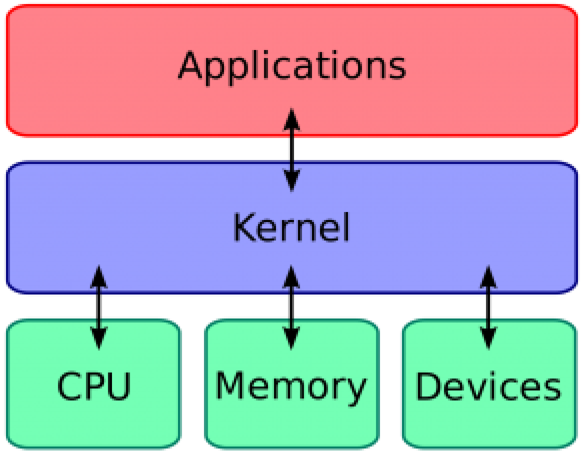
Plan

Chapter 2: Linux – Shell programming Part I

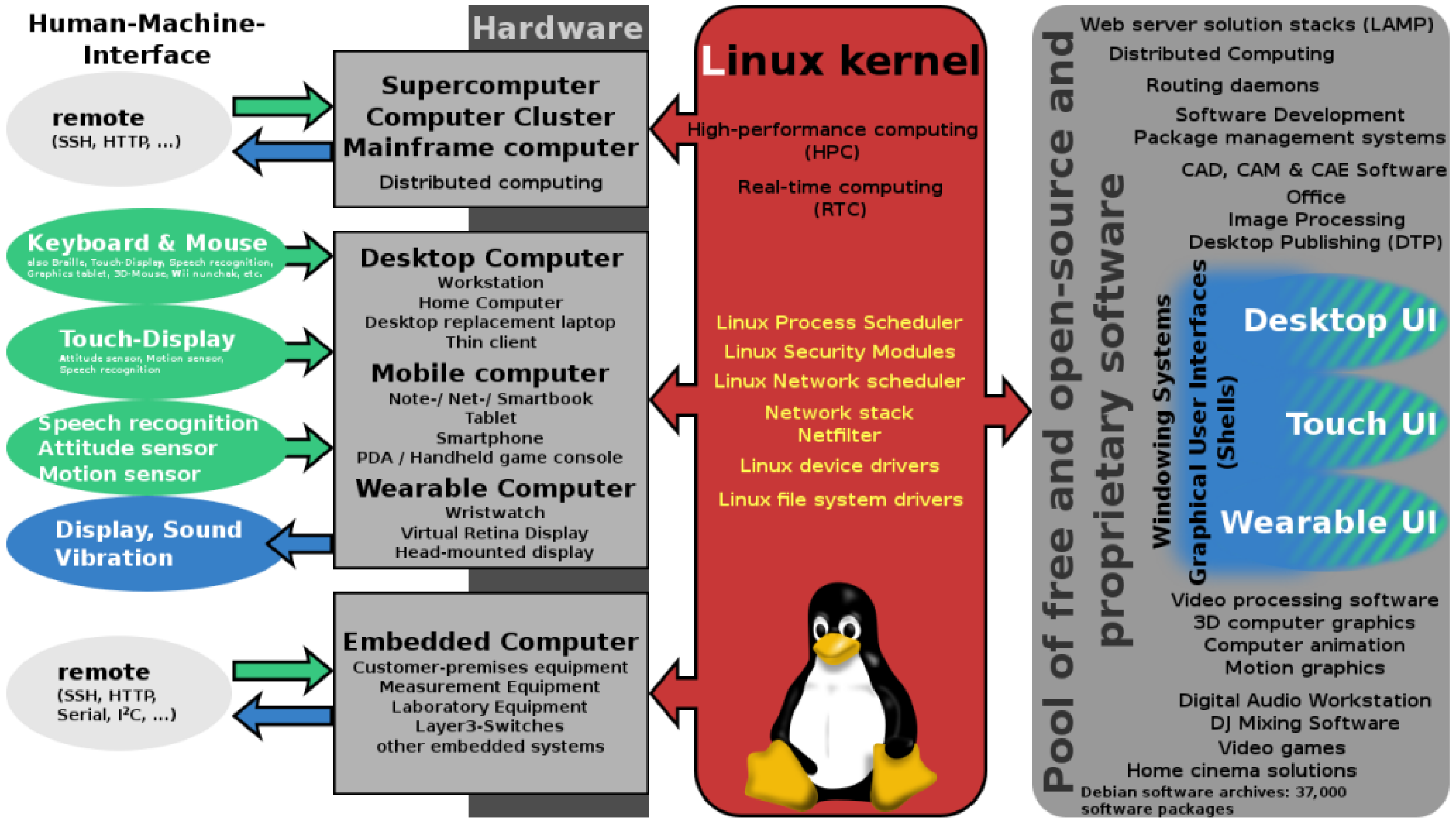
1. Kernel definition

- The kernel is a computer program that is the core of a computer's operating system, with complete control over everything in the system. It manages the following resources of the Linux system –
  - File management
  - Process management
  - I/O management
  - Memory management
  - Device management etc.



- **Complete Linux system** = Kernel + GNU system utilities and libraries + other management scripts + installation scripts.

