Please print in pen:

Waterloo Student ID Number:

WatIAM/Quest Login UserID:

UNIVERSITY OF WATERLOO

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

False! Consider 5 threads, 4 of which with execution time of 1 sec and one with execution time of 1000 sec. Running long thread first leads to average response time of 1002, whereas running short threads first leads to average response time of 202.8. The processor's throughput is 5/1004 (thread per second) in both cases.

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

True! Suppose that response times stay the same or increase for all tasks. This means that the completion time of the last thread does not improve. This in turn means that the throughput of the system does not improve, a contradiction!

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

True False

False! if the rate of receiving data is high, context switching for interrupts adds more overhead than simply polling.

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

True False

True! Hardware first pushes stack pointer and program counter. Interrupt handler then pushes the rest.

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

True False

False! Two stack pointers are stored but only one of them is for the interrupted process. The second stack pointer is for the interrupt handler.

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

True False

False! Arguments first get copied and then validated.

1.g. Kernel interrupt handler is a thread.

True False

False! It's not schedulable.

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

True False

True! Threads are independent and do not share states. Therefore, different interleavings result in the same output.

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

True False

False! Peterson's algorithm only needs atomic load and stores.

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

True False

False! Same reason as above.

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

True False

False! All threads share the memory space.

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

True False

False! It doesn't work in multiprocessors.

1.m. Starvation implies lack of progress.

True False

False! in the BBQ example a thread calling `get()` could starve while the whole system is making progress.

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

True False

True! If one process is in the critical section, other processes which want to access the critical section must wait.

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

True! Initialize semaphore to 1. P() = acquire() and V() = release().

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

True! In the latter case, the kernel also needs to context switch the process address space state (i.e., translation), and the context switch will also result in more cache misses.

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.

```
1  const int MAX = 10;
2
3  class TSQueue {
4     Lock lock;
5     int items[MAX];
6     int front;
7     int nextEmpty;
8   public:
9     TSQueue() {front = nextEmpty = 0;};
10    ~TSQueue() {};
11    bool tryInsert(int item);
12    bool tryRemove(int *item);
13 };
14
15 bool TSQueue::tryRemove(int *item) {
16    bool success = false;
17    lock.acquire();
18    if (front < nextEmpty) {
19       *item = items[front % MAX];
20       front++;
21       success = true;
22    }
23    lock.release();
24    return success;
25 }
26
27 bool TSQueue::tryInsert(int item) {
28    bool success = false;
29    lock.acquire();
30    if ((nextEmpty - front) < MAX) {
31       items[nextEmpty % MAX] = item;
32       nextEmpty++;
33       success = true;
34    }
35    lock.release();
36    return success;
37 }
38
39 int main(int argc, char **argv) {
40    TSQueue *queues[3];
41    thread_t workers[3];
42    int i, j;
43    for (i = 0; i < 3; i++) {
44       queues[i] = new TSQueue();
45       thread_create_p(&workers[i],
46       putSome, queues[i]);
```

```
47      }
48
49      printf("Let's begin!\n");
50
51      thread_join(workers[0]);
52
53      for (i = 0; i < 3; i++)
54          testRemoval(&queues[i], i);
55
56      printf("All done!\n");
57  }
58
59  void *putSome(void *p) {
60      TSQueue *tsq = (TSQueue *)p;
61      int i;
62      for (i = 0; i < 50; i++)
63          tsq->tryInsert(i);
64
65      return NULL;
66  }
67
68  void testRemoval(TSQueue *tsq, int q ) {
69      int i, item;
70
71      for (i = 0; i < 20; i++) {
72          if (tsq->tryRemove(&item))
73              printf("Deleted %d:%d\n", q, item);
74  }
```

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!"?

Max = 4 and min = 1

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

0-9, For queues[0], because of the join, the insertion will put 0-9 into the queue, and the other insertions will have no effect. When main returns from the join, it removes 0-9, and the other removals have no effect.

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

True, all inserts could happen after all removes

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

True, For queues[1] and queue[2], the insertion and removals are concurrent. So, it is possible that all the removals happen after the insertions, in which case 0-9 will be removed. It is possible that the removals will happen before all of the insertions, in which case 0-9 will be put but nothing will be removed. It is also possible that they can be interleaved so that up to 30 items are put into the queue and up to 20 items are removed. he items put (and therefore the items removed) will be sorted but not necessarily sequential (after items 0-9).

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

True, see above.

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

False, see above.

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.

```
1   void main (int argc, char **argv) {
2     int pid = fork(), x = 5;
3     if (!pid) {
4       x += 5;
5     } else {
6       pid = fork();
7       x += 10;
8       if (pid)
9         x += 5;
10    }
11    printf("%d\n", x);
12  }
```

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

3 copies, one for main and two for child processes

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.

