Lab 3: The UNIX File System

Magnus Johansson

May 9, 2007

1 The assignment

In this assignment you will study the UNIX file system. In particular, you will learn how the file system is structured and which system calls you can use to work with the file system in your code.

First you will be given a few tasks designed to make you familiar with the file system, and then you are to write a program that behaves similar to the shell command `ls -lRa`. As before you are free to write it in any programming language you wish, as long as you can do POSIX system calls from it. **Make sure you read the supplied documentation before you proceed!**

1.1 The tasks

1. `ls` has many options for displaying information about files and directories in various formats. Use `ls`, with options where appropriate, to determine the following:

   - In `/stud/docs/kurs/os/test-lab3/alpha/` there are several identical files.
     Which of them are in fact separate and unique (and which are not)? Explain your answer.
   - Create an empty directory in your work directory. Type `ls -l` and look at the entry.
     Why is there a "2" after the permissions, what is the meaning of the number 2 in this context? (i.e. what are the two items?)
     Hint: it is *not* the number of subdirectories.
   - Copy a file to the new directory, and see what happens to the "2".
     Has it changed? Why or why not?
   - Create a directory in the new directory (i.e. you should now have a directory containing a file and an empty directory). Check the number again.
     What has happened? Explain.

2. Use `ln` to create links and investigate the behaviour of the links in different situations:

   - Change to the new directory and use `ln -s` to create a *symbolic link* to the file.
     Does the original file show any indication of the symbolic link?

     Do the same for the directory and see if there is any indication. Try to make a symbolic link to a file that does not exist, or make a symbolic link and then remove the file.

     What happens?
   - Type `ln -s gurka gurka`.
     What happens when you try to read the file `gurka` you just created? Explain.
   - Now make a *hard link* to a file (i.e. `ln` without `-s`).
     What happens now?
   - What happens when you change the permissions of the original file with `chmod`, or update its modification time with `touch`?
   - Apart from the names, how can you tell which was the original file, and which is the link you just created?

- What happens if you remove the original?
- What happens if you try to make a hard link to a directory? To a non-existent file?

After all this, you should understand the difference between hard and symbolic links.

1.2 Programming assignment

In this part of the assignment you will learn about the `stat(2)` system call. The `stat` system call fills in a structure that contains a number of fields with information from a file's inode. Read the manual page for `stat(2)`. Observe that there are several `stat` functions, use the right one in the right place. Ie. make sure that you report information about the file that *holds* a symbolic link and not the file (if any) that the symbolic link points to.

There are some scripts that will help you a bit. First there is build.sh which you should be familiar with by now. It will compile your code. The second script is create_ls.sh. If you run this, yet another script, called ls.sh, will be created. This script will in turn run your program as long as you called it Ls.java. The reason for this extra step is that, as opposed to the other lab assignments, you might want to run Ls.java from different places in the file system. The scripts in the previous assignments all used relative paths, which worked fine as long as you stayed in the lab directory. This time you will probably want to try your program from other places. Then you need an absolute path, and the script create_ls.sh will create this script for you, tailored to your home directory.

1. Start by implementing the method printInfo(String filename). Given a filename or a directory name, it should print the following information from the corresponding inode:

   - mode (permissions)
   - number of links
   - owner's id
   - group id
   - size in bytes
   - size in blocks
   - last modification time
   - name

   In addition, the type of the object should be indicated as follows: if the file is a symbolic link, the file "pointed to" by the link should appear following the name, such as with `ls -l`, see `readlink(2)`. A character should be added to the end of the filename to indicate if it is a directory or an executable file; one of `'/'` or `'*'` as described for `ls -F`.

   Since the calls `getpwuid(3C)` and `getgrgid(3C)` are not implemented in jtux, you do not have to convert the user id or the group id to strings. That is, your printout should look like the one for `ln -n` rather than `ls -l`

   There is a given method that will convert mode bits to a string similar to the one output by `ls -l`.

