```
Sarah Whelan
slw96
10/8/2015
Assignment 3

Concurrent Programming with Monitors

1. Priority-based Searchers/Inserters/Deleters Problem without starvation:
Monitor-based solution.

Alternatives:
pi issues x.wait and pj issues x.signal
1. pj waits until pi leaves monitor (or blocks at a wait statement).
2. pi waits until pj leaves monitor (or blocks at a wait statement).
3. pj exits monitor immediately; pi resumes.

I am using alternative 2 such that the signaled waits until the signaler leaves the
monitor (or blocks at a wait statement).

type SearchInsertDelete = monitor;

var inserting:boolean;
var deleting:boolean;
var sPassingCount:int;

var sPassedCount:int;
var sWaitCount:int;
var iWaitCount:int;
var dWaitCount:int;

var sBlocked:boolean;
var iBlocked:boolean;

var search:condition;
var insert:condition;
var delete:condition;
var starvation:condition;

process entry SearcherEnter(){
    if(sBlocked || deleting || (sPassingCount + sPassedCount) == 10){
        sWaitCount++;
        search.wait;
        sWaitCount--;
    }
    sPassingCount++;
}

process entry SearcherExit(){
    sPassingCount--;
    sPassedCount++;

    // Should we go into starvation mode?
    if((sPassedCount + sPasssingCount) == 10){
        sBlocked = true;
```

```
53          // If there are other searchers still passing fall out and let the last one deal
54          if(sPassingCount <= 0){
55              sPassedCount = 0;
56              if(inserting){
57                  // If there is an inserter still inserting wait it out
58                  insertingBeforeStarvation = true;
59                  starvation.wait;
60              }
61              if(iWaitCount > 0){
62                  iStarvationCount = iWaitCount;
63                  insert.signal;
64              } else if(dWaitCount > 0){
65                  iBlocked = true;
66                  dStarvationCount = dWaitCount;
67                  delete.signal;
68              } else {
69                  // Nothing was waiting
70                  sBlocked = false;
71                  if(sWaitCount > 0){
72                      search.signal;
73                  }
74              }
75          }
76      } else {
77          // Normal Operation
78          if(iWaitCount <=0 && sWaitCount <= 0 && !inserting && sPassingCount <= 0){
79              delete.signal;
80          } else {
81              search.signal;
82              if(!inserting){
83                  insert.signal;
84              }
85          }
86      }
87
88  }
89
90  process entry InserterEntry(){
91      if(iBlocked || inserting || deleting){
92          iWaitCount++;
93          insert.wait;
94          iWaitCount--;
95      }
96      inserting = true;
97  }
98
99  process entry InserterExit(){
100     inserting = false;
101     if(insertingBeforeStarvation){
102         // if this was the inserter inserting while attempting to enter starvation mode
            start starvation mode
103         starvation.signal
104         // and leave
105     } else {
```

```
106             if(sBlocked){
107                 // Starvation Mode
108                 iStarvationCount--;
109                 if(iStarvationCount == 0){
110                     iBlocked = true;
111 
112                     if(dWaitCount > 0){
113                         // This assumes that we are to allow the deleters that came to wait
                            during the inserters, during starvation mode,
114                         // also get to go during the starvation service (ie not just the
                            deleters that were waiting when the searchers reached 10).
115                         // If we only wanted the deleters that were waiting at the 10
                            searcher count we'd put this line before the 10th searcher signaled
                            insert
116                         dStarvationCount = dWaitCount;
117                         delete.signal;
118                     } else {
119                         // End starvation mode
120                         sBlocked = false;
121                         iBlocked = false;
122                         search.signal;
123                         insert.signal;
124                     }
125                 } else {
126                     // Not finished with waiting inserters
127                     insert.signal;
128                 }
129             } else {
130                 // Normal Operation
131                 if(iWaitCount > 0 || sWaitCount > 0){
132                     search.signal;
133                     insert.signal;
134                 } else if(sPassingCount <= 0 && sWaitCount <= 0){
135                     delete.signal;
136                 }
137             }
138         }
139     }
140 
141     process entry DeleterEntry(){
142         if(sPassingCount > 0 || inserting){
143             dWaitCount++;
144             delete.wait;
145             dWaitCount--;
146         }
147         deleting = true;
148     }
149 
150     process entry DeleterExit(){
151         deleting = false;
152         // if We are in starvation mode
153         if(sBlocked && iBlocked){
154             dStarvationCount--;
155             if(dStarvationCount > 0){
```

```
156                 delete.signal;
157             } else {
