Quick Intro to Linux

CS - UTSA

Terminal Commands

Linux has a very powerful command-line interface, which is invoked by typing commands into a terminal or xterm window directly (like the DOS/CMD window in Windows). This small note can help you get started learning some of these commands; remember that much more detailed descriptions are available online, in particular:

`http://www.redhat.com/docs/manuals/linux/RHL-6.2-Manual/getting-started-guide/ch-doslinux.html`
`http://www.hscripts.com/tutorials/linux-commands/`

Log onto one of the Linux machines in the CS Main lab, `SB 3.02.04`. Make sure you are in Linux mode. If computer is in Windows, ask lab assistant to reboot to Linux. Log into your Linux account. You will also be able to connect from any machine on the internet by ssh'ing to one of the machines in the CS Main lab. The machines permanently on linux are **elk01**, . . . **elk08**. You can also do your work on your home machine if you install linux. But, you need to make sure that your programs will compile and run on Linux machines in the CS Labs, because TAs will check your programs on those machines.

Once you log onto a Linux machine in the labs, click on `Applications->accessories->terminal` to open up the terminal window which gives you access to the command prompt.

Here are some common commands:

- `ls` : *list*. Prints out what files are available in the present working directory. Important flags include '`-A`', which says to list hidden files as well as normal files, and '`-l`' which says to list with long format (providing more information about all files.

- `pwd`: *print working directory*. Shows where you are in the file system (i.e. are you at your top directory, or deep in nested folders)? For instance, right after I log in, if I type '`pwd`', I get:

  ```
  main205>pwd
  /home/korkmaz
  ```

- `mkdir`: *make directory* (a directory is often called a *folder* in windows). A directory is a special file on the storage device that can contain other files. For instance, '`mkdir cs2213`' will create the directory/folder `cs2213` in the present working directory.

- `cd` : *change directory*. This is the way to move around in the filesystem. For instance, if I type `cd cs2213` then I will change into the subdirectory (a subdirectory is a folder which is contained within another folder) `cs2213`.

  ⇒ There are two special directory handles that you use when issuing command like `cd`. These are:

1. '`./`': the current directory the command prompt is in. For example '`./xc2f`' runs the program `xc2f` which can be found in the present working directory.
2. '`../`': the directory immediately above the current one. Can be repeated as often as you like. For instance '`../../xc2f`' runs the program `xc2f` which is found in the directory two levels above the current one.

⇒ When you provide where the file is to be found, this is called the *path*. So for '`../../xc2f`', '`../../`' is the *relative path* (because the path is relative to the current working directory), and '`xc2f`' is the *file name*. A *fully qualified path* or *absolute path* is one that is not dependent on the directory where you currently are. For instance '`/home/korkmaz/cs2213`' is a fully qualified path.

- `vi filename` : *an editor*. ESC: go to command mode, i: go to insert mode, ZZ: save and quit, x: delete (more later).
- `cat` : *prints*. Prints out the content of files on the screen.
- `cp`: *copy*. Copy a file to another file. For instance '`cp c2f.c f2c.c`' will copy the file `c2f.c` so that the same data exists as the file `f2c.c` in the same directory.
  - Linux filename commands have the special character '`.`', which means "Using the same filename" when given as a target for commands like `cp` or `mv`. For instance '`cp c2f.c ../.`' will duplicate the file `c2f.c` into the directory above this one, using the filename `c2f.c`.
- `mv`: *move*. Copy a file to a new file/path, and delete the original file. For instance, '`mv c2f.c celc2fair.c`' will result in replicating the file `c2f.c` as `celc2fair.c`, and then deleting the original filename (`c2f.c`).
- `rm`: *remove*. Delete the provided file(s). For instance '`rm c2f.o`' will permanently delete the file `c2f.o`. **Use this command with care**, since you can delete every file you own if you are not careful. I recommend that you always use the `-i` flag, which will cause `rm` to ask you if you really want to delete. You can ensure this by typing '`alias rm rm -i`' before ever using `rm`.
- `rmdir`: *remove directory*. Delete the specified directory. This command only works if the directory is empty.
- `history`: print a listing of most recent commands the user has entered. You can then repeat a command using the `!`. For instance `!110` will repeat the command with the label `110` from the `history` listing.
  - → For most shells, hitting the up arrow will take you one command back in your history, the down arrow will take you one command forward in your history, and the left and right arrows allow you to move around in these remembered commands so that you can edit them.
- `man` : *manual*. Print man page of provided command. For instance, '`man ls`' provides the system help on using the command `ls`.