   For this part you will need to use the following jtux methods and structures:

   `UFile.s_stat` This is the type of the structure that will be filled in by `stat`. See the man page for stat for details. The Java types used in this structure can be found in the documentation included in the lab package.

   `UFile.lstat(String filename, UFile.s_stat statinfo)` This system call will fill in `statinfo` with information about the inode corresponding to `filename`.

   `UConstant.S_IFLNK` This is a constant that you can use to check if a mode has a the `S_IFLNK` bit set (that is, if it is a symbolic link). You use it as follows:
   `if ((statinfo.st_mode & UConstant.S_IFLNK) == UConstant.S_IFLNK)`

   `UFile.readlink(String path, byte[] buf, int bufsize)` Read about this system call in the man pages. This jtux method has a weird interface. You need to give it a byte array which it will fill in with the contents of the symbolic link. It will return the number of bytes filled in. You will then need to use the appropriate String constructor to convert it to a string.

   In addition you will probably want to use the following Java classes:

`Date` Use this to create a more useable time representation that the one you get from `stat`. Note that the one you get from `stat` is the number of seconds from the beginning of 1970, while the Date constructor expects the number of *milli*seconds from the beginning of 1970. You need to take this into account when using Date.

`SimpleDateFormat` Use this class to get something more readable to print to the screen.

2. Next it's time to implement the method printDir(String directory). Now you will learn how to read directories and traverse the file system. When you're done with this part you should have a program that traverses the file system from a starting point provided on the command line, similar to `ls -lRa`.

   Begin by skimming through some of the appropriate manual pages. The major functions you will need are `opendir(3C)`, `readdir(3C)`, `closedir(3C)`, `chdir(2)`, `getcwd(3C)` and `rewinddir(3C)`.

   The `struct dirent` mentioned on the manual page for `readdir(3C)` is documented in the `dirent(4)` manual page. The `dirent` structure contains a number of fields with information from an entry in a directory.

   In `/stud/docs/kurs/os/test-lab3` there are a number of subdirectories and files of different types. Change to that directory and run `ls -lagFR` to get an idea of the kind of output you should expect from your program. Please note that this directory is not accessible from all campus computers. Use e.g. hamberg.

   A natural structure for the method is to traverse the list of files in a *single* directory. When it reaches a directory in the list, it can call itself recursively. To start, just invoke the method with the name of the starting directory.

   You will need to deal with the possibility that you may not have the proper permissions to search or enter certain directories. There are also some other error-like situations that need to be handled properly. In none of these cases should the program need to exit, although it may need to take some special action. See what `ls` does for example. Notice that there is a hidden file that your program should be able to find without crashing. It is called "you_get_it".

   For this part you will probably need to use the following jtux methods and structures:

   `UProcess.chdir(String path)` This method may throw an `UErrorException` that you may want to handle. Check its method `getCode()` to see what happened. If it happens to be `UConstant.EACCES`, then a permission denied error occurred.

   `UDir.opendir(String path)` This method will open a directory for reading. It may throw the same exception as `UProcess.chdir`, so you may want to deal with it in a similar way. It will return a `long` that is a handle to the directory stream. Initially the stream will be positioned at the first entry in the directory.

   `UDir.s_dirent` This is a structure that will be returned by `UDir.readdir()` (see below). Take a look at the documentation included in the lab package for details of its contents.

   `UDir.readdir(long dirp)` This method will return a structure of type `UDir.s_dirent` that represents the next entry in the directory stream. By making subsequent calls to this method you will step through the contents of the directory represented by `dirp`. When there is no more entries in the directory, `null` will be returned.

   `UDir.rewinddir(long dirp)` This method will rewind the stream represented by `dirp` so that it is positioned at the very first entry in the stream.

   `UDir.closedir(long dirp)` This method will close the directory stream.

   In addition to these, you may want to use a few of the methods from the previous step.

2 How to hand in

Send an email to leon.mugwaneza@it.uu.se with answers to the questions and the source code attached (see submission instructions on the course homepage).