ECE391 Exam 1, Fall 2009

Thursday 1 October

Name:

- **Be sure that your exam booklet has 12 pages.**
- **Write your name at the top of each page.**
- **The exam is meant to be taken apart.**
- **This is a closed book exam.**
- **You are allowed one 8.5×11" sheet of notes.**
- **We have provided a scratch sheet and some interfaces at the back.**
- **Absolutely no interaction between students is allowed.**
- **Show all of your work.**
- **Challenge questions are marked with *****
- **Don't panic, and good luck!**

Problem 1	20 points	_____
Problem 2	20 points	_____
Problem 3	20 points	_____
Problem 4	20 points	_____
Problem 5	20 points	_____
Total	100 points	_____

Problem 1 (20 points): Short Answers

Answer the following questions concisely. You should should not need to write more than a sentence or two; if you are writing more, it is probably wrong.

Part A (5 points): In class (and the course notes), we discussed Linux's use of a jump table structure as an abstraction for the interrupt controller. What advantage does this extra abstraction offer relative to requiring programmers to use the interface for the 8259A PIC directly?

Part B (5 points): Recall the user-level test harness provided for your use with MP1. Describe one advantage and one disadvantage of developing and using such a testing strategy when writing new kernel code, relative to doing all testing of the new code directly in the kernel.

Part C (5 points): Describe two scenarios in which use of a source control tool such as Subversion (`svn`) can make a programmer's life easier.

Problem 1, continued:

Part D (5 points): Several programs call the functions below in arbitrary order. These programs share a single copy of the global variables `lock` and `rnum`. Can the call to `printf` ever print a number less than 1000? If so, how?

```
spinlock_t lock = SPIN_LOCK_UNLOCKED;
int        rnum = 0;

void generate ()
{
   spin_lock (&lock);
   rnum = rand (); /* generates a random number from 0 to 2^31 - 1 */
   spin_unlock (&lock);
}

void check ()
{
   int cond = 0;

   spin_lock (&lock);
   if (1000 <= rnum)
       cond = 1;
   spin_unlock (&lock);

   if (1 == cond)
       printf ("the number is %d\n", rnum);
}
```

Problem 2 (20 points): x86 Assembly and C: Where's Waldo?

Where's Waldo (known outside of North America as *Where's Wally?*) is a series of children's books that consist of full-page illustrations of hundreds of people in a frenzy of activity. The intent is for the reader to find a character named Waldo who is hidden in the group. Waldo is always dressed in a red and white striped shirt, a red hat, and glasses.

Fed up with never finding Waldo as a child, your TA Chris has demanded that you write program that finds Waldo for him. To help you along, Chris has written code that takes a *Where's Waldo* image and creates a singly-linked list in which each node holds information about a separate person. The structure of a linked list node is:

```
typedef struct person_t person_t;
struct person_t {
    uint8_t   hat;     // hat color (e.g., 'r' for red)
    uint8_t   glasses; // does person have glasses?  ('y' for yes)
    uint16_t  posn;    // position relative to image
    person_t* next;    // next person in linked list
};
```

In this problem, you will write a function in x86 assembly to search the linked list for a particular "Waldo." We have broken this problem up into parts to simplify it slightly, but we expect that your choices (for example: save, use, and restore of callee-saved registers) will be consistent across the problem.

The C function prototype is:

```
/* where_is_waldo
 *   Description: searches through the linked list of people to find a
 *       match for a specific waldo (by hat color and use of glasses)
 *   Inputs:
 *       waldo - pointer to person_t struct with valid hat and glasses
 *               fields (the values for which the function searches)
 *       list - head of linked list to search for waldo
 *   Returns: the (zero-extended) posn field of the first person in the list
 *       with hat and glasses fields that match the specified waldo;
 *       if no such match is found, returns -1
 */
int32_t where_is_waldo (person_t* waldo, person_t* list);
```

Be sure to obey the standard C calling convention with your function.

