UNIVERSITY OF TORONTO

FACULTY OF APPLIED SCIENCE AND ENGINEERING

MIDTERM EXAMINATION, February, 2018
Third Year – Materials
ECE344H1 - Operating Systems
Calculator Type: 2
Exam Type: A
Examiner – D. Yuan

Please Read All Questions Carefully!

There are __10__ total numbered pages, __5__ 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	21	
Question 2	21	
Question 3	21	
Question 4	12	
Question 5	25	
Total	100	

Question 1 (21 marks): Multiple choices. There are one or more correct choices for each question. Mark all of them. You will lose marks for wrong choice.

(1)(3 marks) 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

Answer: B, C, E

(2) (3 marks) 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

Answer: F

(2) (3 marks) In Unix Shell implementation, exec() is used in

(A) parent process and overwrites child address space
(B) child process and overwrites parent address space
(C) child process and overwrites child address space
(D) parent process and overwrites parent address space

Answer: C

(3) (3 marks) If a kernel-level thread executes "cin >> variable_a" (C++ input operation), which of the followings will happen?

(A) The status of all threads in the same process will be changed from Running to Ready
(B) Only the status of this thread will be changed from Running to Ready
(C) The status of all threads in the same process will be changed from Running to Waiting
(D) Only the status of this thread will be changed from Running to Waiting
(E) The status of all threads in the same process will be changed from Ready to Waiting
(F) Only the status of this thread will be changed from Ready to Waiting
(G) The status of this thread will not be changed

Answer: D

(4) (9 marks) Which of the following operations will trigger the CPU to transition from user mode to kernel mode (i.e., it is an event)? If it is an event, further classify it as either exception or interrupt.

- A process divides an integer by zero

(A) Triggers an exception (B) Triggers an interrupt (C) Not an event

A

- A process accesses memory at address 0x00000000

(A) Triggers an exception (B) Triggers an interrupt (C) Not an event

A

- User presses key "a" on the keyboard when using shell

(A) Triggers an exception (B) Triggers an interrupt (C) Not an event

B

- A process executes test-and-set instruction

(A) Triggers an exception (B) Triggers an interrupt (C) Not an event

C

- The email client receives an email

(A) Triggers an exception (B) Triggers an interrupt (C) Not an event

B

- Function A calls function B (B is not a system call) in a process

(A) Triggers an exception (B) Triggers an interrupt (C) Not an event

C

Question 2 (21 marks): OS161

Consider the following code snippet taken from OS161. (The number at the beginning is the line number.)

```
1  mips_switch:
2     addi sp, sp, -44
3     sw   ra, 40(sp)
4     sw   gp, 36(sp)
5     sw   s8, 32(sp)
6     sw   s7, 28(sp)
7     sw   s6, 24(sp)
```

```
17     lw   sp, 0(a1)
18
19     lw   s0, 0(sp)
20     lw   s1, 4(sp)
21     lw   s2, 8(sp)
22     lw   s3, 12(sp)
23     lw   s4, 16(sp)
```

```
 8      sw   s5, 20(sp)
 9      sw   s4, 16(sp)
10      sw   s3, 12(sp)
11      sw   s2, 8(sp)
12      sw   s1, 4(sp)
13      sw   s0, 0(sp)
14
15      sw   sp, 0(a0)
16
```

```
24      lw   s5, 20(sp)
25      lw   s6, 24(sp)
26      lw   s7, 28(sp)
27      lw   s8, 32(sp)
28      lw   gp, 36(sp)
29      lw   ra, 40(sp)
30      addi sp, sp, 44
31      j ra
32     .end mips_switch
```

Now answer the following questions. Each question should be answered with at most 2 sentences.

(1) Which programming language is this code written in? Choose one below:
(A) C (B) Java (C) x86 assembly (D) MIPS assembly (E) None of the above

Answer: D

(2) Explain the purpose of this entire piece of code.

Context switch between 2 processes (or more precisely, kernel threads).

(3) What are stored in a0 and a1?

a0: address of the PCB of the old process.
a1: address of the PCB of the new process.

(4) If we change the order of line 17 and line 19, what would happen?

New process will have wrong s0 value when it is scheduled (s0 will contain value from the old process).

(5) At line 3, what information does ra contain? How is this value stored in ra?

The PC when the event occurred. It was saved by the CPU.

(6) What is the purpose of line 2?

Allocate space on the stack of the process running when the event occurred.

(7) Which of the following is true?

(A) Interrupts are always disabled when this piece of code is executed
(B) Interrupts are always enabled when this piece of code is executed
(C) None of the above

Answer: A

Question 3 (21 marks): Fork bomb

A fork bomb is a program that will fork an infinite number of *long living* processes. Thus it will exhaust the system resource. Note that if a program forks an infinite number of processes, but only a small number of these processes are long living (i.e., others *terminate* quickly), it is not considered as a fork bomb.

