Reading 02

☆ Book: The Linux Command Line ☆

☆ Chapter 16: Networking

- Command: 'ping' Ex: `ping linuxcommand.org`
  - Sends a special network packet called ICMP ECHO_REQUEST to a specified host.
    - ↳ once started it continues to send packets at a specified interval (default = 1 sec) until it is interrupted
    - ↳ after it is interrupted it will print some statistics
      - ↳ A properly performing network will exhibit 0 percent packet loss.

- Command: 'traceroute' Ex: `traceroot slashdot.org`
  - lists all the "hops" network traffic takes to get from the local system to a specified host.
    - ↳ basically what routers were traversed
    - ↳ displays router's hostname, IP address, and performance data
    - ↳ sometimes router's block displaying data and just show "* * *"
      - ↳ you can potentially overcome this by adding the -T or -I flag.

- Command: 'ip' Ex: `ip address show`
  - ↳ used to look at a variety of features on our network setup.
    - ↳ important things to look for when doing casual network diagnostics:
      - ↳ phrase 'state UP' in the first line for the interface, indicating that it is enabled
      - ↳ presence of a valid IP address in the 'inet' field on the third line.

Ex: `ip route show`

- ↳ shows typical routing table
- ↳ IP addresses that end in zero refer to networks rather than individual hosts

General Usage: `ip [-options] object [command]`

Transporting files over a network

- Command 'ftp' Ex: `ftp fileserver`
  - ↳ "file transfer protocol"
  - ↳ used to be the main method for downloading files from internet.
  - ↳ was insecure, so they made account names and passwords anonymous

- Command: 'lftp'
  - ↳ a better 'ftp'
  - ↳ works much like traditional ftp
  - ↳ improvements include: multiple protocol support (including HTTP), automatic retry on downloads, tab completion of path names

- Command 'curl' Ex: `curl https://linuxcommand.org`
  - ↳ we specify a URL and curl downloads the first page of the URL and outputs it to stdout.
  - ↳ multiple URLs can be specified.
  - ↳ supports most network protocols
  - ↳ Common 'curl' options:

Table 16-2: Common curl options

Option	Description
`-o, --output` *`file`*	Send output to the specified *file* rather than standard output.
`-O, --remote-name`	Like `-o` but name local file the same as the name of the remote file.
`-s, --silent`	Suppress the progress meter and error messages.
`-u, --proxy-user` *`user:password`*	Specify a user name/password combination.
`-v, --verbose`	Display verbose messages as it executes.

The `curl` man page covers all the gruesome details.

Command: 'wget' Ex: `wget http://linuxcommand.org/index.php`

↳ non-interactive network downloader

↳ useful for downloading content from web and FTP sites.

↳ can download single files, multiple files, and entire sites

↳ options allow you to recursively download, download files in the background (allowing you to log off but continue downloading), and complete the download of a partially downloaded file.

Secure Communication with Remote Hosts

Command: 'ssh'

↳ Secure Shell (ssh) was developed to solve the 2 basic problems of secure communication with a remote host:

1. It authenticates that the remote host is who it says it is
2. It encrypts all of the communications between the local and remote hosts

↳ SSH consists of 2 parts:

1. An SSH server runs on the remote host, listening for incoming connections, by default, on port 22.
2. An SSH client is used on the local system to communicate with the remote server.

↳ The following are examples of only running ONE command on a remote system with the hostname "remote-sys"

↳ Ex: `ssh remote-sys 'ls *' > dirlist.txt`

↳ performs ls on remote system and redirects output to dirlist.txt on local system

↳ Ex: `ssh remote-sys 'ls * > dirlist.txt'`

↳ performs ls on remote system and redirects output to dirlist.txt on remote system.

Tunneling with SSH

↳ When you ssh, an encrypted tunnel is created between the local and remote systems.

↳ this allows commands typed locally to be transmitted safely to the remote system and for results to be safely transmitted back

↳ additionally ssh protocol allows most types of network traffic to be sent through the encrypted tunnel, creating a sort of virtual private network (VPN) between the local and remote systems.

