Multicore Programming: Lab 1

Exercise 1.1

Write a Java program that takes as arguments two integers, T and N, and forks T threads that modify a shared volatile integer as follows. Even threads (i.e., threads 0, 2, 4…) will increment the integer N times while odd threads (i.e., threads 1, 3, 5…) will decrement it N times. The program will print the final value of the integer (which should be 0 if T is even, or N otherwise).

Execute the program with N=10'000'000 and T={2,4,8,16}. Report execution times.

Exercise 1.2

Modify the program of 1.1 so that access to the shared integer is protected by a synchronized block. As before, execute the program and report results.

Exercise 1.3

Modify the program of 1.1 so that access to the shared integer is protected by a `ReentrantLock` from the `java.concurrent.util.locks` package. As before, execute the program and report results.

Exercise 1.4

Modify the program of 1.1 so that the shared integer is incremented and decremented atomically in a lock-free manner. You can use an `AtomicInteger` from the `java.concurrent.util.atomic` package. As before, execute the program and report results.

Exercise 1.5

Compare results of the 1.1, 1.2, 1.3 and 1.4. Which programs are correct or incorrect? Which approach is faster? Why?

Exercise 1.6

You have a choice between buying one uniprocessor that executes five zillion instructions per second, or a ten-processor multiprocessor where each processor executes one zillion instructions per second. Using Amdahl's Law, explain how you would decide which to buy for a particular application.

Exercise 1.7 (optional)

Write a Java program that takes as arguments two integers, T and N, and forks T threads that will together search for and print all primes between 1 and N. The program should evenly split the range [1..N] into T sub-ranges assigned to each thread.

Execute the program with N={10'000'000,100'000'000} and T={1,2,4,8,16}. Report execution times.

Exercise 1.8 (optional)

Modify the program of 1.1 so that it uses a shared counter to assign the next number to test to the threads. The shared counter should be protected from concurrent accesses by a monitor (e.g., using synchronized method). As before, execute the program and print the results.

Multicore Programming: Lab 2

Exercise 2.1 (optional)

Implement Lamport's Bakery algorithm and use it in the program of 1.1 to protect the shared integer. As before, execute the program and report results.

Exercise 2.2 (optional)

A group of N Philosophers sits at a round table, where N forks are placed on the table, one between each pair of adjacent philosophers. No philosopher can eat unless he has two forks and he can only use the two forks separating him from his two neighbors. Obviously, adjacent philosophers cannot eat at the same time. Each philosopher alternately eats and sleeps, waiting when necessary for the requisite forks before eating.

Your task is to write code simulating the dining philosophers so that no philosopher starves. You are free to use any Java synchronization constructs.

An obvious protocol that each philosopher might follow is:

```
while (true) {
  grab left fork;
  grab right fork;
  eat;
  release left fork;
  release right fork
  sleep;
}
```

Try to think of a solution which attempts to address potential deadlocks that might occur.

Multicore Programming: Lab 3

Exercise 3.1

Are the following histories linearizable or sequentially consistent? Explain your answers and write the equivalent linearizable/sequential consistent histories where applicable.

1. Read/write register

a. Concurrent threads `A`, `B`, `C`, register `r`.

```
A: r.write(1)
C: r.read()
A: r:void
A: r.write(2)
C: r:2
C: r.read()
B: r.read()
A: r:void
C: r:1
A: r.write(1)
B: r:1
A: r:void
```

b. Concurrent threads `A`, `B`, `C`, register `r`.

```
A: r.write(1)
B: r.read()
A: r:void
A: r.write(2)
A: r:void
A: r.write(1)
B: r:1
C: r.read()
A: r:void
C: r:2
```

2. Stack

We have the following operations:

- push(x) pushes element x on the stack, returns void;
- pop() retrieves an element from the stack;
- empty() returns true if stack is empty and false otherwise.

