CS4023 Week03 Lab Exercise

Lab Objective: We will use this lab to log in to our Linux accounts and to look at some simple programs that perform a few elementary system calls. By the end of the lab we will have developed the machinery to more or less mimic UNIX's `cp` copy command.

Here's a quick summary of the tasks:

❶ Create a series of hierarchical directories to manage our work throughout the remainder of the semester

❷ Copy the `C` source code `readFile.c` into your directory and compile it

❸ Watch it in action, tracing "system events" using the `strace` and `ptrace` utilities

❹ Learn some of the basic tricks of the good programmer

❺ Extend the given program so that it reads a file and writes the contents somewhere else

❻ See how you should submit your program for automatic grading.

❼ Consider what is wrong with the program we have just worked on.

In Detail

❶ Note that there are a few alternatives given below for this first step so please read the entire instructions before acting. Over the semester we will have labs, programming assignments, etc. and it is a good idea to keep these in an organised way. My suggestion is that you create a subdirectory of your home directory called `cs4023` that will be the top-level point for all module-related material. The command to create a directory in Linux is `mkdir`, so you could do

```
mkdir ~/cs4023
```

which makes a subdirectory located in your home directory. Next you should make a subdirectory called `labs` in this for each week's work. This can be done with

```
mkdir ~/cs4023/labs
```

Finally, for this week's lab, Week03, you should create its own subdirectory with

```
mkdir ~/cs4023/labs/week03
```

An alternative to making each level of the hierarchy at a time is to tell mkdir to make the "parent" subdirectory if it doesn't exist. So the following command can take the place of all of the previous ones.

```
mkdir -p ~/cs4023/labs/week03
```

The `-p` is for "make parents if they don't already exist." Note that it is very similar to the command before it, but you had to do a lot of extra work in order to achieve that.

Yet another alternative is to bring up a GUI-based file manager and use the "Create ..." menu option there.

❷ Using the `C` code provided in the class directory, `~cs4023/labs/week03`, copy this file into your directory `~/cs4023/labs/week03` with

```
cp ~cs4023/labs/week03/readFile.c  ~/cs4023/labs/week03
```

Just a single character...

Here's something that gets people every time. The tilde character, `~`, has a special use at the CLI (command-line interface). Several different shells – including `bash`, which we are using – treat the tilde character as denoting a user's home directory. So, no matter where you are in your own filesystem, you can use `~` as a shorthand for your home directory, which, for user `d9994023` will be `/home/d9994023`. But you can also use `~` to refer to somebody else's home directory. In the previous command `~cs4023` is a shorthand for the home directory of user `cs4023`. So there is a huge difference between `~cs4023` and `~/cs4023`: the former is the home directory of user `cs4023` while the latter is a sub-directory called `cs4023` of `your` home directory.
In summary: watch out for not confusing the two arguments to the `cp` command!

Once this is done you can examine the source code in a text editor. The one I use is `emacs`, which may or not be installed on your lab machine. If it is you can fire it up with

```
emacs readFile.c &
```

but you need to be in the correct directory for this. As an alternative editor many people in the past have preferred `gedit` and this is on your machines. You can *change working directory* with the command

```
cd ~/cs4023/labs/week03
```

The ampersand at the end is important for it runs the program in the background, allowing you to give further commands at the shell prompt.

Please spend a while looking at the code; it shouldn't all make sense to you but between looking at the source and running it (later) you should get an idea of what it does.

One thing to look at is the `while` loop. As you know from your Java days `while` executes the body of the while-loop as long as some condition is true. In this case the "condition"

is decided by comparing the result of "`c = getc(from)`" to the special character `EOF`. This special character exists at the end of every UNIX file and is used, for example, by a disk-reading program to tell when to stop reading. As long as `c` is not this special character the while-loop will keep on truckin' (man). And what is the truckin'? The character is "put" to `stdout`, which is the screen in this case.

Now *compile* this program with the command

```
gcc -o readFile readFile.c
```

Actually, the command `gcc` does two jobs: it firstly compiles the file `readFile.c` into CPU-specific code (*machine* or *object* code) and then *links* together this object code with any necessary support modules of code to create an *executable* program.

Once you have compiled your program you should have an executable called `readFile` that you can run as follows:

```
readFile
```

If you have been following along with the previous labs, doing as was suggested, then the above should work. If it *doesn't* work it is likely that you have not modified your `$PATH` environment variable to instruct the shell to look in the current working directory first for commands to execute. This environment variable should be in either of the files `~/.bashrc` or `~/.bash_profile` in your home directory.

Today's Tip

You can avoid always having to specify, `./`, the current working directory, by a quick edit of your `.bash_profile` in your home directory. In this file you will find a line that begins with "`PATH=`". What follows is a list of directories, each separated from the next by a `:`. Whenever you issue a command, this list of directories or locations is where the shell looks to try and find an executable program. So by putting just a simple `.` at the end of the line will now tell the shell interpreter to look in the current directory as a last resort for a command to execute. 1) Don't forget the `:` separator and, 2) this will work next time you log in; for the present session, issue the command `sh $HOME/.bash_profile`.

