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.

The FCFS schedule is 143, 86, 1470, 913, 1774, 948, 1509, 1022, 1750, 130. The total seek distance is 7081.
The SSTF schedule is 143, 130, 86, 913, 948, 1022, 1470, 1509, 1750, 1774. The total seek distance is 1745.
The SCAN schedule is 143, 913, 948, 1022, 1470, 1509, 1750, 1774,4999, 130, 86. The total seek distance is 9769.
The LOOK schedule is 143, 913, 948, 1022, 1470, 1509, 1750, 1774, 130, 86. The total seek distance is 3319.
The C-SCAN schedule is 143, 913, 948, 1022, 1470, 1509, 1750, 1774, 4999, 0, 86, 130. The total seek distance is 9985.
The C-LOOK schedule is 143, 913, 948, 1022, 1470, 1509, 1750, 1774, 86, 130. The total seek distance is 3363.

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?

There have to exist at least 2 blocks that can be read/written completely independently. Therefore, only levels 0, 5 and 6 might work. Level 0 provides no error recovery, so it can not be used in this context. However, level 1+0 mirrors the data to provide redundancy but otherwise works as level 0 does, so it would be a good option.

Level 5 relies on distributed parity blocks, so writes are somewhat expensive as a block from each other disk must be read in order to calculate the new parity value. Only the actual location of the write and the location of the particular distributed parity block are actually written to, so a few writes, the

exact number depending on the size of the array, can occur in parallel. Since level 5 relies on parity for error recovery, it can only correct errors in a single block among each group of blocks (each group contains exactly one block from every disk - the error can be the result of disk failure or just data corruption, which is more likely in this case). Recovery from an error is expensive in the same way writes are expensive, as all other blocks from the group (including the parity block this time) must be read in to calculate the correct value for the block with an error. The number of concurrent writes is limited to slightly more than half of the data containing disks, since each data write must also write a parity block to another disk, but in practice the level of concurrency will be lower since there is some chance that the disk a that a write needs to go to is already being written to by either a data a parity write.

Level 6 is very similar to level 5 with the exception that instead of storing parity, it stores error correcting codes such Reed-Solomon codes. These coding schemes use 2 or more bits per bit of data, but allow recovery from 2 or more blocks in a group (depending on the particular scheme in use). All of the other downsides of 5 are roughly the same - although calculating error correcting codes is more expensive than calculating parity, it is easy to implement in hardware and becomes negligible in the face of having to read in a block from each disk.

Level 1+0 is resistant to a large number of errors, as long as one block in each of the mirrored groups is intact, all data may be recovered. Recovering from errors is very efficient, because it is just copying over the data from a good mirror. By increasing the number of disks in each mirror group (it does not have to be pairs), this scheme can scale to handle very high error rates. Writes must be mirrored to the other disks in the mirror group, but this can occur simultaneously, so there is no overhead for writes, giving much better performance than 5 or 6. Assuming there are N data disks and K mirrors of each, 1+0 can support up to N concurrent writes and N*K concurrent reads. The only real downside is the use of N*K disks, with K >= 2, but for many values of K, we have that many disks, so in this situation it is not really a problem.

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?

Level 1 and level 4 allows for multiple concurrent readers but not for multiple writers, so they are both now possible solutions.

Level 1 has efficient reads and writes but high disk overhead, much like level 1+0. It does not scale to large storage needs, since the maximum data size is limited by the maximum size of a single disk. It can only allow K concurrent readers, where K is the number of disks in a mirror group. Since K is usually 2, this is not that good.

Level 4 is very similar to level 5, with the exception that it always stores parity data on a designated parity disk, rather than distributing the parity data over all disks. This means that all writes must touch the parity disk, which eliminates concurrent writes and tends to overuse the single parity disk. Level 4 has the same error recovery abilities as level 5, as well as the same inefficient write characteristics. If there are N data disks, level 4 can support N concurrent reads.

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?

All variations of level 1 are now unavailable because these levels use at least 2*K disks. Levels 4, 5, and 6 are all somewhat undesirable, as the efficient implementation relies on controllers that are more complex, at least relative to the controller for level 1. The level 6 controller would be the most complex because it has to distribute parity over all of the disks and must calculate error correcting codes. The level 5 controller would be slightly more simple since it only has to calculate parity instead of ECC. The level 4 controller would be the most simple (the closest to crude) of the available levels since it also does not have to distribute parity among all disks, so it would be the best choice.

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.

Pros: No disk space is wasted on unused i-nodes and it's not possible to run out of i-node. Less disk movement for small files is needed since the i-node and the initial data can be read in one operation.

