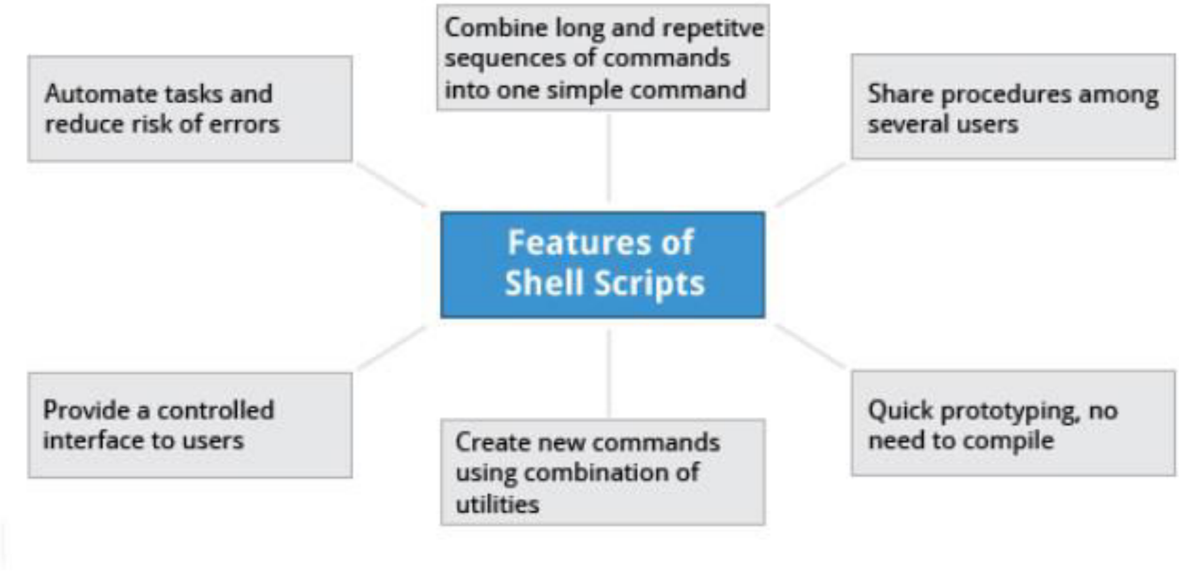
The Bash Shell and Basic Scripting

Shell Scripting:

1. Suppose you want to look up a filename, check if the associated file exists, and then respond accordingly, displaying a message confirming or not confirming the file's existence.
2. If you only need to do it once, you can just type a sequence of commands at a terminal. However, if you need to do this multiple times, automation is the way to go.
3. In order to automate sets of commands, you will need to learn how to write shell scripts.
4. Most commonly in Linux, these scripts are developed to be run under the bash command shell interpreter.
5. The graphic illustrates several of the benefits of deploying scripts.



Features of Shell Scripts

Automate tasks and reduce risk of errors

Combine long and repetitve sequences of commands into one simple command

Share procedures among several users

Provide a controlled interface to users

Create new commands using combination of utilities

Quick prototyping, no need to compile

Features of Shell Scripts

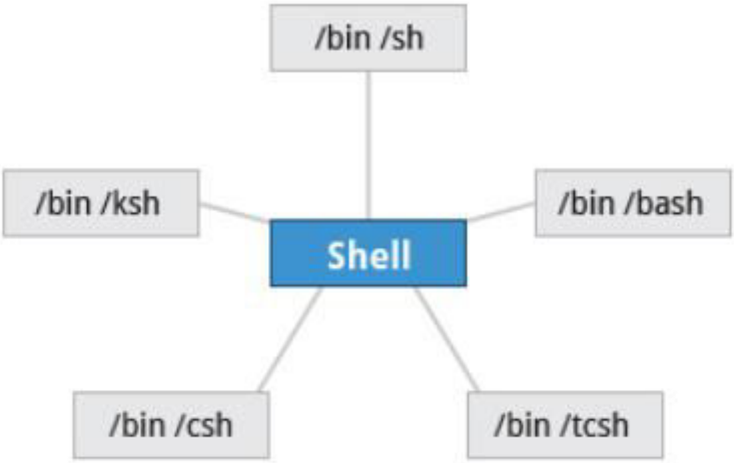
Command Shell Choices:

1. The command interpreter is tasked with executing statements that follow it in the script.
2. Commonly used interpreters include: **/usr/bin/perl, /bin/bash, /bin/csh, /usr/bin/python** and **/bin/sh**.
3. Typing a long sequence of commands at a terminal window can be complicated, time consuming, and error prone.
4. By deploying shell scripts, using the command line becomes an efficient and quick way to launch complex sequences of steps.
5. The fact that shell scripts are saved in a file also makes it easy to use them to create new script variations and share standard procedures with several users.