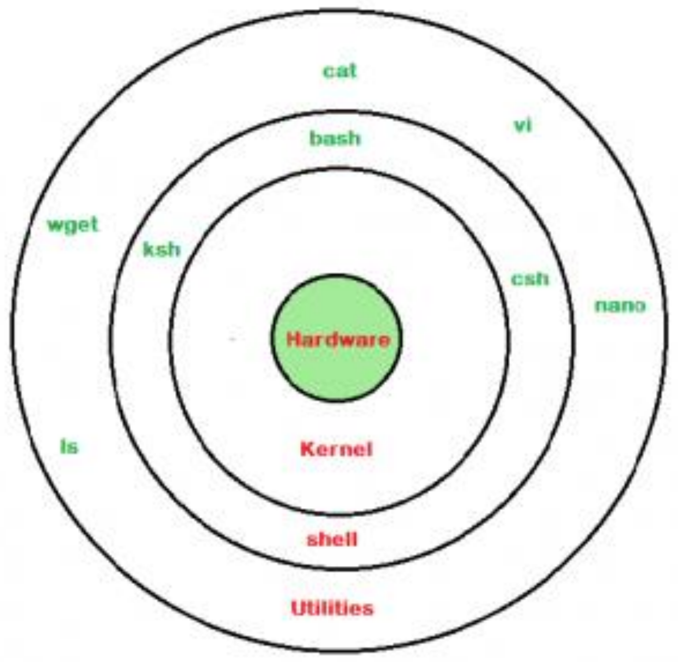
1. Kernel definition



[Understanding the Linux Kernel [Detailed Guide] - Linux Magazine (linuxnetmag.com)]

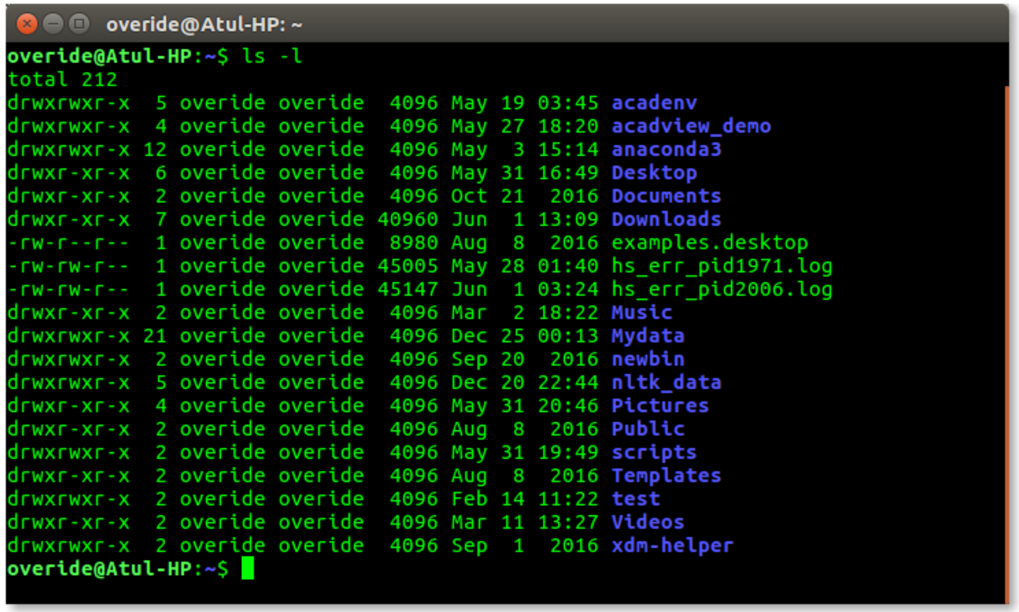
2. Shell definition

- A shell is a special user program that provides an interface for the user to use operating system services. Shell accepts human-readable commands from users and converts them into something which the kernel can understand. It is a command language interpreter that executes commands read from input devices such as keyboards or from files. The shell gets started when the user logs in or starts the terminal.
-

