ECE391 Final Exam, Fall 2019

Wednesday, December 18th, 8AM

Name and NetID:

- **Write your name at the top of each page.**
- **This is a closed book exam.**
- **You are allowed THREE 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	18	points	______
Problem 2	18	points	______
Problem 3	14	points	______
Problem 4	19	points	______
Problem 5	18	points	______
Problem 6	13	points	______
Total	100	points	______

Problem 1 (18 points): Short Answer

Part A (8 points): For each of the following statements, circle TRUE or FALSE.

TRUE **FALSE** When `fish` executing under your MP3 OS is switched from the foreground terminal into the background, no TLB flush is needed, since the video memory's virtual address remains unchanged.

TRUE **FALSE** If a signal generated for a task is currently masked by the task, the signal is discarded.

TRUE FALSE In Linux, memory allocated using a slab cache is always contiguous in physical memory.

TRUE **FALSE** After an invalid memory access causes a page fault, the `cr3` register contains the accessed address.

?

Part B (4 points): Linux divides devices into two types: block devices and character devices.
USING 20 OR FEWER WORDS for each answer, explain two differences between the two types.

1. transfer to/from device are buffered for block devices not buffered for character devices

2. addressed randomly for block devices, could be read sequentially for character devices

Part C (2 points): In MP3, which descriptor table holds a pointer to the TSS?
CIRCLE EXACTLY ONE ANSWER.

GDT LDT IDT

Problem 1, continued:

Part D (2 points): Your friend has implemented the `execute` system call nearly correctly for MP3, but has accidentally written the wrong value into the `esp0` field of the TSS. When does your friend's OS crash?
CIRCLE EXACTLY ONE ANSWER.

A. when an interrupt occurs

?

B. when the user program makes a system call

-1

C. immediately after executing IRET

D. both A and B

E. none of the above

Part E (2 points): In the Linux kernel, can a system call acquire a semaphore (`struct semaphore`) while holding a spin lock (`spinlock_t`)? **USING 30 OR FEWER WORDS**, explain your answer.

No. trying to acquire semaphore may put calling process to sleep. shouldn't happen while holding spin lock.

You should not need to write below this line.

Problem 2 (18 points): Scheduling

Part A (4 points): Consider two simple scheduling schemes—FIFO (First-In, First-Out) and round-robin—defined similarly to those discussed in class in the context of Linux. Both use a queue to keep track of runnable tasks, and both add tasks to the end of the queue when the tasks arrive. The two schedulers differ in how they choose to run tasks in the queue.

The FIFO scheduler removes the task at the head of the queue and executes that task until it completes.

Every 2 msec, the round-robin scheduler removes the task at the head of the queue and executes that task until it completes, or until 2 msec have elapsed. If the task completes first, the scheduler starts over with the task now at the head of the queue. When 2 msec have elapsed, *regardless of whether the task currently being executed is the same as the one started 2 msec ago*, the scheduler places the currently executing task at the end of the queue and starts over from the head of the queue.

For either scheduler, if the queue is empty, the scheduler idles until a task appears.

Being sure to follow the rules given above, fill in the table on the left below to show which tasks execute during each msec using each of the two schedulers and the tasks shown in the table to the right. Both schedulers should execute **A** in the 0-1 msec period, for example.

	Scheduling Scheme	
Time (msec)	FIFO	Round Robin
0 - 1	A	A
1 - 2	A	A
2 - 3	A	B
3 - 4	B	B
4 - 5	B	~~A~~ C
5 - 6	C	~~C~~ A
6 - 7		
7 - 8		
8 - 9	D	D
9 - 10	D	D
10 - 11	D	E
11 - 12	D	E
12 - 13	D	D
13 - 14	E	D
14 - 15	E	F
15 - 16	E	F
16 - 17	E	E
17 - 18	F	E
18 - 19	F	D
19 - 20	F	F
20 - 21	F	F
21 - 22	F	F
22 - 23	F	F
23 - 24		

