ECE 391 Exam 1, Fall 2010

Tuesday 28 September

Name:

- **Be sure that your exam booklet has 11 pages.**
- **Write your name at the top of each page.**
- **This is a closed book exam.**
- **You are allowed one 8.5×11" sheet of notes.**
- **Absolutely no interaction between students is allowed.**
- **Show all of your work.**
- **Don't panic, and good luck!**

Problem 1	25 points	____________
Problem 2	25 points	____________
Problem 3	25 points	____________
Problem 4	25 points	____________
Total	100 points	____________

Problem 1 (25 points): Short Answers

Please answer concisely. If you find yourself writing more than a sentence or two, your answer is probably wrong.

Part A (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 B (5 points): When an interrupt occurs in a generic cascade PIC configuration, both the master and slave don't know the correct vector number to send to the CPU. Explain why this is from each of the individual PIC's viewpoints and how this issue is addressed in the 8259A PIC design?

Problem 1, continued:

Part C (5 points): After a call returns from a C function, why should you remove its arguments off the stack AND not reuse those values?

Part D (5 points): Your friend suggests that mapping the 8259A interrupts into the 0x00 to 0x0F range of the Interrupt Descriptor Table can save time on translations between vector number and IRQ. Explain why such a mapping will not work as intended.

Problem 1, continued:

Part E (5 points): Several programs call the functions below in an 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? Explain.

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

void generate()
{
     spin_lock (&lock);
     rnum = rand();    // Generate a random number from 0 to (2^32 - 1)
     spin_unlock (&lock);
}

void check()
{
     int cond = 0;

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

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

Problem 2 (25 points): Calling Conventions and the Stack

Part A (10 points):

To improve the search speed of the `mp1_blink_struct` list, the linked list is to be converted into an array. You have been provided a skeleton `convert_to_array` function that takes in the current number of elements in the singly-linked list (`list_length`), allocates enough space for an array that will contain all of the current list nodes, fills in the array, and returns a pointer to the array.

```
void add_to_array(mp1_blink_struct *array, mp1_blink_struct *current,
                         int filled_array_elements);

void* mp1_malloc(int size);

mp1_blink_struct* convert_to_array(int list_length);
```

Complete `convert_to_array` by filling in the call to `add_to_array` following the appropriate calling convention. The code is on the following page. If you find yourself writing more than 15 lines of code, you have probably made a mistake.

`convert_to_array` will iterate through every element starting at `mp1_list_head` and call the `add_to_array` function. `add_to_array` takes:

- `array` - the pointer to the allocated array of `mp1_blink_structs`
- `current` - the `mp1_blink_struct` to add to `array`
- `filled_array_elements` - the current number of elements copied into `array`

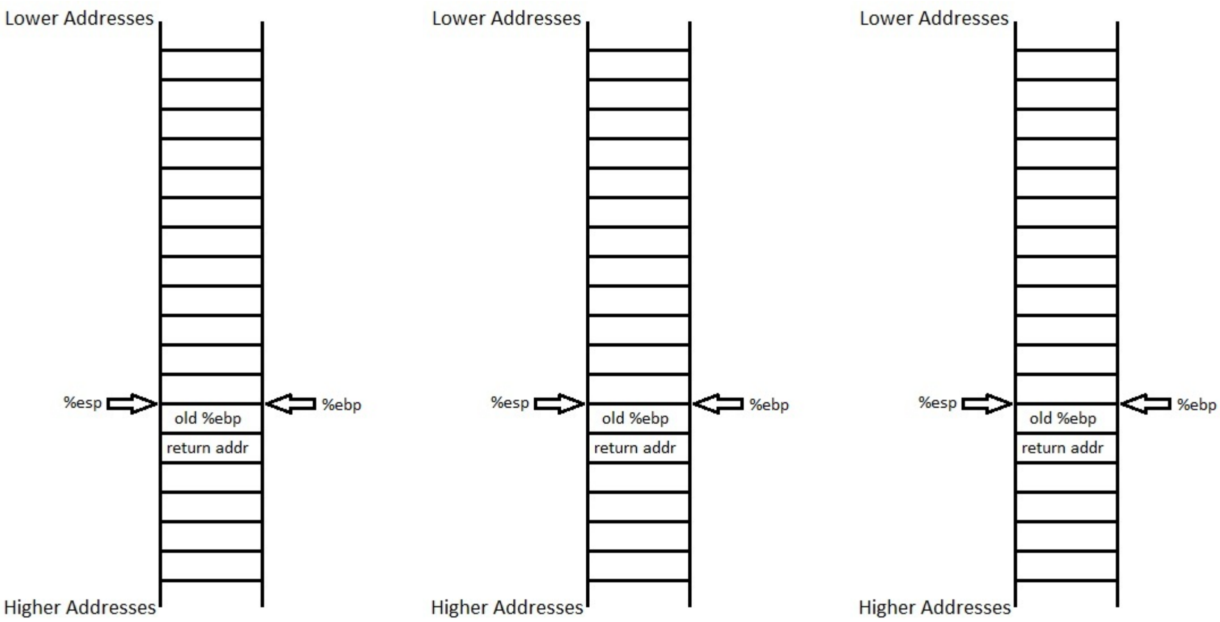
Problem 2, continued:

```
.long mp1_list_head

convert_to_array:
     pushl %ebp
     movl %esp,%ebp
     pushl %ebx
     pushl %esi
     pushl %edi
     movl 8(%ebp), %eax
     movl mp1_list_head, %edx
     imull $STRUCT_SIZE, %eax
     pushl %eax
     call mp1_malloc
     addl $4, %esp
     cmpl $0, %eax
     je MALLOC_FAIL
     movl $0, %ecx
CHECK_NEXT:
     cmpl $0, %edx
     je DONE_INSERTING
           /* Insert call to add_to_array here */












     incl %ecx
     movl %edi, %edx
     jmp CHECK_NEXT
DONE_INSERTING:
     popl %edi
     popl %esi
     popl %ebx
     leave
     ret
MALLOC_FAIL:
     pushl %eax
     call mp1_free
     addl $4, %esp
     movl $0, %eax
     jmp DONE_INSERTING
```

