

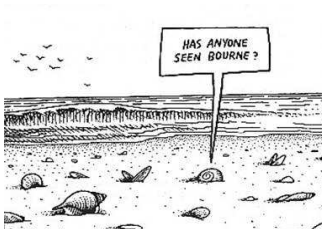
# LINUX SHELLS

## Objectives

- Variables
- Common Shell Features: Understand the shell environment and its features
- Customize the user's bash shell by editing the setup files
- Create aliases (alias)
- Modifying the shell behavior (set)

[ Module 6: Interactive Shell Environment ] - Page 1 •

SYST13416 Introduction to Linux Operating



*There are two primary ways to use the shell: interactively and by writing shell scripts.*

- In the **interactive mode**, the user types a single command (or a short string of commands) and the result is printed out.
- In **shell scripting**, the user types anything from a few lines to an entire program into a text editor, then executes the resulting text file as a shell script.

## Common Shell Features

**History** —access or enter a previous command

**Tokenizing** —parse command line into tokens

**Quoting** —interpret characters and words literally with quote, slash, and backslash.

**Aliases** —define your own commands

**Redirection** —change sources of command input, output, and errors

**Pipes** —send output of a command into another command

**Processing** —handle execution of programs

**Variables** —define words to store values

**Command substitution** —replace a command with its result in a command line (`mkdir backup`date +%d%m%Y``)

**Wildcards (Globbing)** —use special characters to match patterns of filenames

**Sequences** —combine command lines (`cd; ls`)

**Built-in commands** —shells provide some commands that are literally built "inside" the shell.

—called from a shell, that is, executed directly in the shell itself, instead of an external executable program which the shell would load and execute.

—examples:

The command `echo` is a build-in in `csh`, `ksh`, `sh`, `bash`

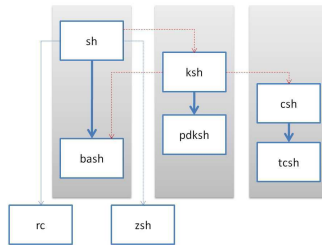
The command `cd` is a build-in `csh`, `ksh`, `sh`

The command `alias` is a built-in in `csh`, `ksh`, `bash`

The command `read` is a built-in in `ksh`, `sh`, `bash`

The command `pwd` is a built-in in `ksh`, `sh`

[ Module 6: Interactive Shell Environment ] - Page 2 •



**Figure:** Shells fit neatly into a few "families" with the exception of a couple stragglers. Each shell in a family shares many characteristics with the others in the same family.

## The Shell

- The shell sits between you and the kernel, acting as a **command interpreter**. It reads your terminal input and translates the commands into actions taken by the system. The shell is analogous to **command.com** in **DOS**.
- When you log into the system you are given a **default shell**. When the shell starts up it reads its startup files and may set **environment variables**, **command search paths**, and **command aliases**, and executes any commands specified in these files.
- The original shell was the Bourne shell, **sh**. Every Unix platform will either have the Bourne shell, or a Bourne compatible shell available. It has very good features for controlling input and output, but is not well suited for the interactive user. To meet this need the C shell, **csh**, was written and is now found on most, but not all, Unix systems. It uses C type syntax, the language Unix is written in.
- The default system **prompt** for the Bourne shell is **\$** (or **#** for the root user). The default prompt for the C shell is **%**. Some flavours of Unix also use the **>** for system prompt.
- Numerous other shells are available from the network. Almost all of them are based on either **sh** or **csh** with extensions to provide job control to **sh**, allow in-line editing of commands, page through previously executed commands, provide command name completion and custom prompt, etc. Some popular shells include the Korn shell, **ksh**, by David Korn, and the Bourne Again Shell, **bash**, from the Free Software Foundations GNU project, both based on **sh**, the TC shell, **tcsh**, and the extended C shell, **cshe**, both based on **csh**.
- The shells have a number of built-in, or native, commands. These commands are executed directly in the shell and don't have to call another program to be run. These built-in commands are different for the different shells. For the Bourne shell some of the more commonly used built-in commands are: **case**, **cd**, **echo**, **eval**, **exec**, **exit**, **export**, **for**, **if**, **pwd**, **read**, **set**, **test**, **trap**, **umask**, **unset**, **wait**, **while**.