6. Linux provides a wide choice of shells; exactly what is available on the system is listed in /etc/shells. Typical choices are:

   /bin/sh
   /bin/bash
   /bin/tcsh
   /bin/csh
   /bin/ksh
   /bin/zsh

7. Most Linux users use the default bash shell, but those with long UNIX backgrounds with other shells may want to override the default.

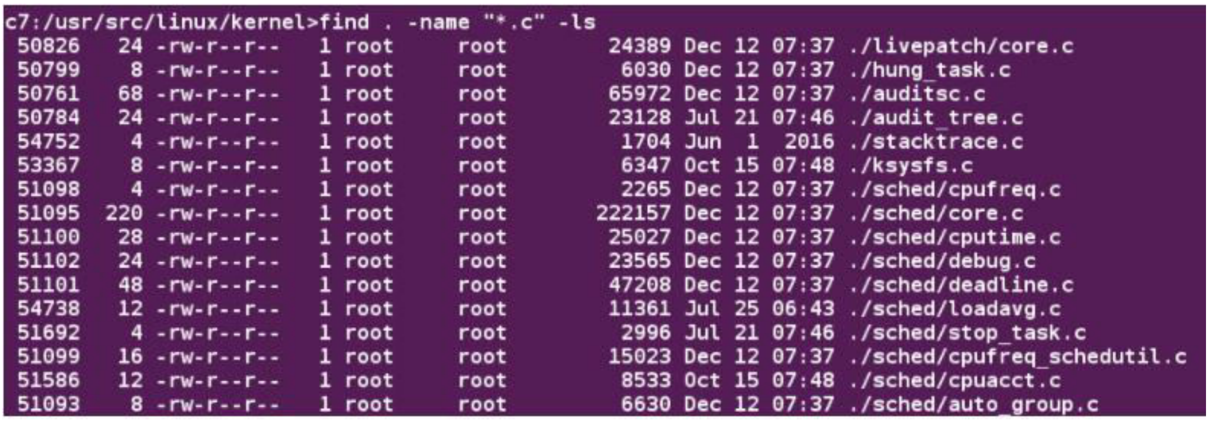


Command Shell Choices

Shell Scripts:

1. Remember from our earlier discussion, a shell is a command line interpreter which provides the user interface for terminal windows.
2. It can also be used to run scripts, even in non-interactive sessions without a terminal window, as if the commands were being directly typed in.
3. For example, typing **find . -name "*.c" -ls** at the command line accomplishes the same thing as executing a script file containing the lines:

   ```
   #!/bin/bash
   find . -name "*.c" -ls
   ```
4. The first line of the script, which starts with **#!**, contains the full path of the command interpreter (in this case /bin/bash) that is to be used on the file.
5. As we have noted, you have quite a few choices for the scripting language you can use, such as /usr/bin/perl, /bin/csh, /usr/bin/python, etc.



```
c7:/usr/src/linux/kernel>find . -name "*.c" -ls
 50826   24 -rw-r--r--   1 root     root        24389 Dec 12 07:37 ./livepatch/core.c
 50799    8 -rw-r--r--   1 root     root         6030 Dec 12 07:37 ./hung_task.c
 50761   68 -rw-r--r--   1 root     root        65972 Dec 12 07:37 ./auditsc.c
 50784   24 -rw-r--r--   1 root     root        23128 Jul 21 07:46 ./audit_tree.c
 54752    4 -rw-r--r--   1 root     root         1704 Jun  1  2016 ./stacktrace.c
 53367    8 -rw-r--r--   1 root     root         6347 Oct 15 07:48 ./ksysfs.c
 51098    4 -rw-r--r--   1 root     root         2265 Dec 12 07:37 ./sched/cpufreq.c
 51095  220 -rw-r--r--   1 root     root       222157 Dec 12 07:37 ./sched/core.c
 51100   28 -rw-r--r--   1 root     root        25027 Dec 12 07:37 ./sched/cputime.c
 51102   24 -rw-r--r--   1 root     root        23565 Dec 12 07:37 ./sched/debug.c
 51101   48 -rw-r--r--   1 root     root        47208 Dec 12 07:37 ./sched/deadline.c
 54738   12 -rw-r--r--   1 root     root        11361 Jul 25 06:43 ./sched/loadavg.c
 51692    4 -rw-r--r--   1 root     root         2996 Jul 21 07:46 ./sched/stop_task.c
 51099   16 -rw-r--r--   1 root     root        15023 Dec 12 07:37 ./sched/cpufreq_schedutil.c
 51586   12 -rw-r--r--   1 root     root         8533 Oct 15 07:48 ./sched/cpuacct.c
 51093    8 -rw-r--r--   1 root     root         6630 Dec 12 07:37 ./sched/auto_group.c
```

