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 SJF scheduling algorithm minimizes average wait time.

T / **F**: After fork()-ing, the parent and child process share the same address space.

T / **F**: A process whose parent has died is called a zombie process.

T / F: During a context switch, when the return-from-trap instruction is executed, instead of returning to the process that was running before the context switch, we resume the execution of another process.

T / F: The hardware exposes the use of protection domains through the mode bit.

T / **F**: If no thread is blocked on a particular condition variable, then a signal on that condition variable will be recorded and let the first arriving thread go right through a pthread_cond_wait() operation on that condition variable without blocking.

T / **F**: A mutex can be implemented with a semaphore that has its initial value set to 0.

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

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

a) System call table

Answer: This contains the mapping between system call numbers and corresponding system call handlers. It serves as a lookup when a system call comes in, to get directly to the right syscall handler.

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) Priority inversion

Answer: Priority inversion happens when a low priority process prevents a high priority process from making progress by holding some resource.

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

a) (3 marks) Name at least two components of program state which are shared between threads of the same process? Explain *in detail* which one is faster: creating a thread or creating a process? (Simply indicating one or the other without any explanation will receive 0 marks)

Ans: heap, code, global variables, etc. [1 mark]

[2 marks – must explain not just state which one] Thread creation is much faster typically, no need to create as much state as for a process.

b) (3 marks) In class we discussed that an OS has several roles. Name and describe at least two such roles.

Answer: (0.5 marks for each role named, 1 mark for each role being explained clearly – no need to write a lot as long as it's clear) Any 2 of these 3 is fine:

- Virtual machine (provides an interface to the physical machine so that programs can make use of H/W resources)

- Resource allocator (acts as a manager of resources like CPU, memory, files, I/O devices, etc., to allow their proper use by the applications that run on the computer system).

- Control program (OS controls the execution of user programs to prevent errors and improper use of the computer, i.e., enforcing protection against malicious or accidental bad behaviour of the user applications)

c) (2 marks) Name two circumstances when a process will be placed in the Blocked state? From the blocked state, what events can trigger the process being placed back in the Ready queue?

Answer: If a process receives a signal like SIGSTOP, or needs to do I/O, it will be placed in the Blocked state. From the Blocked state, a process will be moved back to the Ready queue once the reason for blocking is over (I/O is done. SIGCONT arrives, etc.).

d) (4 marks) You are tasked with designing code that synchronizes several threads which access shared resources in a large-scale system. Threads which cannot proceed should be placed into a wait queue, rather than allow them to do busy-waiting. Threads cannot proceed into their critical section as long as the values of three global parameters A, B, and C are simultaneously above three predefined thresholds respectively. Decide if semaphores or condition variables are most suitable for your code. Explain *in detail* your decision.

Answer: Use condition variables instead of semaphores. This type of semantic where deciding to wait is based on a complex condition, is most suitable for condition variables. It's not impossible to enforce synchronization using semaphores but the code will be much more complex, because semaphores cannot incorporate a complex condition as part of the wait() mechanism as seamlessly as condition variables. While semaphores also enforce an internal wait queue, they are more suited when waiting depends on a single countable resource, which lends itself to being abstracted by the internal counter of a semaphore.

e) (1 mark) Which of the following scheduling algorithms may cause starvation? Circle all that apply.

a. FCFS

b. SJF

c. Round-Robin

d. MLFQ

Answer: b. only (1 mark). 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 a bank that manages several customer accounts, but which allows only one operation for its customers: `transfer_amount(account *a1, account *a2, float amount)`, which transfers the given `amount` from account `a1` to account `a2`. The implementation of this function is given below:

```
typedef struct acct {
      float balance;
      pthread_mutex_t *lock;
} account;

void transfer_amount(account *a1, account *a2, float amount) {
      pthread_mutex_lock(a1->lock);
      pthread_mutex_lock(a2->lock);
      a1->balance -= amount;
      a2->balance += amount;
      pthread_mutex_unlock(a1->lock);
      pthread_mutex_unlock(a2->lock);
}
```

You can assume that all synchronization variables have been properly initialized and that memory has been allocated where necessary. Account balance can be negative too.

a) (2 marks) Provide a sequence of execution which leads to deadlock.

Answer: if two accounts do this at the same time, and each one acquires their own lock, then blocks waiting for the other's lock. Also, self-transfers will deadlock because a thread that acquires a1->lock in the first line of code, then tries to acquire a1->lock again will deadlocks on itself. [2 marks]

b) (2 marks) Consider that instead of using a lock per account like in the code above, we use one global lock (and change `transfer_amount` to acquire the global at the beginning of the function, and release it at the end).
i) Can deadlock occur in this case? If yes, give an example sequence of execution. If no, explain why not.
ii) Do you see some other drawback to using this approach?

Answer: i) No, deadlock cannot occur here because there's only one lock for everyone, so by definition only one thread can be in the transfer_amount critical section.
ii) Yes, a global lock means that ALL account operations have to serialize (no parallelism is possible), which is very inefficient.

c) (5 marks) The bank hires you to change the `transfer_amount` implementation so that the amount can be transferred only if there is at least that amount in the account. If the account were to be left with a negative balance as a result of the transfer, then the operation waits until the account gets sufficient money (from some other transfer).
You may modify the account structure as you see fit, and add in it any other synchronization variables you may need (no need to worry about initializing them, as long as it is clear what your intent is). Your solution should ensure *no deadlocks* occur. You may *NOT declare any global variables for synchronization purposes*.
Note: efficiency does matter (particularly in terms of achievable parallelism)!

```
typedef struct acct {

      float balance;

      pthread_condition_t *cond;
      pthread_mutex_t *lock;

} account;


void transfer_amount(account *a1, account *a2, float amount) {

      /* This is just one example solution; there may be other correct solutions
      Just as long as no deadlock can occur and no excessive synchr. primitives
      are used, that should be fine */

      pthread_mutex_lock(a1->lock);

      while(a1->balance – amount < 0) {
            pthread_cond_wait(a1->cond, a1->lock);
      }

      a1->balance -= amount;

      pthread_mutex_unlock(a1->lock);



      pthread_mutex_lock(a2->lock);

      a2->balance += amount;

      pthread_cond_broadcast(a2->cond); // signal is fine too.

      pthread_mutex_unlock(a2->lock);

}
```

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

Consider a **process scheduling** algorithm that favours processes which have used the least processor time in the recent past. Answer the following questions and explain *in detail* your rationale.

a) Will this algorithm favour either CPU-bound processes or I/O-bound processes?
b) Can this algorithm permanently starve either CPU-bound or I/O-bound processes?

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

Answer: a) Clearly I/O-bound processes only rarely make use of the CPU, so they will always be favourable compared to others according to this scheduler's policy (due to having used the least processor in the recent past).
b) It cannot starve I/O-bound processes, since they are the preferred anyway. It will not starve CPU-bound processes permanently either because I/O-bound processes by nature will give up the CPU for large periods of time in order to do I/O, hence there will always be opportunities for CPU-bound processes to get the CPU.

[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.]