Shell Name	Original Shell It is Derived From	Description
Bash	Bourne and Korn shells	Offers strong programming language features like shell variables, control structures, and logic/math expressions. Combines the best features of the other shells.
tsch	C Shell	Conforms to a programming language format. Shell expressions use operators like those found in the C programming language.
zsh	Korn shell	Reflects the Bash shell in many respects but also has C programming-like syntax. Useful if you are familiar with older Korn shell scripts.

Table 6-1: Linux shells

## Choosing which Shell to Use

- To change your shell you can usually use the **chsh** or **passwd -e** commands. The option flag may vary from system to system, so check the *man* page on your system for proper usage. Sometimes this feature is disabled and you will have to check with your System Administrator.
- The login shell is assigned by the administrator. You can view the defaults in the **/etc/passwd** file (substitute your user name for userID) and examine the last field in the output:  
`cat /etc/passwd | grep userID`
- To start a new session, you chose the shell you wish to use and enter the corresponding command. For example, to start a Bourne shell session, enter:  
**sh**
- Or, to start a Bourne Again session, enter:  
**bash**
- To close each session you open, enter  
**exit**

### Note that in this course, we are using the Bourne family exclusively.

- Therefore, every time you connect to the server through SSH Client, immediately switch to either Bourne (sh) or Bourne Again (bash) shell.

**Optional step if you have access to the administrator's account:**

- Create a new user. Log in as the new user and at the prompt, enter `chsh -l` and press <enter>. (the option is 'el' not l.) Note the output. What is displayed?
- To change shells, you use `chsh` without any options. At the prompt, type `chsh`  
Note the output. When prompted, enter your ordinary user's password.
- At this point, you need to tell the system where to find the new shell. To call up the Berkeley C-shell, you type:  
`/bin/csh`
- To change your shell to Bourne Again, you type:  
`/bin/bash`
- The new shell must be the full path name for a valid shell on the system. Which shells are available to you will vary from system to system. The full path name of a shell may also vary. Normally, the Bourne and C shells are standard, and available as:  
`/bin/sh`  
`/bin/csh`
- Some systems will also have the Korn shell standard, normally as:  
`/bin/ksh`
- Some shells are quite popular, but not normally distributed by the OS vendors: `bash` and `tcsh`. These might be placed in `/bin` or a locally defined directory, such as `/usr/local/bin` or `/opt/local/bin`. Should you choose a shell not standard to the OS make sure that this shell, and all login shells available on the system, are listed in the file **`/etc/shells`**. If this file exists and your shell is not listed in this file the *file transfer protocol daemon*, `ftpd`, will not let you connect to this machine. If this file does not exist only accounts with "standard" shells are allowed to connect via `ftp`.
- You can always try out a shell before you set it as your default shell. To do this just type in the shell name as you would any other command, for example:  
`sh`

## Shell Scripting

Moving from interactive to batch mode (automation)

**Start off with a SHA-BANG (#!)**

- The sha-bang at the head of a script tells your system that this file contains set of commands to be fed to the command interpreter. Immediately following it, is a path name where the program that interprets the commands in your script resides.
- Using `vi`, create the following script named **`myVeryFirst.sh`** (enter all lines, substituting where appropriate. The format of the dates is of your choosing, but be consistent. This is your template for starting every script):  

```
#!/bin/sh
#Program name: myVeryFirst.sh
#Author name: Put your name here
#Date created: Date the file was first created
#Date updated: Date the file was last modified
#Description:
#   A short description of what the purpose of
#   the program is. This script print a couple
#   of strings to the monitor.
echo "Hello, you have successfully run your first script."
echo "Congratulations!"
```