Shell Scripts

A Simple bash Script:

1. Let's write a simple bash script that displays a one line message on the screen. Either type:

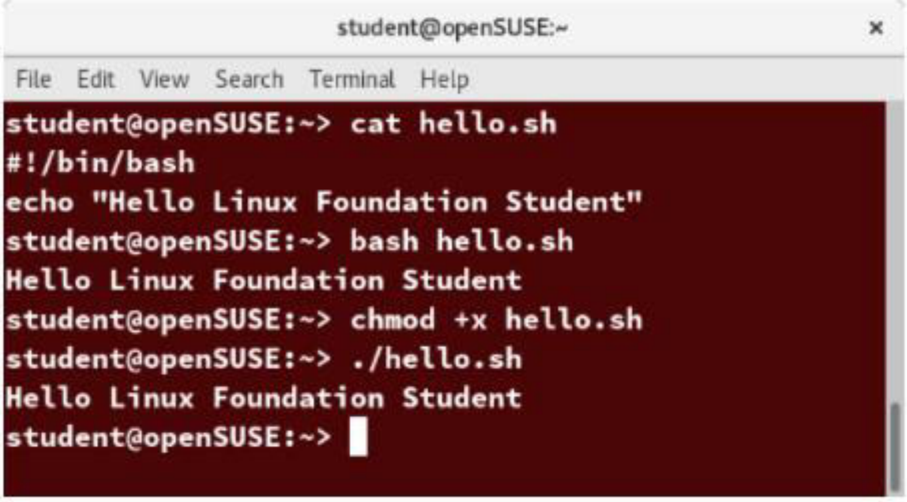
   I/P: $ cat > hello.sh

   O/P:

   #!/bin/bash

   echo "Hello Linux Foundation Student"

2. Now press **ENTER** and **CTRL-D** to save the file, or just create hello.sh in your favorite text editor.
3. Then, type **chmod +x hello.sh** to make the file executable by all users.
4. You can then run the script by typing ./hello.sh or by doing:
   $ bash hello.sh
   Hello Linux Foundation Student
5. **NOTE**: If you use the second form, you do not have to make the file executable.

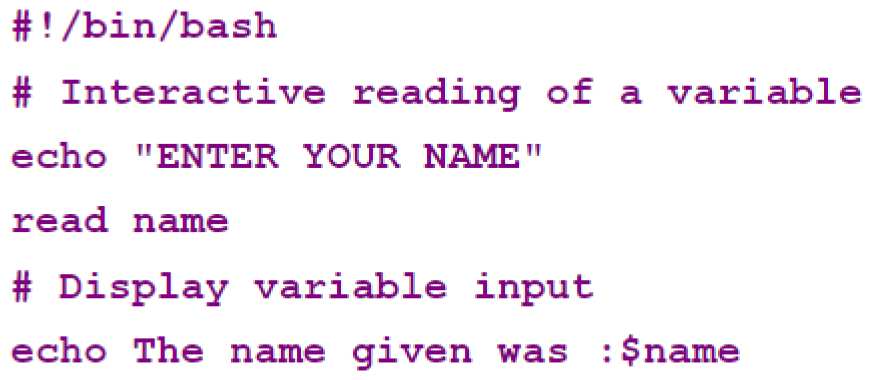


```
student@openSUSE:~> cat hello.sh
#!/bin/bash
echo "Hello Linux Foundation Student"
student@openSUSE:~> bash hello.sh
Hello Linux Foundation Student
student@openSUSE:~> chmod +x hello.sh
student@openSUSE:~> ./hello.sh
Hello Linux Foundation Student
student@openSUSE:~>
```

