## Problem 1 -- Filesystem scavenger hunt

You will write a program which will be given the name of an existing regular file (which is assumed to be readable by the invoking user), and a starting point pathname in the filesystem (which is assumed to be a directory name). It will recursively explore all of the nodes in the filesystem under that starting point, looking for files which are identical to the first file. Only inodes which are Regular Files or Symlinks should be examined.

We will define "identical" as follows: Two files are identical if they have the same number of bytes (size) and each and every byte of one file is the same as the byte at the same offset in the second file. Differences in metadata do not impact "identical". You do not need to consider "sparse allocation" in your determination of identical.

When you find a "hit" you will print out (to standard output) the following:

* The pathname of the matching file. If the starting point was given as an absolute pathname, your output will also be an absolute pathname, but if the starting point was a relative pathname, you are not being asked to translate that into an absolute pathname (doing so can be non-deterministic and requires use of a library function that will make a lot of system calls)

* If the target file has a link count >1 and you have encountered a hard link to the original target, output that fact. (In this case, you will also wind up outputting a "match" if your search area includes the original target. Don’t worry about this spurious output).

*Otherwise, if this matching file is a distinct inode from the target file, output the nlink value of the matching file. If nlink>1 it is possible that you will see this same matching inode multiple times during your search. You are not required to distinguish between multiple matches of distinct inodes and multiple links to a matching inode that is distinct from the original target inode.

* A yes/no answer to the following question: could this matching file be read by an "other" user who happens to know the pathname of the matching file? The invoking user of your program is not necessarily an "other" (so don’t try to use `access`). Be sure to consider if the "other" user has traversal ("x") permission over the directories involved.

If you encounter a symlink inode, there are two possible matching scenarios:

(1) The symlink resolves to the original target file inode. If so, report this.

(2) The symlink resolves to a distinct inode which happens to be identical in content to the target inode. If so, report this and also report the contents of the symlink. You may or may not see the inode to which the symlink resolves before or after seeing the symlink to it, depending on your search area.

Determining the "readable by other" question for symlinks is complicated and therefore you aren’t required to do this.

Also, remember when doing your recursion, do NOT follow symlinks, otherwise you could potentially wind up in an endless loop.

**ERRORS:** Distinguish between critical and non-critical errors. E.g. if you are not able to open or stat the target file, this is a critical error, since there is no meaningful way to proceed. On the other hand, if you encounter an error opening a directory during your traversal, this should not be fatal. You simply will not be able to explore that part of the filesystem. Report all errors, fatal or warning, with proper descriptive error messages containing the 4 basic pieces of information as discussed in Unit #1. Error reports go to standard error, of course.

**Efficiency:** Do not perform unnecessary reads on a potential matching file. Think about ways to disqualify an inode as being a potential match without having to read the contents. For this assignment, you are permitted to use any system calls or stdio library functions to read the target and prospective matching files for comparison. Certain approaches might be more time efficient than others.

**Binary**: Your program must work with any file, including "binary" files (such as executables and media files) that contain non-printable characters. (Do you understand the distinction between strncmp and memcmp, for example?)

It is expected that you will use the standard library functions `opendir`, `readdir` and `closedir` to traverse the directories, and other system calls or library functions covered in Units #1 and #2 to perform the comparisons. If you are using functions that we haven't covered, it is possible that you are making this too complicated. You are allowed to use ordinary standard library functions such as `fprintf`, `strlen` etc. You are **not allowed** to "shell out" to any external program. Any questions on what is permissible: please ask.

During the coding and testing of this program, you might find it helpful to insert additional debugging output, such as the pathname of each inode that you are considering. If you do this, please make sure to turn off the debugging output before making your final submission!

