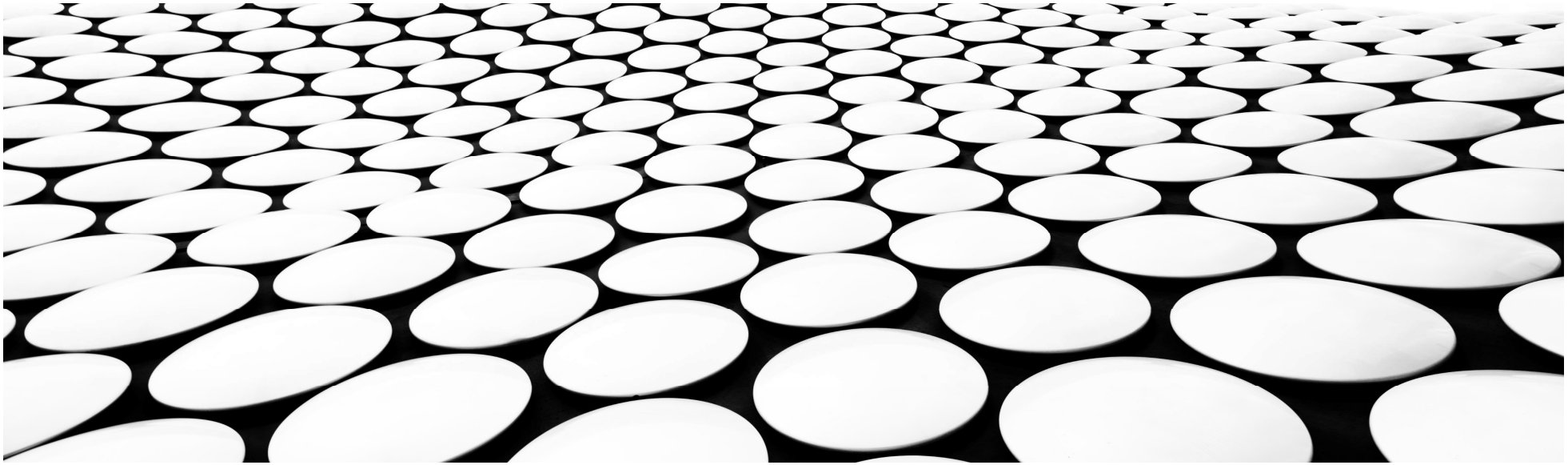
BASH SCRIPTING

BY JAI



LINK FOR SCRIPTS:

- https://github.com/JaiTrieTree/bashscripts-devops

IMPORTANCE OF BASH SCRIPTING FOR DEVOPS

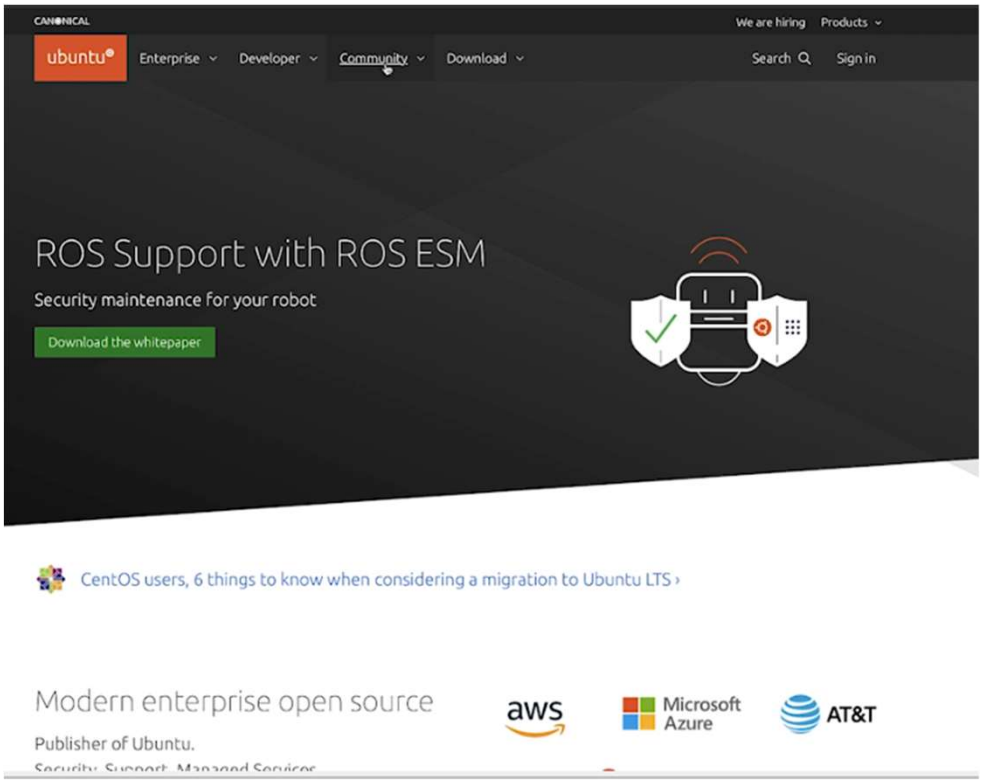
- Bash scripting is an essential skill for DevOps engineers as it enables them to automate tasks and manage infrastructure effectively. Bash scripting is a programming language used to write scripts that can automate tasks in the Unix/Linux operating systems.
- Importance of bash scripting in DevOps and its history:

1. Automation: Bash scripting enables DevOps engineers to automate tasks such as software deployment, configuration management, and system administration. This automation improves efficiency, reduces human error, and ensures consistency in the management of infrastructure.
2. Shell scripting: Bash is a shell scripting language that can interact with the operating system and execute commands. This makes it a powerful tool for DevOps engineers who need to manage and configure servers and applications.
3. History: Bash scripting has a long history dating back to 1987 when Brian Fox created the first version of Bash as part of the GNU project. Bash is now the default shell for many Unix-based operating systems, including Linux.
4. Open-source: Bash is an open-source tool, which means it is free to use and can be customized according to individual needs. This makes it an attractive tool for DevOps engineers who work in organizations of all sizes.
5. Portable: Bash scripts are portable, meaning they can be run on different operating systems without modification. This makes it easy for DevOps engineers to move scripts between different environments and manage infrastructure consistently.

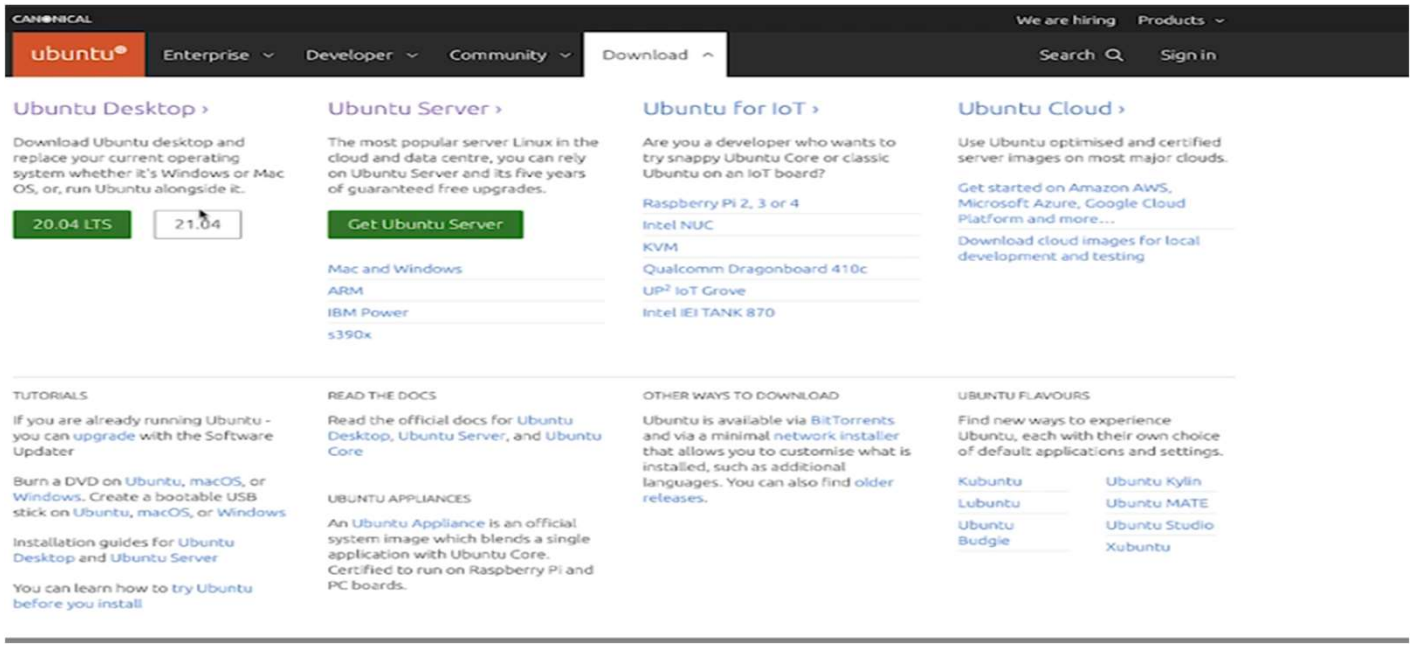
SETTING UP THE COURSE ENVIRONMENT

- Installing Ubuntu Linux
- Installing Red Hat Linux
- Using windows subsystem for Linux

