CS360 Lab 4 -- tarc and tarx

- Developed by: Master Jim Plank
- Lab written in February, 2019.

These are your first real systems program. They will mimic the functionality of the program **tar**, which bundles up files and directories into a single file, and which can then use that file to recreate the files and directories.

There are two parts, **4A** and **4B**, each should require a week's work. Please budget your time accordingly.

- **Lab #4A**: You are to write the program **tarc**, which stands for "tar create", and works similarly to "tar cf".
- **Lab #4B**: You are to write the program **tarx**, which stands for "tar extract", and works similarly to "tar xf".

Both of these are explained in great detail below, with examples.

Description of tarc

Tarc takes a single argument on the command line. This should be the name of a directory. It can be an absolute pathname, or a relative pathname. **Tarc** will then create a *tarfile*, which has enough information to recreate the directory and all of its contents. It prints this tarfile on standard output.

The tarfile has a specific format. For each file and directory, it will print the following information:

- The size of the file's name, as a four-byte integer in little endian.
- The file's name. No null character.
- The file's inode, as an eight byte long in little endian.

If this is the first time that we have encountered a file with that inode, then it will print the following additional information:

- The file's mode, as a four byte integer in little endian.
- The file's last modification time, in seconds, as an eight byte long in little endian.

Finally, if the file is a file, and not a directory, then it prints the following final information:

- The file's size, as an eight byte long in little endian.
- The file's bytes.

This is enough information for you to recreate the directory and its contents, including hard links. A few things of note:

- As in the **prsize** lecture, you should ignore "." and "..".
- You should also ignore symbolic links -- if you encounter one, simply omit it from the tarfile.
- Although you don't have to generate your tarfile to be an exact copy of my tarfile, it has to be readable by my programs. More on that below.
- In the tarfile, a directory must appear before any files in that directory. However, otherwise, directories and files may appear in any order.
- If the specified directory is a relative or absolute pathname with the '/' character in it, then you need to split that pathname into a *prefix* and a *suffix*. In your tarfile, you should *only* include the suffixes.

- Remember that when you extract files from a tarfile (either a regular tar file or ones created by **tarc**), the inodes will be different from what the were in the tar file.

To explore **tarc** go ahead and work in a fresh directory. Obviously, do this on a lab machine:

```
UNIX> ls
UNIX>  # First, unpack d1.tar in the lab directory.  This is a regular tar file.
UNIX> tar xpfv /home/jplank/cs360/labs/lab5/d1.tar
d1/
d1/f1.txt
d1/f2.txt
d1/f3.txt
d1/sub/
d1/sub/f1.txt
UNIX> # Next, let's look at the contents of d1.  You can see f1.txt, f2.txt and f3.txt.
UNIX> # f1.txt and f3.txt are hard links.  F2.txt and sub/f1.txt have different protections:
UNIX> ls -li
total 0
894388406 drwxr-xr-x. 3 jplank loci 55 Feb 16 10:00 d1
UNIX> ls -li d1
total 12
894428207 -rw-r--r--. 2 jplank loci  3 Feb 13 17:00 f1.txt
894750231 -rw-------. 1 jplank loci  6 Feb 14 08:51 f2.txt
894428207 -rw-r--r--. 2 jplank loci  3 Feb 13 17:00 f3.txt
963609657 drwxr-xr-x. 2 jplank loci 19 Feb 14 09:44 sub
UNIX> ls -li d1/sub
total 4
963609658 -r--------. 1 jplank loci 5 Feb 15 18:04 f1.txt
UNIX>  # Let's see what's in those fx.txt files:
UNIX> grep . d1/f?.txt
d1/f1.txt:Hi
d1/f2.txt:There
d1/f3.txt:Hi
UNIX> cat d1/sub/f1.txt
THOR
UNIX>
```