1 2
A-B
2 1 1
B-A-C

A-B-C

A - C
5 4
D - E
4 5 6
E-D-F
3 6 2
D-F-E
F-E-D
3
F

Task	Arrival Time (msec)	Duration (msec)
A	0	3
B	1	2
C	1.5	1
D	8	5
E	9	4
F	11	6

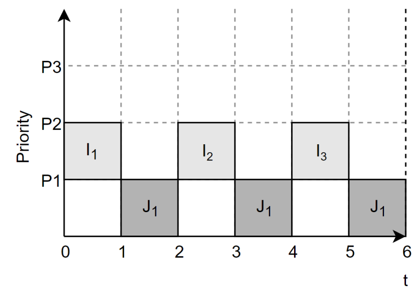
Problem 2, continued:

The mix of tasks for embedded applications is frequently known in advance, and each task needs to execute periodically. In particular, each task has a period and a per-period duration. A task becomes runnable at the start of each of its periods (starting at time 0) and should ideally be run immediately for the duration of the task. However, if more than one task is runnable, the task with the shortest period is given priority, which may mean that lower-priority tasks are delayed. This type of scheduling is known as *priority-based scheduling*.

Consider the example shown to the right, in which tasks I and J are scheduled using *priority-based scheduling*. At time 0 msec, all tasks are runnable, so the scheduler chooses the highest priority task, I, and executes it for I's duration of 1 time unit. Task I is not runnable again until time 2 msec, so at time 1 msec, the scheduler executes task J. At time 2 msec, the scheduler preempts task J to execute the second period of task I (I_2 in the figure). At time 3 msec, only task J is runnable again, and so the first period of task J runs until time 4 msec, at which point the third period of task I becomes runnable and preempts task J again. Task I_3 completes at time 5 msec, allowing the first period of task J to resume execution and to complete at time 6 msec.

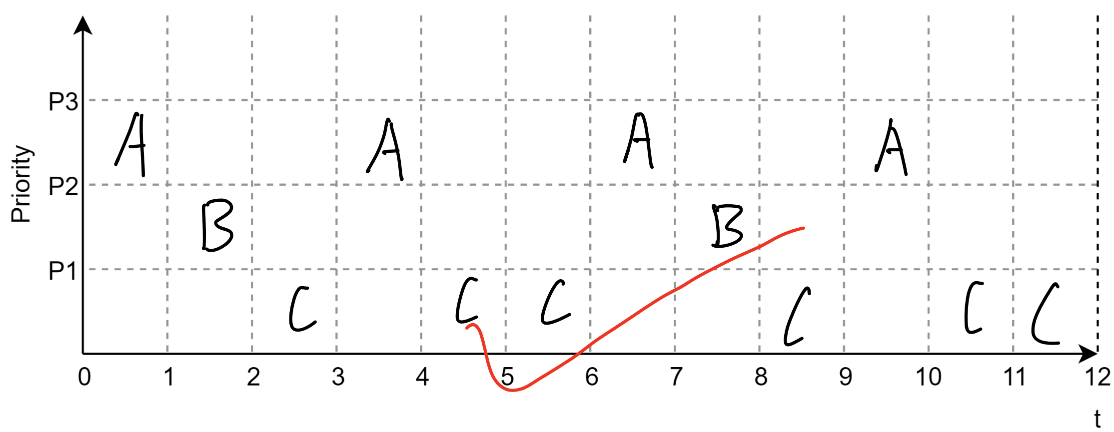
Task	Period	Duration	Priority
I	2 msec	1 msec	P2
J	6 msec	3 msec	P1

$P_2 > P_1$



Part B (6 points): Use *priority-based scheduling* to assign priorities (P1, P2, and P3) to each task in the table below, then complete the diagram to show the schedule from time 0 to 12 msec. Note that P3 denotes the highest priority, while P1 denotes the lowest priority.

Task	Period	Duration	Priority
A	3 msec	1 msec	P3
B	6 msec	1 msec	P2
C	12 msec	6 msec	P1



Part C (2 points): Is it possible for the tasks defined in **Part B** to build up an infinite amount of unfinished work as time progresses? **USING 20 OR FEWER WORDS**, explain your answer in terms of your solution to **Part B**.