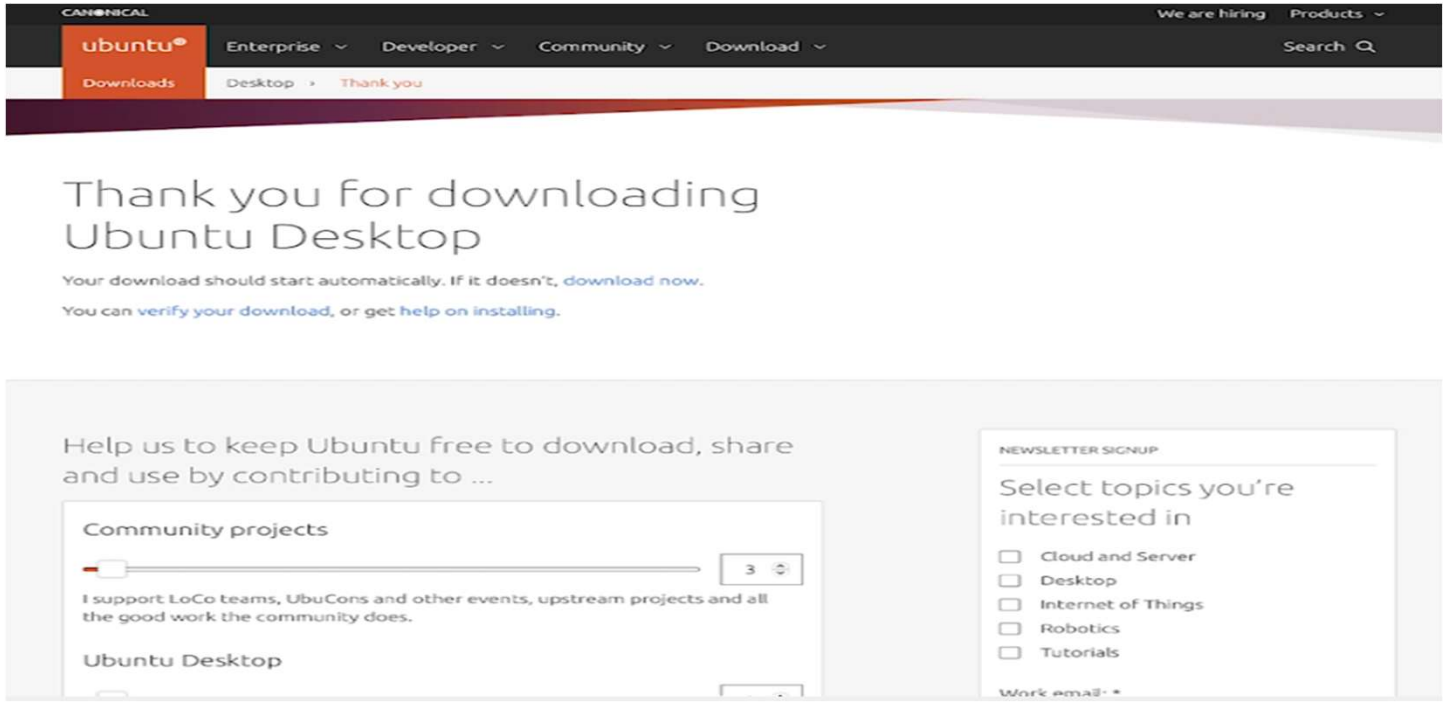
INSTALLING UBUNTU LINUX



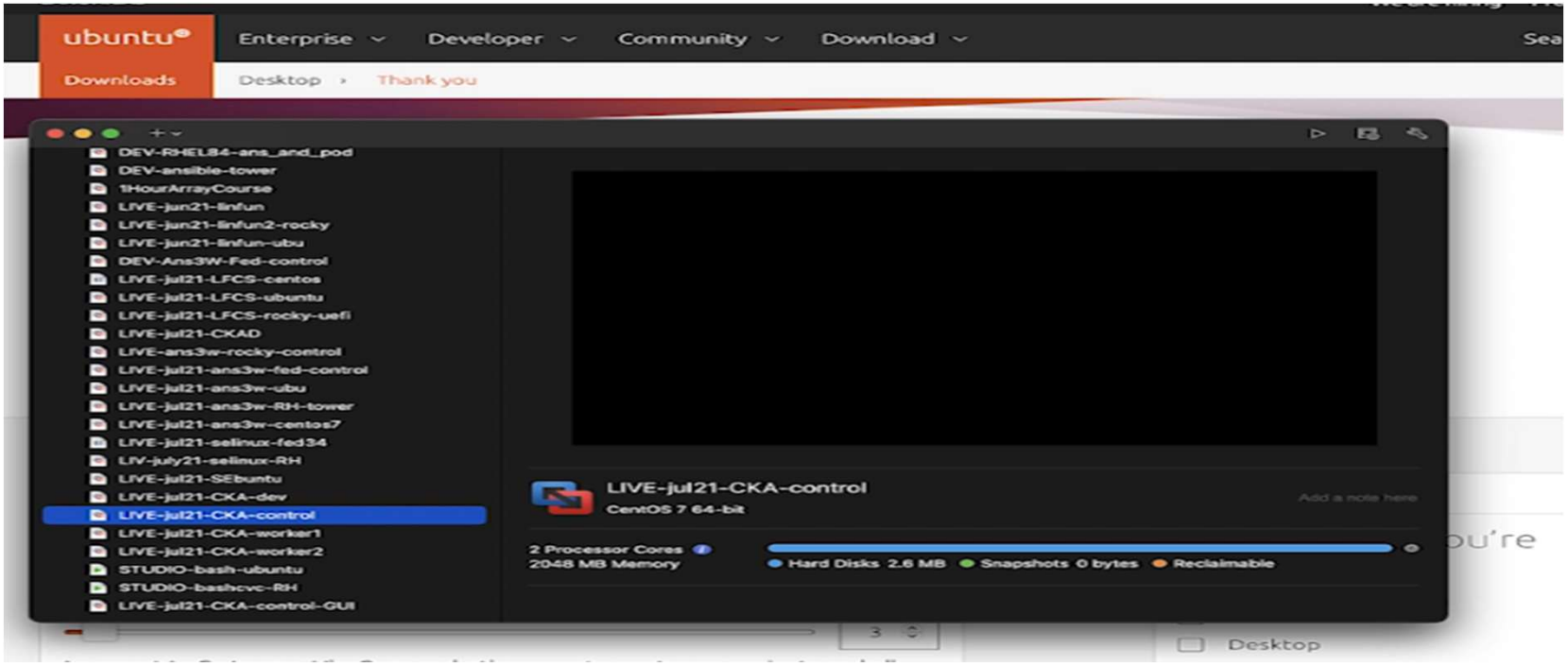
INSTALLING UBUNTU LINUX



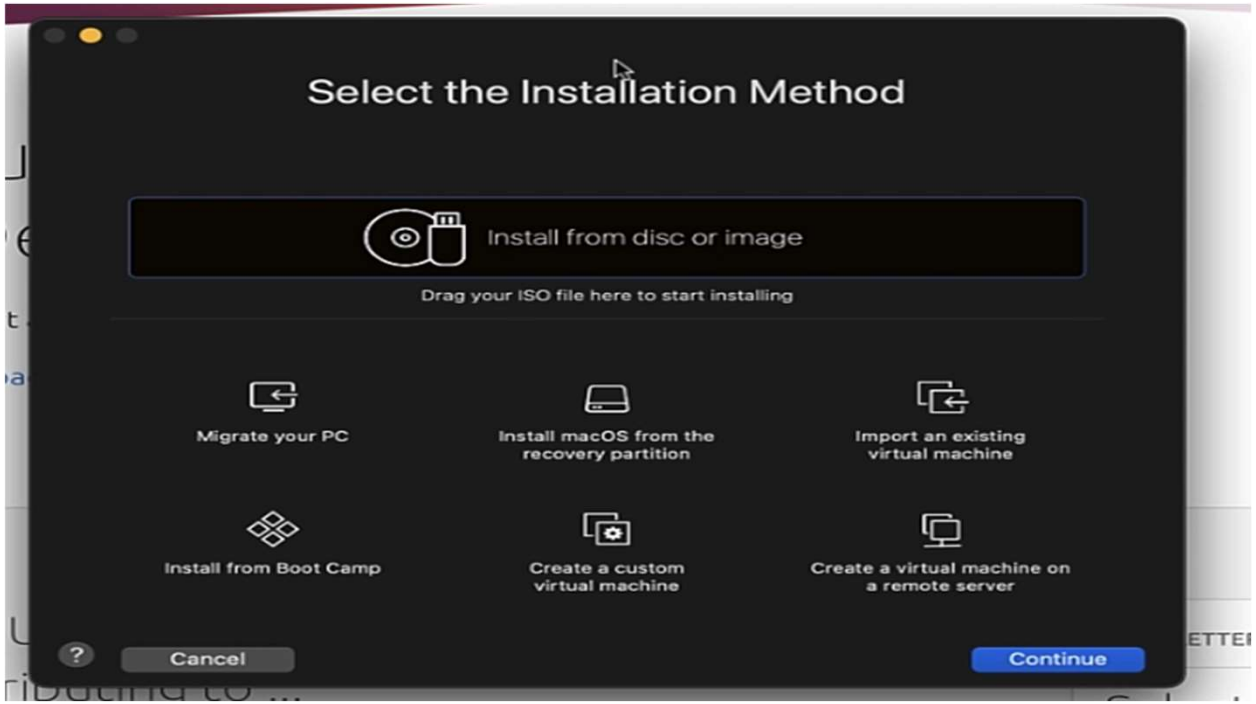
INSTALLING UBUNTU LINUX



INSTALLING UBUNTU LINUX

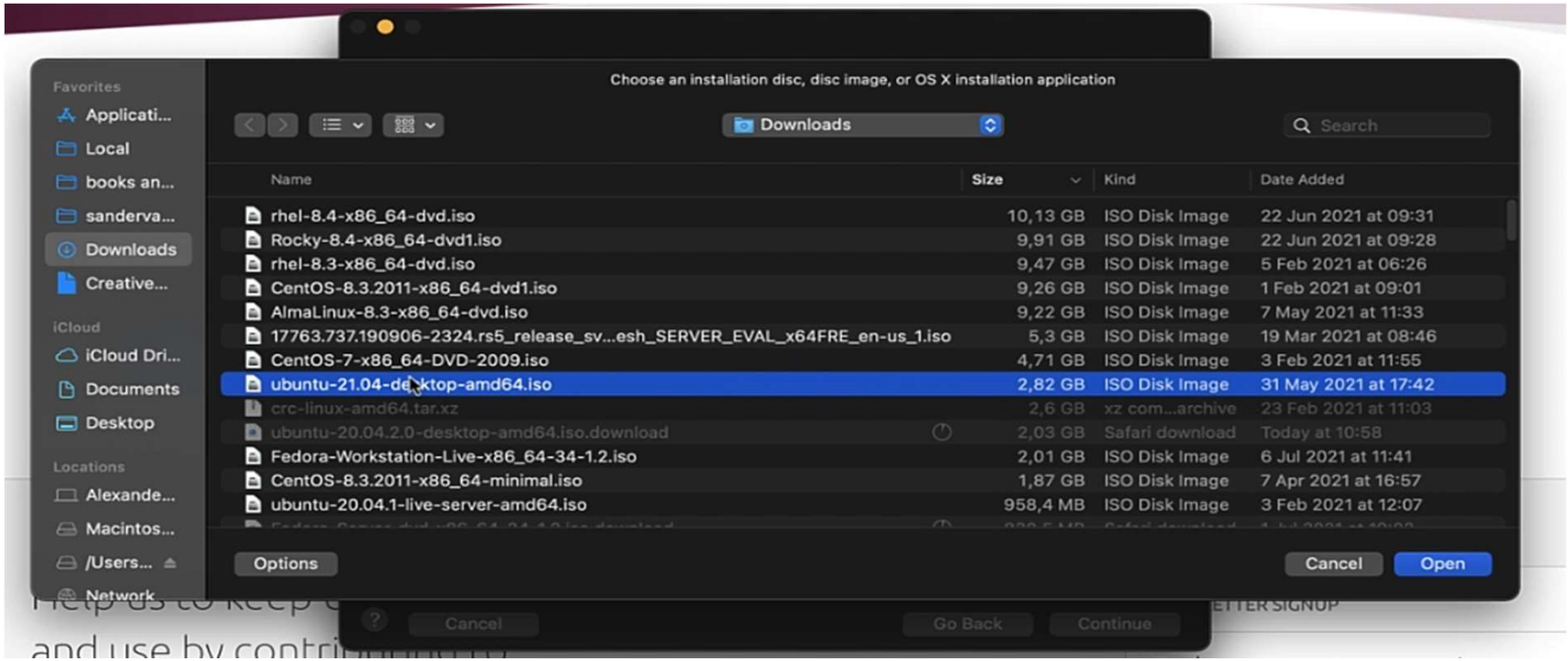


INSTALLING UBUNTU LINUX



INSTALLING UBUNTU LINUX

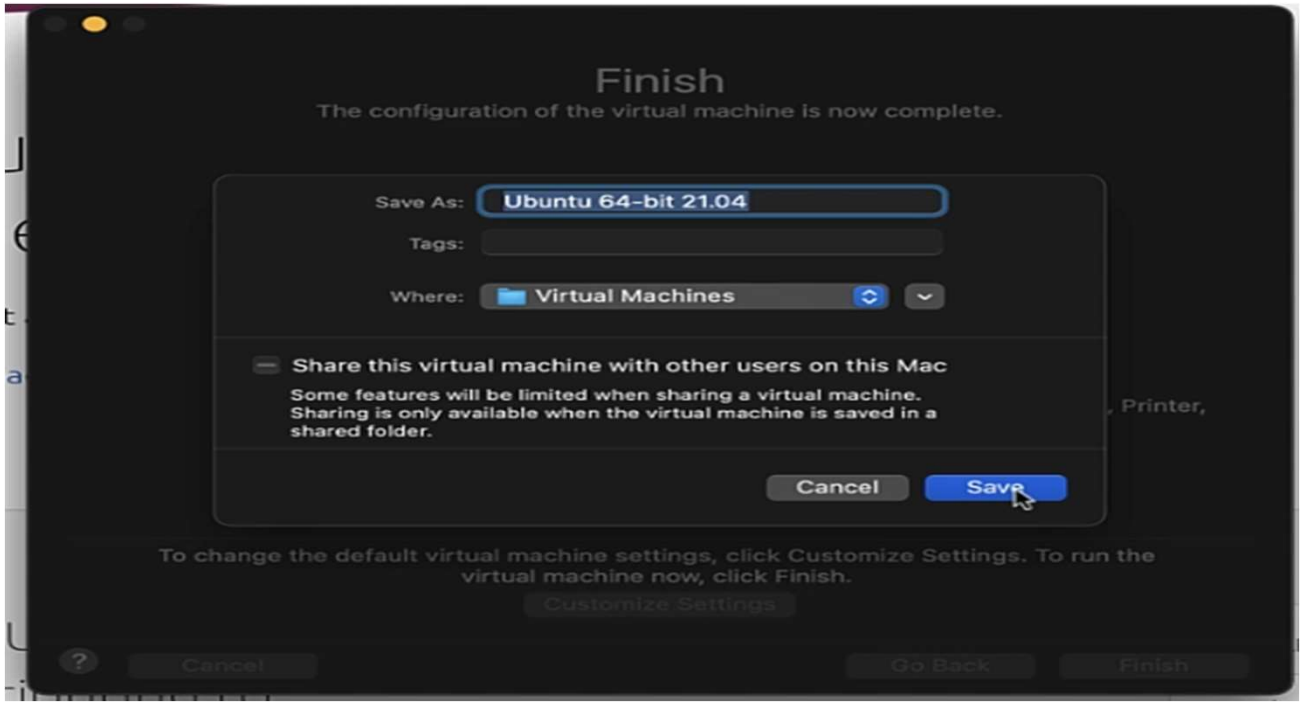
INSTALLING UBUNTU LINUX



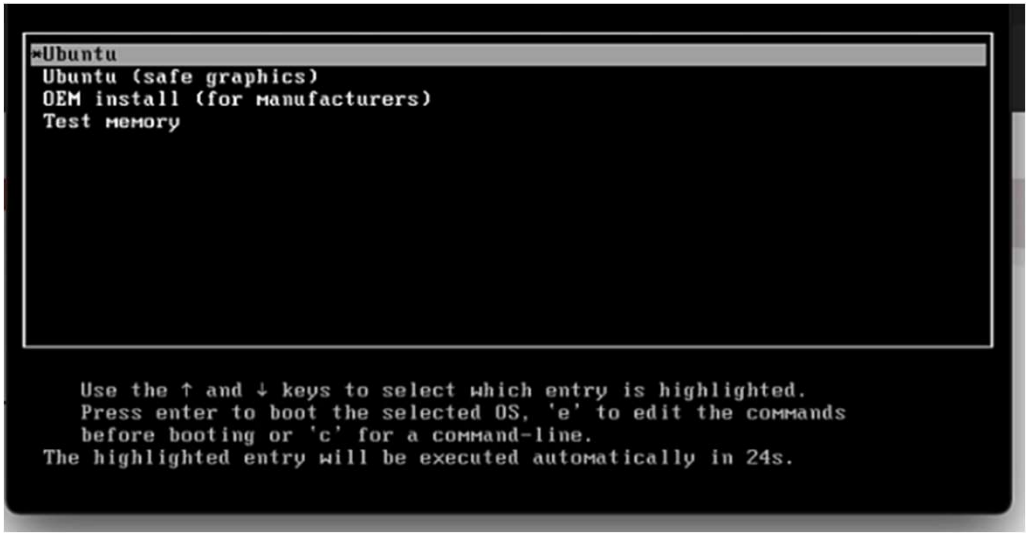
INSTALLING UBUNTU LINUX

INSTALLING UBUNTU LINUX

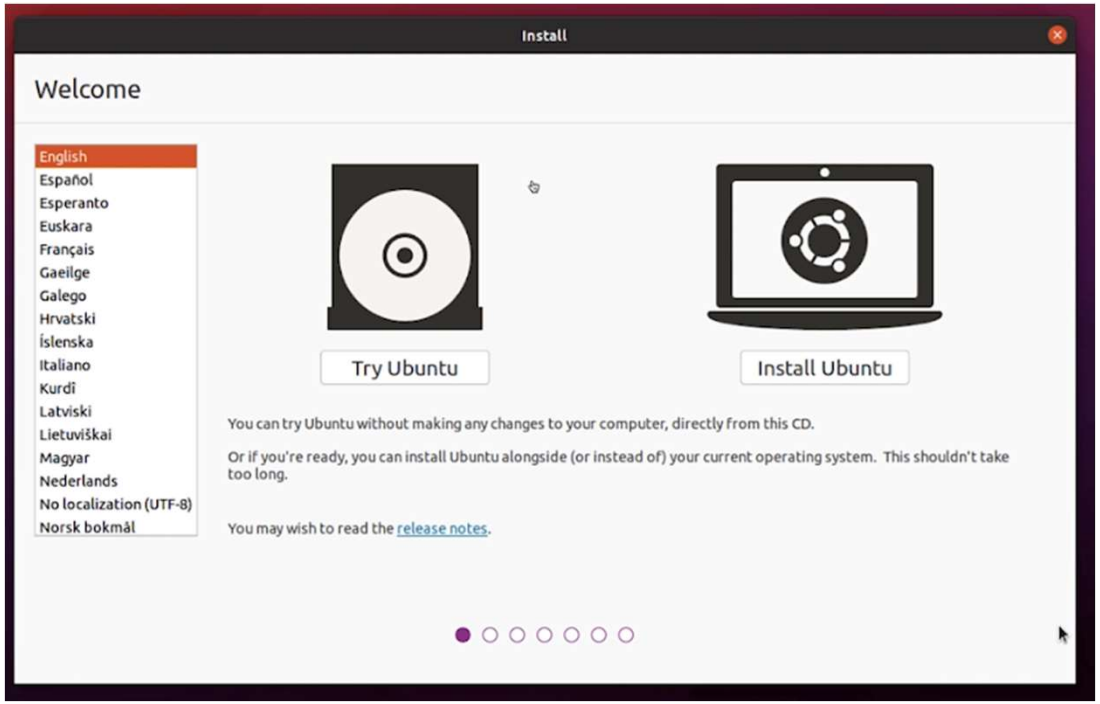
INSTALLING UBUNTU LINUX



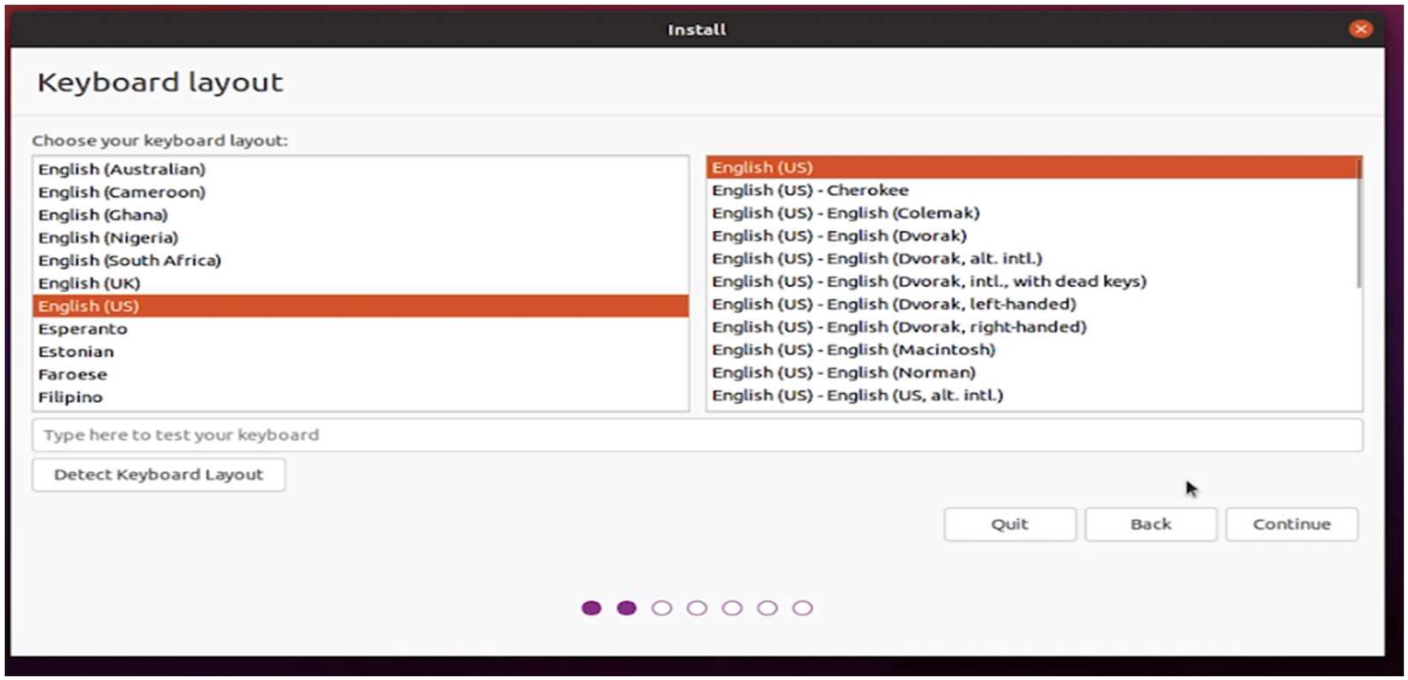
INSTALLING UBUNTU LINUX



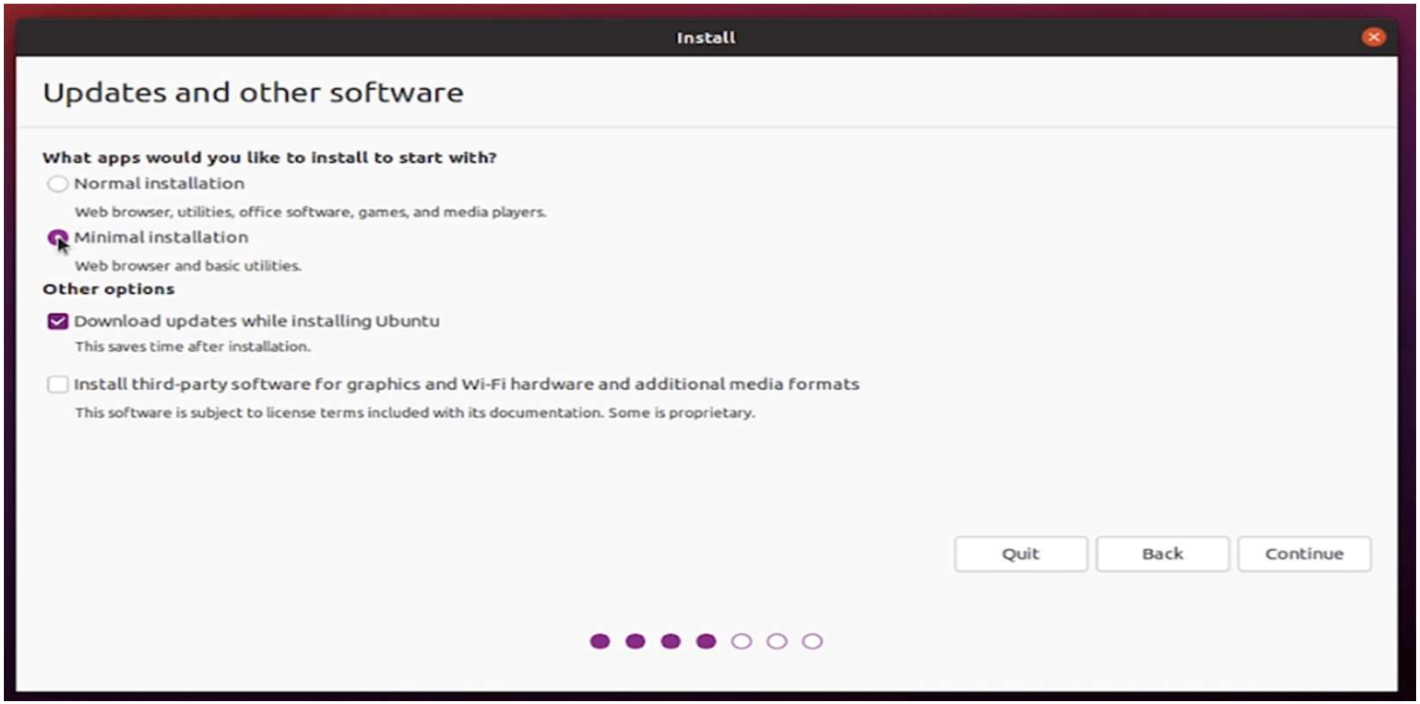
INSTALLING UBUNTU LINUX



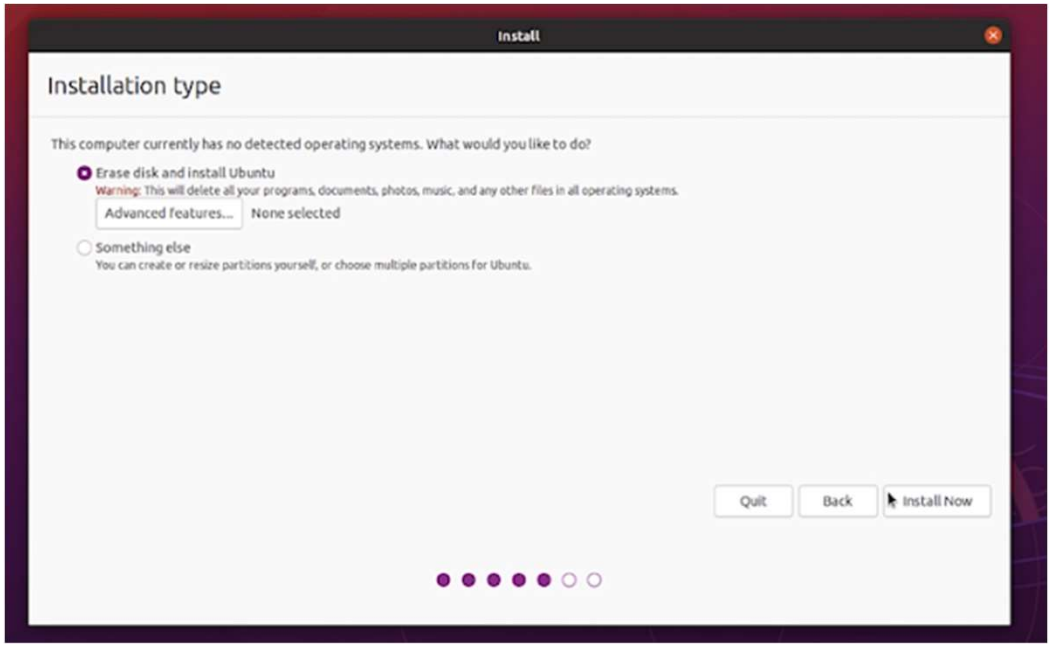
INSTALLING UBUNTU LINUX



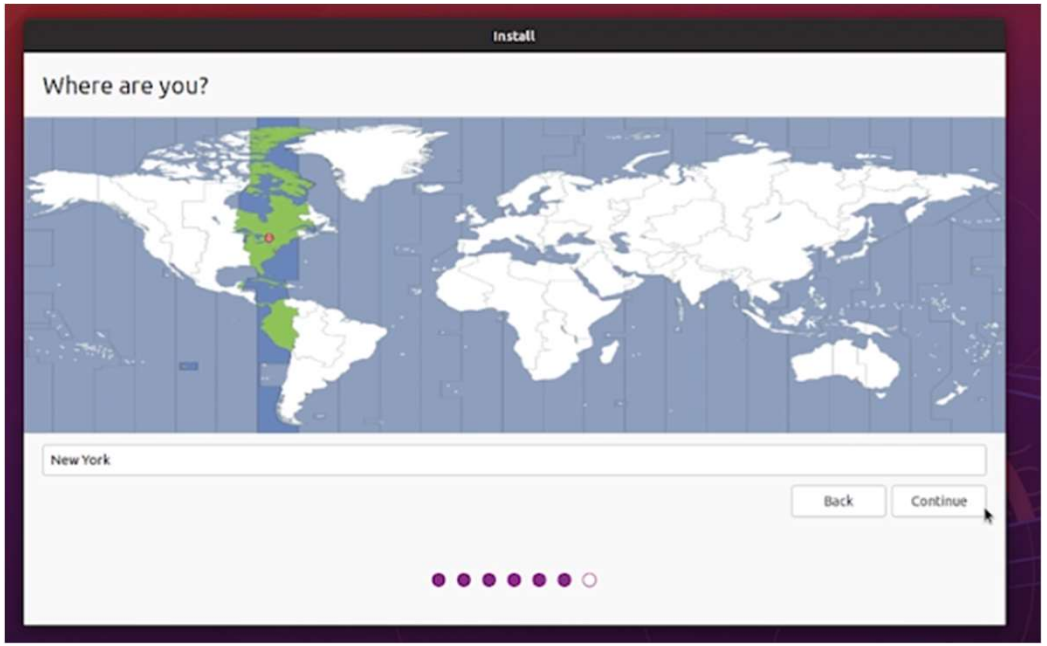
INSTALLING UBUNTU LINUX



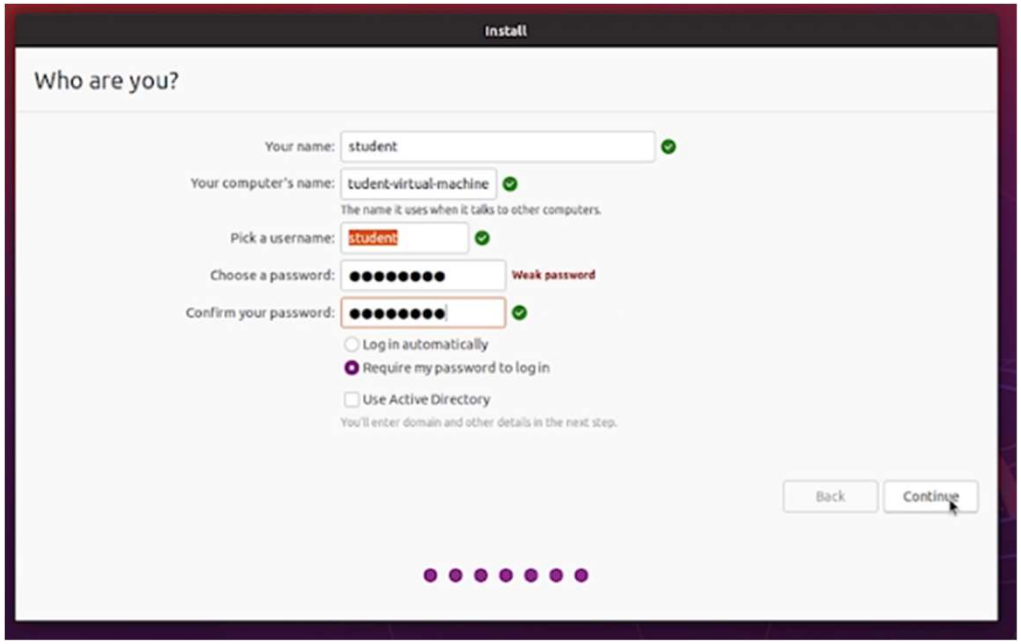
INSTALLING UBUNTU LINUX



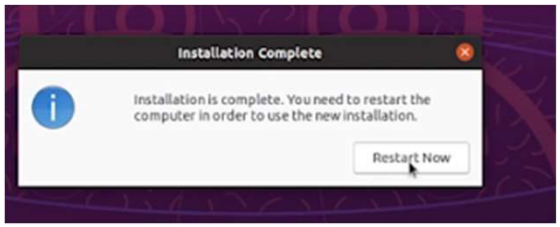
INSTALLING UBUNTU LINUX

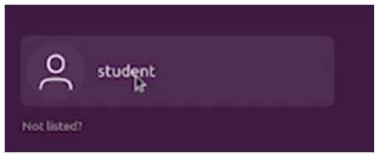