Once you've finished entering the content, save the file and exit `vi`. You can test running your script:

```
» sh ./myVeryFirst.sh
```

If the script executed without a problem, you can change the permissions appropriately and you are ready to give it to another person to use.

```
» chmod 751 myVeryFirst.sh
```

Otherwise, go back and check that you have entered everything properly. Once you make the file executable, you can run it like other commands:

```
» myVeryFirst.sh
```

Or if the directory is not in the `PATH` variable:


```
» ./myVeryFirst.sh
```

# SHELL VARIABLES

## Objectives

- Command: echo - display a line of text
- Variables: symbolic names that represent values stored in memory
- Operators: Assignment of values to variables (name=value)
- Variable evaluation using single, double, and back quotes
- Positional parameters: command line arguments
- Environment variables (env, printenv)
- The PATH variable
- Customizing bash prompt (PS1)

[ Module 6: Interactive Shell Environment ] - Page 7 •



**NAME**  
echo - display a line of text

**SYNOPSIS**  
echo [OPTION] [STRING]

**DESCRIPTION**  
Echo or display the STRING(s) to standard output.

**OPTIONS**

**-n** do not output the trailing newline

## echo Command

- The echo command is used to repeat, or echo, the argument you give it back to the standard output device. It normally ends with a **line-feed**, but you can specify an option (-n) to prevent this.
- A common use of the echo command is to prompt users for input, as you will see later.

### Examples

- To display the string "Hello Class" and move to a new line:  
`echo Hello Class`
- The same output will be produced by quoting the string:  
`echo "Hello Class"`
- At the command line, enter the following sequence, line at a time:  
`cat > whoson`  
`#!/bin/bash`  
`date`  
`echo Users Currently Logged In`  
`who`  
`Ctrl+D`  
`chmod u+x whoson`  
`./whoson`
- Result should look something like the following  
`Fri Jun 17 10:59:40 PDT 2008`  
`Users Currently Logged In`  
`alex tty1 Jun 17 08:26`  
`Jenny tty02 Jun 17 10:04`

[ Module 6: Interactive Shell Environment ] - Page 8 •

## Shell Variables

### Variables

- symbolic names that represent values stored in memory

#### Assignment of values to variables

- To assign a value to a variable, choose a name for your variable.
- Remember that the Linux namespace is case-sensitive.
- Use upper and lower letters, numbers (some symbols are safe to use).
- Generally, variable names begin with a letter but can contain numerals and the underscore
- Avoid using keywords (Keep in mind: if you call your variable cat, it may prevent you from using the command cat)
- Shell variables are not typed: You do not need to declare variable type; all values are treated as strings by default.
- Use double quotes to treat spaces and other symbols as part of the string value
- You can create variables at the command line or in a shell script
- To create local variables by using **variable assignment** in the form of `variable=value`

- NOTE: there is **no whitespace around the symbol =**

#### Examples:

```
DOG="Barky"
PI=3.14159
CAT="Spot"
```

- All shell variables are initialized to null strings by default.
- Explicit assignment of null strings is possible with:
 

```
x=" "
x=' '
x=
```

*There are significant syntax differences between the Bourne shell family and C shell family. Make sure you use **bash***

### Variable Evaluation

- To display variable names and assigned values:
 

```
echo "That DOG is called $DOG"
echo "The value of PI is $PI"
echo "My cat's name is $CAT"
```
- Compare:
 

```
echo "The average pay is $1000"
The average pay is 000
echo The average pay is \$1000
The average pay is $1000
echo 'The average pay is $1000'
The average pay is $1000
```
- Note that the shell evaluated the variable \$1 in the first example, using double-quotes. The variable \$1 is **positional parameters** (see next page).
- Whether you use **double** or **single quotes** depends on whether you want **command substitution** and **variable evaluation** to be enabled.
- Double quotes permit interpretation, single quotes do not.
- Example showing both command substitution and variable evaluation:
 

```
echo "The PATH is $PATH. The current directory is `pwd`"
```
- Other examples:
 

```
mydir=`pwd`; echo $mydir
size=`wc -c < foo.txt`
message=You\ didn't\ enter\ the\ filename.
```
- Note: quotes have to be properly nested.

