Multi-Core Architecture and Systems (0368-4183) Lecturer: Adam Morrison

Home Assignment 3

Due date: **December 25th, 23:59** (submit via moodle)

Submission instructions

- **In this assignment, you can submit either in pairs or individually.** Even if working in a pair, each member should submit the assignment through the moodle.
- Submissions must be PDF, at most 3 pages, minimum 11 pt font.
- A solution submitted t seconds late will have its grade multiplied by $1 - \left(\frac{t}{60 * 60 * 24 * 7}\right)^4$.

1 MCS Lock

Design a version of the MCS lock (Figure 1) in which `unlock` completes in a **constant** number of its steps (and thus, `unlock` completion never depends on the progress of another thread). You may change the code of both `lock` and `unlock`, but **must not** change the structure of a QNode.

Your solution should have the same properties as the original MCS lock:

- *Local spinning.* A thread may only spin-wait on a private variable, not on a globally updated variable like in the TTAS lock.
- *Starvation-freedom.* Any thread trying to acquire the lock eventually succeeds.
- *First-come-first-served.* We can think of the `lock` method as being composed of a wait-free *doorway* section followed by a waiting section. The algorithm guarantees that if thread T completes its doorway before thread T', then T acquires the lock before T'. *Intuitively, you can think of this property as the formal way of saying that threads are queued up in a queue.*
- *Unbounded number of threads.* The algorithm doesn't assume that the number of threads is known.

If you add new global variables, they **may not** be contention hot spots, i.e., written to (with a normal write or an atomic CAS) by every operation. The only global variable allowed to be a contention hot spot is the L variable from the original code.

For a full score, your solution should also maintain the original MCS property that a thread spin-waits on its own node (the node that it receives as the argument to `lock`).

In your write-up, you should (1) describe the idea behind your design; (2) provide pseudo code of the methods you change from the MCS lock; and (3) explain why your design maintains mutual exclusion and starvation-freedom.

Hint: You don't have to worry about memory management of queue nodes; assume that they are garbage collected and you have a fresh one for each `lock` call.

```
1  struct QNode {
2    QNode* next
3    bool locked
4  }
5
6  // Lock initially points to NULL
7  QNode *L = NULL
8
9
10
```

```
11 lock(QNode* n) {
12   n->next = NULL
13
14   QNode* pred = SWAP(&L, n)
15   if (pred != NULL) {
16     n->locked = true
17     pred->next = n
18     while (n->locked) { }
19   }
20 }
```

```
21 unlock(QNode* n) {
22   // gets the same QNode that
23   // was passed to lock()
24   if (n->next == NULL) {
25     if (CAS(&L, n, NULL))
26       return;
27     while (n->next == NULL) { }
28   }
29   n->next->locked = false
30 }
```

Figure 1: MCS lock.

2 Java memory model

In this assignment, you will figure out how to correctly implement the *lazy list* (seen in Lecture #4) in Java.

Background: the lazy list The lazy list data structure is a sorted linked list that implements the standard set operations. The lazy list uses locks to synchronize updates, but allows searches to run concurrently without locking. For full details, see the original paper "A Lazy Concurrent List-Based Set Algorithm" (`http://people.csail.mit.edu/shanir/publications/Lazy_Concurrent.pdf`).

Figure 2 shows a Java implementation of the lazy list data structure; assume that `initialize()` is the constructor: it is always called to initialize the data structure (i.e., it describes the default state of the data structure).

Lazy list vs. JMM Here, we will connect the JMM issues to the high-level correctness of a data structure. For simplicity, we use sequential consistency (SC) as our correctness condition. A data structure implementation is *sequentially consistent* if for every execution E, there exists a legal sequential history S, such that for every thread T, $S|T$ is the sequence of data structure operations performed by T in E.

In the following, assume (1) sequentially consistent hardware and (2) that the value of a `final` field/variable is always the initial value of this field/variable.

1. Show that the Java implementation in Figure 2 is not an SC implementation of a set. To do this, describe an execution E that is valid under the JMM but in which the high-level set operations cannot be totally ordered to obtain a legal sequential history that is compatible with the per-thread order of operations. Explain how the execution E can occur in practice due to compiler transformations.

2. Give a **minimal** set of changes to the code in Figure 2 that will make it be an SC implementation. Your changes must maintain the scalability properties of the original algorithm; for example, protecting the whole list with a coarse-grained lock is not an acceptable solution.

   - Explain why your changes make the set implementation sequentially consistent. Hint: You can rely on the fact that the Lazy List paper proved correctness of the algorithm assuming that reads and writes were sequentially consistent.
   - Show that your proposed changes are minimal. For each change you make, show that if you do not make this change (but keep all others) then an SC violation in the set

is still possible. Explain how this violation could be created in practice by compiler transformations.

```
31 // List node
32 class Node {
33     final int key;  // key is immutable
34     Node next;
35     boolean marked;
36 }
37
38 // Head of list.  Doesn't contain an item.
39 final Node head;
40
41 // Initial list state: head and tail sentinels.
42 void initialize() {
43   head = new Node(−∞); // head.key = −∞
44   head.next = new Node(∞);
45 }
```

```
47 public Node(int key) {  // Node constructor
48     this.key = key;
49     this.next = null;
50     this.marked = false;
51 }
52 boolean contains(int key) {
53     Node curr = head;
54     while (curr.key < key)
55       curr = curr.next;
56     return curr.key == key && !curr.marked;
57 }
58 boolean validate(Node pred, Node curr) {
59   return !pred.marked && !curr.marked &&
60           pred.next == curr;
61 }
```

```
62 boolean add(int key) {
63   while (true) {
64     Node pred = head;
65     Node curr = pred.next;
66     while (curr.key < key) {
67       pred = curr; curr = curr.next;
68     }
69     synchronized(pred) {
70       synchronized(curr) {
71         if (validate(pred, curr)) {
72           if (curr.key == key) {
73             return false;
74           } else {
75             Node node = new Node(key);
76             node.next = curr;
77             pred.next = node;
78             return true;
79 } } } } } }
```

```
80 boolean remove(int key) {
81   while (true) {
82     Node pred = head;
83     Node curr = pred.next;
84     while (curr.key < key) {
85       pred = curr; curr = curr.next;
86     }
87     synchronized(pred) {
88       synchronized(curr) {
89         if (validate(pred, curr)) {
90           if (curr.key != key) {
91             return false;
92           } else {
93             curr.marked = true;
94             pred.next = curr.next;
95             return true;
96 } } } } } }
97
```

Figure 2: Lazy (sorted) linked list.