

# ECE 391 Exam 2, Spring 2010

Apr 7, 2010, 7–9 p.m.

Name	
NetID	

- Be sure that your exam booklet has 12 pages.
- Write your netid at the top of each page.
- This is a closed book exam.
- You are allowed two  $8.5 \times 11$ " sheet of notes.
- Absolutely no interaction between students is allowed.
- Show all of your work.
- Don't panic, and good luck!

NetID: \_\_\_\_\_

2

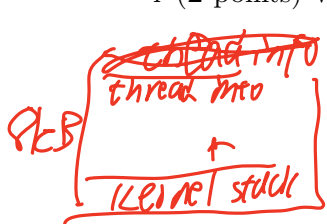
Problem 1	23 points	
Problem 2	11 points	
Problem 3	21 points	
Problem 4	18 points	
Total	73 points	

*(You may use the rest of this page as scratch material)*

## Problem 1 Short Answer (23 points)

## a Kernel memory allocation (8 points)

i (2 points) Why should you not allocate large objects on the stack in the kernel?

 Because a single per-process area in the kernel has only 8kB, containing thread id and kernel stack. If stack goes too large, it will destroy thread id.

↪ The kernel is limited to about 1GB of virtual and physical memory.

ii (4 points) Explain the difference between `kmalloc` and `kmem_cache_alloc` and the rationale for having the two mechanisms.iii (2 points) What mechanism would be used for allocating a `vmarea_struct`?

b (2 points) Explain the purpose of the Translation Lookaside Buffer (TLB)

Because the translation between virtual addr and physical addr involves too many memory access and is too slow, Use TLB to cache the translation and reuse them will make it quicker.

c (2 points) Why must the TLB be flushed when switching between processes?

Because when switching between processes, the ~~CR3 is reloaded~~, changing the base address of directory, causing TLB be flushed.

★ to ensure a process can't access data stored in memory pages of another process

d (2 points) What is the difference between a process and a thread?

*a whole program in execution*  
A process is a unit of scheduling, while thread is scheduled for execution inside a process.  
*a unit of execution*

- e (6 points) Describe two advantages and one disadvantage of using paging *instead of* segmentation.

adv: provide protection

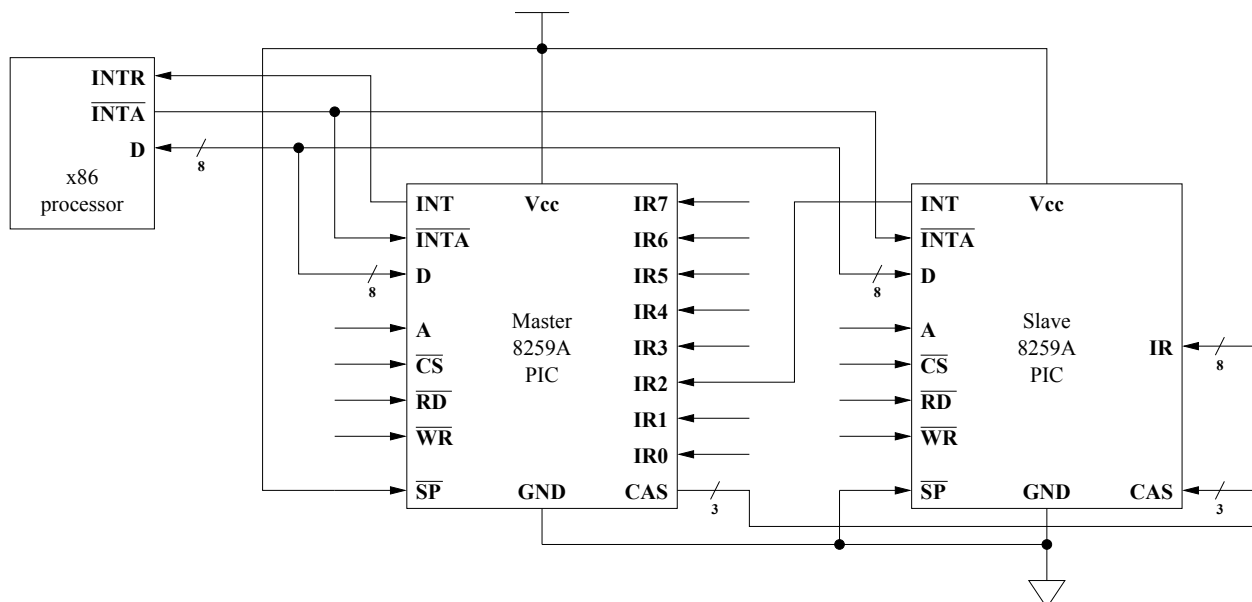
have more effective sharing

disadv: may cause internal fragmentation  
when allocate 4kB pages to very  
small data.

can requires dedicated hardware (TLB)

- f (3 points) List IRQs (0-15) in decreasing priority order (i.e., highest priority first) (Note: you do not have to explain what these IRQs are associated with.) Diagram of the PICs is included below for reference.