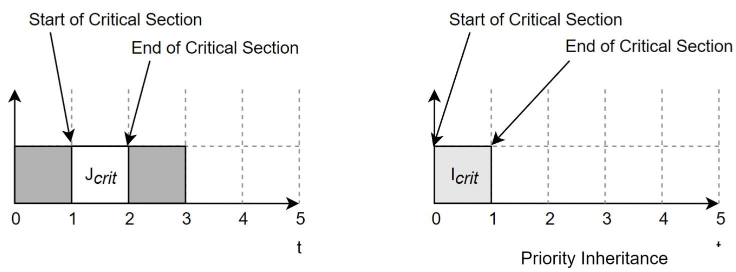
No. Because each task finishes its duration before its period expires.

Problem 2, continued:

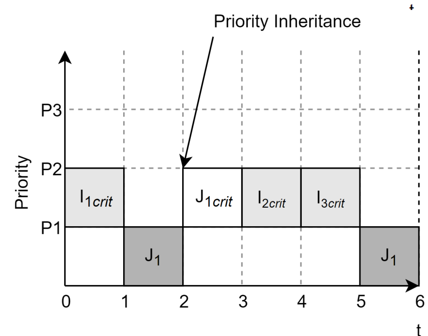
Now imagine that tasks executed by a *priority-based scheduler* share a resource protected by a mutex R. If a low-priority task holds the mutex, a high-priority task that tries to acquire the mutex becomes unrunnable. However, a medium-priority task can still preempt the low-priority task, thus indirectly delaying execution of the high-priority task! This problem is called *priority inversion.*

To solve this dilemma, some smart computer engineers invented *priority inheritance*, in which a task that holds a mutex executes at the maximum priority over all tasks waiting to acquire the mutex (and the mutex owner's priority). With *priority inheritance*, the low-priority task in the example of the previous paragraph executes at high-priority until it releases the mutex, at which point the high-priority task becomes runnable and preempts the low-priority task.

The figure to the right gives a simplified example, illustrating critical sections for tasks I and J (from the previous example). In the example, if task I tries to acquire the mutex while task J holds it, task J is promoted to P2 until it releases the mutex.

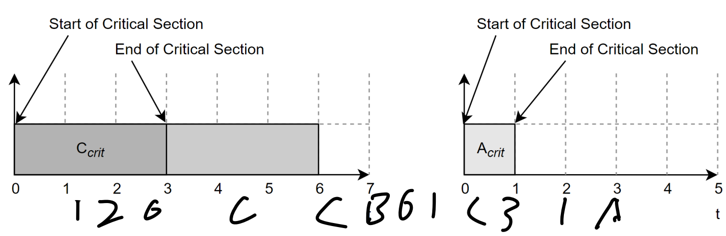


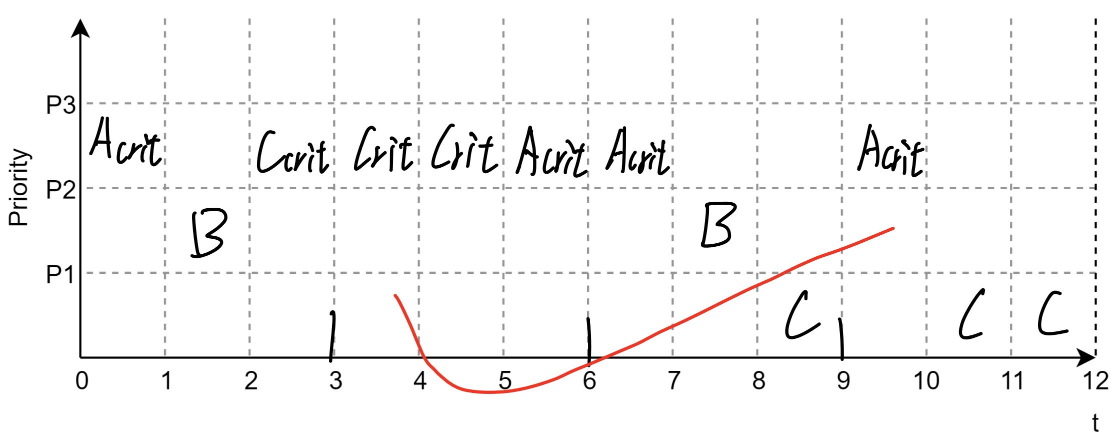
The two tasks are now scheduled as shown to the right. Here we have assumed that task J acquires the mutex just before time 2 msec, and thus inherits the priority of task I at time 2 msec (after a brief execution of task I to attempt to acquire the mutex, which is not shown in the figure).