Now, I'll create a tarfile using **tarc** in the lab directory, and I'll examine its contents using **xxd** (read the man page).

```
UNIX> /home/jplank/cs360/labs/lab5/tarc d1 > d1.tarc
UNIX> xxd -g 1 d1.tarc
0000000: 02 00 00 00 64 31 b6 48 4f 35 00 00 00 00 ed 41  ....d1.HO5.....A
0000010: 00 00 78 25 68 5c 00 00 00 00 09 00 00 00 64 31  ..x%h\........d1
0000020: 2f 66 31 2e 74 78 74 2f e4 4f 35 00 00 00 00 a4  /f1.txt/.O5.....
0000030: 81 00 00 71 93 64 5c 00 00 00 00 03 00 00 00 00  ...q.d\.........
0000040: 00 00 00 48 69 0a 09 00 00 00 64 31 2f 66 32 2e  ...Hi.....d1/f2.
0000050: 74 78 74 17 ce 54 35 00 00 00 00 80 81 00 00 66  txt..T5........f
0000060: 72 65 5c 00 00 00 00 06 00 00 00 00 00 00 00 54  re\............T
0000070: 68 65 72 65 0a 09 00 00 00 64 31 2f 66 33 2e 74  here.....d1/f3.t
0000080: 78 74 2f e4 4f 35 00 00 00 00 06 00 00 00 64 31  xt/.O5........d1
0000090: 2f 73 75 62 39 84 6f 39 00 00 00 00 ed 41 00 00  /sub9.o9.....A..
00000a0: c5 7e 65 5c 00 00 00 00 0d 00 00 00 64 31 2f 73  .~e\........d1/s
00000b0: 75 62 2f 66 31 2e 74 78 74 3a 84 6f 39 00 00 00  ub/f1.txt:.o9...
00000c0: 00 00 81 00 00 61 45 67 5c 00 00 00 00 05 00 00  .....aEg\.......
00000d0: 00 00 00 00 00 54 48 4f 52 0a                    .....THOR.
UNIX>
```

We should be able to account for every byte in this file. To wit:

- **Bytes 0-3:** Filename size = 0x00000002 = 2.
- **Bytes 4-5:** Filename: 0x64 0x31 = "d1".
- **Bytes 6-13:** Inode: 0x00000000354f48b6 = 894388406.

- **Bytes 14-17:** Mode: 0x000041ed = 100 000 111 101 101. (Directory)
- **Bytes 18-25:** Modification time = 0x000000005c682578 = 1550329208.

- **Bytes 26-29:** Filename size = 0x00000009 = 9.
- **Bytes 30-38:** Filename: "d1/f1.txt".
- **Bytes 39-46:** Inode: 0x00000000354fe42f = 894428207.
- **Bytes 47-50:** Mode: 0x000081a4 = 001 000 000 110 100 100.
- **Bytes 51-58:** Modification time = 0x000000005c649371 = 1550095217.
- **Bytes 59-66:** File size = 0x0000000000000003 = 3.
- **Bytes 67-69:** The bytes: 0x48, 0x69, 0x0a = "Hi\n".

- **Bytes 70-73:** Filename size = 0x00000009 = 9.
- **Bytes 74-82:** Filename: "d1/f2.txt".
- **Bytes 83-90:** Inode: 0x000000003554ce17 = 894750231.
- **Bytes 91-94:** Mode: 0x00008180 = 001 000 000 110 000 000.
- **Bytes 95-102:** Modification time = 0x000000005c657266 = 1550152294.
- **Bytes 103-110:** File size = 0x0000000000000006 = 6.
- **Bytes 111-116:** The bytes: 0x54, 0x68, 0x65, 0x72, 0x65, 0x0a= "There\n".

- **Bytes 117-120:** Filename size = 0x00000009 = 9.
- **Bytes 121-129:** Filename: "d1/f3.txt".
- **Bytes 130-137:** Inode: 0x00000000354fe42f = 894428207.
- Since this file has the same inode as "f1.txt", it's a hard link, we're done.

- **Bytes 138-141:** Filename size = 0x00000006 = 6
- **Bytes 142-147:** Filename: "d1/sub".
- **Bytes 148-155:** Inode: 0x00000000396f8439 = 963609657.
- **Bytes 156-159:** Mode: 0x000041ed = 100 000 111 101 101. (Directory)
- **Bytes 160-167:** Modification time = 0x000000005c657ec5 = 1550155461.

