CSC369H1 S2017 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 and read the instructions below.)

Good Luck!

This midterm consists of 5 questions on 10 pages (including this one and blank pages). *When you receive the signal to start, please make sure that your copy is complete.*

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:	_____/9
Q5:	_____/5
Total:	_____/40

[This page is left intentionally blank. Use the space below for rough work. This page will not be marked unless you clearly indicate the part of your work that you want us to mark.]

Q1. (1 mark each) True/False [5 minutes]

Indicate below, for each statement, whether it is (T)rue or (F)alse. Circle the correct answer.

T / [✓ F]: The Ready queue contains only those processes which finished their allocated time quantum and got descheduled.

T / [✓ F]: Disabling interrupts is a method that can be used to ensure mutual exclusion on any system.

[✓ T] / F: During a context switch, the OS has to save the registers for the currently executing process, and restore those for the process that will take over the CPU next.

[✓ T] / F: In the top part of each process's address space, there is a mapping of kernel addresses.

[✓ T] / F: If no thread is blocked on a particular semaphore, then a signal on that semaphore will be recorded and let the first arriving thread go through a wait() operation on that semaphore.

[✓ T] / F: One of the roles of an OS is to handle privileged operations on behalf of user processes.

T / [✓ F]: An atomic operation which modifies the contents of a shared variable may leave this variable in an inconsistent state if an interrupt arrives midway through the operation.

Q2. (2 marks each) (Conceptual) [5 minutes]

Explain briefly the following concepts or terms, in the context of this course:

a) System call

Answer: A function that allows user space processes to request a privileged operation from the OS. The OS handles the privileged operation on behalf of the calling user process.

b) Scheduling quantum

Answer: In a preemptive scheduler, this is the time allotted to a process before context-switching to another process.

c) Mesa semantics

Answer: One of the monitor semantics. A Signal() on a condition variable which threads may be waiting on, places the waiter on the ready queue, but the signaller continues inside the monitor. So, since the signaler keeps executing, the condition is not necessarily true anymore when the waiter resumes (the signaler could modify something). As a result, under Mesa semantics, we must check the condition again in the waiter's procedure.

Q3. (13 marks) (Conceptual + Reasoning) [10 minutes] Answer the following short questions.

a) (3 marks) Sleep locks are preferable to spin locks when the expected wait time is short. Do you agree with this statement? Explain *clearly* why or why not. (A simple yes or no will get 0 marks)

Ans: Nope, it's the exact opposite. Spinlocks do busy-waiting until the lock is available so they are preferred if the length of the wait is very short (e.g., a few CPU instructions). Sleep locks put the thread to sleep until such a time when the lock can be acquired, so they are preferred when the wait time is expected to be much longer.

b) (2 marks) Using multiple threads per process allows a system to better overlap computation and I/O. Do you agree with this statement? Explain *clearly* your rationale. (A simple yes or no will get 0 marks)

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).

c) (3 marks) Explain an advantage and a disadvantage of using user-level threads rather than kernel-level threads?

Ans: [1.5 marks] Advantage: User-level threads are much cheaper and faster. The kernel-level threads are managed by the OS, whereas with UL threads, creating a new thread, switching between them, and synchronizing can be done via procedure calls (no kernel involvement).

[1.5 marks] Disadvantage: User-level threads are not integrated with the OS, so the OS could make poor decisions (e.g., scheduling a process with only idle threads, blocking a process whose thread initiated an I/O, even though the process has other threads that can execute, descheduling a process holding a lock, etc.).

d) (4 marks) You are tasked with designing code for an application and you realize that the synchronization patterns fit the abstraction of a monitor. You realize that to make your code work according to specifications, the condition variables should perform *pthread_cond_wait* and *pthread_cond_broadcast* operations. Based on the monitor semantics we discussed in class, which monitor semantics would be most suitable here? Explain *in detail* your choice, and provide reasons why the other one would not work well.

Answer: [1 mark] Mesa semantics.

[3 marks] With broadcast operations, we are not guaranteed that the condition holds once a process gets unblocked from the condition it waits on. If we signal one process, that process is guaranteed that its condition holds. If we broadcast, the lucky one that gets to run to completion through the monitor, might change data such that the condition no longer holds for the next process. In Mesa semantics, we use a while loop to recheck the condition, so we are guaranteed it will hold.

e) (1 mark) Which of the following are NOT stored in a Process Control Block (PCB)? Circle all that apply.

a. heap

b. open file descriptors

c. program counter

d. system call table

Answer: heap and system call table. (0.5 marks each). Minus 0.5 marks for each incorrect choice picked (no deduction if a correct answer is not circled). Capped at zero, i.e., no negative marks for this question.