↳ running 'xload' on remote system allows you to see the program's graphical output on your local system.

scp and sftp

- programs in the OpenSSH package that can use an SSH-encrypted tunnel to copy files across the network.

- Command: 'scp' Ex: `scp remote-sys:document.txt`

↳ scp (secure copy) is used much like 'cp'.

↳ most notable difference is that source/destination pathnames may be preceded by name of remote host, followed by a colon character.

↳ The Ex copies document.txt from our home directory on the remote system (remote-sys), to the current working directory on our local system

- Command: 'sftp'

↳ secure version of 'ftp'

↳ uses ssh encrypted tunnel instead of transmitting everything in clear text.

↳ does NOT require an FTP server to be running on the remote host

↳ this means that any remote machine that can connect with the SSH client can also be used as an FTP-like server.

↳ Example Session:

```
[me@linuxbox ~]$ sftp remote-sys
Connecting to remote-sys...
me@remote-sys's password:
sftp> ls
ubuntu-24.04-desktop-amd64.iso
sftp> lcd Desktop
sftp> get ubuntu-24.04-desktop-amd64.iso
Fetching /home/me/ubuntu-24.04-desktop-amd64.iso to ubuntu-24.04-
desktop-amd64.iso
/home/me/ubuntu-24.04-desktop-amd64.iso 100%  699MB   7.4MB/s   01:35
sftp> bye
```

☆ Chapter 17 Searching for Files

- Command: 'locate' Ex: `locate bin/zip`
  - ↳ in the Ex locate searches its database of pathnames and outputs any that contain the string bin/zip.
  - ↳ you can obviously combine commands as well Ex: `locate zip | grep bin`
  - ↳ there are variants (check man page)
  - ↳ the locate database is created by another program named 'updatedb'
    - ↳ 'updatedb' is typically run once a day using cron.

- Command 'find' Ex: `find ~ type d`
  - ↳ searches a given directory (and its subdirectories) for files based on a variety of attributes.
  - ↳ the Ex returns a list of all directories from the home directory.
  - ↳ common file types:

File Type	Description
b	Block special device file
c	Character special device file
d	Directory
f	Regular file
l	Symbolic link

  - ↳ Ex: `find ~ -type f -name "*.JPG" -size +1M | wc -l`
    - ↳ counting files that end in ".JPG" and have a size greater than 1 megabyte
  - ↳ Common find size units:

Table 17-2: `find` Size Units

Character	Unit
b	512-byte blocks. This is the default if no unit is specified.
c	Bytes.
w	2-byte words.
k	Kilobytes (units of 1,024 bytes).
M	Megabytes (units of 1,048,576 bytes).
G	Gigabytes (units of 1,073,741,824 bytes).

  - ↳ find has lots of tests (see man page)

↳ you can also use operators with 'find'

Ex: `find ~ \(.type f -not -perm 0600\) -or \(-type d -not -perm 0700 \)`

↳ logical operators:

Table 17-4: `find` Logical Operators

Operator	Description
`-and`	Match if the tests on both sides of the operator are true. This can be shortened to `-a`. Note that when no operator is present, `-and` is implied by default.
`-or`	Match if a test on either side of the operator is true. This can be shortened to `-o`.
`-not`	Match if the test following the operator is false. This can be abbreviated with an exclamation point (`!`).
`( )`	Groups tests and operators together to form larger expressions. This is used to control the precedence of the logical evaluations. By default, `find` evaluates from left to right. It is often necessary to override the default evaluation order to obtain the desired result. Even if not needed, it is helpful sometimes to include the grouping characters to improve the readability of the command. Note that since the parentheses have special meaning to the shell, they must be quoted when using them on the command line to allow them to be passed as arguments to `find`. Usually the backslash character is used to escape them. It is also important that the `(` and `)` characters be surrounded with spaces to separate them from other words in the command. For example, `find ~ \( -type f \)`

↳ Ex: `find -type f -and -not -perm 0600`

↳ searching for files with NOT good permissions (with good permissions for files defined as 0600)

↳ preceeding a character with a backslash (\) tells the shell to ignore the following character.

