Please print in pen:

Waterloo Student ID Number:

WatIAM/Quest Login UserID:

UNIVERSITY OF WATERLOO

Final Exam - Winter 2019 - SE 350

1. Before you begin, make certain that you have one **2-sided booklet with 12 pages**. You have **150 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	34	
2	18	
3	17	
4	12	
5	19	
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. If each waiting thread in a system is waiting for a resource held by another waiting thread, then the system is in deadlock.

True False Why?

False, circular chain of requests is a necessary condition for deadlock, but it's not a sufficient condition.

1.b. For the Communal Dining Politicians problem where there are N chopsticks in the middle of a table and N politicians who can grab one chopstick at a time, there are N unsafe states.

True False Why?

False, there is only one unsafe state in which all politicians hold one chopstick

1.c. In a uniprocessor, for a workload of tasks with fixed sizes, work-conserving schedulers achieve better throughput than non-work conserving schedulers.

True False Why?

True, a non-work conserving scheduler could add to the total execution time of the workload thus reducing the system throughput.

1.d. Assuming zero overhead for Round Robin time slicing, FIFO is always better than Round Robin for average response time.

True False Why?

False, RR with short time quantum achieves better ART than FIFO for a workload with one extremely long job and several small ones.

1.e. The execution time of a parallel job could be more than 9.2 times faster on ten cores than on one core if only 99% of the job's execution time can be parallelized.

True False Why?

False, using Amdahl's Low, the maximum achievable speedup for this job on 10 cores is less than 9.2.

1.f. With virtual memory, the memory content of a user process is hidden from other user processes.

True False Why?

False, with Spectre and Meltdown attacks, an attacker can extract information about other processes' memory content due to side-channel vulnerabilities.

1.g. In a segmented memory, different processes can access the same physical address using different virtual addresses.

True False Why?

True, each process can assign different segment numbers to a shared physical segment which leads to addressing the same physical location with different virtual addresses.

1.h. Using TLBs could increase the cost of address translation for some processes and decrease it for others.

True False Why?

True, TLB, like any other caches, is useful only when the process reuses the cached data. TLB degrades performance if the process doesn't have locality of references.

1.i. On a context switch, TLB does not have to be flushed.

True False Why?

True, by tagging TLB entries with the process ID, modern processors avoid flushing the TLB on every context switch.

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. In a system with *base* and *bound* address translation, virtually addressed caches do not suffer from the aliasing problem.

True False Why?

True, aliasing only is an issue when different virtual addresses can point to the same physical address. However, with base and bound address translation, processes cannot share memory locations.

1.l. If no new task arrives, SJF 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. Threads within the same process can share data with one another by passing pointers to objects on their stacks.

True False Why?

True, threads in the same process share an address space.

1.n. With copy-on-write, immediately after a process has been forked, the same variable in both the parent and the child will have the same virtual address but different physical addresses.

True False Why?

Initially, the child and parent have the same physical memory mapped into their address spaces.

1.o. Disabling interrupts on any computer system that supports it guarantees atomicity.

True False Why?

False, disabling interrupts on a multiprocessor doesn't guarantee atomicity

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_create() will always ensure that another thread starts running.

True False Why?

False, pthread_create() puts the new thread on the ready queue. The thread might never run if the parent process terminates before the thread has started.

2. (18 points) Uniprocessor Scheduling.

2.a. (9 points). Given the following mix of tasks, task lengths, and arrival times, compute the completion time for each task for the FIFO, RR, and SJF algorithms. Assume a zero-cost time slicing of 10 milliseconds and that all times are in milliseconds. For RR, arriving tasks has lower priority than tasks that are already waiting. If a time slice completes at the same time that a job arrives, assume the arriving job has lower priority than tasks that are already waiting, but higher priority than the task completing its quantum.

Task	Length	Arrival Time	FIFO	RR	SJF
0	85	0	85	220	220
1	30	10	115	80	40
2	35	15	150	135	75
3	20	80	170	145	100
4	50	85	220	215	150

Show your work here:

With round robin, the task schedule is: 0, 1, 0, 2, 1, 0, 2, 1, 0, 2, 1 (done at time 80), 0, 2, 4, 5, 0, 2 (done at time 135), 3 (done at time 145), 4, 0, 4, 0, 4, 0, 4 (done at 215), 0 (done at 220). This is slightly better than FIFO, but considerably less than shortest job first.

With shortest job first, the task schedule is: 0 (until time 10), 1 (until it is done at time 40), 2 (until it is done at time 75), 0 (until time 80), 3 (until it is done at time 100), 4 (until it is done at time 150), and then 0 until is done at time 220.

2.b (9 points) Consider the following preemptive priority-scheduling algorithm based on dynamically changing priorities. Larger numbers imply higher priority. Tasks are preempted whenever there is a higher priority task. Assume that time is divided into 1millisecond time quanta and tasks do not arrive in the middle of a time quantum. When a task is waiting for CPU (in the ready queue, but not running), its priority changes at rate a: $P(t) = P(t-1) + a$, and when it is running, the task's priority changes at rate b: $P(t) = P(t-1) + b$. Suppose that $P(t) = 0$ for all $t \leq t_0$, where t_0 is the time at which the task joins the ready queue.