Part A (3 points): Write x86 assembly to set up the stack frame for the function as would a compiler such as `gcc`.

```
where_is_waldo:  # write your code below
```

Problem 2, continued:

Part B (4 points): Write x86 assembly to move the pointer to the head of the linked list into EDX and the pointer to the `waldo` being sought into ECX.

```
# code continued from Part A
```

Part C (10 points): Write x86 assembly to search for Waldo in the linked list. Be sure to set the return value properly. You may use the `DONE` label below to leave this section of the code.

```
# code continued from Part B
```

```
DONE:
```

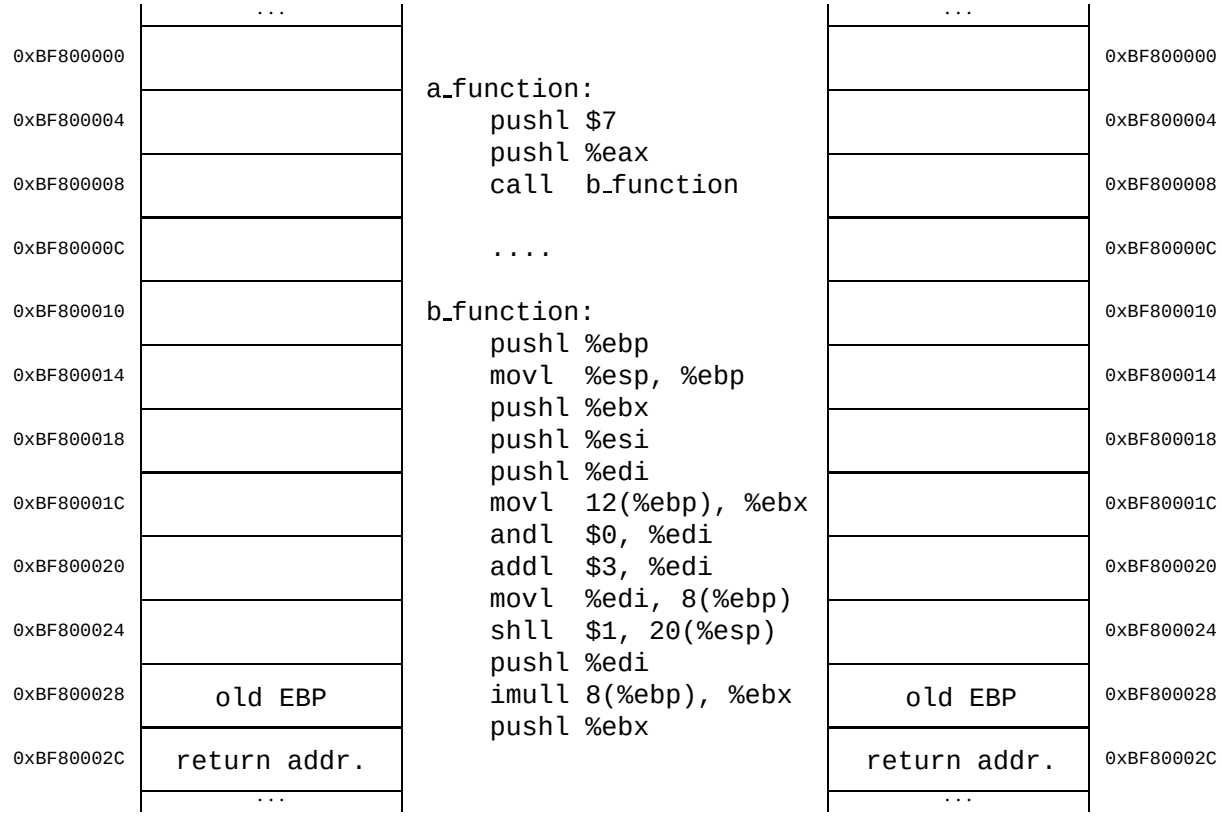
Part D (3 points): Write x86 assembly to tear down the stack frame and return from the function.

```
# code continued from Part C
```

Problem 3 (20 points): Stack Frames and Devices

Part A (15 points): The figures on the left and right below show the state of the stack before executing the code in the center. Code execution starts from the label `a_function` and continues until the processor reaches the bottom of the code shown. **The functions shown use the standard C calling convention.**

Fill in the state of the stack after execution of this code. Registers ESP and EBP both initially hold the value 0xBF800028. You may express stack values as expressions of the initial register values, labels from the code, and numbers; **however, you must use numbers rather than symbolic values when possible** for full credit. Also indicate the final addresses to which EBP and ESP point. You may use one stack for scratch and one for the solution you want graded. **Circle the stack figure that you want graded.**



Part B (5 points): A device command has been stored in the low 8 bits of EAX, and a corresponding data value has been stored in the low 8 bits of ECX. Using as few x86 assembly instructions as possible, write code that writes a one byte command to PORT 0x60 and a one byte value to PORT 0x61.

```
# command is stored in lower byte of EAX (zero extended)
# data value is stored in lower byte of ECX (zero extended)
# (write your code below this line)
```

Problem 4 (20 points): Synchronization

As part of your first job, you are charged with implementing a more fair lock construct. Specifically, you must implement a bakery lock (also called a ticket lock) based on a Linux semaphore.

