CS 452 Lab File System Interface: Information

Overview

The purpose of this lab assignment is to investigate characteristics of modern file systems. Specifically, the goal is to improve your understanding of file storage, file access, file information management, directory structure, and file system traversal.

Hand-in:

- A word document containing the requested information
- Any requested screenshots (uploaded as separate files)
- Program source code (no zip files)

Files

Every file in the UNIX file system, including directories, has an associated *inode*. An inode is a data structure that contains all of the information the system maintains on every file. For example, an inode may contain information on file access permissions, file creation time, block size and disk block addresses. This information is used by the file system to find files on secondary storage, and to perform auditing and administrative functions. The "location" information helps implement the mapping of logical file addresses to physical disk blocks (this is how a filesystem locates specific bytes in a file - reminiscent of paged virtual memory management). The administrative information is what will be investigated in this lab.

Some of the information contained in an inode can be viewed using the `stat()` function. The mechanism should be familiar - the system call is given a filename and provided with a user-supplied structure. It works by filling in the appropriate fields of the predefined structure with the current values from the inode of the specified file; users then access this structure to obtain information about the file the inode describes.

Take a moment to read the man pages for `fstat()` and `stat()`; be sure to read both the `stat(1-2)` pages (user program, system call). You can also visit the man pages online (e.g., https://man7.org/linux/man-pages/man1/stat.1.html)

Then carefully examine the following program that accesses an inode data structure:

sampleProgramOne

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <errno.h>

int main(int argc, char *argv[])
{
   struct stat statBuf;

   if (argc < 2) {
      printf ("Usage: argument (filename) expected\n");
      exit(1);
   }

   if (stat (argv[1], &statBuf) < 0) {
      perror ("Program error: ");
      exit(1);
   }

   printf("value is: %u\n", statBuf.st_mode);

   printf("inode value is: %lu\n", statBuf.st_ino);


   return 0;
}
```

1. Perform the following operations and answer the associated questions (7 points):

a) Access the man pages, what is the difference between **`stat(1)`** and **`stat(2)`**?
b) Compile and test sampleProgramOne. Run it twice: use the sampleProgramOne source code file and then its executable file as test inputs. What *exactly* does sampleProgramOne do?
c) Modify sampleProgramOne so that it reports whether a file is a directory or not. Verify that your program works (as shown below). Include a screenshot of the execution. Also include your source code as an attachment when uploading to blackboard.

- Use sampleProgramOne and your current directory as your test inputs. *Verify* and demonstrate the correctness of your program by testing its output against the output of the **`stat(1)`** utility. For example:
  - **stat sampleProgramOne.c**
  - **./sampleProgramOne sampleProgramOne.c**

Directories

File access proceeds via directories. A directory is itself a file, and simply contains a list of tuples consisting of filenames and their corresponding inode numbers. Programs that need to open files or report information about files (e.g., "**cat**", "**ls**") begin by searching the directory for the specified filename. Upon finding the filename, they use the associated inode number to access the inode and from there, the file.

Each directory entry consists of a <filename : inode #> tuple. Being a file, the contents of a directory can be examined. The relevant system calls are:

- **`opendir()`** - opens the named directory and associates a stream with it
- **`readdir()`** - returns a pointer to the next directory entry
- **`closedir()`** - closes the named directory stream

Time to peruse the man pages again to understand the mechanics of the above functions and the data structures they utilize. Then study the following program.

sampleProgramTwo

```
#include <stdio.h>
#include <dirent.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <errno.h>

int main()
{
   DIR *dirPtr;
   struct dirent *entryPtr;

   dirPtr = opendir (".");

   while ((entryPtr = readdir (dirPtr)))
      printf ("%-20s\n", entryPtr->d_name);

   closedir (dirPtr);
   return 0;
}
```

2. Perform the following operations and answer the associated questions (9 points):

a) Compile and test sampleProgramTwo, what exactly does sampleProgramTwo do?

b) Modify sampleProgramTwo so that it also reports the size of each file (in bytes). Ensure your output is human-friendly (i.e., readable)
c) Verify and demonstrate the correctness of your program by testing it against the ls program using your current directory. For example 'ls -l' should return the same value as './sampleProgamTwo'. Submit your modified program and include a screenshot of the execution.

File Systems

The UNIX file system is hierarchical, and implements a general graph structure that includes hard and symbolic links. Because of its tree-like organization, tree traversal algorithms (such as the recursive depth-first-search and breadth-first-search routines) are generally used to traverse the file system to access files and directories. Note: this process is slightly complicated by the presence of symbolic links and the subsequent possibility of cycles.

The system utility **du** (disk usage) is an example of a file system traversal program. It provides a summary of the amount of disk space currently occupied by a user's files. This information is totaled by directory. The program operates by recursively descending into a user's subdirectories to report usage statistics on each entire subtree.

Read the man pages for the **du** utility and experiment with it. While you are doing this, recall the definitions of depth-first-search (DFS) and breadth-first-search (BFS), as covered in your data structures and/or discrete math class(es).

3. Answer the following questions (9 points):

a) Use **du** to report the usage of all the files in some of your directories (be sure to choose some with subdirectories), based on the *order* of information provided, which of the two tree traversal algorithms does **du** use?
b) What is the default block size used by **du**?
c) Speculate: given the intended purpose of **du**, why is the usage reported in blocks, instead of bytes?

(continue to the next page for programming assignment)

Programming Assignment (`ls` - Directory Listing) (35 points)

Using what you have learned while experimenting with the sample programs given in this lab, write a program that implements a small subset of the functionality of the "**`ls`**" command. Your program should use only the file and directory system calls covered in this lab. Specifically, in addition to listing all files, it should output:

- user and group IDs
- file inode numbers

You can verify your program is correct by using the following commands:

`ls -n` // gives user and group ID for each file (along with some other info)

`ls -i` // gives the inode # for each file

Your program should accept as input the name of any directory, whose contents will then be listed, along with the specified information (just like '**ls**'). Verify that your program works. Submit your source code and include a screenshot of the execution.