↳ in Ex: `expression1 -and expression2`

↳ if expression1 is false, expression2 will not be evaluated

↳ find also allows actions

↳ Some predefined actions:

Table 17-6: Predefined `find` Actions

Action	Description
`-delete`	Delete the currently matching file.
`-ls`	Perform the equivalent of `ls -dils` on the matching file. Output is sent to standard output.
`-print`	Output the full pathname of the matching file to standard output. This is the default action if no other action is specified.
`-quit`	Quit once a match has been made.

↳ Ex: `find ~ -type f -name '*.bak' -delete`

↳ the order of commands does matter!

↳ Ex: `find ~ -type f -name '*.bak' -print`

↳ remember '-and' is implied

↳ -name '*.bak' only performed if -type f is true

↳ likewise -print only performed if previous two tests passed.

↳ find also allows user-defined actions

↳ general template: `-exec command '{}' ';'` or prefered `-ok command '{}' ';'`

↳ command is the desired command (ls, rm, etc.)

↳ '{}' is where the path that find returns is entered (automatically)

↳ ';' is a required delimiter indicating the end of the command.

↳ use `-ok` instead of `-exec` so that the command is interactive.

↳ also if you use the ';' at the end it runs each command one by one (inefficient)

↳ if you use '+' instead of ';' then the results of the search are combined into a single argument list for a single execution of the desired command.

- Command 'xargs' Ex: `find ~ type f -name 'foo*' -print | xargs ls -l`

↳ accepts input from Standard input and converts it into an argument list for a specified command.

↳ if the list is too long, the command runs with the max number of arguments and then repeats the process until all the standard input is exhausted.

↳ Ex: `find ~ iname '*.jpg' -print0 | xargs --null ls -l`

↳ ensures all files, even those containing embedded spaces in their names, are handled correctly.

- Example on how to create 100 subdirectories each containing 26 empty files in 2 lines!

```
[me@linuxbox ~]$ mkdir -p playground/dir-{001..100}
[me@linuxbox ~]$ touch playground/dir-{001..100}/file-{A..Z}
```

- There are also options that are used to control the scope of a find search:

Option	Description
`-depth`	Direct `find` to process a directory's files before the directory itself. This option is automatically applied when the `-delete` action is specified.
`-maxdepth` *levels*	Set the maximum number of levels that `find` will descend into a directory tree when performing tests and actions.
`-mindepth` *levels*	Set the minimum number of levels that `find` will descend into a directory tree before applying tests and actions.
`-mount`	Direct `find` not to traverse directories that are mounted on other filesystems.
`-noleaf`	Direct `find` not to optimize its search based on the assumption that it is searching a Unix-like filesystem. This is needed when scanning DOS/Windows filesystems and CD-ROMs.

Reading 02 Continued

★ Writing Shell Scripts from linuxcommand.org ★

★ 1. Writing Our First Script and Getting it to Work

- To successfully write a shell script, you have to:
  1. Write a script
  2. Give the shell permission to execute it
  3. Put it somewhere the shell can find it

- A shell script is a file that contains ASCII text.

- First Script (hello_world)

```
#!/bin/bash
# My first script
echo "Hello World!"
```

- special construct called a shebang
- indicates what program is to be used to interpret the script
  - In this case its /bin/bash

← comment

echo ← command

↳ Need to give hello_world execute permissions via `chmod 755 hello_world`

↳ then run via `./hello_world`

↳ to run the script, it doesn't need a path because it searches all paths that show up when you use `echo $PATH`

#2: Editing the Scripts We already have

- During our shell session, the system is holding a number of facts about the world in its memory.
  - ↳ this info is called the environment.
    - ↳ the environment contains our path, username, and much more.
      - ↳ use set command to examine a complete list of whats in the environment.
        - ↳ 2 types of commands often in the environment:
          1. aliases
          2. Shell functions

- When we log on to the system, the bash program starts, and reads in a series of configuration scripts called the startup files. These define the default environment shared by all users.

