UNIVERSITY OF TORONTO

FACULTY OF APPLIED SCIENCE AND ENGINEERING

MIDTERM EXAMINATION, Feb, 2014 **[SOLUTION]**
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 **13** 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	20	
Question 2	10	
Question 3	8	
Question 4	10	
Question 5	12	
Question 6	10	
Question 7	10	
Question 8	20	
Total	100	

Question 1 (20 marks, 2 marks each): True or False *(No explanations needed)*

(a) Disabling interrupt can only be performed in kernel mode
XTrue ☐False

(b) When transitioning from user-mode to kernel-mode, hardware (CPU) has to save the program counter (PC)
XTrue ☐False

(c) Timer interrupts are mainly used by OS to show the time
☐True XFalse

(d) On a typical OS, most of the processes are in *Running* state
☐True XFalse

(e) Different threads do not share stacks, but they do share the same heap
XTrue ☐False

(f) Under normal conditions, exec() system call will never return.
XTrue ☐False

(g) Without copy-on-write feature, an OS will not behave correctly for fork() system call
☐True XFalse

(h) OS161 uses test-and-set to implement semaphore
☐True XFalse

(i) In OS161, calling P() on a closed semaphore (i.e., sem->count == 0) will cause OS161 to do a *context switch* instead of making the calling thread *spinning*
XTrue ☐False

(j) A program containing a race condition will always result in data corruption or some other incorrect behavior
☐True XFalse

Question 2 (10 marks): Architecture support

What will happen if a user-mode process tries to execute a protected instruction?

Answer: the CPU will detect the fault, and transfer the control to the OS. (The fault is general protection fault on x86, but the name of this fault is not required to get the full mark).

Question 3 (8 marks): System calls

On a normal OS, processes make a system call to invoke OS services. What happens during a system call that is different than a typical procedure call?

Answer: for system call, there will be a *mode switch* from user-mode to kernel mode, and the *control is transferred by special instruction* (int80h in x86) instead of a normal call instruction.

Question 4 (10 marks): Process management

An OS manages the PCBs by storing them into separate queues instead of one single queue. How does the OS separate the PCBs into different queues? Why does it separate them into different queues instead of using a single queue?

Answer: 1. OS separates the PCBs by the different states processes are in.

2. the reason is that later when an external event arrives, the OS only needs to go into the corresponding queue to search for the appropriate process; and when scheduler is called, it only needs to search from the ready state queue.

Question 5 (12 marks): UNIX Shell

Consider a buggy UNIX shell code like the following:

```
1 shell (..) {
2   while (1) {
3     char *cmd = read_command();
4     int child_pid = fork();
5     if (child_pid != 0) {
6       Manipulate STDIN/OUT/ERR file descriptors for pipes, redirection, etc.
7       exec(cmd);
8       panic("exec failed");
9     } else {
10      waitpid(child_pid);
11    }
12  }
13 }
```

Assume waitpid() will do nothing and return immediately on a child_pid that does not exist or is not a child of the calling process. Now answer the following questions:

(a)(4 marks) Compare this code with the shell implementation we discussed in the lecture, where is the bug (which line)?

Answer: the bug is at line 5. It should be if (child_pid == 0).

(b)(4 marks) Assume you start this shell program, and type “ls” once. What will happen?
Answer: what happens is that the parent process (shell) will execute ls and terminate (goes into zombie state).

The child process will call waitpid(0), and return immediately since 0 is not a valid PID. It then goes back into the loop and effectively becomes a shell.

(c)(4 marks) With this shell implementation, what is its impact, if any, on a *long running* OS?

Answer: The above process will repeat on every user-command. ~~There will be a lot of zombie processes --- and the OS will eventually run out of PID. Thus no new processes can be created.~~ Since the parent process will die before the child (the child always first becomes the shell), the child will be adopted by init process, which will wait the child to terminate and then reap it.

Question 6 (10 marks): User and kernel-level threads

(a)(5 marks) What are the advantages and disadvantages of each?

Answer: See slide 18 in lec4_threads lecture.

(b)(5 marks) Assume we can make system calls as fast as procedure calls using some new hardware mechanism (any other kernel-user mode switches are also fast). Would this make one kind of thread clearly preferable over the other? Explain briefly.

Answer: kernel threads would be preferable. the primary disadvantage of kernel threads is the overhead of trapping into kernel to manipulate them, context-switch, etc., and when that is no longer a problem, kernel threads won't have disadvantage compared with user-level threads and they have additional advantages.

Question 7 (10 marks): Atomic instructions

(a)(4 marks) What is an atomic instruction (e.g., test-and-set)? Why do CPU designers provide such instructions?

Answer: an atomic instruction is an instruction where the hardware guarantees the atomicity of its steps. The reason to provide atomic instructions is that to implement synchronization OS needs atomicity support from the hardware.

(b)(6 marks) Besides test-and-set, another atomic instruction that is commonly supported by CPUs is compare-and-swap. The pseudocode below illustrates how it works:

```
bool compare_and_swap (bool* addr, bool oldval, bool newval) {
  bool old_reg_val = *addr;
  if (old_reg_val == oldval)
     *addr = newval;
  return old_reg_val;
}
```

Basically it compares the current value in addr with oldval, and if they're the same, it further sets *addr to newval. Regardless of the comparison result, it will return the value stored in addr before the comparison.