### Quotation

#### Single Quote (strong quote)

- Without, shell will interpret meta-characters, triggering the substitution of other text.

#### Back Quotes

- Command substitution

#### Double Quote (weak quote)

- Allows interpretation of
  - \$ variable/parameter
  - ` command substitution
  - ! command history

#### Backslash

- Strong quotes

- Using **echo** to output an **unquoted variable set with command substitution** removes trailing newlines characters from the output of the reassigned command

```
dirListing=`ls -l`
```

```
echo $dirListing      # unquoted
```

```
drwxr-x--- 6 bajcar users 8192 Feb 1 17:28 tutorial4 drwxr-x-
-- 3 bajcar users 8192 Feb 1 10:52 week03 -rwxr-x--x 1 bajcar
users 405 Feb 3 15:58 zzz
```

```
echo "$dirListing"    # quoted
```

```
drwxr-x--- 6 bajcar users 8192 Feb 1 17:28 tutorial4
drwxr-x--- 3 bajcar users 8192 Feb 1 10:52 week03
-rwxr-x--x 1 bajcar users 405 Feb 3 15:58 zzz
```

### Shell variables: positional parameters (Command-line Arguments)

Variable	Usage	sh	csh
<b>\$#</b>	Number of arguments on the command line (number of positional parameters)	✓	
<b>\$-</b>	Options supplied to the shell	✓	
<b>\$?</b>	Exit value of the last command executed	✓	
<b>\$\$</b>	Process number of the current process	✓	✓
<b>#!</b>	Process number of the last command done in background	✓	
<b>\$n</b>	Argument on the command line, where n is from 1 through 9, reading left to right, that is \$1, \$2, \$3, \$4, \$5, \$6, \$7, \$8, and \$9, holding the value of the 1st, 2nd, 3rd, 4 <sup>th</sup> , 5 <sup>th</sup> , 6 <sup>th</sup> , 7 <sup>th</sup> , 8 <sup>th</sup> , and 9 <sup>th</sup> argument, respectively.	✓	✓
<b>\$0</b>	The name of the current shell or program (the first token)	✓	✓
<b>\$*</b>	All arguments on the command line (" \$1 \$2 ... \$9")	✓	
<b>\$@</b>	All arguments on the command line, each separately quoted (" \$" "\$2" ... "\$9")	✓	

- Example: enter at the bash command line (See manual pages for set and unset commands.). Note that the variable \$1 holds the value A which is the first parameter in the set command, \$2 holds the value B, and so on.  

```
set A B C
echo $1 $2 $3
```
- Create a script called **args.sh**. Enter the code, save it, and make it executable.  

```
#!/bin/sh
echo "the first argument is $1"
echo "the second is $2"
echo the number off arguments is $#
```
- Try running the script by providing a variety of arguments.

## Using arguments to previous command

- `$_` signifies **the last argument** to the previous command
- Example:  

```
mkdir bar
cd $_
```
- Example:  

```
vi some.sh
exit vi and at the command prompt, give the following command:
$_
```
- `!*`  signifies **all arguments** to the previous command
- Example:  
Create some files, if you already don't have them, and display their content.  

```
touch foo1.sh foo2.sh foo3.sh
cat foo1.sh foo2.sh foo3.sh
```

  
Then move the files expanded from all arguments of previous command  

```
mv !* ../backup
rm !*
```
- Note that if you run the following commands, the `rm` would also run with `-l` as argument and report an error.  

```
ls -l foo
rm !*
```