- 2 types of shell sessions:

1. login:

File	Contents
/etc/profile	A global configuration script that applies to all users.
~/.bash_profile	A user's personal startup file. Can be used to extend or override settings in the global configuration script.
~/.bash_login	If `~/.bash_profile` is not found, bash attempts to read this script.
~/.profile	If neither `~/.bash_profile` nor `~/.bash_login` is found, bash attempts to read this file. This is the default in Debian-based distributions, such as Ubuntu.

2. non-login:

File	Contents
/etc/bash.bashrc	A global configuration script that applies to all users.
~/.bashrc	A user's personal startup file. Can be used to extend or override settings in the global configuration script.

Ex of shell script:

```
# .bash_profile
# Get the aliases and functions
if [ -f ~/.bashrc ]; then
  . ~/.bashrc
fi

# User specific environment and startup programs
PATH=$PATH:$HOME/bin
export PATH
```

if ~/.bashrc exists, then read ~/.bashrc

Setting PATH variable to add the ~/bin directory to the path

Export command tells the shell to make contents of the PATH variable available to child process of this shell.

Aliases:

- An alias is an easy way to create a new command which acts as an abbreviation for a longer one.
  - ↳ General form: `alias name=value`
  - ↳ Ex: `alias l='ls -l'`

Shell functions

- To create something more complex we need shell functions.
  - ↳ these can be thought of as "scripts within scripts"
- Ex:

```
today() {
    echo -n "Today's date is: "
    date +"%A, %B %-d, %Y"
}
```

← basically aliases "today" to the contents of the function

☆ 3: Here Scripts

- A here script (here document) is an additional form of I/O redirection. It provides a way to include content that will be given to the standard input of a command.

A here script is constructed like this:

```
command << token
content to be used as command's standard input
token
```

- "token" can be any string of characters, "_EOF_" is traditional.
- token at the start and end must be the same.

Script:

```
#!/bin/bash

# sysinfo_page - A script to produce an HTML file

cat << _EOF_
<html>
<head>
    <title>
    The title of your page
    </title>
</head>

<body>
    Your page content goes here.
</body>
</html>
_EOF_
```

Script:

Causes bash to ignore leading tabs, but not spaces, in the here script.

```
#!/bin/bash

# sysinfo_page - A script to produce an HTML file

cat <<- _EOF_
    <html>
    <head>
        <title>
        The title of your page
        </title>
    </head>

    <body>
        Your page content goes here.
    </body>
    </html>
_EOF_
```

★ 4: Variables

- Assigning Variables in shell scripts:

  `Variable_name=value` No Spaces Allowed!!

- Using variable:

  `$variable`

- Variable Name Rules:

  1. Must start with letter
  2. No spaces
  3. No punctuation marks

- Environmental Variables
  - Variables defined in startup files
  - Use command 'printenv' to see these
  - Can use these in shell scripts

Continued Example:

```
#!/bin/bash

# sysinfo_page - A script to produce an HTML file

title="System Information for"

cat <<- _EOF_
    <html>
    <head>
        <title>
        $title $HOSTNAME
        </title>
    </head>

    <body>
    <h1>$title $HOSTNAME</h1>
    </body>
    </html>
_EOF_
```

★ 5: Command Substitutions and Constants

- Command Substitution is a technique in which you use "$()" to tell the shell "substitute the results of the enclosed command.

We can also assign the results of a command to a variable:

```
right_now="$(date +"%x %r %Z")"
```

We can even nest the variables (place one inside another), like this:

```
right_now="$(date +"%x %r %Z")"
time_stamp="Updated on $right_now by $USER"
```

- Constants
  - Constants are values that should NOT change
  - Using ALL CAPS to define them is the convention.
    - ↳ this is why environmental variables are ALL CAPS.

Continued Example:

```
#!/bin/bash

# sysinfo_page - A script to produce an HTML file

title="System Information for $HOSTNAME"
RIGHT_NOW="$(date +"%x %r %Z")"
TIME_STAMP="Updated on $RIGHT_NOW by $USER"

cat <<- _EOF_
    <html>
    <head>
        <title>
        $title
        </title>
    </head>

    <body>
    <h1>$title</h1>
    <p>$TIME_STAMP</p>
    </body>
    </html>
_EOF_
```

★ 6: Shell Functions

- top-down design is the process of identifying the top-level steps and developing increasingly detailed views of those steps.

