CSE 451, Winter 1998, Homework 4

1. 10.6

   You can simulate a multilevel directory structure with a single level directory structure and arbitrarily long file names. To do this we will use a special character, say `#`, as the directory delimiter and we will specify a file as its path followed by the `#` symbol then the file name. A program then that wanted to use the concept of a *current working directory* could then just perform this concatenation of the path with the filename when a file is accessed. A file `perl` in directory `/usr/local/bin/` would have a file name of `#usr#local#bin#perl`

   Note that a directory has many other features such as access permissions directory data could be stored in a file that ends with the `#` symbol if necessary. File protection facilities could be added into the operating system as well, but they could not be enforced if this is to be implemented as a user level library. The directory `/usr/local/bin/` would have its directory data in a file called `#usr#local#bin#`.

   If we were to instead limit the file names to seven characters, we could still simulate directories but it is much more complicated. In fact we could simulate the low level inode/block mechanism. Each file in our system would be named with an identifier that could be looked up in another file to get matched with a name. This can be done in exactly the same way that directories are done with inodes in a typical file system.

2. 10.12

   Specifying file access to 4990 of 5000 users.

   (a) In UNIX it can be done one of two ways.

   First, you could make a group, `access`, add all 4990 users that you want to have access to this file in this group, and then `chmod` the file like this:

   ```
   -rw-rw----    jason    access         1266 Mar  4 00:22 sol4.html
   ```

   The second way you can do this in UNIX is more tricky. User group access is checked before global access is checked so you can create a group, `deny`, and `chmod` the file like this:

   ```
   -rw----rw-    jason    deny           1266 Mar  4 00:22 sol4.html
   ```

   (b) This seems pretty effective to me. If we could not do the second scheme in UNIX then I would suggest something like that.

3. 11.1

Given a file with 100 blocks, how many disk I/O operations do the following procedures require for the various allocation strategies. This solution assumes that in contiguous allocation the length of the file (in blocks) and the pointer to the first block is stored in memory. This assumes that the linked allocation method has a pointer to the first, a pointer to the last block, and the length of the file are stored in memory and each block only has a pointer to the next block and not a pointer to the previous block. It assumes that in indexed allocation the index is completely in memory.

We will also be ignoring the cost of manipulating the `free blocks` list. However, if we were to take this into account for each add, it would cost the same as a remove from the beginning with linked allocation method (1 operation) and each remove would be the same as the cost of adding to the beginning of the with the linked allocation (1 operation). This is assuming that the *free blocks* list is actually implemented as a linked list with pointers to the first block in memory.

(a) The block is added at the beginning.

contiguous allocation 201. Each block must be shifted over to the next block. This involved one read and one write per block. Then the new block must be written.

linked allocation 1. Just write the new block making it point to the next block. Update the *first block* pointer in memory.

indexed allocation 1. Just write the new block and update the *index* in memory.

(b) The block is added to the middle.

contiguous allocation 101. 50 blocks (the second half) must be shifted over one block and the new block must be written. As before, the shift takes one read operation and one write operation.

linked allocation 52. I must read 50 blocks to find the middle. Then I must write my new block somewhere with the *next block* pointing to the block after the 50th block. Then I must write the 50th block to point to this new block.

indexed allocation 1. Just write the new block and update the *index* in memory.

(c) The block is added at the end

contiguous allocation 1. Just write the new block.

linked allocation 3 (assuming that in order to modify the a block's *next block* pointer, I must first read the whole block in, then write the whole block out just with that pointer changed). First I read the last block as given by the *last block* pointer, then I write my new block, then I write the last block back modifying its *next block* pointer to point to my new block, then I update my *last block* pointer in memory.

indexed allocation 1. Just write the new block and update the *index* in memory.

(d) The block is removed from the beginning.

contiguous allocation 0. Just point the *first block* pointer to the second block.

linked allocation 1. Read in the first block to get the second block's pointer and then set the *first block* pointer to point to the second block.

indexed allocation 0. Just remove the block from the index in memory.

(e) The block is removed from the middle.

contiguous allocation 98. Assuming that we are removing the 51st block, we will have to move 49 blocks one block over this takes a read and a write operation each.

linked allocation 52. It takes 51 reads to find the pointer to the 52nd block, then we need to update the 50th block's *next block* pointer with this value.

indexed allocation 0. Just remove the block from the index in memory.

(f) The block is removed from the end.

contiguous allocation 0. Just update the *length* information stored in memory.

linked allocation 99 or 100. We need to update the *last block* pointer with the second to last block and the only way to get the second to last block is to read the preceding 99 blocks. Then we probably want to mark the 99th block (the new last block) as having a null pointer and this would give the 100th operation. We save the new last block in our *last block* pointer in memory.

indexed allocation 0. Just remove the block from the index in memory.

