UNIVERSITY OF TORONTO
FACULTY OF APPLIED SCIENCE AND ENGINEERING

MIDTERM EXAMINATION, February, 2024
Third Year – Materials
ECE344H1 - Operating Systems
Calculator Type: 4
Exam Type: A
Examiner – D. Yuan

Duration: 55 minutes

*There are **XX** total numbered pages, 7 Questions.* Read All Questions Carefully!

The teaching staff will not answer any questions about the exam. If any questions appear unclear or ambiguous, make any assumptions you need, state them, and answer the question that way. If you believe there is an error, state what the error is, fix it, and respond as if fixed.

You receive 20% of the mark of each (sub-)question if you leave the answer blank.

Please put your FULL NAME and Student ID on THIS page only.

Name: _____

Student ID: _____

	Total Marks	Marks Received
Question 1	21	
Question 2	12	
Question 3	10	
Question 4	19	
Question 5	9	
Question 6	9	
Question 7	20	
Total	100	

Assume a single core CPU is used in this exam.

Q1 (21 marks): Multiple Choices (choose all correct answers). If there is no correct answer, write "None of the above".

(1) What does the term "critical region" mean?
a) A group of instructions that is performed only once by a single thread.
b) A group of instructions that cannot be interrupted.
c) A group of instructions that executes as a whole in an all-or-nothing manner.
d) A group of instructions that every instruction is an atomic instruction.

(2) What does the term "context switch" refer to?
a) Switching between user mode and kernel mode.
b) Switching between different processes or threads.
c) Switching between different instruction sets.
d) Switching between primary and secondary storage.

(3) Which of the following does **not** trigger an event?
a) Modify the stack pointer register (SP)
b) User space process dereferences a NULL pointer.
c) Execute privileged instructions in user mode.
d) Execute the **test-and-set** atomic instruction.
e) Calling a system call.

(4) Which of the following are the unique advantages of user-level threads (when compared with kernel-level threads)?
a) Block isolation: if one thread blocks on I/O, the other threads in the same process are not affected (i.e., not blocked).
b) Context switch is much faster.
c) Threads can run in parallel.
d) Thread creation and termination are much faster.

(5) When you call fork(), what state is the child process in upon its creation?
a) Blocked
b) Running
c) Ready
d) Zombie

(6) Which of the following are shared among threads in one process?
a) Address space
b) Program counter

c) Open files
d) Stack
e) Registers
f) Global variables
g) Local variables
h) Dynamically allocated variables

(7) Which of the following components can be accessed from other threads in one process?
a) Address space
b) Program counter
c) Open files
d) Stack
e) Registers
f) Global variables
g) Local variables
h) Dynamically allocated variables

Question 2 (12 marks). For each of the following scenarios, specify how the process's state is changed (e.g., from Running to Waiting). You only need to write the first change (in case there are consecutive state changes).

a) Typing into a text editor

Answer: Waiting -> Ready.
(Waiting -> running is an acceptable answer.)

b) Timer interrupt triggering a context switch

Answer: running -> ready.

c) A process calls exit().

Answer: running -> terminated.

d) A process calls printf("ECE344");

Answer: running -> waiting.

Question 3 (10 marks). Please rank the speed of the following operations on a modern computer; #1 being the fastest.

Rank	Operation
3	Trap into the kernel.
3	Load from memory (not cache).
2	Load from CPU cache (level-1 cache).
4	Read from SSD.
5	Read from the hard drive.
1	Read from CPU registers.

Note: Trap into the kernel and Load from memory's order can change, but not the others.

Question 4 (18 marks). System call implementation. Consider the implementation of the system call: open (mode, flags, path) on FreeBSD operating system.

```
1  open:
2    push dword mode
3    push dword flags
4    push dword path
5    mov eax, 5
6    push dword eax ; syscall number
7    int 80h
8    add esp, byte 16
```

(1) (4 marks) Which of the instructions from the above execute in user-space and which execute in kernel-space?

All of them execute in user-space.

(2) (15 marks) Now we are going to make a number of changes to the code. Answer if the change would result in a bug, and if so, what are the symptoms (i.e., consequences). Note that each change is applied in isolation, i.e., they are not cumulative.

Change	Bug?	Consequence (only if there is a bug)

Swap line 2 and 3	Yes	Passing incorrect parameters (mode & flags) to the OS
Swap line 3 and 4	Yes	Passing incorrect parameters (flags & path) to the OS
Swap line 4 and 5	Yes	Passing incorrect parameters (syscall # & path) to the OS
Swap line 5 and 6	Yes	OS doesn't get the correct system call number; could execute the wrong system call.
Swap line 6 and 7	Yes	Same as above
Swap line 7 and 8	Yes	OS doesn't get the correct system call number and parameters. could execute the wrong system call, or use the wrong parameter, etc.
change line 8 to `add esp, byte 20`	Yes	Free more memory than we should from the stack. After open returns the application will use the wrong stack pointer. Many bad things can happen, e.g., return to the wrong address, buffer overrun, etc.

