Computer Operating System Experiment

Laboratory 8

File system

Objective:

- Gain practice in working with the Linux file system interface: This lab will have you work with functions to inspect various attributes of files, as you modify a program for traversing a directory tree.
- Gain more practice with the higher-level file I/O interface: By now, you have used the C/Linux file I/O APIs for lower level syscalls (open, read, etc.) and for higher level streams (fopen, fread, etc.). It pays off to think critically about the differences between the two kinds of interface, so that you understand a little more about the design of the operating system.

Equipment:

VirtualBox with Ubuntu Linux

Methodology:

Program and answer all the questions in this lab sheet.

1. File Management

File management is one of the basic and important features of operating system. Operating system is used to manage files of computer system. All the files with different extensions are managed by operating system.

A file is collection of specific information stored in the memory of computer system. File management is defined as the process of manipulating files in computer system, it management includes the process of creating, modifying and deleting the files.

2. Experiments

2.1 Experiment 1: the Linux file system interface

Create a C program called fdump.c that works as described below.

1) The program must accept the following command line parameters, in the order given: filename (a C array), offset (unsigned integer), and size (unsigned integer). If the three command line parameters are not provided by the user, the program must terminate immediately with an error message and indicate the proper usage (that is, the order and the types of command line parameters expected).

2) The program opens the file indicated by filename with **fopen**, moves forward the file position indicator by the number of bytes indicated by offset, and reads size bytes from filename into a buffer. (Hint: you will need to make a call to a random-access library function to get to the right read location into the user-specified file.)

3) Once that data is read into your program's buffer, make it call the function **hexdump** provided to you in files **hexdump.h** and **hexdump.c**. The output generated will resemble the example below.

   fdump [fileName: char[]] [offset: int] [size: int]

   for example: ./fdump fdump.c 10 32

```
0000000: d4c3 b2a1 0200 0400 0000 0000 0000 0000  ................
0000010: ffff 0000 0100 0000 47c5 a943 fd14 0300  ........G..C....
0000020: 2a00 0000 2a00 0000 ffff ffff ffff 0001  *...*...........
0000030: 039c ffbd 0806 0001 0800 0604 0001 0001  ................
0000040: 039c ffbd c0a8 0166 0000 0000 0000 c0a8  .......f........
0000050: 0101 49c5 a943 5eed 0d00 4000 0000 4000  ..I..C^...@...@.
0000060: 0000 0001 039c ffbd 000c 41a1 f5da 0806  ..........A.....
0000070: 0001 0800 0604 0002 000c 41a1 f5da c0a8  ..........A.....
...
```

Requirements:

1) Create a Makefile to generate your fdump executable. You should compile hexdump.c separately into an object that gets linked with the compilation of fdump.c at a later point.
2) Run your fdump program with the following parameters: filename = hexdump.c, offset=1000, size=128. Explain what you see.
3) Run your fdump program with the following parameters: filename = fdump, offset=500, size=128. Explain what you see.
4) Compare the output you produced for answers (2) and (3). Looking at the hexadecimal dump on the left, in both cases, makes it clear that inside both files, you store information in binary encoding. However, the data to the right of the hexadecimal dump shows something human-readable for (2) and non-human readable for (3). In this answer, you are asked to explain why this is the case.

2.2 Experiment 2: output the file information

The manual page for **fstatvfs** shows that it returns various pieces of information on the underlying file system in which the given file resides. Using this call, you can learn the block size for the file system, the type of the file system, and the maximum length for a file name. The call returns all these data and more in an instance of struct **statvfs** (a pre-defined type), which you must have allocated previously. **fstatvfs** will receive a pointer to your instance of struct statvfs and fill up its various fields of file system information.

```
/*The function statvfs() returns information about a mounted file system. path is the
pathname of any file within the mounted file system. buf is a pointer to a statvfs structure
defined approximately as follows:
*/
#include <sys/statvfs.h>
int fstatvfs(int fd, struct statvfs *buf);

struct statvfs {
        unsigned long    f_bsize;       /* file system block size */
        unsigned long    f_frsize;      /* fragment size */
        fsblkcnt_t       f_blocks;      /* size of fs in f_frsize units */
        fsblkcnt_t       f_bfree;       /* # free blocks */
        fsblkcnt_t       f_bavail;      /* # free blocks for unprivileged users */
        fsfilcnt_t       f_files;       /* # inodes */
        fsfilcnt_t       f_ffree;       /* # free inodes */
        fsfilcnt_t       f_favail;      /* # free inodes for unprivileged users */
        unsigned long    f_fsid;        /* file system ID */
        unsigned long    f_flag;        /* mount flags */
        unsigned long    f_namemax;     /* maximum filename length */
};
```

