Please print in pen:

Waterloo Student ID Number:

WatIAM/Quest Login UserID:

UNIVERSITY OF WATERLOO

Midterm Exam - Winter 2020 - SE 350

1. Before you begin, make certain that you have one **2-sided booklet with 12 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.

2. Please read all of the questions before starting the exam, as some of the questions are substantially more time consuming. 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!**

3. 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 blank space on the last page of the exam clearly labeling the question and indicate that you have done so in the original question.

Good Luck!

Question	Points Assigned	Points Obtained
1	**38**	
2	**15**	
3	**23**	
4	**24**	
Total	**100**	

1. (X 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. With base and bound protection, it is not possible for any two threads to share code.

True False Why?

False, with B&B, sharing memory between two processes is not possible but sharing memory between two threads from the same process is.

1.b. Every I/O request from a user process eventually invokes the device driver's bottom half.

True False Why?

False, kernel might be able to satisfy the request by itself. For example, the content of a file might already be available in the kernel memory from previous calls to read().

1.c. If hardware does not provide atomic read-modify-write instructions, then libraries that implement user-mode threads have to use kernel-implemented locks to provide mutual exclusion.

True False Why?

False, mutual exclusion could be provided by using Peterson's algorithm without using any kernel-implemented locks.

1.d. First-come, first serve could minimizes average response time when tasks have equal length.

True False Why?

True, FCFS becomes SJF when tasks have equal length.

1.e. Consider two systems, A and B, that have identical hardware and software except that A has faster access to its memory cache. Throughput of A is at least as high as that of B AND average response time of A is at least as low as that of B.

True False Why?

True, throughput improves because faster cache access leads to lower context switching overhead. Response time also improves because faster cache access leads to faster execution time.

1.f. Improving average response time leads to improved throughput.

True False Why?

False, average response time and throughput are not necessarily related. For example, FCFS could have different avg. response times depending on the order of tasks while its throughput does not change for different orders.

1.g. Under Round-Robin scheduling, response time of the longest task in a workload does not change for any time quantum.

True False Why?

False, if the time quantum is more than task's length, then response time will depend on the order of tasks.

1.h. Every thread has its own stack.

True False Why?

True, threads from the same process share the same heap but have their own stack.

1.i. Condition variables could be implemented using Semaphores.

True False Why?

True, we can have separate semaphore for each waiting thread and put semaphores in ordered queue.

1.j. Any modification to page table entries in a multiprocessor necessarily requires a TLB shootdown.

True False Why?

False, although we can do a TLB shootdown on every modification to a page table, some modifications, such as adding permission to a page, do not require TLB shootdown.

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

True False Why?

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

1.l. If no new task arrives, the shortest task first scheduler never preempts currently running task.

True False Why?

True, the currently running task has to be the shortest one. If no shorter task arrives, the currently running task will remain the shortest one until it finishes.

1.m. The shortest remaining task first scheduling policy can lead to starvation.

True False Why?

True, consider an infinite stream of tasks that take less than a larger task in the queue.

1.n. “Hyperthreading” refers to the situation in which a modern operating system allows thousands of threads to access the same address space.

True False Why?

False, hyperthreading refers to the functionality (usually on x86 processors) that allows multiple physical threads (with distinct registers, program counters, etc.) to share the functional units in a single pipeline.

1.o. Thread pools are a useful tool to help prevent an overly popular web site from crashing the hosting web servers.

True False Why?

True, thread pools can be used to limit the maximum number of threads in a process.

1.p. Switching the order of two P() semaphore primitives can lead to deadlock.

True False Why?

True, If one P() is used to acquire a lock, and another one to wait(), we can get deadlock if the wait() happens in the critical section without releasing the lock.

1.q. The function thread_yield() will always ensure that another thread starts running.

True False Why?

False, pthread_yield() puts the currently running thread on the ready queue. The thread might run again if there are no other threads on the ready queue.

1.r. Dual-mode operation prevents a process from accessing another process' memory.

True False Why?

False, address translation prevents processes from accessing others' memory.

1.s. A kernel process can always enforce mutual exclusion by disabling interrupts on its running processor.

True False Why?

False, disabling interrupts on one processor does not guarantee mutual exclusion in multiprocessor systems.

2. (15 points) Uniprocessor Scheduling. We are going to compare the behavior of three schedulers on a workload consisting of 4 tasks:

- Task A arrives at time 0 with priority 0, uses the CPU for 1 s, issues I/O that takes 2 s, and then uses the CPU for 3s.
- Task B arrives at time 1 s with priority 1, uses the CPU for 6 s.
- Task C arrives at time 2 s with priority 2, uses the CPU for 2 s, issues I/O that takes 2 s, and then uses the CPU for 2 s.
- Task D arrives at time 3 s with priority 3, uses the CPU for 1 s, issues I/O that takes 1 s, uses CPU for 1 s, issues I/O that takes 1 s, and then uses CPU for 1 s.