INSTALLING UBUNTU LINUX

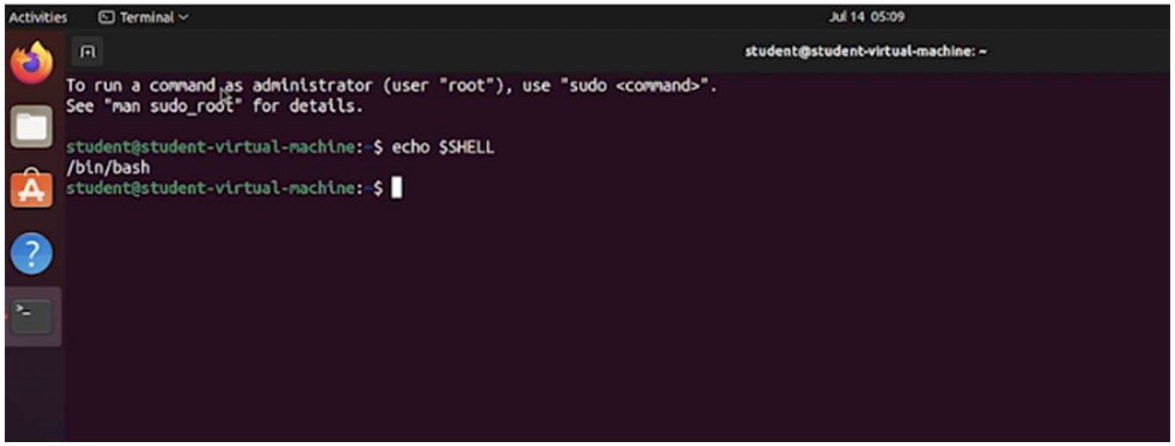


INSTALLING UBUNTU LINUX

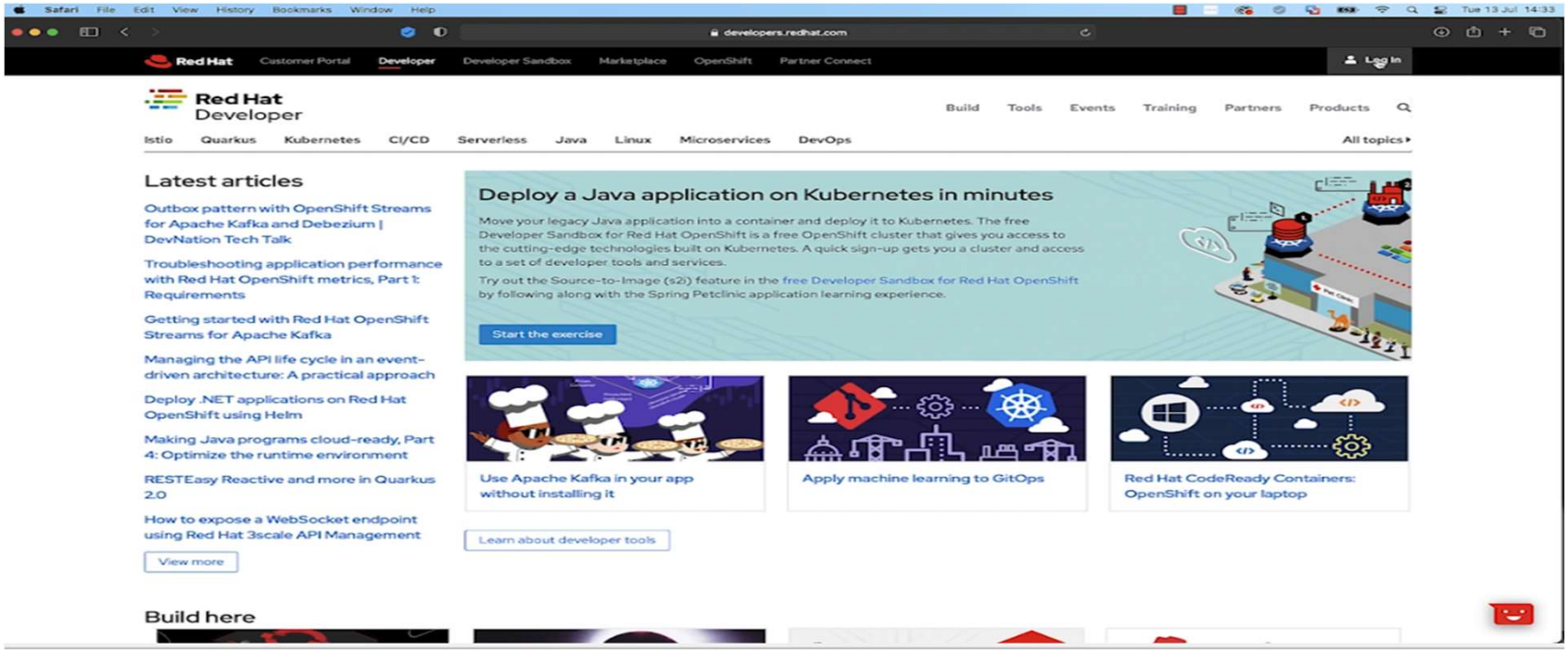




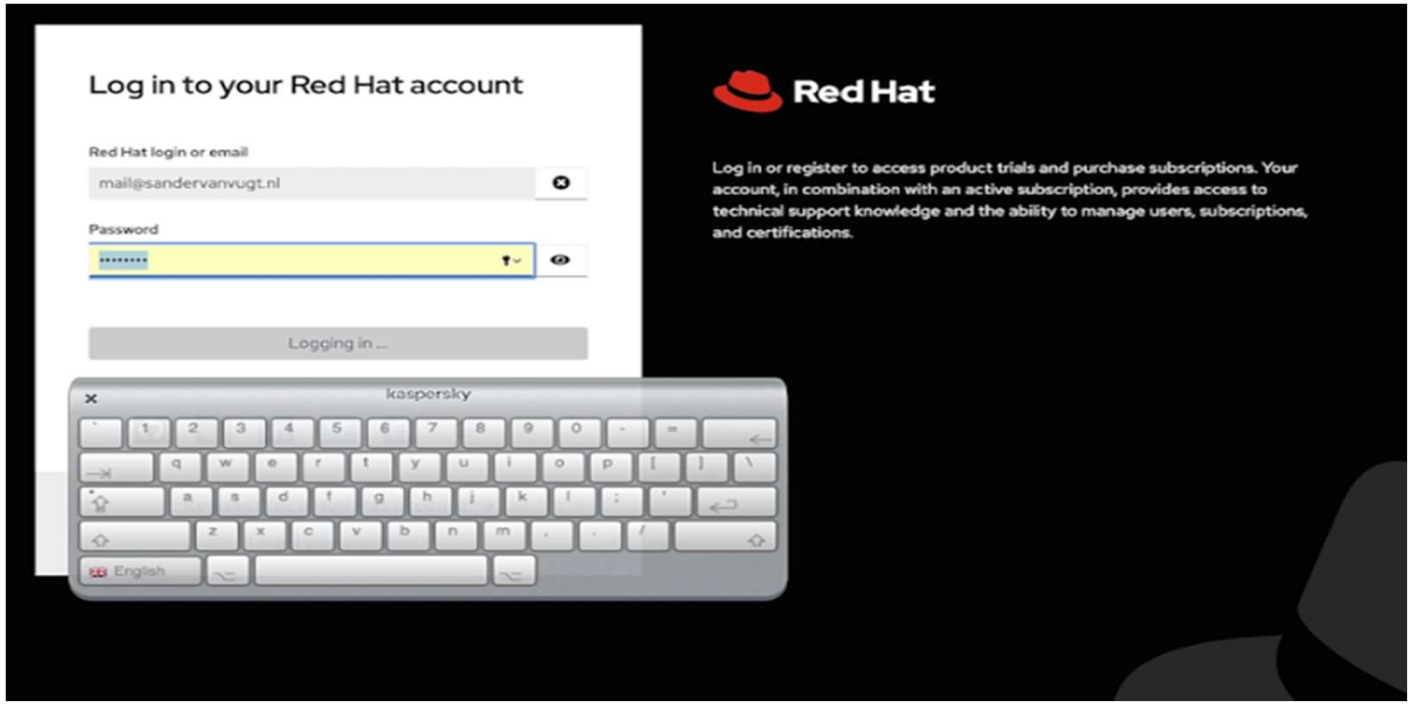
INSTALLING UBUNTU LINUX



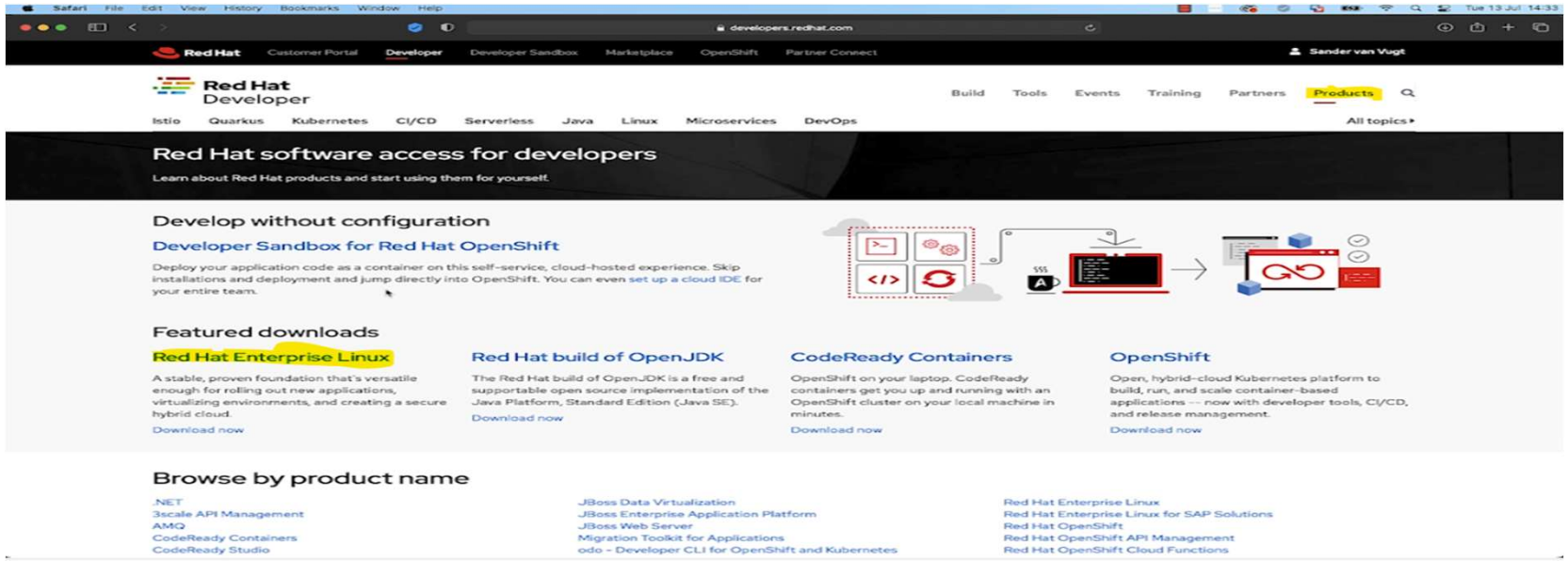
INSTALLING RED HAT LINUX



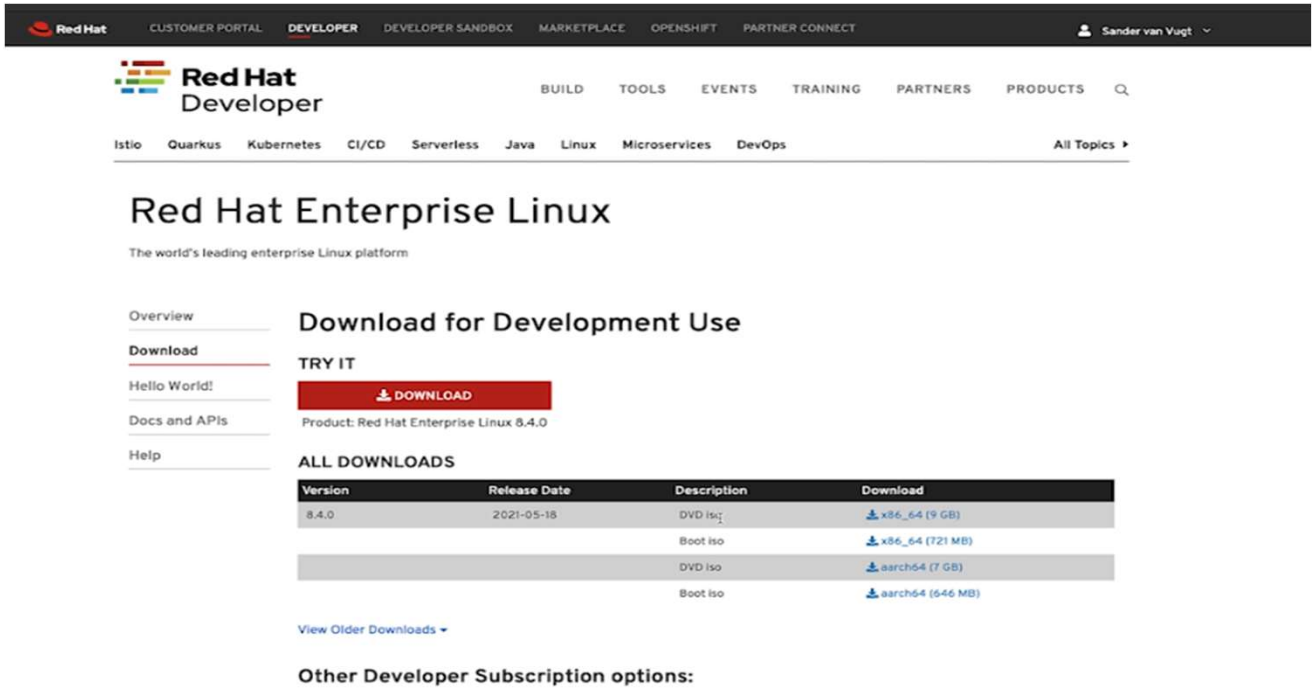
INSTALLING RED HAT LINUX



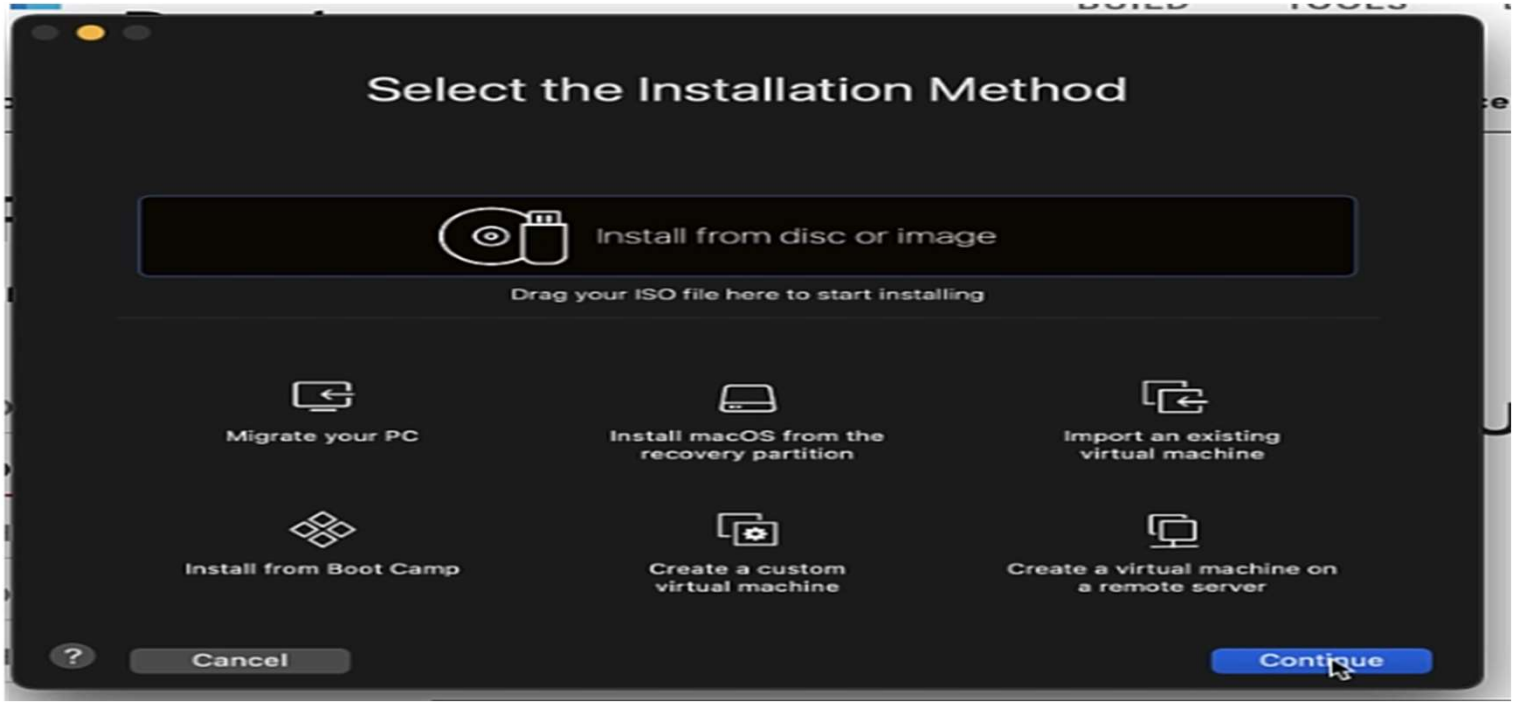
INSTALLING RED HAT LINUX



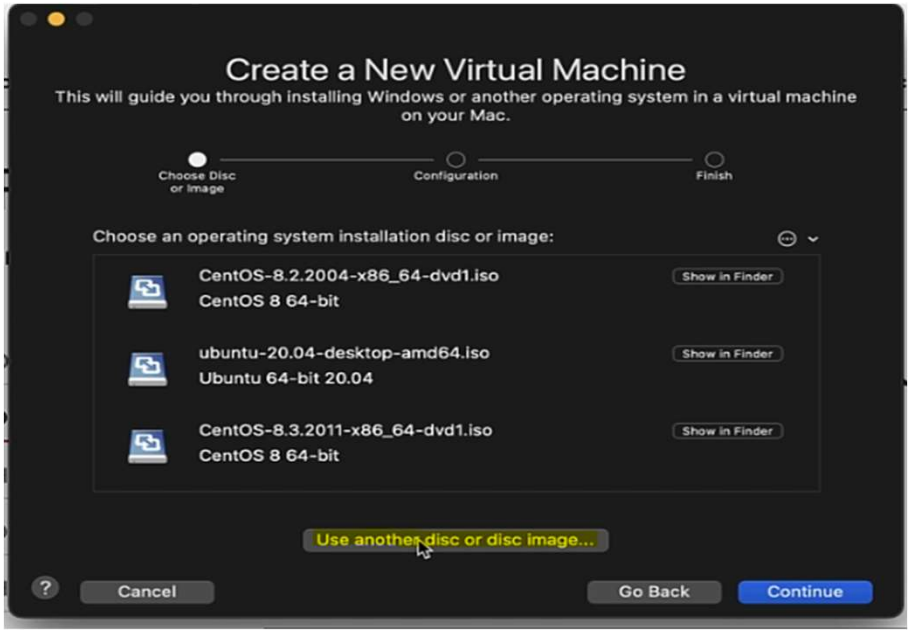
INSTALLING RED HAT LINUX



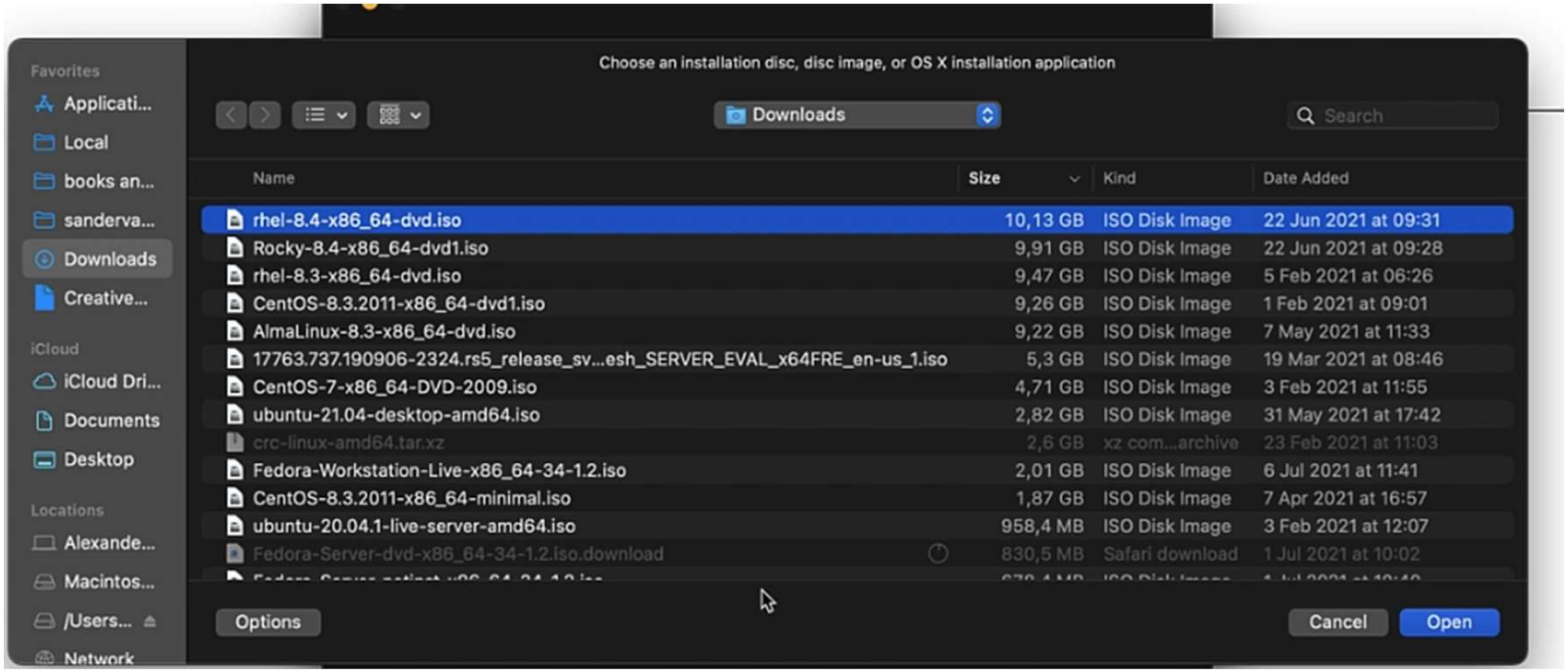
INSTALLING RED HAT LINUX



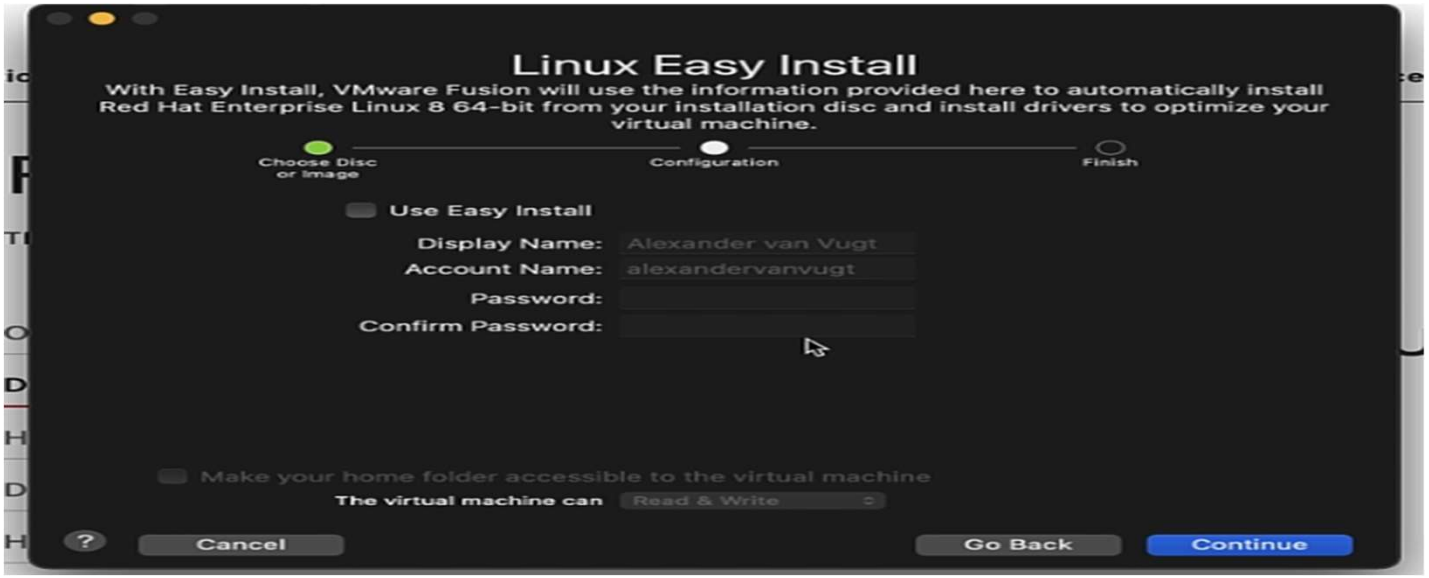
INSTALLING RED HAT LINUX



INSTALLING RED HAT LINUX

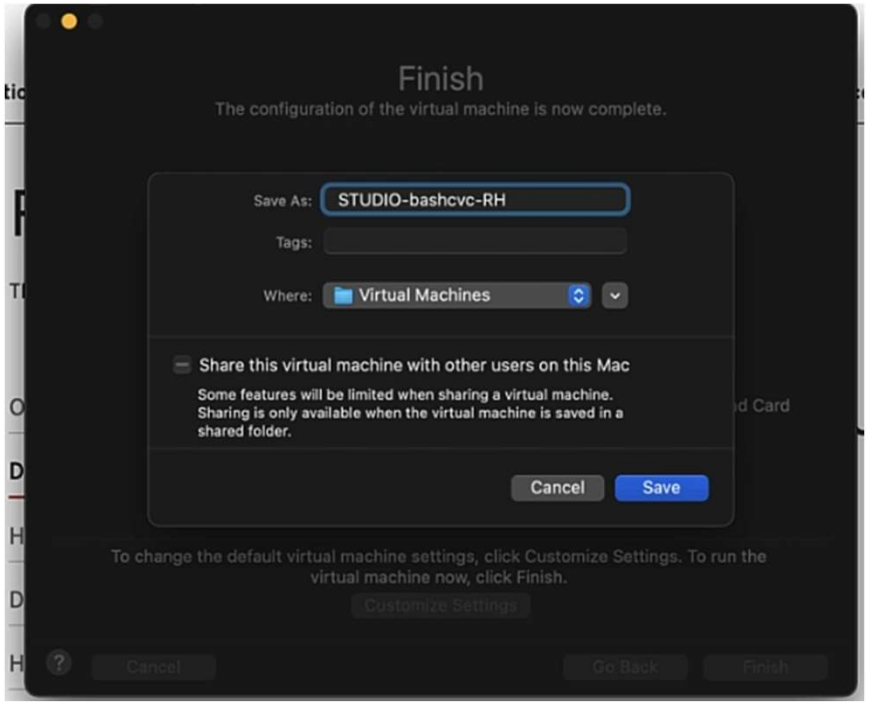


INSTALLING RED HAT LINUX

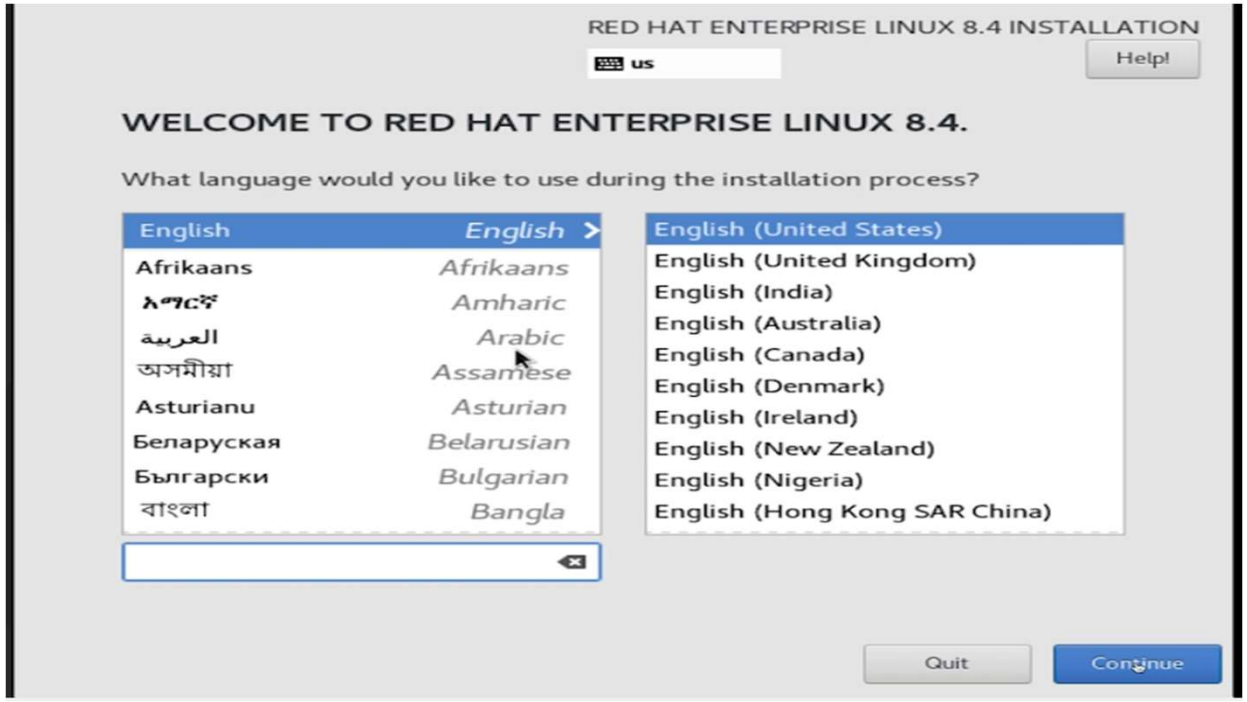


INSTALLING RED HAT LINUX

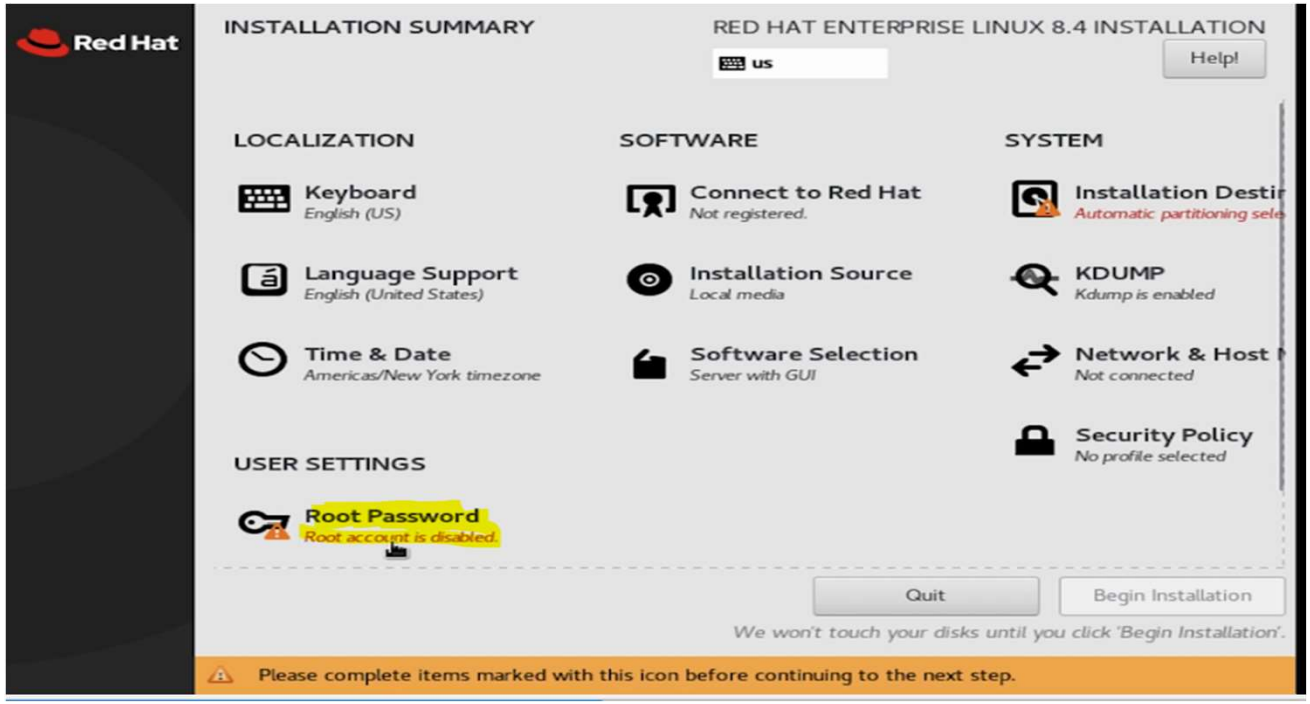
INSTALLING RED HAT LINUX



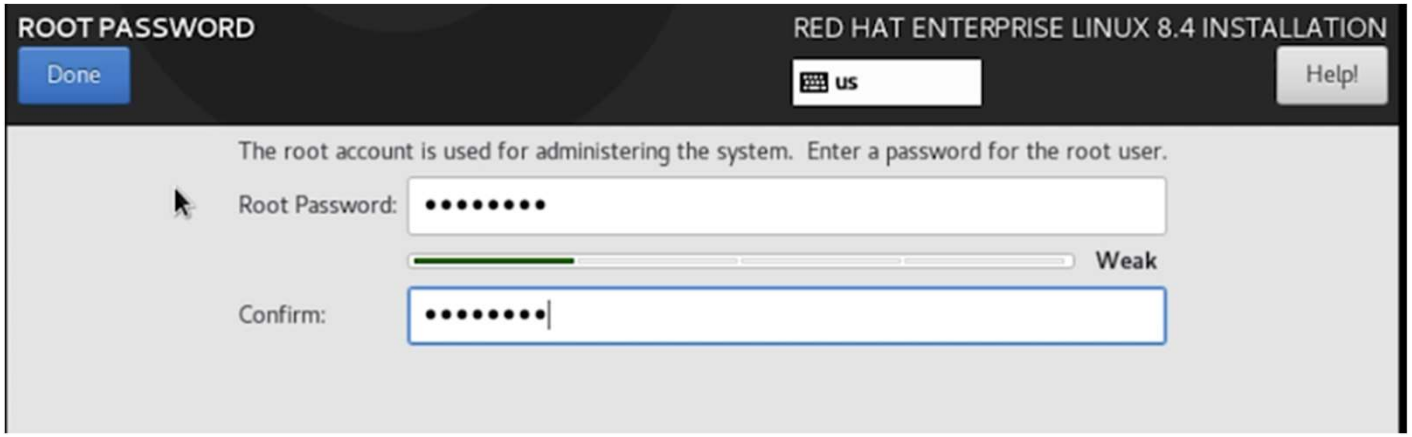
INSTALLING RED HAT LINUX



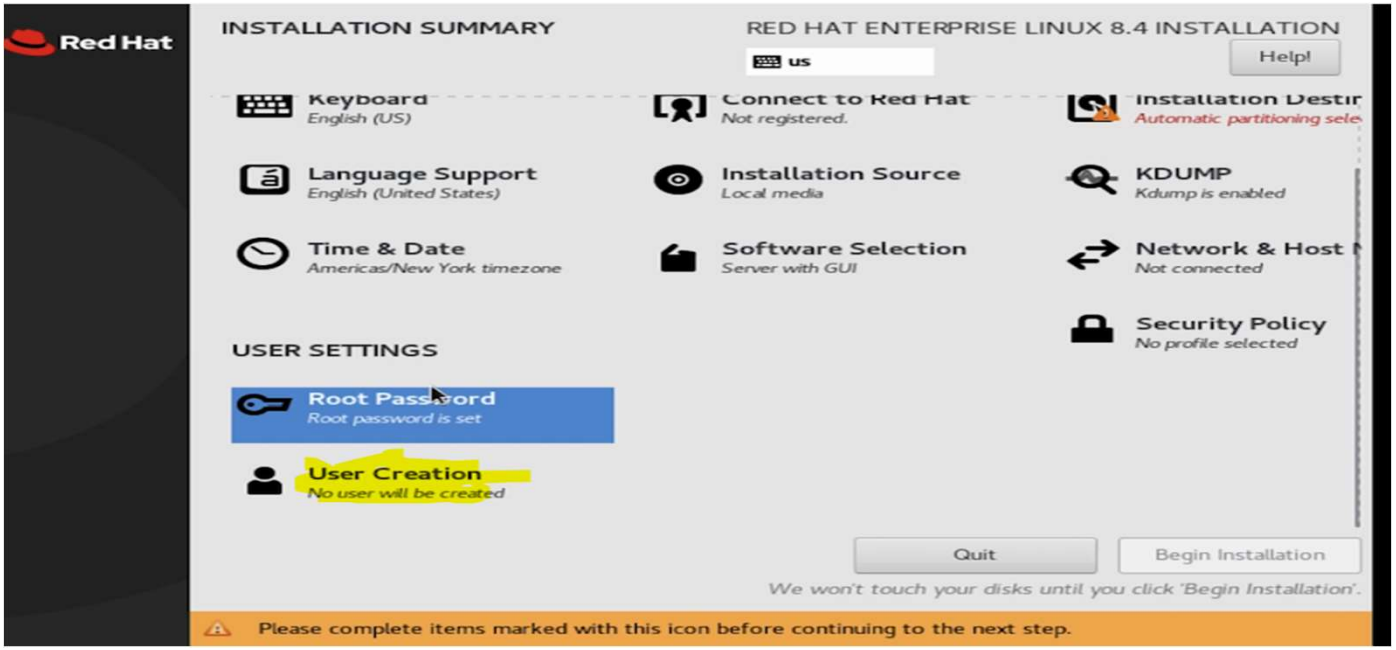