A Simple bash Script

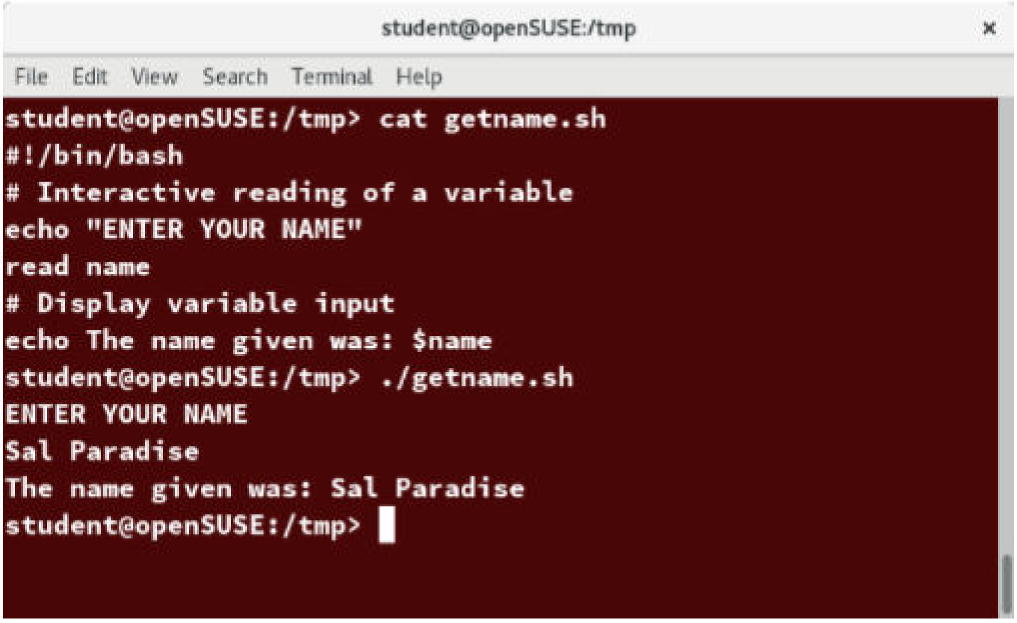
Interactive Example Using bash Scripts:

1. Now, let's see how to create a more interactive example using a bash script.
2. The user will be prompted to enter a value, which is then displayed on the screen.
3. The value is stored in a temporary variable, name. We can reference the value of a shell variable by using a **$** in front of the variable name, such as **$name**.
4. To create this script, you need to create a file named getname.sh in your favorite editor with the following content:



```
#!/bin/bash
# Interactive reading of a variable
echo "ENTER YOUR NAME"
read name
# Display variable input
echo The name given was :$name
```

5. Once again, make it executable by doing **chmod +x getname.sh**.
6. In the above example, when the user types **./getname.sh** and the script is executed, the user is prompted with the string ENTER YOUR NAME.
7. The user then needs to enter a value and press the **Enter** key. The value will then be printed out.
8. **NOTE**: The hash-tag/pound-sign/number-sign **(#)** is used to start comments in the script and can be placed anywhere in the line (the rest of the line is considered a comment). However, note the special magic combination of **#!**, used on the first line, is a unique exception to this rule.

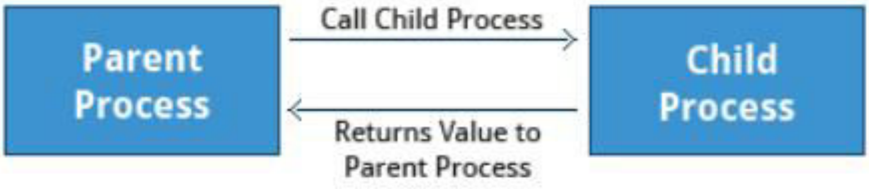


```
student@openSUSE:/tmp> cat getname.sh
#!/bin/bash
# Interactive reading of a variable
echo "ENTER YOUR NAME"
read name
# Display variable input
echo The name given was: $name
student@openSUSE:/tmp> ./getname.sh
ENTER YOUR NAME
Sal Paradise
The name given was: Sal Paradise
student@openSUSE:/tmp> 
```

Interactive Example Using bash Scripts

Return Values:

1. All shell scripts generate a return value upon finishing execution, which can be explicitly set with the exit statement.
2. Return values permit a process to monitor the exit state of another process, often in a parent-child relationship.
3. Knowing how the process terminates enables taking any appropriate steps which are necessary or contingent on success or failure.

