude two *inserters* from inserting new items at about the same time. However, one insert can proceed in parallel with any number of searches.

c) *Deleters* remove items from anywhere in the list, and report success (i.e., the item is found in L and deleted) or no-success (i.e., item is not in L, and could not be deleted) to a log file. At most one *deleter* can access the list L at a time, and the deletion must be mutually exclusive with searches and insertions.

d) ***Initial start.*** *Searcher*, *inserter*, and *deleter* processes are initially launched as follows. A user process that needs a search/insertion/deletion operation to the list L first *forks* a process, and then, in the forked process, performs an *execv* into a *searcher/ inserter/deleter* process.

e) ***Log maintenance.*** Upon start, each *searcher/inserter/deleter* writes to a log file, recording the time of insertion, process id, process type (i.e., *searcher, inserter*, or *deleter*), and the item that is being searched/inserted/deleted.

f) ***Termination.*** Upon successful or unsuccessful completion, each *searcher/inserter/deleter* writes to the same log file, recording the time and the result of its execution.

g) ***Priority-based service between three types.*** *Searchers*, *inserters*, and *deleters* perform their search, insert, delete operations, respectively, ***on a priority basis*** (not on a first-come-first-serve (FCFS) basis) between separate process types (i.e., *searchers, inserters, deleters)* as follows. *Searchers* search with the highest priority; *inserters* insert with the second highest priority, and *deleters* delete with the lowest priority.

h) ***FCFS service within a single type.*** Processes of the same type are serviced FCFS. As an example, among multiple *inserters*, the order of insertions into L is FCFS. Similarly, among multiple *deleters*, the order of deletions into L is FCFS. Note that, among *searchers*, while the start of search among searchers is FCFS, due to concurrent *searcher* execution, the completions of multiple searchers may not be FCFS.

i) ***Starvation avoidance.*** In addition to the above priority-based search/insert/delete operations, the following ***starvation-avoidance rule*** is enforced.

   - ***After 10 consecutive searchers search the list L***, if there is at least one waiting *inserter* or *deleter* then newly arriving *searchers, inserters, deleters* are blocked until (a) all currently waiting *inserters* are first serviced FCFS, and, then (b) all currently waiting *deleters* are serviced FCFS. Then, the standard priority-based service between process types and the FCFS service within a process type resume.

You are to specify a monitor-based algorithm to synchronize *searcher, inserter* and *deleter* processes.

Make sure to

- Explain your algorithm and monitor procedures briefly.
- State any assumptions you make in your solution.
- Specify the initial states of your monitor variables.
- Specify explicitly the condition variable implementation alternative you use in terms of the order in which signaling- and signaled-processes execute.
- Specify explicitly the monitor instance creation as well as the processes' monitor calls.
- Do not bother specifying algorithms for sequential tasks: simply specify a well-defined function/procedure (i.e., one with well-defined input/output/functional specification).

Monitor cond variable alternative rule: *Signaled process waits until signaling process blocks or leaves the monitor.*

```
type SearchInsertDelete=monitor
cond sWait, iWait, dWait, SServiceStart; sBlock;
int sPassingCnt, sPassedCnt, sWaitCnt, iWaitCnt, dWaitCnt, iStarvationServiceCnt, dStarvationServiceCnt;
boolean StarvationService, iPassing, dPassing; siEmptying

procedure entry sEnter ( )
{if (dPassing or StarvationService)
     {sWaitCnt++; sWait.wait; sWaitCnt−−}     // Wait when a deleter is deleting or starvation service is active.
 if ((sPassingCnt+sPassedCnt) ≤10) sPassingCnt++;                    // Either pass or start starvation service.
 else if (iWaitCnt=0 and dWaitCnt=0) {sPassedCnt:=0; sPassingCnt++}  // No ins.s/del.s.  Reset starvation count.
     else {StarvationService:=True; sPassedCnt:=0;                   // Initiate starvation service, but wait for
         if (iPassing or sPassingCnt>0) {siEmptying:=True; SServiceStart.wait}     // passing searchers/inserter.
         iStarvationServiceCnt:=iWaitCnt;  iWaitCnt:=0;
         dStarvationServiceCnt:=dWaitCnt; dWaitCnt:=0;
         if iStarvationServiceCnt≠0 iWait.signal                     // Start starvation service.
         else dWait.signal;
         sBlock.wait}}                                               // Block self.


procedure entry sExit ( )
{sPassingCnt−−;
 if siEmptying                   // Once passing searchers/inserter emptied, signal starvation service.
       { if (sPassingCnt=0 and ¬iPassing) {siEmptying:=False; SServiceStart.signal}}
else {sPassedCnt++;    // Normal service mode.
     if (sPassingCnt=0 and ¬iPassing and dWaitCnt≠0) dWait.signal}
}
```

```
procedure entry iEnter ( )
{if (StarvationService or dPassing or iPassing)   //Wait when a deleter/ inserter is passing or starv. service is active.
        {iWaitCnt++; iWait.wait; iWaitCnt−−}
 iPassing:=True}
```

```
procedure entry iExit ( )
{iPassing:=False;
 if (siEmptying and sPassingCnt=0) {siEmptying:=False; SServiceStart.signal} }
                                   // Once passing searchers/inserter emptied, signal starvation service.
 else if StarvationService                          // In starvation service mode.
      {iStarvationServiceCnt−−;
       if (iStarvationServiceCnt≠0) iWait.signal
       else if (dStarvationServiceCnt≠0) dWait.signal
          else {StarvationService:=False;           // End of starvation service.
                sBlock.signal;                      // Initiate normal service.
                for i from 1 to 9 do sWait.signal; iWait.signal} }
    else {iWait.signal;                             // Normal service mode.
          if (sPassingCnt=0 and iWaitCnt=0 and dWaitCnt≠0) dWait.signal}}
```

```
procedure entry dEnter ( )
{if (StarvationService or dPassing or iPassing or sPassingCnt>1)   //Wait for regular or starvation service.
        {dWaitCnt++; dWait.wait; dWaitCnt−−}
 dPassing:=True}



procedure entry dExit ( )
{dPassing:=False;
 if StarvationService
      {dStarvationServiceCnt−−;
       if (dStarvationServiceCnt≠0) dWait.signal
       else {StarvationService:=False;                       // End of starvation service.
             sBlock.signal;                                  // Initiate normal service.
             for i from 1 to 9 do sWait.signal; iWait.signal} }
 else {iWait.signal;                                         // Normal service mode.
      if (sPassingCnt=0 and iWaitCnt=0 and dWaitCnt≠0) dWait.signal}}


//Monitor Initialization
begin sPassingCnt:=0; sPassedCnt:=0; sWaitCnt:=0; iWaitCnt:=0; dWaitCnt:=0;
      iPassing:=False; dPassing:=False;  iStarvationServiceCnt:=0; dStarvationServiceCnt:=0;
      StarvationService:=False; siEmptying:=False;
End
```

