Assignment-2 ,Concurrent Data Structures

-2018101116
Lakshmi Madhuri

1 MCS Lock

This is different approach for implementing scalable spin locks unlike **Array Based Locks** which is not space sufficient as It requires a known bound n on the maximum number of concurrent threads, and it allocates an array of that size per lock

In MCS Lock, lock is represented as a Linked list of Qnode objects,where Qnode represents either a lock holder or thread waiting for acquiring lock.

Here ,the list is **explicit ,** globally accessible Qnode objects via their next fields. Code:-

```
1.  Class Qnode{
2.      boolean locked = false; //
3.      Qnode next=null;
4.  }
```

Suppose , if a thread wants to acquire a lock ,it has to append its OwnNode to tail of the list. And check if queue is not empty previously then set predecessor node next to current node.

Now spin the thread until its predecessor sets this field to false

```
1.  public void lock() {
2.      QNode qnode = myNode.get();
3.      QNode pred = tail.getAndSet(qnode);
4.      if (pred != null) {
5.          qnode.locked = true;
6.          pred.next = qnode;
7.          // wait until predecessor gives up the lock
8.          while (qnode.locked) {}
9.      }
10.     }
```

Unlock Method():-

To release a lock, the processor must reset the busy-waiting bit of its successor. If the processor has no successor, it sets to nil

Let q be the thread’s current node. To distinguish between these cases, the method applies compareAndSet(q, null) to the tail field. If the call succeeds, then no other thread is trying to acquire the lock, tail is set to null,and the method returns.

```
1.  public void unlock() {
2.          QNode qnode = myNode.get();
3.          if (qnode.next == null) {
4.                  if (tail.compareAndSet(qnode, null))
5.                  return;
6.          // wait until predecessor fills in its next field
7.          while (qnode.next == null) {}
8.          }
9.  }
```

Drawback:-

Releasing a lock requires spinning. Another is that it requires more reads, writes,and compareAndSet() calls than the CLHLock algorithm.

A code will benefit from using MCS lock when there's a high locks contention, i.e., many threads attempting to acquire a lock simultaneously. When using a simple spin lock, all threads are polling a single shared variable, whereas MCS forms a queue of waiting threads, such that each thread is polling on **its predecessor** in the queue. Hence cache coherency is much lighter since waiting is performed "locally".

2 ABA Problem

The *ABA problem* occurs during synchronization: a memory location is read twice and has the same value for both reads. However, another thread has modified the value, performed other work, then modified the value back between the two reads, thereby tricking the first thread into thinking that the value never changed.

Let’s think about a multithreaded scenario when Thread A and Thread B are operating on the same linked list.Suppose threads A,B , both have shared access to linked list.

Linked list is as follows:-

a-->b-->c-->d-->.....

Suppose Thread A operation is ==> delete node a

Thread B operation is ==> delete node b

Both threads are operating concurrently. Suppose,for some reasons Thread A got delayed , meanwhile in that gap Thread B has done its operation of deleting b.

a-->c-->d-->...

Now without acknowledging this , Thread A will delete node a poinintg it’s next to b

a---(got deleted)->b(which was already deleted)

So now, the whole list has been meaningless

This is “**ABA Problem”**

==>One of the solution for this is to tag each atomic reference with a unique stamp using **AtomicStampedReference()** object , which keeps both an object reference and stamp internally using **compareAndSet()** method. We usually keep increasing the value of **stamp** each time we modify the object. The stamp and reference fields can be updated atomically

Comparing and setting AtomicStampedReference :-

The `AtomicStampedReference` class contains a useful method named `compareAndSet()`. The `compareAndSet()` method can compare the reference stored in the `AtomicStampedReference` instance with an expected reference, and the stored stamp with an expected stamp, and if they two references and stamps are the same (not equal as in `equals()` but same as in `==`), then a new reference can be set on the `AtomicStampedReference` instance.

If `compareAndSet()` sets a new reference in the `AtomicStampedReference` the `compareAndSet()` method returns `true`. Otherwise `compareAndSet()` returns `false`.

EXAMPLE:-

```
1. String initialRef   = "initial value referenced";
2. int    initialStamp = 0;
3.
4. AtomicStampedReference<String> atomicStringReference =
5.     new AtomicStampedReference<String>(
6.         initialRef, initialStamp
7.     );
8.
9. String newRef   = "new value referenced";
10.      int    newStamp = initialStamp + 1;
11.
12.      boolean exchanged = atomicStringReference
13.          .compareAndSet(initialRef, newRef, initialStamp, newStamp);
14.      System.out.println("exchanged: " + exchanged);  //true
15.
16.      exchanged = atomicStringReference
17.          .compareAndSet(initialRef, "new string", newStamp, newStamp +
   1);
18.      System.out.println("exchanged: " + exchanged);  //false
19.
20.      exchanged = atomicStringReference
21.          .compareAndSet(newRef, "new string", initialStamp, newStamp +
   1);
22.      System.out.println("exchanged: " + exchanged);  //false
23.
24.      exchanged = atomicStringReference
25.          .compareAndSet(newRef, "new string", newStamp, newStamp + 1);
26.      System.out.println("exchanged: " + exchanged);  //true
```

This example first creates an `AtomicStampedReference` and then uses `compareAndSet()` to swap the reference and stamp.

After the first `compareAndSet()` call the example attempts to swap the reference and stamp two times without success. The first time the `initialRef` is passed as expected reference, but the internally stored reference is `newRef` at this time, so the `compareAndSet()` call fails. The second time the `initialStamp` is passed as the expected stamp, but the internally stored stamp is `newStamp` at this time, so the `compareAndSet()` call fails.

The final `compareAndSet()` call will succeed, because the expected reference is `newRef` and the expected stamp is `newStamp`.