CS-537: Final (Spring 2012)
Your Life, FLASHing Before Your Eyes

Read All Questions Carefully! (Or At Least Read Them)

There are seventeen (17) total numbered pages. For this, I am truly sorry.

Please put your FULL NAME (mandatory) on THIS page only.

Name: ____

Grading Page

	Points	Total Possible
Q1		20
Q2		20
Q3		20
Q4		20
Q5		20
Q6		20
Q7		20
Total		140

"Pathetic earthlings. Hurling your bodies out into the void, without the slightest inkling of who or what is out here. If you had known anything about the true nature of the universe, anything at all, you would've hidden from it in terror."
– Emperor Ming (from "Flash Gordon", probably talking about final exams)

A new technology is sweeping the storage-systems landscape. It is called *flash* and is the current best example of a more general trend towards *solid state* storage, in which electronics without any moving parts are used to persistently store data. The contrast, of course, is the more traditional storage approach of using hard-disk drives.

In this exam, your goal is simple: to understand the impact of flash on all of the file and storage systems we learned about this semester. To do so, of course, you'll need to know a little more about flash and how it works, so read the next few paragraphs very carefully!

A basic flash chip is composed of a large number of **blocks**; we'll assume in this exam that blocks are each 128 KB in size. Each block is further sub-divided into **pages**, each of which is 4 KB. Thus, a 1-MB flash chip consists of 8 blocks (128 KB per block $\times$ 8 blocks $=$ 1 MB); each block consists of 32 pages; there are thus 256 pages in a 1-MB chip (8 blocks $\times$ 32 pages per block). Here is a (bad) diagram of such an arrangement:

```
|            Block 0              |            Block 1              | ...
| Page0 | Page1 | ... | Page 31 | Page32 | Page33 | ... | Page 63 | ...
```

We'll now see why we differentiate between blocks and pages. As it turns out, each flash chip exposes three operations. The simplest is a **read(page)**, which reads a specific page and returns the data therein; this is quite similar to devices with which you are already familiar. Writing a flash is more complicated, however, and requires the use of two operations in tandem. Specifically, one must first call **erase(block)** to erase an entire block of the flash; erasing resets the contents of the entire block (thus losing all the data in the block!). Only after an erase can one then use the **program(page)** interface to write new data (one page at a time) to the block.

The costs of these operations is also uneven. Reads are quite cheap, taking on the order of **10 microseconds to read a page**. Program operations are slightly more expensive, around **50 microseconds to program a page**. Erase operations, however, are terribly expensive, and take **1 millisecond (1000 microseconds) to complete.**

That's it! Read each question carefully, and don't forget to **SHOW YOUR WORK!**

One more piece of advice: **DO PROBLEM #1 FIRST**, it will make your life (somewhat) easier.

Thanks for a fun semester (really!). And, sorry about the long exam.

1. **Basic Flash Performance.**

   (a) Let's start with an easy calculation. Given the performance characteristics described above, how long does it take to perform 32 reads, of size 4 KB, to random locations on a flash device?

   (b) Now let's focus on writes. How long will it take to perform 32 writes, of size 4 KB, to different random locations on the flash device? Assume there is no live data anywhere on the device (and thus it is OK to erase a block without worry).

   (c) Now let's do 32 random 4-KB writes to different random locations on the device, but with a key difference: the rest of the data on the device (and thus each block) is live. Thus, you can't just erase a block and then program a page within it, because that would lose data; instead, you need to perform a read-modify-write of any block you are updating. How long will these 32 writes take?

   (d) Let's do some sequential reads. How long does it take to read 32 consecutive 4-KB chunks from the flash? (assume the first read is aligned with the underlying flash block; i.e., all 32 4-KB reads will read from a single flash 128-KB block)

(e) Our next focus is on sequential writes. How long does it take to write 32 consecutive 4-KB chunks to the flash? (assume the first write is aligned with the underlying flash block; i.e., all 32 4-KB writes will write to a single flash 128-KB block)

(f) Finally, how long does it take to write 32 consecutive 4-KB chunks to the flash, in the worst case where you can no longer make any assumptions about alignment?