- for top-down design we utilize shell functions. Ex: `system_info(){ ... }`

- Functions:
  - must appear (be defined) before we attempt to use them.
  - the function body must contain at least one valid command

- Stubbing is a technique that works like this:
  - imagine you are creating a function called "system_info" but haven't figured out how to code it yet.
    - ↳ rather than hold up the development of the script until we are finished with "system_info" we just add an 'echo' command like this:

```
system_info()
{
    # Temporary function stub
    echo "function system_info"
}
```

    - ↳ this way our script will execute successfully, even though we do not yet have a finished "system_info" function.

Continued Example!

```
#!/bin/bash

# sysinfo_page - A script to produce an system information HTML file

##### Constants

TITLE="System Information for $HOSTNAME"
RIGHT_NOW="$(date +"%x %r %Z")"
TIME_STAMP="Updated on $RIGHT_NOW by $USER"

##### Functions

system_info()
{
    # Temporary function stub
    echo "function system_info"
}


show_uptime()
{
    # Temporary function stub
    echo "function show_uptime"
}


drive_space()
{
    # Temporary function stub
    echo "function drive_space"
}


home_space()
{
    # Temporary function stub
    echo "function home_space"
}
```

```
##### Main

cat <<- _EOF_
  <html>
  <head>
      <title>$TITLE</title>
  </head>

  <body>
      <h1>$TITLE</h1>
      <p>$TIME_STAMP</p>
      $(system_info)
      $(show_uptime)
      $(drive_space)
      $(home_space)
  </body>
  </html>
_EOF_
```

☆ 7: Some Real Work

- defining functions for our script.

- show_uptime()
  - ↳ displaying output of the 'uptime' command.
  - ↳ 'uptime' command outputs several interesting facts about the system, including the length of time the system has been "up" (running) since last reboot, the number of users and recent system load.
  - ↳ uptime:

```
[me@linuxbox me]$ uptime
9:15pm up 2 days, 2:32, 2 users, load average: 0.00, 0.00, 0.00
```

  - ↳ show_uptime():

```
show_uptime()
{
    echo "<h2>System uptime</h2>"
    echo "<pre>"
    uptime
    echo "</pre>"
}
```

- drive_space()
  - ↳ the 'df' command provides a summary of the space used by all of the mounted file systems.
  - ↳ 'df':

```
[me@linuxbox me]$ df
Filesystem   1k-blocks      Used Available Use% Mounted on
/dev/hda2       509992    225772    279080  45% /
/dev/hda1        23324      1796     21288   8% /boot
/dev/hda3     15739176   1748176  13832360  12% /home
/dev/hda5      3123888   3039584     52820  99% /usr
```

  - ↳ drive_space():

```
drive_space()
{
    echo "<h2>Filesystem space</h2>"
    echo "<pre>"
    df
    echo "</pre>"
}
```

- home_space()
  - ↳ displays the amount of space each user is using in their home directory.
  - ↳ it will display this as a list, sorted in descending order by the amount of space used.
  - ↳ home_space():

```
home_space()
{
    echo "<h2>Home directory space by user</h2>"
    echo "<pre>"
    echo "Bytes Directory"
    du -s /home/* | sort -nr
    echo "</pre>"
}
```

  - ↳ NOTE: In order for this function to properly execute, the script must be run by the superuser, since 'du' command is used.

- system_info()
  - ↳ not ready to finish this yet
  - ↳ just improving the stubbins so it produces valid HTML.
  - ↳ system_info():

```
system_info()
{
    echo "<h2>System release info</h2>"
    echo "<p>Function not yet implemented</p>"
}
```

☆ 8: Flow Control - Part 1

- **if** general syntax:

```
if commands; then
    commands
[elif commands; then
    commands...]
[else
    commands]
fi
```

- Commands issue a value to the system when they terminate, called an exit status.
  - ↳ this value ranges from [0, 255]
    - ↳ By convention 0 indicates success, any other number indicates failure.
  - ↳ Example of checking the exit status:

```
[me@linuxbox ~]$ ls -d /usr/bin
/usr/bin
[me@linuxbox ~]$ echo $?
0
[me@linuxbox ~]$ ls -d /bin/usr
ls: cannot access /bin/usr: No such file or directory
[me@linuxbox ~]$ echo $?
2
```

- **test** often used in combination with 'if'.
  - ↳ if given expression is true, test exists with a status of 0 otherwise it exists with a status of 1.

