CSE113 – Spring 2021
Midterm #1
Assigned: April 29 (by midnight)
Due: March 6 (by midnight)

Before you begin: This is an asynchronous exam, and you should treat it like such. Please do not discuss the exam with any other student until after the due date. This exam is open book and open notes. These resources should be sufficient to complete the test.

The test is "open internet", to a degree. You are allowed to search online for general concepts. You are not allowed to search for the exact questions. You are especially NOT allowed to ask test questions on forums like StackOverflow, Reddit, or Quora. There is a zero tolerance policy on cheating. The department does monitor these sites (and others). Students have been reported for academic misconduct.

If you have clarifying questions: Please write a question to the teach mailing list, or an instructor only question on Piazza. We will keep a thread of clarifying questions as a Canvas discussion. Please check there before asking your question.

There are 6 questions, each worth equal points. The material is inclusive up to April 29. The test is designed to take 2 hours, assuming that you have spent enough time studying and do not double check your answers (like what would happen in a traditional synchronous exam). You are free to spend as much time as you'd like on the test.

I will provide the MS word doc, as well as well as the PDF. Your submission should be in PDF form, but we are flexible on the format. You can print it out and fill in the test with pencil/paper and scan/photo your solutions. You can work in the MS word document. As long as your solutions are clear to us. Please show your work.

Good luck!

Question 1: *Data dependency graphs and instruction level parallelism*

Consider the following 3-address code:

```
1: r1 = a  + b;
2: r3 = c  * d;
3: r5 = r1 + 5;
4: r2 = e  * f;
5: r4 = 4  + r2;
6: r6 = 7  * r5;
7: r7 = r4 + r3;
8: r8 = r7 * r6
```

a) Draw a data-dependency graph for the program

b) Assume a superscalar processor that can issue 3 instructions at a time and completes them in 1 clock cycle. Find the optimal ordering for such a processor and report how many clock cycles it would take on this machine.

c) Assume a non-superscalar processor with a pipeline with 3 stages. Each cycle can progress an instruction one stage further through the pipeline. This processor has no speculation; if instruction *i* depends on *i'*, then *i* must wait until *i'* has exited the final pipeline stage. Find the optimal order for such a processor and report on how many clock-cycles the program would take.

Question 2: *buggy mutex implementations*

This question asks questions about small modifications to the implementations of Peterson's 2-threaded mutex and Lamport's bakery algorithm. In each case, the modification consists of re-ordering 2 lines in the original (correct) implementation. For each both cases, describe if the mutex can fail, and if so, describe a situation under which the mutex fails to either provide mutual exclusion, or deadlock freedom. Both algorithm specifications are obtained from the textbook (The Art of Multiprocessing Programming). They are in Java, but you can assume shared memory accesses are atomic. The other conceptual mappings should be straightforward.

a) Peterson's algorithm: if lines 8 and 9 are swapped, i.e. the threads determine the victim before setting the flag, is the mutex still correct? If not, describe a situation where a mutex property is violated.

```
1  class Peterson implements Lock {
2    // thread-local index, 0 or 1
3    private boolean[] flag = new boolean[2];
4    private int victim;
5    public void lock() {
6      int i = ThreadID.get();
7      int j = 1 - i;
8      flag[i] = true;            // I'm interested
9      victim = i;                // you go first
10     while (flag[j] && victim == i) {}; // wait
11   }
12   public void unlock() {
13     int i = ThreadID.get();
14     flag[i] = false;           // I'm not interested
15   }
16 }
```

b) Lamport's algorithm: if lines 13 and 14 are swapped i.e. threads determine their label before setting the flag, is the mutex still correct? If not, describe a situation where a mutex property is violated.

```
1  class Bakery implements Lock {
2    boolean[] flag;
3    Label[] label;
4    public Bakery (int n) {
5      flag  = new boolean[n];
6      label = new Label[n];
7      for (int i = 0; i < n; i++) {
8        flag[i] = false; label[i] = 0;
9      }
10   }
11   public void lock() {
12     int i = ThreadID.get();
13     flag[i] = true;
14     label[i] = max(label[0], ...,label[n-1]) + 1;
15     while ((∃k != i)(flag[k] && (label[k],k) << (label[i],i))) {};
16   }
17   public void unlock() {
18     flag[ThreadID.get()] = false;
19   }
20 }
```

