CS414 SP 2007 Assignment 4

Due Mar. 28 at 11:59pm Submit your assignment using CMS

1. Suppose that a disk drive has 5000 cylinders, numbered 0 to 4999. The drive is currently serving a request at cylinder 143, and the previous was at cylinder 125. The queue of pending requests, in FIFO order, is:

    86, 1470, 913, 1774, 948, 1509, 1022, 1750, 130

Starting from the current head position, what is the total distance (in cylinders) that the disk arm moves to satisfy all the pending requests for FCFS, SSTF, SCAN, LOOK, C-SCAN and C-LOOK.

2. Assume that you are writing a data storage application which must maintain k disks of data, but cannot communicate with the global Internet or, specifically, off-site data backups. Moreover, you find yourself in the unenviable position of being exposed to intermittent sunspots - specifically, the chance of a neutron striking sectors on your disks and altering the data on them is much, much higher than usual! To improve the reliability of your application, you elect to have it use stable storage to detect and, hopefully, correct disk errors. Cost is not necessarily an object, you can easily afford up to 2^k disks, if necessary.

a) If it must support multiple concurrent readers and multiple concurrent writers, which level(s) of RAID could you use? What are the downsides of each?

b) If it must support multiple concurrent readers but need no longer support multiple concurrent writers, which level(s) of RAID could you now use that you could not use before? What are the downsides of each?

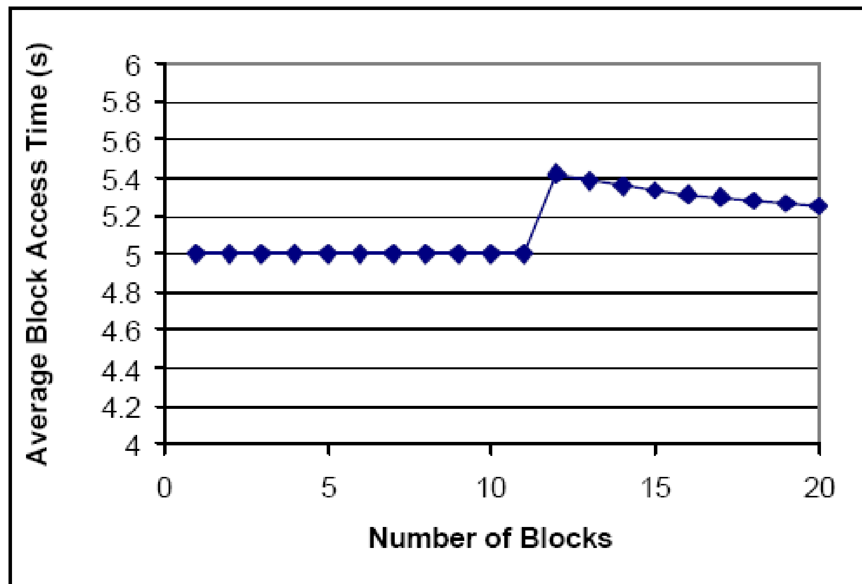
c) Assume (b) again, but now cost has become an issue - you can only afford 2*k-1 disks, and a very crude disk controller. Which levels of RAID become unavailable? Which levels of RAID become undesirable?

3.

(a) In some systems, the i-nodes are kept at the start of the disk. An alternative design is to allocate an i-node when a file is created and put the i-node at the start of the first block of the file. Discuss the pros and cons of this alternative.

(b) Assuming the style of i-node storage where the i-node is stored at the first block of a file, how many disk operations are needed to fetch the i-node for the file /a/b/c/d/e (where a/b/c/d is the absolute directory path, and 'e' is the file name)? Assume that the i-node for the root directory is in the memory, but nothing else along the path is in the memory. Also assume that each directory fits in one disk block.

(c) On a Unix-like filesystem with no bad blocks on disk. A researcher measures the average access time for each block of various N-block files, where he varies N from 1 to 20. The result graph is shown below. What could be the reason for the increase in per block access time at 12 blocks?



4. Consider a standard computer architecture in which the operating system is about to initiate a disk I/O operation initiated by a user program that calls the `write` system call. Assume that the `write` will be one whole disk block and that the argument to the write is a page-aligned region within the user's address space.

(a) Walk us through the steps that the operating system will perform in order to do this `write` and report completion to the user. Assume that we are not writing to the disk buffer pool – the I/O will be done directly to the disk, using DMA transfer from the page in the user's address space. You can assume that the disk is idle and that no other requests are pending.

Note: we're only interested in things the operating system does. And we are looking for an answer expressed in terms of relatively high level functional steps – an answer shouldn't involve more than about 10-15 separate steps. For example, a step might be "mark the process state as waiting and schedule some other process.". Please don't copy-paste chunks of Linux operating system code

(b) Most operating systems provide a user-level implementation of a write routine that converts byte-at-a-time I/O operations (such as characters printed by the C printf() procedure) into block I/O operations. Yet system calls like `write` usually allow the program to indicate how many bytes are to be written, and in principle, nothing stops the application from calling write on each byte, or set of bytes, that it produces. Why do so many operating systems use these intermediary routines? Are there conditions under which such a routine might hurt performance or otherwise give undesired behavior?

5. Modern Unix support the `rename(char *old, char *new)` system call. Unix v6 does not have a system call for rename; instead, rename is an application that make use of the `link` and `unlink` system calls as following:

```
int rename (char *old, char *new) {
      unlink(new);
      link(old, new);
      unlink(old);
}
```

What are the possible outcomes of running `rename("hw4.pdf", "hw5.pdf")` if the computer crashes during the rename call with above implementation? Assume that both "hw4.pdf" and "hw5.pdf" exist in the same directory but in different directory blocks before the call to `rename`.

(Check the Unix man page for details about `rename, link, unlink` system calls)