This is a team project. No collaboration is allowed between different teams. This project is worth 10% of your final grade. The due date for this assignment is 11:59pm Monday, April 23rd, 2018.

Late penalty: Late assignment (even by 2 seconds) will be given a -25% decrease penalty per day, for the first 2 days after the deadline. So, if you send an assignment 1 second late, you will receive 75% of your grade for the assignment. If you send it, 24 hours, and 1 second late, you will receive 50% of your grade for the assignment etc. After 48hrs from the deadline there will be a -90% decrease penalty, so you will receive 10% of your grade.

Goal

The goal of this assignment is to manage a file system by first instantiating a file system based on an existing directory structure, and then manipulating the files in the file system. You will implement three primary data structures in your file system:

1. A directed tree structure **G** representing the hierarchical directory,
2. A linked-list data structure Ldisk representation of disk space - free and in use, and
3. A linked-list data structure Lfile storing disk block addresses for a file.

(The notations of G, Lfile and Ldisk are for explanation purposes only. You can define the names of these three structures in your code as you wish.)

Your program must accept the following parameters at the command prompt:

- *-f* [input files storing information on files]
- -d [input files storing information on directories]
- *-s* [disk size]
- *-b* [block size]

Your program should start by instantiating the file system based on the input files and input parameters. Once the file system structure has been created, your program should accept User Commands (defined at the end of this assignment) until the *exit* command is issued by the user.

Generating Input Files (for –f and -d parameters)

This step is used to construct the input files for *-f* and *-d* parameters. The input files are the results of instantiation of an existing directory structure. To instantiate a file system, your program will use information on the directories, and files of a root directory of your choice (symbolic links need not be handled). You can use the Unix *find* command to obtain this information. For more information on the *find* command, type *man find* at the Unix command prompt. The following command generates a list of all the subdirectories (and all the subdirectories of the subdirectories, etc.) under the current directory, and output the list to a file called *dir_list.txt*:

find ./ -type d > dir_list.txt

The *dir_list.txt* is the input for the -d parameter.

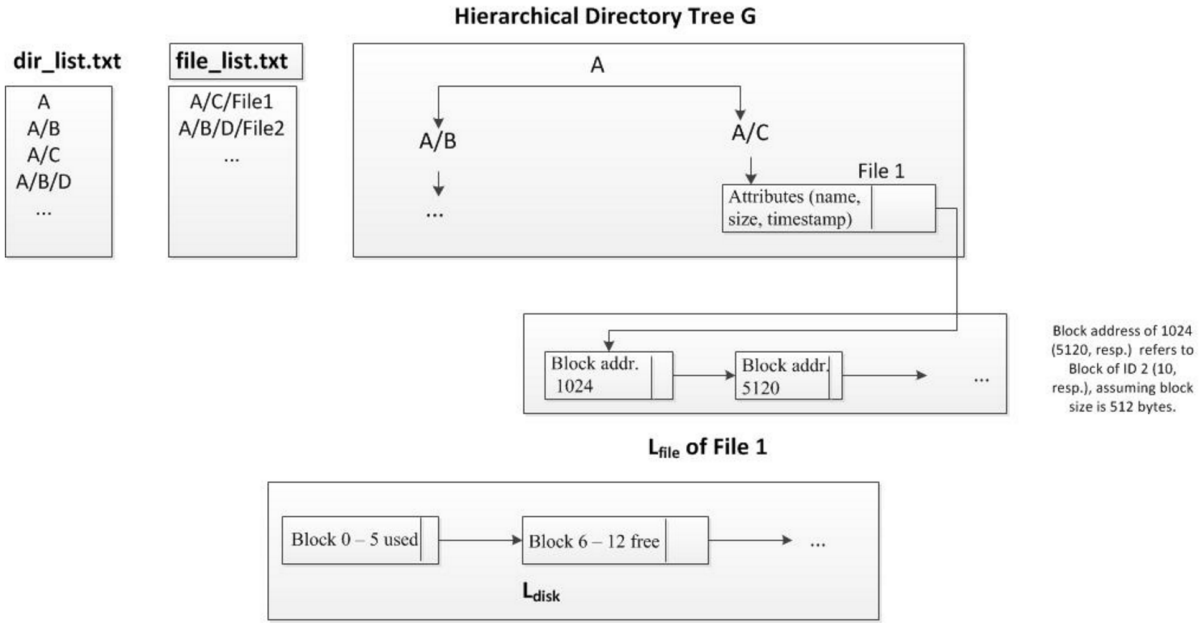
To extract a list of all the files with file size and output it to a file called *file_list.txt*, type at the Unix command prompt:

find ./ -type f -ls > file_list.txt

The *file_list.txt* is the input for the -f parameter. Note that `find ./ -type f -ls` does not just give a list of filenames. It returns other attributes of files too. You need to parse the *file_list.txt* to retrieve the name and size of each file

