Please print in pen.

Waterloo Student ID Number:

WatIAM/Quest Login UserID:

Midterm - Winter 2019 - SE 350

1. Before you begin, make certain that you have one **2-sided booklet with 11 pages**. You have **110 minutes** to answer as many questions as possible. The number in parentheses at the beginning of each question indicates the number of points for that question. You should read all of the questions before starting the exam, as some of the questions are substantially more time consuming.

2. All solutions must be placed in this booklet. If you need more space to complete an answer, you may be writing too much. However, if you need extra space, use the extra blank page at the end of the exam clearly labeling the question and indicate that you have done so in the original question.

3. Read each question carefully. Make your answers as concise as possible. If there is something in a question that you believe is open to interpretation, then please ask us about it!

Good Luck!

Question	Points Assigned	Points Obtained
1	32	
2	22	
3	22	
4	24	
Total	100	

1. (32 points) True-False and Why? For each question:

- CIRCLE YOUR ANSWER
- One point for correct true-false.
- One point for correct explanation.
- No points for any explanation if true-false is not correct.
- No points for an explanation that exceeds 3 sentences.

1.a. For a fixed number of threads in a uniprocessor, reducing threads' average response time necessarily improves system's throughput.

True False

1.b. For a fixed number of threads in a uniprocessor, improving system's throughput necessarily reduces at least one thread's response time.

True False

1.c. Interrupt-driven I/O is always faster than programmed I/O.

True False

1.d. Hardware and interrupt handler together push interrupted process's registers onto the interrupt stack.

True False

1.e. The stack pointer of the interrupted user-level process is stored on the interrupt stack twice.

True False

1.f. To satisfy safety, kernel system call handler copies arguments of the system call to the kernel memory after validating them.

True False

1.g. Kernel interrupt handler is a thread.

True False

1.h. In fork-join parallelism, the output of a multi-threaded program is not affected by different interleavings of threads' executions.

True False

1.i. To implement mutual exclusion in multiprocessors, hardware must provide atomic load-modify-store instructions.

True False

1.j. To implement mutual exclusion in multiprocessors, hardware must provide instructions to disable and enable interrupts.

True False

1.k. Accessing a variable stored in a thread's individual stack is always thread-safe.

True False

1.l. Disabling interrupts is enough to implement mutual exclusion.

True False

1.m. Starvation implies lack of progress.

True False

1.n. Implementing critical sections and mutual exclusion involves waiting

True False

1.p. A binary semaphore (i.e., a semaphore that only takes values 0 and 1; if the value is 1 and V() is called, the value remains 1) is semantically equivalent to a lock.

True False

1.q. Context switching between two threads belonging to the same process is less expensive than context switching between two threads belonging to two processes.

True False

2. (22 points) Thread Safe Queue. Consider the following multithreaded program.

Note that thread_create_p(thread_t *thread, void *(*routine)(void*), void *args) creates a new thread, that will run routine, which gets args pointer as an argument.

```
const int MAX = 10;

class TSQueue {
   Lock lock;
   int items[MAX];
   int front;
   int nextEmpty;
 public:
   TSQueue() {front = nextEmpty = 0;};
   ~TSQueue() {};
   bool tryInsert(int item);
   bool tryRemove(int *item);
};

bool TSQueue::tryRemove(int *item) {
   bool success = false;
   lock.acquire();
   if (front < nextEmpty) {
      *item = items[front % MAX];
      front++;
      success = true;
   }
   lock.release();
   return success;
}

bool TSQueue::tryInsert(int item) {
   bool success = false;
   lock.acquire();
   if ((nextEmpty - front) < MAX) {
      items[nextEmpty % MAX] = item;
      nextEmpty++;
      success = true;
   }
   lock.release();
   return success;
}

int main(int argc, char **argv) {
   TSQueue *queues[3];
   thread_t workers[3];
   int i, j;
   for (i = 0; i < 3; i++) {
      queues[i] = new TSQueue();
      thread_create_p(&workers[i],
      putSome, queues[i]);
   }

   printf("Let's begin!\n");

   thread_join(workers[0]);

   for (i = 0; i < 3; i++)
      testRemoval(&queues[i], i);

   printf("All done!\n");
}

void *putSome(void *p) {
   TSQueue *tsq = (TSQueue *)p;
   int i;
   for (i = 0; i < 50; i++)
      tsq->tryInsert(i);

   return NULL;
}

void testRemoval(TSQueue *tsq, int q ) {
   int i, item;

   for (i = 0; i < 20; i++) {
      if (tsq->tryRemove(&item))
         printf("Deleted %d:%d\n", q, item);
}
```

2.a. (2 points) Including the main thread, what is the maximum and minimum number of concurrently running threads between printing "Let's begin!" and "All done!"?

2.b. (4 points) In "Deleted 0:x," what are all the possible x's? Why?

