HW2 : SUBHAJIT SAHU : 2018801013

MCS QUEUE LOCK

MCS Queue Lock maintains a linked-list for threads waiting to enter critical section (CS).

Each thread that wants to enter CS joins at the end of the queue, and waits for the thread infront of it to finish its CS. So, it locks itself and asks the thread infront of it, to unlock it when he's done. Atomics instructions are used when updating the shared queue. Corner cases are also takes care of.

As each thread waits (spins) on its own "locked" field, this type of lock is suitable for cache-less NUMA architectures.

CODE: https://repl.it/@wolfram77/mcs-lock#QNode.java

```
class QNode {
boolean locked = false;
QNode next = null;
}
```

CODE: https://repl.it/@wolfram77/mcs-lock#MCSLock.java

```
import java.util.concurrent.*;
import java.util.concurrent.locks.*;
import java.util.concurrent.atomic.*;


// MCS Queue Lock maintains a linked-list for
// threads waiting to enter critical section (CS).
//
// Each thread that wants to enter CS joins at the
// end of the queue, and waits for the thread
// infront of it to finish its CS.
//
// So, it locks itself and asks the thread infront
// of it, to unlock it when he's done. Atomics
// instructions are used when updating the shared
// queue. Corner cases are also takes care of.
//
// As each thread waits (spins) on its own "locked"
// field, this type of lock is suitable for
// cache-less NUMA architectures.
```

```
class MCSLock implements Lock {
AtomicReference<QNode> queue;
ThreadLocal<QNode> node;
// queue: points to the tail
// node: is unique for each thread


public MCSLock() {
queue = new AtomicReference<>(null);
node = new ThreadLocal<>() {
protected QNode initialValue() {
return new QNode();
}
};
}


// 1. When thread wants to access critical
// section, it stands at the end of the
// queue (FIFO).
// 2a. If there is no one in queue, it goes head
// with its critical section.
// 2b. Otherwise, it locks itself and asks the
// thread infront of it to unlock it when its
// done with CS.
@Override
public void lock() {
QNode n = node.get(); // 1
QNode m = queue.getAndSet(n); // 1
if (m != null) { // 2b
n.locked = true; // 2b
m.next = n; // 2b
while(n.locked) Thread.yield(); // 2b
} // 2a
}


// 1. When a thread is done with its critical
// section, it needs to unlock any thread
// standing behind it.
// 2a. If there is a thread standing behind,
// then it unlocks him.
// 2b. Otherwise it tries to mark queue as empty.
```

```
// If no one is joining, it leaves.
// 2c. If there is a thread trying to join the
// queue, it waits until he is done, and then
// unlocks him, and leaves.
@Override
public void unlock() {
QNode n = node.get(); // 1
if (n.next == null) { // 2b
if (queue.compareAndSet(n, null)) // 2b
return; // 2b
while(n.next == null) Thread.yield(); // 2c
} // 2a
n.next.locked = false; // 2a
n.next = null; // 2a
}


@Override
public void lockInterruptibly() throws InterruptedException {
lock();
}


@Override
public boolean tryLock() {
lock();
return true;
}


@Override
public boolean tryLock(long arg0, TimeUnit arg1) throws InterruptedException {
lock();
return true;
}


@Override
public Condition newCondition() {
return null;
}
}
```

CODE: https://repl.it/@wolfram77/mcs-lock#Main.java

```
import java.util.function.*;


class Main {
static MCSLock lock;
static double[] sharedData;
static int SD = 100, CS = 100, TH = 10;
// SD: shared data srray size
// CS: critical section executed per thread
// TH: number of threads


// Critical section updated shared data
// with a random value. If all values
// dont match then it was not executed
// atomically!
static void criticalSection() {
try {
double v = Math.random();
for(int i=0; i<SD; i++) {
if(i % (SD/4)==0) Thread.sleep(1);
sharedData[i] = v + v*sharedData[i];
}
}
catch(InterruptedException e) {}
}


// Checks to see if all values match. If not,
// then critical section was not executed
// atomically.
static boolean criticalSectionWasAtomic() {
double v = sharedData[0];
for(int i=0; i<SD; i++)
if(sharedData[i]!=v) return false;
return true;
}


// Unsafe thread executes CS N times, without
// holding a lock. This can cause CS to be
// executed non-atomically which can be detected.
//
// Safe thread executes CS N times, while
// holding a lock. This allows CS to always be
```

```
// executed atomically which can be verified.
static Thread thread(int n, boolean safe) {
Thread t = new Thread(() -> {
for(int i=0; i<CS; i++) {
if(safe) lock.lock();
criticalSection();
if(safe) lock.unlock();
}
log(n+": done");
});
t.start();
return t;
}


// Tests to see if threads execute critical
// section atomically.
static void testThreads(boolean safe) {
String type = safe? "safe" : "unsafe";
log("Starting "+TH+" "+type+" threads ...");
Thread[] threads = new Thread[TH];
for(int i=0; i<TH; i++)
threads[i] = thread(i, safe);
try {
for(int i=0; i<TH; i++)
threads[i].join();
}
catch(InterruptedException e) {}
boolean atomic = criticalSectionWasAtomic();
log("Critical Section was atomic? "+atomic);
log("");
}


public static void main(String[] args) {
lock = new MCSLock();
sharedData = new double[SD];
testThreads(false);
testThreads(true);
}


static void log(String x) {
System.out.println(x);
}
```

```
}
```

OUTPUT: https://mcs-lock.wolfram77.repl.run

```
Starting 10 unsafe threads ...
0: done
1: done
3: done
4: done
6: done
2: done
8: done
5: done
9: done
7: done
Critical Section was atomic? false

Starting 10 safe threads ...
2: done
1: done
5: done
0: done
4: done
3: done
6: done
7: done
8: done
9: done
Critical Section was atomic? true
```