1 OPERATING SYSTEMS LABORATORY I - UNIX TUTORIAL

1.1 Login and Logout

Logging in to a Unix system requires two pieces of information:

- A username,
- and a password.

When you sit down for a Unix session (Ubuntu GNU/Linux in this case), you are given a login prompt that looks like this:

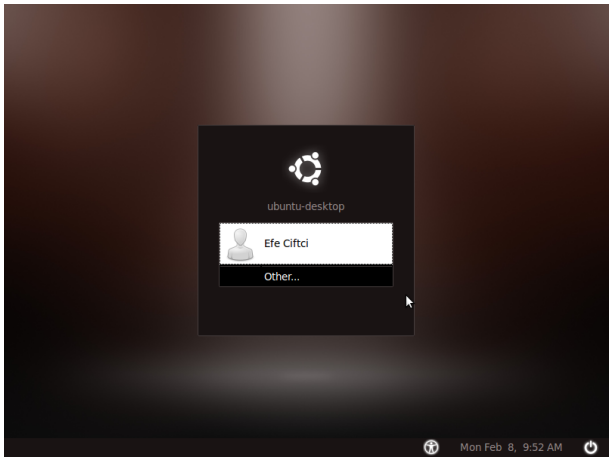

Figure 1: Login Screen

Select / type your username at the login prompt and press the return key. The system will then ask you for your password. After typing your password, press the return key. If you have typed your password correctly, your desktop will be shown.

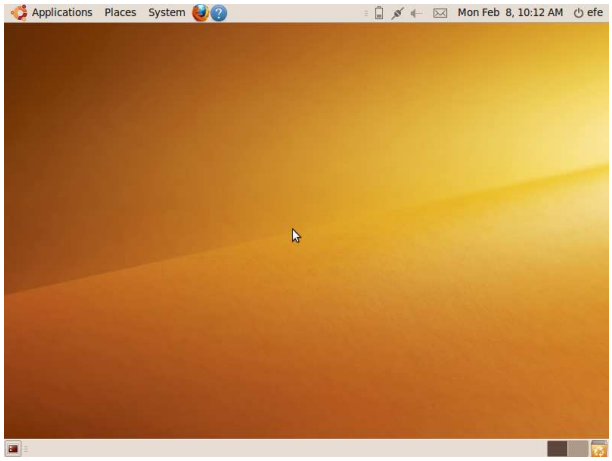

Figure 2: Desktop

1.2 Commands

This part of the laboratory manual will help you learn the basic Unix commands. You should exercise these commands in front of a computer. Read a paragraph, then try the given command. For your own benefit, do NOT copy & paste the commands.

You must first open a terminal application from Applications → Accessories menu on top of the screen. For the first command **date**, see Fig. 3.

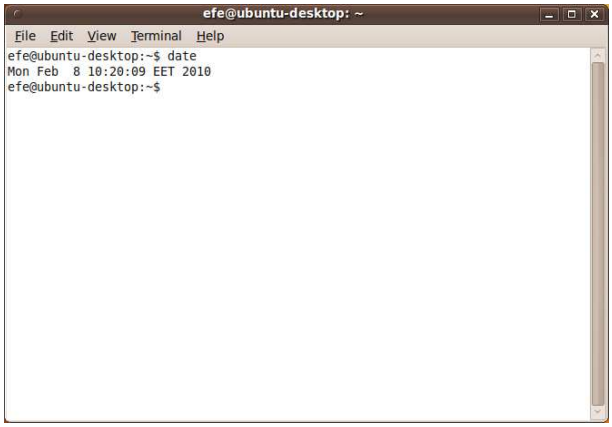

Figure 3: **date** command

1.3 Simple Commands

- **date** - displays the current date and time.
- **whoami** - displays the login name of the current user.
- **echo** - tells the computer to retype the string afer *echo*. Try the following:

  echo This is a test

  echo $USER
  Why the output is different now? What is *$USER*?

  echo 2 + 2 = $((2+2))
  What is the output of *$((2+2))*?