IRQ0, IRQ1, IRQ2~7, IRQ3~7  
master master slave master ✓



## Problem 2 ~~Memory~~ Maps (11 points)

On this page, you are presented with the memory maps for two programs, `cat` and `tac`<sup>1</sup>. These are edited versions of what you would see in `/proc/pid/maps` on Linux.

1.	08048000-0804c000	r-x	4	2296660	/bin/cat
2.	0804c000-0804d000	rw-	1	2296660	/bin/cat
3.	09d82000-09da3000	rw-	33	0	[heap]
4.	b7de0000-b7f18000	r-x	312	13238948	/lib/libc-2.7.so
5.	b7f18000-b7f19000	r--	1	13238948	/lib/libc-2.7.so
6.	b7f19000-b7f1b000	rw-	2	13238948	/lib/libc-2.7.so
7.	b7f26000-b7f40000	r-x	26	13238871	/lib/ld-2.7.so
8.	b7f40000-b7f42000	rw-	2	13238871	/lib/ld-2.7.so
9.	bf8e9000-bf8fe000	rw-	21	0	[stack]
1.	08048000-0804c000	r-x	4	121274994	/usr/bin/tac
2.	0804c000-0804d000	rw-	1	121274994	/usr/bin/tac
3.	08642000-08663000	rw-	33	0	[heap]
4.	b7e56000-b7f8e000	r-x	312	13238948	/lib/libc-2.7.so
5.	b7f8e000-b7f8f000	r--	1	13238948	/lib/libc-2.7.so
6.	b7f8f000-b7f91000	rw-	2	13238948	/lib/libc-2.7.so
7.	b7f9c000-b7fb6000	r-x	26	13238871	/lib/ld-2.7.so
8.	b7fb6000-b7fb8000	rw-	2	13238871	/lib/ld-2.7.so
9.	bf99f000-bf9b4000	rw-	21	0	[stack]

The fields are:

- Starting and ending virtual address of the mapped region
- Permissions (read, write, execute)
- Number of pages in map (decimal)
- i-node of the file used to provide the data, if any
- The file used to provide the data, if any

To help you understand the reading better, here is what each map is used for:

- 1 Program executable code for the `cat/tac` program
- 2 Global variables
- 3 Heap
- 4 Executable code from the C library
- 5 Constants used by the C library
- 6 Global variables used by the C library
- 7 Executable code from the `ld.so` library
- 8 Global variables used by `ld.so` library
- 9 Stack

---

<sup>1</sup>cat backwards

For each question below, please list your reasoning for obtaining the answer, as well as the number.

- a (**3** points) The `cat` process uses a total of 402 pages in memory. How much memory would be used by the `cat` and `tac` processes together? Assume that all pages are memory resident (not swapped out to disk) and do not include memory taken up by page tables or kernel data structures.
- b (**3** points) Now suppose the user starts another `cat` process. How much memory is taken up by the three processes? (`cat`, `cat`, and `tac`)
- c (**3** points) Suppose the `tac` process calls `fork()`. How much total memory would be in use by the four processes after the call `fork()` returns?
- d (**2** points) How many pages are taken up by the page tables for the initial `cat` process?

### Problem 3 ~~CPU Scheduling~~ (21 points)

Listed below is a table of processes and associated arrival and run times.

Process ID	Arrival Time	Run Time
A	0	4
B	2	3
C	4	5
D	6	1

- a (8 points) Show the scheduling order for these processes under First-In-First-Out (FIFO), Shortest-Job First (SJF) with run to completion, and Round-Robin (RR) with a quantum = 2 time units. Assume that the context switch overhead is 0 and new processes should be scheduled first if all else is equal. A process arriving at time  $t$  can first be scheduled for time slot  $t$  to  $t+1$ .