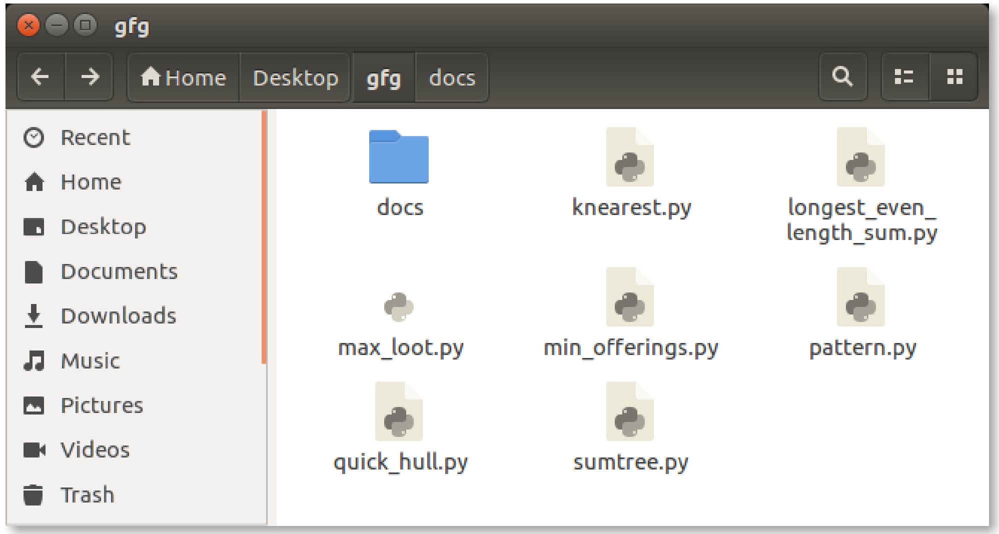


2. Shell definition

- Shell is broadly classified into two categories:
  - Command Line Shell
  - Graphical shell



```
overide@Atul-HP:~$ ls -l
total 212
drwxrwxr-x  5 overide overide  4096 May 19 03:45 acadenv
drwxrwxr-x  4 overide overide  4096 May 27 18:20 acadview_demo
drwxrwxr-x 12 overide overide  4096 May  3 15:14 anaconda3
drwxr-xr-x  6 overide overide  4096 May 31 16:49 Desktop
drwxr-xr-x  2 overide overide  4096 Oct 21  2016 Documents
drwxr-xr-x  7 overide overide 40960 Jun  1 13:09 Downloads
-rw-r--r--  1 overide overide  8980 Aug  8  2016 examples.desktop
-rw-rw-r--  1 overide overide 45005 May 28 01:40 hs_err_pid1971.log
-rw-rw-r--  1 overide overide 45147 Jun  1 03:24 hs_err_pid2006.log
drwxr-xr-x  2 overide overide  4096 Mar  2 18:22 Music
drwxrwxr-x 21 overide overide  4096 Dec 25 00:13 Mydata
drwxrwxr-x  2 overide overide  4096 Sep 20  2016 newbin
drwxrwxr-x  5 overide overide  4096 Dec 20 22:44 nltk_data
drwxr-xr-x  4 overide overide  4096 May 31 20:46 Pictures
drwxr-xr-x  2 overide overide  4096 Aug  8  2016 Public
drwxrwxr-x  2 overide overide  4096 May 31 19:49 scripts
drwxr-xr-x  2 overide overide  4096 Aug  8  2016 Templates
drwxrwxr-x  2 overide overide  4096 Feb 14 11:22 test
drwxr-xr-x  2 overide overide  4096 Mar 11 13:27 Videos
drwxrwxr-x  2 overide overide  4096 Sep  1  2016 xdm-helper
overide@Atul-HP:~$
```



3. Shell Initialisation (Hello World!)

- During session initialization, the shell executes configuration files, which may contain arbitrary commands and are typically used to define environment variables and aliases.
  - csh executes the ~/.cshrc file
  - tcsh executes the ~/.cshrc file
  - sh executes the ~/.profile file
  - bash executes the ~/.bash_profile file, or if not available, the ~/.profile file
- It is notable that these configuration files are hidden files.

