Tema 2

Tema se realizeaza in echipa de maxim 3 studenti. Tema se va trimite prin e-mail catre responsabilul de laborator fie ca arhiva zip cu fisiere incluzand raspunsurile si sursele implementarilor, fie ca link catre un repository, avand ca subiect **TEMA 2 TPM**. In e-mailul trimis se va preciza componenta completa a echipei. Termenul strict de trimitere a temei: **6 ianuarie** (inclusiv).
Finalizarea evaluarii temei se va face intr-o sesiune online sambata 15 ianuarie (to be confirmed), ce va fi programata in ultima saptamana inainte de sesiune, la care vor participa toti membrii echipei.

Orice incercare de frauda in rezolvarea temei va fi penalizata cu depunctarea totala a tuturor membrilor echipei.

1. (5 puncte) Se da urmatoarea implementare de referinta pentru algoritmul de lista coarse-grained in care protejarea operatiilor asupra listei se realizeaza printr-un lock singular membru al listei utilizat de fiecare metoda: CoarseList.java. Identificati cate o linie de cod din metodele *add* si *remove* care corespunde punctelor de linearizare pentru situatiile:
a) adaugare element cu succes in lista;
b) esec la adaugare element in lista;
c) stergere element cu succes din lista;
d) esec la stergere element din lista.
Argumentati raspunsul.

2. Se considera exemplul de coada limitata lock-based discutat in cadrul cursului cu metodele *enq* si *deq* descrise ca in pseudocodul de mai jos.

```
1  public class BoundedQueue<T> {
2    ReentrantLock enqLock, deqLock;
3    AtomicInteger size;
4    Node head, tail;
5    int capacity;
6    Condition notFullCondition, notEmptyCondition;
7
8    public BoundedQueue(int capacity) {
9      this.capacity = capacity;
10     this.head = new Node(null);
11     this.tail = head;
12     this.size = new AtomicInteger(0);
13     this.enqLock = new ReentrantLock();
```

```
14    this.notFullCondition = enqLock.newCondition();
15    this.deqLock = new ReentrantLock();
16    this.notEmptyCondition = deqLock.newCondition();
17  }
18
19
20  public void enq(T x) {
21          boolean mustWakeDequeuers = false;
22
23          enqLock.lock();
24          try {
25                  while (size.get() == capacity) {
26                          notFullCondition.await();
27           }
28                  Node e = new Node(x);
29                  tail.next = e;
30                  tail = tail.next;
31                  if (size.getAndIncrement() == 0) {
32                          mustWakeDequeuers = true;
33           }
34          } finally {
35                  enqLock.unlock();
36          }
37
38          if (mustWakeDequeuers) {
39                  deqLock.lock();
40                  try {
41                  notEmptyCondition.signalAll();
```

```
42                      } finally {
43                      deqLock.unlock();
44              }
45              }
46  }
47
48  public T deq() {
49              boolean mustWakeEnqueuers = false;
50              T v;
51
52              deqLock.lock();
53              try {
54                      while (head.next == null) {
55                              notEmptyCondition.await();
56               }
57                      v = head.next.value;
58                      head = head.next;
59                      if (size.getAndDecrement() == capacity) {
60                              mustWakeEnqueuers = true;
61               }
62              } finally {
63                      deqLock.unlock();
64              }
65
66              if (mustWakeEnqueuers) {
67                      enqLock.lock();
68                      try {
69                              notFullCondition.signalAll();
```

```
70                      } finally {
71                              enqLock.unlock();
72                      }
73              }
74
75              return v;
76
77      }
78
79      protected class Node {
80
81        public T value;
82        public Node next;
83
84        public Node(T x) {
85          value = x;
86          next = null;
87        }
88      }
89
90 }
```

a) (5 puncte) Observam ca membrul *size* este de tip AtomicInteger, ceea ce presupune ca metodele de incrementare, respectiv decrementare sunt atomice. In aceste conditii, in pseudocodul de mai sus:

Grupele TPM1 si TPM2: Mai este necesara in metoda *enq* plasarea *size.getAndIncrement()* in cadrul sectiunii protejate de *enqLock*? Argumentati.

