ECE 391 Exam 1, Spring 2011

Thursday, Feb 24, 2011

Name: ______________________________

NetID: ______________________________

TA's name: ______________________________

- **Be sure that your exam booklet has 14 pages.**
- **Write your net ID 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!**

Page	Points	Score
3	9	
4	4	
5	5	
6	10	
7	5	
8	8	
10	12	
11	30	
Total:	83	

Question 1: Short Answer (18 points)

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

(4) (a) 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.

(5) (b) On an x86 processor, what are the two methods of communicating with an external device? What are the differences between the two? For each method, give an example of a piece of hardware that uses it.

(2) (c) Why does the C calling convention push arguments from right to left?

(2) (d) In the system call calling convention, which registers are caller-saved? Which are callee-saved?

(3) (e) `spin_lock_irqsave` will acquire the spin lock and call `CLI` to clear `IF`. Which happens first? Explain why.

(2) (f) When is it *necessary* to use `spin_lock_irqsave` rather than `spinlock_irq` or `spinlock`?

Question 2: Locks and Synchronization (15 points)

(10) (a) Implement `spin_lock` and `spin_unlock` in assembly using the global variable `lock`. Use the bit test and set atomic operation `bts`.

`bts offset, base`

`bts` selects the bit in `base` at the bit-position specified by the `offset` operand, stores the value of the bit in the carry flag, and sets the selected bit to 1.

```
lock:
    .byte    0

spin_lock:
```

```
spin_unlock:
```

(5) (b) Suppose the following variables are declared globally:

```
int x = 0;
int y = 0;
spinlock_t* lock;
```

Then, the following two threads are run in parallel:

```
void thread1(void)
{
        y++;
        spin_lock(lock);
        x++;
        spin_unlock(lock):
}
```

```
void thread2(void)
{
        spin_lock(lock);
        x++;
        spin_unlock(lock);
        y++;
}
```

What will be the values of x and y after these threads execute? Explain.

Question 3: Calling Conventions and the Stack (20 points)

(8) (a) To improve the search speed of the `mp1_blink_struct` list, Rich has decided to try a data structure he saw on Reddit; Judy arrays. A Judy array is an associative array data structure intended to be fast and have low memory usage. Unlike a normal array, a Judy array can be sparse; that is, it can have indices which are unassigned and unallocated. Judy arrays are allocated on-the-fly, as you insert new elements into them.

You have been provided a skeleton `convert_to_judy` that takes the current linked list and returns a pointer to a Judy array. The Judy array will be indexed by the `location` field from the `mp1_blink_struct`.

```
void add_to_judy(mp1_blink_struct *current, judy_t *judy_array, int index);

judy_t *convert_to_judy(void);

judy_t *judy_init(void);
```

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

- `index` – the index into `judy_array` at which to insert the element. Taken from the `location` field of `mp1_blink_struct`.
- `judy_array` – the pointer to the judy array of `mp1_blink_structs`.
- `current` – the `mp1_blink_struct` to add to `judy_array`.

`judy_init` simply returns a pointer to a new, empty Judy array.

Your task is to complete `convert_to_judy` by filling in the call to `add_to_judy`, following the appropriate calling convention. The code is on the following page. You may assume that calls to `add_to_judy` never fail.

Question 3 continued

```
.long mp1_list_head

convert_to_judy:
     pushl %ebp
     movl %esp,%ebp
     pushl %ebx
     pushl %esi
     pushl %edi
     call judy_init
     movl mp1_list_head, %edx
     xorl %ecx, %ecx
CHECK_NEXT:
     cmpl $0, %edx
     je DONE_INSERTING
     movw LOCATION(%edx), %cx
# Insert call to add_to_judy below











     movl NEXT(%edx), %edx
     jmp CHECK_NEXT
DONE_INSERTING:
     popl %edi
     popl %esi
     popl %ebx
     leave
     ret
```

(12) (b) The figures below are the state of the stack before the execution of the code given below. Please fill in the state of the stack after execution of the code. In addition, please indicate where `%ebp` and `%esp` are pointing to at the end of execution. Treat registers whose values you do not know as variables; otherwise, please fill in the actual value.

Please use fig. 1 for scratch work, and write your final answer in fig. 2. Your work in fig. 1 will **not** be graded.