Note: Each user can add shell commands to their personal profile ~/.bash_profile.

3. Shell Initialisation (Hello World!)

- For example, we can add the following lines:

```
gedit .bash_profile
clear
salutation=“Hello $USER!”

echo $USER
echo “ We are on $(date) ”
```

4. Scripts

- A bash or shell script is a simple executable text file (with execute permission) that starts with the characters #!/bin/bash (which must be the first characters of the file) and can contain commands or instructions. In fact, within the shell script, one can have variables, control structures, loop structures, etc., hence the term shell script.

Example:

```
#!/bin/bash
            ...
echo "This is a bash script"
            ...
```

Like any other program, a shell can also accept arguments in a file. Shell scripts (shell programs) can have arguments.

4. Scripts

- The symbol # is used to indicate a comment line in command files.

```
#author name
#111
```

- The echo command is used to display the expression given as a parameter. This expression can be:

  Either a variable

  Or a string

  Or an expression composed of strings and variables.

4. Scripts

- The read command reads user input from the standard input channel (keyboard) and stores this data into shell variables.
- It reads a line of input, splits it into words separated by spaces, and assigns each word to the variables in the variable-list. The names of these variables are passed as parameters to read, with the following syntax: read var1 [var2 ...].
- If the line read contains more words than there are variables in the variable-list, the remaining words are assigned to the last variable.
- When read is executed, the shell waits for input from the user.

```
echo ”input variables a et b ”
read a b
echo ”a = $a”
echo ”b = $b”
```

5. Variables

Variable Names:

- Variable names can be composed of:
  - A sequence of letters, digits, and the underscore character _: this case corresponds to variables created by the user.
  - A digit: this case corresponds to parameters of command files.
  - One of the following characters * @ # ? -$!: this case corresponds to a set of variables managed by the Shell.

Environment Variables:

- Environment Variables (Predefined) The UNIX system defines for each process a list of environment variables, which allow certain parameters to be defined. Among these variables, we mention:
  - **HOME**: contains the absolute path of the user's login directory
  - **LOGNAME**: contains the user's login name
  - **PATH**: contains the list of directories containing executables separated by ':'. These directories will be searched in order for an external command.
  - **SHELL**: contains the absolute path to the shell program files

Note: The env command displays all environment variables for the active shell.

Example:

```
$ echo "The login directory is: $HOME
```

5. Variables

<u>User Variables:</u>

- To reference the value of a variable, we use the notation consisting of writing the $ sign followed by the variable name.

Example:

```
i=2
echo $i
```

<u>Assigning values to variables</u>

- To assign a value to a variable, simply type the variable name, followed by the equal sign, followed by the value you want to assign to the variable. The syntax of an assignment is as follows:

```
variable-name = string-of-characters
```

Example:

```
Var1=hello
Var2=madam
DIR=/usr/myFolder
PERSON=$USER
```

5. Variables

Special Variables

Special Variable / character	Content
$?	Return code of the last executed command
$$	**P**rocess **ID**entifier of active shell
$!	PID of the last background process launched
$-	The shell options

5. Variables

Positional Parameters

Positional parameters are also special variables used when passing parameters to a script.

Special Variable / character	Content
$0	Command name (of the script)
$1-9 $1, $2, $3…	The first nine parameters passed to the script
$#	Total number of parameters passed to the script
$*	List of all parameters in the format "$1 $2 $3 ..."
$@	List of parameters as separate elements "$1" "$2" "$3" ..."

5. Variables

Numeric Value Tests

Option	Role
-eq	Equal: Equal
-ne	Not Equal: Different
-lt	Less than: Less than
-gt	Greater than: Greater than
-le	Less than or equal
-ge	Greater than or equal

5. Variables

Combined Tests by AND OR NOT

Criteria Multiple tests can be performed with a single instruction. The combination options are the same as for the find command.

