2020

22nd December

LAB 09

OPERATING SYSTEM

BESE-(9A)

Ali Tariq (241916)

Muhammad Abdullah (241548)

Syed Afraz Hasan Kazmi (241847)

File System Simulation

File System (THREADED)

The objective of this lab is to simulate a filesystem based on an existing directory structure given as an input file and allow multiple threads to manipulate the system without inconsistencies.

The program accepts the following parameters:

- -f [input file for directory structure]
- -s [Disk Size]
- -b [Block/Sector Size]

Run using the format,

python3 filename.py -f -s -b

Example: python3 filesystemwolf.py list 65536 512

Note: A prerequisite to this program is installing the `llist` module which is library of Doubly and Singly Linked Lists. To do so, open Linux terminal, cd into the program directory and write the following command 'pip3 install llist'. (Needs to be done only once)

The program will use the input parameters and create a file system structure, against which you will be able to run commands.

Input file (the -f parameter)

The input files are already provided in this repo but a user can make their own input files using the find Unix command. There are two files namely, ‘dir_list.txt’ and ‘file_list.txt’. The directory list file has **absolute** paths of all directories. The file list file has **size** and **absolute** paths of all files

`find ./ -type d > dir_list.txt` - Generates a list of all directories and subdirectories and outputs it into a txt file.

`find ./ -type f -ls > file_list.txt` - Generates a list of all files with their size and outputs into a txt file. (Note: This file may have to be modified to conform to the required format)

Next is the prompt for input thread files which should be in the following format,

<thread-1.txt> <thread-2.txt> <thread-3.txt> <thread-4.txt> <thread-5.txt>

Note: The order of threads can be varied because they anyways have to run concurrently.

Introducing Threading

The data structure in previous lab (lab-06) was designed keeping threading in mind, therefore all that had to be done was, placing the threading functionalities in the right places.

The method we followed was using **mutex locks** imported from the python threading library. The lock had to be acquired before execution of any **core critical function**, and then be released so that other threads can have their turn.

(A core critical function is any function which requires reading or writing to the main file passed as a parameter to this program **OR** any function that wants to manipulate a globally declared entity. Therefore, only one thread should be allowed such access at a given time to prevent conflicts)

Core critical functions include,

- Make directory function
- Remove directory function
- Make file function
- Remove file function
- Append to file function
- Truncate from file function
- Memory map function
- Print disk status
- Print tree hierarchy

The goal is to read every line in an input thread text file and implement it one by one. Every thread does this making it a parallel behavior. To do so the only changes we make to our code is putting mutex locks in the switch case statements that are used to call a core function depending on the line read from the input thread text file.

See the example below for better understanding,

Non-threaded (Lab 06)

```
elif (vect[0] == "printtree"):
    print("")
    printTree(current, 0)
    print("")
```

Threaded (Lab 09)

```
elif (vect[0] == "printtree"):
    mutex.acquire()
    sys.stdout = file
    printTree(current, 0)
    sys.stdout = originalStdout
    mutex.release()
```

As you can see, the threaded version has ‘mutex.acquire()’ before the core critical function ‘printTree’ and ‘mutex.release()’ after it whereas the non-threaded version directly calls the core critical function.