Hierarchical Directory Tree G

Your program will store the directory structure in a tree data structure with the root node representing the root directory. Each node in the directory tree **G** is associated with either a regular file or a directory. Each node in **G** that corresponds to a regular file, stores the attributes (name, size, and time-stamp) of the file and a pointer to Lfile, the linked-list data structure storing the disk block addresses of the file. An example of the directory tree (and other two data structures) are shown below.



Linked List Ldisk of Disk Blocks

The file system starts at block address 0 with a max size given by the disk size input parameter *–s*. The total number of blocks is equal to the disk size divided by the block size input parameter (specified by parameter *–b*). Block addresses increase sequentially with each block address up to the max number of blocks. For example, with a disk size of 512 bytes, block size of 16 bytes, there are 32 blocks, with block addresses going from 0 to 31. The availability of disk blocks should be stored in a linked-list data structure **Ldisk**, with each node representing a contiguous set of blocks that are free or in use (even if these contiguous set of blocks is allocated to more than 1 file). Each node in **Ldisk** consists of: (1) the set of block IDs (not the physical address of the blocks), (2) the status ("free" or "used") of this set of blocks, and (3) the pointer to the next node in **Ldisk.**

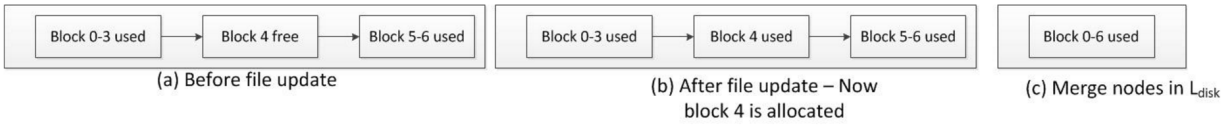
To allocate disk blocks to files, your program should traverse **Ldisk** starting from the first node, looking for the free blocks that can be allocated to the files. For a node in **Ldisk** that contains *x* block IDs with the status "free", the size of the free blocks represented by this node can be calculated as *x * block_size*, where block_size is specified by the parameter -b. If the file needs

less than (x*block_size) bytes, this node will be split into two nodes in **Ldisk**, one for the blocks that are allocated to the file, and the other one for the remaining free blocks.

Disk blocks are allocated to a file from the lowest available address. Blocks of the same file do NOT need to be allocated contiguously. For example, consider a file that requires 4 blocks. Assume that disk blocks 1, 3, 4, 9, 15-18 are available, and the file will be given blocks 1, 3, 4, and 9. The figure above shows an example that a file is allocated with two dis-contiguous blocks, Block 2 & 10.

Disk blocks cannot be split up, so a file will be allocated an entire disk block if it needs any portion of a block.

When you run user commands to update files, the disk allocation must be updated too. You need to make sure that all nodes in **Ldisk** that represent contiguous blocks of the same state ("free" or "used") must be merged to one node in **Ldisk.** The figure below shows an example of merge.



Linked List Lfile for Storing Disk Block Addresses for A File

A linked-list data structure is instantiated for each regular file. For each regular file in your *file_list.txt*, its number of blocks is calculated by dividing its real size (stored in *file_list.txt)* by the size of blocks (specified by parameter *–b*). We assume the disk size specified by the parameter *-s* is large enough to hold all files in your *file_list.txt.*

Each node in **Lfile** represents a file block. It contains: (1) the starting physical address of the block (not the block ID), and (2) a pointer to the next file block. The starting physical address of the block of ID k can be calculated as k * *block_size*, where block_size is specified by the parameter -b.