INSTALLING RED HAT LINUX



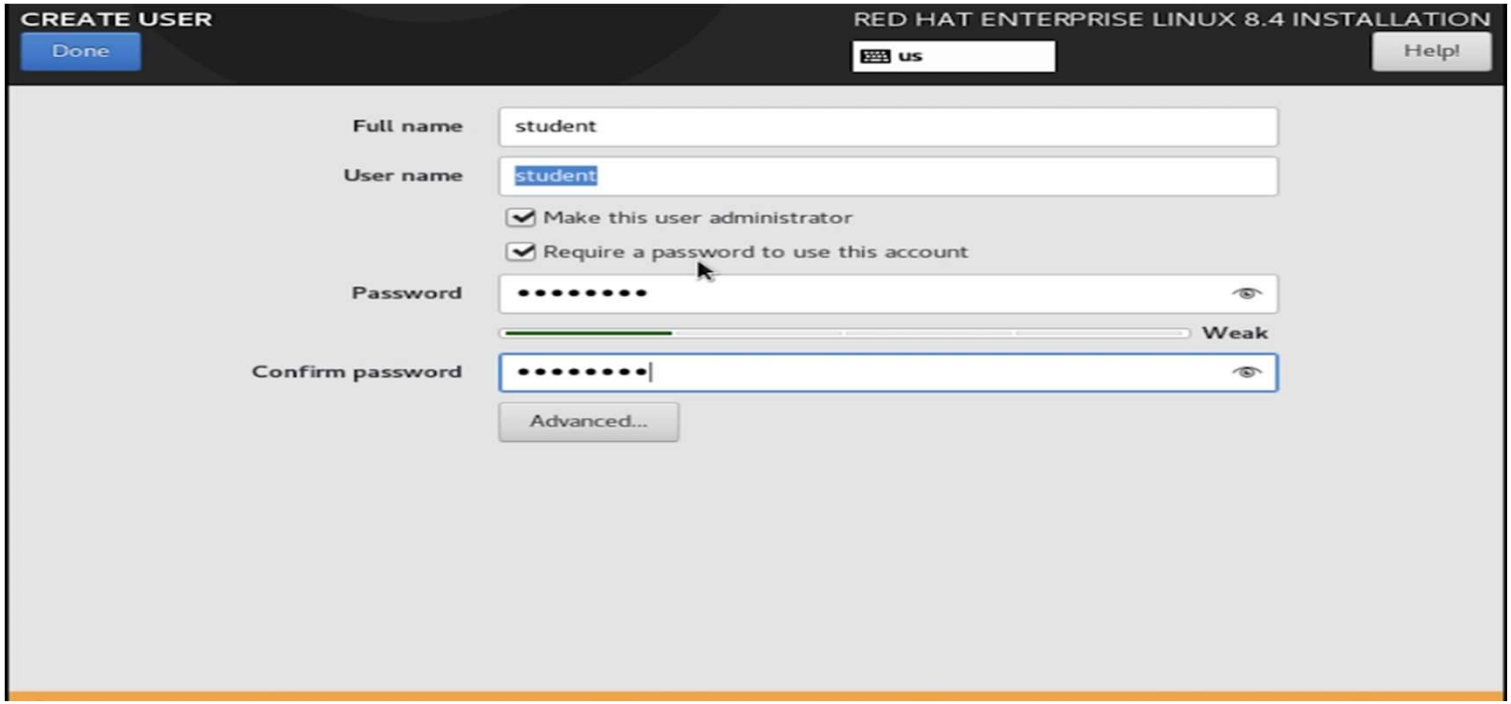
INSTALLING RED HAT LINUX



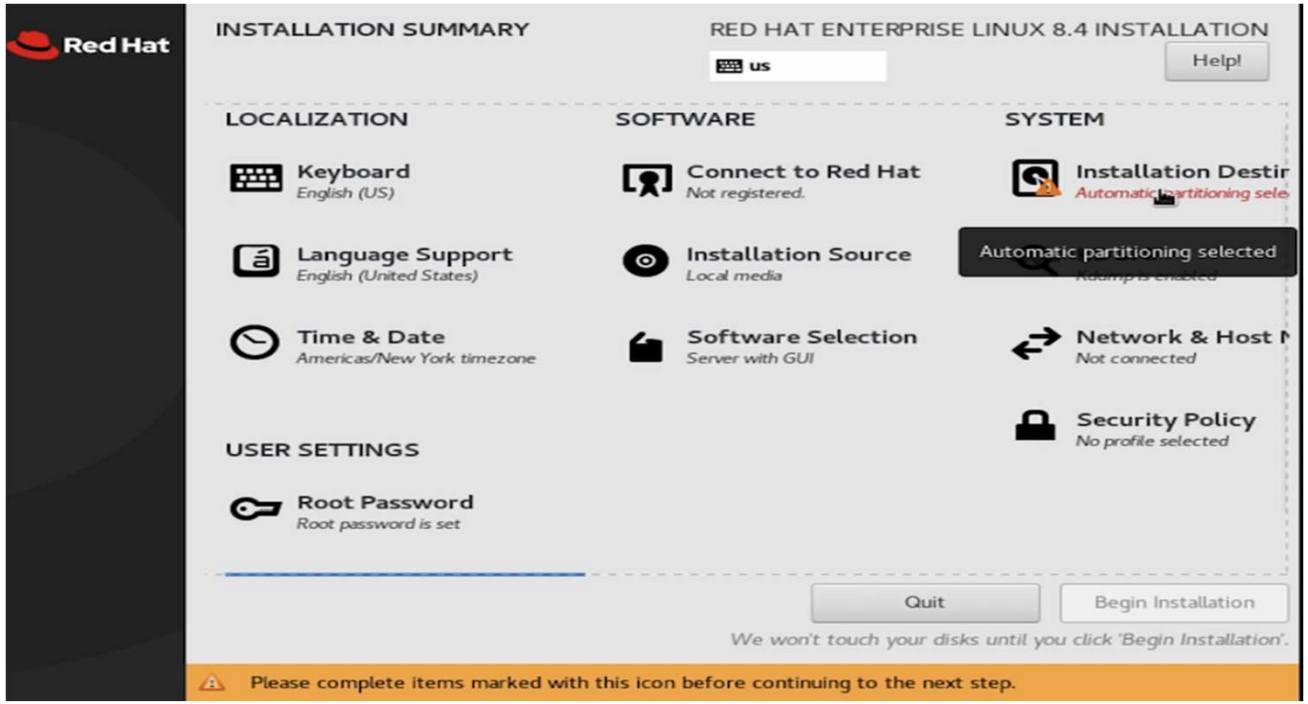
INSTALLING RED HAT LINUX



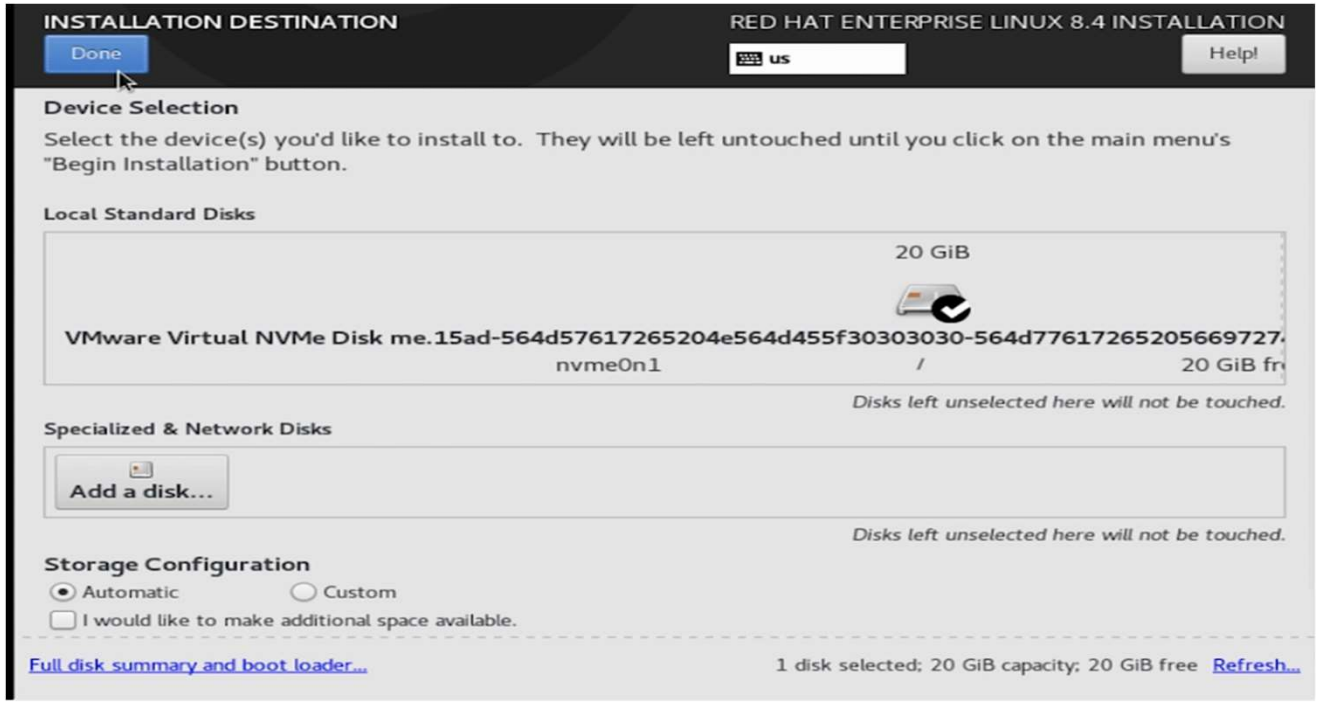
INSTALLING RED HAT LINUX



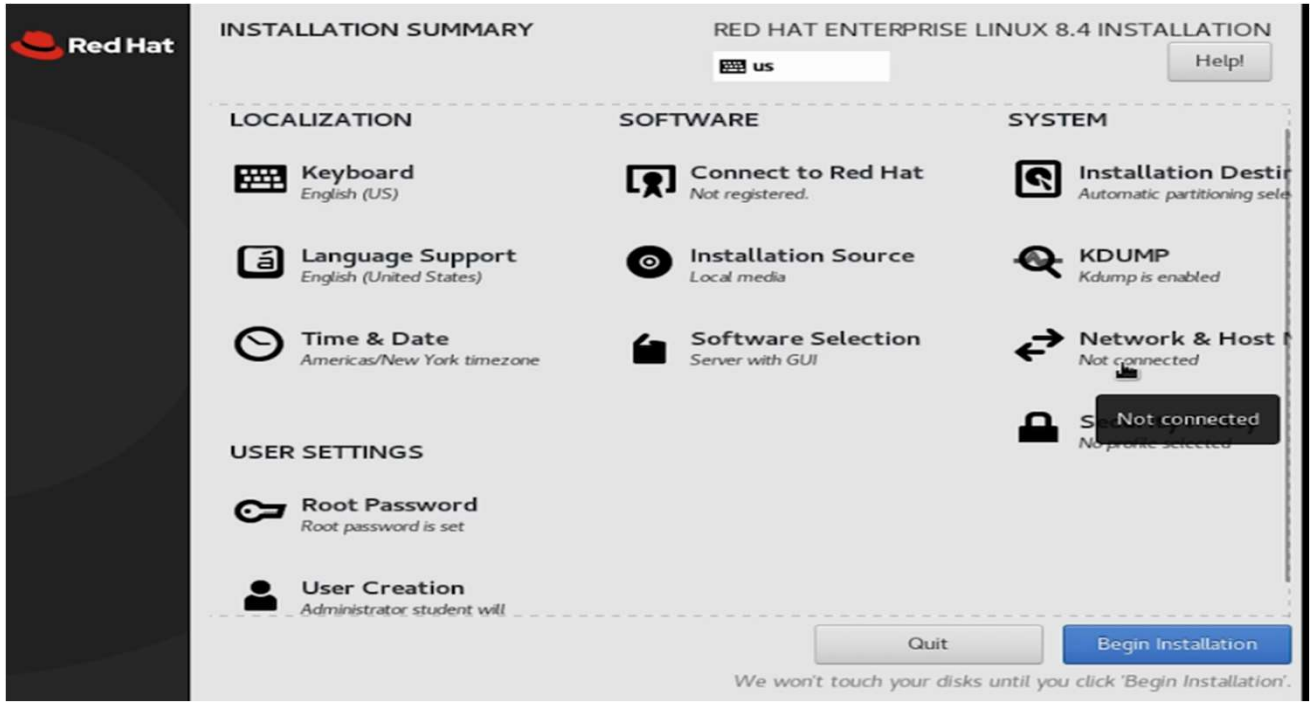
INSTALLING RED HAT LINUX



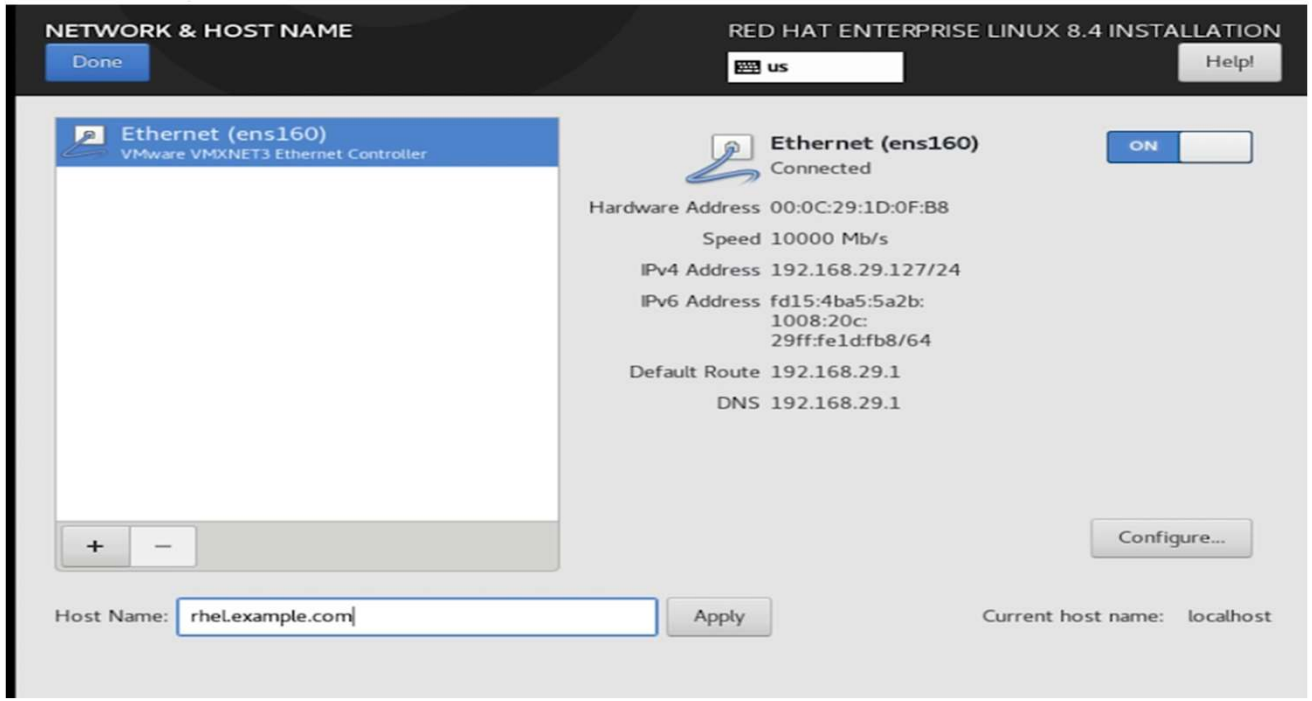
INSTALLING RED HAT LINUX



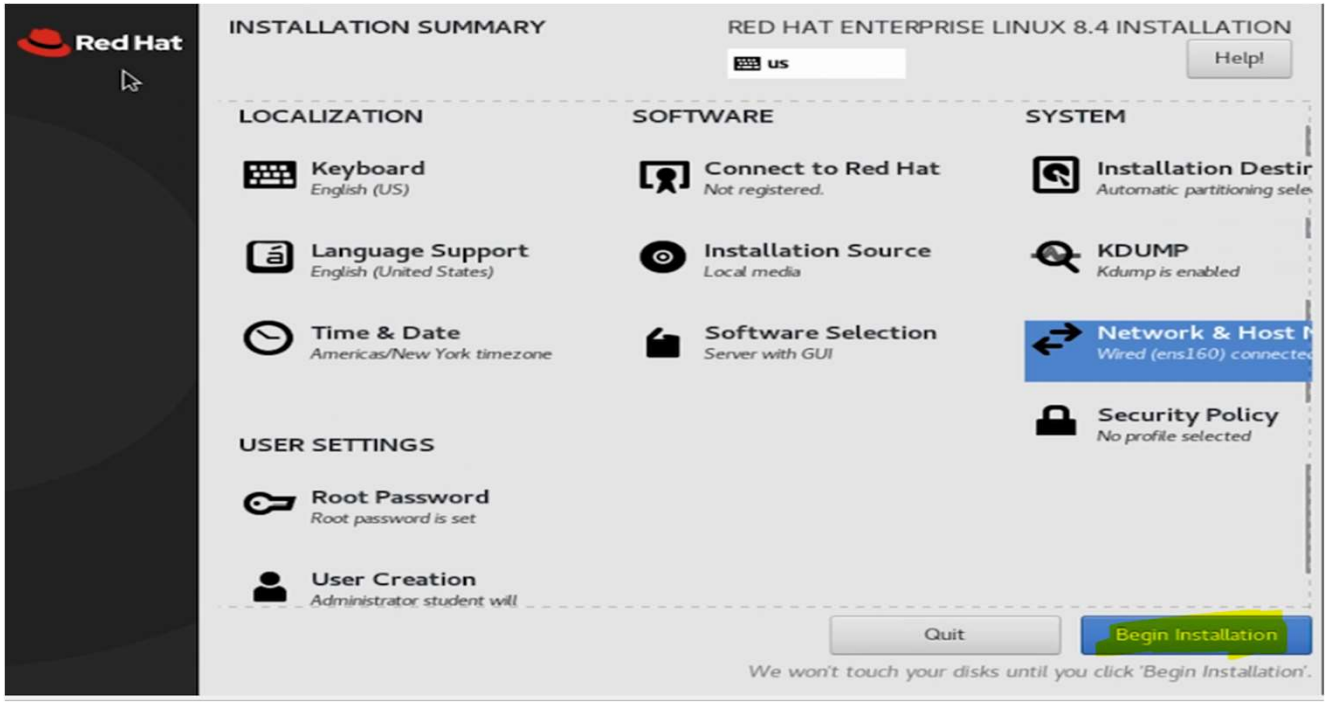
INSTALLING RED HAT LINUX



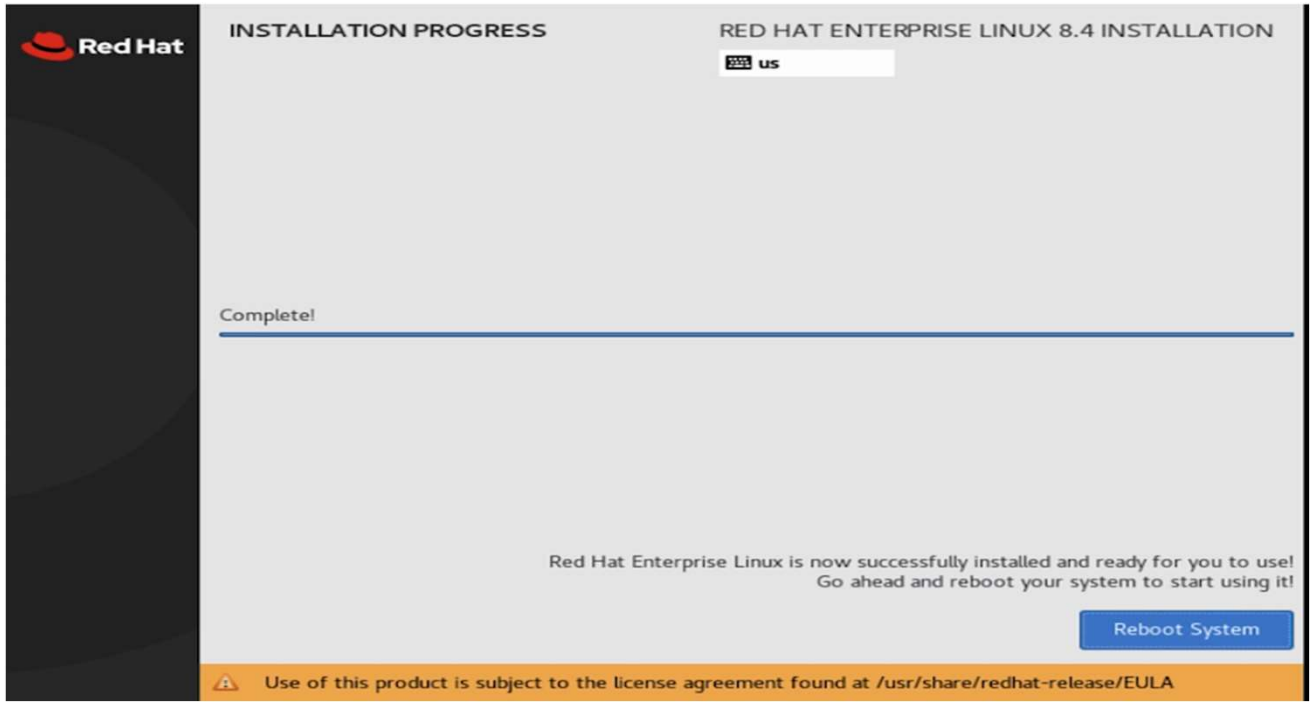
INSTALLING RED HAT LINUX



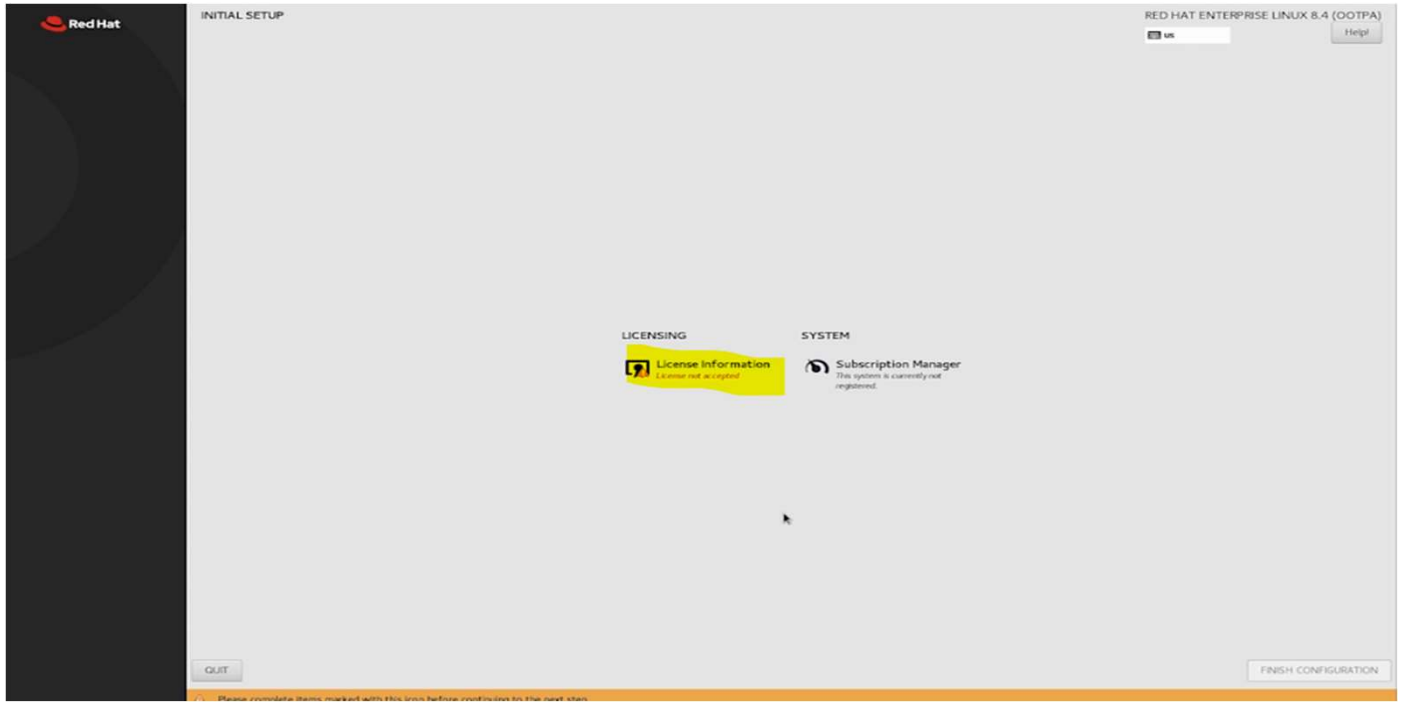
INSTALLING RED HAT LINUX



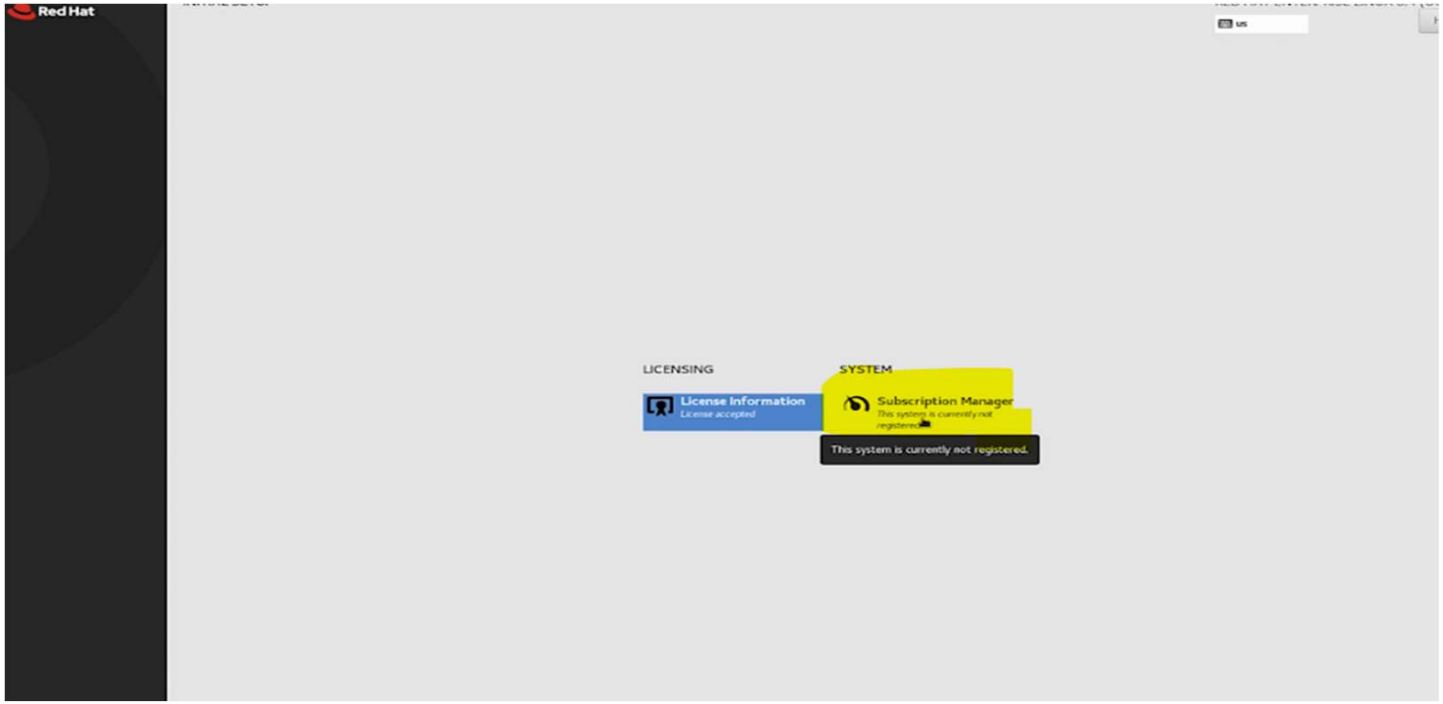
INSTALLING RED HAT LINUX



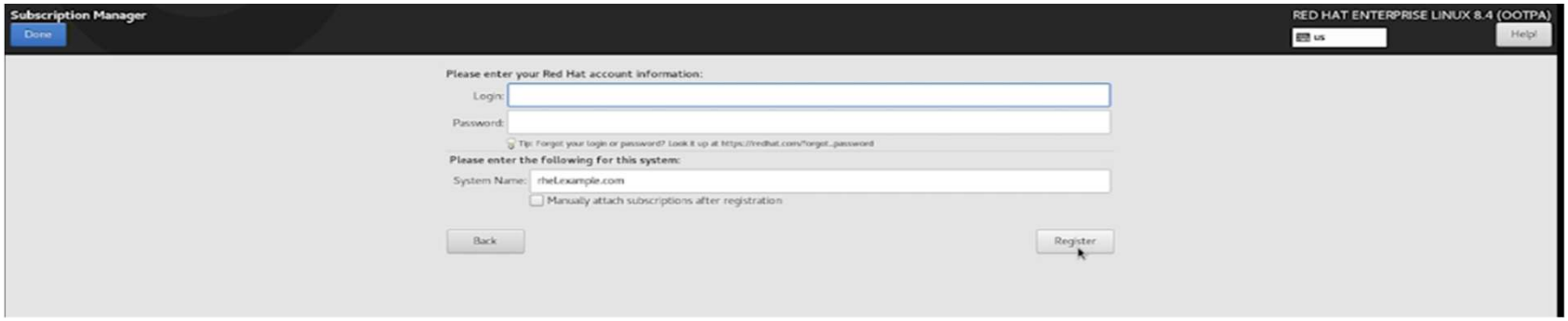
INSTALLING RED HAT LINUX

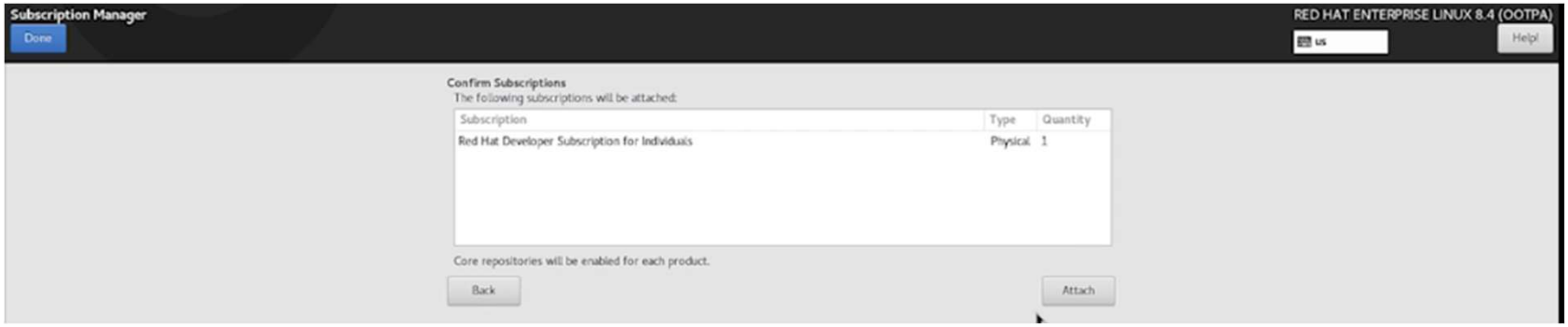


INSTALLING RED HAT LINUX

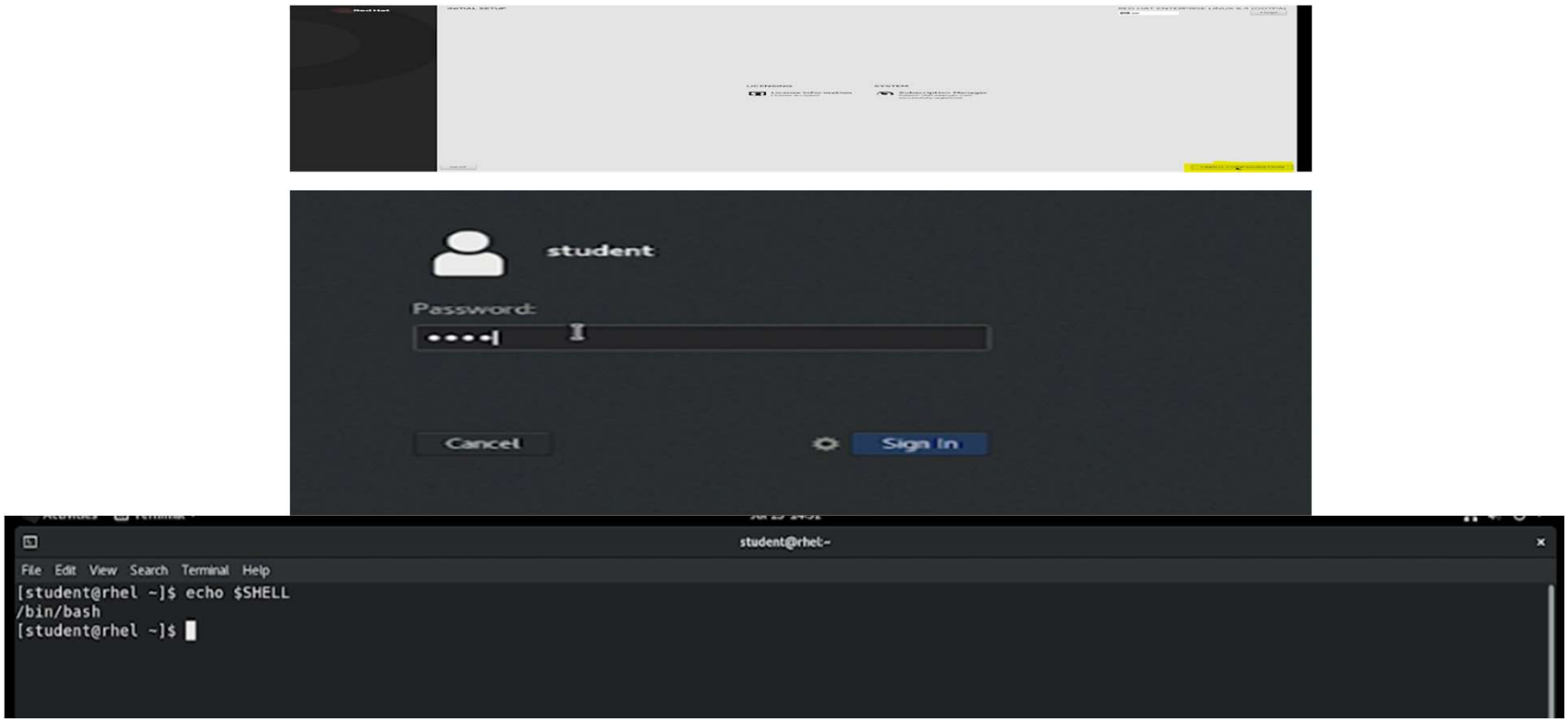


INSTALLING RED HAT LINUX





INSTALLING RED HAT LINUX



USING WINDOWS SUBSYSTEM FOR LINUX

- Windows 10 and later include windows subsystem (WSL) for Linux