Q4. (9 marks) Synchronization (Reasoning) [15 minutes]

Consider the following problem: we have two functions that operate on a list called `listhead`, defined below. The function `delete_node` deletes a node (the first occurrence, if any) with a given value from the list. The function `print_list` traverses the list and prints the value in each element. The head of the list is always an empty node whose value is irrelevant and whose sole purpose is to delimit the start of the list.

a) [7 marks] Using **ONLY semaphores**, ensure that these functions are correctly synchronized, to exhibit the behaviour requirements described above. You may declare additional global variables. You can declare as many semaphores as you need, however your solution will be marked for both ***correctness and efficiency***! *Note:* You want good parallelism, so a solution where each thread runs exclusively will not get any marks!

Useful API:

- Declaration example: `sem_t s;`
- Initialization example: `sem_t s = 0;` (assume that this works, even if it's not quite the right way)
- Operations: `sem_wait(sem_t);` `sem_post(sem_t);`

```
typedef struct _node {
     int value;
     struct _node * next;
} node;
node *listhead;
/* any globals you might want go here; efficiency matters! */
int read_count = 0;
sem_t mutex = 1;
sem_t w_or_r = 1;
```

```
void * print_list() {

   node *curr = listhead->next;

   sem_wait(mutex);
   read_count ++;
   if (read_count == 1) {
     sem_wait(w_or_r);
   }
   sem_post(mutex);

   while(curr != NULL) {

      printf("[%d]-\n",curr->value);

      curr = curr->next;


   }

   sem_wait(mutex);
   read_count --;
   if (read_count == 0){ //last reader?
     sem_post(w_or_r);
   }
   sem_post(mutex);
}
```

```
void * delete_node(int value) {

   node *curr = listhead;

   sem_wait(w_or_r);
   while(curr->next != NULL) {


     if(curr->next->value == value){

        curr->next = curr->next->next;

        break;
     }

     curr = curr->next;


   }
   sem_post(w_or_r);
}
```

Ans: Tentative marking scheme: 5 marks print_list, 2 marks delete_node, 2 declarations (if excessive, 0 marks for this category)

b) [2 marks] Analyzing your solution from part a), can threads running either `print_list` or `delete_node` potentially get starved? Explain why or why not.
Note: Be specific, but you do not need to fill this entire page.

Ans: This really depends on the solution provided. Most likely this will result in the potential for starvation. For example, if the solution is the classic reader-writer, where readers can operate concurrently, then it is possible that for example, readers will keep doing print_list, and end up starving inserter threads.

Q5. (5 marks) Scheduling (Reasoning) [10 minutes]

Assume that we have a multi-level queue scheduler, with 2 queues Q1 and Q2, where Q1 is the higher priority queue. Both queues use a round-robin scheduling algorithm, and new processes start in Q1. Processes which do not finish their timeslice get placed in the lower priority queue Q2, while those that do use their entire timeslice are kept in the same queue.

a) Which type of processes (CPU-bound or I/O-bound) are likely favoured by this scheduler and why?
b) Starvation can occur for processes that end up in Q2. What solution did MLFQ use to prevent this type of situation? Explain how we can use the same strategy in the context of this scheduler.

Note: Do not feel compelled to use this entire page, but be specific and elaborate on your thought process!

Answer: [3 marks] Prioritizes CPU-bound processes: Processes that don't finish their timeslice get placed in the lower priority queue, while those that do use their entire timeslice are kept in Q1. This way CPU-bound processes get prioritized.
[2 marks] Avoid starvation: To avoid starving I/O-bound processes, we can periodically do priority boosting just like in MLFQ, by "promoting" all processes from Q2 to Q1.
(Any other sensible solution that correctly addresses the requirements, should get full marks).

[This page is left intentionally blank. Use the space below for rough work. This page will not be marked unless you clearly indicate the part of your work that you want us to mark.]

[This page is left intentionally blank. Use the space below for rough work. This page will not be marked unless you clearly indicate the part of your work that you want us to mark.]