## Configuration variables

- Used to store information about the setup of the OS and then not changed

## Environment variables

- Environment variables are used to provide information to the program you use.
- You can have both global environment variables and local shell variables.
- Global environment variables are set by your login shell and new programs and shells inherit the environment of their parent shell.
- Local shell variables are used only by that shell and are not passed on to other processes. A child process cannot pass a variable back to its parent process.
- list of common variables
  - DISPLAY** The graphical display to use, e.g. atlas:0.0
  - EDITOR** The path to your default editor, e.g. /usr/bin/vi
  - GROUP** Your login group, e.g. staff
  - HOME** Path to your home directory, e.g. /home/frank
  - HOST** The hostname of your system, e.g. atlas
  - IFS** Internal field separators, usually any white space (defaults to tab, space and <newline>)
  - LOGNAME** The name you login with, e.g. frank
  - PATH** Paths to be searched for commands, e.g. /usr/bin:/usr/ucb:/usr/local/bin
  - PS1** The primary prompt string, Bourne shell only (defaults to \$)
  - PS2** The secondary prompt string, Bourne shell only (defaults to >)
  - SHELL** The login shell you're using, e.g. /usr/bin/csh
  - TERM** Your terminal type, e.g. xterm
  - USER** Your username, e.g. frank
- To list environment variables  

```
env
or
printenv
```
- View an environment variable, use the `echo` command followed by the name of the variable, preceded by the `$`  

```
echo $TERM
```

### Exporting Shell Variables to the Environment

- Shell scripts cannot automatically access variables created on the command line or by other shell scripts. To make a variable available to a shell script, use the `export` command to make it an environment variable  
`export DOG`
- To set a local shell variable, set the variable with the syntax:  
`name=value`
- To create environment variables, you create a local variable which is then exported to the environment:  
`name=value; export name`  
or  
`export name=value`
- To delete or unset the variable, use the `unset` command  
`unset DOG`
- You can also set a variable to read only  
`readonly DOG`
- Note: Variables are exported so the subsequently executed commands will also know about the environment.

### The PATH Environment Variable

- When you run a program at the command line, `bash` actually searches through a list of directories to find the program you requested. For example, when you type `ls`, `bash` doesn't intrinsically know that the `ls` program lives in `/usr/bin`. Instead, `bash` refers to an environment variable called `PATH`, which is a colon-separated list of directories. We can examine the value of `PATH`  
`echo $PATH`  
`/usr/local/bin:/usr/bin:/bin:/usr/sbin:/sbin:/usr/X11R6/bin`
- Given this value of `PATH` (yours may differ), `bash` would first check `/usr/local/bin`, then `/usr/bin` for the `ls` program. Most likely, `ls` is kept in `/usr/bin`, so `bash` would stop at that point.
- To add a directory to the `PATH`  
`export PATH=$PATH:/new/dir`
- Another example, to include the current directory in your `PATH`  
`export PATH=$PATH:.`
- To add your home directory to the `PATH`  
`export PATH=$PATH:$HOME`



- The Bash prompt can do much more than displaying such simple information as your user name, the name of your machine and some indication about the present working directory. We can add other information such as the current date and time, number of connected users etc.
- Before you begin, however, you should save the current prompt in another environment variable:  
`MYPROMPT=$PS1`  
`echo $MYPROMPT`
- The output will look something like the following:  
`[\u@\h \w]\$`
- When you change the prompt now, you can always get our original prompt back with the command `PS1=$MYPROMPT`. You will, of course, also get it back when you reconnect, as long as you just fiddle with the prompt on the command line and avoid putting it in a shell configuration file.
- Another example, to displays time of day and number of running jobs (programs you are running)  
`export PS1="[\t \j] "`
- In order to understand these prompts and the escape sequences used, refer to the man pages.

PROMPTING: Customizing your Bash Prompt (PS1)

- When executing the shell interactively, **bash** displays the primary prompt **PS1** when it is ready to read a command and the secondary prompt **PS2** when it needs more input to complete a command.
- Bash allows these prompt strings to be customized by inserting a number of backslash-escaped special characters that are decoded as follows: (This is a partial listing only, for full listing, see the bash manual pages.)

