CS-537: Final Exam (Spring 2011)
The "Black Box"

Please Read All Questions Carefully!

There are thirteen (13) total numbered pages, with eight (8) questions.

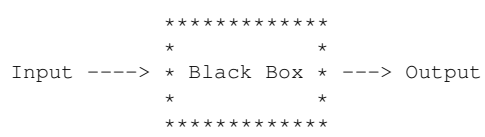
Name: _____

Grading Page

	Points
Q1	
Q2	
Q3	
Q4	
Q5	
Q6	
Q7	
Q8	
Total	

THE BLACK BOX

With any system, one can treat it as a "black box" in order to study its properties and behaviors. The box has input, output, and of course does something (which we may or may not know a lot about). Here is a depiction of such a black-box system.



In this exam, we're going to treat pieces of the the operating system as black boxes, to see if we can learn some things about them.

Each question will specify some aspects of what you can observe; your job is to fill in the rest. For example, you might be given a set of inputs and outputs, and be told to discover how the system works; alternately, you might be given the inputs and, given some knowledge of how the system works, be asked to describe the outputs; you may even be given the outputs, and asked to explain which inputs led to them.

As always, read each question carefully.

Good luck!

PROBLEM 1: VIRTUAL MEMORY AS A BOX

In this question, we'll examine a string of memory references to virtual pages (inputs), and use detailed system counters (outputs) to tell us what happened on such references, i.e., whether a particular memory reference was a TLB hit, TLB miss, or a page fault.

(a) Here is the first trace, from some system:

```
Input                    Output
----------------------   ------------
load to virtual page 0   Page fault
load to virtual page 1   Page fault
load to virtual page 2   Page fault
load to virtual page 3   Page fault
load to virtual page 0   TLB miss
load to virtual page 1   TLB miss
load to virtual page 2   TLB miss
load to virtual page 3   TLB miss
```

Assuming a TLB with LRU-based replacement, what can we conclude is the size of the TLB? (i.e., the number of entries it holds?)

(b) On a different system, we see this trace:

```
Input                    Output
----------------------   ------------
load to virtual page 0   Page fault
load to virtual page 1   Page fault
load to virtual page 2   Page fault
load to virtual page 3   Page fault
load to virtual page 0   TLB hit
load to virtual page 1   TLB hit
load to virtual page 2   TLB hit
load to virtual page 3   TLB hit
```

Again assuming a TLB with LRU-based replacement, what can we conclude about the size of the TLB?

(c) Finally, on some third system we see the following stream:

```
Input                    Output
----------------------   ------------
load to virtual page 0   Page fault
load to virtual page 1   Page fault
load to virtual page 2   Page fault
load to virtual page 3   Page fault
load to virtual page 0   TLB hit
load to virtual page 1   TLB hit
load to virtual page 2   Page fault
load to virtual page 3   TLB miss
```

Assume the TLB can hold two entries, and the entire memory can hold three pages. What can we conclude about the replacement policy used by the OS?

PROBLEM 2: CONCURRENCY IS THE BOX

A multi-threaded program can be viewed as one of our "black boxes". Here, however, you are given the initial state (the "inputs") of the system, and can peer into the "box" a little to see what code is in there. Your task is to figure out the possible outputs of the code.

Here is the initial state of some variables in the first program we examine:

```
int g = 10;
int *f = NULL;
```

Here are the code snippets for each of two threads:

```
Thread 1                                    Thread 2
-----------                                 ------------
lock(m);                                    lock(m);
f = &g;                                     f = NULL;
... // do some other stuff                  unlock(m);
printf("%d\n", *f);
unlock(m);
```

(a) What are all the possible outputs of this system, given an arbitrary interleaving of Threads 1 and 2?

The code gets rewritten as follows, to make the lock more "fine grained" by moving the "other stuff" out of the critical section:

```
Thread 1                                    Thread 2
-----------                                 ------------
lock(m);                                    lock(m);
f = &g;                                     f = NULL;
unlock(m);                                  unlock(m);
... // do some other stuff
lock(m);
printf("%d\n", *f);
unlock(m);
```

(b) What are all the possible outputs of the system now?

(c) You've been taught that locks around critical sections prevent "bad things" from happening. Is that true in part (b)? If so, why? If not, why not?

Let's examine another program, again with two threads:

```
Thread 1                                    Thread 2
-----------                                 ------------
pending = 1;                                pending = 0;
while (pending) {
  printf("hello\n");
}
```

(d) What are the possible outputs of this system, given an arbitrary interleaving of Threads 1 and 2?

(e) How could we re-write the code such that Thread 2 would only run after "hello" has been printed at least one time?

PROBLEM 3: DISK SCHEDULER FUN

This question examines a disk's internal scheduler as a black box, to see if we can learn anything about its behavior. The inputs we give to the disk are a bunch of requests; the outputs we observe are the order the requests are serviced.

