ECE 391 Exam 2, Spring 2013

Tuesday, April 2nd

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.**
- **The last few pages of the exam contain reference information and scratch paper.**
- **Absolutely no interaction between students is allowed.**
- **Show all of your work.**
- **Don't panic, and good luck!**

Total	100 points	________________
Page 3	10 points	________________
Page 4	10 points	________________
Page 5	5 points	________________
Page 6	10 points	________________
Page 8	15 points	________________
Page 9	10 points	________________
Page 10	10 points	________________
Page 11	10 points	________________
Pages 13-14	20 points	________________

Problem 1 (25 points): Short Answers

✗**Part A** (3 points): Suppose a request is made for a large chunk of memory that exceeds the smallest page size supported by the kernel. Give three reasons why it is beneficial for the memory allocator to provide either contiguous page frames or a single large enough page, instead of randomly picking multiple (smaller) free page frames.

① contiguous page frames lead to higher memory throughput

② caching between processor and memory would be faster

③ reduce the chance that some programs' request to allocate large and contiguous memory

reduce the probability of TLB miss / reduce number of page table

✗**Part B** (2 points): Any memory allocator for dynamic requests of various sizes from a common memory pool (such as a buddy allocator) has to implement two core operations on that memory pool, one on allocation and one on deallocation. What are they?

Part C (2 points): Explain why disabling interrupts is the only precaution necessary for protecting a critical section that is shared between an application and an interrupt handler in the development environment for this class.

~~Because~~ the class only consider single ~~process~~

~~So disabling interrupts~~ is sufficient.

~~only~~ a single processor

one processor

Part D (3 points): Give one reason interrupt chaining is useful.

Because the number of IRQ is fixed according to IDT,

interrupt ~~chaining allow for more interrupts,~~

~~support more devices~~

① one device needs to have more than one handler

②

Problem 1, continued:

Part E (2 points): Name one disadvantage AND one limitation of the EXT2 file system.

dis advantage: i_block has fixed size 15, take too much space for small files

lack of journling high data fragmentation

limitation: block size can't be dynmically changes

maximum file size

Part F (3 points):

Please fill-in the following sentence with the following terms: User Program, Interrupt Handler, System Call

Operating systems leverage special assembly linkage within the (A) and a particular calling convention to allow the (B) to request the service of the desired (C).

A) ~~system call~~ Interrupt Handler

B) User Program

C) ~~Interrupt Handler~~ system call

Part G (2 points): Linux uses x86 segmentation in a very limited way. All processes in user mode use the same pair of segments, user code segment and user data segment. Similarly, all processes in kernel mode use a pair of segments, kernel code segment and kernel data segment. The Linux operating system effectively makes segmentation a pass-through operation by the way it establishes those segments. Please write the base address (starting address) of these four segments in Linux.

0x00000 for all of them

✓

Part H (3 points): Virtual memory acts as a logical layer between the application memory requests and the hardware Memory Management Unit (MMU). Please list three advantages of using virtual memory instead of sending application memory addresses directly to the MMU.

~~① decrease internal fragmentation~~

② add one more indirect layer, protection up↗

③

Problem 1, continued:

Part I (2 points): In lecture, one of the solutions we came up with for connecting multiple devices to the single interrupt pin was to connect all devices to an OR gate. Give two reasons why this solution is not used.

don't know which device raise

lack priority

Part J (3 points): Explain why neither the master nor slave PIC alone has enough information to know what data to send to the CPU during an INTA. For full credit, you must explain why that is the case for both PICs.

because neither of them know whether if it's their IRQ raise the interrupt

Problem 2 (10 points):

It is fairly common for FPGA's to have larger memory storage standards than an x86-style 32-bit addressable memory. One of your friends has designed a Memory-Management Unit for an x86-style architecture (byte-addressable) to deal with translating a 36bit virtual address space into a 32bit physical address space. There is an x86_32-style 2-level paging system (Page Directory → Page Table → Page) and each entry is 32 bits. The fields in the virtual address are split evenly for use in indexing into the page directory, the page table, and the page itself.

Part A (2 points): How many entries are in the PD? in the PT?