a. Concurrent threads A, B, C, stack s.

```
C: s.empty()
A: s.push(10)
B: s.pop()
A: s:void
A: s.push(20)
B: s:10
A: s:void
C: s:true
```

b. Concurrent threads A, B, C, stack s.

```
A: s.push(10)
B: s.push(10)
A: s:void
A: s.pop()
B: s:void
B: s.empty()
A: s:10
B: s:true
A: s.pop()
A: s:10
```

3. Queue

We have the following operations:

- enq(x) inserts element x in the queue, returns void;
- deq() retrieves an element from the queue.

a. Concurrent threads A, B, C, queue q.

```
A: q.enq(x)
B: q.enq(y)
A: q:void
B: q:void
A: q.deq()
C: q.deq()
A: q:y
C: q:y
```

Multicore Programming: Lab 4

Exercise 4.1

Write a Java program that takes as arguments two integers, T and N, and forks T threads that collectively increment a shared volatile integer until it reaches (at least) the value N. Increments should be protected by a TAS lock.

Execute the program with N=10'000'000 and T={2,4,8,16}. Report execution times.

Exercise 4.2

Modify the program of 4.1 so that access to the shared integer is protected by a TTAS lock. As before, execute the program and report results.

Exercise 4.3

Modify the program of 4.1 so that the shared integer is incremented atomically in a lock-free manner. As before, execute the program and report results.

Exercise 4.4

Can we implement `getAndIncrement()` with `compareAndSet()`? And the other way around? If so, show how.

Multicore Programming: Lab 5

Exercise 5.1

Write a Java program that takes as arguments two integers, T and N, and forks T producer threads and T consumer threads. The producers will concurrently add N arbitrary objects each in a shared queue. The consumers will concurrently consume the objects from the queue until each of them has consumed N elements. Try to use the following queue implementations: `java.util.LinkedList`, without and with mutual exclusion, and `java.util.Concurrent.ConcurrentLinkedQueue`.

Execute the 3 programs with N=10'000'000 and T=2. Report execution times.

Exercise 5.2

Consider the classical producer-consumer problem: a group of P producer threads and a group of C consumer threads share a bounded circular buffer. If the buffer is not full, producers are allowed to add elements; if the buffer is not empty, consumers can consume elements. Let's assume that the system uses the buffer implementation below, where `head` and `tail` point to the ends of the buffer where the items can be consumed, respectively produced.

```
public class BoundedBuffer {
  int head = 0, tail = 0;
  Object[QSIZE] items;

  public synchronized void put(Object x) {
    while (tail - head == QSIZE)
      this.wait();
    items[tail % QSIZE] = x;
    tail++;
    this.notifyAll();
  }
  public synchronized Object get() {
    while (tail == head)
      this.wait();
    Object x = items[head % QSIZE];
    head++;
    this.notifyAll();
    return x;
  }
}
```

Can this implementation deadlock? And what it we replace calls to `notifyAll()` by calls to `notify()`? Justify your answer.

Exercise 5.3

Consider the bounded lock-free queue below. We assume that integer overflows are not an issue. What is the problem with this implementation?

```
class LockFreeQueue {
  static final int MAX = 256;
  AtomicInteger head, tail;
  int queue[] = new int[MAX];

  public void enq(int v) {
    while (true) {
      int t = tail.get();
      if (t == head.get() + MAX)
        throw new FullQueueException();
      if (tail.compareAndSet(t, t + 1)) {
        queue[t % MAX] = v;
        break;
      }
    }
  }

  public int deq() {
    while (true) {
      int h = head.get();
      if (h == tail.get())
        throw new EmptyQueueException();
      if (head.compareAndSet(h, h + 1)) {
        return queue[h % MAX];
      }
    }
  }
}
```

Exercise 5.4

Consider an unbounded lock-based queue with the following deq() method:

```
public T deq() throws Exception {
  T result;
  deqLock.lock();
  try {
    if (head.next == null) {
      System.out.println("Queue empty");
      throw new Exception();
    }
    result = head.next.value;
    head = head.next;
  } finally {
    deqLock().unlock();
  }
  return result;
}
```

Is it necessary to protect the check for a non-empty queue in the deq() method with a lock or can the check be done outside the locked part? Explain.