For example, we might issue requests to blocks 0, 100, and 200 at the same time. The disk might decide to service 100, then 200, and then 0, depending on its internal scheduling algorithm.

We also know some details of the disk. It has one surface with 100 tracks, each of which have 100 sectors. The outer track (track=0) contains sectors 0...99, the next track (track=1) contains sectors 100...199, and so forth.

In this question, each question gives a workload to a particular disk; you are then asked to determine what you think the scheduler of that disk is doing by servicing requests in that order. In other words, what can you say about the algorithm in use? What about the initial state of the disk, before the requests arrived?

(a) Disk Model A

```
Requests           : 0, 1, 2, 3, 4, 5
Completed in order : 0, 1, 2, 3, 4, 5
```

What do you think Disk A's scheduler is doing? What might the initial state of the disk have been?

(b) Disk Model B

```
Requests           : 0, 500, 200, 400, 300, 100
Completed in order : 0, 100, 200, 300, 400, 500
```

What do you think Disk B's scheduler is doing? What might the initial state of the disk have been?

(c) Disk Model C

```
Requests           : 0, 500, 200, 400, 300, 100
Completed in order : 200, 300, 400, 500, 100, 0
```

What do you think Disk C's scheduler is doing? What might the initial state of the disk have been?

(d) Disk Model D

```
Requests           : 0, 50, 110, 600
Completed in order : 0, 110, 50, 600
```

What do you think Disk D's scheduler is doing? What might the initial state of the disk have been?

(e) Could these disks (A through D) actually be using the same scheduler? If so, why? If not, why not?

PROBLEM 4: UNVEILING THE RAID BOX

RAID disk systems can also be treated as black boxes. In this question, you can see the inputs to a RAID (i.e., a read() or write() request for a particular block) and observe the low-level outputs of the RAID (i.e., which disk or disks it reads from or writes to in order to service the request).

The requests that come into the RAID are either read(blk) or write(blk), which read or write the given block "blk" respectively. The outputs are in the form of low-level block-read (bread) or block-write (bwrite) commands, which read directly from or write directly to a specific disk.

Your task, for each question, is to observe the high-level read/write operations, as well as their underlying block reads/writes to one or more disks, in order to determine what type of RAID the system is.

Possible answers are: RAID-0 (striping), RAID-1 (mirroring), RAID-4 (parity), RAID-5 (rotating parity), or in fact more than one of these (e.g., RAID-0 or RAID-1).

All systems have 4 disks.

(a) What type of RAID is this?

```
write(0) -> [ bwrite(disk=0, block=0), bwrite(disk=1, block=0) ] in parallel
write(1) -> [ bwrite(disk=2, block=0), bwrite(disk=3, block=0) ] in parallel
write(2) -> [ bwrite(disk=0, block=1), bwrite(disk=1, block=1) ] in parallel
write(3) -> [ bwrite(disk=2, block=1), bwrite(disk=3, block=1) ] in parallel
```

(b) What type of RAID is this?

```
read(0)  -> [ bread(disk=0, block=0) ]
```

(c) What type of RAID is this?

```
write(0) -> [  bread(disk=0, block=0),  bread(disk=3, block=0) ] in parallel,
            followed by
            [ bwrite(disk=0, block=0), bwrite(disk=3, block=0) ] in parallel
write(3) -> [  bread(disk=0, block=1),  bread(disk=3, block=1) ] in parallel,
            followed by
            [ bwrite(disk=0, block=1), bwrite(disk=3, block=1) ] in parallel
```

(d) What type of RAID is this?

```
read(0) -> [ bread(disk=0, block=0) ]
read(1) -> [ bread(disk=1, block=0) ]
read(2) -> [ bread(disk=0, block=0), bread(disk=1, block=0), bread(disk=3, block=0) ] in parallel
read(3) -> [ bread(disk=0, block=1) ]
read(4) -> [ bread(disk=1, block=1) ]
read(5) -> [ bread(disk=0, block=1), bread(disk=1, block=1), bread(disk=3, block=1) ] in parallel
```

PROBLEM 5: INPUTS TO THE BLACK BOX FS

The INITIAL STATE, or state(i), of a very simple file system is shown:

```
Inode Bitmap : 10000000
Inode Table  : [size=1,ptr=0,type=d] [] [] [] [] [] [] []
Data Bitmap  : 10000000
Data         : [("." 0), (".." 0)] [] [] [] [] [] [] []
```

There are only eight inodes and eight data blocks; each of these is managed by a corresponding bitmap. The inode table shows the contents of each of eight inodes, with an individual inode enclosed between square brackets; in the initial state above, only inode 0 is in use. When an inode is used, its size and pointer field are updated accordingly (in this question, files can only be one block in size; hence a single inode pointer); when an inode is free, it is marked with a pair of empty brackets like these "[]". Note there are only two file types: directories (type=d) and regular files (type=r). Data blocks are either "in use" and filled with something, or "free" and marked accordingly with "[]". Directory contents are shown in data blocks as comma-separated lists of tuples like: ("name", inode number). The root inode number is zero.

