Problem 1 -- A simple hypothetical filesystem

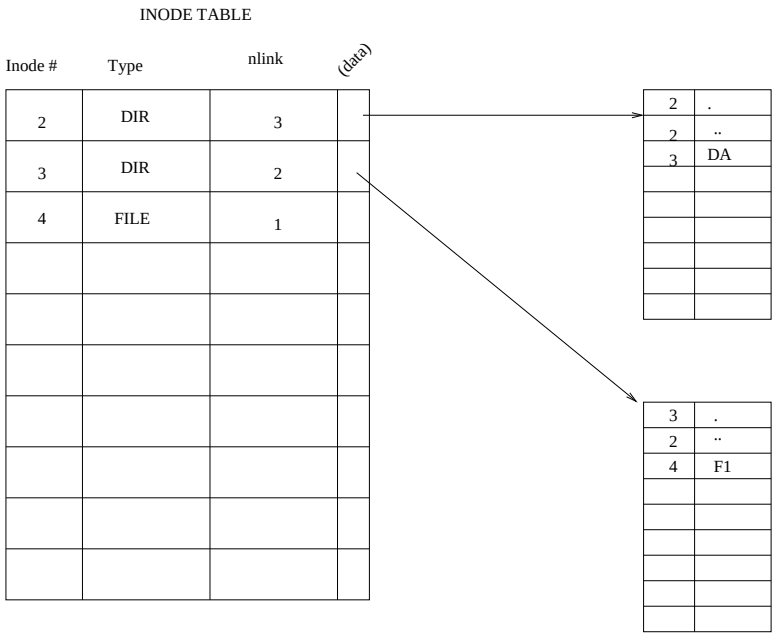
In this problem, you'll trace out the operation of a simple, hypothetical filesystem which follows the principles laid out under "UNIX Filesystems" in the lecture notes. Specifically, we envision this filesystem as having a superblock, inode table, free map and data block areas. To simplify this problem, we can represent the inode "table" schematically, as in the lecture notes, as a table with rows representing inodes and columns the fields of the inode. I.e. we won't concern ourselves with the actual on-disk layout of the inode. We'll also ignore the superblock and the free map. We shall not worry about how the filesystem actually allocates blocks. Directories will be represented as a table of name/inode# pairs. We won't worry about how the directory is actually laid out, or how much disk space it takes. Here are a few more assumptions:

- The first inode is #2 (as is conventional)
- When allocating an inode, the lowest numbered free inode is used (Deleted inode numbers are re-used)
- Directories are in order of directory entry creation time (not sorted by name or inode #, etc.)

Now, let's get started. I initialized a volume and made the first file and the first subdirectory:

```
mkfs /dev/sda1    #Don't try this at home, folks!
mount /dev/sda1 /mnt1
mkdir /mnt1/DA
echo "TEST123" >/mnt1/DA/F1
```

This is what the volume looks like at this point, so you have a clear idea of the representational format:



Now I perform the following operations:

```
mkdir /mnt1/DA/DB
cp /mnt1/DA/F1 /mnt1/DA/DB/F2
ln /mnt1/DA/DB/F2 /mnt1/DA/F3
ln -s /mnt1/DA/DB /mnt1/DC
rm /mnt1/DA/DB/F2
echo "xyz" >/mnt1/DC/F4
```

Sketch the inode table and directories at this point.

Problem 2 -- Exploratory Questions

Answer each of the following with a paragraph or two **in your own words**. After you read the lecture notes, you'll have enough material to answer.

A) A filesystem (volume) that does NOT employ journalling is mounted and is under heavy use, including files and directories being created and removed, when the system suddenly and unexpectedly halts (e.g. kernel crash, hardware failure, power failure). After the system comes back up, we attempt to mount this volume.

i) What program will need to be run before the volume can be mounted?

ii) What sort of issues do we expect this program will find with the volume? Give a specific example of at least one issue.

iii) If this is a 2TB volume with 1,000,000 allocated inodes, will it take a long time? Why or why not?

B) I try to `rm /dir/foo` which is the valid name of a regular file. Give two possible reasons why this would fail with the error "Permission Denied" corresponding to system call error `EACCES` against the `unlink` system call.

C) A user purchases and installs a 4TB SATA disk to hold a collection of 4K-quality vids and installs it on a desktop PC running Linux. Each video file is exactly 2^30 bytes long. Of course, this user follows good security practices and does not download or watch the videos as "root." The expectation was that approx. 4000+ videos would fit on this disk, but the actuality was noticeably less. Give at least two reasons why this would be the case.

D) A user runs the command `mv /A/B/F1 /A/C/F2;` where `F1` is a fairly large file. This command runs very quickly. Then the user runs `mv /A/C/F2 /A/Z/F3;` but this takes quite a while to run. Why might that be the case?

SUBMISSION Submit problems 1 and 2 as a clean, computer-printed hard-copy. No hand-written diagrams, no hand-written corrections!

Problem 3 -- Recursive filesystem lister

The unix `find (1)` command is a powerful tool, at the heart of which is an engine which can perform a recursive walk of the filesystem. Similar functionality is provided by the `ls` command with the `-R` option, or the `ftw(3)` library function. This problem set will require recursively walking a filesystem tree, obtaining the metadata information at each node, and presenting it in a human-friendly format.

Your command will be invoked like this:

```
programname options starting_path
```

Your program will begin the walk at the given starting path name (which could be relative or absolute) and will recursively explore all nodes of the filesystem at and below that point. At each node, it will print out some basic information, which is detailed below.