- Use WSL 2 for full functionality
- To use it, You will install a virtual machine and that require a computers with Hyper-V virtualization support
- Installing WSL 2 is a 4 step procedure
  - Enable WSL
  - Enable virtual machine Platform
  - Set WSL 2 as the default
  - Install a Linux Distribution

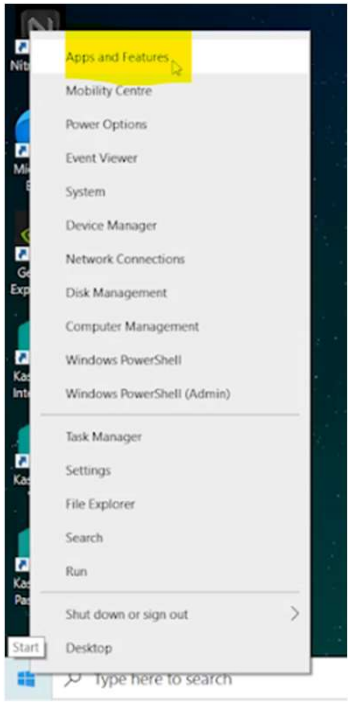
USING WINDOWS SUBSYSTEM FOR LINUX

- Right-click Windows menu
- Select **Apps & Features**
- Scroll all the way down and select **Programs and Features**
- Select **Turn Windows Features on or off**
- Select **Hyper-V** as well as **Windows Subsystem for Linux**
- Reboot when asked for

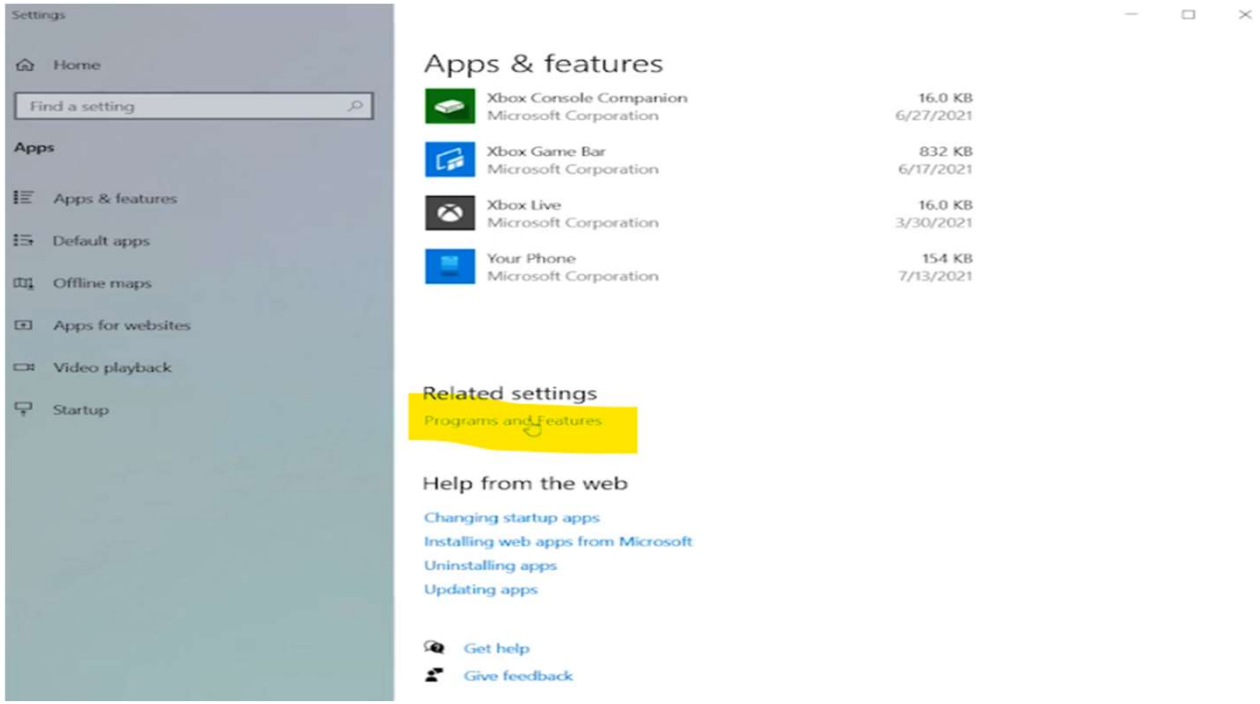
USING WINDOWS SUBSYSTEM FOR LINUX

- Open the **Microsoft Store** app
- From here, look for a Linux distribution (such as Ubuntu 20.04 or later version)
- Also install the **open source Windows Terminal** app
- After installation, a shortcut is added to the Start Menu, use it to open the Linux distribution you have installed