4. 11.2

(a) If a pointer to the free blocks list is lost and a backup copy is not stored on disk, then the free blocks list can be reconstructed by garbage collecting around the file system. All used blocks on a file system are linked to from the root directory. Anything not used is free.

(b) One scheme to insure that a pointer is never lost due to a memory failure is to keep the "valid" copy of that pointer on disk in a designated place. Note that a secondary copy of this pointer could be in memory. This pointer could still be lost or corrupted is the system crashes in the middle of an operation that manipulates the free blocks list. Typically when systems crash in the middle of a disk operation they spend a long time at boot time checking and restoring the file system to a consistent state.

If any of you run Linux boxes, you know that if the system gets powered down before the disk drives are unmounted, then `e2fsck` *gets run on the disks on boot to correct any inconsistencies.*

5. 13.2

The system just read blocks 125 and then 143.

FCFS The seek order is: 86, 1470, 913, 1774, 948, 1509, 1022, 1750, 130.
The total number of cylinder's seeked over is: 7081.

SSTF The seek order is: 130, 86, 913, 948, 1022, 1470, 1509, 1750, 1774.
The total number of cylinder's seeked over is: 1745.

SCAN The seek order is: 913, 948, 1022, 1470, 1509, 1750, 1774, 4999, 130, 86.
The total number of cylinder's seeked over is: 9769.

LOOK The seek order is: 913, 948, 1022, 1470, 1509, 1750, 1774, 130, 86.
The total number of cylinder's seeked over is: 3319.

C-SCAN The seek order is: 913, 948, 1022, 1470, 1509, 1750, 1774, 4999, 0, 86, 130.
The total number of cylinder's seeked over is: 9985.

6. 13.3

Physics

(a) The seek time is proportional to the square root of the seek distance because if $d = \frac{1}{2}at^2$ then $t = \sqrt{\frac{2}{a}d}$.

(b) Using the equation $t = c + d\sqrt{L}$ (I do not like to use x and y as coefficients), we can calculate c and d from our two data points (1cyl, 1ms) and (4999cyl, 18ms). Given two unknowns and an do data points we can solve for those unknowns. It turns out that $c = .76$ and $d = .24$. However, we still haven not said what these coefficients really represent in terms of the acceleration, a, of the disk head, the actually seek time t_s and the overhead of doing a seek t_o. The total seek time we will call t_t. First off, $t_t = t_s + t_o$. Also half the seek time is given by the formula $t = \sqrt{\frac{2}{a}d}$ from above with d as half the cylinder distance. This is because it accelerates half way there and then decelerates the rest of the way. So, $t_s = 2\sqrt{\frac{L}{a}}$. Plugging this in with our formula for t_t we get:

$$t_t = t_o + 2\sqrt{\frac{L}{a}}$$

This means that $c = t_o$ and $d = 2\sqrt{\frac{1}{a}}$. We could not solve for a if we wanted. *Note that t_o is close to 1. If we had made the simple approximation that the (1cyl, 1ms) data point was purely overhead and that the (4999cyl, 18ms) data point was 1ms overhead and 17ms seek time, we would not have been far off.*

(c) Plugging the formula $t_t = 0.76 + 0.24\sqrt{L}$ in for each seek in the schedule we get the following seek times:

FCFS 64ms.

SSTF 31ms.

SCAN 61ms.

LOOK 40ms.

C-SCAN 64ms.

Just for your enlightenment, I came up with my solution to the with the following perl script. Computers are for saving people time. Use them.

```
#!/usr/bin/perl -n
# the -n flag does a 'while (<>) { ... }' around this program

chomp;                                  # remove trailing \n
@order = split /\s*\,\s*/;              # split string on ,

$prev = 143;
$sum = 0;
$time = 0;
foreach(@order)
{
   $time += 0.76 + 0.24 * sqrt(abs($prev - $_));
   $sum += abs($prev - $_);
   $prev = $_;
}
print "Sum: $sum\n";
print "Seek Time: $time\n";
```

I ran this program in input of the following form:

```
86, 1470, 913, 1774, 948, 1509, 1022, 1750, 130
130, 86, 913, 948, 1022, 1470, 1509, 1750, 1774
913, 948, 1022, 1470, 1509, 1750, 1774, 4999, 130, 86
913, 948, 1022, 1470, 1509, 1750, 1774, 130, 86
913, 948, 1022, 1470, 1509, 1750, 1774, 4999, 0, 86, 130
```

(d) The fastest schedule was SSTF. Calculating the percentage speedup when compared to FCFS, we get:

$$100 \times \frac{64 - 31}{64} = 52\%$$

.

7. Given the file `fizbin` that is in some directory, possibly my home directory, in order to read that file, I must load the directories in order, outer most directory first. For example, if the fully qualified path of `fizbin` is `/grizzly/home/hartline/fizbin` then I would have to access `/` to find the inode for `grizzly`, `grizzly` to find the inode for `home`, then I would have to access `home` to find the inode for `hartline`, then I could find the inode for `fizbin`.