10 15 20	10 20 15	15 10 20	15 20 10	20 10 15	20 15 10		

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.

```
1  void* f1(void* args) {
2     printf("F1: %d\n", *((int*) args));
3     return NULL;
4  }
5
6  void* f2(void* args) {
7     printf("F2: %d\n",  *((int*) args));
8     exit(0);
9  }
10
11 void main (void) {
12    int val = 5;
13    thread_t myT;
14    int pid = fork();
15
16    If(!pid) {
17       pthread_create_p(&myT, f2, &val);
18    } else {
19       val += 5;
20       waitpid(pid);
21       pthread_create_p(&myT, f1, &val);
22       thread_join(myT);
23    }
24
25    printf("Val: %d\n", val);
26    exit(0);
27 }
```

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

Max: 2 processes and 4 threads Min: 2 proc and 3 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.

Val: 5 F1: 10 Val: 10	F2: 5 F1: 10 Val: 10	Val: 5 F2: 5 F1: 10 Val: 10	F2: 5 Val: 5 F1: 10 Val: 10				

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.

```
1  class BBQ {
2   private:
3     Lock lock;
4     CV itemAdded, itemRemoved;
5     int items[MAX];
6     int front, nextEmpty;
7   public:
8     BBQ() {front = nextEmpty = 0;};
9     ~BBQ() {};
10    void insert(int item);
11    int remove();
12 };
13
14 void BBQ::insert(int item) {
15    lock.acquire();
16    while ((nextEmpty - front) == MAX) {
17       itemRemoved.wait(&lock);
18    }
19    items[nextEmpty % MAX] = item;
20    nextEmpty++;
21    itemAdded.signal();
22    lock.release();
23 }
24
25 int BBQ::remove() {
26    int item;
27    lock.acquire();
28    while (front == nextEmpty) {
29       itemAdded.wait(&lock);
30    }
31    item = items[front % MAX];
32    if ((nextEmpty - front) == MAX)
33       itemRemoved.signal();
34    front++;
35    lock.release();
36    return item;
37 }
```

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

No! if the queue is empty, the thread will have to wait. When itemAdded is signaled, some other waiting thread could wake up and remove the item. Another scenario happens when a new removing thread acquires lock and removes the item before the woken-up thread has a chance to do so.

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

First scenario: The queue is full. The inserting thread calls wait and goes to sleep. Once the queue has an item, the thread is signalled. Before the thread has a chance to acquire the lock, another inserting thread comes and inserts an item and makes the queue full again. The signalled inserting thread checks the queue. It's full. It calls wait and goes to sleep. And this happens iteratively.

Second scenario: Suppose queue is full and 10 inserting threads are waiting. Then 10 removing threads come and remove items. Only for the first removing thread nextEmpty – front == Max which means only the first removing thread will call signal and the rest of them do not signal. This means that only one of the inserting threads wakes up and the rest could wait forever.

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.

```
1  int nextToGo = 0;
2  int numInserting = 0;
3
4  void BBQ::insert(int item) {
5     lock.acquire();
6     myPos = numInserting++;
7     while ((nextEmpty - front) == MAX
8                || myPos > nextToGo) {
9        itemRemoved.wait(&lock);
10    }
11    items[nextEmpty % MAX] = item;
12    nextEmpty++;
13    nextToGo++;
14    itemAdded.signal();
15    lock.release();
16 }
```

```
17
18
19
20 int BBQ::remove() {
21    int item;
22    lock.acquire();
23    while (front == nextEmpty) {
24       itemAdded.wait(&lock);
25    }
26    item = items[front % MAX];
27    if ((nextEmpty - front) == MAX)
28       itemRemoved.signal();
29    front++;
30    lock.release();
31    return item;
32 }
```

Signal from a removing thread could wake up a wrong inserting thread and after that all the inserting threads will wait because myPos > nextToGo and no further signal is coming for removing threads.

Also, the second scenario described above could still happen!

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.

```
1  Queue ins;
2  CV next;
3  _____
4  void BBQ::insert(int item) {
5     lock.acquire();
6     myCV = new CV();
7     ins.append(myCV);
8     _____
9     _____
10    while ((nextEmpty - front) == MAX
11          || myCV != ins.front()  ) {
12       myCV.wait(&lock);
13       _____
14    }
15    items[nextEmpty % MAX] = item;
16    nextEmpty++;
17    ins.removeFront();
18    if (next = ins.front()) next.signal();
19    itemAdded.signal();
20    lock.release();
21 }
22
```

```
23 int BBQ::remove() {
24    int item;
25    lock.acquire();
26    while (front == nextEmpty) {
27       itemAdded.wait(&lock);
28    }
29    item = items[front % MAX];
30    if ((nextEmpty - front) == MAX) {
31       If (next = ins.front()) next.signal();
32       _____
33       _____
34    }
35    front++;
36    lock.release();
37    return item;
38 }
```

There is also another implementation which uses broadcast and doesn't need a queue to store CVs.