Ex:

Spaces required

```
if [ -f .bash_profile ]; then
    echo "You have a .bash_profile. Things are fine."
else
    echo "Yikes! You have no .bash_profile!"
fi
```

↳ the test command checks "Is .bash_profile a file?"

↳ Table of test commands:

Expression	Description
-d *file*	True if *file* is a directory.
-e *file*	True if *file* exists.
-f *file*	True if *file* exists and is a regular file.
-L *file*	True if *file* is a symbolic link.
-r *file*	True if *file* is a file readable by you.
-w *file*	True if *file* is a file writable by you.
-x *file*	True if *file* is a file executable by you.
file1 -nt *file2*	True if *file1* is newer than (according to modification time) *file2*.
file1 -ot *file2*	True if *file1* is older than *file2*.
-z *string*	True if *string* is empty.
-n *string*	True if *string* is not empty.
string1 = *string2*	True if *string1* equals *string2*.
string1 != *string2*	True if *string1* does not equal *string2*.

- Also you can use a semicolon to break up commands.
  - Ex: `clear ; ls`

- exit Ex: exit 0 (Success) or exit 1 (failure)
  - ↳ Causes script to terminate immediately and sets the exit status to whatever is specified.

- Ex function:

prints numeric user id of the current user

```
function home_space {
    # Only the superuser can get this information
    if [ "$(id -u)" = "0" ]; then
        echo "<h2>Home directory space by user</h2>"
        echo "<pre>"
        echo "Bytes Directory"
            du -s /home/* | sort -nr
        echo "</pre>"
    fi
}   # end of home_space
```

☆ 9! Stay Out of Trouble

- it is ok to set a variable to nothing via `my_variable=`
- Always consider what happens downstream if a variable is set equal to nothing (in test expressions especially)
- Always put double quotes around parameters that undergo expansion.

Debugging Via Tracing!

↳ If we modify the first line by adding "-x" : `#!/bin/bash -x`

↳ The output will display each line (with expansion performed)

↳ Ex:

```
[me@linuxbox me]$ ./trouble.bash
+ number=1
+ '[' 1 = 1 ']'
+ echo 'Number equals 1'
Number equals 1
```

↳ you can also specify specific blocks to do this to using 'set -x' at the start and 'set +x' at the end

↳ Ex:

```
#!/bin/bash

number=1

set -x
if [ $number = "1" ]; then
    echo "Number equals 1"
else
    echo "Number does not equal 1"
fi
set +x
```

★ 10: Keyboard Input and Arithmetic

Read:

↳ Allows you to get user input

↳ Ex:

keeps cursor on same line

```
#!/bin/bash
echo -n "Enter some text > "
read text
echo "You entered: $text"
```

→

```
[me@linuxbox me]$ read_demo.bash
Enter some text > this is some text
You entered: this is some text
```

↳ Options:

↳ '-p': Allows us to specify a prompt to precede the user's input:

↳ Ex:

```
#!/bin/bash

read -p "Enter some text > " text
echo "You entered: $text"
```

↳ '-t': Sets time limit (in seconds) until the script just moves on (in case no user input)

↳ Ex:

```
#!/bin/bash

echo -n "Hurry up and type something! > "
if read -t 3 response; then
    echo "Great, you made it in time!"
else
    echo "Sorry, you are too slow!"
fi
```

↳ '-s': Causes users typing not to be displayed

- Example of arithmetic:

```
#!/bin/bash

first_num=0
second_num=0

read -p "Enter the first number --> " first_num
read -p "Enter the second number -> " second_num

echo "first number + second number = $((first_num + second_num))"
echo "first number - second number = $((first_num - second_num))"
echo "first number * second number = $((first_num * second_num))"
echo "first number / second number = $((first_num / second_num))"
echo "first number % second number = $((first_num % second_num))"
echo "first number raised to the"
echo "power of the second number   = $((first_num ** second_num))"
```