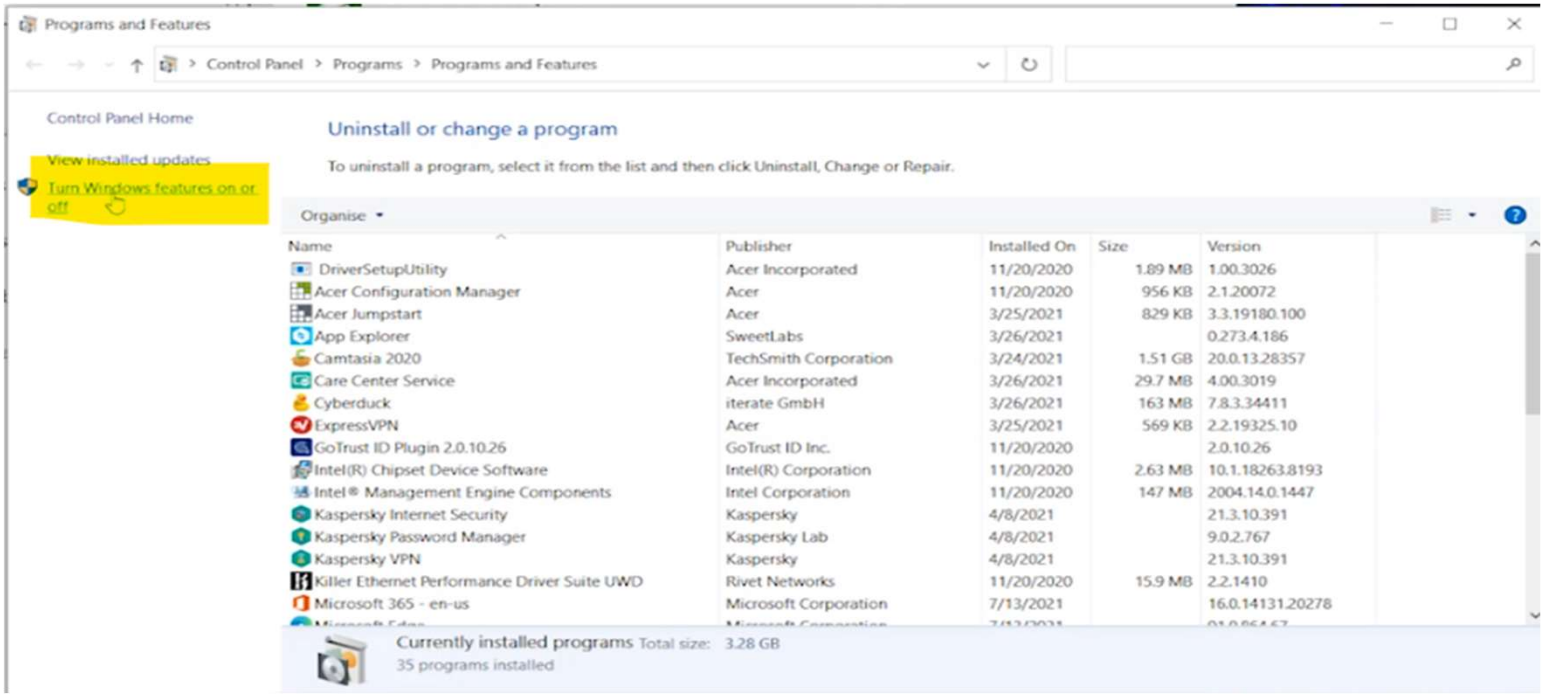
USING WINDOWS SUBSYSTEM FOR LINUX



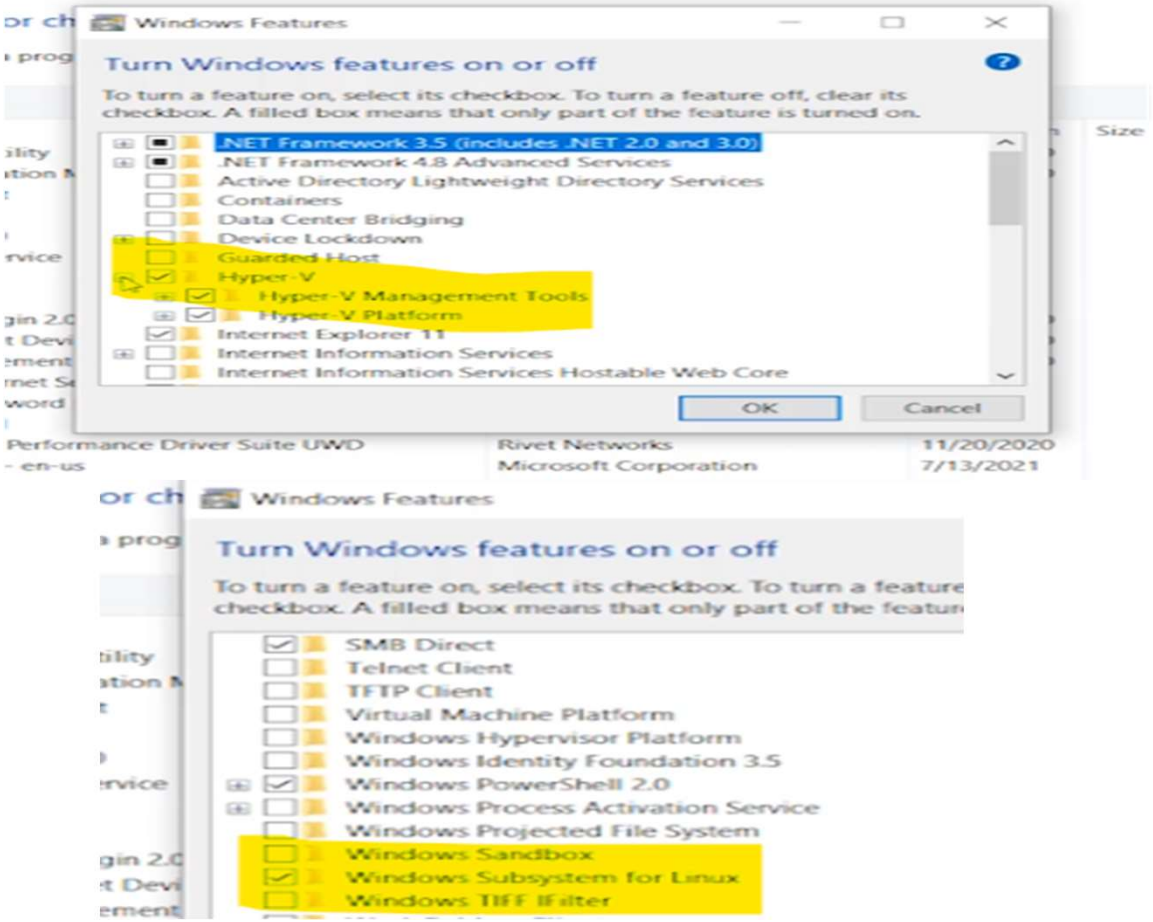
USING WINDOWS SUBSYSTEM FOR LINUX



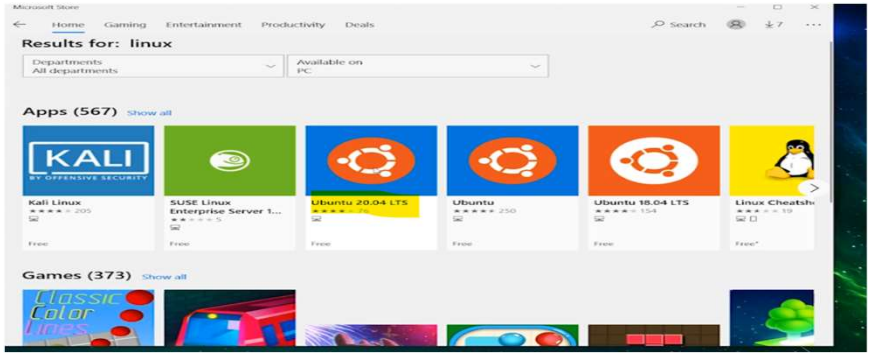
USING WINDOWS SUBSYSTEM FOR LINUX

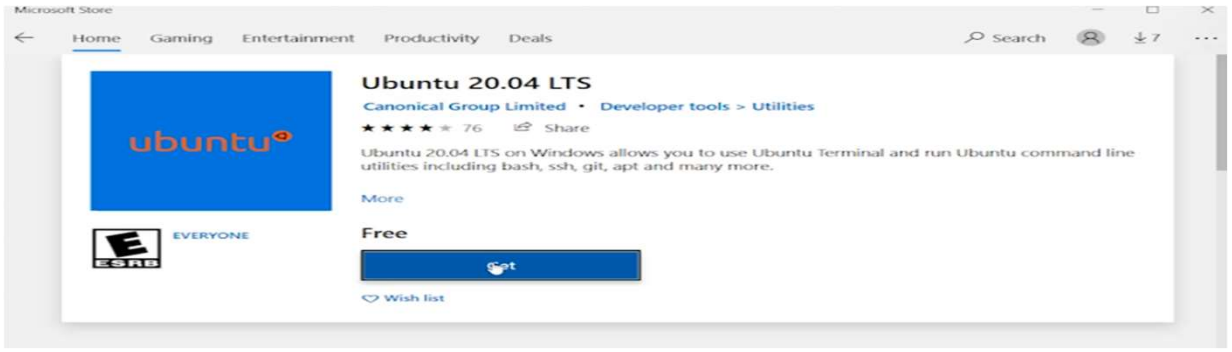


USING WINDOWS SUBSYSTEM FOR LINUX

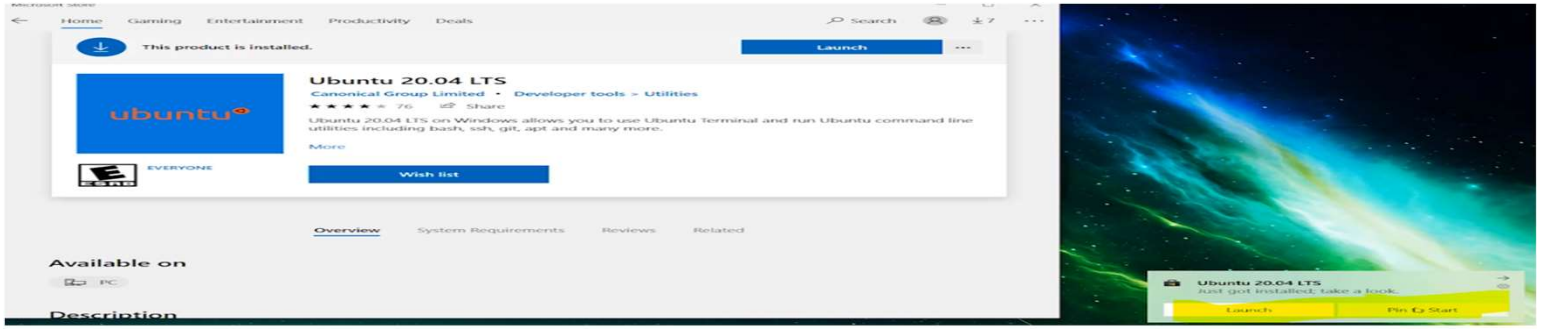


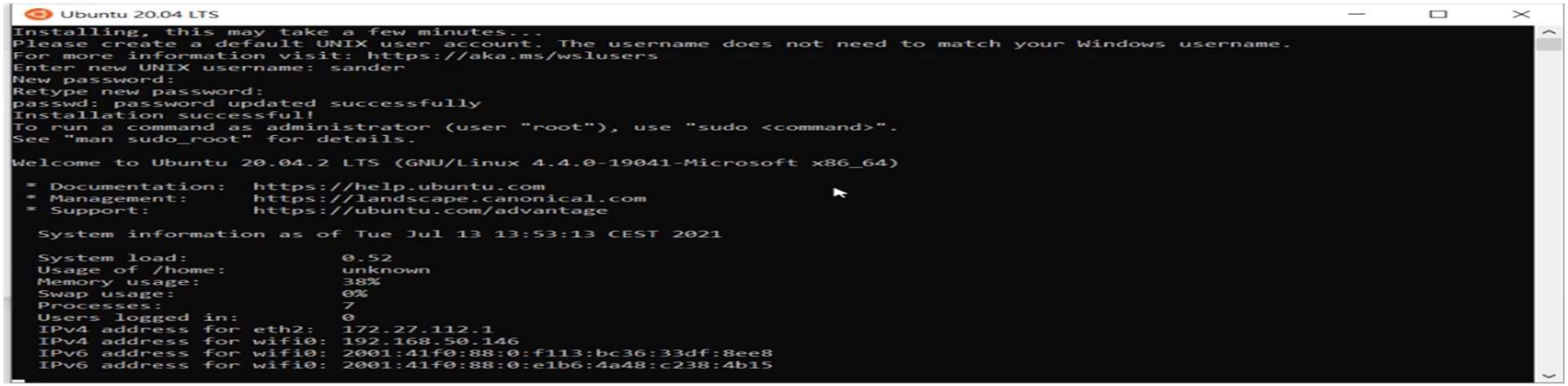
USING WINDOWS SUBSYSTEM FOR LINUX





USING WINDOWS SUBSYSTEM FOR LINUX





USING WINDOWS SUBSYSTEM FOR LINUX

$ Cat /etc/os-releases
$ free –m
$ echo $SHELL

BASH INTRODUCTION

- Understanding the Role of Bash
- Using STDIN, STDOUT,STDERR and I/O Redirection
- Using Internal Commands
- Using Variables
- Working with alias
- Using Bash Start-up Files
- Understanding Alternative Shells
- Understanding Exit Codes
- Using Bash

UNDERSTANDING THE ROLE OF BASH

- Bash (short for Bourne-Again SHell) is a command-line interface and scripting language used in many Unix-based operating systems. It is a powerful tool for interacting with a computer system and performing a wide range of tasks, from simple file management to complex system administration.
- Here are some key points to help understand the role of Bash:

1. Bash is the default shell for most Unix-based operating systems. When you open a terminal window on a Linux or macOS system, you are typically interacting with Bash.
2. Bash allows users to execute commands and scripts, making it a powerful tool for automation and system administration.
3. Bash is highly customizable, with many configuration options and the ability to create custom scripts and aliases.
4. Bash supports variables and functions, making it possible to create complex scripts that can take user input and perform a series of actions based on that input.
5. Bash is often used in conjunction with other command-line tools, such as grep, awk, and sed, to perform more complex operations on files and directories.
6. Bash can be used to create scripts that can be run automatically on a schedule or in response to specific events.

Overall, Bash plays a crucial role in the Unix-based computing ecosystem, allowing users to interact with and automate complex system operations. Its power and flexibility make it an essential tool for system administrators, developers, and anyone else who works with Unix-based operating systems.

USING STDIN, STDOUT,STDERR AND I/O REDIRECTION IN LINUX

- STDIN, STDOUT, and STDERR are streams that are used in Linux and Unix-based operating systems to manage input and output from various commands and programs. I/O redirection is a technique that allows you to redirect these streams to different sources or destinations, allowing you to control the flow of data in your system. Here are some key points about STDIN, STDOUT, STDERR, and I/O redirection in Linux:

1. STDIN is a stream used for input to a program. It is typically used to read user input from the keyboard or to read data from a file.
2. STDOUT is a stream used for output from a program. It is typically used to display information on the terminal screen or to write data to a file.
3. STDERR is a stream used for error output from a program. It is typically used to display error messages or warnings on the terminal screen or to write error data to a file.
4. I/O redirection is a technique that allows you to redirect the streams of a program to different sources or destinations. This is done using the > and < operators in the command line.
5. The > operator is used to redirect STDOUT to a file or another program. For example, the command "ls > myfile.txt" will redirect the output of the "ls" command to the "myfile.txt" file.
6. The < operator is used to redirect STDIN to a program or file. For example, the command "sort < myfile.txt" will sort the contents of the "myfile.txt" file.
7. The 2> operator is used to redirect STDERR to a file or another program. For example, the command "ls /nofile 2> error.log" will write any error messages to the "error.log" file.
8. The | (pipe) operator is used to redirect the output of one program to the input of another program. For example, the command "ls | grep myfile" will display a list of files and directories that contain the word "myfile".

Overall, STDIN, STDOUT, STDERR, and I/O redirection are powerful tools that can help you manage input and output from various programs and commands in Linux. By using these streams and redirection techniques, you can control the flow of data in your system and automate complex tasks more effectively.

EXAMPLES

- Reading input from a file using STDIN: This command reads the contents of the myfile.txt file and sorts them alphabetically using the sort command.
- sort < myfile.txt

- Redirecting STDOUT to a file: This command lists all the files and directories in the current directory and redirects the output to the filelist.txt file.
- ls > filelist.txt

- Redirecting STDERR to a file: This command tries to list a file that does not exist and redirects any error messages to the error.log file.
- ls /nofile 2> error.log

- Redirecting STDOUT to STDERR: This command redirects the output of the ls command to STDERR instead of STDOUT.
- ls /nofile 1>&2

- Piping STDOUT to another program: This command lists all the files and directories in the current directory and pipes the output to the grep command, which searches for the word "myfile" and displays the results on the terminal.
- ls | grep myfile

- Redirecting both STDOUT and STDERR to a file: This command lists a file that does not exist and redirects both the STDOUT and STDERR streams to the output.log file.
- ls /nofile &> output.log

Overall, these examples demonstrate how STDIN, STDOUT, STDERR, and I/O redirection can be used to manage input and output in Linux and Unix-based operating systems.

USING PIPES ALONG WITH STDIN, STDOUT, STDERR, AND I/O REDIRECTION IN LINUX:

- Piping the output of one command to another command, and redirecting the output of the second command to a file:
- ls | grep myfile > filelist.txt
- This command lists all the files and directories in the current directory, pipes the output to the grep command, which searches for the word "myfile", and redirects the output of grep to the filelist.txt file.

- Redirecting STDERR to STDOUT using pipes:
- ls /nofile 2>&1 | grep "No such file"
- This command tries to list a file that does not exist, redirects the STDERR stream to the STDOUT stream using 2>&1, pipes the output to the grep command, which searches for the phrase "No such file", and displays the results on the terminal.

- Using pipes to concatenate two or more commands:
- ls | sort | uniq
- This command lists all the files and directories in the current directory, pipes the output to the sort command, which sorts the list alphabetically, and pipes the output to the uniq command, which removes any duplicates.

Overall, these examples demonstrate how pipes can be used along with STDIN, STDOUT, STDERR, and I/O redirection to perform complex tasks in Linux and Unix-based operating systems. By combining these tools, you can create powerful and flexible workflows that can automate many different types of tasks.

INTERNAL COMMANDS

- In Linux, internal commands are commands that are built into the shell itself, rather than being separate executable programs. They are typically faster and more efficient than external commands because they don't require the overhead of creating a new process. Here are some key points about using internal commands in Linux:
- Internal commands are typically invoked using the shell itself, rather than being called as separate programs. For example, the cd command is an internal command that is used to change the current working directory.
- Internal commands are often used for tasks that are fundamental to the operation of the shell, such as managing processes, changing permissions, and manipulating files and directories.
- Because internal commands are built into the shell, they are typically faster and more efficient than external commands that must be executed as separate processes.
- Internal commands often have additional options and functionality beyond what is available with external commands.
- Common internal commands in Linux include:

- cd: Changes the current working directory
- echo: Displays a message on the terminal
- exit: Exits the current shell session
- alias: Creates or modifies a command alias
- history: Displays a list of recently executed commands
- export: Sets an environment variable

- Many internal commands have a corresponding external command with the same name. In these cases, the external command will be executed if it exists, rather than the internal command.
- Internal commands are often used in shell scripts, which are collections of commands and instructions that can be executed as a single unit.

Overall, internal commands are an important part of the Linux shell environment, providing fast and efficient access to fundamental system operations. By learning how to use internal commands effectively, you can become more proficient at using the Linux shell and automate many common tasks.

USING VARIABLES

- In Bash scripting, variables are used to store and manipulate data. They can be assigned a value, which can be a string, number, or other data type, and then used in various commands and operations. Here are some key points about using variables in Bash scripting, along with examples:

Variables in Bash scripting are typically defined using the following syntax:

- variable_name=value

For example, the following code defines a variable named name and assigns it the value "Jai":

- name="Jai“

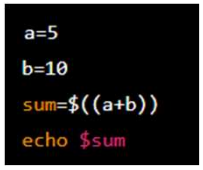
Variables in Bash scripting can be referenced using the $ symbol followed by the variable name. For example, the following code uses the echo command to display the value of the name variable:

- echo $name

This will display "Jai" on the terminal.

Variables in Bash scripting can be used in arithmetic operations using the $(( )) syntax. For example, the following code defines two variables, a and b, and uses them in an arithmetic operation to calculate the sum:

```
a=5
b=10
sum=$((a+b))
echo $sum
```



This will display "15" on the terminal.

Variables in Bash scripting can be passed as arguments to functions or scripts. For example, the following code defines a function that takes a variable as an argument and displays its value:

- function display_name {
- echo "Name: $1"
- }
- name="John"
- display_name $name
- This will display "Name: John" on the terminal.

```
function display_name {
    echo "Name: $1"
}

name="John"
display_name $name
```

Variables in Bash scripting can be manipulated using various string operations, such as concatenation, substring extraction, and pattern matching. For example, the following code defines a variable named greeting and concatenates it with the name variable to create a personalized greeting:

- greeting="Hello"
- name="John"
- echo "$greeting, $name!"
- This will display "Hello, John!" on the terminal.

```
greeting="Hello"
name="John"
echo "$greeting, $name!"
```

Overall, variables are a powerful tool in Bash scripting, allowing you to store and manipulate data, pass arguments between functions and scripts, and perform a wide range of operations using various string and arithmetic operations. By mastering the use of variables in Bash scripting, you can create flexible and powerful scripts and automate many common tasks.

```
student@student-virtual-machine:~$ echo $SHELL
/bin/bash
student@student-virtual-machine:~$ echo $PATH
/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games:/usr/local/games:/snap/bin:/snap/bin
student@student-virtual-machine:~$ myvar=green
student@student-virtual-machine:~$ echo $myvar
green
student@student-virtual-machine:~$ echo $myvar1

student@student-virtual-machine:~$ echo ${myvar}1
green1
student@student-virtual-machine:~$ bash
student@student-virtual-machine:~$ echo $myvar

student@student-virtual-machine:~$ exit
exit
student@student-virtual-machine:~$
```

```
student@student-virtual-machine:~$ exit
exit
student@student-virtual-machine:~$ export mynewvar=red
student@student-virtual-machine:~$ bash
student@student-virtual-machine:~$ echo $mynewvar
red
student@student-virtual-machine:~$ exit
```

ALIAS IN LINUX

- **alias** is a Bash internal command that allows you to define your own commands
- It is convenient to use **alias** to replace long commands with something shorter
- Type **alias** in a shell to get a list of current aliases
- Best practice: Do NOT use alias in shell scripts, as alias settings are not universal and might not exist on other computers where the shell script is used

- In Linux, an alias is a way to create a shortcut or abbreviation for a longer command or series of commands. It allows you to save time and simplify your workflow by typing a shorter command instead of a longer one. Here are some key points about working with aliases in Linux:

Aliases can be defined using the following syntax:

- alias new_command='original_command'

Aliases can be removed using the following syntax:

- unalias new_command

Aliases can be used with command-line arguments. For example, the following alias defines a shortcut for searching for a file using the find command:

- alias findfile='find / -name'
- With this alias, you can search for a file by typing findfile myfile.txt instead of the longer find / -name myfile.txt.

Aliases can also be used to create more complex shortcuts that involve multiple commands. For example, the following alias defines a shortcut for cleaning up temporary files and directories:

- alias clean='rm -rf /tmp/*; rm -rf ~/.cache/*'
- With this alias, you can clean up temporary files and directories by typing clean instead of the longer rm -rf /tmp/*; rm -rf ~/.cache/*.

BASH STARTUP FILES

- In Bash scripting, startup files are scripts that are run automatically when a new shell session is started. These scripts are used to set environment variables, define aliases, and perform other tasks that are necessary for the shell to function properly. Here are some key points about using Bash startup files:

Bash startup files are typically located in the user's home directory. The filenames for these files may differ depending on the type of shell and the specific Linux distribution being used. Some common filenames for Bash startup files include:

- ~/.bashrc: This file is run every time a new interactive shell session is started. It is typically used to set environment variables, define aliases, and perform other tasks that are specific to the user.
- ~/.bash_profile: This file is run only when the user logs in to the system. It is typically used to set up the user's environment, such as defining the PATH variable and running other scripts.
- ~/.bash_login: This file is run only when the user logs in to the system and does not have a .bash_profile file. It is typically used to set up the user's environment, such as defining the PATH variable and running other scripts.
- ~/.profile: This file is run by some other shells besides Bash, and is typically used to set up the user's environment, such as defining the PATH variable and running other scripts.

Bash startup files are executed in a specific order, depending on the type of shell and the specific Linux distribution being used. Typically, the ~/.bashrc file is executed first, followed by the ~/.bash_profile or ~/.profile file.

Bash startup files can contain any valid Bash commands or scripts, including setting environment variables, defining aliases, running other scripts, and performing other tasks that are specific to the user or the system.

Changes made to Bash startup files will take effect the next time a new shell session is started. To apply the changes immediately, the source command can be used to reload the startup files. For example, the following command will reload the ~/.bashrc file:

```
source ~/.bashrc
```

- Bash startup files can also be used to set up global environment variables and aliases that are available to all users on the system. These files are typically located in the /etc directory and have names such as /etc/profile, /etc/bash.bashrc, and /etc/bashrc.

- Overall, Bash startup files are an essential part of the Linux shell environment, providing a way to set up the user's environment, define aliases, and perform other tasks that are necessary for the shell to function properly. By mastering the use of Bash startup files, you can create customized and efficient shell environments that meet your specific needs and preferences.

UNDERSTANDING ALTERNATIVE SHELLS IN LINUX

- The Bash shell goes back to shells that were created for use in UNIX in the 1970s
- Bourne shell (/bin/sh) was the original shell
- C-shell (/bin/csh) was developed as a shell that is very close to the C programming language
- Korne shell (/bin/ksh) was created as a shell that offers the best of Bourne and C-shell

- Bash is Bourne Again Shell, a remake of the original Bourne shell that was invented in the early 1970's
- Bash is the default shell on most Linux distributions
- All other common Linux shells are a fork of Bourne shell
- Another common shell is Zsh, which is used as the default shell on MacOS
- And yet another common shell is Dash, which is used frequently in Debian environments
- While writing shell scripts, Bash is the standard and it's very easy to make Bash work, even if you're in a non-Bash shell

EXIT CODES

- After execution, a command generates an exit code
- The last exit code generated can be requested using **echo $?**
- If 0, the command was executed successfully
- If 1, there was a generic error
- The developer of a program can decide to code other exit codes as well
- In shell scripts, this is done by using **exit *n*** in case an error condition occurs

```
student@student-virtual-machine:~$ ls
Desktop  Documents  Downloads  grepout.txt  Music  outfile  Pictures  Public  Templates  Videos
student@student-virtual-machine:~$ echo $?
0
student@student-virtual-machine:~$ ls kjhkjhkjhkjghku
ls: cannot access 'kjhkjhkjhkjghku': No such file or directory
student@student-virtual-machine:~$ echo $?
2
student@student-virtual-machine:~$ man ls
student@student-virtual-machine:~$ ls /root
ls: cannot open directory '/root': Permission denied
student@student-virtual-machine:~$ echo $?
2
student@student-virtual-machine:~$
```

- In Linux, exit codes are numerical values that are returned by a command or script when it completes execution. These codes provide information about the success or failure of the command or script, and are used by other programs and scripts to determine the appropriate action to take. Here are some key points about exit codes in Linux:

- Exit codes are typically defined as integers, with 0 representing success and non-zero values representing failure. Different exit codes may be used to indicate different types of failure, such as syntax errors, missing files, or permission issues.

- Exit codes are returned by both internal commands (commands built into the shell) and external commands (standalone programs that are executed by the shell).

- Exit codes can be accessed using the $? variable, which contains the exit code of the most recently executed command. For example, the following code runs the ls command and then displays its exit code:

```
ls
echo $?
```

- Exit codes can be used in shell scripts to perform different actions based on the success or failure of a command or script. For example, the following code checks whether a file exists, and then performs different actions based on the result:

```
if [ -e myfile.txt ]
then
    echo "File exists"
else
    echo "File does not exist"
fi
```

- Different programs and scripts may define their own exit codes to provide more detailed information about the success or failure of their specific operation.
- Common exit codes in Linux include:
- 0: Success
- 1: General error
- 2: Misuse of shell builtins
- 126: Command cannot execute
- 127: Command not found
- 128: Invalid argument to exit
- 130: Command terminated by Ctrl-C

Overall, exit codes provide a way to communicate the success or failure of a command or script to other programs and scripts in the Linux environment. By understanding how exit codes work and how to use them in shell scripts, users can create more robust and reliable scripts that can handle different types of errors and failures.