Below is some sample output. **Your program is not required to produce identically formatted output!** This is just an example to illustrate the results and the information that is expected.

```
$ ./hunt testtarget  ..
DEBUG: Target is 13432 bytes long, dev 908 ino 14819843
Warning: Can’t open directory ../secret:Permission denied
../week2/hardlink       HARD LINK TO TARGET     OK READ by OTHER
../week2/candidate1     DUPLICATE OF TARGET (nlink=2)   OK READ by OTHER
../week2/testtarget     HARD LINK TO TARGET     OK READ by OTHER
../week2/candidate3     DUPLICATE OF TARGET (nlink=2)   OK READ by OTHER
DEBUG: ../week2/crazylink links to something not a file, skipping
../week1/goodlink       SYMLINK RESOLVES TO TARGET
../week100/badlink1     SYMLINK (../week2/candidate1) RESOLVES TO DUPLICATE
../week100/badlink2     SYMLINK RESOLVES TO TARGET
../week100/badlink3     HARD LINK TO TARGET     NOT READABLE by OTHER
```

## Problem 2 -- Extra Credit Filesystem corruption and recovery-- 1pt

Attach screenshots (or terminal session text pastes) as you perform the following system maintenance tasks. Note: you must do these as **root** (uid 0) on a real UNIX system, not Cygwin. The examples below are specific to Linux. Be very careful since an error in one of these commands could destroy data on your hard disk! There are numerous online resources to help you with this process.

1) Obtain a USB memory stick or USB external hard disk that you don't care about.

2) Attach this device to your system. Using the `dmesg` command you should see some system log messages which reveal which device node the kernel has assigned to the drive. A typical series of log messages is below:

```
[1309116.520803] usb 1-4: new high-speed USB device number 3 using ehci-pci
[1309116.648472] usb 1-4: New USB device found, idVendor=154b, idProduct=005b
[1309116.648475] usb 1-4: New USB device strings: Mfr=1, Product=2, SerialNumber=3
[1309116.648477] usb 1-4: Product: USB 2.0 FD
[1309116.648479] usb 1-4: Manufacturer: PNY Technologies
[1309116.648481] usb 1-4: SerialNumber: AA00000000000033
[1309116.648836] usb-storage 1-4:1.0: USB Mass Storage device detected
[1309116.648947] scsi10 : usb-storage 1-4:1.0
[1309118.001013] scsi 10:0:0:0: Direct-Access     PNY      USB 2.0 FD       1100 PQ: 0 ANSI:
[1309118.001181] sd 10:0:0:0: Attached scsi generic sg8 type 0
[1309118.002042] sd 10:0:0:0: [sdh] 63901440 512-byte logical blocks: (32.7 GB/30.4 GiB)
[1309118.002862] sd 10:0:0:0: [sdh] Write Protect is off
[1309118.002864] sd 10:0:0:0: [sdh] Mode Sense: 43 00 00 00
[1309118.003578] sd 10:0:0:0: [sdh] No Caching mode page found
[1309118.003580] sd 10:0:0:0: [sdh] Assuming drive cache: write through
[1309118.007737]  sdh: sdh1
[1309118.010834] sd 10:0:0:0: [sdh] Attached SCSI removable disk
```

Verify that the description of the device matches what you just inserted! Note that the kernel has assigned `/dev/sdh` to the overall disk, and that there is one partition `/dev/sdh1`.

3) Make a Linux EXT2 filesystem on the disk (OK to use either the partition or the entire disk). Make sure journalling is NOT enabled in this case. Show the commands and responses.

4) Mount the disk. Show the output of the `mount` commands (without any arguments) which shows the mounted volume. Run a test program to generate endless disk metadata activity, e.g. repeatedly create, rename and delete files. If this test program produces a lot of debugging output you don't need to include that in the screenshots.

5) Disconnect the disk while it is active. CAUTION: this may hang your system for a while, and may require a reboot to recover.

6) Reconnect the disk. Run fsck to repair the filesystem (note, you may have to use `fsck.ext2` specifically, depending on your distribution, as it may attempt to fsck the disk as a FAT filesystem first). You should see at the very least that the filesystem was "not cleanly unmounted." If you've generated enough corruption from your interrupted activity, you may see other errors as well. Answer "y" to allow repairs to take place. Now mount it and see if any files have appeared in `lost+found`. Show all these commands and outputs. Umount the filesystem. We

are done with this first test.

7) Make a new Linux EXT3 or EXT4 filesystem on the disk (overwriting the previous filesystem). Make sure journalling IS enabled this time.

8) Mount the filesystem and perform your steps 4 and 5 again.

9) Reconnect the disk and attempt to mount it. Verify from the logs that the journal was recovered. Show (highlight) the relevant lines of the dmesg logs relating to journal recovery.