Now consider the following three programs, and answer (1) whether it is a fork bomb, and (2) what are the first 3 lines of output.

Make the following assumptions:

- The OS scheduler is first-come-first-serve, i.e., it will schedule the processes in the order of their creation.
- Whenever a process makes a system call, the OS will perform a context switch (as long as there are other *ready* processes), even if the process that is making the system call is older than other ready processes.
- The PID of a newly created process is always the smallest positive integer that is unused (so that when a process is terminated and reaped its PID will be reused).
- Initially, only PID 1 in the system is used (so the first newly created process will have PID 2), and you can assume the process with PID 1 is always sleeping.
- The three programs below are executed independently of each other.

Program 1:

```
int g = 0; // global variable
int main(int argc, char *argv[]) {
    pid_t pid;
    g++;
    if (g < 3) {
        pid = fork();
        if (pid) {
          waitpid(pid);
          exec(argv[0]); // argv[0] contains the pathname of this program
        }
        printf("Process: %d\n", getpid());
    }
}
```

Is this program a fork bomb? Choose one from below:

(A) This program is a fork bomb
(B) This program is not a fork bomb, but it will fork infinite number of processes
(C) This program is not a fork bomb, and it will not fork infinite number of processes
(D) None of the above

Answer: B

Write the first three lines of output below?

Process: 3
Process: 3
Process: 3

Program 2:

```
int main(int argc, char *argv[]) {
    pid_t pid;
    for (int i = 0; i < 3; i++) {
      printf("Process: %d, start iteration: %d\n", getpid(), i);
      pid = fork();
      if (pid == 0)
        exec (argv[0]); // argv[0] contains the pathname of this program

      waitpid(pid);
      printf("Process: %d, end iteration: %d\n", getpid(), i);
    }
}
```

Is this program a fork bomb? Choose one from below:

(A) This program is a fork bomb
(B) This program is not a fork bomb, but it will fork infinite number of processes
(C) This program is not a fork bomb, and it will not fork infinite number of processes
(D) None of the above

Answer: A

Write the first three lines of output below?

Process 2, start iteration: 0

Process 3, start iteration: 0
Process 4, start iteration: 0

Program 3:

```
int main(int argc, char *argv[]) {
    int pid = fork();
    for (int i = 0; pid; i++) {
        waitpid(pid);
        printf ("PID1: %d, PID2: %d, i = %d\n", getpid(), pid, i);
        pid = fork();
    }
}
```

Is this program a fork bomb? Choose one from below:

(A) This program is a fork bomb
(B) This program is not a fork bomb, but it will fork infinite number of processes
(C) This program is not a fork bomb, and it will not fork infinite number of processes
(D) None of the above

Answer: B

Write the first three lines of output below?

PID1: 2, PID2: 3, i = 0
PID1: 2, PID2: 3, i = 1
PID1: 2, PID2: 3, i = 2

Question 4 (12 marks): Condition variable and semaphore

In OS161, suppose we are now using semaphore to implement condition variable. (You only need to consider cv_wait and cv_signal; you do not need to consider cv_broadcast.) Here is the definition of the condition variable:

```
struct cv {
    char* name;
    struct semaphore *sem;
};
```

Consider the following implementation of cv_wait() and cv_signal():

```
struct cv* cv_wait(const cv *cv, struct lock *lock) {
    lock_release(lock);
    P(cv->sem);
    lock_acquire(lock);
}
```

```
struct cv* cv_signal(struct cv* cv, struct lock *lock) {
    V(cv->sem);
}
```

Assume other parts of the code are the same as in OS161. Does this work? Explain your answer briefly. If you think it does not work, give a thread interleaving to show the problem.

No it doesn't work. Here is the interleaving:

```
Thread 1: cv_signal()
                     Thread 2: cv_wait() <- this WON'T cause thread 2 to go
to sleep!
```

Question 5 (25 marks): Multiprocessor synchronization

We have seen how the semaphore primitives P and V are implemented for uniprocessors in OS161. In this problem, we will implement these primitives for multiprocessors. The relevant code for the OS161 version of these primitives is shown below. The `schedule()` function moves the current thread to wait queue and chooses another thread from ready queue to run.

```
P(sem) {
  spl = splhigh();
  while (sem->count==0) {
    thread_sleep(sem);
  }
  sem->count--;
  splx(spl);
}
```

```
V(sem) {
  spl = splhigh();
  sem->count++;
  thread_wakeup(sem);
  splx(spl);
}
```

```
thread_sleep(void *cond) {
  add_to_wait_queue(cond);
  schedule();
}
```

```
thread_wakeup(void *cond) {
  thread = remove_from_wait_queue(cond);
  add_to_ready_queue(thread);
}
```

In the four questions shown below, circle all (one or more) correct answers. In the answers below, “corrupt” means an incorrect update. Be careful, the questions might be harder than you thought, and marks will be deducted for each incorrect answer.