- **Bytes 168-171:** Filename size = 0x0000000d = 13.
- **Bytes 172-184:** Filename: "d1/sub/f1.txt".
- **Bytes 185-192:** Inode: 0x00000000396f843a = 963609658.
- **Bytes 193-196:** Mode: 0x00008100 = 001 000 000 100 000 000.
- **Bytes 197-204:** Modification time = 0x000000005c674561 = 1550271841.
- **Bytes 205-212:** File size = 0x0000000000000005 = 5.
- **Bytes 213-217:** The bytes: 0x54, 0x48, 0x4f, 0x52, 0x0a = "THOR\n".

Spend some time on the list above and how it maps to the bytes of the tarfile. You need to understand this exactly to do the lab.

Now, suppose I specify **d2**, but with a prefix -- "./../d1/../d1" -- that specifies the same directory as before, but in a different way. As you'll notice, the two tarfiles are identical, because the prefix has been stripped off:

```
UNIX> /home/jplank/cs360/labs/lab5/tarc ./../d1/../d1 > d1a.tarc
UNIX> diff d1.tarc d1a.tarc
UNIX>
```

Suppose that you recreate the directory **d1** from its original tar file. You'll see that the names, modes, bytes and modification times are guaranteed to be the same as the first time, but but the inodes will be different. We can see that reflected in the tarfile.

```
UNIX> rm -rf d1 d1a.tarc
UNIX> tar xpfv /home/jplank/cs360/labs/lab5/d1.tar
d1/
d1/f1.txt
d1/f2.txt
```

```
d1/f3.txt
d1/sub/
d1/sub/f1.txt
UNIX> /home/jplank/cs360/labs/lab5/tarc ././d1/../d1 > d1b.tarc
UNIX> diff d1.tarc d1b.tarc
Binary files d1.tarc and d1b.tarc differ
UNIX> xxd -g 1 d1.tarc | head -n 6
0000000: 02 00 00 00 64 31 b6 48 4f 35 00 00 00 00 ed 41  ....d1.HO5.....A // -- inode of d1
0000010: 00 00 78 25 68 5c 00 00 00 00 09 00 00 00 64 31  ..x%h\........d1
0000020: 2f 66 31 2e 74 78 74 2f e4 4f 35 00 00 00 00 a4  /f1.txt/.O5..... // -- inode of d1/f1.txt
0000030: 81 00 00 71 93 64 5c 00 00 00 00 03 00 00 00 00  ...q.d\.........
0000040: 00 00 00 48 69 0a 09 00 00 00 64 31 2f 66 32 2e  ...Hi.....d1/f2.
0000050: 74 78 74 17 ce 54 35 00 00 00 00 80 81 00 00 66  txt..T5........f // -- inode of d1/f2.txt
UNIX> xxd -g 1 d1b.tarc | head -n 6
0000000: 02 00 00 00 64 31 28 14 52 3d 00 00 00 00 ed 41  ....d1(.R=.....A // -- inode of d1
0000010: 00 00 78 25 68 5c 00 00 00 00 09 00 00 00 64 31  ..x%h\........d1
0000020: 2f 66 31 2e 74 78 74 29 14 52 3d 00 00 00 00 a4  /f1.txt).R=..... // -- inode of d1/f1.txt
0000030: 81 00 00 71 93 64 5c 00 00 00 00 03 00 00 00 00  ...q.d\.........
0000040: 00 00 00 48 69 0a 09 00 00 00 64 31 2f 66 32 2e  ...Hi.....d1/f2.
0000050: 74 78 74 2a 14 52 3d 00 00 00 00 80 81 00 00 66  txt*.R=........f // -- inode of d1/f2.txt
UNIX>
```

The mrd program

The program **mrd** is a program that will create a directory full of files and directories that exhibit the range of behaviors that we're expecting you to handle with **tarc**. It is called as follows:

```
usage: mrd seed dir-level fpd randprot(n|y|d) linkperc stem
```

The arguments are:

- **seed** for **srand48()**. Yes, it's a bad RNG, but it's fine for this purpose.
- **dir-level**: this is the maximum number of nested subdirectories. They will always be a chain of directories of length **dir-level**. There may be more than one.
- **fpd**: Maximum files per directory.
- **randprot**: If 'y', set the protection bits of files randomly. The "user read" bit and the "world read" will always be set. If 'd', also set the protection bits of directories randomly. The "user/world read" and "user/world execute" bits will always be set.
- **linkperc** the percentage of times that the program will create hard links to existing files rather than new files.
- **stem** this is the directory into which to put the files.