2. **Hard Disk Drives.** Let's now do the same set of calculations for a hard drive. Assume, for this question, the disk we are using has an average seek time of 4 milliseconds; assume further an RPM (rotations per minute) of 15,000. Finally, assume a transfer rate of 100 MB/sec.

   (a) Let's start with an easy calculation. Given the performance characteristics described above, how long does it take to perform 32 reads, of size 4 KB, to random locations on the hard disk?

   (b) Now let's focus on writes. How long will it take to perform 32 writes, of size 4 KB, to random locations on the disk?

   (c) Let's do some sequential reads. How long does it take to read 32 consecutive 4-KB chunks from the disk?

(d) Our next focus is on sequential writes. How long does it take to write 32 consecutive 4-KB chunks to the disk?

(e) Given what you know from the previous question (about flash) and this question (about hard drives), when should you use flash? When should you use hard drives?

3. **File System Basics.** In this question, we now examine file system performance on flash devices and hard drives. Assume something like the very simple file system we first studied in class, with a single super block, a single data bitmap block (DB), a single inode bitmap block (IB), a series of inode blocks (I), and a series of data blocks (D); all blocks are 4 KB. Assume, for simplicity, that each inode is of size 4 KB (i.e., each inode takes up one full 4-KB block on disk, which is outrageous but makes your life easier).

   (a) Assume we are reading two files in the root directory, "foo" and "bar", each of which is 4 KB in size. In reading these two files in their entirety, which blocks will the file system read from disk? (assume that no file systems blocks save the superblock are in memory to begin with; assume also that blocks, once read, stay in the memory cache; assume there is no write traffic of any kind; finally, assume that the root directory only has these two files in it)

   (b) We are interested in how long this sequence of reads will take. Let's first assume we are running it on a **hard drive**, as described earlier (average seek of 4 ms, rotation of 15000 RPM, transfer rate of 100 MB/sec). Assuming we first read "foo", and then read "bar", how long will it take to read both files from disk?

   (c) Now assume we are reading the files (first "foo", then "bar") from a **flash** device, as described earlier (10 microseconds to read a page, 50 microseconds to program a page, and 1 ms to erase a block). How long does it take to read "foo" then "bar" from the flash?

(d) Now assume we are reading the files from a smarter file system, such as the Fast File System (FFS). How much does performance change when reading “foo” then “bar” from a disk, using FFS instead of a simple file system? Assume the same disk (4 ms average seek, 15000 RPM rotation, 100 MB/sec transfer) as before. Estimate costs if you need to.

(e) Finally, assume we are running FFS on top of a flash. How much does performance change when reading “foo” then “bar” from a flash, using FFS instead of a simple file system? Is FFS needed on this device?

4. **Journaling File Systems.** We now turn our attention to journaling file systems, such as Linux ext3. Such file systems use a small "journal" (or "write-ahead log") to record information about pending file system updates before committing said updates, in order to be able to recover quickly from a crash.

   (a) Before getting into journaling, let's review what happens when we're creating a file on disk. Assuming a basic file system structure (such as the very simple file system with bitmaps for inode and data block allocation, inodes, and data blocks), what blocks are written to disk during the creation of a 4-KB file in the root directory? (assume there is no journaling)

   (b) Assuming there is now **data journaling**, in which all blocks are logged; what exact sequence of writes takes place to the underlying storage device during the file creation described above?

   (c) Now assume this traffic is directed to a hard disk, with the same basic parameters (4 ms average seek, 15000 RPM, 100 MB/s transfer). How long does it take to complete all of the writes under the file creation? (to both the journal and the regular file system)

(d) Now assume the write traffic is directed to a flash device, as described above (50 microseconds to program a page and 1000 microseconds to erase a block). Making any assumptions you need to about the locations of various file system structures; how long will the file creation on this journaling file system take on a flash device?

(e) Finally, let us now think about the nature of journaling and the problem it solves. Given that writing to the device at an arbitrary location requires a read-modify-write cycle (as described in Problem 1), is your journaling strategy safe?

5. **Network File System (NFS).** Sun's Network File System (NFS) makes server crash recovery simple by handling failures in a uniform and simple way. In this question, we'll explore how a server side flash cache affects NFS performance.