\a	an ASCII bell character (07)
\d	the date in "Weekday Month Date" format (e.g., "Tue May 26")
\h	displays the hostname up to the first `.'
\H	the hostname
\j	the number of jobs currently managed by the shell
\l	the basename of the shell's terminal device name
\n	displays a newline
\s	displays the name of the shell, the basename of \$0 (the portion following the final slash)
\u	the username of the current user
\v	the version of bash (e.g., 2.00)
\w	the current working directory, with \$HOME abbreviated with a tilde (uses the \$PROMPT_DIRTRIM variable)
\W	the basename of the current working directory, with \$HOME abbreviated with a tilde
!\	the history number of this command
\#	the command number of this command
\\$	displays a # if root is the user, otherwise displays a \$
\nnn	the character corresponding to the octal number nnn
\\	a backslash
\[	begin a sequence of non-printing characters, which could be used to embed a terminal control sequence into the prompt
\]	end a sequence of non-printing characters
\t	displays the time
\\$PWD	displays the path of the current working directory

Sheridan

SYST13416  
Introduction to Linux  
Operating  
The Shell

# SHELL SETUP FILES

Objectives

- Understand the shell environment and its setup files
- Customize the user's bash shell by editing the setup files
- Give nicknames with **alias** command

### Customizing Shell Environment

- When work requirements center on computer programming and shell scripting
- customize environment by modifying the initial settings in the login scripts.
- The .bashrc file that resides in your home directory can be used to establish customizations that take effect for each login session. The .bashrc script is executed each time you generate a shell, such as when you run a shell script.

File	What It Does
/etc/profile	Executes automatically at login
~/ .bash_profile, ~/ .bash_login, ~/ .profile	Executes automatically at login
~/ .bashrc	Executes automatically at shell login
~/ .bash_logout	Executes automatically at logout
~/ .bash_history	Records last session's commands
/etc/passwd	Source of home directories for --name abbreviations
~/ .cshrc or ~/ .tcshrc	Executes at each shell startup
~/ .login	Executed by login shell after cshrc (or tcshrc) at login
~/ .cshdirs	Executes after tcsh_login
~/ .logout	Executes at logout from csh or tcsh

### Shell Setup Files

- Examine the manual pages and view all mentioned files using cat or more command. **Do not load the files into vi at this time. Always make a copy of the files you intend to change first.**
- The prompt configuration line that you want is best put in your shell configuration file, **~/ .bashrc**.
- Prompts can execute shell scripts and behave differently under different conditions. You can even have the prompt play a tune every time you issue a command, although this way it gets boring pretty soon.
- When entering the ls -al command to get a long listing of all files, including the ones starting with a dot, in your home directory, you will see one or more files starting with a period (.) and ending in rc. For the case of bash, this is **.bashrc**. This is the counterpart of the system-wide configuration file **/etc/bashrc**.
- When logging into an interactive login shell, login will do the authentication, set the environment and start your shell. In the case of bash, the next step is reading the general profile from /etc, if that file exists.
- bash then looks for **~/ .bash\_profile**, **~/ .bash\_login** and **~/ .profile**, in that order, and reads and executes commands from the first one that exists and is readable. If none exists, **/etc/bashrc** is applied.
- When a login shell exits, bash reads and executes commands from the file **~/ .bash\_logout**, if it exists.
- This procedure is explained in detail in the login and bash man pages.
- When work requirements center on computer programming and shell scripting, customize environment by modifying the initial settings in the login scripts. The .bashrc file that resides in your home directory can be used to establish customizations that take effect for each login session. The .bashrc script is executed each time you generate a shell, such as when you run a shell script.