Here's an example:

```
UNIX> ls
UNIX> /home/jplank/cs360/labs/lab5/mrd
usage: mrd seed dir-level fpd randprot(n|y|d) linkperc stem
UNIX> /home/jplank/cs360/labs/lab5/mrd 2 1 15 y 50 ex1
UNIX> ls -li --full-time ex1
total 52
553947704 -rwx-w-r--. 1 jplank loci 74 2009-07-13 23:07:50.000000000 -0400 BBQEF
553947709 -r-x---r--. 3 jplank loci  2 2009-01-12 15:53:33.000000000 -0500 dBVq4ob3
553947709 -r-x---r--. 3 jplank loci  2 2009-01-12 15:53:33.000000000 -0500 eIPt2ytg1e7gwhmc
553947706 -r-x--xr-x. 3 jplank loci  3 2009-03-31 15:06:37.000000000 -0400 Frhxr
553947707 -r-----r-x. 1 jplank loci 69 2009-10-23 07:25:54.000000000 -0400 HHOSU2o5
553947706 -r-x--xr-x. 3 jplank loci  3 2009-03-31 15:06:37.000000000 -0400 JikTtd0eo
553947710 -r---wxr--. 2 jplank loci 70 2009-05-14 15:08:31.000000000 -0400 kQV5
553947706 -r-x--xr-x. 3 jplank loci  3 2009-03-31 15:06:37.000000000 -0400 mAVDOy
553947710 -r---wxr--. 2 jplank loci 70 2009-05-14 15:08:31.000000000 -0400 sAEuZwlRtaTPX5v
```

```
553947711 -r-----r-x. 1 jplank loci 11 2009-10-05 07:53:15.000000000 -0400 wb31yIuVqOuDZ
553772035 -r-----r--. 1 jplank loci 50 2009-01-14 21:20:06.000000000 -0500 yqOzTZXl3SZ94Vkv
553947708 -r-x-w-r-x. 1 jplank loci 45 2009-10-22 00:43:57.000000000 -0400 YTn9k
553947709 -r-x---r--. 3 jplank loci  2 2009-01-12 15:53:33.000000000 -0500 ZUl
UNIX>
```

As you can see, it created a directory **ex1** with 13 files, many of which are hard links to each other (e.g. **dBVq4ob3** and **eIPt2ytg1e7gwhmc**). The bytes in the files are random, BTW, so if you want to look at them, use **xxd**.

One of the nice things about **mrd** is that if you give it the same parameters, it always creates the same directory structure with the same file/directory names and the same modification/access times. Therefore, you can use it to help test your **tarc** programs.

In the following example, I specify random protections on directories too:

```
UNIX> /home/jplank/cs360/labs/lab5/mrd 2 2 5 d 50 ex2
UNIX> ls -li ex2
total 4
846068234 dr-x-w-r-x. 2 jplank loci 45 Apr 21  2009 iqOuDZX
674147657 dr-xrw-r-x. 2 jplank loci 26 Jun 17  2009 JfHOSU2o56Nxb31y
895575575 dr-x---r-x. 2 jplank loci 51 Jun 25  2009 kQV5
615228457 -r-----r--. 2 jplank loci 73 Apr  9  2009 S0eo1jYBQ
756542011 drwx-w-r-x. 2 jplank loci  6 Oct 29  2009 XxUlqOhUik
UNIX> ls -li ex2/iqOuDZX
total 8
615228457 -r-----r--. 2 jplank loci 73 Apr  9  2009 eIPt2ytg1e7gwhmc
895575577 -r--r--r-x. 3 jplank loci 15 Sep 17  2009 zRtaTPX5v
UNIX> ls -li ex2/JfHOSU2o56Nxb31y
total 4
895575577 -r--r--r-x. 3 jplank loci 15 Sep 17  2009 gdBVq4ob376lA
UNIX> ls -li ex2/kQV5
total 12
895575577 -r--r--r-x. 3 jplank loci 15 Sep 17  2009 al3SZ94Vkv3aOCTn
895575578 -r--r-xr-x. 2 jplank loci 52 May  4  2009 DVD
895575578 -r--r-xr-x. 2 jplank loci 52 May  4  2009 tZUqO
UNIX> ls -li ex2/XxUlqOhUik
total 0
UNIX>
```

You'll note that **ex2/S0eo1jYBQ** and **ex2/iqOuDZX/eIPt2ytg1e7gwhmc** are hard links to each other.