This workload is illustrated below as if each executes on its own dedicated CPU.

	0	*1*	*2*	*3*	*4*	*5*	*6*	*7*
A	CPU	I/O	I/O	CPU	CPU	CPU		
B		CPU	CPU	CPU	CPU	CPU	CPU	
C			CPU	CPU	I/O	I/O	CPU	CPU
D				CPU	I/O	CPU	I/O	CPU

You will consider three scheduling algorithms: (1) first-come, first-serve (FCFS), (2) round robin (RR), and (3) shortest remaining time first (SRTF). The following apply:

- All schedulers are preemptive.
- The time quantum of the RR scheduler is 1.
- The moment a task arrives, it can be scheduled.
- If two tasks arrive at the same time, they are inserted in the ready queue in the lexicographical order. For example, if B and C arrive at the same time, B is inserted first, and C second in the ready queue.
- In the case of FCFS, once a task issues I/O, the scheduler schedules the next runnable task if there are any. This means that one the I/O issuing task is done with I/O, it is inserted at the end of the read list before any newly arriving tasks at that time.
- In the case of the RR scheduler, a new arriving task is inserted at the end of the ready queue. When the RR time quantum expires, the currently running task is added at the end of the ready list before any newly arriving tasks at that time.
- The RR and FCFS schedulers ignore the priorities.
- The SRTF scheduler uses priorities to break ties, i.e., if two tasks have the same remaining time, we schedule the one with the highest priority (3 is highest and 0 is the lowest).
- I/O requests for A, C, and D are sent to different devices and could be done in parallel.
- I/O time is not considered as tasks' waiting time.

Given the above information, calculate average response time and average waiting time for the three schedulers. Below is a scratch area for you to keep track of what each scheduler does. We will not grade the grid, but it may help you work out the question.

Time	FCFS	RR	SRTF
0	A	A	A
1	B (A I/O)	B (A I/O)	B (A I/O)
2	B (A I/O)	B (A I/O)	C (A I/O)
3	B	C	D
4	B	B	C (D I/O)
5	B	A	D (C I/O)
6	B	D	A (C and D I/O)
7	C	C (D I/O)	D
8	C	B (C I/O)	C
9	A (C I/O)	A (C I/O)	C
10	A (C I/O)	D	A
11	A	B (D I/O)	A
12	D	A	B
13	C (D I/O)	C	B
14	C	B	B
15	D	D	B
16	(D I/O)	C	B
17	D		
18			
19			
Average Response Time	(12+6+13+15)/4	(13+14+15+13)/4	(12+16+8+5)/4
Average Waiting Time	(6+0+7+10)/4	(7+8+9+8)/4	(6+10+2+0)/4

Duplicate of table for your convenience:

	0	*1*	*2*	*3*	*4*	*5*	*6*	*7*
A	CPU	I/O	I/O	CPU	CPU	CPU		
B		CPU	CPU	CPU	CPU	CPU	CPU	
C			CPU	CPU	I/O	I/O	CPU	CPU
D				CPU	I/O	CPU	I/O	CPU

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 always succeed.

```
void* func1(int i) {
        printf("%d\n", i);
        return NULL;
}
int main(void) {
        pid_t pid;
        thread_t thread[2];
        int status;
        pid = fork();
        if (!pid) {
                status = 12;
                thread_create(&thread[1], func1, 11);
        } else {
                status = 16;
                wait(pid);
                thread_create(&thread[1], func1, 14);
                printf("15\n");
        }
        printf("%d\n", status);
        return 0;
}
```

3.a (5 points) what is the minimum and maximum number of threads that will run?

Min: 2 and Max 4

3.b. (9 points) What are all the possible outputs in standard output? If there are multiple possibilities, put each one in a different box. You may not need all the boxes.

11 12 14 15 16	12 11 14 15 16	12 14 15 16	11 12 15 14 16	12 11 15 14 16	12 15 14 16	11 12 15 16 14	12 11 15 16 14	12 15 16 14	11 12 15 16	12 11 15 16	12 15 16

3.c (9 points) Modify the code such that the output will always be:

11
13
14
15
16

Fill in the blanks to show your changes to the original program.

