CSE 120: Homework #4 Solutions

Fall 2015

Out: Friday November 20

Due: Thursday December 3 at the start of class
(No late homeworks starting Saturday December 5)

1. On a Unix-style file system, how many disk read operations are required to read the first block of the file "/usr/include/stdio.h"? Assume that the master block is in memory, but nothing else. Also assume that all directories and inodes are one block in size.

```
Read inode for "/"
Read data block for "/"
Read inode for "usr"
Read data block for "usr"
Read inode for "include"
Read data block for "include"
Read inode for "stdio.h"
Read data block for "stdio.h"

8 reads
```

2. Consider a UNIX-style inode with 10 direct pointers, one single- indirect pointer, and one double-indirect pointer only. Assume that the block size is 4K bytes, and that the size of a pointer is 4 bytes. How large a file can be indexed using such an inode?

   We have 10 direct pointers that each point directly to a 4K data block, so we can point to 10 * 4K = 40K with the direct pointers.

   We have one single-indirect pointer that points to a 4K disk block (an "index block") which contains nothing but pointers to data blocks. Since each disk block pointer is 4 bytes, the index block can hold 4K / 4 = 1K pointers to data blocks. So, the single-indirect pointer indirectly points to 1K * 4K = 4M ($2^{10} * 2^{12} = 2^{22}$ bytes).

   We have one double-indirect pointer that points to an index block that contains 1K pointers to more index blocks. These leaf index blocks each contain 1K pointers to data blocks. So, the double-indirect pointer indirectly points to 1K * 1K * 4K = 4G ($2^{10} * 2^{10} * 2^{12} = 2^{32}$ bytes).

   Adding everything up, the maximum file size is 40K + 4M + 4G.

3. Consider a file archival system, like the programs zip or tar. Such systems copy files into a backup file and restore files from the backup. For example, from the zip documentation:

   The zip program puts one or more compressed files into a single zip archive, along

with information about the files (name, path, date, time of last modification, protection, and check information to verify file integrity).

When a file is restored, it is given the same name, time of last modification, protection, and so on. If desired, it can even be put into the same directory in which it was originally located.

Can zip restore the file into the same inode as well? Briefly explain your answer.

In general, the archive program cannot guarantee that the file will be restored to the same inode.

User programs like zip and tar use the logical file system (filenames and directories). inodes are part of the physical file system (superblock and inodes). The logical file system is built on top of the physical file system, and, in general, the kernel only provides a syscall interface for the logical file system.

So if an archive program wanted to guarantee that a file is restored to its original inode, it would have to write directly to the hard disk (bypassing the file system entirely), since there is no syscall interface to write to a particular inode. This means that the archive program would need root access, and it would need to understand the layout of the physical file system on the hard disk.

In particular, writing a file to its original inode becomes very difficult if the original inode is in use by some other file *f*. The archive program would have to relocate *f* to another inode, which is hard because the archive program would have to search the disk for all references to *f*'s old inode, and make them all point to *f*'s new inode. This is extremely difficult if the file system is mounted and there are other running processes that use the file system.

4. [Silberschatz] Consider a system that supports 5000 users. Suppose that you want to allow 4990 of these users to be able to access one file.
   a. How would you specify this protection scheme in Unix?

      The Unix file protection system allows users to specify protection at the granularity of a single user (owner), a group of users (group), and all users (other). To specify this policy on Unix, you would create a group with the 4990 users who are able to access the file and give the file the appropriate group permissions (e.g., read and write).

   b. Could you suggest another protection scheme that can be used more effectively for this purpose than the scheme provided by Unix?

      If we are not constrained by what Unix provides, more compact approaches would involve some way to specify who does *not* have access. One approach would be to extend the Unix group model to be able to list users who are *not* in the group. With this mechanism, you would create a group as before, but list the 10 users who should not get access. The file would then get the same group permissions, but the group would be interpreted differently.

      Another approach would be to be able to specify negative permissions on a file. In this approach, you would create a group with the 10 users who should not access a file,

and then assign group permissions on the file as negative permissions (e.g., users in this should group should not be able to access the file).

A third approach would be to have very flexible access control lists where you can associate an arbitrary number of access control entries (not just "owner", "group", "other"). In such a system, you could create access control entries for each of the 10 users specifying that they did not have access to the file, and then have a wildcard entry specifying that every other user (who does not specifically match the first 10) can access the file. (For those who have ever created a .htaccess file for Apache, you likely have done something like this with a "deny from all" or "accept from all".)

5. [Silberschatz] How does a file cache help improve performance? Why do systems not use much larger caches if they are so useful?

   File caches improve performance by converting what would be disk I/O operations (read, write) into memory operations (reading from memory, writing into memory). For those operations that can be served from the cache (e.g., one user accessing "/usr/include/stdio.h" and then another user accessing it soon after), they operate at the speed of memory rather than the speed of disk. Since disk I/O is much slower than memory, reducing the number of disk I/Os substantially improves performance.

   But there is always a tradeoff. In this case, the file buffer cache uses physical memory to store the file data. As a result, it reduces the amount of physical memory that the OS can use for mapping virtual pages. As more physical memory is used for the file buffer cache, less is used to store virtual pages and processes may start to page fault more. At some point, growing the file buffer cache will result in more disk I/O (for handling page faults) than the file buffer cache saves in caching file data. This situation reduces performance overall, which we want to avoid.

---

voelker@cs.ucsd.edu