1.4 Working With Files

- **cd <directory name>** - (**c**hange **d**irectory) used for changing the current working directory. If no directory names are given, then the working directory will be changed to home directory.
- **pwd** - (**p**rint **w**orking **d**irectory) tells in which directory we currently are.
- **echo $HOME** - Note that **echo $HOME** has exactly the same effect as **pwd**. Figure out what your home directory is. Now, What is *$HOME*?
- **cat > dict**
  red: rojo
  green: verde
  blue: azul
  <control-d>

By <**control-d**> we mean: hold the control key down; while it is down press "d". We have just used cat to create a short English - Spanish dictionary. This dictionary resides in the file *dict*.

- **ls -l** - lists the files in the current directory. For the moment there is only one, namely *dict*.
- **cat dict** - shows us what is in *dict*.
- **wc dict** - counts words (and more). In the case at hand it tells us that dict contains 3 lines, 6 words, and 34 characters ("letters").
- **grep white dict** - looks for the word *green* in the file *dict* and displays the lines in which this word appears. It gives us a way to search through files.
- **sort dict** - command does just what it says.
- **sort dict > dict2** - the use of the "into" symbol ">". In our example it had the effect of directing the output of the **sort** command from the screen to the file *dict2*.
  ls -l dict dict2 - be sure that **dict2** was there.
  cat dict2 - be sure that the content is correct.
- **rm dict2** - r**em**ove the file *dict2*.

1.5 Working With Directories

- **mkdir letters** - (**m**ake **d**irectory), create a new directory named *letters*.
  ls -l
- **mv dict letters** - (**m**o**v**e), move the *dict* into the directory *letters*.
  ls -l letters/dict
- **mv dict* letters** - here the character * matches any sequence of characters, including the null string. Thus files starting with **dict** would all be moved into letters.
- **cd letters** - (**ch**hange **d**irectory), work inside the directory *letters*.
  ls -l dict
- **cd** - to go back to our home directory.
- **ls -l** - check what our home directory contains.
- **mkdir cprogs; mv *.c cprogs; ls -l cprogs/ letters/**
- **pwd**
  cd letters
  pwd
  rm * - removes all files in the current directory
  cd .. - changes the current directory to the parent of the current one.
  rmdir letters - (**r**emove **d**irectory), to remove a directory we first remove all the file in it, then remove the directory.
- **man** - manual/help, to investigate other flags to the command you are interested in type:
  man *commandname*
  man ls - to investigate other flags, such as "which flags will display file size and ownership?"
  To quit **man** simply type the letter q.
- **ls -l filename** - will list the long directory list entry (which includes owner and permission bits) and the group of a file. The output looks something like:

```
permission    owner group                     filename
-rw-r----- 1 ozdogan ozdogan 65538 Feb 6 01:44 commands.html
```

- The Permission Bits;
  - The first position (which is not set) specifies what type of file this is. If it were set, it would probably be a d (for directory) or l (for link).
  - The next nine positions are divided into three sets of binary numbers and determine permissions for three different sets of people.

    ```
    u    g   o
    421 421 421
    rw- r-- ---
    6   4    0
    ```

    * The file has "mode" 640.
    * The first bits, set to "r + w" (4+2=6) in our example, specify the permissions for the user who owns the files (u).
    * The user who owns the file can read or write (which includes delete) the file.
    * The next trio of bits, set to "r" (4) in our example, specify access to the file for other users in the same group (g) as the group of the file.
    * In this case the group is ug – all members of the ug group can read the file (print it out, copy it, or display it using more).
    * Finally, all other users (o) are given no access to the file.
    * The one form of access which no one is given, even the owner, is "x" (for execute).
    * This is because the file is not a program to be executed.
    * It is probably a text file which would have no meaning to the computer. The x would appear in the third position if it was an executable file.
  - If you wanted to make the file readable to all other users, you could type:
    chmod 644 filename or **chmod o+r filename**
- **rm -i filename** - would return a prompt asking if you are certain you want to delete that file.
- **du** - display disk usage of the current directory and its subdirectories.
  - **du -s** - display only total disk usage.
  - **du -s -k** - some versions of UNIX, such as Solaris, need -k to report kilobytes.
- **df** - to examine what disks and partitions exist and are mounted.
- **ps ux** - to list your own processes.
- **top** - an interactive command that displays and periodically updates the top *cpu processes*, ranked by raw cpu percentage. To quit **top** simply type the letter q.

