

**Lab 8 Report**

**Report Subject: OS Experiment - Lab 8**

**Student ID: 2018380039**

**Student Name: Dikshya Kafle**

**Submission Date: 2021/12/1**

**Computer Operating System Experiment**

**Computer Operating System Experiment**

**Laboratory 8**

**File system**

1. **Objective:**

* Gain practice in working with the Linux file system interface.
* Gain more practice with the higher-level file I/O interface.

1. **Equipment:**

* VirtualBox with Ubuntu Linux

1. **Experiments:**

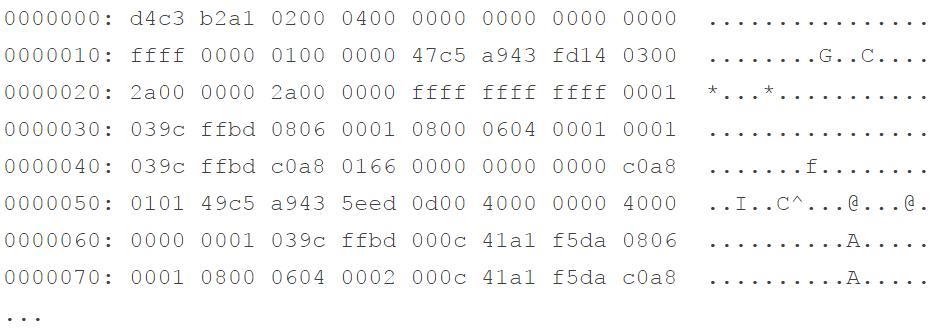
**Experiment 1: The Linux file system interface.**

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

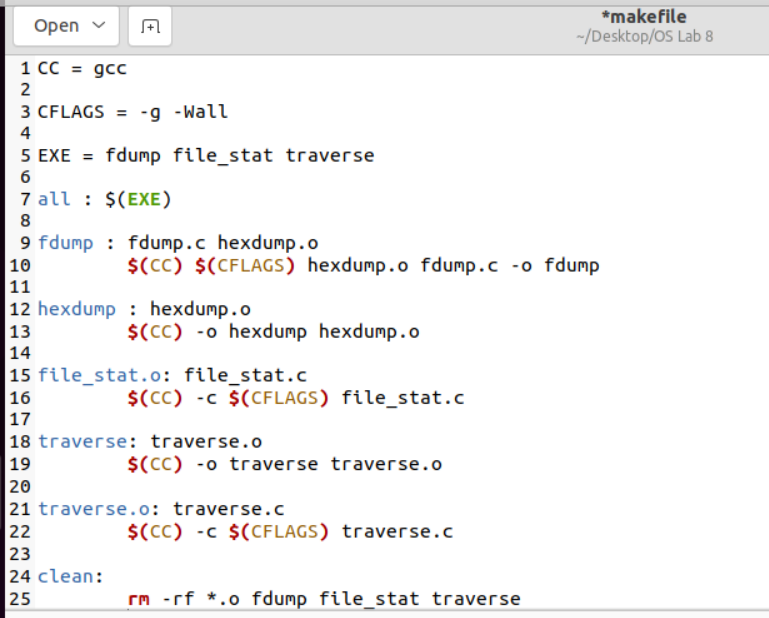
1. The program must accept the following command line parameters, in the order given: filename (a C array), offset (unsigned integer), and size (unsigned integer). If the three command line parameters are not provided by the user, the program must terminate immediately with an error message and indicate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

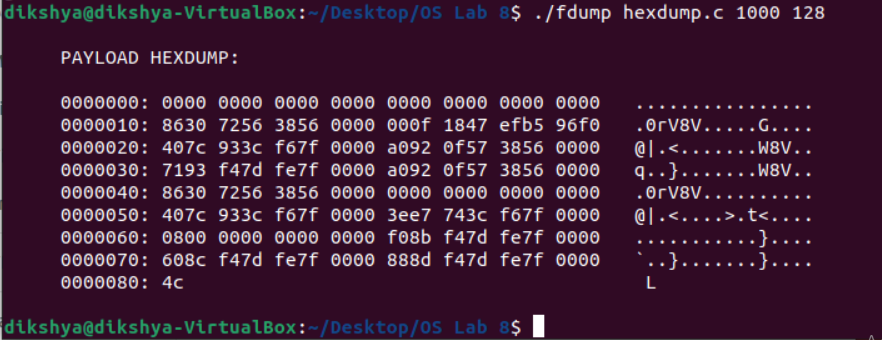


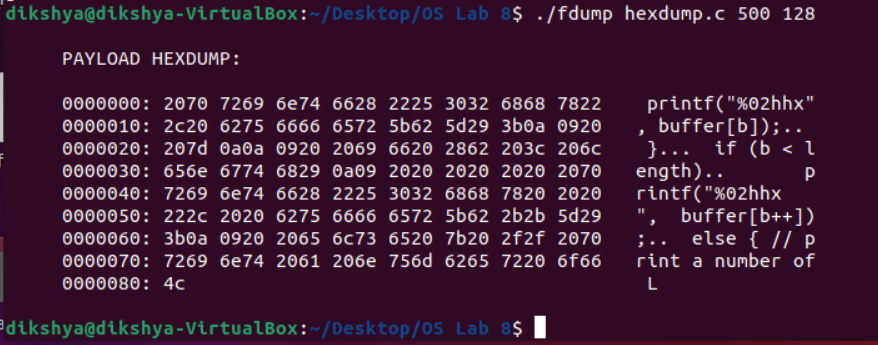
1. We create a Makefile to generate the fdump executable. The hexdump.c is compiled separately into an object that gets linked with the compilation of fdump.c.