Your task in this black-box experiment: figure out what file system operation(s) must have taken place in order to transition the file system from some INITIAL STATE to some FINAL STATE. You can describe the operations with words (e.g., file "/x" was created, file "/y" was written to, etc.) or with the actual system calls (e.g., create(), write(), etc.).

Note that these questions build as they go, i.e., the file system starts in the initial state "state(i)" (as above), and then goes through states(a), (b), (c), (d), (e), and finally (f) in the questions below. Thus, your task is to figure out what operation(s) caused these changes as the question progresses.

(a) INITIAL STATE: state(i) as above to FINAL STATE (a):

```
Inode Bitmap : 11000000
Inode Table  : [size=1,ptr=0,type=d] [size=0,ptr=-,type=r] [] [] [] [] [] []
Data Bitmap  : 10000000
Data         : [("." 0),("..." 0),("f" 1)] [] [] [] [] [] [] []
```

Operation that caused this change?

(b) INITIAL STATE: state(a) as in part (a) of this question to FINAL STATE (b):

```
Inode Bitmap : 11000000
Inode Table  : [size=1,ptr=0,type=d] [size=1,ptr=7,type=r] [] [] [] [] [] []
Data Bitmap  : 10000001
Data         : [("." 0),("..." 0),("f" 1)] [] [] [] [] [] [] [SOMEDATA]
```

Operation that caused this change?

(c) INITIAL STATE: state(b) to FINAL STATE (c):

```
Inode Bitmap : 11000000
Inode Table  : [size=1,ptr=0,type=d] [size=1,ptr=7,type=r] [] [] [] [] [] []
Data Bitmap  : 10000001
Data         : [("." 0),("..." 0),("f" 1),("b" 1)] [] [] [] [] [] [] [SOMEDATA]
```

Operation that caused this change?

(d) INITIAL STATE: state(c) to FINAL STATE (d):

```
Inode Bitmap : 11000000
Inode Table  : [size=1,ptr=0,type=d] [size=1,ptr=7,type=r] [] [] [] [] [] []
Data Bitmap  : 10000001
Data         : [("." 0),(".." 0),("b" 1)] [] [] [] [] [] [] [SOMEDATA]
```

Operation that caused this change?

(e) INITIAL STATE: state(d) to FINAL STATE (e):

```
Inode Bitmap : 11000000
Inode Table  : [size=1,ptr=0,type=d] [size=1,ptr=7,type=r] [] [] [] [] [] []
Data Bitmap  : 10000001
Data         : [("." 0),(".." 0),("b" 1)] [] [] [] [] [] [] [OTHERDATA]
```

Operation that caused this change?

(f) INITIAL STATE: state(e) to FINAL STATE (f):

```
Inode Bitmap : 10000000
Inode Table  : [size=1,ptr=0,type=d] [] [] [] [] [] [] []
Data Bitmap  : 10000000
Data         : [("." 0),(".." 0)] [] [] [] [] [] [] []
```

Operation that caused this change?

PROBLEM 6: CRASH CONSISTENCY AS A BLACK BOX

File systems crash. One sad outcome of a crash is that the file system is left in an inconsistent state. In this question, you'll be given the final output (on-disk state) of a file system, and try to determine if an inconsistency has arisen. If one has, write "inconsistent" and suggest a fix! Otherwise, write down that the file system is "consistent".

We use the same basic notation as in the previous file system question (Q5); refer to that question for details.

(a) FILE SYSTEM STATE: Consistent or inconsistent? If inconsistent, how to fix?

```
Inode Bitmap : 11111111
Inode Table  : [size=1,ptr=0,type=d] [] [] [] [] [] [] []
Data Bitmap  : 10000000
Data         : [("." 0),(".." 0)] [] [] [] [] [] [] []
```

(b) FILE SYSTEM STATE: Consistent or inconsistent? If inconsistent, how to fix?

```
Inode Bitmap : 11100000
Inode Table  : [size=1,ptr=0,type=d] [size=1,ptr=1,type=r] [size=1,ptr=2,type=r] [] [] [] [] []
Data Bitmap  : 10000000
Data         : [("." 0),(".." 0),("b" 1),("c" 2)] [DATA] [DATA] [] [] [] [] []
```

(c) FILE SYSTEM STATE: Consistent or inconsistent? If inconsistent, how to fix?

```
Inode Bitmap : 10000000
Inode Table  : [size=1,ptr=0,type=d] [] [] [] [] [] [] []
Data Bitmap  : 11110000
Data         : [("." 0),(".." 0)] [] [] [] [] [] [] []
```

