UNIVERSITY OF TORONTO

FACULTY OF APPLIED SCIENCE AND ENGINEERING

Midterm EXAMINATION , February, 2015
Third Year – Materials
ECE344H1 - Operating Systems
Calculator Type: 2
Exam Type: A
Examiner – D. Yuan

Please Read All Questions Carefully! Please keep your answers as concise as possible.
You do not need to fill the whole space provided for answers.

There are 15 total numbered pages, 8 Questions.
You have 2 hours. Budget your time carefully!

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

Name: _____

UTORid: _____

Student ID: _____

Grading Page

	Total Marks	Marks Received
Question 1	22	
Question 2	6	
Question 3	6	
Question 4	19	
Question 5	10	
Question 6	9	
Question 7	28	
Total	100	

Question 1 (22 marks, 2 marks each): True or False *(No explanations needed)*

(a) Application programmers use library functions (e.g., printf()) more often than system calls (e.g., write()) because the former is faster than the latter.

_True X False

(b) One cannot implement an OS without the hardware support of user/kernel mode.

_True X False

(c) Spinlock (i.e., busy waiting) is always less efficient than a blocking wait operation

_True X False

(d) User-level thread should be used when every thread frequently performs I/O operation

_True X False

(e) On a machine with a single core CPU, there can be only one process that is in RUNNING state.

X True ☐False

For each of the following instructions/operations, answer whether it is a protected instruction.

(f) load (to load the content stored at an address into a register)

_Protected instruction X Not a protected instruction

(g) Modifying the values of stack pointer registers

_Protected instruction X Not a protected instruction

(h) Modifying the value of PC (i.e., program counter)

_Protected instruction X Not a protected instruction

(i) Modify the CPU register that controls whether the CPU executes in user-mode or kernel-mode.

X Protected instruction ☐Not a protected instruction

(j) test-and-set

_Protected instruction X Not a protected instruction

(k) splhigh() in OS161

X Protected instruction ☐Not a protected instruction

Question 2 (6 marks) When a divide-by-zero instruction is executed by a process, it crashes. Describe the important steps that take place from the execution of divide-by-zero instruction to the crash of the process.

Answer:
1. CPU detects the divide-by-zero fault
2. CPU saves all the states (e.g., register values), switches to kernel mode and executes the handler function for this fault
3. The handler function (part of the OS kernel) checks the fault, finds out that the user process does not register a handler for this fault
4. Therefore the handler function sends a signal to the user process to have it killed.

Question 3 (6 marks) Here is the implementation of the 'open (path, flags, mode)' system call in FreeBSD:

```
open: ; FreeBSD convention:
      ; parameters via stacks.
   push dword mode
   push dword flags
   push dword path
   mov eax, 5
   push dword eax ; syscall number
   int 80h
   add esp, byte 16
```

(a)(3 marks) Who executes this code snippet? User-level processes or OS kernel?
Answer: user-level process will call open.

(b)(3 marks) What is the purpose of "add esp, byte 16" at the last line?
Answer: It is to rewind the stack to delete the "mode, flags, paths, and 5 (syscall number)".

Question 4 (19 marks) Consider the following code snippet:

```
// program 1
for (i = 0; fork(); i++) {
   if (i == 4) { break; }
   printf ("PID: %d, i = %d\n", getpid(), i);
}
```

Assume that initially the process ID executing this code is 2394, and the PID always increases sequentially by 1. For example, the first time this process calls fork() the PID of the child is 2395. You can also assume no other processes in the system are calling fork() after this program starts to execute.

(a)(4 marks) What are the possible outputs of this program?
PID: 2394, i = 0
PID: 2394, i = 1
PID: 2394, i = 2
PID: 2394, i = 3

This is the only possible output it can have.

(b)(3 marks) During the entire lifespan of this program, how many times is “fork()” being called?
Answer: fork() is called 5 times.

Now consider this code snippet:

```
// program 2
for (i = 0; i<4; i++) {
  if (fork()) { break; }
  printf ("PID: %d, i = %d\n", getpid(), i);
}
```

Under the same assumption as above, answer the following questions.

(c)(4 marks) What are the possible outputs of this program?
PID: 2395, i = 0
PID: 2396, i = 1
PID: 2397, i = 2
PID: 2398, i = 3

This is the only possible output.

(d)(3 marks) During the entire lifespan of this program, how many times is “fork()” being called?
Answer: 4 times.

(e)(5 marks) If you run these two programs from a command line shell (e.g., bash), you will notice one difference: when running program 1, your shell’s next prompt will always appear after the last output line of the program, while for program 2 the next prompt may appear in the middle of the outputs of your program. For example, the output can be like:

```
user@machine$ ./program2.out
PID: XXX, i = XXX
user@machine$
PID: XXX, i = XXX
.. ..
```

```
user@machine$ ./program1.out
PID: XXX, i = XXX
PID: XXX, i = XXX
.. ..
user@machine$
```

Why?