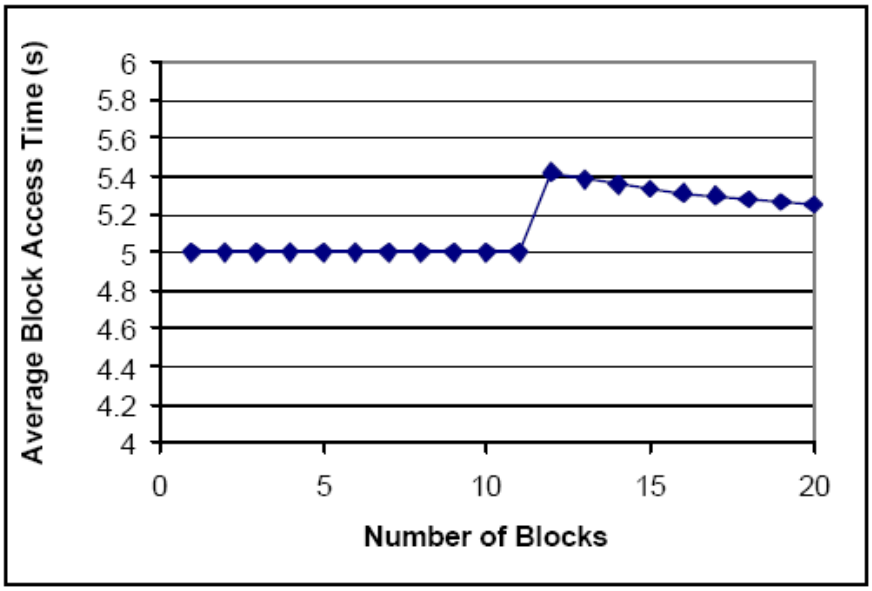
Cons: file system integrity checks will be slower because of the need to read an entire block for each i-node and because i-nodes will be scattered all over the disk. Files whose size has been carefully designed to fit the block size will no longer fit due to the i-node, messing up performance.

(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.

Let's assume that for any directory, it takes one disk operation to retrieve the i-node, and one more operation for the directory content. Then the following disk reads are needed:

1. dir content for /
2. i-node for a
3. dir content for a
4. i-node for b
5. dir content for b
6. i-node for c
7. dir content for c
8. i-node for d
9. dir content for d
10. i-node for e

(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?



The i-node only holds 11 direct blocks, so on the 12th block, an indirect block must be loaded into memory to find the actual location of the 12th block, so the access time jumps up. Since the same indirect block contains a long run of blocks after 12, those blocks can found from the same indirect block, so as the file size increases past 12, the average access time decreases, since the time spent loading the indirect block is amortized over several file blocks instead of just the one.

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 invol ve 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

Note: The order the events need not to be exactly as the solution below, but some steps have to happen before others (such as step 1). Here we refer to the process that calls write as Process W

1. Switch from user mode to kernel mode, save registers.
2. Verify the parameter (validity of the file descriptor, opened and writable; memory validity of write buffer address, length, etc.)

3. Identify the physical disk blocks to be written (allocate new one if needed). (This step actually contain omits many filesystem details, such as, checks to ensure resulting file does not exceed some limits;
4. Setup the necessary information for lower-level component (such as device driver). Device driver schedules the I/O, eventually sends command to DMA controller.
5. Mark process W's state as waiting and schedule some other process
6. DMA operates the device hardware to transfer data from process W's address space to disk. When the transfer is done, it raises an interrupt.
7. Switch to kernel mode if not already in it. Save registers. Service the interrupt, store any necessary data, signals the device driver, and return from the interrupt
8. The device drive determines the result of the I/O operation (success or not). Marking process W's state as runnable.
9. (Sometime later, or maybe immediately) Process W gets to run, update the corresponding inodes, setups the return value (completion or error code), return from the system call. Switch back to user mode, restore the registers.

(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?

The user level routine is convenient in that the application can write whenever it has data that needs to be written. The OS worries about when to optimize the writes by caching them until there is enough aggregate data to make an actual write to disk efficient. The downside is that if there is a system crash or the disk is removed, the cached but not yet written to the disk data will be lost.

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)

Because the effects of the calls in the rename implementation are not persistent unless the corresponding block writes go out to disk, it is *not* correct simply to consider the failure happening before or after each call. Instead, we need to think about the possible block writes that can happen before the failure.
The rename implementation above causes three block writes First, the directory entry for `new` is removed from its block. Second, the directory entry for *new* is added back to its block, but with the i-node for *old*. Third, the directory entry for *old* is removed from its block.

The possible outcomes are:

- Nothing done: no changes written to disk yet.
- The first write went to disk: *old* remains as before, but *new* doesn't exist at more (if it has exist before)
- The first and second writes went to disk: *old* remains as before, but now *new* points at the same file as *old*.
- The first, second, and third writes went to disk: *old* is gone, and *new* points to where *old* was rename succeeded.