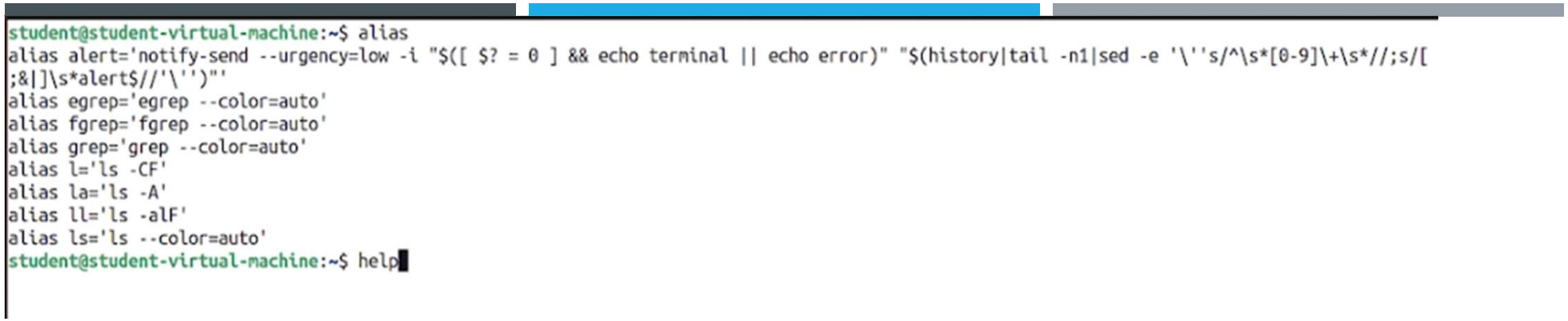
DO IT YOURSELF

- On your favorite working environment, open a terminal
- Type **echo $SHELL** to find which shell is currently used
- If this is not Bash, use **chsh -s /bin/bash** to make Bash the default shell
- Type **env** to get a list of current environment variables
- Type **cat /etc/profile | less** to print the contents of the /etc/profile file
- Type **alias** to get a list of aliases that are set
- Use **help** to show a list of internal commands

SOLUTION

```
student@student-virtual-machine:~$ echo $SHELL
/bin/bash
student@student-virtual-machine:~$ which zsh
student@student-virtual-machine:~$ sudo apt install zsh
[sudo] password for student:
Reading package lists... Done
Building dependency tree... Done
Reading state information... Done
The following additional packages will be installed:
  zsh-common
Suggested packages:
  zsh-doc
The following NEW packages will be installed:
  zsh zsh-common
0 upgraded, 2 newly installed, 0 to remove and 0 not upgraded.
Need to get 4,457 kB of archives.
After this operation, 18.0 MB of additional disk space will be used.
Do you want to continue? [Y/n] Y
Get:1 http://us.archive.ubuntu.com/ubuntu hirsute/main amd64 zsh-common all 5.8-6 [3,746 kB]
7% [1 zsh-common 370 kB/3,746 kB 10%]
```

```
student@student-virtual-machine:~$ echo $SHELL
/bin/bash
student@student-virtual-machine:~$ chsh -s /bin/zsh
Password:
student@student-virtual-machine:~$ echo $SHELL
/bin/bash
student@student-virtual-machine:~$ chsh -s /bin/bash
Password:
student@student-virtual-machine:~$ env
student@student-virtual-machine:~$ cat /etc/profile
```


student@student-virtual-machine:~$ alias
alias alert='notify-send --urgency=low -i "$([$? = 0] && echo terminal || echo error)" "$(history|tail -n1|sed -e '\''s/^\s*[0-9]\+\s*//;s/[
;&|]\s*alert$//'\'')"'
alias egrep='egrep --color=auto'
alias fgrep='fgrep --color=auto'
alias grep='grep --color=auto'
alias l='ls -CF'
alias la='ls -A'
alias ll='ls -alF'
alias ls='ls --color=auto'
student@student-virtual-machine:~$ help

SHELL SCRIPTS FOR DEVOPS ENVIRONMENT

- What is Shell Script?
- What is a DevOps Environment?
- Bash and Other Shells
- Shell Scripts Vs . Automation
- Shell Scripts Vs. Python
- Bash Shell Scripts Vs. Power Shell Scripts

- Do it yourself: Running a Bash Shell in Zsh
- Solution : Running a Bash Shell in Zsh

WHAT IS A SHELL SCRIPTS ?

- A shell script is a computer program designed to run in a shell
- Scripts can be written in different scripting languages
- Typical functions are file manipulation, program executing and printing text

- Shell scripts are a part of the default working environment (shell)
- Shell scripts are strong in manipulating data
- You can use them, for instance, to filter ranges, change file names, and change data on a large scale easily
  - Shell scripts are easy to develop, and run from the leading Linux operating system
  - Shell scripts are commonly used in data science and other professional environments

DEVOPS ENVIRONMENT?

- DevOps is a set of practices that combines software development (dev) with IT Operations (ops)
- The purpose of DevOps is to shorten the system development life cycle
- DevOps is a generic approach, which can be implemented in different ways
  - Toolchains
  - CI/CD pipelines
  - 12-factor application development
  - Deployment strategies
- In DevOps, toolchains are typically used to bring an application from source code to full operation
  - Coding
  - Building
  - Testing
  - Packaging
  - Releasing
  - Configuring
  - Monitoring

- CI is Continuous Integration
- is Continuous Development
- CI/ CD car-I be automated in a pipeline

- The 12-factor App is a methodology for building software-as-a-service apps that defines different factors which should be used in the apps
  - Codebase: one code base, tracked by revision control
  - Dependencies: explicit and isolated dependencies
  - Config: Configuration as code, stored in the environment
  - Backing services: treated as attached resources
  - Build. release, run: separate build and run stages
  - Processes: execute the app as stateless process
  - Port Binding: to expose services
  - Concurrency: the option to scale up and down
  - Disposability: each instance can be replaced
  - Dev/prod parity: keep all stages as similar as possible
  - Logs: treat logs as separate event streams
  - Admin processes: treated separately

- No matter which flow you're using in DevOps, it all comes down to processing files through different stages
- This is a task that can be done perfectly using shell scripts
- Shell scripts can pick up files, filter them, rename them and process them for further treatment using a wide range of tools that are native to the Linux operating system
- While using shell scripts in DevOps, it's important to develop them in an idempotent way

BASH AND OTHER SHELLS

- Bash is the default shell in most Linux distributions and is widely used by system administrators and developers. However, there are other shells available in Linux that offer different features, syntax, and interfaces. Here are some key points about Bash and other shells in Linux:

1. Bash is a widely used shell that offers advanced features such as command-line completion, command history, and job control. It is the default shell in most Linux distributions and is compatible with many existing shell scripts and programs.
2. Zsh is an alternative shell that offers advanced features such as spelling correction, autosuggestions, and theme customization. It also has advanced command-line completion features that make it popular among developers and power users.
3. Fish is a shell that features syntax highlighting, autosuggestions, and built-in functions for common tasks. It has a simplified syntax that makes it easy to use for beginners and is popular among users who want a more user-friendly shell experience.
4. Tcsh is a shell that offers command-line editing, file name completion, and job control features. It is popular among users who are migrating from other Unix systems and are familiar with the C shell syntax.
5. Other shells such as Ksh, Dash, and Ash are also available in Linux and offer different features and syntax.
6. Shells can be customized using configuration files, which are typically located in the user's home directory. Configuration files can be used to set up environment variables, define aliases, and customize the shell prompt appearance.
7. Shells can also be customized using third-party tools, such as Oh My Zsh and Powerline, which provide pre-configured themes and plugins for popular shells.

- Overall, Bash and other shells in Linux provide users with different options and features for customizing their shell environment. By exploring different shells and experimenting with their features and syntax, users can create a customized and efficient command-line experience that meets their specific needs and preferences.

SHELL WITH OS

- Bash is the default shell in most Unix-based operating systems, including Linux, macOS, and BSD. However, there are several other shells that are used in different operating systems, each with their own unique features and syntax. Here are some key points about the different shells used in different operating systems:

1. Bash: Bash (Bourne-Again SHell) is the default shell in most Unix-based operating systems, including Linux, macOS, and BSD. It is a powerful and versatile shell that offers a wide range of features, including command-line completion, shell scripting, and job control.
2. Zsh: Zsh (Z Shell) is a popular alternative shell that is known for its advanced command-line completion features, spelling correction, and theme customization. It is available for most Unix-based operating systems and is highly customizable.
3. Fish: Fish (Friendly Interactive SHell) is a shell that features syntax highlighting, autosuggestions, and built-in functions for common tasks. It is designed to be more user-friendly than other shells and offers a streamlined interface.
4. Csh: Csh (C SHell) is a shell that is used in some Unix-based operating systems, such as FreeBSD. It offers command-line editing, file name completion, and job control features, but has a different syntax and syntax style than Bash.
5. Windows PowerShell: Windows PowerShell is a shell that is included in Windows operating systems. It offers a wide range of features, including object-oriented scripting, remote management, and support for .NET Framework.
6. cmd.exe: cmd.exe is the default shell in Windows operating systems. It offers a basic command-line interface with limited features compared to Bash and other Unix-based shells.

- Overall, different operating systems and environments may use different shells, each with their own unique features and syntax. By understanding the different shells available and their strengths and weaknesses, users can choose the best shell for their specific needs and preferences.

SHELL SCRIPT VS AUTOMATION

- The aim of automation tools is configuration management
- In automation, tools like Ansible, Puppet, Chef and others are used to get managed systems in a desired state
- To do so, the desired state is described in a file, often written in YAML
- The automation tool compares the current state of managed systems to the desired state and takes action if needed
- If no action is needed, nothing will happen
- Running the Automation tool multiple times, should not lead to anything different than implementation of the desired state; this feature is known as idempotency
- Automation tools can address a wide range of managed assets, with or without using any agents
- Bash is not used to define a desired state
- A Bash script defines actions to be accomplished
- Managing idempotency in Bash scripts is much harder to achieve
- Bash, however, is much more than configuration management; it’s a programming language that helps in processing data, dealing with files, and running very specific tasks
- Automation doesn't replace the need for Bash scripts, both solutions are complimentary to each other

SHELL SCRIPTS VS PYTHON

- Python is an object-oriented programming language that can be used for almost any purpose
- Python comes with many libraries that extend its functionality and make it useful in different environments
- Python also has more advanced debugging tools and error handling features, which make it a better solution for writing bigger programs
- Bash scripts are using Bash shell, and for that reason are easier to program
- Core parts of Linux are written in Bash
- Bash can use command line utilities without any modification
- Bash is part of the shell, and for that reason uses shell features in a more efficient way

BASH SCRIPTS VS POWER SCRIPTS

- PowerShell is the native shell environment for Windows
- PowerShell is more than a shell; it's a complete scripting environment
- PowerShell invokes lightweight commands, the cmdlets to get things done in an efficient way
- Because of its integration with .NET and other core Windows components, PowerShell is the #1 solution for managing Windows

DO IT YOURSELF
RUNNING A BASH SHELL IN ZSH

- Set your current shell to zsh (without changing the default shell).
- Write a simple bash script that prints the text "hello world" on screen. Ensure that it is executed as a Bash script, even if Zsh is the current scripting environment.

SOLUTION

- echo $SHELL
- ZSh

zsh • not found, but can be installed

- sudo apt install zsh

- Zsh

- vim helloworld.sh #INSTALL VIM IF NOT AVAILABLE

- vim helloworld.sh

#!/bin/bash

echo hello world

- chmod +x helloworld.sh
- ./helloworld.sh

LEARNING LINUX ESSENTIALS FOR SHELL SCRIPTING

- Using echo
- Using Printf
- Using Bash Options
- Using Patterns
- Using grep
- Understanding Regular Expressions
- Using Cut and Sort
- Using tall and head
- Using sed
- Using awk
- Considering External tools and performance

- Do It Yourself : Using Linux Commands
- Solution : Using Linux Commands

USING ECHO

- **echo** is a Bash internal that is used to print text on screen
- Different options are supported to manage special characters
- To use the formatting options, use **echo -e** and put the string between double quotes
  - **\b** is backspace**: echo -e "b\bc"**
  - **\n** is newline: **echo -e "b\nc"**
  - **\t** is tab: **echo -e "b\tc"**
- **printf** can be used as an alternative, but has its origin from the C-shell

```
student@student-virtual-machine:~$ echo hello
hello
student@student-virtual-machine:~$ echo -e "b\bc"
c
student@student-virtual-machine:~$ echo -e "b\nc"
b
c
student@student-virtual-machine:~$ echo -e "b\tc\td\ne\tf\t\g"
b       c       d
e       f       \g
student@student-virtual-machine:~$ echo -e "b\tc\td\ne\tf\tg"
b       c       d
e       f       g
student@student-virtual-machine:~$
```

USING PRINTF

- **printf** is used to print text on screen, but has more formatting options than **echo**
- **printf** does not print a new line character by default, use **printf "%s\n" "hello world"** to print a new line
  - **"%s"** is the format string, **"\n"** is the newline formatting character
  - **"\t"** is also common and used to print tab stops
  - The format string is applied to all arguments used with **printf**
- Compare the following commands:
  - **printf "%s\n" hello world**
  - **printf "%s\n" "hello world"**
  - **printf "%s\n" "hello" "world"**

```
student@student-virtual-machine:~$ printf "%s\n" hello world
hello
world
student@student-virtual-machine:~$ printf "%s\n" "hello world"
hello world
student@student-virtual-machine:~$ printf "%s\n" "hello" "world"
hello
world
student@student-virtual-machine:~$ printf "%s\n" "hello" "my world"
hello
my world
student@student-virtual-machine:~$
```

- **"%s"** is used to identify the arguments as strings
- **"%d"** is used to identify the arguments as integers
- **"%f"** is used to identify the arguments as floats
- While using formatting strings, further specifiers can be used
  - **printf "%f\n" 255**
  - **printf "%.1f\n" 255**
  - **for i in $(seq 1 10); do printf "%04d\t" "$i"; done**

```
student@student-virtual-machine:~$ printf "%f\n" 255
255.000000
student@student-virtual-machine:~$ printf "%s\n" 255
255
student@student-virtual-machine:~$ printf "%d\n" 255
255
student@student-virtual-machine:~$ printf "%.1f\n" 255
255.0
student@student-virtual-machine:~$ for i in $( seq 1 10 ); do printf "%04d\t" "$i"; done
0001    0002    0003    0004    0005    0006    0007    0008    0009    0010    student@student-virtual-machine:~$
student@student-virtual-machine:~$
```

```
student@student-virtual-machine:~$ printf "%s\n" 255 0xff 0377
255
0xff
0377
student@student-virtual-machine:~$ printf "%d\n" 255 0xff 0377
255
255
255
student@student-virtual-machine:~$ printf "%f\n" 255 0xff 0377
255.000000
255.000000
377.000000
student@student-virtual-machine:~$
```

BASH OPTIONS

- The Bash shell can be configured with additional options, using **set**
  - **set -x**
  - **ls**
  - **set +x**
- Use **set** in a terminal to make it the standard behavior from that terminal
- Use the set option directly after the shebang in a script to use it in a script only: **#!/bin/bash -x**

- Additional options can be configured from a script using **shopt**
  - **shopt +s extglob** enables extended globbing patterns
- Using these options allows you to add features to the shell and your scripts
  - **shopt -s checkjobs**
  - **sleep 3600 &**
  - **exit**
- Use **shopt** without arguments to print a list of current options

```
student@student-virtual-machine:~$ ls
Desktop  Documents  Downloads  helloworld.sh  Music  Pictures  Public  Templates  Videos
student@student-virtual-machine:~$ set -x
student@student-virtual-machine:~$ ls
+ ls --color=auto
Desktop  Documents  Downloads  helloworld.sh  Music  Pictures  Public  Templates  Videos
student@student-virtual-machine:~$ set +x
+ set +x
student@student-virtual-machine:~$ ls
Desktop  Documents  Downloads  helloworld.sh  Music  Pictures  Public  Templates  Videos
student@student-virtual-machine:~$ shop
```

```
student@student-virtual-machine:~$ shopt -s checkjobs
student@student-virtual-machine:~$ sleep 3600 &
[1] 133661
student@student-virtual-machine:~$ exit
exit
There are running jobs.
[1]+  Running                 sleep 3600 &
student@student-virtual-machine:~$
```

USING PATTERNS

- Regular expressions are patterns that are used by specific tools
- Globbing is for file patterns in Bash
- Basic globbing applies to standard features
  - * matches zero or more characters
  - ? matches any single character
  - [...] matches any of the characters listed
- Extended globbing (must be enabled with **shopt +s extglob**) provides additional options
- ?(patterns): matches zero or one occurences of pattern
- *(patterns): matches zero or more occurences of pattern
- +(patterns): matches one more more occurences of pattern
- @(patterns): matches one occurence of pattern

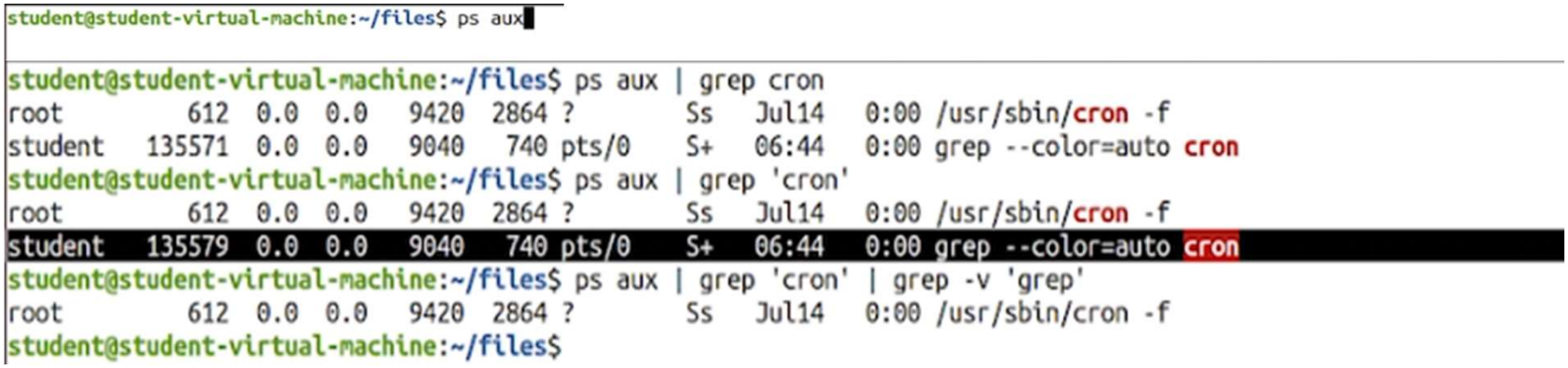
- shopt +s extglob
- touch .txt e.txt ee.txt eee.txt
- ls *.txt
- ls ?(e).txt
- ls *(e).txt
- ls +(e).txt
- ls @(e).txt

```
student@student-virtual-machine:~$ shopt +s extglob
bash: shopt: +s: invalid shell option name
extglob         on
student@student-virtual-machine:~$
```

```
student@student-virtual-machine:~$ mkdir files
student@student-virtual-machine:~$ cd files
student@student-virtual-machine:~/files$ touch .txt e.txt ee.txt eee.txt
student@student-virtual-machine:~/files$ ls -a
.  ..  eee.txt  ee.txt  e.txt  .txt
student@student-virtual-machine:~/files$ ls *.txt
eee.txt  ee.txt  e.txt
student@student-virtual-machine:~/files$ ls -a *.txt
eee.txt  ee.txt  e.txt
student@student-virtual-machine:~/files$ ls ?(e).txt
e.txt  .txt
student@student-virtual-machine:~/files$ ls *(e).txt
eee.txt  ee.txt  e.txt  .txt
student@student-virtual-machine:~/files$ ls +(e).txt
eee.txt  ee.txt  e.txt
student@student-virtual-machine:~/files$ ls @(e).txt
e.txt
student@student-virtual-machine:~/files$
```

USING GREP

- **grep** is an external command that helps you filter text
- Use it to find files containing specific text, or filter command output on occurrence of specific text
- When using **grep**, it is recommended to put the text pattern you're searching for between single quotes to avoid interpretation by the shell
  - **grep 'root' ***
  - **ps aux | grep 'ssh'**
- **grep** uses regular expressions, which are advanced text filters that help you find text patterns in a flexible way



REGULAR EXPRESSION

- Regular expressions, also known as regex or regexp, are a powerful tool for manipulating and searching text in Bash scripting and other programming languages. Regular expressions use a set of characters and operators to define a pattern, which can then be used to match and manipulate text. Here are some key points about regular expressions:
- Regular expressions are used to match patterns in text. The pattern is defined using a combination of characters and operators that specify the type of characters to match and how many times they should be matched.
- Regular expressions can be used to search, replace, and manipulate text in Bash scripting and other programming languages. They are commonly used for tasks such as data validation, parsing, and text manipulation.
- Regular expressions use special characters, such as the dot (.), asterisk (*), and plus sign (+), to match patterns of characters. These characters can be used to match specific characters, groups of characters, or a range of characters.
- Regular expressions can also use character classes to match specific types of characters, such as digits, letters, or whitespace. Character classes are denoted by brackets ([]) and can be used to match a single character from a set of possible characters.
- Regular expressions can be combined with other Bash features, such as variables, loops, and conditional statements, to create powerful text manipulation operations.
- Regular expressions can be tested and debugged using online tools and command-line utilities, such as grep and sed, which provide functionality for searching, replacing, and manipulating text using regular expressions.

Overall, regular expressions are a powerful tool for manipulating and searching text in Bash scripting and other programming languages. By mastering the use of regular expressions, users can create more efficient and flexible text manipulation operations that can handle a wide range of text processing needs.

UNDERSTANDING REGULAR EXPRESSIONS

- Regular Expressions are text patterns that are used by tools like grep and others
- Always put your regex between single quotes!
- Don't confuse regular expressions with globbing (shell wildcards)!
- They look like file globbing, but they are not the same
  - Grep ‘a*’ a*
- For use with specific tools only (grep, vim, awk, sed)
- See man 7 regex for details

- Basic regular expressions work with most tools
- Extended regular expressions don't always work. Use grep -E if it is an extended regular expression
- Some scripting languages (like perl) come with their own regular expressions

EXAMPLES

```
Hello, this is a sample text file.
It contains some text that can be searched and manipulated using regular expressior
1234 is an example of a number that can be matched using regular expressions.
The string "foo" can be replaced with the string "bar" using regular expressions.
```

Hello, this is a file.txt file.
It contains some text that can be searched and manipulated using regular expressions. 1234 is an example of a number that can be matched using regular expressions. The string "foo" can be replaced with the string "bar" using regular expressions.

Matching a specific string: The following regular expression matches the string "hello" in a text file.

- grep "hello" file.txt

Matching a pattern of characters: The following regular expression matches any sequence of digits in a text file.

- grep "[0-9]+" file.txt

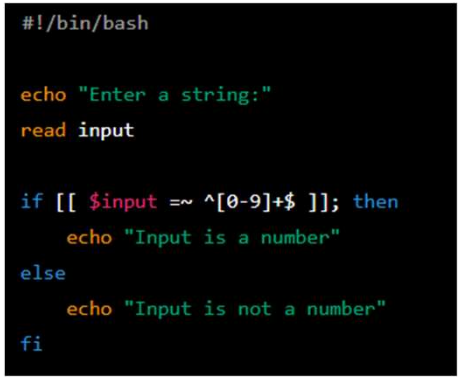
Matching a range of characters: The following regular expression matches any sequence of lowercase letters in a text file.

- grep "[a-z]+" file.txt

Matching multiple patterns: The following regular expression matches any line that contains either "hello" or "world" in a text file.

- grep "hello\|world" file.txt

- Using regular expressions in a script: The following script prompts the user to enter a string and then checks whether the string matches a specific regular expression pattern.



```bash
#!/bin/bash

echo "Enter a string:"
read input

if [[ $input =~ ^[0-9]+$ ]]; then
    echo "Input is a number"
else
    echo "Input is not a number"
fi
```

- Replacing text using regular expressions: The following command replaces all occurrences of the string "foo" with the string "bar" in a text file.
- sed 's/foo/bar/g' file.txt
- These are just a few examples of the many ways that regular expressions can be used in Bash scripting and other programming languages. By mastering the use of regular expressions, users can create more powerful and flexible text manipulation operations that can handle a wide range of text processing needs.

MORE EXAMPLES

- **^** beginning of the line: **grep '^l' myfile**
- **$** end of the line: **grep 'anna$' myfile**
- **\b** end of word: **grep '^lea\b' myfile** will find lines starting with lea, but not with leanne
- * zero or more times: **grep 'n.*x' myfile**
- **+** one or more times (extended regex!): **grep -E 'bi+t' myfile**
- **?** zero or one time (extended regex!): **grep -E 'bi?t' myfile**
- **n\{3\}** *n* occurs 3 times: **grep 'bon\{3\}nen' myfile**
- **.** one character: **grep '^.$' myfile**

USING CUT AND SORT

- **cut** allows you to filter out fields, based on a field separator: **cut -d : -f 1 /etc/passwd**
  - Use **awk** for more advanced options
- **sort** allows you to sort items. Without any options, sort will happen based on the order of characters in the ASCII table
  - Use **sort -n** for numeric sort
  - Use **sort -d** for dictionary order
  - Many other useful options are available, consult the man page

  - Commands like **cut**, and many more, work based on the Internal Field Separator (IFS)
  - IFS by default is a space
  - Many commands have options to define the IFS that should be used

cut command is used to extract specific columns from a text file. The basic syntax is as follows:

- cut -f [fields] [input_file]

where [fields] is a comma-separated list of field numbers or ranges to extract, and [input_file] is the name of the file from which to extract the fields.

For example, suppose we have a file called data.txt that contains the following data:

- 1,John,Doe,42
- 2,Jane,Smith,35
- 3,Bob,Johnson,27

To extract the first and last name columns (columns 2 and 3), we can use the following command:

- cut -f 2,3 -d ',' data.txt

The -d option specifies the delimiter used in the input file (in this case, a comma).

