UNIVERSITY OF TORONTO

FACULTY OF APPLIED SCIENCE AND ENGINEERING

MIDTERM EXAMINATION, February, 2020
Third Year – Materials
ECE344H1 - Operating Systems
Calculator Type: 2
Exam Type: A
Examiner – D. Yuan

Please Read All Questions Carefully!

*There are **8** total numbered pages, **4** Questions.*
You have 60 minutes. Budget your time carefully!

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

Name: _____

Student ID: _____

	Total Marks	Marks Received
Question 1	36	
Question 2	16	
Question 3	20	
Question 4	28	
Total	100	

Question 1 (36 marks): Multiple choices and short answers.

(1) Rank the following by their speed (i.e., access time), from the fastest to the slowest:

(A) Main memory
(B) CPU cache
(C) Registers
(D) Hard Drive

C, B, A, D

For each of the following questions, circle all correct answers.

(2) Consider printf():

(A) printf() is a system call
(B) printf() is a library call that invokes a system call
(C) printf() is a library call that does not invoke system call
(D) None of the above

B

(3) If CPU does not provide user/kernel mode (i.e., it only has a single mode), then:

(A) We cannot implement a functional OS running untrusted applications
(B) We can implement a functional OS running untrusted applications, but it is slow
(C) None of the above

B

(4) Which of the following is/are protected instruction?

(A) Load (read from memory)
(B) Store (write to memory)
(C) Halt instruction
(D) Modifying the CPU mode bit
(E) call instruction
(F) None of the above

C, D

(5) Two threads in the same process share:

(A) Program counter
(B) Heap
(C) File descriptors
(D) Stack pointer
(E) Global variables
(F) None of the above

B, C, E

(6) Two threads from *different* processes share:
(A) Program counter
(B) Heap
(C) File descriptors
(D) Stack pointer
(E) Global variables
(F) None of the above

F

(7) On process states:
(A) On a uniprocessor machine, multiple processes can be in running state at the same time
(B) Performing I/O could cause a process's state to be changed from ready to waiting
(C) Performing I/O could cause a process's state to be changed from running to waiting
(D) OS uses timer interrupts to prevent a process from using the CPU forever
(E) None of the above

C, D

(8) On a multi-processors machine:
(A) Multiple processes can run in parallel
(B) Multiple kernel-level threads can run in parallel
(C) Multiple user-level threads can run in parallel (assuming they're mapped to a single kernel-level thread)
(D) One processor could be running in user mode while another in kernel mode
(E) None of the above

A, B, D

(9) On process and thread:
(A) A process may contain multiple threads, but each thread belongs to only one process
(B) The process ID of a zombie process can be reused by another process
(C) If the memory usage of process A is twice the size as process B, then the latency of fork() in process A is roughly twice as the latency of fork() in process B.
(D) None of the above

A

(10) On synchronization primitives:
(A) Condition variables are used for a thread to wait outside of the monitor.
(B) Semaphores are used for a thread to wait outside of the monitor.

(C) When V() is called on a semaphore, if there are multiple threads waiting in P(), only one of them will return from P()
(D) When V() is called on a semaphore, if there is no thread waiting, then the next thread that calls P() could still block
(E) When signal() is called for a monitor, if there is no thread waiting, then the next thread that calls wait() could still block
(F) None of the above

A, C, E

(11) For os161 labs:
(A) You program your labs in Java
(B) We use Git as our version control system
(C) OS161 runs on MIPS architecture
(D) None of the above

B, C

(12) For os161:
(A) It uses disabling interrupt to provide critical region
(B) It uses test-and-set to provide critical region
(C) Your lab 2 code (synchronization lab) runs in user mode
(D) None of the above

A

Question 2 (16 marks): Unix shell

Consider the following four attempts to implement a Unix shell. Assume that for waitpid(pid), if pid equal to 0, it will wait for any child process of the calling process to terminate. If the calling process does not have any child process, it sleeps forever.

```
while (1) {
  char *cmd = read_command();
  int child_pid = fork();
  exec(cmd);
  printf("exec failed");
}
```

(A)

```
while (1) {
  char *cmd = read_command();
  int child_pid = fork();
  if (child_pid != 0) {
    waitpid(child_pid);
  } else {
    exec(cmd);
    printf("exec failed");
  }
}
```

(B)

```
while (1) {
  char *cmd = read_command();
  int child_pid = fork();
  if (child_pid != 0) {
    exec(cmd);
    printf(“exec failed”);
  } else {
    waitpid(child_pid);
  }
}
```

(C)

```
while (1) {
  char *cmd = read_command();
  int child_pid = fork();
  if (child_pid == 0) {
    exec(cmd);
    printf(“exec failed”);
  }
}
```

(D)

(1) Is there a correct implementation? If so, which one(s)? No explanations needed.