☆ 11: Flow Control - Part 2

Case:

↳ general form:

```
case word in
    patterns ) commands ;;
esac
```

↳ if to case example:

```
#!/bin/bash

read -p "Enter a number between 1 and 3 inclusive > " character
if [ "$character" = "1" ]; then
    echo "You entered one."
elif [ "$character" = "2" ]; then
    echo "You entered two."
elif [ "$character" = "3" ]; then
    echo "You entered three."
else
    echo "You did not enter a number between 1 and 3."
fi
```

→

```
#!/bin/bash

read -p "Enter a number between 1 and 3 inclusive > " character
case $character in
    1 ) echo "You entered one."
        ;;
    2 ) echo "You entered two."
        ;;
    3 ) echo "You entered three."
        ;;
    * ) echo "You did not enter a number between 1 and 3."
esac
```

- While loop:

```
#!/bin/bash

number=0
while [ "$number" -lt 10 ]; do
    echo "Number = $number"
    number=$((number + 1))
done
```

"less than"

- until command (intuitive)

```
#!/bin/bash

number=0
while [ "$number" -lt 10 ]; do
    echo "Number = $number"
    number=$((number + 1))
done
```

- Example of menu system:

```
#!/bin/bash

press_enter()
{
    echo -en "\nPress Enter to continue"
    read
    clear
}

selection=
until [ "$selection" = "0" ]; do
    echo "
    PROGRAM MENU
    1 - display free disk space
    2 - display free memory

    0 - exit program
"
    echo -n "Enter selection: "
    read selection
    echo ""
    case $selection in
        1 ) df ; press_enter ;;
        2 ) free ; press_enter ;;
        0 ) exit ;;
        * ) echo "Please enter 1, 2, or 0"; press_enter
    esac
done
```

Reading 02 continued...

★ 12: Positional Parameters

- Positional parameters are a series of special variables ($0 through $9) that contain the contents of the command line.

↳ Ex:

```
[me@linuxbox me]$ some_program word1 word2 word3
```

↳ $0 contains "word1"

↳ $1 contains "word2"

↳ $2 contains "word3"

↳ the shell maintains a variable called $# that contains the number of items on the command line in addition to the name of the command $0.

↳ Ex: checking if $1 contains anything:

```
#!/bin/bash

if [ "$1" != "" ]; then
    echo "Positional parameter 1 contains something"
else
    echo "Positional parameter 1 is empty"
fi
```

↳ Ex: checking if command line arguments were used:

```
#!/bin/bash

if [ $# -gt 0 ]; then
    echo "Your command line contains $# arguments"
else
    echo "Your command line contains no arguments"
fi
```

↳ 'shift' is a command that shifts all the positional parameters down by one.

↳ $2 becomes $1

↳ $3 becomes $2 ...

↳ new code to implement into our running program!

```
interactive=
filename=~/sysinfo_page.html

while [ "$1" != "" ]; do
    case $1 in
        -f | --file )           shift
                                filename="$1"
                                ;;
        -i | --interactive )    interactive=1
                                ;;
        -h | --help )           usage
                                exit
                                ;;
        * )                     usage
                                exit 1
    esac
    shift
done
```

☆ 13: Flow Control - Part 3

- For loops

↳ General Syntax:

```
for variable in words; do
    commands
done
```

Ex:

```
#!/bin/bash

for i in word1 word2 word3; do
    echo "$i"
done
```

Ex: displaying character count of each word:

```
#!/bin/bash

count=0
for i in $(cat ~/.bash_profile); do
    count=$((count + 1))
    echo "Word $count ($i) contains $(echo -n $i | wc -c) characters"
done
```

Ex: loop for command line arguments

Shell variable that contains a list of command line arguments.

```
#!/bin/bash

for filename in "$@"; do
    result=
    if [ -f "$filename" ]; then
        result="$filename is a regular file"
    else
        if [ -d "$filename" ]; then
            result="$filename is a directory"
        fi
    fi
    if [ -w "$filename" ]; then
        result="$result and it is writable"
    else
        result="$result and it is not writable"
    fi
    echo "$result"
done
```

- Note! $@ is assumed if the in clause is omitted.