$\frac{36}{3} = 12$ $\quad 2^{12}$ ✓ $\quad 2^{12}$ ✓

4096 $\quad$ 4096

Part B (2 points): What is the biggest possible page size? (Assume the Page Size Extension bit has been set)

$2^{12} \cdot 2^{12} \cdot 1B = 16MB$ ✓

Part C (2 points): If only 10 bits are used for options/modifiers in the PTE, what is the smallest possible page granularity supported by a PTE?

2^{22} $\quad 2^{10}$ ✓ $\quad$ 1kB

Part D (2 points): Why is using the smallest granularity possible in the PTE a bad idea for this paging scheme?

Because the max offset is 2^{12} which is larger than $2^{10} = 1kB$ ✓ causing lots of waste

Part E (2 points): Why would recycling this MMU scheme for a 32bit virtual address space (and zero-extending by 4 bits) be a bad idea?

lots of PDE can't be used

causing lots of waste ✓

?

Problem 3 (15 points): Fill in the blanks on the next page with entries from the following list of possible tasks. (note: you will not use every listed task and you may leave some blanks empt)

This particular system call has the following prototype:

```
long sys_open (const char* name, int flags, int mode);
```

open_wrapper • Save all register values • Restore all register values • Save callee-saved registers 1 • Restore callee-saved registers • Push the parameters to stack 2 •Pop the parameters off stack • Move the parameters into registers • Call the system_call function • Throw the 0x80 interrupt (int 0x80) 3 • RET • IRET • Return • Jump back to label • Push the system call number to stack • Pop the system call number off stack • Push the system call pointer to stack • Pop the system call pointer off stack • Move the system call number to a register • Move the system call pointer to a register	system_call • Save all register values • Restore all register values • Save callee-saved registers 4 • Restore callee-saved registers • Push the parameters to stack • Pop the parameters off stack • Move the parameters into registers 5 • Call the system_call function • Throw the 0x80 interrupt (int 0x80) • RET • IRET • Return • Jump back to label • Call via given pointer to system call • Call via system call number and syscall table 6 • Push the return value to stack • Pop the return value off stack • Ensure the return value is in a register • Ensure the return value is on the stack
sys_open: • RET • IRET • return • Jump back to label	

func1();
open();

Name: ______________________________

Problem 3, continued:

user stack open_wrapper (x86)	kernel stack system_call (x86)	kernel stack sys_open (C)

...
// note: assume all registers are in use
// Open the desired file:
save cullee reg
push para on stack
____
____
____
____
→
←
____
____
____
____
____
//Done: ready to return
RET
	__________
	__________
	__________
	__________
	→
	←
	__________
	__________
	put return value in proper place
	__________
	__________
	//Done: ready to return
	IRET
		systemcall
		//Execute the desired action
		//Done: ready to return
		return

Problem 4 (10 points): Buddy allocator

Let the pages available to the kernel memory allocator be defined in terms of offset P starting from a base index. It may be assumed that the page size is fixed. It may further be assumed that contiguous page indices refer to contiguous page frames. Contiguous page frames are used to form blocks, whose sizes are determined by the parameter `order`. The size of a block is 2^{order} pages. Each block is identified by the offset of the first page in the memory block.

We use buddy system algorithm for managing page allocation and de-allocation requests. The buddy allocator has two methods, shown below, and a current state given by the following table. The table lists the free blocks available at each order.

```
/*
This function allocates contiguous pages.
Input: order: size of block as 2^order pages
Output: On success returns page index allocated.
        On failure to allocate a page, returns  -1.
*/
int pg_alloc(int order);

/*
Input:
    block_id: First page index of the block being freed
    order: size of block as 2^order pages
Output: Final order of the block after de-allocation is done.
*/
int pg_free(int block_id, int order);
```

Order	Free Block Ids
0	1, 3, 5, 8
1	6, 12, 16, 24
2	
3	
4	80, 112, 144
5	32, 160, 256
6	
7	
8	

The following calls are invoked on this buddy allocator in sequence. Write the value returned by each function call in the space provided next to that function call, and the final state of the buddy allocator in the table below, using the same format as the initial table above.