Problem 2, continued:

Part B (15 points):

The figures below are the state of the stack before the execution of the given code. Please fill in the state of the stack after execution and also indicate where %ebp and %esp are pointing to. Treat registers whose values you do not know as variables, otherwise fill in the actual number. We give you multiple figures for scratch work purposes but one must contain your final answer. **Circle the stack you want us to grade!**
If you do not circle a stack, we will give you a zero on the problem.




```
     pushl $20
     pushl %ebx
     pushl %ecx
     call foo
...
foo:
     pushl %ebp
     movl %esp, %ebp
     pushl %ebx
     pushl %esi
     pushl %edi
     addl $-8, %esp
     movl 8(%ebp), %edi
     movl 12(%ebp), %esi
     xorl %ebx, %ebx
     movl %ebx, 4(%esp)
     addl $4, %esi
     imull %esi, %esi
     addl $-4, %esp
     movl %esi, (%esp)
```

Problem 3 (25 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 the Linux semaphore.

A bakery lock is analogous to the physical mechanism used at many busy bakeries. When a customer enters the bakery, he/she takes the next paper ticket from the dispenser. Each ticket is marked with a unique number, and consecutive tickets have 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 his/her number. When the display shows his/her number, the customer can order (that is, he/she owns the lock). After the customer orders (that is, when realizing 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.

Note: Multiple people should be able to take a ticket while other people are waiting to be served

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

Part A (4 points): Implement `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 (7 points): Implement `bakery_lock` function below. Return only after the bakery lock is owned by the caller.

```
void bakery_lock (bakery_lock_t* b)
{




}
```

Problem 3, continued:

Part C (4 points): Implement `bakery_unlock` function below.

```
void bakery_unlock (bakery_lock_t* b)
{



}
```

Part D (5 points): After you implement your lock, a friend in 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 (5 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 bakery lock.

Problem 4 (25 points): x86 Assembly

Your math assignment requires you to perform numerous dot products on sets of two vectors. Being a computer engineer, you decide to flex your knowledge of x86 assembly to write a dot product calculator and thus make your math assignment a joke. Write the function `dot_product` that iterates through two equal length singly-linked lists that represent the two vectors, and returns their dot product. The heads to the linked lists are `vect1_head` and `vect2_head`, which are global variables. The linked list structure is defined below:

```
struct vect_node_t {
     vect_node_t* next;
     int value;
};
```

How to perform the dot product: given two vectors **A** and **B** where **A**=$\{1,2,3\}$ and **B**=$\{4,5,6\}$, their dot product is $(1 * 4) + (2 * 5) + (3 * 6) = 32$. If you have any questions on the semantics of the dot product, please ask a TA.

Write your x86 assembly code on the next page. You may use the space provided on the current page to write the code in C, but your C code will not be graded. **Assume equal length vectors and the result will not overflow a 32 bit integer.**

Problem 4, continued:

```
.global vect1_head
.global vect2_head

dot_product:
```

(scratch paper)