One thing about these directories -- when the protection has changed so that the write bit isn't set, you can't delete them, even with "rm -rf", without changing the protection back. For example, if I try to delete **ex2**, I get an error:

```
UNIX> rm -rf ex2
rm: cannot remove â€˜ex2/JfHOSU2o56Nxb31y/gdBVq4ob376lAâ€™: Permission denied
rm: cannot remove â€˜ex2/iqOuDZX/eIPt2ytg1e7gwhmcâ€™: Permission denied
rm: cannot remove â€˜ex2/iqOuDZX/zRtaTPX5vâ€™: Permission denied
rm: cannot remove â€˜ex2/kQV5/al3SZ94Vkv3aOCTnâ€™: Permission denied
rm: cannot remove â€˜ex2/kQV5/tZUqOâ€™: Permission denied
rm: cannot remove â€˜ex2/kQV5/DVDâ€™: Permission denied
UNIX>
```

To delete it, I need to first recursively set the protections to something I can delete:

```
UNIX> chmod -R 0755 ex2
UNIX> rm -rf ex2
UNIX>
```

The tarf program

You call **tarf** in one of two ways:

```
tarf < tarfile       -- Prints out information about the tarfile.
tarf debug < tarfile -- Also prints out information while reading the tarfile.
```

The information that **tarf** prints out is sorted by filename. Here's an example. I'm going to recreate **d1** again, so that it will have different inodes than before:

```
UNIX> tar xpfv /home/jplank/cs360/labs/lab5/d1.tar
d1/
d1/f1.txt
d1/f2.txt
d1/f3.txt
d1/sub/
d1/sub/f1.txt
UNIX> /home/jplank/cs360/labs/lab5/tarc d1 > d1.tarc
UNIX>
```

Now, I call **tarf** on it, and it prints out information about all of the files/directories, sorted by inode:

```
UNIX> /home/jplank/cs360/labs/lab5/tarf < d1.tarc
Inode 362606622 Mode 40755 Hash 00000000 Mtime 1550329208
Name: d1

Inode 362606623 Mode 100644 Hash 0b87f2c0 Mtime 1550095217
Name: d1/f1.txt
Name: d1/f3.txt

Inode 362606624 Mode 100600 Hash d363a247 Mtime 1550152294
Name: d1/f2.txt

Inode 425644584 Mode 40755 Hash 00000000 Mtime 1550155461
Name: d1/sub

Inode 425644585 Mode 100400 Hash 0e17e06c Mtime 1550271841
Name: d1/sub/f1.txt

UNIX>
```

You see that the two hard links are printed together. Also, I calculate a hash of each file's contents. Directories have hash values of zero. **Tarf** is nice because if you have two tarfiles for the same directory, even though the tarfiles may differ (because of the order in which you specify files and directories), the output of **tarf** will be the same. If I'm bored someday, I'll demonstrate it. For now, just accept that it's true.

If you call **tarf** with "debug", it prints out the information as it reads it in from the tarfile. This may be helpful to you for debugging when your tarfiles have errors.

The gradescript for this part of the lab is `gradescript-5a`, and the gradeall is `gradeall-5a`. The gradescript goes through the following steps for gradescript *i*.

- It calls my **tarc** on the file in **Gradescript-Examples/*i***. It puts the tarfile into **tmp-*i*-correct-tarfile.txt**.
- It calls your **tarc** on the file in **Gradescript-Examples/*i***. It puts the tarfile into **tmp-*i*-your-tarfile.txt**.
- When it makes that call, it limits you to ten file descriptors, so be aware of that. (It does that by calling your program with the program `limit-shell`) in the lab directory. You can read the source code of that if you care to.
- If your program had an error, it will stop there.
- It calls **tarf** on my tarfile. That goes into **tmp-*i*-correct-stdout.txt**.
- It calls **tarf** on your tarfile. That goes into **tmp-*i*-your-stdout.txt**.
- It tests to see that those two files are identical.

