CSL373/CSL633 Minor 2 Exam
Operating Systems
Sem II, 2013-14

Answer all 4 questions Max. Marks: 34

1. True or False. Explain briefly. No marks for no explanation/incorrect explanation.

a. Using spinlocks on uniprocessor systems will result in a deadlock. If true, explain why. If false, explain why not. You may want to discuss how spinlocks are implemented to justify your answer. [2]

b. Blocking locks are more efficient than spinlocks as they avoid wasting CPU cycles. Justify your answer by providing details on why/why not or when/when not. [2]

c. Reader-Writer locks must be implemented as blocking locks, i.e., they cannot be implemented as spinlocks. Justify your answer by providing details on why/why not or when/when not. [2]

d. Consider the following function foo() that takes two arguments which are pointers to shared objects:

```
void foo(shared_object *a, shared_object *b)
{
   ….
   acquire(&a->lock);
   ….
   if (some_condition) {
      release(&a->lock);
      acquire(&b->lock);
   }
   ….
   ….
   if (some_condition) {    // this condition is the same as that used in first ‘if’ statement
      release(&b->lock);
   } else {
      release(&a->lock);
   }
   ….
   return;
}
```

This code can deadlock if executed by multiple threads (with potentially different arguments). If true, explain a situation where it can deadlock. If false, explain why you think it will never deadlock. [3]

e. In xv6, while a process is executing in user mode, the corresponding trapframe pointer (p->tf) stored in the corresponding process control block (struct proc) must point to the top of the processes' kstack. If true, explain why. If false, explain why not and say what should be the value of p->tf at that time. [3]

f. In xv6, the 'RUNNING' state to represent the current state of a process (in proc->state) is redundant, i.e., instead it would have sufficed to simply use the 'RUNNABLE' state for any process that is currently running. If this is true, explain why. If this is false, explain what could go wrong, if we used the RUNNABLE state for a process that is currently running. [2]

2. Implementing spinlocks.

In class, we discussed an implementation of a the lock acquire() routine using the atomic ‘xchg’ instruction, available on x86. Assume that an architecture does not provide the ‘xchg’ instruction. Instead, it provides a ‘test_and_set’ instruction with the following syntax/semantics:

Syntax:

```
test_and_set    <memory-address>,   <Register>
```

Semantics:

This instruction atomically tests the current value at <memory-address>, and if it is zero, then sets it to one. If the tested value is non-zero, then it leaves it unchanged. It also saves the tested value at <memory-address> in the <Register> (second operand). This whole operation is atomic with respect to other accesses to the same memory address.

For example, if initially, the contents at memory-address 0x123456 are 0, then the execution of the following instruction:

```
test_and_set    0x123456, %eax
```

will result in the value at address 0x123456 to change to 1; also, the value of the register %eax will be set to 0 (the tested value).

On the other hand, if the initial contents at memory address 0x123456 are 1, then the execution of the same instruction (test_and_set 0x123456, %eax) will result in the value at 0x123456 to remain unchanged, while the register %eax will be set to 1 (the tested value).

Use the ‘test_and_set’ instruction to implement spinlocks. In particular, implement the ‘acquire()’ and ‘release()’ routines.   [6]

3. Condition Variables

Consider the producer consumer queue example, implemented using condition variables and locks, as discussed in class. Assume that there is only a single producer and a single consumer thread. In this case, the following code correctly synchronizes accesses to the shared queue:

```
char q[MAX];
int head = 0, tail = 0;
struct lock mutex;

void produce(char c) {
    acquire(&mutex);
    if (/*  queue is full  */) {
        wait(&not_full, &mutex);
    }
    /*  produce an element  */
    notify(&not_empty);
    release(&mutex);
}

char consume(void) {
    acquire(&mutex);
    if (/*  queue is empty  */) {
        wait(&not_empty, &mutex);
    }
    /* consume an element */
    notify(&not_full);
    release(&mutex);
}
```

a. This code is correct if there is only one producer and only one consumer. But it becomes incorrect if there are multiple producers or multiple consumers. Why does it become incorrect with multiple producers or multiple consumers. [2]

b. How will you change the code so that it becomes correct, even in the presence of multiple producers and multiple consumers? [2]

c. Is it okay to release the mutex before calling notify, as follows (in either the producer code or in the consumer code or both)?

```
void produce(char c) {
   acquire(&mutex);
   if (/*  queue is full  */) {
      wait(&not_full, &mutex);
   }
   /*  produce an element  */
   release(&mutex);
   notify(&not_empty);  //the notify statement has been moved after the mutex release
}
```

If it is okay, explain why. If it is not okay, discuss the problem. [6]

4. In xv6, the ‘fetchint()’ function [3267] is used to fetch an integer at a memory address that was provided by a user program (e.g., an argument to a system call).

a. Why does this function first check the value of the memory address against proc->sz [3269] before dereferencing it [3271]?   [2]

b. Why does this function check both ‘addr’ and ‘addr + 4’ against proc->sz [3269]? [2]