1. `pg_alloc(8)` Returns:
2. `pg_alloc(2)` Returns:
3. `pg_free(22, 1)` Returns:
4. `pg_free(96, 4)` Returns:
5. `pg_free(128, 4)` Returns:

Order	Free Block IDs
0	
1	
2	
3	
4	
5	
6	
7	
8	

Problem 5 (10 points): MP2 - ModeX

In MP2, you drew an image into the build buffer, then copied the entire screen area from the build buffer to the io-mapped video memory. This only worked for the MP due to some subtleties with using a VM. In this question, you will be taking advantage of the plane system of ModeX in order to draw a solid shape directly to video memory. Implement the `draw_horizontal_line` function below using the following information:

- `(x, y)` is the leftmost coordinate of the line.
- `length` is the length of the line in pixels.
- Assume the line fits within the valid screen area.
- `color` is the palette index for the fill color of the line.
- `SET_MASK(x)` will set the VGA plane mask based on the low 4 bits of x.
  i.e. 0001 sets the VGA to write to plane 0, 1010 sets the VGA to write to plane 3 and 1

NOTE: there are 10 blanks for you to fill in

```
#define SCREEN_X_DIM        320
#define SCREEN_Y_DIM        200
#define SCROLL_X_WIDTH   (SCREEN_DIM_X / 4);
static char *const mem_image; //points to the start of video memory

void draw_horizontal_line(int x, int y, int length, char color)
{
    int i;
    int end_x;
    int start_plane, end_plane;
    int start_addr, end_addr;
    if(length <= 0) {
        return;
    }
    end_x = x + length - 1; //x coordinate of the last pixel

    start_plane = x%4; //plane for first pixel

    end_plane = end_x%4; //plane for last pixel

    start_addr = x / 4 + SCROLL_X_WIDTH * y; //address of first pixel

    end_addr = end_x / 4 + SCROLL_X_WIDTH * y; //address of last pixel

    if(start_addr == end_addr) { //draw line which doesn't cross addresses
        SET_MASK((0xf<<start_plane) & (0xf>>(3 - end_plane))); //set plane mask

        mem_image[start_addr] = color;
    } else {
        SET_MASK(0xf<<start_plane); //set mask for first address

        mem_image[start_addr] = color;

        SET_MASK(0xf); //set mask for addresses between start and end
        for(i = start_addr + 1; i < end_addr; i++) {

            mem_image[i] =color;
        }
        SET_MASK(0xf>>(3 - end_plane)); //set mask for last address

        mem_image[end_addr] = color;
    }
}
```

1

1 2 3 0 1

0x1110

0x0111

Problem 6 (10 points): PIC design and interface

For this problem you may assume the system uses two cascaded Intel 8259A PICs. You may also assume the defined constants (i.e. EOI, MASTER_8259_PORT) are correct and defined elsewhere.

Ben Bitdiddle wrote your MP3 group's send_eoi function. Here is his code:

```
void send_eoi(uint32_t irq_num)
{

    if (irq_num & 8) {

        outb( (irq_num-8) | EOI, SLAVE_8259_PORT);

    } else {

        outb(irq_num | EOI, MASTER_8259_PORT);

    }

}
```

0~7
8~15

Ben says his send_eoi works, but he didn't test it very thoroughly. When you test it you notice that after a while some interrupts are no longer received. Provide a short answer for each of the following questions for full credit.

Part A (5 points): What is wrong with the function?

Part B (3 points):

Why does it only affect some interrupt lines?

Part C (2 points):

Which specific interrupt lines are affected?

Problem 7 (20 points): MP3 Filesystem

Consider the simplified file system described in your MP3 specification sheet (attached as a reference at the end of the exam). You are now required to extend the given file system to allow writes. To keep things simple, you must use the given constants and structs defined below and you are only required to implement a function that creates and writes to a new file.

```
void write_new_file (const uint8_t* filename, const void* buf, int32_t filelength);
```

This function takes in a file name, a data buffer, number of bytes to write. `write_new_file` creates a new directory entry at index 12. You then have to create a new inode for this file at index 24. Lastly, you have to create data blocks for the file and write the contents of the buffer to the data blocks starting at index 36.