B

(2) For all the other implementations, explain what happens when a user inputs: "ls" in the shell. Make sure to include the following in your explanation: (1) how many processes are being created, (2) whether the "exec failed" gets printed, and if so, how many times, and (3) what is observed by the user.

A: the terminal creates a child terminal process, and both parent and child become ls. After ls is executed, both processes terminate. "exec failed" is not printed. The user observes that the current directory content gets printed, and then the terminal is lost.

C: the terminal creates a child terminal process, and the parent becomes ls and terminates afterward. The child terminal process calls waitpid(0), which will sleep forever. The user will see ls being executed, but afterward the terminal freezes. "exec failed" won't be printed.

D: the terminal creates a child terminal process, and the child executes ls and terminates. However, the parent process doesn't wait for the child, so user will see that the terminal command line (e.g., $) is garbled with the ls output. "exec failed" won't be printed.

Question 3 (20 marks): Thread operations in OS161
Consider the following code snippets from OS161 and answer the following questions.

```
 1 void thread_XXX (const void *addr) {
 2    curthread->t_sleepaddr = addr;
 3    mi_switch(S_SLEEP);
 4    curthread->t_sleepaddr = NULL;
 5 }

 6 void thread_YYY (const void *addr) {
 7     int i, result;
 8     for (i=0; i<array_getnum(sleepers); i++) {
 9         struct thread *t = array_getguy(sleepers, i);
10         if (t->t_sleepaddr == addr) {
11             // Remove from list
12             array_remove(sleepers, i);
13             i--;
14
15             result = make_runnable(t);
16        }
17     }
18 }
```

(1) (5 marks) What are these two functions? I.e., replace "XXX" and "YYY" with the actual name.

thread_sleep and thread_wakeup

(2) (5 marks) Explain what's `addr` used for.

Each sleeping thread is associated with an addr so that later thread_wakeup wakes up all the threads associated with a particular addr, but not others.

(3) (10 marks) thread_YYY is not very efficient. Why? And how to optimize it?

Instead of iterating through all the sleepers, we can separate the sleeping threads to different queues, one for each addr, so that thread_wakeup only needs to iterate and wake up each thread in the queue that is associated with the addr.

Question 4 (28 marks): Synchronization

Suppose that you write a program that creates 3 threads, each thread executes a function named "`increment()`" that increments a global variable `i` before terminating. Assume the initial value of `i` is 0. Answer the following questions. **No explanations needed for question (2)-(7).**

(1) In C/C++, "`i++`" increments the value of a variable `i`. Is "i++" atomic? Why?

It requires three instructions: load i to a register, increment the register, and store the value back to memory.

(2) Consider the following implementation of `increment()`:

```
void increment() {
    i++;
}
```

Assume a uni-processor machine. When all three threads terminate, what are the possible values of `i`?

(3) Now assume the machine has multi-processors, what are the possible values of `i` at the end?

1, 2, 3

(4) From now on let's always assume the machine has multi-processors. You changed the implementation to the following:

```
void increment() {
    lock_acquire(lock);
    i++;
    lock_release (lock);
}
```

What are the possible values of i at the end? Assume `lock_acquire()` and `lock_release()` are correctly implemented?

3

(5) You changed the implementation again to the following:

```
void increment() {
    i++;
    i++;
}
```

What are the possible values of i at the end?
2, 3, 4, 5, 6

(6) You changed the code to the following:

```
void increment() {
    lock_acquire(lock);
    i++;
    lock_release(lock);
    lock_acquire(lock);
    i++;
    lock_release(lock);
}
```

What are the possible values of i at the end?

6

(7) You changed the code to the following:

```
void increment() {
    lock_acquire(lockA);
    i++;
    lock_release(lockA);
    lock_acquire(lockB); // Note: here we use a different lock
    i++;
    lock_release(lockB);
}
```

What are the possible values of i at the end?

4, 5, 6

This page is intentionally left blank.