2020

1st December

LAB 06

# OPERATING SYSTEM

BESE-(9A)

Ali Tariq (241916)

Muhammad Abdullah (241548)

Syed Afraz Hasan Kazmi (241847)

## File System Simulation

# File System

**The objective of this project is to simulate a filesystem based on an existing directory structure given as an input file**

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)

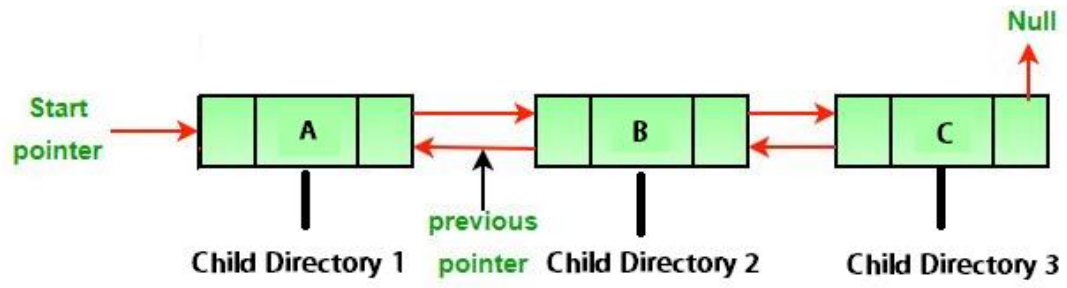
# Data Structuring

Simply put the overall structure is a **Hierarchical Directory Tree** with root node representing the starting directory. Every child node in the tree can either represent a **directory** or a **file.** The file node and directory node have a few key differences,

**Directory node** has the following attributes:

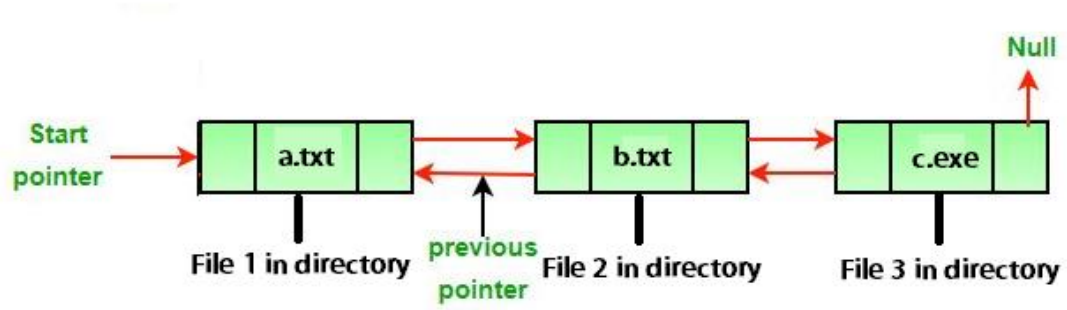
- **Pointer to parent node**
- **Full directory name** (Example: ./User/Afraz/Lab06)
- **Directory name** (Example: Lab06)
- **Directory path** (Example: ./User/Afraz)
- **A Doubly Linked List specifying all of its child directories**
- **A Doubly Linked List specifying all of its child files**

**Doubly Linked List with all child directories look as follows,**




**Every node in this linked list is also a Directory node, meaning it will have its own attributes and also a doubly linked list of its children directories. This pattern follows throughout forming a hierarchical tree.**

**Doubly Linked List with all child files look as follows,**

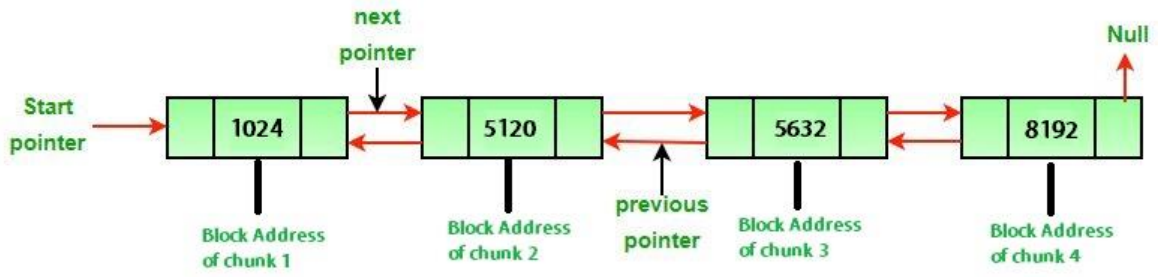



**Every node in this list is a File node with its own attributes and also a doubly linked list of its storage distribution on disk. (See next)**

