



UNIVERSITY OF ENGINEERING AND TECHNOLOGY, TAXILA
FACULTY OF TELECOMMUNICATION AND INFORMATION ENGINEERING



COMPUTER ENGINEERING DEPARTMENT

Operating systems

LINUX SHELLS

LAB MANUAL 4

Date:	
Name:	
Reg#:	Group:
Marks:	Signature:



UNIVERSITY OF ENGINEERING AND TECHNOLOGY, TAXILA

FACULTY OF TELECOMMUNICATION AND INFORMATION ENGINEERING



COMPUTER ENGINEERING DEPARTMENT

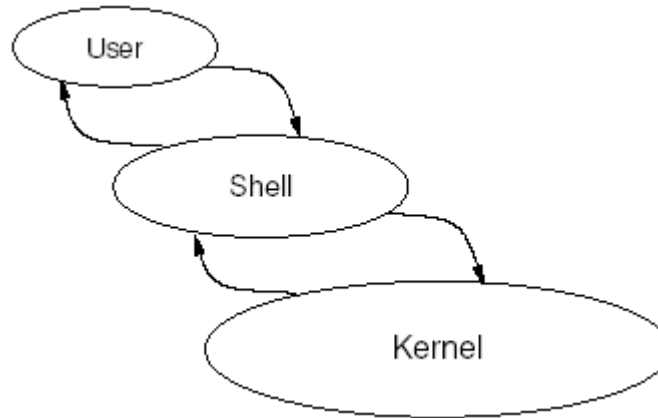
Operating systems

Lab objective

This lab will give overview of Linux shells. You will get insight of 'bash' shell.

Shells

- ✚ A **shell** provides an interface between the user and the operating system kernel
- ✚ Either a **command interpreter** or a graphical user interface
- ✚ Traditional Unix shells are **command-line interfaces** (CLIs)
- ✚ Usually started automatically when you log in or open a terminal



The Bash Shell

Linux's most popular command interpreter is called bash

- ✚ The **Bourne-Again Shell**
 - More sophisticated than the original sh by Steve Bourne
 - Can be run as sh, as a replacement for the original Unix shell
 - Gives you a prompt and waits for a command to be entered
- ✚ Although this course concentrates on Bash, the shell tcsh is also popular
 - Based on the design of the older C Shell (csh)

Shell Commands

- ✚ Shell commands entered consist of words
 - Separated by spaces (whitespace)



UNIVERSITY OF ENGINEERING AND TECHNOLOGY, TAXILA

FACULTY OF TELECOMMUNICATION AND INFORMATION ENGINEERING



COMPUTER ENGINEERING DEPARTMENT

Operating systems

- The first word is the command to run
- Subsequent words are options or arguments to the command
- ✚ For several reasons, some commands are built into the shell itself
 - Called **builtins**
 - Only a small number of commands are builtins, most are separate programs

Command-Line Arguments

- ✚ The words after the command name are passed to a command as a list of **arguments**
- ✚ Most commands group these words into two categories:
 - Options, usually starting with one or two hyphens
 - Filenames, directories, etc., on which to operate
- ✚ The options usually come first, but for most commands they do not need to
- ✚ There is a special option '--' which indicates the end of the options
 - Nothing after the double hyphen is treated as an option, even if it starts with -

Syntax of Command-Line Options

- ✚ Most Unix commands have a consistent syntax for options:
 - Single letter options start with a hyphen, e.g., -B
 - Less cryptic options are whole words or phrases, and start with two hyphens, for example
--ignore-backups
 - For example, long options (not single letters) sometimes start with a single - rather than --



UNIVERSITY OF ENGINEERING AND TECHNOLOGY, TAXILA






FACULTY OF TELECOMMUNICATION AND INFORMATION ENGINEERING







COMPUTER ENGINEERING DEPARTMENT

Operating systems

Examples of Command-Line Options

-  List all the files in the current directory:
\$ ls
-  List the files in the 'long format' (giving more information):
\$ ls -l
-  List full information about some specific files:
\$ ls -l notes.txt report.txt
-  List full information about all the *.txt* files:
\$ ls -l *.txt
-  List all files in long format, even the hidden ones:
\$ ls -l -a
\$ ls -la

