Sungkyunkwan University
School of Information and Communication Engineering

CSE3008: Operating Systems
Midterm Exam. (Fall 2009)

13:30 - 14:50, October 19, 2009. Instructor: Jin-Soo Kim

Student ID: _____

Name: _____

Q1 (20)		Q4 (20)	
Q2 (20)		Q5 (30)	
Q3 (10)		Q6 (20)	
		Total (120)	

1. The following shows some of instructions defined in the Intel IA-32 instruction set architecture. Among the listed instructions, which are *protected instructions* that cannot be executed in the user mode? Choose all that apply. (20 points, Do not take guesses. Each wrong answer will be penalized.)

HLT:	stops instruction execution and places the processor in a HALT state.
ADD:	adds the first operand and the second operand and stores the result in the destination operand.
CALL:	saves procedure linking information on the stack and branches to the procedure specified with the destination operand.
INT:	generates a call to the interrupt or exception handler specified with the destination operand.
POP:	loads the value from the top of the stack to the location specified with the destination operand and then increments the stack pointer.
OUT:	copies the value from the second operand to the I/O port specified with the destination operand.
JMP:	transfers program control to a different point in the instruction stream without recording return information.
CPUID:	provides processor identification information in registers.
CMP:	compares the first source operand with the second source operand and sets the status flags according to the results.
CLI:	clears the interrupt flag, which causes the processor to ignore maskable external interrupts.
MOVSW:	moves the word specified with the second operand to the location specified with the first operand.
SAL:	arithmetically shifts the bits in the destination operand to the left.

2. The following code fragment shows the Peterson's algorithm to solve the critical-section problem between two processes.
Suppose we exchange two lines, ***S0*** and ***S1***, in the original Peterson's algorithm, by moving the sentence ***S1*** above the sentence ***S0***. Does this modified algorithm still guarantee mutual exclusion of the critical section? If your answer is yes, prove that the algorithm still satisfies the mutual exclusion requirement. Otherwise, give a counter example, in which the algorithm fails. (20 points)

```
int turn;
int interested[2];
void acquire (int me) {
    int other = 1 - me;
S0:   interested[me] = 1;
S1:   turn = other;
    while (interested[other] && turn == other);
}
void release (int me) {
   interested[me] = 0;
}
```

3. All of the following operations disrupt the CPU's sequential flow of instruction execution.

A. Procedure call
B. Kernel-level thread switching
C. User-level thread switching
D. Unconditional branch (jump) instruction

Sort out the four operations in terms of increasing cost to perform. (10 points, no partial points)

[ ] < [ ] < [ ] < [ ]

4. Show whether each of the following functions, **sum0()** ~ **sum3()**, is multithread-safe or not. Explain why? (20 points)

(1)

```
int sum0 (int n, int a) {
    int i;
    for (i = 0; i < a; i++)
        n += a;
    return n;
}
```

(2)

```
int sum1 (int a) {
    int n = 0;
    int i;
    for (i = 0; i < a; i++)
        n += a;
    return n;
}
```

(3)

```
int n = 0;
int sum2 (int a) {
    int i;
    for (i = 0; i < a; i++)
        n += a;
    return n;
}
```

(4)

```
int sum3 (int a) {
    static int n = 0;
    int i;
    for (i = 0; i < a; i++)
        n += a;
    return n;
}
```

5. **[Unisex bathroom problem]** A single bathroom is used as a unisex bathroom. The following constraints must be maintained:

(1) If the bathroom is vacant, any man or woman can enter.
(2) Any number of men can be in the bathroom at the same time.
(3) Any number of women can be in the bathroom at the same time.
(4) There cannot be men and women in the bathroom at the same time.

Now we would like to solve this synchronization problem using semaphores. Suppose we have two kinds of threads, *men threads* and *women threads*. Men threads call `men_enter()` when they try to use the bathroom and call `men_leave()` when they exit. Similarly, women threads call `women_enter()` and `women_leave()` when they enter and exit, respectively. According to the above constraints, men threads should be blocked in `men_enter()` if the bathroom is already occupied by women, and vice versa. Complete the following functions using the semaphore interface provided by Pintos. (30 points)

```
/* global variables */









/* initialization function */
void init()
{









}

void men_enter()
{









}
```

```
void men_leave()
{









}
void women_enter()
{









}
void women_leave()
{









}
```

* Semaphore-related functions available in Pintos:

```
void sema_init (struct semaphore *sema, unsigned value);
void sema_down (struct semaphore *sema);                    // wait()
void sema_up (struct semaphore *sema);                      // signal()
```

6. The following code is executed on Pintos. Assume that the priority scheduling and the priority donation are already implemented in the Pintos kernel as suggested in the first project assignment. What is the output generated as a result of executing the function `foo()`? Explain why. (20 points)
NOTE: In Pintos, lower priority values mean lower priorities.

```
#include <stdio.h>
#include "tests/threads/tests.h"
#include "threads/init.h"
#include "threads/synch.h"
#include "threads/thread.h"

static thread_func a_thread_func;

static void a_thread_func (void *lock_){
  struct lock *lock = lock_;
  lock_acquire (lock);
  lock_release (lock);
}
```

```
void foo (void){
  struct lock a, b, c;
  thread_set_priority(31);

  lock_init (&a);
  lock_init (&b);
  lock_init (&c);

  lock_acquire (&a);
  lock_acquire (&b);
  lock_acquire (&c);

  thread_create ("a", 32, a_thread_func, &a);
  thread_create ("b", 33, a_thread_func, &b);
  thread_create ("c", 34, a_thread_func, &c);

  printf("%d ", thread_get_priority());
  lock_release (&b);
  printf("%d ", thread_get_priority());
  lock_release (&c);
  printf("%d ", thread_get_priority());
  lock_release (&a);
  printf("%d ", thread_get_priority());
}
```