The following helper functions are to be written by you to aid in implementing `write_new_file` and for partial credit:

```
void create_dir_entry (const uint8_t* filename);
void create_inode (int32_t filelength);
void write_data_blocks (const void* buf, int32_t filelength);
```

Things you may assume:
- All memory is statically allocated for you already to create this new file.
- Directory entry at index 12 is the next free directory entry index.
- Inode 24 is the next free inode index and the first data block starts right after the created inode(s) for the new file.
- Similarly, data block index 36 is the next free data block index and data is written to data blocks contiguously.
- The new file will not take up more than one inode.
- Arguments to your functions will be valid.

```
// Variables:
bootblock_t* Fs_start //starting address of file system

// Constants:
#define BLOCK_SIZE      4096

// Library functions:
void* memcpy(void* dest, const void* src, uint32_t n);
int8_t* strcpy(int8_t* dest, const int8_t* src);

// Directory entry structure
typedef struct dentry {
  int8_t filename[32];
  int32_t filetype;
  int32_t inode;
  uint8_t de_reserved[24];
} dentry_t;

// Boot block structure
typedef struct bootblock {
  int32_t dir_ent_number;
  int32_t inode_number;
  int32_t data_block_number;
  uint8_t bb_reserved[52];
  dentry_t dir_ent_table[63];
} bootblock_t;

// Inode structure
typedef struct inode {
  int32_t filelength;
  int32_t  data_block_table[1023];
} inode_t;
```

Problem 7, continued:

Write your code here. Aside from the ones we already gave you, STATE ANY ASSUMPTIONS!

```
/*
 * create_dir_entry:
 *   DESCRIPTION: this function creates a new directory entry at index 12.
 *   INPUTS: filename: name of new file
 *   OUTPUTS: None
 *   RETURN VALUE: Void
 */
void create_dir_entry (const uint8_t* filename)
{
    dentry_t newdentry.
    strcpy (newdentry.filename, (int8_t*) filename );
  newdentry.filetype = 2;
   newdentry.inode = 24;
    Fs.start ->dir_ent_number,
    Fs.start ->dir_ent_table[12]= newdentry;
}


/*
 * create_inode:
 *   DESCRIPTION: this function creates new inode(s) at index 24.
 *   INPUTS: filelength: length of new file
 *   OUTPUTS: None
 *   RETURN VALUE: Void
 */
void create_inode (int32_t filelength)
{
    Fs.start->inode_number++;
    inode_t* inodes =(inode_t*)(Fs.start +1);
    inodes += 24;
    inodes -> filelength = filelength;
    ~~inodes ->data_block_table[0]=36;~~
   int block_num  ;
}
   if (filelength%4096 ==0 ) {
         blocknum= filelength/4096;
    } else{
         blocknum = filelength/4096+1;
    }
    int i=36; int j=0;
     while (blocknum >0){
```

blocknum - - ;
inodes -> data_block_table[j++] = i++)

3

Problem 7, continued:

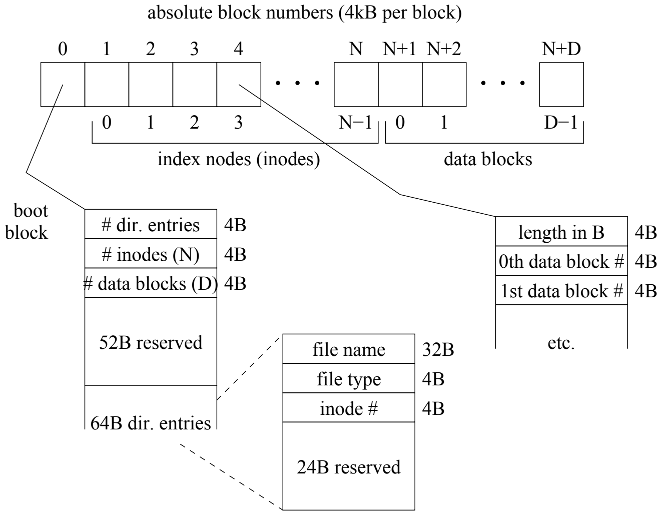
```
/*
 * write_data_blocks:
 *   DESCRIPTION: this function creates new datablock(s) at index 36.
 *   INPUTS: buf: pointer to a buffer that holds contents of new file
 *      filelength: length of new file
 *   OUTPUTS: None
 *   RETURN VALUE: Void
 */
void write_data_blocks (const void* buf, int32_t filelength)
{
        inode_t* inodes = (inode_t*)(FS start+1);
        int8_t* datablocks = (int8_t*)(inodes+25);
        datablocks += BLOCK_SIZE*36;
        memcpy (datablocks, buf, filelength)

}

/*
 * write_new_file:
 *   DESCRIPTION: this function creates and writes to a new file. This involves
 *       creating a directory entry, inode(s), and data block(s) in the file system.
 *   INPUTS: filename: name of new file
 *           buf: pointer to a buffer that holds contents of new file
 *      filelength: length of new file
 *   OUTPUTS: None
 *   RETURN VALUE: Void
 */
void write_new_file (const uint8_t* filename, const void* buf, int32_t filelength)
{
        create_dir_entry(filename);
        create_inode [ filelength);
        write_data_blocks ( buf, filelength);

}
```