1) Would the semaphore code shown above work for multiprocessors?
(A) No, it could corrupt the sem->count variable.
(B) No, it could corrupt the wait queue.
(C) No, it could corrupt the ready queue.
(D) No, it could cause a thread to wait forever.
(E) Yes, it would work.

Answer: A, B, C, D

2) Your partner argues that the code would clearly not work for multiprocessors and suggests a minimal change: add spin locks in the thread sleep and thread wakeup implementation. (Recall that spin locks use test-and-set instead of disabling interrupt.) No other change is made to other part of the code.

```
thread_sleep(void *cond) {
  spin_lock(schedlock);
  add_to_wait_queue(cond);
  schedule();
  spin_unlock(schedlock);
}
```

```
thread_wakeup(void *cond) {
  spin_lock(schedlock);
  thread = remove_from_wait_queue(cond);
  add_to_ready_queue(thread);
  spin_unlock(schedlock);
}
```

Would the semaphore code shown earlier now work for multiprocessors?
(A) No, it could corrupt the sem->count variable.
(B) No, it could corrupt the wait queue.
(C) No, it could corrupt the ready queue.
(D) No, it could cause a thread to wait forever.
(E) Yes, it would work.

Answer: A, D.
Wait queue and ready queue have no races as they're protected by schedlock, but P and V still have data race. Goes to sleep with schedlock, causing thread wait forever.

3) Your friend from Pepperoni University argues that the code would clearly still not work for multiprocessors and suggests another minimal change (in addition to the change made in part 2): replace interrupt disabling with spin locks in the P and V implementation.

```
P(sem) {
  spin_lock(sem->spinlock);
  while (sem->count==0) {
    thread_sleep(sem);
  }
  sem->count--;
```

```
V(sem) {
  spin_lock(sem->spinlock);
  sem->count++;
  thread_wakeup(sem);
  spin_unlock(sem->spinlock);
}
```

```
  spin_unlock(sem->spinlock);
}
```

Would this semaphore code work for multiprocessors?
(A) No, it could corrupt the sem->count variable.
(B) No, it could corrupt the wait queue.
(C) No, it could corrupt the ready queue.
(D) No, it could cause a thread to wait forever.
(E) Yes, it would work.

Answer: D. Goes to sleep with schedlock and sem->spinlock, causing thread wait forever.

4) Your partner looks at the code, mutters something about Pepperoni Univ., and suggests adding spin unlock and spin lock around thread sleep in the P implementation (together with the change made in part 2 of this question):

```
P(sem) {
  spin_lock(sem->spinlock);
  while (sem->count==0) {
    spin_unlock(sem->spinlock);
    thread_sleep(sem);
    spin_lock(sem->spinlock);
  }
  sem->count--;
  spin_unlock(sem->spinlock);
}
```

Would this semaphore code work for multiprocessors?
(A) No, it could corrupt the sem->count variable.
(B) No, it could corrupt the wait queue.
(C) No, it could corrupt the ready queue.
(D) No, it could cause a thread to wait forever.
(E) Yes, it would work.

Answer: D. Goes to sleep with schedlock, causing thread wait forever.

5) After looking at the your partner's solution in part 4, your friend from Pepperoni U proposes another solution by keeping the new change in part 4 while reverting the changes made in part 2. The complete solution is as follows:

```
P(sem) {
  spin_lock(sem->spinlock);
```
	```
	V(sem) {
	  spin_lock(sem->spinlock);
	```

```
  while (sem->count==0) {
    spin_unlock(sem->spinlock);
    thread_sleep(sem);
    spin_lock(sem->spinlock);
  }
  sem->count--;
  spin_unlock(sem->spinlock);
}
```

```
  sem->count++;
  thread_wakeup(sem);
  spin_unlock(sem->spinlock);
}
```

```
thread_sleep(void *cond) {
  add_to_wait_queue(cond);
  schedule();
}
```

```
thread_wakeup(void *cond) {
  thread = remove_from_wait_queue(cond);
  add_to_ready_queue(thread);
}
```

Would this semaphore code work for multiprocessors?
(A) No, it could corrupt the sem->count variable.
(B) No, it could corrupt the wait queue.
(C) No, it could corrupt the ready queue.
(D) No, it could cause a thread to wait forever.
(E) Yes, it would work.

Answer: B, C, D.

```
B: b/c now wait_queue in thread_sleep is not protected;
C: b/c schedule and add_to_ready_queue access ready queue unprotected.
D: because the following interleaving will cause a V to be lost:

T1                                 T2
P() {
  spin_lock(..);
  while (sem->count==0) {
    spin_unlock(..);
--------------- context switch -----------
                                 V(sem)
                                 spin_lock(..);
                                 sem->count++;
                                 thread_wakeup(sem);
                                 spin_unlock(..);
--------------- context switch -----------
    add_to_wait_queue
    schedule();

T1 sleeps forever!
```