Setting Shell Variables

-  **Shell variables** can be used to store temporary values
-  Set a shell variable's value as follows:
\$ files="notes.txt report.txt"
 - The double quotes are needed because the value contains a space
 - Easiest to put them in all the time
-  Print out the value of a shell variable with the echo command:
\$ echo \$files
 - The dollar (\$) tells the shell to insert the variable's value into the command line
-  Use the set command (with no arguments) to list all the shell variables

Environment Variables

-  Shell variables are private to the shell



UNIVERSITY OF ENGINEERING AND TECHNOLOGY, TAXILA

FACULTY OF TELECOMMUNICATION AND INFORMATION ENGINEERING



COMPUTER ENGINEERING DEPARTMENT

Operating systems

- ✚ A special type of shell variables called **environment variables** are passed to programs run from the shell
- ✚ A program's **environment** is the set of environment variables it can access
 - In Bash, use export to export a shell variable into the environment:
\$ files="notes.txt report.txt"
\$ export files
 - Or combine those into one line:
\$ export files="notes.txt report.txt"
- ✚ The env command lists environment variables

Where Programs are found

- ✚ The location of a program can be specified explicitly:
 - ./sample runs the sample program in the current directory
 - /bin/ls runs the ls command in the /bin directory
- ✚ Otherwise, the shell looks in standard places for the program
 - The variable called PATH lists the directories to search in
 - Directory names are separated by colon, for example:
\$ echo \$PATH
/bin:/usr/bin:/usr/local/bin
 - So running whoami will run /bin/whoami or /usr/bin/whoami or /usr/local/bin/whoami (whichever is found first)

Bash Configuration Variables

- ✚ Some variables contain information which Bash itself uses
 - The variable called PS1 (Prompt String 1) specifies how to display the shell prompt
- ✚ Use the echo command with a \$ sign before a variable name to see its value, e.g.



UNIVERSITY OF ENGINEERING AND TECHNOLOGY, TAXILA

FACULTY OF TELECOMMUNICATION AND INFORMATION ENGINEERING



COMPUTER ENGINEERING DEPARTMENT

Operating systems

\$ echo \$PS1

[\u@\h \W]\\$

- ✚ The special characters `\u`, `\h` and `\W` represent shell variables containing, respectively, your user/login name, machine's hostname and current working directory, i.e.,
 - \$USER, \$HOSTNAME, \$PWD

Using History

- ✚ Previously executed commands can be edited with the Up or Ctrl+P keys
- ✚ This allows old commands to be executed again without re-entering
- ✚ Bash stores a **history** of old commands in memory
 - Use the built-in command history to display the lines remembered
 - History is stored between sessions in the file `~/.bash_history`
- ✚ Bash uses the readline library to read input from the user
 - Allows Emacs-like editing of the command line
 - Left and Right cursor keys and Delete work as expected

Reusing History Items

- ✚ Previous commands can be used to build new commands, using **history expansion**
- ✚ Use `!!` to refer to the previous command, for example:

```
$ rm index.html
$ echo !!
echo rm index.html
rm index.html
```
- ✚ More often useful is `!string`, which inserts the most recent command which started with *string*



UNIVERSITY OF ENGINEERING AND TECHNOLOGY, TAXILA

FACULTY OF TELECOMMUNICATION AND INFORMATION ENGINEERING



COMPUTER ENGINEERING DEPARTMENT

Operating systems

- Useful for repeating particular commands without modification:

```
$ ls *.txt
```

```
notes.txt report.txt
```

```
$ !ls
```

```
ls *.txt
```

```
notes.txt report.txt
```

Retrieving Arguments from the History

- 🔧 The event designator !\$ refers to the last argument of the previous command:

```
$ ls -l long_file_name.html
```

```
-rw-r--r-- 1 jeff users 11170 Feb 20 10:47 long_file_name.html
```

```
$ rm !$
```

```
rm long_file_name.html
```

- 🔧 Similarly, !^ refers to the first argument

Summary of Bash Editing Keys

- 🔧 These are the basic editing commands by default:

- Right — move cursor to the right
- Left — move cursor to the left
- Up — previous history line
- Down — next history line
- Ctrl+A — move to start of line
- Ctrl+E — move to end of line
- Ctrl+D — delete current character

- 🔧 There are alternative keys, as for the Emacs editor, which can be more comfortable to use than the cursor keys