- You must use thread_create at least once.
- You may not call printf or the helper function, func1, directly.
- Write at most one statement per line. You may not need all lines.

```
int main(void) {
        pid_t pid;
        thread_t thread[2];
        int status;
        pid = fork();
        _____________________________________

        if (!pid) {
                status = 12;
                thread_create(&thread[1], func1, 11);

                thread_join(thread[1]);_____________

                thread_create(&thread[0],  func1, 13);
                _____________________________________

                thread_join(thread[0]);_____________

                exit(0);____________________________

                _____________________________________

        } else {
                status = 16;
                wait(pid);

                _____________________________________

                thread_create(&thread[1], func1, 14);

                thread_join(thread[1]);_____________

                _____________________________________

                printf("15\n");

                _____________________________________

        }
        printf("%d\n", status);
        return 0;
}
```

4. (24 points) Locks, Semaphore, and Barriers.

4.a. (8 points) Fetch and Increment. We want to implement a lock using the fetch_and_increment primitive, which has the following semantics, and is executed **atomically**.

```
int fetch_and_increment(int *a) {
        int temp = *a;
        *a = temp + 1;
        return temp;
}
```

You have to implement acquire() and release() functions. Note that it is OK to busy wait. Write at most one statement per line. You may not need all the blank lines.

```
class Lock {
   private:
        int ticket_number;
        int turn;
  public:
        Lock() {
                ticket_number = 0;
                turn = 0;
        }
        void acquire() {

                int my_turn = fetche_and_increment(&ticket_number);

                while (lock.turn != my_turn) {};

                _____________________________


        }
        void release() {

                fetch_and_increment(&turn); or turn++;

                _____________________________


        }
}
```

4.b. (8 points) Sleeping Barber Problem. A barber shop has one barber chair, and a waiting room with *N* chairs. The barber goes to sleep if there are no customers. If a customer arrives, and the barber chair along with all *N* chairs in the waiting room are occupied, he leaves the shop. Otherwise, the customer sits on the waiting room chair and waits. If the barber is asleep in his chair, the customer wakes up the barber.
Consider that the barber and each of the customers are independent, concurrent threads. Fill in the following blanks to ensure that the barber and the customers are synchronized and deadlock free. Assume each semaphore has P() and V() functions available as semaphore.P() and semaphore.V(). Write at most one statement per line. You may not need all the blank lines.

```
Semaphore barberReady = 0;
Semaphore accessWaitRoomSeats = 1;
Semaphore customerReady = 0;
int numberOfFreeWaitRoomSeats = N;

void Barber () {
        while (true) {
                customerReady.P();

                accessWaitRoomSeats.P();

                numberOfFreeWaitRoomSeats += 1;

                accessWaitRoomSeats.V();

                cutHair();

                barberReady.V();
        }
}

void Customer () {
        accessWaitRoomSeats.P();

        if (numberOfFreeWaitRoomSeats > 0) {
                numberOfFreeWaitRoomSeats -= 1;

                accessWaitRoomSeats.V();

                customerReady.V();

                barberReady.P();

                getHairCut(); // Customer gets haircut :)
        } else {
                accessWaitRoomSeats.V();

                leaveWithoutHaircut(); // No haircut :(
        }
}
```

Partial credit is given (to varying extents, depending on the situation) if you had mixed up the P()/V() calls or customerReady/barberReady semaphores.

Full credit is given for accessWaitRoomSeats.P() and accessWaitRoomSeats.V() if they were before and after the numberOfFreeWaitRoomSeats access/modification, respectively.

4.c. (8 points). Synchronization Barrier. With *data parallel programming*, the computation executes in parallel across a data set, with each thread operating on a different partition of the data. Once all threads have completed their work, they can safely use each other's results in the next (data parallel) step in the algorithm. Google MapReduce is an example of data parallel programming, but there are many other systems with the same structure. For this to work, we need an efficient way to check whether all *N* threads have finished their work. This is called a *synchronization barrier*, which has only one operation—checkin. A thread calls checkin() when it has completed its work; no thread may return from checkin until all *N* threads have checked in. Once all threads have checked in, it is safe to use the results of the previous step. In this question have to implement checkin(). Your code has to be efficient in the sense that each checked-in thread has to receive at most one signal once all threads check in. Write at most one statement per line. You may not need all the blank lines.

```
class Barrier {
   private:
        Lock lock;
        CV all_checked_in;

        int num_of_threads;
        int num_of_entered;
  public:
        Barrier (int n) {
                num_of_threads = n;
                num_of_entered = 0;
        }
        void checkin() {

                lock.acquire();_____________________

                num_of_entered++;___________________

                if (num_of_entered < num_of_threads) {

                _____while (num_of_entered < num_of_threads)

                ____________all_checked_in.wait(&lock);

                } else {____________________________

                _____all_checked_in.broadcast();____

                }___________________________________

                lock.release();"____________________

                ____________________________________

                ____________________________________

        }
}
```