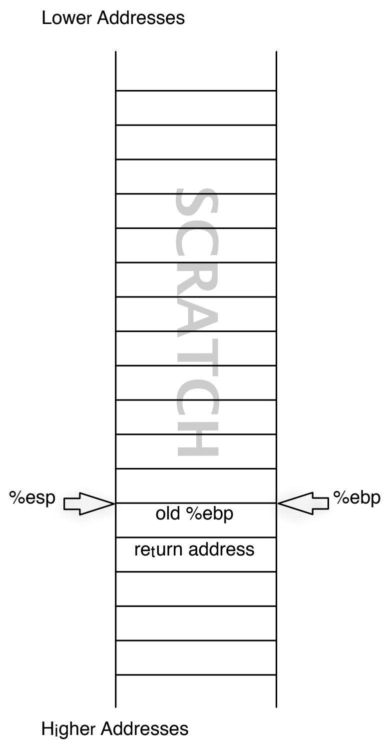


Fig. 1: Use this for scratch work.

```
        pushl  $0
        pushl  $21
        pushl  $38
        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, %edi
        pushl %edi
        leal  (%ebx,%esi,2), %eax
        movl %eax, 4(%esp)
```

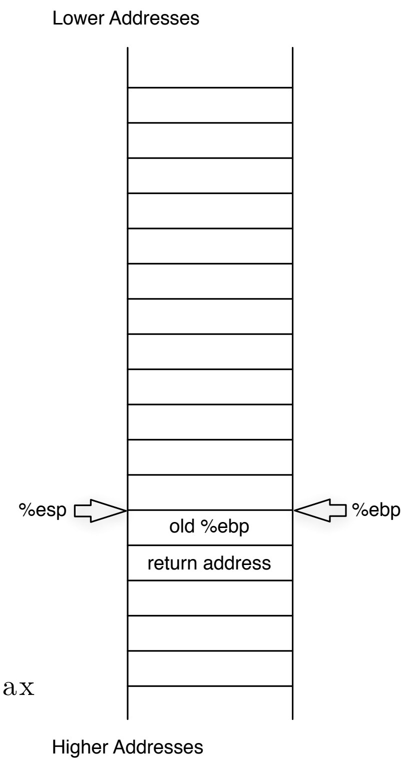


Fig. 2: Write your answer here.

Question 4: x86 Assembly (30 points)

A palindrome is a word that reads the same backward and forward. For example, RADAR is a palindrome. Your task is to write a recursive function in **x86 assembly** that detects whether a given string is a palindrome **using recursion**. The string is stored as a doubly-linked list and the structure of the linked list node is:

```
typedef struct node_t {
   char letter;          /* letter */
   struct node_t* next;  /* Pointer to the next element in the linked list */
   struct node_t* prev;  /* Pointer to the previous element in the linked list */
} node_t;
```

The C function prototype is:

```
/* is_palindrome()
 *  Description: Recursive function that checks if the string passed in
 *        between left and right is a palindrome
 *  Input: left - left pointer of the string being checked
 *         right - right pointer of the string being checked
 *  Output: -1 if string is not a palindrome, 0 if string is a palindrome
 */
int is_palindrome(node_t* left, node_t* right);
```

Additional Notes:

- Assume no compiler padding.
- You can assume the arguments passed in are valid types (No `NULL` checking or type checking required)
- To simplify things, you can assume the length of the string is odd.
- The initial call to `is_palindrome()` has the head and tail of the string as arguments.
- You must adhere to the rules of the C calling convention taught in class.

You may wish to write the function in C first, using the space below. Your C code **will not** be graded. It is for your convenience only.

Question 4 continued

```
# typedef struct node_t {
#    char letter;          /* letter */
#    struct node_t* next;  /* Pointer to the next element in the linked list */
#    struct node_t* prev;  /* Pointer to the previous element in the linked list */
# } node_t;

# is_palindrome()
#  Description: Recursive function that checks if the string passed in
#        between left and right is a palindrome
#  Input: left - left pointer of the string being checked
#         right - right pointer of the string being checked
#  Output: -1 if string is not a palindrome, 0 if string is a palindrome
#
# int is_palindrome(node_t* left, node_t* right);


is_palindrome:
```

You may tear off this page to use as a reference

Synchronization API reference

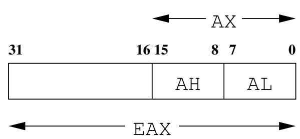
`spinlock_t lock;`	Declare an uninitialized spinlock
`spinlock_t lock1 = SPIN_LOCK_UNLOCKED;` `spinlock_t lock2 = SPIN_LOCK_LOCKED;`	Declare a spinlock and initialize it
`void spin_lock_init(spinlock_t* lock);`	Initialize a dynamically-allocated spin lock (set to unlocked)
`void spin_lock(spinlock_t *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 and then set processor 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	