Grupele TPM3 si TPM4: Mai este necesara in metoda *deq* plasarea *size.getAndDecrement()* in cadrul sectiunii protejate de *deqLock*? Argumentati.

b) (5 puncte) Presupunem ca in clasa interna *Node* ar exista un membru ReentrantLock *nodelock* (similar cu lacatul din structura unui nod individual din lista fine-grained), si ca ne folosim de acest lacat din nodul *head* in locul *enqLock* si respectiv de lacatul din nodul *tail* in locul *deqLock*, ca mai jos:

```
1  public void enq(T x) {
2          boolean mustWakeDequeuers = false;
3
4          head.nodelock.lock();
5          try {
6                  while (size.get() == capacity) {
7                          notFullCondition.await();
8           }
9                  Node e = new Node(x);
10                 tail.next = e;
11                 tail = tail.next;
12                 if (size.getAndIncrement() == 0) {
13                         mustWakeDequeuers = true;
14          }
15         } finally {
16                 head.nodelock.unlock();
17         }
18
19         if (mustWakeDequeuers) {
20                 tail.nodelock.lock();
21                 try {
22                 notEmptyCondition.signalAll();
23                 } finally {
24                 tail.nodelock.unlock();
25         }
26         }
27 }
28
```

```
29  public T deq() {
30          boolean mustWakeEnqueuers = false;
31          T v;
32
33          tail.nodelock.lock();
34          try {
35                  while (head.next == null) {
36                          notEmptyCondition.await();
37           }
38                  v = head.next.value;
39                  head = head.next;
40                  if (size.getAndDecrement() == capacity) {
41                          mustWakeEnqueuers = true;
42           }
43          } finally {
44                  tail.nodelock.unlock();
45          }
46
47          if (mustWakeEnqueuers) {
48                  head.nodelock.lock();
49                  try {
50                          notFullCondition.signalAll();
51                  } finally {
52                          head.nodelock.unlock();
53                  }
54          }
55
56          return v;
```

```
57
58  }
59
60  protected class Node {
61
62    public T value;
63    public Node next;
64    public ReentrantLock nodelock;
65
66    public Node(T x) {
67      value = x;
68      next = null;
69      nodelock = new ReentrantLock();
70    }
71  }
```

Grupele TPM1 si TPM2: Va mai functiona in acest caz metoda *enq* corect pastrand caracterul FIFO al cozii? Argumentati.

Grupele TPM3 si TPM4: Va mai functiona in acest caz metoda *deq* corect pastrand caracterul FIFO al cozii? Argumentati.

c) (7 puncte) Presupunem ca in loc de utilizarea conditiilor *notFullCondition* si *notEmptyCondition*, si a flagurilor *mustWakeDequers*, respectiv *mustWakeEnqueuers*, pentru notificari intre threaduri, metodele *enq* si *deq* se vor folosi pur si simplu de o operatie de spinning, ca mai jos:

```
1  public void enq(T x) {
2          boolean mustWakeDequeuers = false;
3
4          enqLock.lock();
5          try {
6                  while (size.get() == capacity) {}; //spinning
7                  Node e = new Node(x);
```

```
 8                      tail.next = e;
 9                      tail = tail.next;
10                      size.getAndIncrement();
11              } finally {
12                      enqLock.unlock();
13              }
14      }
15
16      public T deq() {
17              boolean mustWakeEnqueuers = false;
18              T v;
19
20              deqLock.lock();
21              try {
22                      while (head.next == null) {}; //spinning
23                      v = head.next.value;
24                      head = head.next;
25                      size.getAndDecrement();
26              } finally {
27                      return v;
28                      deqLock.unlock();
29              }
30
31      }
```

