CS-537: Midterm Exam (Fall 2004)

Exam III: Revenge of the Sith

Please Read All Questions Carefully!

There are eight (7) total numbered pages.

Please put your Name and student ID on this page and your student ID (but NOT YOUR NAME) on every other page.

Name and Student ID: _____

Grading Page

	Points	Total Possible
Part II: Long Answers		$(2 \times 20) \rightarrow 100$
Total		100

Part I: Long Questions

This exam consists solely of five long questions, **each worth 20 points (total 100)**.

1. **Right on schedule.**

   We now explore disk scheduling.

   (a) A common disk scheduling algorithm is called "shortest seek time first" (SSTF). Describe SSTF, and what its primary problem is.

   (b) You develop a new algorithm, called **two-queue SSTF**. The idea is simple: you put the first N requests into the first of two disk scheduling queues, and start by servicing those requests in SSTF-based order. However, as the next so many requests arrive into the system, they are put into a different "second" queue. No request in this other queue is serviced until all of the requests in the first queue have been. At this point, the second queue starts getting serviced (again in SSTF order), and requests that arrive during this phase are put into the first queue (and not serviced until all requests in the second queue have been). And so forth. How does this two-queue policy compare to SSTF?

   (c) Most disks have a small cache called a "track" buffer inside of them. Assume you know the contents of the cache, e.g., which blocks when you request them will be serviced from the cache, versus which really have to go to disk. How would you modify SSTF in order to take into account this knowledge so as to deliver the best possible performance?

   (d) In a mirrored RAID disk system, when reading data, you have the choice of which of two disks to read the data from. How would you incorporate this decision into your SSTF-based disk scheduler? (i.e., how would you change SSTF if it was running on top of two disks with identical contents, particularly when considering reads from disk?)

2. **DegRAIDing your disk system.**

   RAID systems typically have a *hot spare*. When a disk in the RAID fails, the hot spare (which was previously unused) is activated, and takes the place of the broken disk. In this question, you will answer some questions about this process, known as *reconstruction*.

   For all parts of this question, assume **each disk has D blocks**, and that the **time to read each block is T milliseconds**. Also assume that the only thing that really takes any substantial time is disk I/O – all other costs are irrelevant.

   (a) Assume we have a mirrored (RAID-1) disk system. One disk fails. How many blocks must be read and written in order to fully reconstruct the RAID?

   (b) How long does the RAID-1 reconstruction take? (assume the best possible implementation you can think of)

   (c) Now assume we have a parity-based RAID-4 system. One disk fails (assume it is not the parity disk). How many blocks must be read and how many written to fully reconstruct the RAID?

   (d) How long does the RAID-4 reconstruction take? (again, assume the best possible implementation you can think of)

3. **Do I-no-de answer?**

In this question, we consider a non-standard Unix file system, which instead of using inodes, instead stores most information about a file in the directory entry for that file. We call our new system the *Inode-Free File System*, or *IFFS*.

(a) Of the following, which are usually found in a standard Unix inode? **Circle all that apply:**

i. Direct pointers to data blocks
ii. The name of the file
iii. Some statistics about when the file has been accessed, updated, etc.
iv. The inode number of the file's parent directory
v. The current position of the file pointer

(b) In a standard Unix file system, how many **disk reads** would it take to read a single block from the file */this/path/is/toolong* from disk? (you should assume nothing is cached, i.e., everything starts on disk)

(c) Now compare this to the number of reads it would take to read a single block from the same file */this/path/is/toolong* in IFFS. (again assume nothing is cached, i.e., everything starts on disk)

(d) IFFS doesn't support **hard links**. Why do you think so? (explain)

4. **A Faster File System.**

The Berkeley Fast File System was pioneering in its understanding that a file system must be tuned to the storage system underneath of it. However, it does make some assumptions about how files are accessed in order to improve performance. In this question, we explore the limits of some of those assumptions.

(a) FFS uses **cylinder groups** in order to group "related" items on disk. What related data items does FFS try to place in a cylinder group?

(b) Assume we have a workload that accesses files across two different directories repeatedly, e.g., the workload reads /dir1/foo1, then /dir2/foo1, then /dir1/foo6, then /dir2/foo15, etc. Why does FFS not perform particularly well for this workload?

(c) Assume that those two directories in the previous example, /dir1 and /dir2, were created at the nearly the same time. Design a modified FFS allocation policy that would do better than the FFS standard policy.

(d) Now describe a workload that does better under standard FFS than it does with your new file allocation policy (or demonstrate that your approach is strictly better).

5. **Consistency: The Hobgoblin of Small Minds.**

   In this question, we explore issues of on-disk consistency.

   (a) You create a new file called **foo** that has a single data block in a directory **dir**. Assuming an FFS-based file system, describe all the changes to the on-disk structures of the file system, i.e., which blocks get written to disk because of this new file creation?

   - *inode bitmap (allocating the inode for* **foo***)*
   - *data bitmap (allocating a block for* **foo***)*
   - *inode block (containing* **foo's** *new inode)*
   - *data block (the new data in* **foo***)*
   - *data block (adding new directory entry in* **dir** *for foo)*
   - *inode block (changing the update time and size of* **dir's** *inode)*

   Note that this assumes that 'dir' did not increase substantially (i.e., 'dir' did not require a new block to add the new entry for 'foo'.

   (b) The on-disk state of a file system can be corrupted with an untimely crash. Show **one example write ordering**, with a crash labeled at an untimely point, that leads to an **inconsistent** on-disk state for the meta-data of the file system. For example, if you are writing blocks A, B, and C to disk (in that order), you might write down A, crash, B, C to indicate that a crash after A was written leads to an inconsistency.

   Many possible answers here. Recall that an inconsistency in the meta-data means that two (or more) separate pieces of meta-data don't agree as to the on-disk state of the file system.

   An example: write the **inode for foo** *(which points to the data block of foo), but then* **crash** *before writing the* **data bitmap for foo**. *In this case, the bitmap indicates the block is free, but the inode indicates that the block is allocated. Hence, an inconsistency.*

   (c) Now, show **one example write ordering** where the file system meta-data is perfectly consistent, but where a crash leads the file to have the wrong contents.

   The file can end up with bad contents when an inode is written and points to a block that has not yet been written.

   Hence, the ordering: write **inode for foo** *and then* **crash** *before the* **data block for foo** *is written leads to this type of "data inconsistency".*

   (d) Finally, **journaling** can be used to solve some of these problems. Describe how data and meta-data must be written to the journal and then to their final on-disk locations in order to avoid the two problems you demonstrated above.

   Journaling uses a write-ahead log to commit updates to disk atomically. Either all of the updates happen, or none – no inconsistent states are left.

   There are lots of different journaling modes (as discussed in class), but the simplest is to write everything to the journal first, and then to checkpoint it to the final on-disk location later.

   Hence, we would do the following:

   - *write a* **transaction start** *block to the journal*
   - *write a* **transaction descriptor** *block to the journal (this tells us what blocks are a part of the transaction)*
   - *write all the blocks of the transaction to the journal (all the blocks from the first part of this question)*
   - *wait for these to complete*
   - *write a* **transaction commit** *block to the journal*
   - *wait for this to complete*
   - *copy the blocks of the transaction (bitmaps, inodes, data blocks) from the journal to their final locations in the file system*
   - *wait for this to complete*
   - *mark the transaction as completed in the journal*

   Other answers are possible. Importantly, though, to avoid the possibility of "data inconsistency", we must make sure that if we are not using data journaling (as above), to write the data block to disk first before the inode that points to it.