Similar to test-and-set, the underlying hardware will guarantee that these steps are performed atomically.

Now, given compare_and_swap, how do you implement lock_acquire and release? *Note the only hardware support you can only use is compare_and_swap(). Both spin-lock and sleep-based semantics are acceptable.* Write your code below:

```
struct lock {
    bool held = false;
}
```

```
void acquire (lock) {
   // write your code here
   while (compare-and-swap(&lock->held, false, true));
```

```
}

void release (lock) {
    // write your code here
    lock->held = false;
```

```
}
```

Question 8 (20 marks): Threads and synchronization

Now that you have finished the lab assignment 1, consider the following programming problem. *Assume the semantics of the synchronization libraries (e.g., lock_acquire/release, cv_wait/broadcast/signal) are exactly the same as the ones in lab assignment 1.*

(a)(6 marks) Consider the Bounded Buffer (producer/consumer) problem: there are *multiple* producer threads and *multiple* consumer threads. All of them operate on a shared bounded buffer. A producer inserts data into the buffer one item at a time, and a consumer removes data from the buffer one item at a time. The correctness criteria is that the producers cannot insert into a buffer that is full, and consumers cannot remove any items from an empty buffer.

Now consider the following code:

```
void *producer(void *arg) {
  while (true) {
    lock_acquire(&lock);
    while (number_of_items(buffer) == MAX) // MAX is the buffer length.
      cv_wait(&cv, &lock);
    insert_item(buffer); // inserts an item into shared buffer
    cv_broadcast(&cv);
    lock_release(&lock);
  }
}

void *consumer(void *arg) {
  while (true) {
    lock_acquire(&lock);
    while (number_of_items(buffer) == 0)
      cv_wait(&cv, &lock);
    tmp = consume_item(buffer); // removes an item from the shared buffer
    cv_broadcast(&cv);
    lock_release(&lock);
    kprintf("%d\n", tmp);
  }
}
```

Assume the lock and condition variable are properly initialized. Now, does this code work correctly? Why?

Answer: yes. This code seemingly makes the mistake of only using one condition variable in the producer/consumer problem, but solves it by using broadcast to wake all waiting threads. Because each thread re-checks the condition when awoken, only those

who should be able to make progress will do so. Of course, this can be inefficient, which is why we normally use two CVs and signal accordingly thus waking up only those who need to be awoken (solution to part (c)).

(b)(6 marks) Now let's make a small change to the code above: replace "`cv_broadcast`" with "`cv_signal`". The modified code is as below:

```
void *producer(void *arg) {
  while (true) {
    lock_acquire(&lock);
    while (number_of_items(buffer) == MAX) // MAX is the buffer length.
      cv_wait(&cv, &lock);
    insert_item(buffer); // inserts an item into shared buffer
    cv_signal(&cv); // here!
    lock_release(&lock);
  }
}

void *consumer(void *arg) {
  while (true) {
    lock_acquire(&lock);
    while (number_of_items(buffer) == 0)
      cv_wait(&cv, &lock);
    tmp = consume_item(buffer); // removes an item from the shared buffer
    cv_signal(&cv); // here!
    lock_release(&lock);
    kprintf("%d\n", tmp);
  }
}
```

Does this code work correctly? Why?

Answer: it no longer works. Consider this case: assume now multiple producers and consumers are waiting on the condition variable, and only one consumer is running. Now at this point, the buffer has only one item left, the consumer consumes it, and wakes up one blocking thread. But since the decision is on the scheduler, it might pick another consumer thread to be waken up. Now this thread wakes up, but find the buffer is empty, so goes back to waiting state. At this point, all threads are waiting, and we reach a deadlock situation.

(c)(8 marks) Can you write an optimized producer/consumer code that is both correct and faster than the code in (a) and (b)? *The synchronization libraries you can use are limited to the ones used above (i.e., "cv_wait", "cv_signal", "cv_broadcast", "lock_acquire/release"). No other synchronizations, such as semaphores, are allowed.* Write your code below and briefly explain why the performance is better:

Answer:
The problem we try to address is the inefficiency with only one condition variable (as described in the answer in part b). We solve it by using two condition variables: full and empty.

```
void *producer(void *arg) {


 while (true) {
  lock_acquire(&lock);
  while (number_of_items(buffer) == MAX)
    cv_wait(&full, &lock);
  insert_item(buffer);
  cv_signal(&empty);
   // wake up one blocking consumer!
  lock_release(&lock);
 }

}
```

```
void *consumer(void *arg) {


 while(true) {
  lock_acquire(&lock);
  while (number_of_items(buffer) == 0)
    cv_wait(&empty, &lock);
  tmp = consume_item(buffer);
  cv_signal(&full);
  // wake up one blocking producer
  lock_release(&lock);
  printf("%d\n", tmp);
 }
}
```

Note: we can use signal now because a producer is guaranteed to wake up a consumer, and vice-versa.