UNIVERSITY OF ENGINEERING AND TECHNOLOGY, TAXILA

FACULTY OF TELECOMMUNICATION AND INFORMATION ENGINEERING



COMPUTER ENGINEERING DEPARTMENT

Operating systems

- There are other, less often used keys, which are documented in the bash man page (section 'Readline')

Combining Commands on One Line

- You can write multiple commands on one line by separating them with ;
- Useful when the first command might take a long time:
time-consuming-program; ls
- Alternatively, use && to arrange for subsequent commands to run only if earlier ones succeeded:
time-consuming-potentially-failing-program && ls

Repeating Commands with 'for'

- Commands can be repeated several times using for
 - Structure: for *varname* in *list*; do *commands*...; done
- For example, to rename all *.txt* files to *.txt.old* :
\$ for file in *.txt;
> do
> mv -v \$file \$file.old;
> done
barbie.txt -> barbie.txt.old
food.txt -> food.txt.old
quirks.txt -> quirks.txt.old

- The command above could also be written on a single line

Command Substitution

- Command substitution** allows the output of one command to be used as arguments to another



UNIVERSITY OF ENGINEERING AND TECHNOLOGY, TAXILA

FACULTY OF TELECOMMUNICATION AND INFORMATION ENGINEERING



COMPUTER ENGINEERING DEPARTMENT

Operating systems

- For example, use the locate command to find all files called *manual.html* and print information about them with ls:

```
$ ls -l $(locate manual.html)
```

```
$ ls -l 'locate manual.html'
```

- The punctuation marks on the second form are opening single quote characters, called **backticks**
 - The $()$ form is usually preferred, but backticks are widely used
- Line breaks in the output are converted to spaces

Finding Files More Flexibly: 'find'

- locate only finds files by name
- find* can find files by any combination of a wide number of criteria, including name
- Structure: *find directories criteria*
- Simplest possible example: *find .*
- Finding files with a simple criterion:

```
$ find . -name manual.html
```

Looks for files under the current directory whose name is *manual.html*
- The *criteria* always begin with a single hyphen, even though they have long names

'find' Criteria

- find accepts many different criteria; two of the most useful are:
 - name *pattern*: selects files whose name matches the shell-style wildcard *pattern*
 - type d, -type f: select directories or plain files, respectively
- You can have complex selections involving 'and', 'or', and 'not'



Operating systems

‘find’ Actions: Executing Programs

✚ *find* lets you specify an action for each file found; the default action is simply to print out the name

- You can alternatively write that explicitly as `-print`

✚ Other actions include executing a program; for example, to delete all files whose name starts

✚ with *manual*:

```
find . -name 'manual*' -exec rm '{}' ';'
```

✚ The command `rm '{}'` is run for each file, with `'{'` replaced by the filename

✚ The `{}` and `;` are required by *find*, but must be quoted to protect them from the shell

Exercises

Q1

- Use the `df` command to display the amount of used and available space on your hard drive.
- Check the man page for `df`, and use it to find an option to the command which will display the free space in a more human-friendly form. Try both the single-letter and long-style options.
- Run the shell, `bash`, and see what happens. Remember that you were already running it to start with. Try leaving the shell you have started with the `exit` command.

Q2

- Try `ls` with the `-a` and `-A` options. What is the difference between them?
- Write a for loop which goes through all the files in a directory and prints out their names with `echo`. If you write the whole thing on one line, then it will be easy to repeat it using the command line history.



UNIVERSITY OF ENGINEERING AND TECHNOLOGY, TAXILA
FACULTY OF TELECOMMUNICATION AND INFORMATION ENGINEERING



COMPUTER ENGINEERING DEPARTMENT

Operating systems

- c. Change the loop so that it goes through the names of the people in the room (which needn't be the names of files) and print greetings to them.
- d. Of course, a simpler way to print a list of filenames is `echo *`. Why might this be useful, when we usually use the `ls` command?

Q3

- a. Use the `find` command to list all the files and directories under your home directory. Try the `-type d` and `-type f` criteria to show just files and just directories.
- b. Use `'locate'` to find files whose name contains the string `'bashbug'`. Try the same search with `find`, looking over all files on the system. You'll need to use the `*` wildcard at the end of the pattern to match files with extensions.
- c. Find out what the `find` criterion `-iname` does.