SFDV3006
Concurrent Programming

Lecture 10 – Wait Free Synchronization

Outline

- Synchronization and its problems and solutions
- CAS
- Wait free and lock free synchronization
- Atomic variables
- Example using atomic variable

Background

- Multiprocessor systems are becoming cheap and commonly available
- There is built in support for multiprocessing on multiprocessing systems as well as single CPU using HyperThreading or dual core
- How to effectively use MP hardware?
  - Applications are designed and structured as threads, each of which runs on a separate processor giving higher throughput
  - The design must ensure that threads actually spend time doing work, rather than waiting for more work or waiting for locks or shared objects.

Synchronization

- Most of the concurrent applications need coordination between threads
  - Example: Producer and Consumer sharing a buffer
  - Example: Thread pools – where tasks are generally executed independently of each other.
- If a thread pool uses a common queue, elements (threads in the queue) need to added or removed to and from the queue. These operations must be thread safe
- The way to solve these problems
  - To coordinate access to shared fields or objects using synchronization
  - With synchronization a thread will have exclusive access to the shared object / variable / resource

Synchronization example

```
public class SynchronizedCounter {
    private int counter = 0;
    public synchronized int getValue(){
        return counter;
    }
    public synchronized int increment() {
        return counter++;
    }
    public synchronized int decrement(){
        return counter--;
    }
}
```

Assume that many threads are using a shared object of this *SynchronizedCounter* class

Dow we really need to use synchronization?

Read-modify-write sequence but atomic since we are using synchronized

How is the performance under low contention and under heavy contention?

Disadvantages of synchronization

- When a synchronization lock is heavily contended, throughput can suffer.
- If a thread holding a lock is delayed (due to some error such as page fault, scheduling delay) then no thread requiring that lock will make progress.
- When a thread is waiting for a lock it cannot do anything else until the lock becomes available
  - This can lead to *priority inversion* – where a high priority task is waiting for low priority task to release the lock.
- When locks are acquired in an inconsistent manner it can lead to a deadlock
- For simple operations such as ++ and -- using locks is a bit too much when all that is needed is managing concurrent updates to individual variables

Solutions to synchronization problems

- *volatile variables*: volatile writes are guaranteed to be visible to other threads.
- *volatile* basically solves the problem with the JMM(Java Memory Model) which allows threads to cache variables in their local registers.
- With volatile the latest updates are immediately flushed to main memory so that all threads will see the latest consistent value.
- Problem:Read-modify-write sequence is still not atomic which means volatile cannot be used to implement a mutex or counter.

Lock contention and performance

- Under heavy contention the performance suffers
  - JVM spends more time dealing with scheduling threads and managing contention and queues of waiting threads
  - It spends less time doing real work such as incrementing and decrementing the values
- One way to improve performance is by using a *ReentrantLock* which was introduced in Java 1.5
- Is there another even better way to improve performance?
- Whether we use a lock using *synchronized* or a *ReentrantLock* we still loose concurrency since the threads are effectively *serialized* – they have to execute one after another

ReentrantLock example

```
Lock lock = new ReentrantLock();

lock.tryLock(); //only acquire if the lock is free

try {
   //critical section
}
catch (Exception ex) {
      //restore any invariants
}
finally {
     lock.unlock()
}
```

ReentrantLock is slightly better than *synchronized* locks

It also gives you better control on acquiring or not acquiring a lock

Also it does not lock all the methods in a shared object

Has better scalability under high contention since it uses hardware techniques such as *test and set* or *compare and swap (CAS)*

Wait free synchronization

- Many modern processors support instructions for the special requirements of multiprocessing
- There is an instruction for updating a shared variable in a way that can either detect or prevent concurrent access from other processors (or threads)
- This instruction is called *CAS* (compare and swap)
- Architectures such as Intel and Sparc implement a primitive called *compare-and-swap*
- On Intel this is *cmpxchg* family of instructions
- On some architectures this is called *testAndSet*
- *Note that these instruction are atomic*

How CAS works?

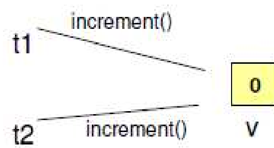
- The CAS operation includes three operands:
  - a memory location *V*
  - the expected old value *A*
  - and the new value *B*
- The processor will *atomically* update the location(V) to the new value(B), if the value that is there matches the old value(A), otherwise it will do nothing.
- In either case, it returns the value that was at that location before the CAS operation

CAS algorithm

- CAS algorithm works as follows:
  - Read a value A from address V
  - Perform a multistep operation such as ++ or - - to get the new value B
  - Now use CAS to change value of V from A to B, CAS succeeds if the value at V has not changed in the meantime (by another thread).
  - if V has value A
    - put value B in V
  - else
    - do nothing
  - return value at V

CAS is implemented in the hardware – its an instruction of the processor

How does CAS handle multiple updates?



One scenario with two concurrent threads updating a shared variable:
There are two threads t1 and t2

Suppose t1 starts first
reads the value 0 at V
goes to do the increment ++ operation

In the meantime t2 increments 0 to 1

When t1 does CAS it finds the value at V has changed (from 0 to 1), so it retries the operation with the latest value.

There is no race condition here nor there is any locking of the shared variable

Lock free and wait free syncrhonization

- Concurrent algorithms based on CAS are called *lock free algorithms* because threads do not ever have to wait for a lock.
- Lock free algorithms require that only some thread always makes progress.
  - Either the CAS succeeds or fails, but in either case is completes in a predictable amount of time.
- *Wait free algorithm*: an algorithm is said to be wait free if every thread will continue to make progress even when there is random delay or failure of other threads.
  - This is not the case with locks.

Non blocking algorithms and their advantages

- Wait free and lock free algorithms are called *nonblocking algorithms*
- Nonblocking algorithms are used widely in operating systems and JVM level for tasks such as thread and process scheduling
- Advantages over lock based algorithms
  - Prevent problems such as priority inversion and deadlocks
  - Contention is less expensive
  - There is high degree of parallelism which leads to better performance

Java and Non blocking algorithms

- Since JDK1.5 it has become possible to write wait free and lock free algorithms in Java.
- *Atomic variable classes* have been introduced in a new package *java.util.concurrent.atomic*
- The atomic variable classes expose a compare-and-set primitive (similar to CAS) which is implemented using the fastest native construct on the platform (CAS / LLSC).
- For example, instead of using a primitive type such as an *int,* we can now use the *AtomicInteger* class
- Also for increment and decrement the normal + and – cannot be used – instead methods have been provided for decrement and increment which use CAS on the hardware via the JVM

Counter using atomic variables

```
import java.util.concurrent.atomic.*;
public class SynchronizedCounter {
    private AtomicInteger counter = new AtomicInteger(0);
    public int getValue(){
        return counter.get();
    }
    public int increment() {
        return counter.incrementAndGet();
    }
    public int decrement(){
        return counter.decrementAndGet();
    }
}
```

What about race conditions for a shared object of this class used by many threads?

Now using atomic operations No need to use synchronized Or ReentrantLocks

How is the performance under low contention and under heavy contention now compared to the version which is on slide 5?