CSC 369H1 S 2020 Midterm Test
Duration — 50 minutes
Aids allowed: none

Student Number: |_|_|_|_|_|_|_|_|_|_|_|_|_|

Last Name: _____ **First Name:** _____

Lecture Section: L0201 **Instructor: Reid**

---

*Do **not** turn this page until you have received the signal to start.*
(Please fill out the identification section above, **write your name on the back of the test**, and read the instructions below.)
Good Luck!

This midterm consists of 6 questions on 8 pages (including this one). *When you receive the signal to start, please make sure that your copy is complete.* Pseudo code is acceptable where code is required. Answer the questions concisely and legibly. Answers that include both correct and incorrect or irrelevant statements will not receive full marks.
If you use any space for rough work, indicate clearly what you want marked.

1: _____/ 8
2: _____/ 3
3: _____/ 5
4: _____/ 8
5: _____/ 5
6: _____/ 4

TOTAL: _____/33

Question 1. [8 MARKS]

TRUE FALSE All system calls will cause a thread (or process) to block.

TRUE FALSE An interrupt will lead to a switch from user-mode to kernel-mode.

TRUE FALSE "." and ".." are hard links to directories.

TRUE FALSE When we create a hard link, there can be multiple inodes for the same file.

TRUE FALSE A lock prevents a thread from context switching so that it doesn't get interrupted while executing a critical section.

TRUE FALSE A program containing a race condition may never result in data corruption or some other incorrect behavior.

TRUE FALSE Any solution to the mutual exclusion problem that satisfies the progress requirement will also satisfy the starvation (or bounded waiting) requirement.

TRUE FALSE A semaphore is initialized to 0 when we plan to use it in the same way as a lock.

Question 2. [3 MARKS]

For each of the pseudo code lock implementations below, check **all** the true statements. (Assume the locks are correctly initialized.)

Part (a) [1 MARK]

```
int lock;
void acquire(lock) {
    while (lock);
    lock = 1;
}
void release (lock) {
    lock = 0;
}
```

- ☐ It works on machines with only one CPU (single core) because it disables context switches within critical region
- ☐ It works on machines with only one CPU even when there are context switches within critical region
- ☐ It does not work on machines with only one CPU
- ☐ It does not work on machines with multiple CPUs

Part (b) [1 MARK]

```
int lock;
void acquire (lock) {
    while(testandset(&lock));
}

void release (lock) {
    lock = 0;
}
```

- ☐ It works on machines with only one CPU because it disables context switches within critical region
- ☐ It works on machines with only one CPU even when there are context switches within critical region
- ☐ It does not work on machines with only one CPU
- ☐ It does not work on machines with multiple CPUs

Part (c) [1 MARK]

```
int lock;
void acquire(lock) {
    disable interrupts;
}
void release(lock) {
    enable interrupts;
}
```

- ☐ It works on machines with only one CPU because it disables context switches within critical region
- ☐ It works on machines with only one CPU even when there are context switches within critical region
- ☐ It does not work on machines with only one CPU

Question 3. [5 MARKS]

In class we discussed the rendezvous problem. If `thread_a` and `thread_b` are running at the same time, we want to ensure that A1 is printed before B2 and that B1 is printed before A2. Re-write the functions below using locks and condition variables to implement the synchronization. Use the smallest number of locks and condition variables possible and do not introduce any unnecessary constraints. (C-like pseudo code is fine.)

```
void *thread_a(void *) {
    fprintf(stderr, "A1\n");
    fprintf(stderr, "A2\n")
}
```

```
void *thread_b(void *) {
    fprintf(stderr, "B1\n");
    fprintf(stderr, "B2\n")
}
```

```
// Declare and initialize global variables
```

Question 4. [8 MARKS]

Consider the following description of the contents of an ext2 file system with a block size of 4096. The numbers in parentheses indicate the size of the directory or file in bytes. (131072 == 128 KiB)

For the questions below, assume that this is the entire file system and there are no reserved inodes other than the root inode.

```
/ (4096)
|__ largefile (131072)
|__ adir (4096)
     |__ emptydir (4096)
     |__ smallfile (2048)
```

Part (a) [4 MARKS]

How many inodes are in use?	
How many data blocks are in use?	
What is the value of the links field in the inode for `emptydir`?	
What is the value of the links field in the inode for `adir`?	

Part (b) [2 MARKS]

Suppose we run `mkdir /adir/emptydir/bdir`. Identify the changes in the file system considering inodes, data blocks, and bit maps in your answer. Be specific.

Part (c) [2 MARKS]

For the operation in **Part (b)**, in which order would you write the blocks to the disk to minimize the chance of leaving the file system in an inconsistent state if the computer were to lose power before all blocks were written? Explain your rationale.

Question 5. [5 MARKS]

Part (a) [1 MARK]

Briefly explain what happens when a process running in user-mode executes a privileged instruction.

Part (b) [1 MARK]

In assignment 1, some students proposed that each `a1fs_write` call that required a new block would start a new extent. Briefly explain the main problem with this approach.

Part (c) [1 MARK]

Can a process make the transition from the ready state to the blocked state? Explain why or why not. *No. A process moves to the blocked state because it made a system call that blocks or because it received a signal that will cause it to block (e.g. SIGSTOP). In either case, it must be running before it blocks.*

Part (d) [2 MARKS]

Consider the dining philosopher's problem. In class, students proposed a solution where a philosopher must acquire a global lock before checking whether both their chopsticks are free. If both chopsticks are free, then the philosopher will release the global lock, eat and then put down the chopsticks. Otherwise, the philosopher will release the global lock and try again.

Explain the major drawback to this solution.

Question 6. [4 MARKS]

Explain how the `fork` system call is implemented. Include the steps that are taken from the point that a process calls fork until the point that control is transferred back to a user process.