A bakery lock is analogous to the physical mechanism used at many busy bakeries. When a customer enters the bakery, they take the next paper ticket from a dispenser. Each ticket is marked with a unique number, and consecutive tickets have consecutive numbers (1, 2, 3, *etc.*). A display indicates the ticket number currently being served. After taking a ticket, a customer watches the display for their number. When the display shows their number, they can order (that is, they own the lock). After a customer orders (that is, when releasing the lock), the number displayed is incremented.

Using the following data structure and the Linux semaphore API (provided on the last page of the exam), implement the `bakery_init`, `bakery_lock`, and `bakery_unlock` functions and answer the subsequent questions.

```
typedef struct bakery_lock_t bakery_lock_t;
struct bakery_lock_t {
    struct semaphore      sem;
    unsigned int          dispenser;
    volatile unsigned int display;
};
```

Part A (3 points): Implement the `bakery_init` function below such that the display is initially set to 1. Be sure that the other fields are set properly to work with your answer to **Part B**.

```
void bakery_init (bakery_lock_t* b)
{




}
```

Part B (6 points): Implement the `bakery_lock` function below. Return only after the bakery lock is owned by the caller.

```
void bakery_lock (bakery_lock_t* b)
{




}
```

Problem 4, continued:

Part C (3 points): Implement the `bakery_unlock` function below.

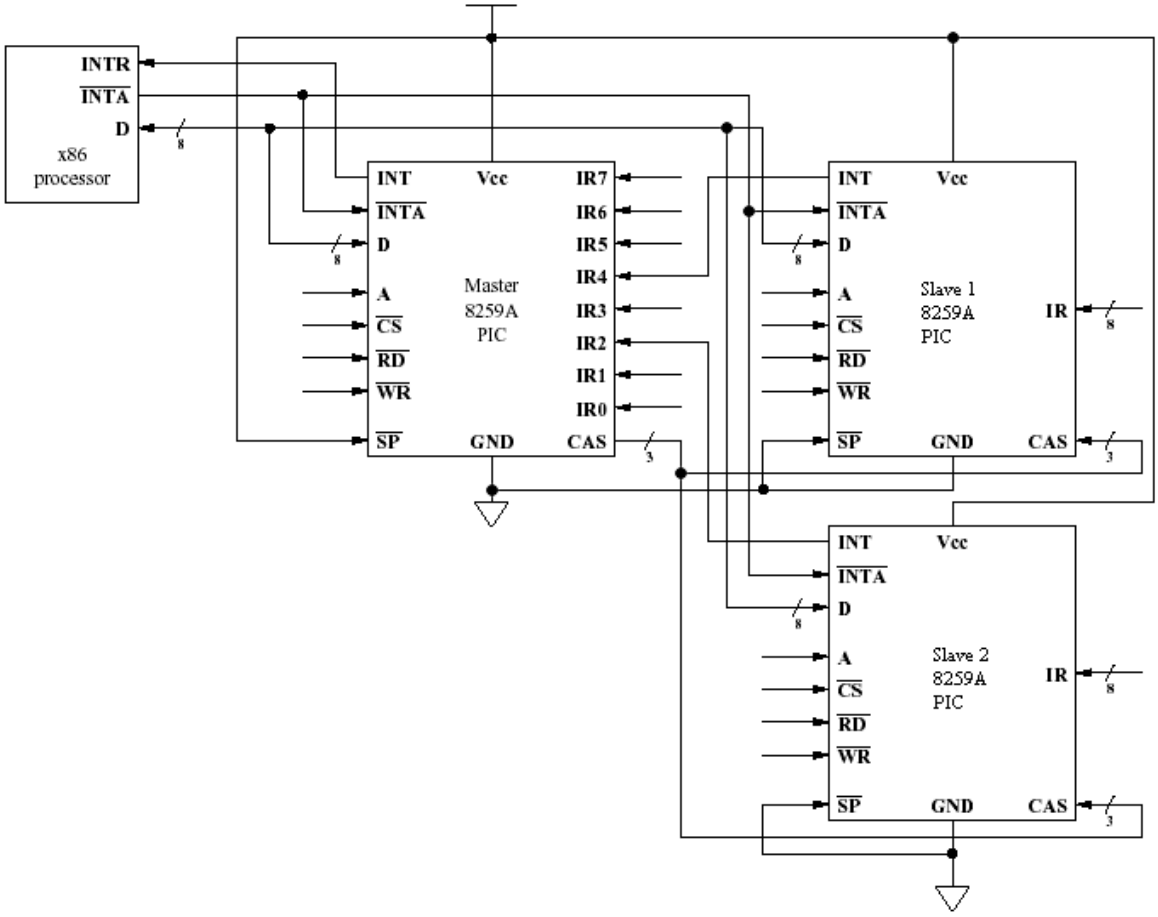
```
void bakery_unlock (bakery_lock_t* b)
{

}
```

