ECE344 Operating Systems: Midterm (Winter 2013)

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 **10** total numbered pages, **10** Questions.*
You have 2 hours. Budget your time carefully!

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

Name: _____**Solution**_____

UTORid: _____

Student ID: _____

Grading Page

	Total Marks	Marks Received
Question 1	10	
Question 2	10	
Question 3	8	
Question 4	8	
Question 5	8	
Question 6	10	
Question 7	8	
Question 8	10	
Question 9	20	
Question 10	8	
Total	100	

1. **(10 marks)** Recall that protected instructions can only be executed in the kernel mode (i.e., can only be executed by the OS but not by user-level processes). For each instruction below, is it a protected instruction? ***(No explanations needed).***

   (A) load instruction (read a value from memory into a register) (2 marks)
   No. Load, modify PC, and modify SP are instructions that need to be executed by any running user-level process. Therefore it is not a protected instruction.

   (B) modify the PC register (program counter) (2 marks)
   No.

   (C) modify the SP register (stack pointer) (2 marks)
   No.

   (D) modify the register that controls kernel/user mode (2 marks)
   *Yes. Otherwise **any** process can make itself run in kernel mode.*

   (E) direct access I/O device (2 marks)
   Yes. I/O devices are only directly accessible by the OS.

2. **(10 marks)** Most hardware provides user mode and kernel mode; user processes are run in user mode, while the OS runs in kernel mode. The tricky part is transitioning from user mode to the kernel mode, which is caused by an event.

   (A) Please give two examples of events, and pick one example to further explain what takes place during that event. (6 marks)
   Examples of events: Exception, fault, interrupt, system call, software interrupt, and any specific example of any of them. (2 marks)

   For any events, once the CPU detects it, the following sequence of actions will take place:
   1. *switch from user-mode to kernel-mode (set Mode bit to kernel mode)*
   2. *Saves user process's states (PC, registers, etc.)*
   3. *Call the specific event handler*
   4. *Restore the process's states*
   5. *Set mode bit to User Mode*

   Grading policy (4 marks total): 2 marks for each of the above steps, needs to cover at least 2 to get full marks.

   (B) On all current computers, at least part of OS's event handling sequence is written in assembly language, instead of higher-level language like C. Why? (4 marks)
   Many of the above operations, such as step 1 and 2 above, are architecture specific (for example, how many registers each CPU has is architecture specific), therefore has to be written in assembly code.

   Note: "programs written in assembly are more efficient" is not a correct answer for this problem.

3. **(8 marks)** Timer interrupts are a useful mechanism for the OS. Why?

It allows the OS to retain control of the CPU.

4. **(8 marks)** You write a UNIX shell, but instead of calling fork() then exec() to launch a new job, you instead insert a subtle difference: the code first calls exec() and then calls fork() like the following:

```
shell (..) {
  .. ..
  exec (cmd, args);
  fork();
  .. ..
}
```

Does it work? What is the impact of this change to the shell, if any? (Explain)

Doesn't work (2 marks). Shell's address space is entirely replaced with the new comman (cmd), therefore the shell will terminate once cmd is terminated.

5. **(8 marks)** If a multithreaded process forks, a problem occurs if the child gets a copy of all the parent's threads. Suppose that one of the original threads was waiting for keyboard input. Now after fork(), two threads are waiting for keyboard input, one in each process. Will this problem ever occur in single-threaded processes (i.e., each process has only one thread)? Why?

No it won't occur in single-threaded processes: the problem will occur if the fork() can overlap with the keyboard read. When read() is called, it will cause the calling thread to block until the keyboard input arrives. In a multi-threaded process, another thread within the same process could make a fork(), overlapping with the outstanding read(). Then a child process will be created, and now both the parent and the child have one thread with an outstanding read(), waiting for the next keyboard input. ***But from the OS perspective, it only received one read() call (from the parent).*** *Here the problem occurs: when the next keyboard input arrives, the OS can only deliver this to either the parent or child, unblocking one of them, but not both, because again the OS only received one read() call. So either the read() in parent or the child will never be unblocked. The fundamental reason for this is that the fork() is overlapping with the read().*

However, if it is single-threaded process, there is just one execution unit in the parent process, so the fork() occurs either before the read() completes or or after it completes, but the fork() can never overlap with the read(). In the former case, the OS will receive two read() calls, and in the latter case, the OS receives one read() call, serves it, and then fork() occurs.

Grading policy: 1 mark if only the "No" is right but the explanation is wrong.

6. **(10 marks)** Kernel-level thread versus user-level thread.