(Scratch paper. Remove if desired. Contents of this page will not be considered during grading.)

You may tear off this page to use as a reference

MP3 Filesystem

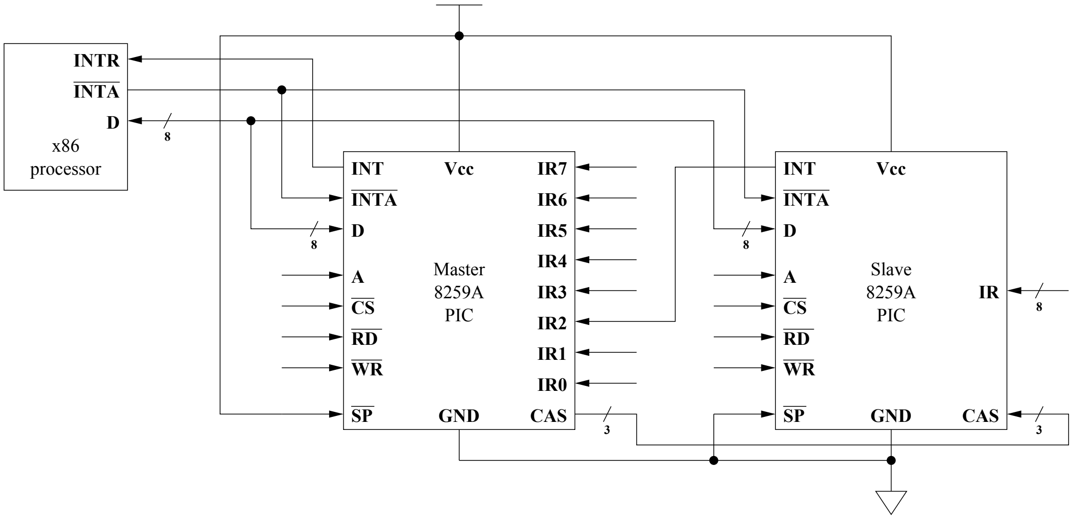
File System Utilities: The figure below shows the structure and contents of the file system. The file system memory is divided into 4 kB blocks. The first block is called the boot block, and holds both file system statistics and the directory entries. Both the statistics and each directory entry occupy 64B, so the file system can hold up to 63 files. The first directory entry always refers to the directory itself, and is named ".", so it can really hold only 62 files.



Each directory entry gives a name (up to 32 characters, zero-padded, but *not necessarily including a terminal EOS or 0-byte*), a file type, and an index node number for the file. File types are 0 for a file giving user-level access to the real-time clock, 1 for the directory, and 2 for a regular file. The index node number is only meaningful for regular files and should be ignored for the RTC and directory types.

Each regular file is described by an index node that specifies the file's size in bytes and the data blocks that make up the file. Each block contains 4 kB; only those blocks necessary to contain the specified size need be valid, so be careful not to read and make use of block numbers that lie beyond those necessary to contain the file data.

8259A Master-Slave PIC configuration


INTR
INTA
D
8
x86 processor
INT
INTA
D
8
A
CS
RD
WR
SP
Vcc
Master 8259A PIC
GND
IR7
IR6
IR5
IR4
IR3
IR2
IR1
IR0
CAS
3
INT
INTA
D
8
A
CS
RD
WR
SP
Vcc
Slave 8259A PIC
GND
IR
8
CAS
3