Question 3: *Fix concurrent code*

The following C++ code snippet shows 4 functions that are being called concurrently. You have two mutexes that you can use. Use the lock and unlock methods of these mutexes to protect concurrent accesses to the shared data. Use one mutex per function, and use both mutexes. Pay attention to the arguments that are passed to the functions, as well as the scope of variable names!

```
void foo(int *a, int *b, int *c, mutex *m0, mutex *m1) {
    if (b[0] > 0) {
        a[0] = b[0];
    }
    else if (b[0] > -16) {
        a[0] = 128;
        return;
    }
    else {
        a[0] = 256;
    }
   a[0]++;
}

void bar(int *a, int *b, int *c, mutex *m0, mutex *m1) {
    c[0] = a[0];
}

void baz(int *a, int *b, int *c, mutex *m0, mutex *m1) {
    a[0] = b[0]
}

void qux(int *a, int *b, int *c, mutex *m0, mutex *m1) {
   c[0] = b[0] + c[0];
}

int main() {
   int *a = new int[1];
   int *b = new int[1];
   int *c = new int[1];
   mutex m0, m1;

   thread tfoo = thread(foo, a, b, c, &m0, &m1);
   thread tbar = thread(bar, c, c, c, &m0, &m1);
   thread tbaz = thread(baz, a, b, c, &m1, &m0);
   thread tqux = thread(qux, a, b, c, &m0, &m1);

   // assume we join and cleanup memory after this
}
```

Question 4: *Mutex implementation*

Assume you are on a system that only supports bits and atomic bits, with the types called “bit” and “atomic_bit”, respectively. The following atomic RMWs are provided:

```
bit atomic_fetch_and(atomic_bit *b, bit n);

bit atomic_fetch_or(atomic_bit *b, bit n);

bit atomic_fetch_xor(atomic_bit *b, bit n);
```

Each of these instructions, performs the binary op (and, or, xor) on the location pointed to by b, with the value n as the second operand. The operation is performed atomically, i.e., it cannot be interleaved. The value at b, before the operation, is returned.

a) implement an atomic RMW mutex using these operations. It should resemble a lock like the CAS or exchange lock.

b) the architects are willing to make one of the atomic operations much faster than any of the others. You can pick one of the operations, and with that operation, you should implement a “relaxed peeking” optimization to the lock function of your solution in part (a).

Question 5: *More functionality for the CAS lock*

The simple CAS lock is given here as a reference. This question will ask you to provide two different types of locks built on top of the CAS lock. You cannot use any atomic RMW other than CAS.

```
class CAS_mutex {
  public:
    CAS_mutex() {
        flag = 0;
    }

    void lock(int thread_id) {
         int e = 0;
         while (atomic_compare_exchange(&flag, &e, 1) == false) {
             e = 0;
         }
    }

    void unlock(int thread_id) {
          flag.store(0);
    }

   private:
      atomic_int flag;
}
```

a) Reentrant lock: The current CAS lock implementation will deadlock if a thread that already holds the mutex calls the 'lock' function again. A *reentrant* lock has the property that if a thread 't' holds a mutex 'm', then 't' can call 'lock' on mutex 'm' without deadlock.

Your job is to modify the "lock" function of the CAS lock so that it is reentrant, i.e., if it is called by a thread that already holds the mutex, then it will simply return instead of deadlocking. The unlock function needs to only be called a single time. i.e., a thread can call "lock" many times, but unlocks the mutex with one call to "unlock".

b) A fairer CAS lock: The current CAS lock has no fairness guarantees. This is especially problematic if threads are executing concurrently on the same core. Consider a situation where two threads, t0 and t1, are executing on the same core, and both contending for a CAS mutex. It is possible that one of the threads, say t0, continually acquires the mutex, especially if the OS schedules t0 on the core instead of t1.

Develop an addition to the CAS lock which encourages more fair behavior by trying to prevent a single thread from constantly acquiring the mutex over and over again without any thread getting in a chance. You can use the sleep function, yield function and add extra variables to the mutex class.

Question 6: *Sequentially Consistent Executions*

This question consists of 3 small concurrent programs, each containing a question about an execution. You need to determine if the execution is allowed or not under sequential consistency. If it is allowed, give the sequential sequence of object methods calls. If not, provide a detailed description why it is not. That is, show a partial sequential sequence with a description about why it cannot be finished.

The CQueue object refers to a concurrent queue, with enq and deq methods as discussed in class. Dequeuing from an empty queue returns 0. The object is initially empty.

(a)

```
Global variables:
CQueue q0;

Thread 0        Thread 1
q0.enq(1);      int t0 = q0.deq();
q0.enq(2);      int t1 = q0.deq();
q0.enq(3);      int t2 = q0.deq();
q0.enq(4);      int t3 = q0.deq();


At the end of the execution can:
t0 == 1 and t1 == 2 and t2 == 3 and t3 == 0?
```

(b)

```
Global variables:
CQueue q0, q1;

Thread 0        Thread 1      Thread 2
q0.enq(1);      q1.enq(1);    int t0 = q0.deq();
                              int t1 = q1.deq();

At the end of the execution can:
t0 == 1 and t1 == 0?
```

(c)

```
Global variables:
CQueue q0, q1;

Thread 0        Thread 1      Thread 2                Thread 3
q0.enq(1);      q1.enq(1);    int t0 = q0.deq();      int t2 = q1.deq();
                              int t1 = q1.deq();      int t3 = q0.deq();


At the end of the execution can:
t0 == 1 and t1 == 0 and t2 == 1 and t3 == 0?
```