Monitor Instance Declaration:
… var *MySearchInsertDelete:SearchInsertDelete;*

SEARCHER USE:
…..
MySearchInsertDelete.sEnter ();
SEARCH-AND-LOG-RESULTS (l, item);
MySearchInsertDelete.sExit ();

INSERTER USE:
…..
MySearchInsertDelete.iEnter ();
SEARCH-INSERT-AND-LOG-RESULTS (l, item);
MySearchInsertDelete.iExit ();

DELETER USE:
…..
MySearchInsertDelete.dEnter ();
SEARCH-DELETE-AND-LOG-RESULTS (l, item);
MySearchInsertDelete.dExit ();

(40 pts) **2. Four-of-a-Kind Problem** is defined as follows.

- There is a deck of 24 cards, split into 6 different kinds, 4 cards of each kind.
- There are 4 players (i.e., processes) $P_i, 0 \leq i \leq 3$; each player can hold 4 cards.
- Between each pair of adjacent (i.e., seated next to each other) players, there is a pile of cards.
- The game begins by
  - someone dealing four cards to each player, and putting two cards on the pile between each pair of adjacent players, and
  - P_0 starting the game. If P_0 has four-of-a-kind, P_0 wins. Whoever gets four-of-a-kind first wins.
- Players take turns to play clockwise. That is, P_0 plays, P_1 plays, P_2 plays, P_3 plays, P_0 plays, etc.
- Each player behaves as follows.
  - So long as no one has won, keep playing.
  - If it is my turn and no one has won:
    - Check for Four-of-a-Kind. If yes, claim victory. Otherwise discard a card into the pile on the right; pick up a card from the pile on the left; and, check again: If Four-of-a-Kind, claim victory; otherwise revise turn so that the next player plays and wait for your turn.
- There are no ties; when a player has claimed victory, all other players stop (when their turns to play come up).
- You are to a monitor-based solution to the Four-of-a-Kind problem.

Make sure to

- Explain your algorithm and monitor procedures briefly.
- State any assumptions you make in your solution.
- Specify the initial states of your monitor variables.
- Specify explicitly the condition variable implementation alternative you use in terms of the order in which signaling- and signaled-processes execute.
- Specify explicitly the monitor instance creation as well as the players' monitor calls.
- Do not bother specifying algorithms for sequential tasks: simply specify a well-defined function/procedure (i.e., one with well-defined input/output/functional specification).

Monitor cond variable alternative rule: *Signaled process waits until signaling process blocks or leaves the monitor.*

type *FourofaKind=**monitor***

enumerated *card: {kind1a, kind1b, kind1c, kind1d, kind2a, kind2b, kind2c, kind2d, …, kind6a, kind6b, kind6c, kind6d}*

(int, *card**)** **array** hand[0..3, 0..3];* ***// Each of the four players can have up to 4 cards at hand***

// at any time during the play.

(int, *card**)** **array** pile[0..3, 0..2];*

int *turn;*

boolean *GameWon;*

```
boolean function entry Play (int i)
{if turn≠i  a.wait;
 if GameWon return False;
 if FourOfAKind(i)  {Print ("I, player", i, "win!"); GameWon:=True; return False}
 else {DiscardCardToRightPile (pile[(i,], hand[i, ]);
      PickUpCardFromLeftPile (pile[(i+1)mod 4,], hand[i, ]);
      if FourOfAKind(i) {Print ("I, player", i, "win!"); GameWon:=True; return False} }
 turn:=(i+1) mod 4; a.signal; return True}

boolean function FourofaKind (int i) {…}  //Returns True if all four cards in hand [i,*] are the same kind,
                                          // else returns False.
procedure DiscardCardToRightPile (pile[,], hand[, ]) {..}

procedure PickUpCardFromLeftPile (pile[], hand[,]) {..}

procedure Initialize-Hands-and-Piles-Randomly (pile[,], hand[,]) {…}
                                                                  .
begin Initialize-Hands-and-Piles-Randomly (pile[][], hand[][]){…};        //Monitor initialization
      turn:=0; GameWon:=False end.
```

Monitor Instance Declaration:
var *MyFourofaKind: FourofaKind;*

PLAYER i USE:
{…
while *Play.MyFourofaKind(i)* **do** *no-op;*
 … }