Part D (6 points):

Suppose that tasks A and C from **Part B** share a resource protected by mutex Q. The critical sections for the two tasks are shown to the right. Using the priorities from **Part B** and the idea of *priority inheritance*, fill in the diagram below to indicate when each job is scheduled and at what priority.





Problem 3 (14 points): MP3

John and Stacy Stackman, and Ben and Betty Bitdiddle are working on MP3 together. Your task is to help them by answering the questions below correctly.

Part A (6 points): Betty was tasked with implementing the common system call assembly linkage for the ten system calls in MP3. Her implementation is given below. Unfortunately, Betty has **THREE** bugs in her implementation. **Clearly label** the line(s) where the bugs are introduced in a similar manner as shown and explain what the effect of each bug would be **USING 20 OR FEWER WORDS** for each explanation. One has already been marked for you.

```
system_call_linkage:
    pushl %ebp
    movl %esp, %ebp
    pushal

    cmpl $0, %eax
    je invalid_call
    cmpl $10, %eax
    jae invalid_call          — - - - #XXXX-- BUG B ---XXX

    pushl %edi
    pushl %esi
    pushl %edx
    pushl %ecx
    pushl %ebx
    jmp *sys_calls(, %eax, 4)            # XXXX--- BUG A ---XXXX
    popl %ebx
    popl %ecx
    popl %edx
    popl %esi
    popl %edi
    jmp exit_sys_linkage

invalid_call:
    movl $-1, %eax

exit_sys_linkage:
    popal   bug return value lost   #XXXX --- BUG C ---XXX
    iret

sys_calls:          1      2         3     4      5     6       7        8
    .long 0, halt, execute, read, write, open, close, getargs, vidmap,
          set_handler, sigreturn
               9          10
```

Bug A effect: should use call instead of jmp, the sys-call func can't return to this linkage RET in system call function may cause an exception EBX is stored on top of stack

Bug B effect: can not use syscall sigreturn, which has syscall number 10

Bug C effect: pushled ebp is not poped, so the stack is damaged, IRET will return to caller's EBP

Problem 3, continued:

Part B (5 points): During **Checkpoint 4**, John was tasked with implementing the `vidmap` system call. His implementation is shown below. Unfortunately, his implementation contains a couple of errors and is incomplete. Complete and correct John's implementation while making as few changes as possible by writing in the blank spaces and/or striking out lines of the code below. **Making unnecessary changes will result in negative points**.

Note: Video memory is present at physical address `0xB8000` and the user page starts at virtual address `0x08000000`. You may assume that the team does not have to handle multiple terminals and does not need to implement a scheduler for this question. `pcb_pointer` is a global variable that contains a pointer to the current process' pcb.

```
/* initialize_vidmap_pages
 * DESCRIPTION: Sets up the page directory entry and page table entry for
 *              vidmap video memory
 * INPUTS:  position -- index in the Page Table
 */
void initialize_vidmap_pages(uint32_t position);

/* 8. vidmap
 * DESCRIPTION: Provides a location in memory to the user to access video memory
 * INPUTS:  screen_start -- double pointer that needs to point to a user-level
 *                          copy of video memory
 * RETURNS: 0 if successful, -1 if failed
 */
int32_t vidmap(uint8_t** screen_start) {

  if (screen_start == NULL_____________ ||

      (uint32_t)screen_start < 0x08000000 ||

      (uint32_t)screen_start > 0x08400000_____ ) {
                               0x083FFFFC
      return -1;
  }

  ________________________________________

  pcb_pointer->is_vidmapped = 1;
                                        takes index
  initialize_vidmap_pages(0xB8000);
  *screen_start = (uint8_t*) 0x08400000;

  **screen_start = (void*)(0x070B8000);

  ________________________________________

  return 0;
}
```

20

27

24

2^{27} 2^{22}

128MB + 4MB

$2^{7}+2^{2}$

Problem 3, continued:

Part C (3 points): Stacy is trying to add the functionality of multiple terminals to the team's OS before they implement scheduling. She decides to implement the following strategy to handle video memory access within the kernel for the three terminals.