Part D (4 points): After you implement your lock, a friend at the company confides in you that they are nervous about the correctness of your implementation and would like to add a second synchronization variable to the bakery lock. Specifically, the friend suggests adding a reader-writer semaphore to the `bakery_lock_t` structure, acquiring a read lock in `bakery_lock` while waiting for the `display` field to show the caller's number, and acquiring a write lock in `bakery_unlock` before changing the display. Explain why you refuse to adopt your friend's suggestion.

Part ***E (4 points): Based on the design of the bakery lock in this problem, is it safe for a program that holds a bakery lock to acquire a Linux semaphore? Explain briefly why acquiring a semaphore (as opposed to a spin lock) might be a concern and why that particular problem will or will not show up with the bakery lock.

Problem 5 (20 points): The 8259A PIC

A new x86-based computer system is being developed for the ECE391 learning platform. This system requires 20 interrupts. To accommodate this need, three 8259A PICS have been connected as in the diagram below. (Note: There are no surprise elements in the diagram; we have simply created a standard cascade of three 8259A PICs.)



Given the in-service mask for each PIC, describe the sequence of events and interactions that occur between the slave PICs, master PIC, and x86 processor when a device raises the IR4 line on the slave 1 PIC to signal an interrupt. The in-service masks are of the form ($IR_7IR_6IR_5IR_4IR_3IR_2IR_1IR_0$), with the IR7 line as the most-significant bit and IR0 line the least-significant bit. As is usually the case, IR0 has the highest priority. Assume that any interrupts that are currently in-service do not finish in the time covered by your answer.

The following example illustrates the expected form of the answer.

Example problem: Master in-service mask: 00010100
Slave 1 in-service mask: 00001000
Slave 2 in-service mask: 01000000

Example answer: • *IR4 raised on slave 1*
• *IR3 already in service on slave 1 → suppressed for now*
(had a higher-priority IR not been in service already, slave 1 would have raised its INT output, and the answer would continue with the next chip)

Problem 5, continued:

Part A (6 points):
Master in-service mask: 00100100
Slave 1 in-service mask: 01000000
Slave 2 in-service mask: 00001000

Part B (8 points):
Master in-service mask: 00100000
Slave 1 in-service mask: 01000000
Slave 2 in-service mask: 00000000

Part C (6 points): Suppose that we needed even more devices. Using a total of **five 8259A PICs**, what is the maximum number of devices that we could support?

(scratch paper)

Linux synchronization functions (comments give return values)

```
void spin_lock_init   (spinlock_t* lock);
void spin_lock        (spinlock_t* lock);
void spin_unlock      (spinlock_t* lock);
int  spin_is_locked   (spinlock_t* lock); /* 1 if held, 0 if not */
int  spin_try_lock    (spinlock_t* lock); /* 1 on success, 0 on failure */
void spin_unlock_wait (spinlock_t* lock);

void sema_init         (struct semaphore* sem, int val);
void init_MUTEX        (struct semaphore* sem);
void init_MUTEX_LOCKED (struct semaphore* sem);
void down              (struct semaphore* sem);
void up                (struct semaphore* sem);
int  down_trylock      (struct semaphore* sem); /* 0 on success, 1 on failure */

void rwlock_init   (rwlock_t* rw);
void read_lock     (rwlock_t* rw);
void write_lock    (rwlock_t* rw);
void read_unlock   (rwlock_t* rw);
void write_unlock  (rwlock_t* rw);
void write_trylock (rwlock_t* rw); /* 1 on success, 0 on failure */

void init_rwsem (struct rw_semaphore* sem);
void down_read  (struct rw_semaphore* sem);
void down_write (struct rw_semaphore* sem);
void up_read    (struct rw_semaphore* sem);
void up_write   (struct rw_semaphore* sem);
```