CS444/544 Operating Systems II Instructor: Yeongjin Jang

Name: OSU ID:

CS 444/544 Operating Systems II
Sample Quiz #2

You have 30 minutes to answer the questions in this quiz. In order to receive credit you must answer the question as precisely as possible.

If you find any ambiguity in the questions, be sure to write down any assumptions you make. You do not have to list all of the assumptions.

In case if we cannot read your answer nor interpret your answer, we can't give you credit.

Write your name and OSU ID on this cover sheet, and make sure you have all pages with the quiz sheet package: Quiz #2 should have total 8 pages.

NO Internet access, NO communication with other students, and NO consulting to textbook, slides, laptop, etc.

Section	I	II	Total
Score			
Max Score	20	80	100

II. Multiple choices (20 pts, 4 pts each)

II-1. In x86, which of the following instruction runs atomically?

a) cmpxchg	b) pusha	c) lea	d) **xchg**	e) mov

xchg in x86 is a hardware atomic instruction.

II-2. In x86, which of the following instruction runs atomic test and test-and-set?

a) cmpxchg	b) int $0x30	c) **lock cmpxchg**	d) lock	e) xchg

cmpxchg in x86 is not a hardware atomic instruction. However, when used with the lock prefix, the instruction will be an atomic test and test-and-set instruction.

II-3. Can we implement locks without using any of hardware atomic instructions?

a) YES	b) **NO**			

No, we must have a hardware atomic instruction to implement a lock.

II-4. Which register is being used for storing ‘compare’ value when running the cmpxchg instruction (in 32-bit x86) ?

a) CR3	b) **EAX**	c) EBX	d) ECX	e) ESP

EAX stores the ‘compare’ value.

II-5. Which of the following term is not relevant to data racing / thread synchronization?

a) LOCK	b) Intel TSX	c) LL/SC	d) **Page Table**	e) test-and-set

Lock will synchronize threads.
Intel TSX is a lock-less synchronization mechanism. LL/SC (load-link / store-conditional) is another lock-less synchronization mechanism.
Test-and-set is a name of one of mechanisms for building synchronization primitives.
Page table is for virtual memory, and it is not relevant to thread synchronization.

V. Locks (80 pts)

Beaver the Hacker implements a spin lock as follows.
Please answer the following questions regarding the implementation.

```
struct lock {
    uint32_t lock_variable; // 0: available, 1: locked
};

void spin_lock(struct lock *l) {
    while(xchg(&l->lock_variable, 1) != 0);
}

void spin_unlock(struct lock *l) {
    while(xchg(&l->lock_variable, 0) == 0);
}

uint32_t xchg(volatile uint32_t *addr, uint32_t newval) {
    uint32_t result;
    asm volatile("lock; xchgl %0, %1"
             : "+m" (addr), "=a" (result)
             : "1" (newval)
             : "cc");
    return result;
}
```

(See next pages)

1) (10 pts) The code what Beaver has written seems not optimal. So Duck has suggested the following implementation:

```
void spin_lock(struct lock *l) {
    while(l->lock_variable);
    l->lock_variable = 1;
}

void spin_unlock(struct lock *l) {
    l->lock_variable = 0;
}
```

Is this implementation correct (for a spin lock)? Please provide your answer (correct or incorrect) and provide a justification for your answer (i.e., If correct, why can this implementation synchronize threads? If incorrect, why threads are not able be synchronized with these functions?)

No. The implementation above does not check the value of the lock variable (testing 0, `while(l->lock_variable))` and set of the value of the lock variable (set 1) atomically. So there could be a chance that, after one thread passes the check (lock_variable == 0), and before the thread sets the variable as 1 (lock_variable = 1), other threads may pass the check, too. Allowing such a case will result in allowing two or more threads to pass the check. In such a case, two or more thread could be running in the critical section made by those lock/unlock functions.

2) (10 pts) spin_unlock() implemented by Beaver is slow. Can we use Duck's implementation (spin_unlock()) to replace Beaver's spin_unlock(), like the following?

```
void spin_lock(struct lock *l) {
    while(xchg(&l->lock_variable, 1) != 0);
}

void spin_unlock(struct lock *l) {
    l->lock_variable = 0;
}
```

Please provide your answer (yes or no) and provide a justification for your answer.

Yes. The lock implementation above uses an atomic test-and-set instruction, so it will allow only one thread to pass the bar and run in the critical section created by the lock function.
If we assume that the programmer places lock and unlock correctly to create a critical section, then only one thread (who acquired the lock) will call 'unlock', so setting the lock variable to zero without using a hardware atomic instruction will not cause any synchronization problem.

3) (20 pts) spin_lock() implemented by Beaver is also somewhat slow. Can you describe why Beaver's implementation using xchg() is slow and suggest a better implementation (please provide a pseudo-code for your implementation suggestion)?

The implementation will update the value of the lock variable regardless of the value that was stored in the lock variable. For instance, even for the case that lock variable were storing '1', which means there is another thread who acquired the lock, using of xchg() will update value from 1 to 1 atomically. This will generate lots of cache evictions in a multi-core machine, as running the instruction always updates the value, and such evictions will generate memory contention, which will make the lock slow.

As a solution, we may use test and test-and-set (cmpxchg in x86) to avoid unnecessary value update (from 1 to 1), and we may also apply exponential backoff to release lock contention among multiple threads.

A possible implementation would be:

```
1. Set the backoff value as 1.
2. test if cmpxchg succeed
3. if it is, acquire the lock and return.
4. otherwise, pause the cpu (run no-op) for 'backoff' times.
5. Double the backoff value, and go back to 2.
```

4) (20 pts) Beaver is now encountered a case of deadlock while running the following two threads:

```
struct lock l1, l2;

Thread 1:

spin_lock(&l1);
spin_lock(&l2);

Thread 2:
spin_lock(&l2);
spin_lock(&l1);
```

a) Can you describe a possible scenario of deadlock when Beaver runs these two threads?

Take the following case as an example.

Thread 1 has executed: `spin_lock(&l1);`
l1 is acquired by Thread 1

Thread 2 has executed: `spin_lock(&l2);`
l2 is acquired by Thread 2

and then,

Thread 1 would like to execute: `spin_lock(&l2);`
but it should wait until Thread 2 releases l2.

At the same time,
Thread 2 would like to execute: `spin_lock(&l1);`
but it should wait until Thread 1 releases l1.

Thread 1 waits for Thread 2, and Thread 2 waits for Thread 1; deadlock happens.

b) Can you resolve the problem? Please provide your fix as pseudocode (or description).

We may break the circular dependency among locks and threads by taking either of the following approach (full points will be given if you can answer one of the following).

Solution 1:

We may use a meta lock to acquire both locks atomically.

```
Thread 1:
spin_lock(&meta);
spin_lock(&l1);
spin_lock(&l2);

Thread 2:
spin_lock(&meta);
spin_lock(&l2);
spin_lock(&l1);

(must unlock meta after unlocking both l1 and l2)
```

Then, the meta lock will synchronize two threads, and the threads will not create the case of having circular dependencies.

Solution 2:

We may sort the order of acquiring locks.

```
Thread 1:
spin_lock(&l1);
spin_lock(&l2);

Thread 2:
spin_lock(&l1);
spin_lock(&l2);
```

Then, there will be no circular dependency between Thread 1 and Thread 2 (requirement: all thread must acquire the locks in the same order).