(d) FILE SYSTEM STATE: Consistent or inconsistent? If inconsistent, how to fix?

```
Inode Bitmap : 11000000
Inode Table  : [size=1,ptr=0,type=d] [size=1,ptr=1,type=d] [] [] [] [] [] []
Data Bitmap  : 11000000
Data         : [("." 0),(".." 0),("a" 1)] [("." 1),(".." 1)] [] [] [] [] [] []
```

(e) FILE SYSTEM STATE: Consistent or inconsistent? If inconsistent, how to fix?

```
Inode Bitmap : 11000000
Inode Table  : [size=1,ptr=0,type=d] [size=1,ptr=1,type=r] [] [] [] [] [] []
Data Bitmap  : 11000000
Data         : [("." 0),(".." 0),("a" 1)] [("." 1),(".." 1)] [] [] [] [] [] []
```

(f) FILE SYSTEM STATE: Consistent or inconsistent? If inconsistent, how to fix?

```
Inode Bitmap : 11100000
Inode Table  : [size=1,ptr=0,type=d] [size=1,ptr=1,type=r] [size=1,ptr=2,type=r] [] [] [] [] []
Data Bitmap  : 11100000
Data         : [("." 0),(".." 0)] [DATA] [DATA] [] [] [] [] []
```

PROBLEM 7: THE LOG-STRUCTURED BLACK BOX

The log-structured file system buffers updates in memory (in segments) and then writes them out to disk sequentially. In this question, you'll be able to watch the traffic stream of writes to disk performed by LFS. Your task is to figure out what the inputs to the system (i.e., the system call(s) that took place) that caused these writes to happen.

Before these writes, you can assume the file system was basically empty (save for a root directory). You can also assume that a single inode takes up an entire block (for simplicity). The LFS inode map is called the "imap" below and of course is also updated as needed.

(a) Segment written starting at disk address 100, in a segment of size 4:

```
block 100: [("." 0), (".." 0), ("foo" 1)]   // a data block
block 101: [size=1,ptr=100,type=d]          // an inode
block 102: [size=0,ptr=-,type=r]            // an inode
block 103: [imap: 0->101,1->102]            // a piece of the imap
```

What file system operation(s) led to this segment write?

(b) Segment written to disk address 104, in a segment of size 4:

```
block 104: [SOME DATA]                      // a data block
block 105: [SOME DATA]                      // a data block
block 106: [size=2,ptr=104,ptr=105,type=r]  // an inode
block 107: [imap: 0->101,1->106]            // a piece of the imap
```

What file system operation(s) led to this segment write?

(c) Segment written to disk address 108, in a segment of size 4:

```
block 108: [SOME DATA]                      // a data block
block 109: [SOME DATA]                      // a data block
block 110: [size=2,ptr=108,ptr=109,type=r]  // an inode
block 111: [imap: 0->101,1->110]
```

What file system operation(s) led to this segment write?

(d) Segment written to disk address 112, in a segment of size 4:

```
block 112: [SOME DATA]                      // a data block
block 113: [SOME DATA]                      // a data block
block 114: [size=4,ptr=108,ptr=109,ptr=112,ptr=113,type=r] //inode
block 115: [imap: 0->101,1->114]            // a piece of the imap
```

What file system operation(s) led to this segment write?

(e) After all of those writes, how much garbage was left on the disk? (i.e., write down which blocks are filled with garbage, if any)

PROBLEM 8: NFS BLACK BOXES

The NFS client-side file system is responsible for turning its inputs (system calls) into outputs (protocol requests). Assume you have the following protocol requests at your disposal:

```
LOOKUP(parent file handle, "name") -> returns "file handle" of "name" (or fails)
CREATE(parent file handle, "name") -> creates "name" in directory of parent,
                                      returns "file handle" of "name"
READ(file handle, offset, size) -> returns "size" bytes of file from "offset"
WRITE(file handle, data, offset, size) -> writes "size" bytes of "data"
                                          to file at "offset"
GETATTR(file handle) -> returns attributes of file
```

Assume the following sequence of system calls take place, in order from (a) down to (g); these are the inputs to the client-side NFS file system.

Your job is to write the outputs of the client in the form of protocol requests.

You can assume you know the root file handle to begin with, which you can call ROOT FILE HANDLE or something equally clever. Assume all requests are made to valid and existing files, and also assume nothing is in the client-side cache to begin with (but the cache may become populated as the system calls in (a) through (g) execute).

(a) fd = open("/foo/bar", O_RDONLY); // assume file is only 1 4KB block in size

(b) read(fd, buffer, 4096);

(c) close(fd)

(d) fd = open("/foo/bar2", O_WRONLY | O_CREAT);

(e) lseek(fd, SEEK_END, 0); // seeks to end of file

(f) write(fd, buffer, 4096);

(g) close(fd);