2.c. (16 points) True-False and Why? (2 points for T-F and 2 points for explanation)

1. "Deleted 1:0" may not be printed. True False

2. “Deleted 2:10" could be printed. True False

3. Up to 30 items could be inserted and up to 20 items could be removed from queue[1]. True False

4. Items inserted and removed are sorted and sequential. True False

3. (22 points) What the Fork()! Consider the following program. Assume that the compiler and the hardware do not reorder instructions, all instructions are atomic, and calls to `fork` and `thread_create_p` always succeed.

```
void main (int argc, char **argv) {
   int pid = fork(), x = 5;
   if (!pid) {
      x += 5;
   } else {
      pid = fork();
      x += 10;
      if (pid)
         x += 5;
   }
   printf(“%d\n”, x);
}
```

3.a. (4 points) How many different copies of the variable x will be created on memory?

3.b. (6 points) What are all possible outputs in standard output? If there are multiple possibilities, put each in its own box. You may not need all the boxes.

Now, consider the following code. Note that exit(0) terminates the entire process and waitpid(pid) pauses the process until the child process specified by pid has exited.

```
void* f1(void* args) {
    printf("F1: %d\n", *((int*) args));
    return NULL;
}

void* f2(void* args) {
    printf("F2: %d\n",  *((int*) args));
    exit(0);
}

void main (void) {
    int val = 5;
    thread_t myT;
    int pid = fork();

    If(!pid) {
        pthread_create_p(&myT, f2, &val);
    } else {
        val += 5;
        waitpid(pid);
        pthread_create_p(&myT, f1, &val);
        thread_join(myT);
    }

    printf("Val: %d\n", val);
    exit(0);
}
```

3.c. (4 points) Including the original process and thread, what is the maximum and minimum number of created processes and threads?

3.d. (8 points) List all possible outputs in standard output. If there are multiple possibilities, put each in its own box. You may not need all the boxes.

4. (24 points) Starvation. Consider the following implementation of blocking bounded queue. Suppose that `MAX` is 20 and we iteratively create threads that call `insert` and threads that call `remove`. Assume that the compiler and the hardware do not reorder instructions and also assuming that all instructions are atomic.

```
class BBQ {
 private:
   Lock lock;
   CV itemAdded, itemRemoved;
   int items[MAX];
   int front, nextEmpty;
 public:
   BBQ() {front = nextEmpty = 0;};
   ~BBQ() {};
   void insert(int item);
   int remove();
};

void BBQ::insert(int item) {
   lock.acquire();
   while ((nextEmpty - front) == MAX) {
      itemRemoved.wait(&lock);
   }
   items[nextEmpty % MAX] = item;
   nextEmpty++;
   itemAdded.signal();
   lock.release();
}

int BBQ::remove() {
   int item;
   lock.acquire();
   while (front == nextEmpty) {
      itemAdded.wait(&lock);
   }
   item = items[front % MAX];
   if ((nextEmpty - front) == MAX)
      itemRemoved.signal();
   front++;
   lock.release();
   return item;
}
```

4.a. (4 points) Does the 10th removing thread that acquires the lock always remove the 10th item inserted? Why?

4.b. (4 points) Explain in what scenario an inserting thread is starved.

4.c. (8 points) Rollen wants to solve the starvation problem for inserting threads, but since he hates removing threads, he wants to allow them to get starved. Rollen googles this and finds a code. But then he notices that the code does not do what he wants. He thinks that this is a good midterm question. So, here we are! Explain why the following code does not prevent starvation of an inserting thread.

```
int nextToGo = 0;
int numInserting = 0;

void BBQ::insert(int item) {
   lock.acquire();
   myPos = numInserting++;
   while ((nextEmpty - front) == MAX
               || myPos > nextToGo) {
      itemRemoved.wait(&lock);
   }
   items[nextEmpty % MAX] = item;
   nextEmpty++;
   nextToGo++;
   itemAdded.signal();
   lock.release();
}
```

```



int BBQ::remove() {
   int item;
   lock.acquire();
   while (front == nextEmpty) {
      itemAdded.wait(&lock);
   }
   item = items[front % MAX];
   if ((nextEmpty - front) == MAX)
      itemRemoved.signal();
   front++;
   lock.release();
   return item;
}
```

4.d. (8 points) Rollen does not have time to google again or solve this himself. So, again, here we are! Complete the following code such that inserting threads do not starve but removing threads could starve. Your code should work for any sequence and number of calls to `insert` and `remove`. You may not need all the blank lines.

```
_______________________
_______________________
_______________________
void BBQ::insert(int item) {
  lock.acquire();
  _______________________
  _______________________
  _______________________
  _______________________
  while ((nextEmpty - front) == MAX
          _____________________) {
    _______________________
    _______________________
  }
  items[nextEmpty % MAX] = item;
  nextEmpty++;
  _______________________
  _______________________
  itemAdded.signal();
  lock.release();
}

int BBQ::remove() {
  int item;
  lock.acquire();
  while (front == nextEmpty) {
    itemAdded.wait(&lock);
  }
  item = items[front % MAX];
  if ((nextEmpty - front) == MAX) {
    _______________________
    _______________________
    _______________________
  }
  front++;
  lock.release();
  return item;
}
```