On the issue of open files -- I'm limiting your process to 10 open files, because I want you to pay attention to keeping that number low. For that reason you need to:

- Call **fclose()** when you're done with a file that you have opened with **fopen()**.
- Call **closedir()** when you're done with a directory that you have opened with **opendir()**.
- Test the return values of **fopen()** and **opendir()**, and if they are NULL, then use **perror()** to print the error, and then exit. This will save you countless hours of headaches on segmentation violations.

If you want to limit the number of file descriptors in your **tarc.c** program to ten, to help you debug, then you can put the following into your code **/home/jplank/junk.txt**

```
#include <sys/time.h>
#include <sys/resource.h>

/* ..... then, inside your main(): */

  int i;
  struct rlimit l;

  i = getrlimit(RLIMIT_NOFILE, &l);
  l.rlim_cur = 10;
  i = setrlimit(RLIMIT_NOFILE, &l);
```

You don't have to do this, because my gradescript does it using **limit-shell**. You also cannot raise your file descriptor limit in this way. Sorry.

Description of tarx

Tarx recreates a directory from a tarfile, including all file and directory protections and modification times. It reads the tarfile from standard input.

That's really all of the description **tarx** needs. But I'll walk you through an example. Take a look at the directory **test-dir**, in the lab directory:

```
UNIX> ls -li /home/jplank/cs360/labs/lab5 | grep test-dir
963647548 drwxr-xr-x.   4 jplank loci    79 Dec 25  2009 test-dir
UNIX> ( cd /home/jplank/cs360/labs/lab5 ; ls -li test-dir )
total 12
 963647550 -r-x-wxr--. 2 jplank loci 30 Apr  8  2009 lTFYSC
1016031860 drwx--xr-x. 2 jplank loci 72 Aug 23  2009 Md1jUzS6
 963647551 -rw----r-x. 1 jplank loci  3 Feb 12  2009 sOdKRmzyPT3mXX4
  15975445 dr-xr--r-x. 2 jplank loci 79 Jul  5  2009 T7ZdQl
 963647549 -r-x-w-r--. 1 jplank loci 32 Sep 29  2009 TkB
UNIX> ( cd /home/jplank/cs360/labs/lab5 ; ls -li test-dir/Md1jUzS6 )
total 16
1016031862 -rwxr--r-x. 1 jplank loci 78 Oct 16  2009 aA4X7tu6u
1016031861 -r--r--r--. 1 jplank loci 17 Sep  7  2009 QFIoNAT
  15975447 -r--rw-r--. 2 jplank loci  8 Apr 18  2009 tBSlghe
1016031863 -r-----r--. 1 jplank loci 71 Jul 14  2009 ymSZrTV0u0vrAHL
UNIX> ( cd /home/jplank/cs360/labs/lab5 ; ls -li test-dir/T7ZdQl )
total 16
 15975447 -r--rw-r--. 2 jplank loci  8 Apr 18  2009 aDH
 15975448 -r-x--xr-x. 1 jplank loci 53 May 20  2009 i4obCmgB1ZHpDy7q
 15975446 -r---wxr--. 1 jplank loci 32 Oct 10  2009 JJbBXDSpYO23pb6-
963647550 -r-x-wxr--. 2 jplank loci 30 Apr  8  2009 RCPdAWLBsz
UNIX>
```

