CSC369H1 S2016 Midterm Test
Instructor: Bogdan Simion

Duration - 50 minutes
Aids allowed: none

Student number: _____

Last name: _____ First name: _____

Lecture section: L0101(day) L5101(evening) (circle only one)

---

*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 5 questions on 8 pages (including this one and blank pages). *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.

Q1:	_____/7
Q2:	_____/6
Q3:	_____/13
Q4:	_____/6
Q5:	_____/4
Total:	_____/36

Q1. (1 mark each) True/False Indicate below, for each statement, whether it is (T)rue or (F)alse. Circle the correct answer.

T / F: The SJF scheduling algorithm may suffer from starvation.

T / **F**: Processes have their own address space so they are faster to create and destroy compared to threads.

T / **F**: Hoare monitor semantics allow for broadcast operations on condition variables.

T / **F**: If no thread is waiting on a particular condition variable, a signal operation on that condition variable will be recorded for later use.

T / **F**: User-level applications must be able to reserve more memory for themselves without OS support.

T / F: In UNIX-like systems, a program does not need to run with root privileges to issue a system call.

T / F: A context switch is performed with the support of hardware and the operating system.

Q2. (2 marks each) (Conceptual) Explain briefly the following concepts or terms, in the context of this course:

a) Critical section

Ans: Section of code that must be executed atomically to avoid race conditions.

b) Preemptive scheduling

A process can be interrupted by the OS. After its timeslice is finished, the OS can then schedule another process into the running state.

c) Starvation

Answer: When a process/thread/task can never get to run (or has an unbounded waiting time). There's basically always some other thread preferred in its place.

Q3. (13 marks) (Conceptual) Answer the following short questions.

a) (1 mark) What x86 register contains the system call number, when a system call is issued?

Answer: AX, or EAX (whichever is fine).

b) (2 marks) In which queue does a process's PCB get linked when the process receives a SIGSTOP signal?

Answer: The process goes in the Blocked queue, so its PCB gets linked to the Blocked wait queue.

c) (2 marks) When an interrupt comes in, how does the system know what to do?
Answer: the reason is stored in some register, and this is used to invoke an interrupt handler.

d) (2 marks) Using multiple threads per process allows a system to better overlap computation and I/O. Do you agree with this statement? Explain your rationale.
Answer: True. By having multiple threads within a process, we could have some of its threads do computation and others do I/O. This way we could better overlap computations and I/O (say, while a thread computes something, other threads pre-fetch the next data we need).

e) (4 marks) Explain how the MLFQ scheduler works.
What type of problems does the MLFQ scheduler need to consider?
Answer: MLFQ uses a number of distinct queues, each one assigned a different priority level. Each queue has multiple ready-to-run jobs, and the scheduler always chooses to run the jobs in the queue with the highest priority. Jobs start in the highest priority queue. If a job uses an entire time slice, its priority gets reduced (gets moved to a lower queue). If the job gives up the CPU before the timeslice is up, it stays at the same priority level.
Problems:
- avoiding starvation
- preventing processes from gaming the scheduler (a sneaky process could yield the CPU at 99% of its timeslice on purpose, to stay at a high priority)

f) (2 marks) Does disabling interrupts ensure mutual exclusion is achieved? Explain why or why not.
Answer: It depends. In a single-processor system, yes! In a multi-processor system, it won't achieve this goal. Disabling interrupts only applies to the specific core assigned to the thread that disabled the interrupts. So whether the current CPU has interrupts disabled has no impact on other CPUs, and thus does not prevent them from entering the same critical section.
Full marks for any reasonably specific and correct explanation along these lines.

Q4. (6 marks) Synchronization (Reasoning)

At Hogwarts school of magic, all wizards (students and professors) use a social network to establish groups of friends. The code behind this application uses some synchronization, as shown below.

You can assume that a *wizard_list* is a data type that represents a basic linked list, where each node contains a wizard structure and a next pointer. The list supports the following operations:

```
void list_add (wizard_list *l, wizard *w);

int wizard_is_in_list (wizard_list *l, wizard *w);
```

The first operation adds a wizard to a given list. The second operation checks if a wizard is in a given list (returns an integer: 1=yes, 0=no).

The function *request_friend()* is called whenever a wizard decides to befriend another wizard. You can assume that, as shown in *accept_friend()*, the request is always granted, just as long as the friendship is not already established.

Your task is to decide whether the two functions are correct, or whether one or several problems may arise. Indicate below what your conclusions are, explaining in detail your reasoning. Give examples if necessary.

```
typedef struct wiz {
    char *name;
    wizard_list *friends;
    pthread_mutex_t *lock;
} wizard;
```

```
int request_friend(wizard *me, wizard *newfriend) {

    pthread_mutex_lock(me->lock);

    if (accept_friend(newfriend, me)) {
        list_add(me->friends, newfriend);
        printf("%s is now connected to %s\n",
                me->name, newfriend->name);

        return 1;
    }
    pthread_mutex_unlock(me->lock);
    return 0;
}
```

```
int accept_friend(wizard *me, wizard *newfriend) {

    pthread_mutex_lock(me->lock);

    if ( ! wizard_is_in_list(me->friends, newfriend) ) {
        list_add(me->friends, newfriend);
        printf("%s is now connected to %s\n",
                me->name, newfriend->name);

        return 1;
    }
    pthread_mutex_unlock(me->lock);
    return 0;
}
```

Answer: Problems are: 1) no unlock for early return. 2) even if unlocking for early return, the code can deadlock given that when calling accept_friend, we hold the lock for the requestor wizard. So, if we have something like this:
Thread A *Thread B*
request_friend(harry, snape) *request_friend(snape, harry)*
Then deadlock may occur.
3) nothing stopping a wizard from trying to befriend themselves. [1 mark for observing this less obvious problem.]

Q5. (4 marks) Scheduling (Reasoning)

Assume that we have a multi-level queue scheduler, with 3 queues Q0, Q1, and Q2, where Q0 is the highest priority queue, and Q2 is the lowest.
All three queues use a round-robin scheduling algorithm, with a time quantum of 2. New processes and processes returning from I/O start at the front of Q0 (among these two categories, new processes go first). When a process finishes its time quantum, it gets preempted and placed at the back of Q2.

After having a look over this scheduler, Linus Torvalds comes and says "This scheduler is really stupid. Whoever implemented is going to hear a piece of my mind. What is the point of ..", then the conversation cuts off.

a) What did Linus have against this scheduler?
b) Can you conceptually redesign a better scheduler, such that it has the exact same outcome in terms of which process gets scheduled next? Explain briefly how that would work.

Answer: a) [2 marks] This scheduler is too complex for no good reason. First of all, why maintain 3 queues, if they have the exact same policy, and new jobs and those returning from I/O always start at the front of Q0, while those preempted go all the way at the back of Q2. Also, what is the point of Q1? There can never be any process in that queue.
b) [2 marks] A better policy that has the exact same effect would be to simply use a single queue and have new processes or those returning from I/O start at the front of the queue, and preempted processes go at the back.
As long as the reason is valid and it is indeed more efficient, partial marks may be awarded, even if not realizing that the queues can just be collapsed into one anyway.