   (a) First, let's understand how NFS basically works. When a client issues a request and fails to get a response, it simply waits a while and tries again. What are the different cases where the client won't get a reply back from the server?

   (b) What property of each protocol request is needed for it to be OK to keep retrying a request?

   (c) Sometimes not all requests in NFS have the property described in the question above. For example, consider `NFS_Create(pfid, filename)`, which creates the file `filename` in the directory referred to by `pfid`, if the file doesn't already exist. If it does exist, the request returns an error. What odd thing can happen upon retry of this request?

(d) Caching plays a major role in achieving reasonable NFS performance. On the client side, writes can be buffered for a while before flushing them to the server upon close (i.e., when the file is closed, the client file system will flush all of its dirty blocks to the server). Can a block be written to the server *before* the file is closed? Describe.

(e) Now finally we get to the server side. Each `NFS_Write()` request must be written to stable storage (e.g., a disk) before replying to the client. Why is that?

(f) Because each request must be written to stable storage before replying, some companies put a flash-based cache in the server to absorb those writes. Estimate what the possible performance impact of such a cache is on write performance, making whatever assumptions you need.

6. **A RAID of Flash Devices.** RAID is used on hard-drive based systems to tolerate the loss of one drive (and with some advanced schemes, more). In this question, we examine how traditional RAIDs work, and how to adapt them to work on a flash-based medium.

   For most of this problem, we'll focus on RAID-4 with 4 data disks and 1 parity disk. The chunk size of this RAID is 128 KB; this means that in a stripe, the first 128 KB of data are placed on disk 0, the next 128 KB on disk 1, and so on. Assume further that all reads and writes to the RAID (from a file system above) occur in multiples of 128 KB.

   Assume again the usual flash characteristics: 10 microsecond read, 50 microsecond program, 1000 microsecond erase.

   NOTE: Assume that everything is "aligned" to make reads and writes take as little time as possible.

   (a) How long does a large aligned read of 512 KB take on our flash-based RAID?

   (b) How long does a large aligned write of 512 KB take?

   (c) How long does a single aligned 128-KB read take?

(d) How long does a single aligned 128-KB write take?

(e) How long do two random aligned 128-KB writes take? Is there any way for such random "small" writes to occur in parallel on RAID-4? Explain.

(f) Finally, when a device "fails" in a RAID, its data can be *reconstructed* from the other devices. How long does a random read of a 128-KB block that is on a failed device take?

7. **Log-structuring.** Log-structuring is a concept we saw in the discussion of the log-structured file system. In this problem, we apply the idea of log-structuring to build a better flash-based device.

   The basic idea is simple, and starts by organizing the flash device into a log. When a file system writes a 4-KB page to location X, the device does not write it to that location; rather, the device instead writes it to the end of its log (at location Y), and then records the mapping (from X to Y) in an in-memory mapping table known as the **flash translation layer** or **FTL**.

   For example, imagine that the file system wrote three 4-KB blocks to locations 1000, 500, and 4050, respectively. Assuming these were the first writes to the flash, the flash would then look like this:

```
| Page=0 holds 1000 | Page=1 holds 500 | Page=2 holds 4050 | ...
```

   The FTL would then contain the following “file-system block” to “flash page” mappings:
   (1000 $\rightarrow$ 0), (500 $\rightarrow$ 1), and (4050 $\rightarrow$ 2).

   One main reason for this log-structuring is performance. Now, when writing to a flash device, erases take place only once per (128-KB) block, when the flash log crosses from one block to the next. Further, there is no read for a read-modify-write cycle; in the common case, the block that is about to be erased is empty to begin with.

   (a) Given such a design, how much mapping information is needed in the FTL to map a 1 gigabyte (GB) flash device?

   (b) Given such an FTL design, how long would a file creation on a file system such as FFS take? (Ignore reads, and assume all writes fit within a single 128-KB flash block)

   (c) How long would a file creation take on this device, now using a journaling file system?

(d) In all of these cases, what is the impact of this FTL design on read performance?

(e) One final problem that arises in log-based FTLs is one of **garbage collection**, similar to the same problem in LFS. Describe a simple example in which garbage is created, and how the flash device might go about fixing this problem.