This is a challenging directory from a **tarx** perspective. But first, let's go ahead and create a **tarc** file, and call **tarf** on it to verify our understanding of it. Do this in a fresh directory:

```
UNIX> ls
UNIX> /home/jplank/cs360/labs/lab5/tarc /home/jplank/cs360/labs/lab5/test-dir > td.tarc
UNIX> xxd -g 1 td.tarc | head -n 2
0000000: 08 00 00 00 74 65 73 74 2d 64 69 72 3c 18 70 39  ....test-dir<.p9
0000010: 00 00 00 00 ed 41 00 00 95 8b 35 4b 00 00 00 00  .....A....5K....
UNIX> /home/jplank/cs360/labs/lab5/tarf < td.tarc | head -n 15
Inode 963647548 Mode 40755 Hash 00000000 Mtime 1261800341
Name: test-dir

Inode 1016031860 Mode 40715 Hash 00000000 Mtime 1251081491
Name: test-dir/Md1jUzS6

Inode 1016031861 Mode 100444 Hash 9802f4b8 Mtime 1252360768
Name: test-dir/Md1jUzS6/QFIoNAT

Inode 1016031862 Mode 100745 Hash 2a0b3a18 Mtime 1255692354
Name: test-dir/Md1jUzS6/aA4X7tu6u

Inode 15975447 Mode 100464 Hash d629e03e Mtime 1240070091
Name: test-dir/Md1jUzS6/tBSlghe
Name: test-dir/T7ZdQl/aDH
UNIX>
```

As you can see, the files **test-dir/Md1jUzS6/tBSlghe** and **test-dir/T7ZdQl/aDH** are hard links to each other. Also, to repeat myself, the bytes in the files are random, so don't try to **cat** them. Instead, use **xxd**:

```
UNIX> xxd -g 1 /home/jplank/cs360/labs/lab5/test-dir/lTFYSC
0000000: d3 44 27 cf a1 b6 06 4b ca 03 e2 fc b8 2f 34 67  .D'....K...../4g
0000010: a8 4c c6 9e 94 5e 3e 0d 6c 80 d6 99 e1 92        .L...^>.l.....
UNIX>
```

What makes this directory a challenge is the subdirectory **test-dir/T7ZdQl**. You'll note that its write protection is not set. For that reason, you cannot set its mode until after you have created all of its files and subdirectories. That's not a difficult matter, so long as you pay attention to it.

I'll call my **tarx** on the tarfile, and you'll see that it recreates the files and directories with the same protection modes and modification times:

```
UNIX> /home/jplank/cs360/labs/lab5/tarx < td.tarc
UNIX> ls -li
total 4
756586508 -rw-r--r--. 1 jplank loci 1018 Feb 17 10:49 td.tarc
492968489 drwxr-xr-x. 4 jplank loci   79 Dec 25  2009 test-dir
UNIX> ls -li test-dir
total 12
492968491 -r-x-wxr--. 2 jplank loci 30 Apr  8  2009 lTFYSC
554271758 drwx--xr-x. 2 jplank loci 72 Aug 23  2009 Md1jUzS6
492968492 -rw----r-x. 1 jplank loci  3 Feb 12  2009 sOdKRmzyPT3mXX4
654220282 dr-xr--r-x. 2 jplank loci 79 Jul  5  2009 T7ZdQl
492968490 -r-x-w-r--. 1 jplank loci 32 Sep 29  2009 TkB
UNIX> ls -li test-dir/Md1jUzS6
total 16
554271761 -rwxr--r-x. 1 jplank loci 78 Oct 16  2009 aA4X7tu6u
554271760 -r--r--r--. 1 jplank loci 17 Sep  7  2009 QFIoNAT
554271763 -r--rw-r--. 2 jplank loci  8 Apr 18  2009 tBSlghe
554271762 -r-----r--. 1 jplank loci 71 Jul 14  2009 ymSZrTV0u0vrAHL
UNIX> ls -li test-dir/T7ZdQl
total 16
554271763 -r--rw-r--. 2 jplank loci  8 Apr 18  2009 aDH
610087472 -r-x--xr-x. 1 jplank loci 53 May 20  2009 i4obCmgB1ZHpDy7q
```

```
610085407 -r---wxr--. 1 jplank loci 32 Oct 10  2009 JJbBXDSpYO23pb6-
492968491 -r-x-wxr--. 2 jplank loci 30 Apr  8  2009 RCPdAWLBsz
UNIX> xxd -g 1 test-dir/lTFYSC
0000000: d3 44 27 cf a1 b6 06 4b ca 03 e2 fc b8 2f 34 67  .D'....K...../4g
0000010: a8 4c c6 9e 94 5e 3e 0d 6c 80 d6 99 e1 92        .L...^>.l.....
UNIX>
```

The Gradescript

The gradescript for this part of the lab is `gradescript-5b`, and the gradeall is `gradeall-5b`. The gradescript goes through the following steps for gradescript *i*.