1.6 Compiling A C Program

- Lets assume there is a file named code1.c that we want to compile. We will do so using a command line similar to this:
  gcc -c code1.c - to compile
  gcc -o code1 code1.o - to link. Suppose that you want the resulting program to be called "code1"
  gcc -o code1 code1.c - just use this command for combined action of compiling and linking.

1.6.1 Running The Resulting Program

- **code1** - Once we created the program, we wish to run it. This is usually done by simply typing its name. However, this requires that the current directory be in our PATH.
- PATH is an environment variable telling our Unix shell where to look for programs we're trying to run. To see your current PATH variable, type **echo $PATH**.

- **./code1** - In many cases, this directory is not placed in our PATH. This time we explicitly told our Unix shell that we want to run the program from the current directory.
- However, yet one more obstacle could block our path - file permission flags.
  ls -l code1
  chmod u+rwx code1 - we set the permissions of the file properly. This means the user ('u') should be given ('+') permissions read ('r'), write ('w') and execute ('x') to the file 'code1'.
  ls -l code1

1.7 Compiling A Multi-Source C Program

- So we learned how to compile a single-source program properly. Yet, sooner or later you'll see that having all the source in a single file is rather limiting, for several reasons:
  - As the file grows, compilation time tends to grow, and for each little change, the whole program has to be re-compiled,
  - It is very hard, if not impossible, that several people will work on the same project together in this manner,
  - Managing your code becomes harder. Backing out erroneous changes becomes nearly impossible.
- The solution to this would be to split the source code into multiple files, each containing a set of closely-related functions.
- There are two possible ways to compile a multi-source C program.
  - The first is to use a single command line to compile all the files. Suppose that we have a program whose source is found in files code2.c, code3.cand code4.c. Analyze these files by opening *kdevelop*. We could compile it this way:
    gcc -o code2 code2.c code3.c code4.c
    This will cause the compiler to compile each of the given files separately, and then link them all together to one executable file named "code2".
  - The problem with this way of compilation is that even if we only make a change in one of the source files, all of them will be re-compiled when we run the compiler again. In order to overcome this limitation, we could divide the compilation process into two phases - **compiling**, and **linking**.
    gcc -c code2.c
    gcc -c code3.c
    gcc -c code4.c
    gcc -o code2 code2.o code3.o code4.o
    - The first 3 commands have each taken one source file, and compiled it into something called "object file", with the same names, but with a ".o" suffix.
    - It is the "-c" flag that tells the compiler only to create an object file, and not to generate a final executable file just yet.
    - The object file contains the code for the source file in machine language, but with some unresolved symbols. For example, the "code2.o" file refers to a symbol named "func_a", which is a function defined in file "code3.c".
    - Surely we cannot run the code like that. Thus, after creating the 3 object files, we use the 4th command to link the 3 object files into one program.
    - The linker (which is invoked by the compiler now) takes all the symbols from the 3 object files, and links them together - it makes sure that when "func_a" is invoked from the code in object file "code2.o", the function code in object file "code3.o" gets executed.
    - **nm code2** - try this command and recognize the definitions for "func_a" and "func_b".

* Further more, the linker also links the standard C library into the program, in this case, to resolve the "printf" symbol properly.

1.8 Exercises

The UNIX shell is case-sensitive, meaning that an uppercase letter is not equivalent to the same lower case letter (i.e., "A" is not equal to "a"). Most all UNIX commands are lower case. Find out the correct command for the followings:

1. Changing to your home directory.
2. Changing access permissions. Change the access permissions of a file or directory.
3. Displaying current variables. Say, to display the value of PATH environment variable (command **export**).
4. Changing default access permissions. Use **umask**, first start with **man umask**.

```
who | wc -l
ps aux | grep 'your username' |sort +5 -6|more
cat dict | head -5 | tail -3
grep 'your username' /etc/passwd
```

5. **man grep**, **man sort**, **man more**, **man head**, **man tail**
6. What is the relative pathname?
7. When you execute a non built-in shell command, the shell asks the kernel to create a new subprocess (called a "child" process) to perform the command. The child process exists just long enough to execute the command. The shell waits until the child process finishes before it will accept the next command. Explain why the exit (logout) procedure must be built in to the shell.