> Allocate three 4 kB pages in physical memory for each terminal at addresses `0xB8000`, `0xB9000`, and `0xBA000` respectively. When switching from terminal 1 to terminal 2, change the paging structures such that the virtual video memory address (`0xB8000`) points to the new terminal's physical page.

Will Stacy's implementation work? Clearly **CIRCLE ONE OPTION**.

YES NO

Justify your choice **USING 20 OR FEWER WORDS**.

~~Because the data of terminal 2 stored in its physical page will be overwritten by terminal 1.~~

physical video memory always resides at 0xB8000, so only terminal #1 is visible at any time

You should not need to write below this line.

Problem 4 (19 points): Memory Odyssey

The memory maps for two programs, `cat1` and `cat2`, are shown below. These are edited versions of what you might see in `/proc/pid/maps` when running these programs concurrently on Linux.

```
1.  08048000-0804e000 r-x 6  33360    /usr/bin/cat1
2.  0804e000-0804f000 rw- 1  33360    /usr/bin/cat1
3.  09d82000-09d8b000 rw- 9  0        [heap]
4.  b7de0000-b7df9000 r-x 25 13238    /lib/libc-2.7.so
5.  b7df9000-b7dfa000 r-- 1  13238    /lib/libc-2.7.so
6.  b7dfa000-b7dfc000 rw- 2  13238    /lib/libc-2.7.so
7.  bf8e9000-bf8f0000 rw- 7  0        [stack]

1.  08048000-0804e000 r-x 6  33361    /usr/bin/cat2
2.  0804e000-0804f000 rw- 1  33361    /usr/bin/cat2
3.  08642000-0864b000 rw- 9  0        [heap]
4.  b7e56000-b7e6f000 r-x 25 13238    /lib/libc-2.7.so
5.  b7e6f000-b7e70000 r-- 1  13238    /lib/libc-2.7.so
6.  b7e70000-b7e72000 rw- 2  13238    /lib/libc-2.7.so
7.  bf99f000-bf9a6000 rw- 7  0        [stack]
```

From left to right, the fields are:

- The memory region number within the process.
- Starting and ending virtual address of the mapped region
- Mapped region permissions (read, write, execute)
- Number of pages in map (decimal)
- inode of the file used to provide the data, if any
- The file used to provide the data, if any

Each row represents a specific memory region. In particular, the seven rows in each block are memory usage records for the following:

1. Program executable code
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. Program stack

Problem 4, continued:

Part A (2 points): One advantage of virtual memory is the ability to share physical memory between programs. **USING 30 OR FEWER WORDS**, explain how this advantage is realized in the example programs on the previous page. Be specific when mentioning regions (by number).

cat1 and cat2 can have shared library and that shared physical memory, which are regions 4,5,~~6~~.

Part B (2 points): Your friend suggests that one can further reduce wasted memory by sharing the heap across all programs executing at the same time, since partially-filled pages can then be filled with other programs' live data. **USING 20 OR FEWER WORDS**, explain one disadvantage of doing as your friend suggests.

The protection between programs are destoried as they can access eachother's heap.

Part C (3 points): On some systems, when a library is statically-linked, the compiler includes only those library functions actually used by the program in the program image. When the library is dynamically-linked, the entire library is mapped into memory.

Given the statistics on the C library from the previous page, and assuming that each program uses only 10% of the library code, how many programs must be executed concurrently before the use of shared libraries saves memory overall? Explain your answer **USING 50 OR FEWER WORDS.** For full credit, be specific about per-program memory usage.

10 ?