- It calls your **tarx** on the file **Gradescript-Examples/*i*.tarc**. That should create the directory *i* in the current directory. It also puts stdout and stderr into **tmp-*i*-your-stdout.txt**. and **tmp-*i*-your-stderr.txt**.
- It limits the number of file descriptors to 10 in the **tarx** call.
- It calls **tarf** on **Gradescript-Examples/*i*.tarc**, stripping out the inodes from stdout. It puts stdout and stderr into **tmp-*i*-correct-stdout.txt**. and **tmp-*i*-correct-stderr.txt**.
- If one of them flagged an error and the other one didn't (which is checked by seeing if either program wrote into stderr), then it's an error.
- If both flagged errors, then your program is correct, and the gradescript exits.
- Otherwise, it calls my **tarc** on your new directory *i* and flags errors if there are any.
- If no errors, then it calls my **tarc** on your new directory *i* and pipes the output to **tarf**, stripping out the inodes, and putting the output into **tmp-*i*-tarf-output.txt**.
- It tests to see that **tmp-*i*-correct-stdout.txt** are the same **tmp-*i*-tarf-output.txt**.

You should note that the tarfiles may or may not be created by my **tarc**. However, with the exception of gradescripts 41-50, they will be legal tarfiles. With gradescripts 41-50, the tarfiles are bad, and your program must print an error message on standard error and exit. Your error messages do not have to match mine exactly.

Ten Open Files

The gradescripts limit you to ten open files when you run both **tarc** and **tarx**. To test yourself, the simplest way to do this is to do, for example,

```
UNIX> csh -c "limit descriptors 10; ./tarx < td.tarc"
```

That way you execute the command in a sub-shell that is limited to 10 open files, but you don't limit your own shell to 10 open files.

Useful System Calls

You will most definitely need to use the following system calls. Make sure you read their man pages:

- **lstat()**. Obviously, **stat()/lstat()** have their own lecture, so make sure you go through those lecture notes. You'll be making use of the **st_mtime** field of the **struct stat**, which on some machines is not straightforward. Fortunately, on our lab machines, it is straightforward, so I encourage you to read the man page and develop your code on a lab machine. If you don't, just be aware that different machines have different mechanisms regarding **st_mtime**. For example, on my Macintosh (OS X 10.11.6), the **struct stat** does not have **st_mtime**, but instead has:

```
struct timespec st_mtimespec;
```

  Later in the man page, it says:

```
The time-related fields of struct stat are as follows:

st_atime         Time when file data last accessed.  Changed by the mknod(2),
                 utimes(2) and read(2) system calls.

st_mtime         Time when file data last modified.  Changed by the mknod(2),
                 utimes(2) and write(2) system calls.
```

So, what gives? Well, if you look at **/usr/include/sys/stat.h**, you'll see:

```
#define st_mtime st_mtimespec.tv_sec
```

This means that you can use **st_mtime** as if it is a member of the **struct stat**, but it is a little confusing, isn't it?

- **chmod()**. This is nice and straightforward.
- **utimes()**. Ok, so this is the world that we're in:
    - **st_mtime** is a **time_t**, and is in some operating systems' **struct stat** specifications, but not in others (although it's supported by a **#define**).
    - **st_mtimespec** is a **struct timespec**, which, if you look it up, has two fields to deal with sub-second times: **tv_sec** and **tv_nsec** (nanoseconds). Fortunately, that latter one fits into a 32-bit data type (although I'm not overly sure if that's what's used. I would be it's inconsistent).
    - **utimes()** has a second parameter which is a pointer to two **struct timeval**'s. Really? That one has two fields: **tv_sec** and **tv_usec** (microseconds).

  I understand that this is a pain, but hopefully this prose helps you a little. You'll have to create an array of two **struct timeval**'s for **utimes()**. Go ahead and set the first one to the current time (I don't do any testing of the access time in this lab, so you don't have to worry about it). As for the second one, set the seconds to your stored **mtime**, and then set the microseconds to zero. Done.