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]

False. Interrupts will cause the process to be switched out, even if it is spinning on the lock.

Give 1.5 marks if says true because interrupts disabled in spinlock (why would you use spinlocks that disable interrupts on a uniprocessor!)

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]

False. Blocking is not free and involves a thread/process context switch, and scheduler logic. It depends on the size of the critical section and the expected concurrency/contention.
Larger critical section → blocking locks are better
small critical section → spinlocks are better

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]

False. Reader/writer locks can be implemented as either spinlocks or blocking locks. Waiting to acquire the lock can be implemented by either spinning or blocking.

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]

No, this code can not result in a deadlock, as at most one lock is held at any time.

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]

False. The value of the trapframe pointer (p->tf) is undefined at this point, i.e., it is not used and is overwritten on the next interrupt/exception/syscall.

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]

False. If a running process is marked runnable, two different CPUs can schedule the same process simultaneously, resulting in bad things to happen (e.g., overwrites on the kernel stack of the same process by multiple CPUs). [deduct 1 mark if do not mention an example]

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]

```
int test_and_set_function(int *addr) {
    fence
    asm(“test_and_set  addr, %eax”);
    return %eax
}
void acquire(struct lock *l) {
    while (test_and_set_function(&l->locked) == 1);
}

void release(struct lock *l) {
    fence;
    l->locked = 0;
}
```

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]

It is possible that
1. two producers are waiting on not_full
2. one consumer notifies not_full
3. first producer wakes up, acquires lock, produces element. the queue is now full
4. second producer wakes up, acquires lock, produces element to full queue! incorrect!

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

Replace the 'if' check with a 'while' check.

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]

No, this is not okay. Here, there is no problem of 'lost notify', but there is a problem of 'spurious notify' (i.e., notify getting called when it was not supposed to be called). Consider the following situation:
1. queue is empty
2. producer produces one element, releases lock, but does not call notify yet.
3. consumer consumes one element. queue is now empty.
4. consumer tries to consume another element, finds queue empty, waits on not_empty.
5. producer calls notify (from step 2)
6. consumer wakes up and tries to consume from an empty queue! incorrect!

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]

To ensure that the user-supplied memory address points to user-accessible memory. Else, a user may be able to trick the kernel into either accessing kernel memory or accessing unmapped region.

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

To ensure that there is no problem at the boundary cases.
1. If it only checked addr with proc->sz, it is possible for the user to supply addr at the boundary, such that addr+3 may not be mapped (or may belong to kernel)
2. If it only checked addr+4 with proc->sz, it is possible for the user to supply addr such that addr lies in kernel memory, but addr+4 lies in user memory. e.g., addr = 0xfffffffe on 32-bit machine. Here addr+4=0x2 (<proc->sz) but this access is invalid.

[1 mark for each point]