12 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15

0x200 0x600

Problem 4, continued:

Consider a buddy system that controls a total of 4 kB of memory and allocates a minimum of 256 B. The table to the right shows the current state of the free lists. In these lists, memory chunks are listed as offsets from the start of the 4 kB block, and the size of each chunk is 2^{order} B. For example, the second row shows the presence of three free memory chunks of size 2^9 = 512 B each at offsets 0x600, 0xE00, and 0x200.

Order	Offset 12
8	0x900, 0x100, 0xC00
9	0x600, 0xE00, 0x200
10	empty
11	empty
12	empty

10000 0000 2^8=256B

ASSUME THAT THE FOLLOWING OPERATIONS AND QUESTIONS OCCUR SEQUENTIALLY.

Part D (4 points):

Starting from the initial state above, the chunk of order 8 at offset 0x800 is freed. You may insert newly freed chunks on either side of the correct free list (but not both, if more than one must be inserted). Fill in the table to the right with the state of the free lists after the free operation completes.

Order	Offset
8	0x100, 0xC00
9	0x800, 0x600, 0xE00, 0x200
10	empty
11	empty
12	empty

Part E (2 points): After completing **Part D**, a request to allocate a chunk of order 10 fails, even though a contiguous region of the appropriate size is free. **USING NO 20 OR FEWER WORDS**, explain why the buddy system cannot satisfy the request.

Beause the contiguous region of appropiate is not in a single 2^{10} block.

only buddies can be merged, 0x600's buddy is 0x400, and 0x800's is 0xA00

Part F (4 points):

Starting from the state reached in **Part D**, the chunk of order 8 at offset 0xD00 is freed. Insert any newly freed chunks on the same side as you chose in **Part D**. Fill in the table to the right with the state of the free lists after the free operation completes.

Order	Offset
8	0x100
9	0x800, 0x600, 0x200
10	0xC00
11	empty
12	empty

Part G (2 points): After completing **Part F**, what offset is returned by a request to allocate a chunk of order 10?

0xC00

Name: ______________________________

Problem 5 (18 points): User Signals

Part A (2 points): Each signal type has a predefined default action. Which of the below is **NOT** one of the predefined default actions? **CIRCLE EXACTLY ONE ANSWER.**

A. The process terminates

B. The process terminates and dumps core

C. The process stops until restarted by a SIGCONT signal

D. The process ignores the signal

E. The process is rescheduled with a smaller time quantum

Part B (2 points): On signal generation, which of the following rules is **NOT** part of the signal generation permission check? **CIRCLE EXACTLY ONE ANSWER.**

A. sysadmin can always send a signal

B. process with same user id can always send a signal

C. process with same login session can send SIGABRT

D. process with same login session can send SIGCONT

Part C (3 points): Name **TWO SIMILARITIES** and **ONE DIFFERENCE** between signals and hardware interrupts:

asynchronous

Similarity: not queued or caught

can be

Similarity: has default action and can be changed by num

Difference: generated by software without any device

Problem 5, continued:

```
#include <stdio.h>
#include <signal.h>

int foo = 0;

void handler(int sig) {
     if (++foo < 5)
          printf("NO\n");
          alarm(1);
     else {
          printf("YES\n");
          exit(0);
     }
}

main() {
     signal(SIGALRM, handler);

     alarm(1);

     while (1) {
     }
}
```

1 2 3 4

Part D (6 points): Does the above code, when executed without additional user inputs, output anything to the terminal? Clearly **circle one** option.

YES NO

If you answered yes, what does it output?

1: NO

2: NO

3: NO

4: NO

5: YES

6: ____

Problem 5, continued:

Would any combination of keystrokes inputed to the terminal while the program is running modify the above output? Clearly **circle one** option.

YES NO

If you answered yes, what is the keystroke combination and what does it output? Note that if many outputs are possible, please select a single, nonempty output.

keystroke combination: crtl + C

1: No

2: No

3:

4:

5:

6:

Problem 5, continued:

```
#include <stdlib.h>
#include <stdio.h>
#include <signal.h>

void handler(int sig) {
     printf("YES\n");
     exit(0);
}

main() {
     signal(SIGINT, handler);
     while(1) {
     }
}
```

Part E (5 points): Does the above code, when executed without additional user inputs, output anything to the terminal? Clearly **circle one** option.