158                 // No more deleters for starvation mode end starvation mode
159                 sBlocked = false;
160                 iBlocked = false;
161                 if(sWaitCount > 0 || iWaitCount > 0){
162                     search.signal;
163                     insert.signal;
164                 } else {
165                     delete.signal;
166                 }
167             }
168         } else {
169             // Normal Mode
170             if(sWaitCount > 0 || iWaitCount > 0){
171                 search.signal;
172                 insert.signal;
173             } else {
174                 delete.signal;
175             }
176         }
177     }
178
179     // Initialize variables
180     begin
181     sWaitCount = 0;
182     iWaitCount = 0;
183     dWaitCount = 0;
184     inserting = false;
185     deleting = false;
186     sPassingCount = 0;
187     sPassedCount = 0;
188     sBlocked = false;
189     iBlocked = false;
190     end
191
192     // The processes using the monitor as defined above:
193
194     // Declare and initialize the shared monitor to be used by each forked process
195     var monitor:SearchInsertDelete;
196
197     process searcher(L, item){
198         monitor.SearcherEntry();
199         SEARCH-AND-LOG-RESULTS(L, item);
200         monitor.SearcherExit();
201     }
202
203     process inserter(L, item){
204         monitor.InserterEntry();
205         INSERT-AND-LOG-RESULTS(L, item);
206         monitor.InserterExit();
207     }
208
209     process deleter(L, item){
```

```
210         monitor.DeleterEntry();
211         DELETE-AND-LOG-RESULTS(L, item);
212         monitor.DeleterExit();
213     }
214
215     2. Four-of-a-Kind Problem
216
217     Alternatives:
218     pi issues x.wait and pj issues x.signal
219     1. pj waits until pi leaves monitor (or blocks at a wait statement).
220     2. pi waits until pj leaves monitor (or blocks at a wait statement).
221     3. pj exits monitor immediately; pi resumes.
222
223     I am using alternative 2 such that the signaled waits until the signaler leaves the
        monitor (or blocks at a wait statement).
224
225     type FourOfAKind = monitor;
226
227     var turnId:int;
228     var gameWon:boolean;
229
230     // A deck of 24 cards split into 6 different kinds 4 cards of each kind
231     enumerated card: {
232         1a, 1b, 1c, 1d,
233         2a, 2b, 2c, 2d,
234         3a, 3b, 3c, 3d,
235         4a, 4b, 4c, 4d,
236         5a, 5b, 5c, 5d,
237         6a, 6b, 6c, 6d
238     }
239
240     // An array of arrays of cards for the player's hands
241     card[][] hands;
242
243     // An array of arrays of cards for the discard/pickup piles
244     card[][] piles;
245
246     var turn:condition;
247
248     process entry boolean play(i){
249         if(!gameWon){
250             if(turnId != i){
251                 turn.wait;
252             }
253             if(turnId == i){
254                 if(FOUR-OF-A-KIND(hands[i])){
255                     PRINT("I player " + i + "win!");
256                     gameWon = true;
257                 } else {
258                     DISCARD-TO-PILE(piles[i], hands[i]);
259                     PICK-UP-CARD-FROM-PILE(piles[(i+1)mod4], hands[i]);
260                     if(FOUR-OF-A-KIND(hands[i])){
261                         PRINT("I player " + i + "win!");
262                         gameWon = true;
```

```
263                 } else {
264                     turn = (turn + 1) mod 4;
265                 }
266             }
267         }
268         turn.signal;
269         return !gameWon;
270     }
271     return false; //don't keep looping
272 }
273 
274 process boolean FOUR-OF-A-KIND(card[] hand){
275     // returns if hand (an array of cards of length four) is a four of a kind
276 }
277 
278 process DISCARD-TO-PILE(card[] pile, card[] hand){
279     // Takes a card from hand and adds it to pile must handle the re-sizing of the array
280     // All these are pointers and if the pile here is changed the shared variable
    (piles) is also changed
281 }
282 
283 process PICK-UP-CARD-FROM-PILE(card[] pile, card[] hand){
284     // Takes a card from pile and adds it to hand hand should have 4 spots so no
    resizing there but may want to make the pile array smaller
285     // All these are pointers and if the pile here is changed the shared variable
    (piles) is also changed
286 }
287 
288 process entry DEAL(){
289     // Deal out the cards to the hands and piles
290     // Does not use any condition variables but does use hands and piles which are
    monitor variables so this goes here.
291 }
292 
293 // Initialize variables
294 begin
295 turnId = 0;
296 gameWon = false;
297 // Initialize the array to have four arrays of length four of cards
298 hands = new card[4][4];
299 // Initialize the array to have four arrays of length two of cards
300 piles = new card[4][2];
301 end
302 
303 // The shared monitor for the player processes
304 var monitor:FourOfAKind;
305 
306 // The parent "game" process must "deal" the cards and give values of cards to the
    hands and piles arrays
307 monitor.DEAL();
308 
309 // Then the parent process will fork four children that will use the following process
    to play the game
310 
```

```
311  process player(i){
312      while(monitor.play(i));
313  }
```