Reading about **fstat**, you will see that it returns a stat struct, which is another pre-defined type. This struct contains information such as the user id and the group id for the owner of the file, its protection bits (for user, group and other), the file size in numbers of blocks, and the times when it was last accessed, modified, and created. You must pass to **fstat** the pointer to an instance of stat struct, which you allocated previously; the system call will fill up the various fields with information on the specific file.

```
//stat, fstat, lstat - get file status

#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>

int stat(const char *path, struct stat *buf);
int fstat(int fd, struct stat *buf);
int lstat(const char *path, struct stat *buf);
struct stat {
        dev_t       st_dev;       /* ID of device containing file */
        ino_t       st_ino;       /* inode number */
        mode_t      st_mode;      /* protection */
        nlink_t     st_nlink;     /* number of hard links */
        uid_t       st_uid;       /* user ID of owner */
        gid_t       st_gid;       /* group ID of owner */
        dev_t       st_rdev;      /* device ID (if special file) */
        off_t       st_size;      /* total size, in bytes */
        blksize_t st_blksize; /* blocksize for file system I/O */
        blkcnt_t    st_blocks;    /* number of 512B blocks allocated */
        time_t      st_atime;     /* time of last access */
        time_t      st_mtime;     /* time of last modification */
        time_t      st_ctime;     /* time of last status change */
};
```

Reading the source code given to you, you will notice that you must open the file before you call either **fstatvfs** or **fstat**. The code contains missing portions that you will construct in this lab. Look for them where you find comments containing the string TO-DO. Once you have filled in the missing pieces, the output produced by your file_stat executable on file file_stat.c should be somewhat similar to what is presented below (minus the text in red).

Augment the program to print the time of last access and the time of status change for the file given as command-line parameter. Note that you will need to read the man pages for **getpwuid** and **getgrgid** to learn how to translate numeric USER ID and GROUP IP to strings, respectively.

```
$ file_stat file_stat.c
== FILE SYSTEM INFO ============================
file system fstatvfs() call successful
file system block size: 65536 <———————— For you to do
max. file name length: 255 <———————— For you to do
== FILE INFO ============================
file fstat() call successful
file protection bits = 0644
file protection string = rw-r–r– <———————— For you to do
file protection mode (u:g:o) = 6:4:4    <———————— For you to do
owner user name = perrone <———————————— For you to do
owner group name = cs <———————————————— For you to do
mode = x <—— For you to do (x may be file, link, directory, socket, etc.)
time of last modification: Thu Nov 14 15:04:21 2014
time of last access: <———————————————— For you to do
time of status change: <—————————————— For you to do
```

2.3 Experiment 3: traverse files

You need to create a new program called **traverse.c**, which will traverse a given directory tree, printing to the standard output the following information:

- The value of the smallest, the largest, and the average file size.
- Total number of directories.
- Total number of regular files, that is, those which are not directories, symbolic links, devices, sockets, or fifos.
- The name of the file that was most recently modified, and the one that was least recently modified in the directory tree.

Note that the size of a file can be accessed from the struct stat returned by calling **fstat**. Read the program **file_stat.c** and the manual pages **fstat** and **lstat** for more information. Make sure to modify the Makefile given to you so that it will build the newly created traverse program.

When you complete the previous step, run your program in a directory tree where there is no symbolic link and observe its behavior. Next, in a directory of your own, create a symbolic link which links it to its parent directory. Note that you are creating a loop in the directory graph. Run the program again and note what happens.