YES NO

If you answered yes, what does it output?

1: ______________________________

2: ______________________________

3: ______________________________

4: ______________________________

5: ______________________________

6: ______________________________

Problem 5, continued:

Would any combination of keystrokes inputed to the terminal while the program is running modify the above output? Clearly **circle one** option.

YES NO

If you answered yes, what is the keystroke combination and what does it output? Note that if many outputs are possible, please select a single, nonempty output.

keystroke combination: ctrl + C

1: YES

2:

3:

4:

5:

6:

Name: ______________________________

Problem 6 (13 points): Devices and Drivers

```
crw-------   1 aamir    staff            0,   0 Nov 25 18:14 console
br--r-----   1 andy     staff            1,  11 Dec  2 11:46 disk2s2
brw-r-----   1 fred     staff            1,   0 Nov 25 18:09 disk0
crw-r-----   1 harsh    staff            1,   3 Nov 25 18:09 rdisk1
crw-rw-rw-   1 yiran    staff          3 0,   0 Nov 25 18:09 fbt
crw-rw-rw-   1 root     wheel          2 2,   0 Dec  2 15:11 tty
crw-rw-rw-   1 root     wheel          1 4,   0 Nov 25 18:09 ttyp0
```

Ben recently found out that in Linux there are special files, device files, typically stored in /dev, that allow you to interact with devices through standard file programs such as 'ls' and 'cat'. Above is the output of the 'ls' program run on the /dev directory showing these files on Ben's computer. While reading the 'ls' man page, Ben noted two items of particular interest. The first character in the above listing indicates the type of file (for example, block special file, character special file, directory, regular file). The comma separated pair immediately to the left of the date field indicates the major and minor device numbers.

Part A (1 points): Help Ben recognize, in the list above, which files represent *character devices*, and which of them represent *block devices*. Write the number of character devices and the number of block devices below.

number of character devices: 5

number of block devices: 2

Part B (2 points): Given that Ben knows the major and minor numbers of the devices, help Ben recognize how many *distinct* device drivers are used for the devices shown in the above list for his operating system. Write the number below.

number of device drivers: ~~4~~ 5

Part C (1 points): In order to test Ben's understanding, help Ben recognize how many *distinct* devices are shown in the above list for his operating system. Write the number below.

number of devices: 6

Problem 6, continued:

Part D (5 points): Armed with newfound confidence Ben tries to implement the Imail service, but, again, poor Ben forgot what Professor Lumetta said in class. Help Ben match the file operations *f_ops* with the Imail operations to which they correspond, just as we did in class.

```
static ssize_t Imail_read (struct file*, char*, size_t, loff_t*);
static ssize_t Imail_write (struct file*, const char*, size_t, loff_t*);
static unsigned int Imail_poll (struct file*, struct poll_table_struct*);
static int Imail_ioctl (struct inode*, struct file*, unsigned int, unsigned long);
static int Imail_release (struct inode*, struct file*);
static int Imail_fsync (struct file*, struct dentry*, int datasync);
```

Using the above functions fill in the table below. You need not use every function, and functions can be used more than once. Two examples have been filled in for you already.

Wait for a message	Imail_poll
Read a message	Imail_read
Write a message	Imail_write
Send message after writing	Imail_fsync
Authenticate	Imail_ioctl
Add a new user	Imail_ioctl
Delete a user	Imail_ioctl

Part E (4 points): Ben Bitdiddle is trying to add a new feature to Imail in which users can collaboratively edit messages before sending. In Ben's implementation a shared message starts as a normal message, created by one user, and is shared by having other users point their *writing* message pointer to this shared message. The shared message is owned by the user that created it; in other words, access is controlled by the creating user's *user data semaphore*. In answering the followoing questions, assume that Ben made no changes to the locking structure of Imail.

Which lock(s) will User B need to hold in order to modify a shared message draft created by User A?

User A user data semaphore

and User B

Will this design work in the current implementation of I-mail? Why or why not?
Explain **USING 30 OR FEWER WORDS**.

No. The other user try to access shared message will try to hold two user data semphore, which could cause dead lock.