[ Module 6: Interactive Shell Environment ] - Page 19 •



**Caution**  
BEFORE YOU EDIT ANY FILE, MAKE A BACKUP OF THE ORIGINAL FILE WITH MEANINGFUL EXTENSIONS.

sh uses the startup file .profile in your home directory. There may also be a system-wide startup file, such as /etc/profile. If so, the system-wide will be executed before your local one. A simple .profile could be the following:

```
PATH=/usr/bin:/usr/ucb:
/usr/local/bin:.
export PATH
#Set a prompt
PS="`hostname`whoami`"
#functions
ls() { /bin/ls -sbF "$@";}
ll() { ls -al "$@";}
#Set the terminal type
# set Control-H to be the
erase key
stty erase ^H
#prompt for the terminal
type, assume xterm
eval `tset -Q -s -m
':?xterm`
#
umask 077
```

### Customizing User's bash Shell

- The bash shell uses two files to configure itself. The file .profile is read only once, when you log in, and the .bashrc file is read every time a shell is started.
- Log on. Make sure you are using the sh or bash shell. In your user's home directory, make a backup copy of your .bashrc file:  
cp .bashrc .bashrc\_original
- Note that the -a option for ls command will list all hidden files. Any file that starts with a dot is treated as a hidden file.**
- Using the vi editor, load the .bashrc file and add the following lines to the end:  
# the following command sets the command-line  
# editing to vi style, will not log you out if you  
# accidentally type Ctrl+D at the command line, and  
# will not blindly overwrite or delete files  
set -o vi ignoreeof noclobber  
  
# sets the prompt line  
PS1='\u \t \W \! bash\$ '  
  
# includes current directory in the PATH  
PATH=\$PATH:.  
  
Start a bash shell and have it source the .bashrc file.  
bash  
. ~/.bashrc
- Correct errors if need be.

/bin/bash	The bash executable
/etc/profile	The systemwide initialization file, executed for login shells
~/ .bash_profile	The personal initialization file, executed for login shells
~/ .bashrc	The individual per-interactive-shell startup file
~/ .bash_logout	The individual login shell cleanup file, executed when a login shell exits
~/ .inputrc	Individual readline initialization file

[ Module 6: Interactive Shell Environment ] - Page 20 •

## Giving nicknames with alias

- An **alias** is a name that represents another command, use to simplify frequently used commands. For example, the following command sets up an alias for the rm command:  

```
alias rm="rm -i"
alias ls="ls -l"
```
- An alias allows you to use the specified alias name instead of the full command. Make an alias lll which actually executes ls -la.  

```
alias lll="ls -la"
```
- You can tell which "ls" command is in your path with the built-in which command. For example,  

```
which ls
```
- These aliases can be included in the startup files to make them available every time you log into the system. Make a backup copy of your .profile file in your user's home directory. Add the following lines to .profile:  

```
HISTSIZE=75
export HISTSIZE
# some useful aliases
alias ls="ls -F"
alias who="who | sort"
alias cp="/bin/cp -i"
alias rm="/bin/rm -i"
```
- Add your own favourite aliases to the appropriate startup script. NOTE that the aliases for cp and rm are redundant with  

```
set -o noclobber
```
- Log out and in. What can you observe about your environment? Where would the administrator need to make the changes to provide all users with the same default environment?

# Table of Contents

Common Shell Features.....	2	Using arguments to previous command .....	13
The Shell .....	3	Configuration variables.....	13
Choosing which Shell to Use.....	4	Environment variables.....	14
Shell Scripting.....	6	Exporting Shell Variables to the Environment .....	15
Start off with a SHA-BANG (#!) .....	6	The PATH Environment Variable .....	16
echo Command .....	8	PROMPTING: Customizing your Bash Prompt (PS1) ....	17
Shell Variables .....	9	Shell Setup Files.....	19
Variables .....	9	Customizing User's bash Shell .....	20
Assignment of values to variables .....	9	Giving nicknames with alias .....	21
Variable Evaluation .....	10		
Quotation .....	10		
Shell variables: positional parameters (Command-line Arguments) .....	12		