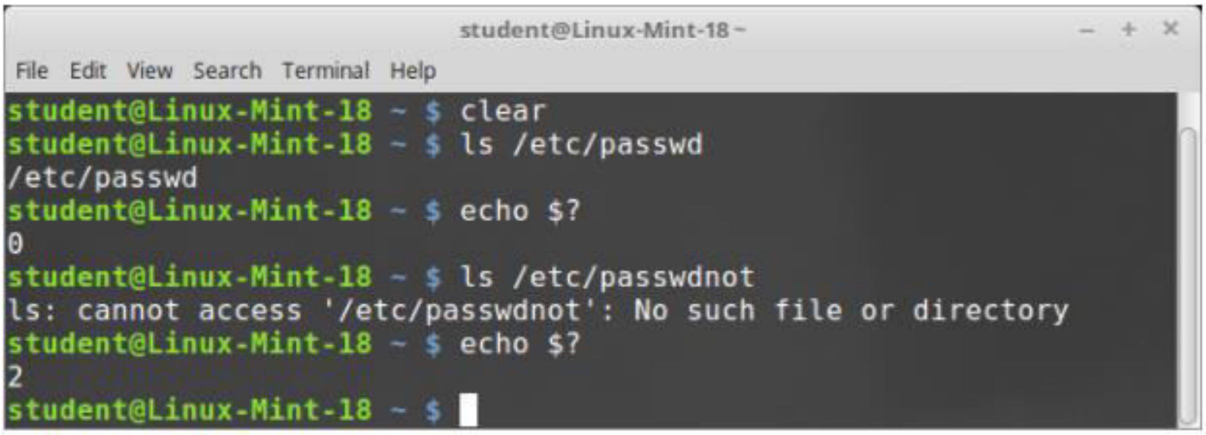


Return Values

Viewing Return Values:

1. As a script executes, one can check for a specific value or condition and return success or failure as the result.
2. By convention, success is returned as 0, and failure is returned as a non-zero value.
3. An easy way to demonstrate success and failure completion is to execute ls on a file that exists as well as one that does not, the return value is stored in the environment variable represented by **$?:**

4. In this example, the system is able to locate the file /etc/logrotate.conf and **ls** returns a value of **0** to indicate success.
5. When run on a non-existing file, it returns 2.
6. Applications often translate these return values into meaningful messages easily understood by the user.



```
student@Linux-Mint-18 ~ $ clear
student@Linux-Mint-18 ~ $ ls /etc/passwd
/etc/passwd
student@Linux-Mint-18 ~ $ echo $?
0
student@Linux-Mint-18 ~ $ ls /etc/passwdnot
ls: cannot access '/etc/passwdnot': No such file or directory
student@Linux-Mint-18 ~ $ echo $?
2
student@Linux-Mint-18 ~ $ 
```

Viewing Return Values

Lab: Exit Status Codes:

Write a script which:

- Does `ls` for a non-existent file, and then displays the resulting exit status.
- Creates a file and does `ls` for it, and then once again displays the resulting exit status.

Solution:

Lab Solution: Exit Status Codes

Create a file named `testls.sh`, with the content below.

```
#!/bin/bash
#
# check for non-existent file, exit status will be 2
#
ls SoMeFiLe.ext
echo "status: $?"

# create file, and do again, exit status will be 0
touch SoMeFiLe.ext
ls SoMeFiLe.ext
echo "status: $?"

# remove the file to clean up
rm SoMeFiLe.ext
```

Make it executable and run it:

```
student:/tmp> chmod +x testls.sh
student:/tmp> ./testls.sh

ls: cannot access SoMeFiLe.ext: No such file or directory
status: 2
SoMeFiLe.ext
status: 0
```