ECE 391 Exam 1, Spring 2012

Thursday February 23rd

Name:

- **Be sure that your exam booklet has 16 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	23 points	________
Problem 2	12 points	________
Problem 3	20 points	________
Problem 4	15 points	________
Problem 5	20 points	________
Problem 6	10 points	________
Total	100 points	________

Problem 1 (23 points): Short Answers

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

Part A (4 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): You have a device attached to IRQ0 on the PIC. Everytime that device generates an interrupt, the `divide_by_zero` exception handler is invoked instead of your device handler. What have you set up incorrectly and how do you fix it?

Problem 1, continued:

Part C (4 points): Why is it necessary to save the state of the caller saved registers immediately after receiving an interrupt?

Part D (5 points): Why does Linux make use of tasklets (i.e., software interrupts) instead of executing all interrupt-related activity in the (hardware) interrupt handler?

Problem 1, continued:

Part E (5 points): Explain why it is not enough to do CLI/STI when synchronizing accesses to resources shared by your code and interrupt handlers in a multiprocessor setting.

Problem 2 (12 points): PIC Design Rationale and Issues

You may find it helpful to consult the 8259A diagram on the back of the exam for this problem.

Part A (4 points): Explain the role of the CAS (cascade) bus in Intel's 8259A PIC. Specifically, why it is necessary, and how is it used?

Part B (4 points): Three 8259A PICs are cascaded together, with slave X occupying IR0 on the master PIC and slave Y occupying IR4. Assuming that the standard priority scheme is used on each PIC (IR0 is high, IR7 is low), show the overall priority scheme for the lines on the master M (call them M0 through M7) and slaves X (X0 through X7) and Y (Y0 through Y7).

Part C (4 points): Draw the glue logic necessary to connect the A (address, 1 bit) and $\overline{\text{CS}}$ (chip select, 1 bit, active low) ports of an 8259A PIC to the 16-bit address bus of a processor such that the PIC occupies ports 0x100 and 0x101. Your diagram should not be gate-level, but be sure that any component meanings are clear.

Problem 3 (20 points): Calling Convention

You have been asked to write a recursive function to emulate an iteration of a sum-reduction collective operation (multiprocessor function to get the sum of an extremely large set of data). The C code implementation is already written for you below (and at the end of the exam). The x86 code is generated from this C code. You are to fill in the missing x86 GNU Assembly instructions related to the C calling convention you learned in class.

```
typedef struct elem{
     int value;
     elem_t* next
} elem_t;

// Returns the length of the new "sum linked-list" made during traversal
int recursive_reduce_iter(int blocksize, elem_t* start)
{
     int new_list_len, sum;
     elem_t* end = start;
     int temp = blocksize;
     if (start == NULL)       // Base Case
         return 0;
     while(temp != 0){      // Compute next block's address
         end = end->next;
         temp--;
     }
     end = end->next;

     new_list_len = recursive_reduce_iter(blocksize, end);   // Recursive Call
     sum = sum_nodes(start, end);    // sum_nodes call

     start->value = sum;   // Update node's value and next ptr
     start->next = end;
     return new_list_len+1;     // Return the length of the new list
}

int sum_nodes(elem_t* start_node, elem_t* end_node);  // Helper func, returns sum
```


The x86 Assembly is below and continues to the next page. Remember to **follow the C calling convention** and **only add code where the x86 comments say to add code.**

```
recursive_reduce_iter:
   pushl %ebp
   movl %esp, %ebp
                                             # ADD YOUR CODE HERE TO...
                                             # blocksize => ecx
                                             # start => eax
                                             # new_list_len => local var on stack
   movl %eax, %edx
   cmpl $0, %eax
   je base_case
block_loop:
   cmpl $0, %ecx
   je next_block
   movl NEXT(%edx), %edx
   decl %ecx
   jmp block_loop
```

Problem 3, continued:

```
next_block:
   movl NEXT(%edx), %edx
```



```
                                                      # ADD YOUR CODE HERE TO...
                                                      # Do recursive call
                                                      # followed by sum_nodes call

   movl %ecx, VALUE(%eax)
   movl %edx, NEXT(%eax)
                                                      # ADD YOUR CODE HERE TO....
                                               # Set return value to new_list_len+1
done:
   leave
   ret
base_case:
   movl $0, %eax
   jmp done
```