2.b.1 (3 points). What is the algorithm that results from $b > a > 0$?

FIFO; A running task gains priority faster than any waiting tasks, and any waiting tasks will have a priority greater than any task that arrives later. Therefore, this will act like FIFO.

2.b.2 (3 points). What is the algorithm that results from $a < b < 0$?

LIFO; Any newly arrived task will have higher priority than any waiting or running task, and so will be immediately scheduled. Once it is running, its priority will decrease more slowly than any waiting tasks, so it will keep the processor until another task arrives or it completes. Thus, this is last-in-first-out (LIFO).

2.b.3 (3 points). Suppose that tasks retain their priority when they are preempted. What happens if two tasks arrive at nearly the same time and $a > 0 > b$?

RR; Tasks gain priority as they wait and lose priority as they get CPU time. If the two tasks started with priorities near 0, then whoever ran first would have the CPU for a short time before being preempted and will get the CPU back quickly. This will be similar to round robin with a very short time quantum.

3. (17 points) Magnetic Disk. Consider a magnetic disk with the following spec.

Seek time from middle to outer track	10.3 ms
Seek time from middle to inner track	10.1 ms
Seek time from inner to outer track	19 ms
Average seek time	10.5 ms
Rotation time	8.3 ms
Transfer rate	54 - 128 MB/s
Bytes per sector	512

3.a. (6 points) SPTF is not optimal. Assume that the disk's head is on the middle track. Suppose that there are two sets of pending requests. The first set is 1000 requests to read each of the 1000 sectors on the inner track of the disk; the second set is 2000 requests to read each of the 2000 sectors on the outer track of the disk. Compare the average response time per request (i.e., the time for a request to complete, form when a request arrives until it is done, **excluding the transfer time**) for the SPTF schedule (first read the "nearby" inner track and then read the outer track) and the alternative of reading the outer track first and then the inner track. Write your final answer in the boxes.

SPTF: 32.43 ms

Outer-track-first: 23.53 ms

Show your work here:

SPTF: (1000 (10.1 + 4.15) + 2000 (10.1 + 8.3 + 19 + 4.15)) / 3000 = 32.45 ms

Outer-track-first: (2000 (10.3 + 4.15) + 1000 (10.3 + 8.3 + 19 + 4.15)) / 3000 = 23.55 ms

3.b. (3 points) Sequential Access. Now suppose that the head of the disk is on a random track. Consider 1000 read requests for sequential sectors on the outer track. How long does serving these requests take on average (considering the transfer time and out of order read)? Write your answer in the box.

Average sequential access time: 17.62 ms

Show your work here:

10.5 (seek time) + 4.15 (average rotation time) + 4ms (transfer time: 1000 sector * 512 bytes/sector * 1/(128 MB/s)) – 1.03 (out of order read = 8.3 * ½ * ¼) = 17.62 ms

3.c. (2 points). Effective Bandwidth. In 3.b. what fraction of the disk's bandwidth is realized? Write your answer in the box.

Effective bandwidth: $\boxed{\text{29.05 MB/s}}$

Show your work here:

Effective bandwidth = data / time = 1000 sectors * 512 bytes/sector * (1/17.62 ms) = 29.05 MB/s

3.d. (6 points) Linked List vs. Tree. Consider two different file systems FAT and FFS. For FFS, there are twelve direct pointers, one indirect pointer, one double indirect pointer, and one triple indirect pointer. Suppose that data blocks are 4KB in both file systems and suppose that FFS has 4-byte block pointers. Suppose that we create a new file, write 4 KB at offset 0, seek to block offset 2^{17}, and write another 4 KB. How many blocks does each file system use to store the file? Write your answer in the boxes.

FAT: $\boxed{2^{17}+1}$ FFS: $\boxed{4}$

Show your work here:

FAT uses link list representation in *file allocation table* which means it has to allocate all $2^{17}+1$ blocks. FFS uses four blocks: two data blocks, a double indirect block, and a single indirect block.

4. (12 points) Address Translation. Consider a system with the following parameters.

Variable	Measurement	Value
P_{TLB}	Probability of TLB hit	0.95
P_{L1}	Probability of a first-level cache hit for all accesses	0.99
P_F	Probability of a page fault when a TLB miss occurs on user pages (assume page faults do not occur on page tables).	0.001
T_{TLB}	Time to access TLB	1ns
T_{L1}	Time to access L1 cache	10 ns
T_M	Time to access DRAM	250 ns
T_D	Time to transfer a page to/from disk	10^7 ns

Suppose that the system has a 3-level page table that is stored in DRAM and are cached like other accesses. Also assume that the costs of the page replacement algorithm and updates to the page table are included in the T_D measurement. Suppose that pages mapped on a page fault are not cached and hardware automatically fills TLB on a miss.

4.a. (4 points). How long does it take for a user program to do one memory reference if the address translation is cached in TLB? Write your answer in the box.