- updated homespace():

```
home_space() {
    echo "<h2>Home directory space by user</h2>"
    echo "<pre>"
    format="%8s%10s%10s   %-s\n"
    printf "$format" "Dirs" "Files" "Blocks" "Directory"
    printf "$format" "----" "-----" "------" "---------"
    if [ $(id -u) = "0" ]; then
        dir_list="/home/*"
    else
        dir_list=$HOME
    fi
    for home_dir in $dir_list; do
        total_dirs=$(find $home_dir -type d | wc -l)
        total_files=$(find $home_dir -type f | wc -l)
        total_blocks=$(du -s $home_dir)
        printf "$format" "$total_dirs" "$total_files" "$total_blocks"
    done
    echo "</pre>"

}   # end of home_space
```

- Updated system_info():

```
system_info() {
    # Find any release files in /etc

    if ls /etc/*release 1>/dev/null 2>&1; then
        echo "<h2>System release info</h2>"
        echo "<pre>"
        for i in /etc/*release; do

            # Since we can't be sure of the
            # length of the file, only
            # display the first line.

            head -n 1 "$i"
        done
        uname -orp
        echo "</pre>"
    fi

}   # end of system_info
```

- The 'uname -orp' command gives some additional info on the system

☆ 14: Errors and Signals and Traps Part 1 !

- Always check and return exit status when needed.
  - ↳ be precautionary.
- An error function can be useful:
  - ↳ Ex:

```
# An error exit function

error_exit()
{
  echo "$1" 1>&2
  exit 1
}

# Using error_exit

if cd "$some_directory"; then
  rm ./*
else
  error_exit "Cannot change directory! Aborting."
fi
```

- Ex: how AND / OR operators work, and when each clause is / is not executed:

```
[me@linuxbox]$ true || echo "echo executed"
[me@linuxbox]$ false || echo "echo executed"
echo executed
[me@linuxbox]$ true && echo "echo executed"
echo executed
[me@linuxbox]$ false && echo "echo executed"
[me@linuxbox]$
```

- Ex: `${1:- "Unknown Error"}`
  - ↳ if $1 is undefined, substitute the string "Unknown Error" in its place.

☆ 15. Errors and Signals and Traps Part 2

- The 'trap' command allows us to execute a command when our script recieves a signal.
  - ↳ general form: `trap arg signals`
    - ↳ "signals" is a list of signals to intercept and "arg" is a command to execute when one of the signals is recieved.
  - ↳ Ex:

```
#!/bin/bash

# Program to print a text file with headers and footers

TEMP_FILE=/tmp/printfile.txt

trap "rm $TEMP_FILE; exit" SIGHUP SIGINT SIGTERM

pr $1 > "$TEMP_FILE"

read -p "Print file? [y/n]: "
if [ "$REPLY" = "y" ]; then
  lpr "$TEMP_FILE"
fi
rm "$TEMP_FILE"
```

- Note, SIGKILL or Signal 9 cannot be trapped. SIGKILL is initiated by `Kill -9` and should be used as a last resort.

- Clean Up function
  - Since the trap command only takes a single string containing the command to be performed, using a clean_up() function is a good decision.
    - ↳ Ex:

```
#!/bin/bash

# Program to print a text file with headers and footers

TEMP_FILE=/tmp/printfile.txt

clean_up() {

  # Perform program exit housekeeping
  rm "$TEMP_FILE"
  exit
}

trap clean_up SIGHUP SIGINT SIGTERM

pr $1 > "$TEMP_FILE"

read -p "Print file? [y/n]: "
if [ "$REPLY" = "y" ]; then
  lpr "$TEMP_FILE"
fi
clean_up
```

- Remember when (or if) you put files in the /tmp folder everyone can write files here, so its important to name the files something thats easily identifiable and unpredictable.