The options are, in fact optional (for the invoker of the program, not for you) and can be any one or more of the

following (if you don't know how to parse command-line arguments in C, take a look at `getopt(3)`:

`-m mtime`: If mtime is a positive integer, only list nodes which have NOT been modified in at least that many seconds, i.e. they are at least that many seconds old. If mtime is negative, only list nodes which have been modified no more than -mtime seconds ago. Note that `find` syntax expresses this in terms of days, not seconds.

`-v`: When this option is specified, stay within the same mounted filesystem (we called this a "volume" in class) as the one in which your traversal began. Do not descend into the other volume. Print a message to `stderr`, such as illustrated in the example output below.

At each node which is to be listed, your program will output one line in a format *similar* (it need not be absolutely identical) to the output of the command `find -ls`

```
$ /tmp/rls3 -v /
        2      0 drwxr-xr-x  22  root     root          720 Sep 17 22:17 .
   145460      0 lrwxrwxrwx   1  root     root            5 Sep 17 22:17 ./SAMPLE-SYMLINK ->
/home
   114896      4 -rw-r--r--   1       555      666       17 Sep 17 22:16 ./baduser
   114895      4 -rw-r--r--   1  syslog   root           12 Sep 17 22:16 ./example-file
    14816      0 drwx------   3  root     root           80 Sep 17 21:44 ./.cache
Can't open directory ./.cache:Permission denied
    10485      4 drwxr-xr-x   2  root     root           40 May  8  2018 ./modules
   154093      4 -rwxr-xr-x   1  bin      root           19 Sep 17 22:16 ./modules/example2
        1      0 dr-xr-xr-x 183  root     root            0 Sep  9 13:51 ./proc
note: not crossing mount point at ./proc
       21      0 drwxr-xr-x   4  root     root           60 Sep  9 18:16 ./home
note: not crossing mount point at ./home
       18      2 drwxr-xr-x  15  root     root         2048 Jul 21  2018 ./usr
note: not crossing mount point at ./usr
        1      0 dr-xr-xr-x  12  root     root            0 Sep  9 13:51 ./sys
note: not crossing mount point at ./sys
    15412      0 drwxrwxrwt   6  root     root          220 Sep 17 22:26 ./tmp
note: not crossing mount point at ./tmp
     1025      0 drwxr-xr-x  22  root     root        19260 Sep  9 17:51 ./dev
note: not crossing mount point at ./dev
```

The following items are required:

- The inode number of the node.
- The number of disk allocation blocks that the file consumes, expressed in units of 1K (1024).
- The type of node and its permissions mask, in the format which `ls` uses. Read the `ls` man pages carefully and refer to lecture notes. Consider all of the cases (e.g. the set-gid bit is on but the group execute bit is off). **Do not cheat** by using the `strfmode` library function. Decode the bits yourself.
- The number of links to the node
- The owner (user) of the node. Print this out as a name, but if there is no integer->name translation available, print it out as an integer. Look at the man page `getpwuid(3)`.
- The group owner of the node. Similar deal as above, `getgrgid(3)`.
- The size of the node, in bytes. For BLOCK SPECIAL or CHAR SPECIAL device nodes, the size field doesn't make any sense, so print the raw device number as `major,minor` with these numbers expressed in decimal, the

same as the find or ls command output. Look at the man pages for `makedev(3)` or `major(3)`.
•The modification time of the node. Use any reasonable format for printing out the date/time of this timestamp, as long as it is correct! The ls and find commands use a different format depending on whether the time is "recent" or not. The library function `strftime(3)` could be useful to you here.
•The full path name of the node. If the starting path of the program was specified as a relative pathname, then display a relative pathname. I.e. it is not necessary to translate a relative pathname into an absolute pathname. Note that `find -ls` gives the full (relative) pathname while `ls` usually gives just the component name.
•If the node is a symbolic link, append the string " -> " and the contents of the link (via `readlink(2)`). This behavior is what find and ls both do.

Robustness & errors: If you encounter a filesystem which is "deeper" than the maximum number of open file descriptors (typically 1024) then you'll have a problem. The real find/ls have ways to handle this which are not germane to this assignment. However, you should properly detect and report any system call or library errors. It is acceptable to exit after a fatal unexpected error, but bear in mind that the following conditions are not actually errors and should not result in termination: Being unable to translate an integer uid/gid to a name, getting an EACCES error trying to open a subdirectory (just skip it).

Your code should be free from "leaks" such as failure to `free` storage that you allocated with `malloc` (if applicable), and failure to close directories.

Note that the order in which the directory entries are printed is not necessarily sorted. It is simply the order in which `readdir` returns the entries. Along with that, the order of recursion (e.g. depth-first) is not defined. Subdirectories are recursively descended as they are encountered (as opposed to, for example, exploring all the subdirectories first before listing the contents of the current directory).

You aren't required to detect loops in the filesystem tree, which after all could only happen if someone managed to create a forbidden hard link to a directory.

Submission: the program source code, some screenshots showing the program running. You can trim the output to show just a few meaningful examples.

Extra Credit (+1 pt)

For 1 point of extra credit, implement the following additional option

```
-u user
```

This creates an additional filter which will only print those nodes which would be readable by the user specified as `user`. Note that this could be a name which is a known user on this system (look at the `getpwnam` library function) or it could be a decimal integer uid. To evaluate whether a node could be opened for reading, you must replicate the 3-part test that the UNIX kernel performs for a given node (this is described in the lecture notes). In order to do this, you'll need to use the library function `getgrouplist` to retrieve the group membership list for the specified user.