Problem 4 (15 points): Synchronization

There is another synchronization method other than the ones taught in class called a barrier. Barriers make sure that all threads stop at a certain point before continuing. An example would be a parallel read/sort function. Several threads would read some data in parallel and stop at a barrier before moving on to do the actual sort.

```
extern static int NUM_THREADS;    // assume the number of threads does not change

// you may add additional members to this struct,
// but NO NEW synchronization primatives
typedef struct {
    spinlock_t lock;



} barrier_t;
```

Part A (5 points): Implement the initialization function below. Remember to initialize any members you added to the `barrier_t` struct.


```
void barrier_init(barrier_t *b)
{

}
```

Part B (10 points): Implement a `barrier_wait` function below.

```
void barrier_wait(barrier_t *b)
{

}
```

Problem 5 (20 points): x86 Assembly and C

You've decided to implement a new function to supplement those you created in MP1. The purpose of the function is to make the fish appear dead, and you intend on calling it at the end of the user level test harness. To achieve this goal, you will replace the eye of the fish with an 'x' during the "off frame". In Part A, you will implement the function as a new ioctl using x86 assembly.

```
/\/\/\/\/\/\/\/\/\/\/\/\
         O
           o    o
       o
             o
        o     O
    _    \
 |\/.\   | \/  /  /
 |=  _>   \|   \ /
 |/\_/    |/   |/
----------M----M--------
```

```
\/\/\/\/\/\/\/\/\/\/\/\/
           o    o
        o
             o
        o     o
    _   /
 |\/x\  \ \/  \  /
 |=  _>  \ \   \|
 |/\_>    |/   |/
----------M----M--------
```


Traverse the `mp1_list_head` list, looking for an element whose `ON_CHAR` field matches the ascii value for '.'(0x2E). If there is such an element, replace the `OFF_CHAR` field with 'x'(0x78). You are guaranteed that there is at most one '.' in the list and that this always corresponds to the "eye" you should replace. You are **NOT** guaranteed that the '.' is at a fixed location, so you must search the list based on the `ON_CHAR` field rather than the `LOCATION` field. The argument is only present for the sake of consistency, and contains only garbage. Return 0 on success, and -1 on failure. Insert the code to implement `mp1_ioctl_kill` in mp1.S shown on the next page.

Use x86 GNU Assembly for this part!


```
.data
# Useful offset constants for accessing members of a mp1_blink_struct structure
LOCATION   = 0
ON_CHAR    = 2
OFF_CHAR   = 3
ON_LENGTH  = 4
OFF_LENGTH = 6
COUNTDOWN  = 8
STATUS     = 10
NEXT       = 12
```

Problem 5, continued:

```
.global mp1_ioctl_kill

# Pointer to head of list (initialized to NULL)
mp1_list_head: .long   0

# int mp1_ioctl_kill(unsigned long arg)
#  follows C calling convention
#  %ecx MUST mantain list pointer

mp1_ioctl_kill:
```



```
LOOP:
```



```
FOUND:



NOT_FOUND:
```

Problem 6 (10 points): Debugging(***)

There is one bug in the following code where unexpected behavior may happen. Explain **what conditions must occur for the bug to happen, what occurs as a result of this bug, and how you would fix it.**

Assume the operations done on `arg1` and `arg2` do not overflow the int return value from that operation. The (***) means this is a challenge problem. Proportion your time appropriately.

```
# int dispatcher(unsigned int arg1, unsigned int arg2, unsigned int operation)
#    Dispatcher function that uses a function pointer jump table
#       to execute the appropriate operation function.
#
#    Inputs: operation - index into function pointer jump table
#            arg1, arg2 - argumets that the functions operate on
#
#    Outputs: Returns -1 if operation is out of array bounds, otherwise
#             the function that is jumped to sets the return value
#
#    Note: The function calling dispatcher as well as each of the functions
#          in the jump table follow the C calling convention. Recall that the
#          dispatcher is a special function (MP1's mpi_ioctl that you wrote
#          was a dispatcher)

dispatcher:
   movl 12(%esp), %ecx
   cmpl $0, %ecx
   jl bad_op
   cmpl $2, %ecx
   jg bad_op
   jmp *jumptable(,%ecx,4)
bad_op:
   movl $-1, %eax
   leave
   ret


# int op_<function> (unsigned int arg1, unsigned int arg2)
#    Note: Assume that the op_<function> does not overflow
#          the return value

jumptable:
   .long op_add, op_mult, op_abs
```