Question 5 (9 marks). Recall that in Lab2, semaphore is implemented using splhigh() and splx().

1) (3 marks) Describe what these two functions do.

Answer: They will disable and enable interrupts. (More precisely, splx() will reset the interrupt to the previous value.)

2) (6 marks) Your classmate Lok Sachs argues that the combination of splhigh() and splx() can replace atomic instructions like test-and-set. Give him two reasons why atomic instructions cannot be replaced by splhigh() and splx().

Answer: (1) disabling interrupt only works for kernel code (cannot be allowed for user-space synchronization); (2) only works on single core.

Question 6 (9 marks). What are the possible outputs of this problem below? You may ignore any output produced by "an_executable". You must consider the scenario when a system call fails (a system call will return -1 if it fails). Clearly specify each scenario and each line of outputs under that scenario.

```
 1 int ret = fork();
 2 if (ret == 0) {
 3   printf ("L1\n");
 4   exec ("an_executable");
 5   printf ("L2\n");
 6 } else if (ret > 0) {
 8   wait();
 9   printf("L3\n");
10 } else {
11   printf("L4\n");
12 }
```

Answer:
Scenarios 1: All system calls succeed:
L1
L3

Scenario 2: fork() succeeds, but exec fails
L1
L2
L3

Scenario 3: fork() fails (exec won't be called)
L4

Question 7 (XX points) Consider the reader/writer problem we discussed in the lecture, where multiple threads are executing reader and writer functions to read/write the shared object. Recall that the correctness criteria is that there can be multiple readers at the same time, but only one writer.

Consider the code snippet below:

```
 1  // number of readers
 2  int readcount = 0;
 3  // mutual exclusion to readcount
 4  Semaphore mutex = 1;
 5  // exclusive writer or reader
 6  Semaphore w_or_r = 1;
 7
 8  writer {
 9    P(w_or_r); // lock out readers
10    Write;
11    V(w_or_r); // up for grabs
12  }
```

```
13
14  reader {
15    P(mutex);        // lock readcount
16    readcount += 1; // one more reader
17    if (readcount == 1) {
18      V(mutex);    // unlock readcount
19      P(w_or_r); // synch w/ writers
20    }
21    Read;
22    P(mutex);       // lock readcount
23    readcount -= 1; // one less reader
24    if (readcount == 0)
25      V(w_or_r); // up for grabs
26    V(mutex);    // unlock readcount}
27  }
```

(a)(4 marks) This code can trigger a **deadlock**, where not a single thread can continue its execution. Show an interleaving that results in a deadlock.

Answer: There could the following context switch:

```
reader 1 {
  P(mutex);        // lock readcount
  readcount += 1; // one more reader
  if (readcount == 1) {
    V(mutex);    // unlock readcount
    P(w_or_r); // synch w/ writers
       ← context switch to another reader

                              reader 2 {
                                P(mutex); <-- At this moment, a deadlock
will occur, no thread will be able to make progress.
                                readcount += 1;
                                 if (readcount == 1) {
                                   // readcount == 2
                                 }
                                 Read;
                                P(mutex); <-- Stuck
```

Note that it can trigger another non-deadlock bug where writers write at the same time of reads, but that's not deadlock.

(b)(4 marks) Fix this bug.

Correct code:

```
reader {
  P(mutex);        // lock readcount
  readcount += 1; // one more reader
  if (readcount == 1) {
---  V(mutex);    // unlock readcount
    P(w_or_r); // synch w/ writers
  }
+++V(mutex);    // unlock readcount
  Read;
  P(mutex);      // lock readcount
  readcount -= 1; // one less reader
  if (readcount == 0)
     V(w_or_r); // up for grabs
  V(mutex);    // unlock readcount}
}
```

(c)(2 marks) One problem with the solution we discussed in the lecture is starvation. Please use no more than 2 sentences to describe the problem.

Answer: Starvation is the problem where if more readers keep come, a writer thread will never get its chance to execute.

(d)(10 marks) Fix the starvation issue by using only one additional mutex (i.e., binary semaphore). You cannot add any other variables (whether local or global ones). Note that you can not make any assumption on the scheduler.

You should not add/change/remove more than **4 lines** of code.

Answer:

```
writer{
    P(writer_waiting_mutex);
    P(w_or_r); // lock out readers
    Write;
    V(w_or_r); // up for grabs
    V(writer_waiting_mutex);
}

reader {
  P(writer_waiting_mutex);
  V(writer_waiting_mutex);
```

```
  P(mutex);       // lock readcount
  readcount += 1; // one more reader
  if (readcount == 1) {
    P(w_or_r); // synch w/ writers
  }
  V(mutex);   // unlock readcount

  Read;
  P(mutex);     // lock readcount
  readcount -= 1; // one less reader
  if (readcount == 0)
     V(w_or_r); // up for grabs
  V(mutex);   // unlock readcount
}
```