When you do get the program to run, it should report back immediately with `file.txt: No such file or directory`. This is because we don't have such a file in this directory. In order to keep `readFile` happy please create a file called `file.txt` now. You can edit this from scratch with anything you like or you can do something like

```
head -5 readFile.c > file.txt
```

which will insert the first 5 lines of your `C` source file into the file `file.txt`. If parsimoniousness is your concern you could even do

```
touch file.txt
```

(Eventhough we mentioned it in class recently this one is worth doing a `man` on. Do it now. That is, at the shell prompt give the command `man touch`. Note that order is important,

this is not the same as `touch man`.[1])

Once you have created something and called it `file.txt` you should be able to run `readFile` and have it exit successfully.

Now we will cause the program to fail and this will demonstrate the power of the `perror()` function call in the program. Change the protections on the file `file.txt` so that it is not readable. You can do this with

```
chmod 0333 file.txt
```

or

```
chmod ugo-r file.txt
```

Now run `readFile` again. Note the different error message. When attempting to open the file `fopen()` returned `NULL` and we then had `perror()` decode the reason why it failed. Please, *please* get in the habit of testing the result of every system call you make – `fopen()` in this case – and reporting the error with `perror()` in the case of failure.

Note also the `fclose()` system call that closes the file opened for reading. Just like your mother told you to always close gates that you opened, so too should you fclose files fopened by you.

Don't forget to change back to something sensible the protections on the file `file.txt` now.

❸ A very useful tool to help in debugging programs that make system calls or otherwise interact with the kernel is the `strace` program. This program and its (simpler and less verbose) counterpart, `ltrace`, allow you to see a report of system calls and library calls – `strace` is for system calls and `ltrace` is for library calls.

They do their job by you giving your command as normal and just prepending either one or the other. For example

```
ltrace readFile
```

reports the library calls made in running our program. It is the program `ltrace` that is executed first and this manages the running of our program `readFile`.

❺ You will now write a new program called `copyFile` that will read the file `file.txt` and now write the contents to the file `copy.txt`. Firstly make a copy of the `readFile.c` source code and save it in `copyFile.c`. Now edit[2] this file so that the copying of `file.txt` gets done. This involves two main steps:

- opening – and *closing* – two files instead of one
- writing each byte read from `file.txt` to `copy.txt`

You will need to look at the syntax of the `fopen()` system call to find out how to open a file for writing. Do `man fopen` to learn this.

[1] Every lab sheet should have at least one lousy pun.

[2] Use any editor you like. I use `emacs` because it "knows about" `C` code and it colour codes keywords, etc.

The second item will require you to modify the body of the `while` loop in the program so that the character is no longer put to standard output (`stdout`). You will need to put it to the destination file instead.

❻ This is the first lab that you will be assessed on. There will be 6 assessments in total, each worth 5%. To be assessed with no penalty charged you will need to have the program written, working properly and handed in by 16.00, Tue. Week05.

The command to submit your program for this week will be:

```
handin -m cs4023 -p w03
```

Please type this command exactly as given.

`handin` works by taking a copy of your source code and storing it in a safe place. This copy is then compiled according to the same instructions as you have been given. The resulting program is then run on various test data and the output is compared character-for-character to what the "correct" result should be. If there is *any* deviation then your program is considered to have failed that test.

So, over the semester, when you are given a programming assignment please make sure that the output of your program matches *exactly* with the output you were asked to generate.

Labs that are to be handed in are open for handin-gin at 09.00 on Tuesday of the week it is *assigned* and, in order to get full marks, are due by 16.00 on Tuesday of the *following* week[3]; a lateness penalty applies to submissions made until 18.00, Friday after that. This gives you at least one week to work on each lab that is to be assessed and handin without penalty. The lab sheet will be available for reading prior to the first lab so that you can read it beforehand and come to the lab with questions.

Subject to last-minute changes, the planned schedule of lab assignments due for handing in are:

[3]There are a couple of exceptions where you will be given 2 weeks to complete the lab.

Lab. Week	Assessed	DueDate (16.00, Tue)
Week01	✗	
Week02	✗	
Week03	✓	Week04
Week04	✓	Week06
Week05	✗	
Week06	✓	Week07
Week07	✗	
Week08	✓	Week09
Week09	✓	Week10
Week10	✓	Week11
Week11	✗	

❼ Closing comments:

The trouble with this copyFile program is its inflexibility: it is impossible to read from a file other than `file.txt` or to write it to anything other than `copy.txt`. What we would like to be able to do is to specify the name of the file to read as a part of the command. This is where command-line arguments come in to play and we will discuss these next time.