Va mai functiona in acest caz algoritmul pentru coada corect (ignorand scaderile in performanta)? Ce s-ar intampla daca s-ar amesteca cele doua abordari in modul urmator?:

```
1  public void enq(T x) {
2               boolean mustWakeDequeuers = false;
```

```
3
4           enqLock.lock();
5           try {
6                   while (size.get() == capacity) {
7                           notFullCondition.await();
8            }
9                   Node e = new Node(x);
10                  tail.next = e;
11                  tail = tail.next;
12                  if (size.getAndIncrement() == 0) {
13                          mustWakeDequeuers = true;
14           }
15          } finally {
16                  enqLock.unlock();
17          }
18
19          if (mustWakeDequeuers) {
20                  deqLock.lock();
21                  try {
22                  notEmptyCondition.signalAll();
23                  } finally {
24                  deqLock.unlock();
25          }
26          }
27  }
28
29  public T deq() {
30          boolean mustWakeEnqueuers = false;
```

```
31          T v;
32
33          deqLock.lock();
34          try {
35                  if (head.next == null) {
36                          notEmptyCondition.await();
37           }
38
39                  while (size.get() == 0) {}; //spinning
40
41                  v = head.next.value;
42                  head = head.next;
43                  if (size.getAndDecrement() == capacity) {
44                          mustWakeEnqueuers = true;
45           }
46          } finally {
47                  deqLock.unlock();
48          }
49
50          if (mustWakeEnqueuers) {
51                  enqLock.lock();
52                  try {
53                          notFullCondition.signalAll();
54                  } finally {
55                          enqLock.unlock();
56                  }
57          }
58
```

```
59          return v;
60
61  }
```

d) (3 puncte) Se da pseudocodul de mai jos pentru o coada lock-based de aceasta data in varianta nelimitata. Este necesar ca verificarea pentru coada nevida din metoda *deq()* sa fie neaparat plasata in sectiunea protejata prin lock sau ar putea fi plasata si in afara sectiunii protejate prin lock? Argumentati.

```
1  public class UnboundedQueue<T> {
2   ReentrantLock enqLock, deqLock;
3   Node head, tail;
4
5
6   public void enq (T value) {
7           enqLock.lock();
8           try {
9                   Node newNode = new Node(value);
10                  tail.next = newNode;
11                  tail = newNode;
12          } finally {
13                  enqLock().unlock();
14          }
15  }
16
17  public T deq() throws Exception {
18          T result;
19          deqLock.lock();
20          try {
21                  if (head.next == null) {
22                          System.out.println("queue empty");
23                          throw new Exception();
```

```
24                      }
25                      result = head.next.value;
26                      head = head.next;
27              } finally {
28                      deqLock().unlock();
29              }
30              return result;
31      }
32
33      public UnboundedQueue() {
34       head = new Node(null);
35       tail = head;
36       enqLock = new ReentrantLock();
37       deqLock = new ReentrantLock();
38      }
39
40      protected class Node {
41              public T value;
42              public Node next;
43              public Node (T value) {
44                      this.value = value;
45                      next = null;
46      }
47 }
```

3. (5 puncte) Adaptati algoritmul de lista optimista prezentat in cadrul cursului si pentru care a fost oferit un exemplu de implementare in cadrul laboratorului (OptimisticList.java) prin introducerea unei versionari, astfel incat operatiile ce implica modificarea listei sa incrementeze la fiecare schimbare un numar de versiune. Incercati sa va folositi de aceasta versiune pentru eficientizarea fazei de validare. Testati implementarea si comparati performanta acesteia cu varianta initiala de lista optimista - se cere evaluarea timpului mediu pentru o operatie de add/remove/contains la un experiment cu 4 thread-uri ce executa aceeasi operatie pe o lista cu 100000 de elemente. Realizati un raport in care sa prezentati

rezultatele comparative si sa explicati pe scurt ideea implementarii.
Se vor considera in evaluare corectitudinea si performanta implementarii realizate.

Hint: Pentru testarea corectitudinii se poate pleca de la un test similar celui propus ca exercitiu in laboratorul 8, reamintit mai jos.
Se considera un set de threaduri concurente care sa adauge intregi (Integer) de la 1 la 100000 in lista, respectiv un set de threaduri concurente care sa elimine intregii pari din lista de la 1 la 25000. Se distribuie elementele adaugate, respectiv cele eliminate in mod egal intre numarul de threaduri folosite in test. Se sincronizeaza startul threadurilor ce elimina elemente astfel incat sa nu porneasca inainte de adaugarea intregilor de la 1 la 25000. Se verifica daca setul de elemente ramase dupa operatii este cel asteptat. Se urmaresc in realizarea testului problemele de sincronizare descrise in curs.
Variatii la aceasta idee de test pot sa apara evident in functie de particularitatile implementarii cu versionare.