Exercise 5.5 (optional)

Write a simple lock-based queue that supports concurrent enqueues and dequeues (i.e., it must be possible for 2 threads to operate simultaneously on both ends of the queue if it is not empty). You are free to choose the design of your implementation (blocking or non-blocking, bounded or unbounded).

Exercise 5.6 (optional)

A cyclic counter is a counter that can be incremented until it reaches an upper limit (specified when creating the counter), after which it rolls over to 0. Implement a lock-free cyclic counter in Java (without using monitors).

Multicore Programming: Lab 6

Exercise 6.1

Consider the following code that implements the well-known "double-checked locking" software design pattern. Try to understand and explain the rationale of this approach. In particular, why is it better than a simpler implementation in which the `getHelper()` method would be synchronized?

```
class Foo {
  private volatile Helper helper = null;
  public Helper getHelper() {
    Helper result = helper;
    if (result == null) {
      synchronized(this) {
        result = helper;
        if (result == null) {
          helper = result = new Helper();
        }
      }
    }
    return result;
  }
  //
  ...
}
```

Exercise 6.2 (optional)

A read-write lock allows either a single writer or multiple readers to execute in a critical section. Provide an implementation of a read-write lock in Java. You can use synchronized methods and the wait/notify mechanism if you wish. The class should provide the 4 methods `lockRead()`, `unlockRead()`, `lockWrite()`, and `unlockWrite()`. This implementation does not need to be FIFO, starvation-free, nor reentrant.

HINT: you might want to keep track of the number of readers and writers.

Exercise 6.3 (optional)

A semaphore[1] is an abstract data type used for controlling access, by multiple processes, to a common resource in a parallel programming environment. A useful way to think of a semaphore is as a record of how many units of a particular resource are available, coupled with operations to safely (i.e., without race conditions) adjust that record as units are required or become free, and, if necessary, wait until a unit of the resource becomes available. Semaphores that allow an arbitrary resource count are called counting semaphores.

[1] https://en.wikipedia.org/wiki/Semaphore_(programming)

Consider the following implementation of a counting semaphore:

```
public class CountingSemaphore {
  private int available;
  public CountingSemaphore(int n) {
    available = n;
  }
  public synchronized void acquire() {
    while(available == 0)
      try { wait(); } catch(InterruptedException e) { }
    available--;
  }
  public synchronized void release() {
    available++;
    notify();
  }
}
```

Propose an equivalent implementation of a counting semaphore that is lock-free.

Multicore Programming: Lab 7

Exercise 7.1

Write a program that takes two random matrices A and B of size N×N and computes their product C = A·C using one task per cell of C. Use the mechanisms provided by the `Future`, `Callable` and `Executors` classes and interfaces.

Execute the program with N={100,1'000}. Report execution times.

Multicore Programming: Lab 8

Exercise 8.1

Modify the last program of 5.1 so that the producers wait for each other on a `CyclicBarrier` before starting adding elements to the queue, and the consumers use a `CountDownLatch` to wait until all producers have finished their insertions before removing elements.

Exercise 8.2 (optional)

Consider the following interface for a Consensus object:

```
public interface Consensus {
  // Propose value v and return agreed-upon value
  Object decide(Object v);
}
```

The consensus requires each thread to start with a non-null input value, given as parameter to the `decide()` method in this simplified interface. All threads must eventually agree on a common input value, which is returned by the `decide()` method.

We can easily implement an algorithm using locks that allows an arbitrary number of threads to reach consensus, as follows:

```
public class LockConsensus implements Consensus {
  Object decision = null;

  synchronized Object decide(Object v) {
    if (decision == null)
      decision = v;
    return decision;
  }
}
```

Can you propose a lock-free consensus algorithm between an arbitrary number of threads with compare-and-swap?