   (A) Assume you want to implement a web-server for YouTube by using multithreading, where each thread serves one incoming request by loading a video file from the disk. Assume the OS only provides the normal blocking `read` system call for disk reads, do you think user-level threads or kernel-level threads should be used? Why? (5 marks)

Kernel-level threads. Because each thread will make blocking I/O calls. With kernel-level thread, one thread won't block others. But if user-level thread is used, then one thread will block all other threads. (4 marks)

   (B) Now you want to implement a web-server for Facebook, to serve each user's "Home" page (the first page you see after you log in). This time your web-server needs to perform many tasks: load the news feeds from each of your friends, load the advertisement, check for new messages, etc. Now you want to implement your web-server by using multithreading, and have one thread to perform each of the tasks, and later these threads will cooperate with each other to collectively construct the "Home" page. For performance reasons, Facebook makes sure that all the data these threads need is already cached in the memory (so they don't have to perform any disk I/O). Do you think user-level threads or kernel-level threads should be used? Why? (5 marks)

User-level thread. Here since the concern of user-level thread, namely "one thread can block all other threads within the same process", no longer exists (as threads won't make blocking I/O calls), so we can use user-level thread for its efficiency. This is in particular beneficial since these threads needs to communicate frequently with each other. If kernel-thread is used, everytime such communication needs to go through the kernel, which is more expensive.

7. **(8 marks)** A spin lock acquire() can be implemented with a test-and-set instruction as follows:

```
while (test-and-set(&lock->held) == 1)
   ; // spin
```

Recall that `test-and-set()` returns the old value at the address while atomically setting it to 1. Now a new lock acquire() is implemented as follows:

```
1: while (1) {
2:  while (lock->held > 0)
3:      ; // spin
4:  if (test-and-set(&lock->held) == 0)
5:      return;
6: }
```

Does it work? How does it change the behavior of the lock compared to the first implementation?

It works (3 marks). It spins until it thinks the lock is free (line 2-3), then uses atomic test-and-set to acquire the lock. If this atomic lock acquisition failed, it goes back to the outer while loop to try again.

Change of behavior (5 marks): compared to the first implementation, this implementation it will call "test-and-set" much less frequently, since it will first spin until it thinks the lock is free (line 2-3).

In fact, this implementation is called "test+test-and-set", which is the actual implementation in the real world locks. The reason is that 'test-and-set' instruction is pretty expensive (imagine that for it to work on multi-processor machins it might need to stop all the other processors' execution).

8. **(10 marks)** Recall the Bounded Buffer problem we discussed in the lecture: There is a set of resource buffers shared by producer and consumer threads. Producer inserts resources into the buffer set, and consumer removes resources from the buffer set. A producer goes to sleep if the buffer is full, and a consumer goes to sleep if the buffer is empty.

(A) In class we discussed this broken solution:

```
Semaphore mutex=1; //mutual exclusion to shared buffers
Semaphore empty=N; //count of empty buffers(all empty to start)

producer {
 while (1) {
   Produce new resource;
   P(empty); // wait for empty buffer
   P(mutex); // lock buffer list
   Add resource to an empty buffer;
   V(mutex); // unlock buffer list
 }
}

consumer {
 while (1) {
   P(mutex);      // lock buffer list
   Remove resource from a full buffer;
   V(mutex); // unlock buffer list
   V(empty); // note an empty buffer
   Consume resource;
 }
```

```
}
```

Describe why this solution is broken, and demonstrate it with a specific example of thread interleaving. How do you fix this? (5 marks)

It cannot block consumers from coming in. So imagine this interleaving:
// buffer is empty..
Consumer
Consumer
.. ..
It is clearly wrong as consumers should be blocked when buffer is empty. (2 marks)

To fix, we need another semaphore: full. (3 marks)

```
Semaphre full = 0; // count of full buffers
consumer {
  while (1) {
    P(full);        // wait for a full buffer
    P(mutex);    // lock buffer list
    Remove resource from a full buffer;
    V(mutex); // unlock buffer list
    V(empty); // note an empty buffer
    Consume resource;
  }
}

producer {
  while (1) {
    Produce new resource;
    P(empty); // wait for empty buffer
    P(mutex); // lock buffer list
    Add resource to an empty buffer;
    V(mutex); // unlock buffer list
    V(full);      // note a full buffer
  }
}
```

(B) Assume we modify the code above a little bit, that we reverse the order of "`P(empty)`" and "`P(mutex)`" in the producer code and have the following implementation:

```
producer {
 while (1) {
   Produce new resource;
   P(mutex); // lock buffer list
   P(empty); // wait for empty buffer
   Add resource to an empty buffer;
   V(mutex); // unlock buffer list
 }
}
/* All other code remains unchanged from part (A) */
```

This change will introduce another problem (different from the one you described in part (A)). What is it? Demonstrate it with a specific example of thread interleaving. (5 marks)

It will cause a deadlock. Consider this interleaving:

Producer // empty = N
Producer // empty = N-1
...
Producer // empty = 1
// Now the buffer is full, empty = 0
Producer {
P(mutex) // succeed
P(empty) // empty = 0, block
..
}

Now this producer is blocked with mutex held, so later even other consumers come in, they won't be able to remove an resources as they won't be able to acquire the mutex!