Time Slot	FIFO	SJF	RR
0-1			
1-2			
2-3			
3-4			
4-5			
5-6			
6-7			
7-8			
8-9			
9-10			
10-11			
11-12			
12-13			
13-14			
14-15			
15-16			
16-17			
17-18			



- b (4 points) For each process indicate the turnaround time (TT). The turnaround time is defined as the time a process takes to complete after it arrives.

Scheduler	Process A	Process B	Process C	Process D	Average
FIFO					
SJF					
RR					

- c (2 points) In practice, why is Shortest Job First (SJF) not always practical?
- d (3 points) Compute the average process turnaround time for Round Robin if a quantum size of 1 was used.
- e (4 points) What advantage is given by a shorter quantum time? In practice, why not make your quantum size as small as possible?

## Problem 4 Virtual Memory (18 points)

A 4-level page table system (Page Directory → Page Table 1 → Page Table 2 → Page) has been created for a computer with a 32-bit address. It is your job to create a page table walking function for this new page table system.

a **Parameters** (4 points) From the documentation of this 4-level page table system, you recognize the following properties about the size of the tables:

- The sizes of Page Directories, Tables, and Pages could be different (i.e., they are not all the size of a page like in Linux's 3-level page table system).
- The total size of the Page Directory is 4KB.
- The sizes of Page Table 1 and Page Table 2 are the same.
- The offset field in the virtual address is 10 bits wide.
- Page Table and Directory entries are 32 bits long.

Given this, list out the lengths of each of the fields in the 32-bit virtual address:

<b>Page</b>	PDE	=	<u>10</u>	bits
	PTE1	=	<u>6</u>	bits
	PTE2	=	<u>6</u>	bits
	Offset	=	<u>10</u>	bits

b (9 points) Next you found the important information on how each of the entries are stored:

- The Present bit is the last bit of each of the entries. If the Present bit is 1, then the entry is valid. Otherwise the entry is invalid.
- The address of the next level of the page tables is stored in the top bits of the entry.
- The other bits are reserved. You should not change or access them.
- Each Page Directory, Page Table, and Page are aligned to its own size.

Write the helper functions below to move from one level of the page tables to the next.

```
/* Takes in the base address of Page Directory and virtual address as the argument.
 * Returns the base address of Page Table 1 if valid.
 * If invalid, returns NULL
 */
```

```
uint32_t* PD_to_PT1 (uint32_t* PD_addr, uint32_t v_addr)
```

```
{
    if (PD_addr == NULL) return NULL;
    uint32 offset = (v_addr & 0xFFC00000) >> 22;
    uint32_t* PT1 = PD_addr + offset;
    if (PT1 & 0x1 != 1) return NULL;
    return (PT1 & 0xFFFFF00);
}
```

```
}
```

```

/* Takes in the base address of Page Table 1 and virtual address as the argument.
 * Returns the base address of Page Table 2 if valid.
 * If invalid, returns NULL
 */

```

```

uint32_t* PT1_to_PT2 (uint32_t* PT1_addr, uint32_t v_addr)
{

```

*Handwritten code:*

```

    if (PT1_addr == NULL) return NULL;
    uint32 offset = (v_addr & 0x003F0000) >> 16;
    uint32_t* PT2 = PT1_addr + offset;
    if (PT2 & 0x1 != 1) return NULL;
    return (PT2 & 0xFFFFF00);

```

```

}

```

```

/* Takes in the base address of Page Table 2 and virtual address as the argument.
 * Returns the base address of the Page if valid.
 * If invalid, returns NULL
 */

```

```

uint8_t* PT2_to_Page (uint32_t* PT2_addr, uint32_t v_addr)
{

```

*Handwritten code:*

```

    if (PT2_addr == NULL) return NULL;
    uint32 offset = (v_addr & 0x0000FC00) >> 10;
    uint32_t* Page = PT2_addr + offset;
    if (Page & 0x1 != 1) return NULL;
    return (Page & 0xFFFFC00);

```

*Handwritten note:* 1B addressable

```

}

```

- c (5 points) Write the full page table walking function to translate a virtual address into its corresponding physical address.

```
uint32_t* PD_to_PT1 (uint32_t* PD_addr, uint32_t v_addr);
uint32_t* PT1_to_PT2 (uint32_t* PT1_addr, uint32_t v_addr);
uint8_t* PT2_to_Page (uint32_t* PT2_addr, uint32_t v_addr);

uint32_t get_Physical_Address (uint32_t* PDBR, uint32_t v_addr)
{
```

Same as Fall 2015

```
}
```