- `pine`: a simple non-graphical e-mail client
- `gcc` : invoke the C compiler. (more information is in the next page)

Creating Files & Compiling C Programs

- To create a file with the name of `program.c`, we need a text editor:

  1. Text editors that work within a terminal window include pico, vi, vim, and emacs. To use these simply issue a command like: `pico program.c` in the terminal window.
     Editing with `vi`: Change to the ∼`/tmp` directory, execute `vimtutor` and follow directions in order to get an introduction to the `vi` editor. (The `vi` editor that you will be using is really `vim`) This will take a while but will save considerable time later. You can also use the cheat sheet provided at the class web page.
  2. You can also use a graphical text editor similar to Notepad by `Applications->accessories->text editor` or simply issue a command like: `gedit program.c`. If you use this, make sure the directory of your terminal window matches where you save the file to (in the terminal window, `pwd` will display your present working directory [i.e, where you are in the filesystem]).

- Once you are in a text editor, you can type any text and save it. Here is a famous C program that worths typing and saving:

```
#include <stdio.h>

int main()
{
   printf("Hello, world.\n");

   return 0;
}
```

- To compile the program that has been saved to the filename `program.c` in the present working directory, and place the executable in the filename `program` in the same directory (under windows, a `.exe` file extension will be automatically added to the filename, but this is not true for linux):

```
gcc program.c -o program
gcc -o program program.c
```

  (where of course `program` is replaced by the names of your new programs).

  – To get the compiler to give you a bunch of warnings about sloppy programming, add the `-DWall` flag to your compilation. i.e., type:

```
gcc -Wall program.c -o program
```

- To execute the program, simply type `./program`.

- To print the program, issue `lpr program.c` in a terminal window.

- To print results of a run of your program, redirect the output to a file, and then simply print that file, as in:

```
./program > output.txt
lpr output.txt
```

Make a directory hierarchy for this class

1. Log onto one of the Linux machines in the CS labs., You will also be able to connect from any machine on the internet by ssh'ing to one of the machines in the CS Main lab. The machines permanently on linux are **elk01**, ... **elk08**.
2. Make the directory `courses` under your home directory with the command `mkdir courses`
3. Change directories to your `courses` directory with the command `cd courses`
4. Make the directory `cs` under your `courses` directory.
5. Change directories to your `cs` directory.
6. Make the directory `2213` under your `cs` directory.
7. Change directories to your `2213` directory.
8. Change to your home directory using `cd`. Notice that a `cd` with no argument changes to your home directory.
9. Make an alias to change directories to your newly created directory, `~/courses/cs/2213`, with the command `cs2213`, i.e.
   `alias cs2213 cd ~/courses/cs/2213`
   Notice that this only works in the shell in which it was executed, not in any other shell (window).
10. Define a cdpath to your `cs` directory with the command
    `set cdpath=( ~/courses/cs ~ )`
11. Print the working directory using the command `pwd`.
12. Jump back and forth between your home directory and your cs2213 directory using the two commands `cd 2213` and `cs2213`. The command `cd 2213` uses the cdpath in the following way: it first checks for a `2213` subdirectory in the current directory and, upon not finding one, looks for a `2213` subdirectory in each directory in the cdpath, changing to the first one found. After each change of directory print your current working directory.
13. Change to your home directory and make subdirectories called `tmp, bin, src, etc`. (if they don't already exist)

A sample session

The following is an actual screen dump of a terminal session showing how one can use some of the outlined commands (I have added some blank lines to make it slightly easier to follow). I log into the machine `main201.cs.utsa.edu` in the CS Main lab.

In this session, as defined above, I create a directory hierarchy for all of my cs 2213 files. Subdirectories will allow us to easily find files later. Note that the sentences prefaced with `#` were not typed in by me or printed out by the terminal: they are comments that I have added to explain what I was doing during the session.

The *prompt* is where you type the commands in the terminal window. Different users have different prompts (and you can change them if you know how). My prompt is `machine_name>`. So, on `main201` it will be '`main201>`'. Your prompt may differ.

The machines `elk01` through `elk05` should be available for remote login to linux 24 hours day. Other machines are available or not depending on if they have been booted into Linux or Windows. These machines include `ant00` – `ant34`, `bat00` – `bat55`, `cat00` – `cat32`, `dog00` – `dog62`.

Remember that if you type the first few letters of a file name and then hit the `Tab` key, the shell will usually autocomplete the file name!

```
main201>pwd                                     # shows current directory
/home/korkmaz
main201>mkdir courses                           # create courses dir in home area
main201>cd courses                              # go to courses subdir
main201>ls                                      # no files in subdir yet
main201>ls -a                                   # -a shows hidden/virtual files
./  ../
main201>pwd                                     # shows current directory
/home/korkmaz/courses
main201>mkdir cs                                # create cs dir in courses
main201>cd cs                                   # go to cs subdir under courses
main201>mkdir 2213; cd 2213

main201>pico program.c     #gedit program.c &   # create a file, edit/save as above
main201>cp program.c prog.c                     # copy program.c to prog.c

main201>gcc -Wall -o prog prog.c                # compile prog.c
main201>ls                                      # see what files I have now
program.c prog.c prog

main201>rm *.exe x* prog prog.c~                # get rid of unneeded files
main201>ls
program.c prog.c prog

main201>mv prog.c helloworld.c                  # get better name for program
main201>ls
program.c helloworld.c
```

```
main201>gcc -Wall -g -o hello helloworld.c      # compile program: -g for debug

main201>mkdir ex1                               # create directory for 1st example
main201>mv *.c ex1/.                            # put c files into it
main201>ls
hello ex1/
main201>ls ex1/                                 # make sure they are there
program.c helloworld.c
```