(scratch paper)

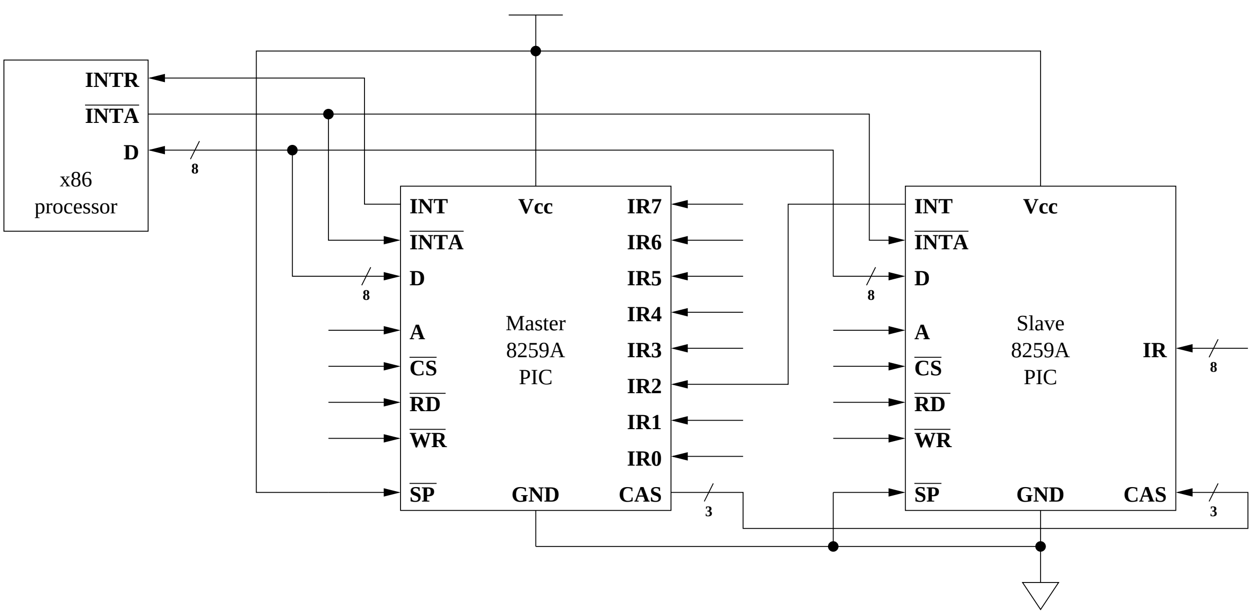
You may tear off this page to use as a reference

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

a PICture of the 8259A



You may tear off this page to use as a reference

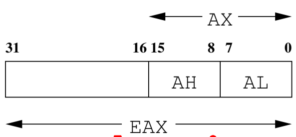
Recursive_reduce_iter() C code

```
typedef struct elem{
     int value;
     elem_t* next
} elem_t;

// Returns the length of the new "sum linked-list" made during traversal
int recursive_reduce_iter(int blocksize, elem_t* start)
{
     int new_list_len, sum;
     elem_t* end = start;
     int temp = blocksize;
     if (start == NULL)      // Base Case
          return 0;
     while(temp != 0){     // Compute next block's address
         end = end->next;
         temp--;
     }
     end = end->next;

     new_list_len = recursive_reduce_iter(blocksize, end);  // Recursive Call
     sum = sum_nodes(start, end);      // sum_nodes call

     start->value = sum;   // Update node's value and next ptr
     start->next = end;
     return new_list_len+1;     // Return the length of the new list
}

int sum_nodes(elem_t* start_node, elem_t* end_node);  // Helper func, returns sum
```

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	