**File node** has the following attributes:

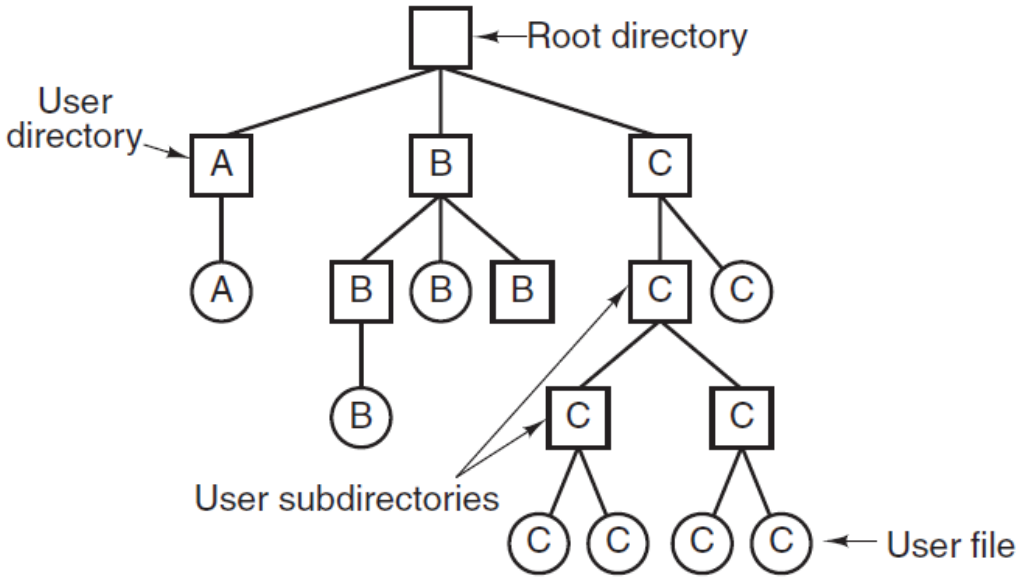
- **Full name** (Example: ./User/Afraz/Lab06/Description.txt)
- **File name** (Example: Description.txt)
- **File path** (Example: ./User/Afraz/Lab06)
- **File size** (in bytes)
- **A Doubly Linked List specifying block address of all file chunks stored in Disk.**

**Doubly linked list in the file node** looks as follows,




As shown, it contains the starting addresses of blocks in disk that have chunks of this file. The file here seems to have been divided into 4 chunks where first chunk is stored at starting address 1024 in disk, second chunk at 5120 and so on… Note: The block number of disk can be calculated by dividing starting address by block size given as parameter -b.

The **structure of tree** can be understood by the following diagram,




**dir_list.txt**

```
./
./1A
./1A/2A
./1A/2A/3C
./1A/2A/3C/4C
./1A/2A/3C/4A
./1A/2A/3C/4B
./1A/2A/3B
```

**file_list.txt**

```
1016 ./f1c.c
0 ./file_list.txt
1604 ./f1b.o
334 ./1A/2A/f3d.txt
82 ./1A/2A/f3a.c
1016 ./1A/2A/f3c.c
334 ./1A/2A/3C/f4d.txt
```

Other than these, a separate Doubly Linked list for Disk is maintained which carries the information on used and free blocks. Every node in this list is a block node which tells from which ID to which ID the disk is used or free.

**Block node** has the following attributes:

- **Status** (whether block free or used)
- **Begin ID**
- **End ID**

**Doubly Linked List with all block nodes look as follows,**


## Implementation

The Disk size is given by the -s parameter and it starts at Block ID 0. The total number of blocks is equal to disk size divided by the block size (-b parameter). For example, a given disk of size 1024 bytes, block size of 16 will yield 64 blocks with block IDs/addresses going from 0 to 63. At any given time, blocks can be either free or used. Contiguous blocks may be allocated to more than 1 file.

Every time a new file is created, we traverse the disk linked list to find free blocks. Only the required number of blocks with status “free” are set to “used”. Disk blocks are allocated to a file from the lowest available address.

A block of size -b as per the parameters cannot be split, so if a file needs only some space from the block, the whole block will be given to the file. The remaining(left over) space in the block is considered to be **Disk Fragmentation.** We keep track of the total fragmentation just as a real file system would.

Lastly, we can append or remove bytes from a file. Appending may result addition to the file node linked list which contains the file storage distribution, but first it will fill the left-over space in the disk block(if applicable)

For example, consider a file of 14 bytes stored in a block of 16 bytes. As you may guess, there is fragmentation of 2 bytes. Now we append 4 bytes to this file. The new file size is 18 bytes. 2 bytes of fragmentation is removed and block is filled to the max. Still we have 2 bytes remaining to be allocated. These 2 bytes are then stored in a new free block on disk. The starting address of this block is appended to the Doubly linked list of file node. Also note that the block with just 2 bytes has now increased total disk fragmentation by 14 bytes(16 -2 = 14). Similar scenario occurs in removing bytes from a file aka shortening file.

Brief notes on each command

- When a directory is created, we append a new directory node to the doubly linked list of parent directory node. i.e. place node in correct position in tree.
- When a directory is deleted, we remove the corresponding directory node from the appropriate doubly linked list i.e. from the tree. (Note: Cannot delete a directory with subdirectories)
- When a file is created, we add the file node to the appropriate files doubly linked list and allocate space to is in disk
- When a file is removed, we remove the file node from the appropriate doubly linked list and free up space in disk.
- We cd into directories by looking up the directory linked list.