Let's create an another program under `ex1` to convert Centigrade to Fahrenheit:

```
main201>cd ex1                              # go to ex1 directory
main201>mv program.c c2f.c                  # move program.c to a new file named c2f.c
main201>pico c2f.c    # gedit c2f.c &       # open the file with an editor
```

In the editor, modify the content so that you will have the followings

```
/*
 * Include C header files
 */
#include <stdio.h>
#include <stdlib.h>

int main(int nargs, char **args)
{
   double temp_c,   /* temperature in Centigrade */
          temp_f;   /* temperature in Fahrenheit */
   /*
    * Prompt for centigrade temperature
    */
   printf("What is the temperature in Centigrade? ");
   scanf("%lf", &temp_c);
   /*
    * Convert it to Fahrenheit
    */
   temp_f = 9.0*temp_c/5.0 + 32.0;
   printf("%.2f degrees Centigrade is %.2f degrees in Fahrenheit\n",
          temp_c, temp_f);
   return 0;        /* signal normal end of program */
}
```

Now let's compile, run, and debug it.

```
main201>gcc -Wall -g -o c2f c2f.c           # compile program: -g for debug

main201>ls                                  # see if executable is there
c2f.c  c2f* helloworld.c

main201>./c2f                               # run program
```

```
 What is the temperature in Centigrade? 100
   100.0000     degrees Centigrade is    212.0000     degrees in Fahrenheit

main201> gdb c2f                  # debug the program

main201> ddd c2f                  # debug the program  with a better interface
```

Another Example of Compiling, Executing, and Debugging

1. Type in the following C program into the file fact.c:

```
#include <stdio.h>
#include <stdlib.h>

int factorial(int number);

int main(int argc, char *argv[]) {
  char input[1024];
  int  num, fact;

  printf("Enter a nonnegative number: ");
  while ( gets(input) != NULL ) {
    num = atoi(input);
    if ( num >= 0 ) {
      fact = factorial(num);
      printf("\n\n    %d! = %d\n\n",num,fact);
    }
    printf("Enter a nonnegative number: ");
  }
  exit(0);
}
```

2. Type in the following C function into the file factsub.c:

```
int factorial(int number) {

  if ( number <= 0 ) {
    return(1);
  } else {
    return( number*factorial(number-1));
  }
}
```

3. Compile the program in the two files fact.c and factsub.c calling the executable fac. Use the command

```
gcc -g -Wall fact.c factsub.c -o fac
```

This will compile both c modules, creating object modules for each and then link together the modules to create the executable. The object modules will not be written to the disk in this case. The end of input is indicated by a ctl-D ($\hat{\text{D}}$).

4. It should be noted that there are serious problems with this program. It uses `gets()` which creates a buffer overflow hazard (a security concern). The input is not carefully checked for valid user input.

5. Compile the program and look at its execution in an X-based debugger. On the suns and linux machines you can use the X front end for gdb, the Gnu debugger, `ddd`.

   (a) Compile your program so that it contains debugging information (-g switch):

   ```
   gcc -Wall -g fact.c factsub.c -o fac
   ```

   Now execute `fac` in the debugger:

   ```
   ddd fac
   ```

   You will see the ddd screen and in the source window you will see your main program. You can set a breakpoint anywhere in the program by merely clicking on a line and then clicking on the break button. When the program executes, it will stop at the breaks allowing you to see what values are in the variables.

   There should be breakpoint set at the first line of the program so if you click on the run button, ddd will stop the execution of your program on the first line. Do so!

   Now set a breakpoint on the first printf in the while loop by clicking on the line and clicking on the break button. You should see a little red stop sign appear to the left of the line.

   Now click on the `cont` (continue) button. In the bottom box you will see the output asking for a nonnegative number. You need to input your number, say 4, and then the program will continue to the breakpoint.

   Any time that ddd has control (at a breakpoint, etc), you can look at any of the variable values. So to see the current value of `num`, click on the variable in the program window, anywhere in the code, and then click on the print button. You will see the value in the bottom window. If you want to see the string read into the input array, you can click on the print button then you will see all of the character values for the entire array. The string ends at the first `'\000'`. Now look at the value of the fact variable.

   You can also step through a program, either stepping across functions or stepping into functions. The step button will step into functions while next will step across functions. If you single step using next until the pointer points at the `factorial` call and then click on the `step` button. You will see the source code from the factsub.c file and your current location. Keep using the `step` button to go deeper into the recursive calls, looking at the value of `number` each time that you make a call.

   What happens if you compute the factorial of 17? Of 34? What is the reason for this behavior?