9. **(20 marks)** In OS/161, the semaphore functions are implemented as below:

```
 1 void                                  void
 2 P(struct semaphore *sem)             V(struct semaphore *sem)
 3 {                                    {
 4   spl = splhigh();                     spl = splhigh();
 5   while (sem->count==0) {              sem->count++;
 6     thread_sleep(sem);                 thread_wakeup(sem);
 7   }                                    splx(spl);
 8   sem->count--;                      }
 9   splx(spl);
10 }
```

(A) Sys161 (the simulated hardware OS161 runs on) is a uniprocessor machine, therefore the code above works. Does this code work on multiprocessor machines? Why? (5 marks)

No. Because even with interrupt off on all processors, multiple processors can execute multiple threads at the same time and accessing the shared memory at the same time.

(B) What is the purpose of the argument passed to `thread_sleep()`? (5 marks)
The argument (the semaphore) serves as and "ID"

This semaphore variable is used so later a thread_wakeup(sem) can wake up those threads that sleep on the same semaphore variable "sem".

(C) Can we change the line 5 of P() from "while (sem->count==0)" to "if (sem->count==0)"? If yes, why? If not, show a specific example of thread interleaving. (5 marks)

No. Because "thread_wakeup" will wake-up all the threads, and if it is changed to "if", then more than one thread can mistakenly enter a critical section. Consider the following interleaving:
// assume binary semaphore

T1:	*T2:*	*T3:*
P (mutex) // succeeds		
Critical section		
	// sem->count is 0 now	
	P (mutex):	
	splhigh();	
	if (sem->count==0)	

thread_sleep(sem); ← *sleep*

// sem->count is 0 now
P (mutex):
splhigh();
if (sem->count==0)
thread_sleep(sem); ← *sleep*

V (mutex):
..
thread_wakeup(sem);
← *wakeup T2 & T3*
// V ends

// thread_sleep returns
P returns, critical section

// thread_sleep returns
P returns, critical section

(D) If we switch the order of "sem->count++" and "thread_wakeup(sem)" in V() as below, does it work in OS/161? P() is unchanged. If yes, why? If not, show a specific example of thread interleaving. (5 marks)

```
 1 void                                  void
 2 P(struct semaphore *sem)              V(struct semaphore *sem)
 3 {                                     {
 4   spl = splhigh();                      spl = splhigh();
 5   while (sem->count==0) {               thread_wakeup(sem);
 6     thread_sleep(sem);                  sem->count++;
 7   }                                     splx(spl);
 8   sem->count--;                       }
 9   splx(spl);
10 }
```

It still works, since these two changed lines are within the critical region (interrupt disabled), so a context switch cannot happen in between.

10. ***(8 marks)*** *Observe the following multithread-safe list insertion code:*

```
typedef struct __node_t {
  int key;
  struct __node_t *next;
} node_t;

mutex_t m = PTHREAD_MUTEX_INITIALIZER;
node_t *head = NULL;

int List_Insert(int key) {
  mutex_lock(&m);
  node_t *n = malloc(sizeof(node_t));
  if (n == NULL) {
    return -1;  // failed to insert
  }
  n->key = key;
  n->next = head;
  head = n; // insert at head
  mutex_unlock(&m);
  return 0; // success!
}
```

The code has some problems. In this question, you need to insert a change to fix the code and make it work correctly (feel free to augment the code above with your answer).

Solution 1: "mutex_unlock (&m)" before "return -1"
Solution 2: critical region is unnecessarily large. Before "n->next = head", everything are local variables and not shared. So we can do this:

```
int List_Insert(int key) {
- mutex_lock(&m);
  node_t *n = malloc(sizeof(node_t));
  if (n == NULL) {
    return -1;  // failed to insert
  }
  n->key = key;
+ mutex_lock(&m);
  n->next = head;
  head = n; // insert at head
  mutex_unlock(&m);
  return 0; // success!
}
```

Grading policy: both answers are correct and will receive full marks. Since the 2nd solution is better, we give 2 bonus points if you come up with the 2nd answer.