1. We run your fdump program with the following parameters: filename = hexdump.c, offset=1000, size=128. To do this we run the command: ***“./fdump hexdump.c 1000 128”.***

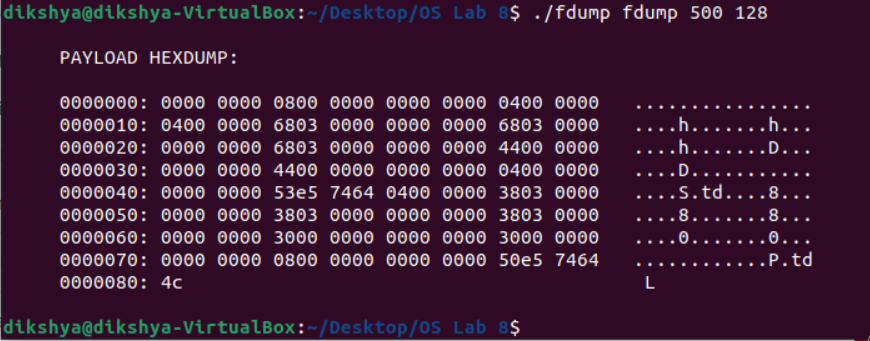
The output shows hex values in an array format. To the right of the hex values, there is a text line that says "counter for the number of bytes (half the number of hex digits) ...".





1. Run your fdump program with the following parameters: filename = fdump, offset=500, size=128. To do this we run the command: ***“./fdump fdump 1000 128”.***

The output shows the hexcode of the fdump file and to the right of it is a message that contains a lot of periods and some random letters.



1. In question 2.1, the data to the right of the hexadecimal dump shows human-readable text while the data to the right of the hex dump in 2.2 shows a non-human readable output.

This is because, in 2.1, we dump a file written in C, <hexdump.c>, which contains human-readable code and text. In 2.2, we dump an executable file, <fdump>, that consists of mostly non-human readable language which is why we see the difference in the format of the data to the right of the dump.

**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.)

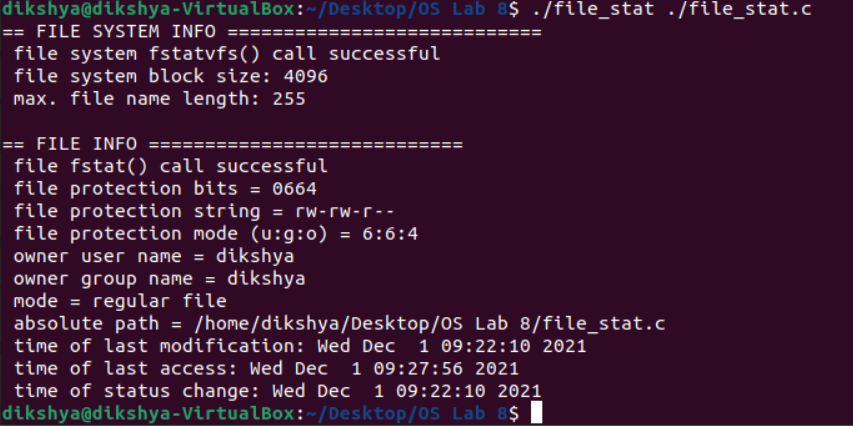
time of last modification: Thu Nov 14 15:04:21 2014

time of last access: <————————— For you to do

time of status change: <————————— For you to do

After filling the missing portions, the output produced by the file\_stat executable on file

file\_stat.c is as following:



**Experiment 3: Traverse files.**

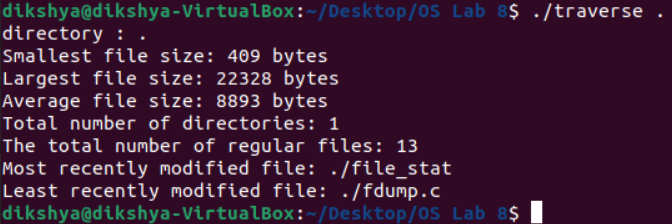
You need to create a new program called **traverse.c**, which will traverse a given directory tree, printing to the standard output the following information:

* 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.

* Running the program in a directory tree where there is no symbolic link:

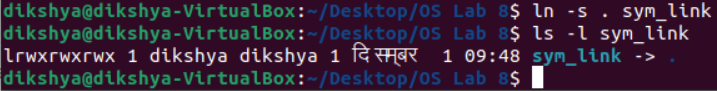


* Running the program in a directory tree where there is a symbolic link:

Creating the symbolic link:



Testing the symbolic link:



Traversing