Show your work: Time: 13.5 ns

$T_{TLB} + T_{L1} + (1 - P_{L1}) \times T_M$ = 1 + 10 + 0.01 * 250 = 13.5 ns

4.b. (4 points). How long does it take for a user program to do one memory reference if the address translation is not cached in TLB? Write your answer in the box.

Show your work: Time: 1050.98 ns

$(1 - P_F) \times \left(T_{TLB} + 4 \times (T_{L1} + (1 - P_{L1}) \times T_M)\right) + P_F \times (T_{TLB} + 6 \times (T_{L1} + (1 - P_{L1}) \times T_M) + T_D + T_{TLB} + T_{L1} + T_M)$ = 0.999 * (1 + 4 * 12.5) + 0.0001 * (1 + 6 * 12.5 + 10,000,000 + 10 + 250) = 1050.98 ns

4.b. (4 points). How long does it take for a user program to do one memory reference? Write your answer in the box.

Show your work: Time: 65.37 ns

$P_{TLB} * 13.5 + (1 - P_{TLB})1050.98$ = 0.95 * 13.5 + 0.05 * 1050.98 = 65.37 ns

5. (19 points) Locks and Deadlocks.
5.a. (5 points) Spinlock Using Swap. We want to implement locks using the swap() primitive. swap() has the following semantics, and is executed atomically:

```
void swap(int *a, int *b) {
        int temp = *a;
        *a = *b;
        *b = temp;
}
```

You have to implement Initialize, Acquire and Release for the lock operations. It is OK to busy wait.

```
void Initialize(int* lock) {

        *lock = 0;

}
void Acquire(int* lock) {

        int l = 1;
        do {
                swap(&l,
        lock);
        } while (l == 1);
        // Or
        // while(l == 1) {
        // swap(&l, lock);
        // }
}
void Release(int* lock) {

        *lock = 0;

}
```

5.b. (4 points). A Two-Phase Locking Paradigm. Suppose that we break up the modification of shared data into "two phases", this is what gives the process its name. There are actually three activities that take place in the "two-phase" update algorithm: (1) Lock Acquisition; (2) Modification of Data; (3) Release Locks. The modification of data, and the subsequent release of the locks that protected the data are grouped together and called the second phase. Consider a system with four mutual exclusion locks (A, B, C, and D) and a readers/writers lock (E) which allows multiple “reader” threads to simultaneously access the shared data. For E, any number of threads can safely read shared data at the same time, as long as no thread is modifying the data. However, only one “writer” thread may hold E at any one time. Suppose the programmer follows these rules:

a) During the first phase, no lock may be released, and, if E is held in writing mode, it cannot be downgraded to reading mode. Furthermore, lock A may not be acquired if any of locks B, C, D, or E are held in any mode. Lock B may not be acquired if any of locks C, D, or E are held in any mode. Lock C may not be acquired if any of locks D or E are held in any mode. Lock D may not be acquired if lock E is held in any mode. Lock E may always be acquired in read mode or write mode, and it can be upgraded from read to write mode but not downgraded from write to read mode.

b) During the second phase, any lock may be released, and lock E may be downgraded from write mode to read mode; releases and downgrades can happen in any order; by the end of part 2, all locks must be released; and no locks may be acquired or upgraded.

These rules ensure freedom from deadlock.

True False Why?

False, the problem is that upgrading E to a write lock can wait for a read lock on E held by some other thread, so there can be a cycle among threads holding a read lock on E and trying to upgrade to a write lock.

5.c. (10 points) Banker's Algorithm. Suppose there are three jobs A, B, and C running in a multi-core processor. Each job requires 4 resources (CPUs, last-level cache, memory capacity, and memory bandwidth) to run. Suppose that the multi-core processor has 16 cores, 12 MB last-level cache, 32 GB memory, and 70 GB/s memory bandwidth. For each job, the maximum possible usage for each resource is specified in the table below.

Job	CPU	Cache (MB)	Mem. Capacity (GB)	Mem. Bandwidth (GB/s)
A	4	8	20	16
B	6	6	10	32
C	10	4	14	36

5.c.1. (5 points). Consider the following allocation. Is the system in a safe state? If so, give an example guaranteed safe execution. If not, give an example of requests that could deadlock the system. Circle your answer and show your work.

Job	CPU	Cache (MB)	Mem. Capacity (GB)	Mem. Bandwidth (GB/s)
A	2	2	10	8
B	3	5	8	30
C	8	2	12	20

Yes No Why?

Yes, B -> A (or C) -> C (or A)

5.c.2. (5 points). Consider the following safe allocation. Is it safe to allocate to C four more cores? If so, give an example guaranteed safe execution. If not, give an example of additional requests that could deadlock the system. Show your work.

Job	**CPU**	**Cache (MB)**	**Mem. Capacity (GB)**	**Mem. Bandwidth (GB/s)**
A	3	7	18	16
B	4	3	6	20
C	5	1	6	34

Yes No Why?

No, by allocating four cores to C, there would be no enough resources to safely finish any of the jobs.