Criteria	Action
-a	AND, logic
-o	OR, logic
!	NOT, logic

6. Exercises: Basics Shell script & file management

1. Create a directory named "backup_scripts" in your home directory;
2. Navigate to the "backup_scripts" directory;
3. Using a text editor like Nano or Vim, create a new Bash script named "backup.sh“;
4. Write a script that copies the contents of a specified directory to a backup directory;
5. Prompt the user to enter the directory they want to back up and the directory where they want to store the backup;
6. Implement error handling to ensure the specified directories exist;
7. Execute the backup operation and display appropriate messages to the user indicating success or failure;
8. Save and exit the text editor;
9. Make the script executable using the command: **chmod +x backup.sh**;
10. Test the script by running it with different directories to ensure it works as expected.

Plan

Chapter 3: Linux – Shell programming Part II

1. If structure

Syntax

```
if [ condition ]; then
    # Commands to execute if the condition is true
else
    # Commands to execute if the condition is false
fi
```

Example

```
if [ "$1" -gt 10 ]; then
    echo "The number is greater than 10."
else
    echo "The number is not greater than 10."
fi
```

2. Case structure

Syntax

```
case variable in
   pattern1)
      # Commands to execute if variable matches pattern1
      ;;
   pattern2)
      # Commands to execute if variable matches pattern2
      ;;
   *)
      # Default case: commands to execute if variable matches
none of the above patterns
      ;;
esac
```

Example

```
fruit="apple"
case $fruit in
   "apple")
      echo "It's an apple."
      ;;
   "banana" | "orange")
      echo "It's a banana or an orange."
      ;;
   *)
      echo "It's something else."
      ;;
esac
```

3. Loops

"for" syntax

```
for variable in item1 item2 ... itemN
do
    # Commands to execute for each iteration
    # You can use "$variable" to reference the current item
done
```

example

```
for fruit in apple banana orange
do
    echo "I like $fruit"
done
```

3. Loops

“while” syntax

```
while [ condition ]
do
    # Commands to execute as long as the condition is true
done
```

example

```
count=1
while [ $count -le 5 ]
do
    echo "Count is $count"
    ((count++))
done
```

3. Loops

“until” syntax

```
until [ condition ]
do
    # Commands to execute as long as the condition is false
done
```

example

```
count=1
until [ $count -gt 5 ]
do
    echo "Count is $count"
    ((count++))
done
```

4. Arrays

- There are two types of arrays: indexed arrays and associative arrays;
- An array can be created explicitly using the keyword **declare** (and can also be initialized at the same time), followed by the option **-a** for an indexed array, and **-A** for an associative array.

Example

```
declare -a indexedArray=("one" "two" "three" "four")
declare -A associativeArray=(['one']="un" ['two']="deux"
['three']="trois")
```

4. Arrays

- The display of an array is done with the syntax **${myArray[*]}** or **${myArray[@]}**. The difference between using **@** and ***** is identical to that between the special shell variables **$@** and **$***

Example

```
declare -a indexedArray=("one" "two" "three" "four")
echo ${indexedArray[@]}  # Output: one two three four

declare -A associativeArray=(['one']="un" ['two']="deux"
['three']="trois")
echo ${associativeArray[@]}  # Output: un deux trois
```

4. Arrays

- The use of braces is necessary to avoid conflicts with path expansions (where square brackets have a special meaning). It is possible to obtain the list of keys of an array using the syntax **${!array[@]}**. In the case of an indexed array, you get its indices.

Example

```
declare -a indexedArray=("one" "two" "three" "four")
echo ${!indexedArray[@]}  # Output: 0 1 2 3

declare -A associativeArray=(['one']="un" ['two']="deux"
['three']="trois")
echo ${!associativeArray[@]}  # Output: one two three
```

4. Arrays

- This allows you to check the indices assigned to the values during the initialization of the array;

Example

```
declare -a indexedArray=("fry" "leela" [42]="bender" "flexo")
echo ${!indexedArray[@]}  # Output: 0 1 42 43
declare -a indexedArray=("one" "two" "three" "four")
echo ${#indexedArray[@]}  # Output: 4

declare -A associativeArray=(['one']="un" ['two']="deux"
['three']="trois")
echo ${#associativeArray[@]}  # Output: 3
```

5. Exercises

1. Write a Bash script that iterates through numbers from 1 to 20. For each number, if it's divisible by 3, print "Fizz". If it's divisible by 5, print "Buzz". If it's divisible by both 3 and 5, print "FizzBuzz". Otherwise, print the number itself.
2. Write a Bash script that prompts the user to guess a secret number between 1 and 100. The script should generate a random number as the secret number. The user should continue guessing until they correctly guess the secret number. After each guess, the script should provide feedback on whether the guess is too high, too low, or correct.