When files are updated (with new bytes appended and deleted), **Lfile** needs to be updated too**.** Additional nodes are added to **Lfile** as new data is appended to the file. When a file is shortened by some number of bytes, it should be taken away from the end of the file, possibly resulting in the removal of nodes from **Lfile.**

There may be space left over in the last block of the file. When a file is appended with some new bytes, it must fill up the last block before being allocated another file block. For example, if the blocks are of size 16 bytes, and a file uses up only 4 bytes of its last block. Then, when it is appended by 12 bytes, then it will simply fill up the last disk block without the need of allocating another block.

Manipulating Files and Directories via User Commands

Your program must accept the following user commands:

- *cd [directory] - set specified directory as the current directory*
- *cd.. - set parent directory as current directory*
- *ls - list all files and sub-directories in current directory*
- *mkdir [name] - create a new directory in the current directory*
- *create [name] - create a new file in the current directory*
- *append [name] [bytes] - append a number of bytes to the file*
- *remove [name] [bytes] - delete a number of bytes from the file*
- *delete [name] - delete the file or directory*

- *exit - deallocate data structures and exit program*
- *dir - print out directory tree in breadth-first order*
- *prfiles - print out all file information*
- *prdisk - print out disk space information*

For each command, update the hierarchical directory tree, linked-list for disk space management, linked-list for file block management, and time-stamp for the file as follows.

- **When a directory is created:** append a new node to the directory tree **G** at the appropriate branch.
- **When a directory is deleted:** remove the corresponding directory node in the tree **G**; Do NOT allow the directory to be deleted if the directory is not empty. Instead printout an error message, "Directory is not empty"
- **When a file is created:** instantiate a linked-list for the new file (though no disk blocks need to be allocated at this time); add the file to the directory tree **G** as a new node; and update the time-stamp of the file.
- **When a file is appended:** additional disk blocks may need to be allocated for the file (look for first free block in disk space); update the file's linked-list data structure **Lfile**; and update the file's time-stamp. Note that additional disk blocks may not need to be allocated if there's enough space in the last disk block allocated to the file.
- **When a file is shortened:** update file's linked-list **Lfile** by de-allocating disk blocks as specified by the number of bytes to remove; update the file's time-stamp.
- **When a file is deleted:** free the disk blocks allocated to the file; de-allocate the file's linked-list **Lfile**; remove the file node from the directory tree; and update the parent directory's time-stamp.
- **Whenever there is no space** to create or grow a regular file, the program should print the error message "Out of space".

File system Footprint

- **Directory Structure**

Print out the directory structure in breadth-first order when the command dir is issued.

- **File Footprint**

When the command prfiles is issued, printout the file attributes and block addresses for each file in the directory tree. If files were appended, created, deleted, then you should see that the files have not been allocated contiguous space.

- **Disk Footprint**

When the command prdisk is issued, print out the range of block addresses that are in use and the range that are free, ordered by block address (see below for example). Print out the total amount of internal fragmentation (wasted space within a file block that is unused by file).

In use: 0-60
Free: 61-78
In use: 79-92

Free: 93-100
fragmentation: 26 bytes

Implementation Requirements:

- The program must be written in C or C++, and run on a linux machine. (coordinate with your CA)
- ALL source code you submit must be well documented (documentation is an indicator of understanding!)
- Programs that cannot be compiled by CA will receive an automatic grade of zero (0)

Grade Based On:

- Instantiation of existing file system (15%)
- Creation of directory tree (10%)
- Correct use of linked-list to manage disk space (15%)
- Correct implementation of directory creation/deletion commands (20%)
- Correct implementation of regular file creation/deletion/append/remove commands (25%)
- Printing file system footprint (10%)
- Documentation (comments in code) (5%)

Submission:

- Create a zip file of your assignment. The zip file should include your code, file_list.txt, dir_list.txt, and a readme.txt file that contains instructions of how to run your code.
- Submit the zip file using Canvas. Login using your Stevens Pipeline account.
- Ensure your name and login name appear in each file you submit.
- You are advised to commit your contributions to the bitbucket/Github.

Hints and help:

- Use the Unix man command to find out about how to use the various function calls mentioned above.
- Remember to flush out standard out so that you can see your print statements (fflush(stdout))
- Start early. Tackle the problem in units.
- Please don’t hesitate to address questions to the CAs or the instructor.