Answer: This is because the shell only waits for the first process it creates (i.e., process with PID = 2394 in this case) to finish. In program 1, all the output lines are from process 2394, therefore the bash only prints the prompt after the last line of output is printed. In program 2, however, all the 4 lines of output are from the children of process 2394. Since the shell only waits for the termination of 2394, and 2394 does not waits for its children to terminate, therefore the shell can print the next prompt before the other children processes.

Question 5 (10 marks) Consider the thread_fork() function in OS161:

```
/* Make a new thread, which will start executing at "func".  The
 * "data" arguments (one pointer, one integer) are passed to the
 * function. */
int thread_fork(const char *name,
                void *data1, unsigned long data2,
                void (*func)(void *, unsigned long),
                struct thread **ret);
```

Now consider the following code:

```
void func () {
  int i;
  for (i = 0; i < 10; i++)
    thread_fork("test", (void *) &i, 0, thread_func, NULL);
}
void thread_func(void *ptr, unsigned long dummy) {
  int id = *((int *)ptr);
  kprintf("id = %d\n", id);
}
```

(a)(5 marks) If we define the correctness criteria for this program is that it should output 10 lines, each prints a unique integer value between [0, 9], is this program correct? Why?

Answer: Not correct, because the parameter passed in to "thread_func" is an address of a local variable in func. By the time thread_func gets executed, the content of this memory, which is "i" in func(), might have been modified since the time a thread_fork was called. We can end up with output like:

id = 2
id = 2
.. ..

(b)(5 marks) If we change the code to the following:

```
void func () {
  int i;
  for (i = 0; i < 10; i++)
    thread_fork("test", NULL, (unsigned long) i, thread_func, NULL);
}
void thread_func(void *ptr, unsigned long i) {
  int id = (int) i;
```

```
  kprintf("id = %d\n", id);
}
```

Is this program correct? Why?

Answer: Yes, this version of the program works correctly. This is because now "i" is passed in by value, where another copy of the value i is created and passed into thread_func.

Question 6 (9 marks) Locks should be used to protect the critical regions like the following:

```
acquire(lock);
Critical region...
release(lock);
```

For each of the lock implementations below, choose *one or more* correct choices from the following (no explanations needed):

A. It works on single-core machines because it disables context switches within critical region
B. It works on single-core machines even when there are context switches within critical region
C. It does not work on single core CPU
D. It does not work on multi-core CPU

(a)(3 marks) Lock implementation 1:

```
int lock;

void acquire (lock) {
    while (lock);
    lock = 1;
}
void release (lock) {
    lock = 0;
}
```

Answer: C, D.

(b)(3 marks) Lock implementation 2:

```
int lock;

void acquire (lock) {
    while (test-and-set(&lock));
}
void release (lock) {
    lock = 0;
}
```

Answer: B

(c)(3 marks) Lock implementation 3:

```
void acquire (lock) {
    disable interrupts;
}
void release (lock) {
    enable interrupts;
}
```

Answer: A, D.

Question 7 (28 marks) 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:

```
// number of readers
int readcount = 0;
```

```
// mutual exclusion to readcount
Semaphore mutex = 1;
// exclusive writer or reader
Semaphore w_or_r = 1;

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

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

(a)(5 marks) Does it work? Why? If it doesn't work, fix the bug.

Answer: No it doesn't work.

Reason: there could the following context switch:

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

                          reader 2 {
                            P(mutex);
                            readcount += 1;
```

```
                                  if (readcount == 1) {
                                    // readcount == 2
                                  }
                                  Read;
                                   ← context switch to a writer
```

Now the writer can proceed to write because w_or_r is still open. This violates the correctness semantic.

Another problematic situation:

```
reader {
  P(mutex);       // lock readcount
  readcount += 1; // one more reader
  if (readcount == 1) {
    V(mutex);   // unlock readcount
    P(w_or_r); // synch w/ writers
  }
  Read;
            ← context switch to another reader
                             reader 2 {
                               P(mutex);
                               readcount += 1;
                                if (readcount == 1) {
                                  // skip, since readcount == 2
                                }
                                Read;
                                P(mutex) ← now, it's a deadlock!
```

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}
}
```

(b)(3 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.

(c)(20 marks) How do you fix the starvation problem? Write the code. Note that you can not make any assumption on the scheduler. In other words, you can only change the writer and reader functions. You can also add additional synchronizations, but you are only allowed to use locks or semaphores.

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
}
```