You may tear off this page to use as a reference

Synchronization API reference

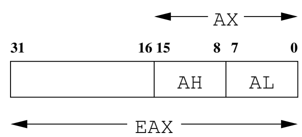
`spinlock_t lock;`	Declare an uninitialized spinlock
`spinlock_t lock1 = SPIN_LOCK_UNLOCKED;` `spinlock_t lock2 = SPIN_LOCK_LOCKED;`	Declare a spinlock and initialize it
`void spin_lock_init(spinlock_t* lock);`	Initialize a dynamically-allocated spin lock (set to unlocked)
`void spin_lock(spinlock_t *lock);`	Obtain a spin lock; waits until available
`void spin_unlock(spinlock_t *lock);`	Release a spin lock
`void spin_lock_irqsave(spinlock_t *lock, unsigned long& flags);`	Save processor status in `flags`, mask interrupts and obtain spin lock (note: flags passed by name (macro))
`void spin_lock_irqrestore(spinlock_t *lock, unsigned long flags);`	Release a spin lock, then set processar status to `flags`

`struct semaphore sem;`	Declare an uninitialized semaphore
`static DECLARE_SEMAPHORE_GENERIC (sem, val);`	Allocate statically and initialize to `val`
`DECLARE_MUTEX (mutex);`	Allocate on stack and initialize to one
`DECLARE_MUTEX_LOCKED (mutex);`	Allocate on stack and initialize to zero
`void sema_init(struct semaphore *sem, int val);`	Initialize a dynamically allocated semaphore to `val`
`void init_MUTEX(struct semaphore *sem);`	Initialize a dynamically allocated semaphore to one.
`void init_MUTEX_LOCKED(struct semaphore *sem);`	Initialize a dynamically allocated semaphore to zero.
`void down(struct semaphore *sem);`	Wait until semaphore is available and decrement (P)
`vod up(struct semaphore *sem);`	Increment the semaphore

You may tear off this page to use as a reference

x86 reference

32–bit	16–bit	8–bit high	8–bit low
`EAX`	`AX`	`AH`	`AL`
`EBX`	`BX`	`BH`	`BL`
`ECX`	`CX`	`CH`	`CL`
`EDX`	`DX`	`DH`	`DL`
`ESI`	`SI`		
`EDI`	`DI`		
`EBP`	`BP`		
`ESP`	`SP`		



`jb`	below	CF is set
`jbe`	below or equal	CF or ZF is set
`je`	equal	ZF is set
`jl`	less	SF ≠ OF
`jle`	less or equal	(SF ≠ OF) or ZF is set
`jo`	overflow	OF is set
`jp`	parity	PF is set (even parity)
`js`	sign	SF is set (negative)

```
movb  (%ebp),%al            # AL ← M[EBP]
movb  -4(%esp),%al          # AL ← M[ESP - 4]
movb  (%ebx,%edx),%al       # AL ← M[EBX + EDX]
movb  13(%ecx,%ebp),%al     # AL ← M[ECX + EBP + 13]
movb  (,%ecx,4),%al         # AL ← M[ECX * 4]
movb  -6(,%edx,2),%al       # AL ← M[EDX * 2 - 6]
movb  (%esi,%eax,2),%al     # AL ← M[ESI + EAX * 2]
movb  24(%eax,%esi,8),%al   # AL ← M[EAX + ESI * 8 + 24]
movb  100,%al               # AL ← M[100]
movb  label,%al             # AL ← M[label]
movb  label+10,%al          # AL ← M[label+10]
movb  10(label),%al         # NOT LEGAL!

movb  label(%eax),%al       # AL ← M[EAX + label]
movb  7*6+label(%edx),%al   # AL ← M[EDX + label + 42]

movw  $label,%eax           # EAX ← label
movw  $label+10,%eax        # EAX ← label+10
movw  $label(%eax),%eax     # NOT LEGAL!

call  printf                # (push EIP), EIP ← printf
call  *%eax                 # (push EIP), EIP ← EAX
call  *(%eax)               # (push EIP), EIP ← M[EAX]
call  *fptr                 # (push EIP), EIP ← M[fptr]
call  *10(%eax,%edx,2)      # (push EIP), EIP ←
                            #         M[EAX + EDX*2 + 10]
```

Conditional branch sense is inverted by inserting an "N" after initial "J," *e.g.*, JNB. Preferred forms in table below are those used by debugger in disassembly. Table use: after a comparison such as

```
cmp %ebx,%esi   # set flags based on (ESI - EBX)
```

choose the operator to place between ESI and EBX, based on the data type. For example, if ESI and EBX hold unsigned values, and the branch should be taken if ESI $\leq$ EBX, use either JBE or JNA. For branches other than JE/JNE based on instructions other than CMP, check the branch conditions above instead.

	`jnz`	`jnae`	`jna`	`jz`	`jnb`	`jnbe`	unsigned comparisons
preferred form	`jne`	`jb`	`jbe`	`je`	`jae`	`ja`	
	$\neq$	$<$	$\leq$	$=$	$\geq$	$>$	
preferred form	`jne`	`jl`	`jle`	`je`	`jge`	`jg`	signed comparisons
	`jnz`	`jnge`	`jng`	`jz`	`jnl`	`jnle`	