The output of the above command will be:

- John,Doe
- Jane,Smith
- Bob,Johnson

sort command is used to sort the contents of a text file. The basic syntax is as follows:

- sort [options] [input_file]

where [options] is a list of sorting options and [input_file] is the name of the file to be sorted.

For example, suppose we have a file called names.txt that contains the following data:

- John
- Jane
- Bob
- Alice

To sort the names in alphabetical order, we can use the following command:

- sort names.txt

The output of the above command will be:

- Alice
- Bob
- Jane
- John

To sort the names in reverse alphabetical order, we can use the -r option:

- sort -r names.txt

The output of the above command will be:

- John
- Jane
- Bob
- Alice

To sort the names in numerical order (if the file contains numbers), we can use the -n option:

- sort -n numbers.txt

USING TAIL AND HEAD

head command is used to display the first few lines of a text file. The basic syntax is as follows:

- head [options] [input_file]

where [options] is a list of options that modify the behavior of the command (e.g., how many lines to display), and [input_file] is the name of the file to be displayed.

For example, suppose we have a file called data.txt that contains the following data:

- 1,John,Doe,42
- 2,Jane,Smith,35
- 3,Bob,Johnson,27
- 4,Alice,Williams,51
- 5,Michael,Davis,45

To display the first three lines of the file, we can use the following command:

- head -n 3 data.txt

The output of the above command will be:

- 1,John,Doe,42
- 2,Jane,Smith,35
- 3,Bob,Johnson,27

tail command is used to display the last few lines of a text file. The basic syntax is as follows:

- tail [options] [input_file]

where [options] is a list of options that modify the behavior of the command (e.g., how many lines to display), and [input_file] is the name of the file to be displayed.

For example, suppose we have a file called data.txt that contains the following data:

- 1,John,Doe,42
- 2,Jane,Smith,35
- 3,Bob,Johnson,27
- 4,Alice,Williams,51
- 5,Michael,Davis,45

To display the last three lines of the file, we can use the following command:

- tail -n 3 data.txt

The output of the above command will be:

- 3,Bob,Johnson,27
- 4,Alice,Williams,51
- 5,Michael,Davis,45

To continuously display the last few lines of a file as they are added to it (useful for monitoring log files), we can use the -f option:

- tail -f [input_file]
- where [input_file] is the name of the file to be monitored.

USING SED

sed (short for stream editor) is a powerful text manipulation tool in Linux. It is used to perform basic text transformations on an input stream (a file or input from a pipeline).

The basic syntax for using sed is as follows:

- sed 's/pattern/replacement/g' [input_file]

where s is a command that tells sed to substitute one pattern with another, pattern is the regular expression to be replaced, replacement is the text to replace the pattern with, and g stands for global and tells sed to replace all occurrences of the pattern in the line. [input_file] is the name of the file to be processed.

For example, suppose we have a file called data.txt that contains the following data:

- John,Doe,42
- Jane,Smith,35
- Bob,Johnson,27

To replace all occurrences of the comma with a semicolon, we can use the following command:

- sed 's/,/;/g' data.txt

The output of the above command will be:

- John;Doe;42
- Jane;Smith;35
- Bob;Johnson;27

To replace the first occurrence of a pattern in each line, we can use the following command:

- sed 's/,/;/' data.txt

The output of the above command will be:

- John;Doe,42
- Jane;Smith,35
- Bob;Johnson,27

To delete a line from the file, we can use the following command:

- sed '/Jane/d' data.txt

The output of the above command will be:

- John,Doe,42
- Bob,Johnson,27

In the above example, the d command tells sed to delete the line that contains the pattern Jane.

To append a line after a specific line, we can use the following command:

- sed '/Jane/a\
- New line added' data.txt

The output of the above command will be:

- John,Doe,42
- Jane,Smith,35
- New line added
- Bob,Johnson,27

In the above example, the a command tells sed to append the text New line added after the line that contains the pattern Jane.

USING AWK

awk is a powerful text processing tool in Linux. It is used to manipulate and analyze text files, especially when the data is organized in fields or columns. awk reads each line of a file, divides it into fields, and then performs actions based on the content of the fields.

The basic syntax for using awk is as follows:

- awk 'pattern {action}' [input_file]

where pattern is a regular expression that defines the records to process, action is a series of commands to perform on the records that match the pattern, and [input_file] is the name of the file to be processed.

For example, suppose we have a file called data.txt that contains the following data:

- John,Doe,42
- Jane,Smith,35
- Bob,Johnson,27

To print the first and last names (fields 1 and 2) from each line of the file, we can use the following command:

- awk -F, '{print $1,$2}' data.txt

The output of the above command will be:

- John Doe
- Jane Smith
- Bob Johnson

In the above example, the -F option specifies the field separator (a comma in this case), and the print command tells awk to print the first and second fields of each record.

To perform arithmetic operations on the data, we can use the following command to compute the average age:

- awk -F, '{sum+=$3} END {print "Average age:",sum/NR}' data.txt

The output of the above command will be:

- Average age: 34.6667

In the above example, the sum+=$3 command adds the third field (age) to the variable sum for each record, and the END keyword tells awk to perform the following command after all records have been processed. The NR variable contains the total number of records, so sum/NR computes the average age.

To filter the data based on a condition, we can use the following command to print the names of people who are older than 30:

- awk -F, '$3 > 30 {print $1,$2}' data.txt

The output of the above command will be:

- John Doe
- Jane Smith
- Alice Williams
- Michael Davis

In the above example, the $3 > 30 condition tells awk to process only the records where the third field (age) is greater than 30, and the print command tells awk to print the first and second fields of each matching record.

CONSIDERING EXTERNAL TOOLS AND PERFORMANCE

- Best practice: try to avoid using external tools
  - External tools and all the libraries they require need to be fetched from disk, which is relatively slow
  - Also, external tools are operating system specific, sometimes even distribution specific, which makes your script less portable
- Use **type** to find out if a tool is external or not
- To measure performance, use **time myscript** to measure the amount of time it takes to run a script with or without external tools

```
student@student-virtual-machine:~$ time ls
a.txt  b.txt  c.txt  Desktop  Documents  Downloads  files  helloworld.sh  Music  Pictures  Public  sometext  Templates  Videos

real    0m0.001s
user    0m0.001s
sys     0m0.000s
student@student-virtual-machine:~$ which ls
/usr/bin/ls
student@student-virtual-machine:~$
```

```
student@student-virtual-machine:~$ which time
/usr/bin/time
student@student-virtual-machine:~$ /usr/bin/time ls
a.txt  b.txt  c.txt  Desktop  Documents  Downloads  files  helloworld.sh  Music  Pictures  Public  sometext  Templates  Videos
0.00user 0.00system 0:00.00elapsed 100%CPU (0avgtext+0avgdata 2572maxresident)k
0inputs+0outputs (0major+113minor)pagefaults 0swaps
student@student-virtual-machine:~$
```

```
student@student-virtual-machine:~$ type time
time is a shell keyword
student@student-virtual-machine:~$ type cd
cd is a shell builtin
student@student-virtual-machine:~$ time awk -F : '{ print $1 }' /etc/passwd
```

```
student@student-virtual-machine:~$ time cut -d : -f 1 /etc/passwd
```

DO IT YOURSELF

- Use the appropriate commands to create a list of all users on your Linux system that have a UID higher than 1000. The list needs to meet the following criteria:
  - The full lines from /etc/passwd are printed
  - The list is alphabetically sorted by username
  - Any occurrence of /bin/bash is replaced with /bin/zsh
  - Changes are not written to /etc/passwd but to /tmp/myusers

SOLUTION

```
~$ cat /etc/passwd
```

```
student@student-virtual-machine:~$ awk -F : '$3 > 999 { print $0 }' /etc/passwd
nobody:x:65534:65534:nobody:/nonexistent:/usr/sbin/nologin
student:x:1000:1000:student,,,:/home/student:/bin/bash
anna:x:1001:1001::/home/anna:/bin/sh
linda:x:1002:1002::/home/linda:/bin/sh
student@student-virtual-machine:~$ awk -F : '$3 > 999 { print $0 }' /etc/passwd | sort
anna:x:1001:1001::/home/anna:/bin/sh
linda:x:1002:1002::/home/linda:/bin/sh
nobody:x:65534:65534:nobody:/nonexistent:/usr/sbin/nologin
student:x:1000:1000:student,,,:/home/student:/bin/bash
student@student-virtual-machine:~$ awk -F : '$3 > 999 { print $0 }' /etc/passwd | sort | sed 's/old/new/'
```

```
~$ awk -F : '$3 > 999 { print $0 }' /etc/passwd | sort | sed 's/\/bin\/sh/\/bin\/zsh/'
```

```
~$ echo $SHELL
```

```
student@student-virtual-machine:~$ ls -l $(which sh) $(which bash)
-rwxr-xr-x 1 root root 1183448 Feb 25  2020 /usr/bin/bash
lrwxrwxrwx 1 root root       4 Jul 13 08:23 /usr/bin/sh -> dash
student@student-virtual-machine:~$ ls -l $(which dash) $(which bash)
-rwxr-xr-x 1 root root 1183448 Feb 25  2020 /usr/bin/bash
-rwxr-xr-x 1 root root  129816 Jul 18  2019 /usr/bin/dash
student@student-virtual-machine:~$
```

SHELL SCRIPTING FUNDAMENTALS

- Creating your first Shell Scripting
- Working with Variables and Arguments
- Transforming Input

Creating your first Shell Scripting

- Choosing an editor
- Shell Scripts and IDEs
- Core Bash script ingredients
- Running the scripts
- Finding help about scripting components
- Lab : writing your first script
- Lab Solution : writing your first script

CHOOSING AN EDITOR

- To write shell scripts, you need an editor
- This editor should offer syntax highlighting
- Any common editor offers syntax highlighting for shell scripting
- Common Linux editors are **vim**, **nano** and **gedit**

SHELL SCRIPTS AND IDES

- An Integrated Development Environment (IDE) provides comprehensive tools for programmers to develop software
- IDEs should at least contain the following:
  - Source code editor
  - Build automation tool
  - Debugger
- As shell scripts don't go through the normal cycle of software development, there are no IDEs that have been developed specifically for shell scripting
- Just make sure you're using an editor with decent syntax highlighting

CORE BASH SCRIPT INGREDIENTS

- Best practice: make sure your scripts always include the following
  - Shebang: the indicator of the shell used to run the script code on the first line of the script:
    - **#!/bin/bash** or **#!/usr/bin/env bash** to identify Bash as the script interpreter
  - Comment to explain what the script is doing
  - White lines to increase readability
  - Different block of code to easily distinguish between parts of the script
- Script names are arbitrary
- Extensions are not required but may be convenient for scripts users from non-Linux operating systems
- Because of Linux $PATH restrictions, scripts cannot be executed from the current directory
- Consider using **~/bin** to store scripts for personal use
- Consider using **/usr/local/bin** for scripts that should be available for all users

```
~$ sudo apt install git
~$ cd bash-scripting/
$ vim hello-world
~$ sudo apt install git
```

```
#!/bin/bash
#
# This is some comment
# Here you can put how to use the script

clear
echo hello world

exit 0
~
~
~
~
~
~
```

```
student@student-virtual-machine:~/bash-scripting$ chmod +x hello-world
student@student-virtual-machine:~/bash-scripting$ ls -l hello-world
-rwxrwxr-x 1 student student 110 Jul 17 08:23 hello-world
student@student-virtual-machine:~/bash-scripting$ ./hello-world
student@student-virtual-machine:~/bash-scripting$ pwd
/home/student/bash-scripting
student@student-virtual-machine:~/bash-scripting$ echo $PATH
/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games:/usr/local/games:/snap/bin
student@student-virtual-machine:~/bash-scripting$ echo $?
0
student@student-virtual-machine:~/bash-scripting$

student@student-virtual-machine:~/bash-scripting$ echo $?
5
student@student-virtual-machine:~/bash-scripting$
```

RUNNING THE SCRIPTS

- To run a script as a separate program, you need to set the execute permission: **chmod +x myscript**
- If the directory containing the script is not in the $PATH, run it using **./myscript**
- Scripts can also be started as an argument to the shell, in which case no execute permission and $PATH is needed: **bash myscript**
- Best practice: to avoid confusion, include a shebang on the first line of the script and put it in the $PATH so that it can run as an individual program

FINDING HELP ABOUT SCRIPTING COMPONENTS

- **help** provides an overview of Bash internal commands, which are the core components of any shell script
- **man bash** offers more detailed information about these commands
- Online resources offer additional information and examples: the Advanced Bash Scripting Guide (https://tldp.org/LDP/abs/html/) has been the key reference for many years

```
student@student-virtual-machine:~/bash-scripting$ help
student@student-virtual-machine:~/bash-scripting$ man bash
```

DO IT YOUR SELF
WRITING YOUR FIRST SCRIPT

- Write a script that clears your screen and prints the message "hello world".
- Make sure this script meets all recommended syntax requirements.
- Also make sure the script is in your computers $PATH in such a way that any user can execute it.

SOLUTION
WRITING YOUR FIRST SCRIPT

```
virtual-machine:~/bash-scripting$ vim myfirstscript

#!/bin/bash
# lab solution for scripting course

clear
printf "%s\n" "hello world"
~
~/bash-scripting$ vim myfirstscript
~/bash-scripting$ chmod +x myfirstscript
~/bash-scripting$ ./myfirstscript
```

```
student@student-virtual-machine:~/bash-scripting$ echo $PATH
/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games:/usr/local/games:/snap/bin
student@student-virtual-machine:~/bash-scripting$ cp myfirstscript /usr/local/bin
cp: cannot create regular file '/usr/local/bin/myfirstscript': Permission denied
student@student-virtual-machine:~/bash-scripting$ ls -ld /usr/local/bin
drwxr-xr-x 2 root root 4096 Apr 23  2020 /usr/local/bin
student@student-virtual-machine:~/bash-scripting$
```

```
~/bash-scripting$ whoami

~/bash-scripting$ sudo cp myfirstscript /usr/local/bin

~/bash-scripting$

~/bash-scripting$ id
```

WORKING WITH VARIABLES AND ARGUMENTS

- About Terminology
- Quoting
- Defining and Using Variables
- Defining Variables with read
- Viewing Variables
- Handling Script Arguments
- Using shift
- Using command substitution
- Using here Documents
- Using Functions
- Do It yourself : Working with Variables and Arguments

ABOUT TERMINOLOGY

- An *argument* is anything that can be put behind the name of a command or script
  - **ls -l /etc** has 2 arguments
- An *option* is an argument that changes the behavior of the command or script, and its functionality is programmed into the command
  - In **ls -l /etc**, **-l** is used as an option
- A *positional parameter* is another word for an argument
- A *variable* is a key with a name that can refer to a specific value
- While being handled, all are further treated as *variables*

QUOTING

- Escaping is the solution to take away special meaning from characters
- To apply escaping, use quotes
- Double quotes are used to avoid interpretation of spaces
  - **echo "my value"**
- Single quotes are used to avoid interpretation of anything
  - **echo the current '$SHELL' is $SHELL**
- Backslash is used to avoid interpretation of the next character
  - **echo the current \$SHELL is $SHELL**

```
student@student-virtual-machine:~/bash-scripting$ echo 'hello world'
hello world
student@student-virtual-machine:~/bash-scripting$ echo "hello world"
hello world
student@student-virtual-machine:~/bash-scripting$ echo "hello the value of $SHELL is $SHELL"
hello the value of /bin/bash is /bin/bash
student@student-virtual-machine:~/bash-scripting$ echo "hello the value of \$SHELL is $SHELL"
hello the value of $SHELL is /bin/bash
student@student-virtual-machine:~/bash-scripting$
```

```
student@student-virtual-machine:~/bash-scripting$ echo the value of '$SHELL' is $SHELL
the value of $SHELL is /bin/bash
student@student-virtual-machine:~/bash-scripting$ echo 'the value of '$SHELL' is $SHELL'
the value of /bin/bash is $SHELL
student@student-virtual-machine:~/bash-scripting$ echo "the value of '$SHELL' is $SHELL"
the value of '/bin/bash' is /bin/bash
student@student-virtual-machine:~/bash-scripting$
```

DEFINING AND USING VARIABLES

- A local variable works in the current shell only
- An environment variable is an operating system setting that is set while booting
- Arrays are special multi-valued variables – we will cover more on this later

- Bash variables don't use data types
- Variables can contain a number, character or a string of characters
- **declare** can be used to set specific variable attributes
  - **declare -r ANSWER=yes** sets $ANSWER as a read-only variable
  - **declare [-a|-A] MYARRAY** is used to define an indexed or associative array (covered later)
- Using **declare** is NOT required to set a variable, but may be required to set an array
- Use **declare -p** to find out which type of variable something is:
  - compare **declare -p GROUPS** with **declare -p PATH**

DEFINING AND USING VARIABLES

```
student@student-virtual-machine:~/bash-scripting$ declare -r ANSWER=no
student@student-virtual-machine:~/bash-scripting$ echo $ANSWER
no
student@student-virtual-machine:~/bash-scripting$ ANSWER=yes
bash: ANSWER: readonly variable
student@student-virtual-machine:~/bash-scripting$ declare -p
```


```
student@student-virtual-machine:~/bash-scripting$ declare -r ANSWER=no
student@student-virtual-machine:~/bash-scripting$ echo $ANSWER
no
student@student-virtual-machine:~/bash-scripting$ ANSWER=yes
bash: ANSWER: readonly variable
student@student-virtual-machine:~/bash-scripting$ declare -p
```


```
student@student-virtual-machine:~/bash-scripting$ help declare
```

DEFINING VARIABLES

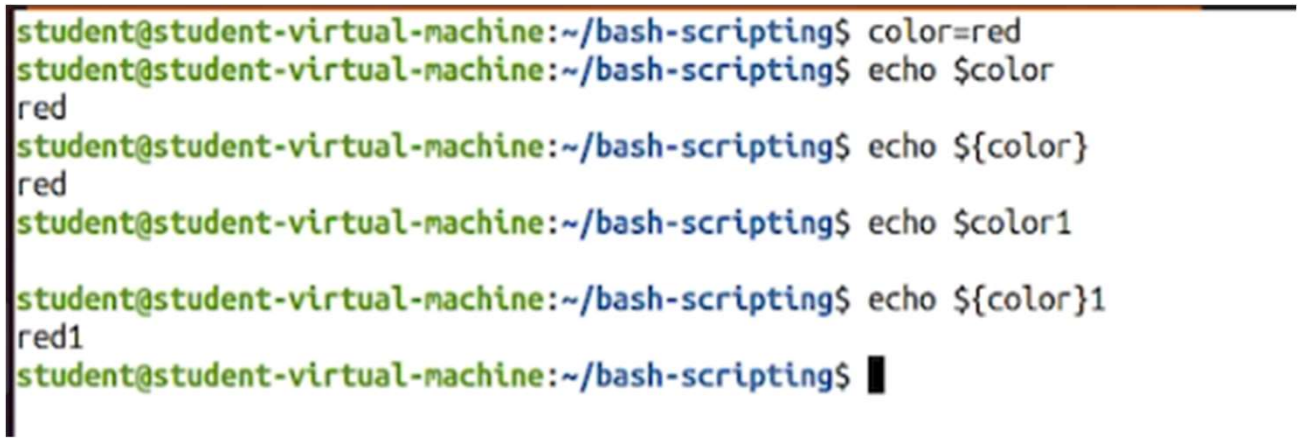
- Defining a variable can be easy: **key=value**
- Variables are not case sensitive
  - Environment variables, by default, are written in uppercase
  - Local variables can be written in any case
- After defining, a variable is available in the current shell only
- To make variables available in subshell also, use **export key=value**
- To clear variable contents, use **key=**
- Use **env** to get access to environment variables

```
student@student-virtual-machine:~/bash-scripting$ color=red
student@student-virtual-machine:~/bash-scripting$ echo $color
red
student@student-virtual-machine:~/bash-scripting$ bash
student@student-virtual-machine:~/bash-scripting$ echo $color

student@student-virtual-machine:~/bash-scripting$ exit
exit
student@student-virtual-machine:~/bash-scripting$ color=
student@student-virtual-machine:~/bash-scripting$ echo $color

student@student-virtual-machine:~/bash-scripting$
```

```
student@student-virtual-machine:~/bash-scripting$ echo $color

student@student-virtual-machine:~/bash-scripting$ unset color
student@student-virtual-machine:~/bash-scripting$ export color=blue
student@student-virtual-machine:~/bash-scripting$ echo $color
blue
student@student-virtual-machine:~/bash-scripting$ bash
student@student-virtual-machine:~/bash-scripting$ echo $color
blue
student@student-virtual-machine:~/bash-scripting$ env
```

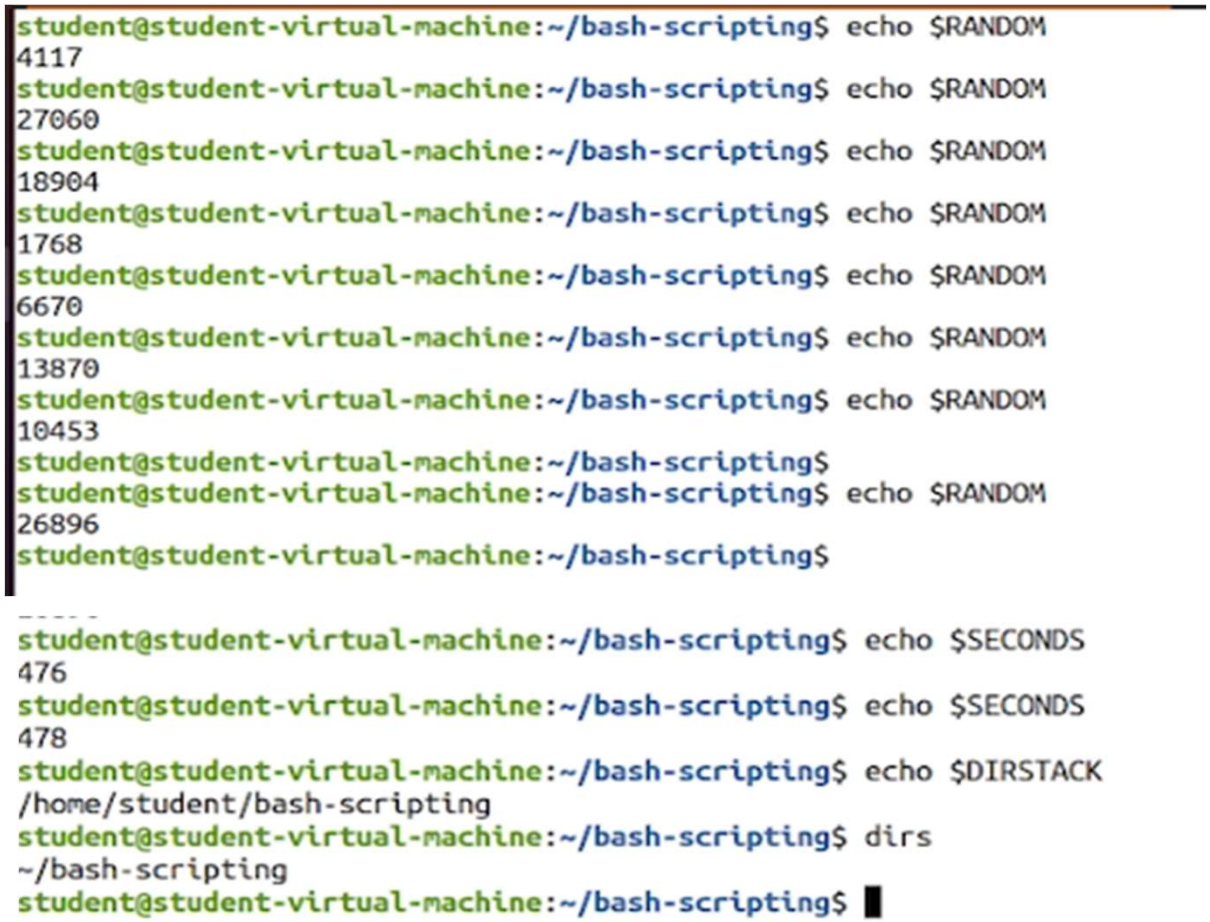
USING VARIABLES

- To use the current value assigned to a variable, put a $ in front of the variable name
- To better deal with variables, it is recommended, though not mandatory, to put the variable name between { }
  - **color=red**
  - **echo $color**
  - **echo ${color}**
- To avoid confusion on specific types of variables, it is recommended, though not mandatory, to put the variable name between double quotes
  - **echo "${color}"**


student@student-virtual-machine:~/bash-scripting$ color=red
student@student-virtual-machine:~/bash-scripting$ echo $color
red
student@student-virtual-machine:~/bash-scripting$ echo ${color}
red
student@student-virtual-machine:~/bash-scripting$ echo $color1
student@student-virtual-machine:~/bash-scripting$ echo ${color}1
red1
student@student-virtual-machine:~/bash-scripting$

SPECIAL VARIABLES

- Bash works with some special variables, that have an automatically assigned value
  - $RANDOM: a random number
  - $SECONDS: the number of seconds this shell has been running
  - $LINENO: the line in the current script
  - $HISTCMD: the number of this command in history
  - $GROUPS: an array that holds the names of groups that the current user is a member of
    - (compare **echo $GROUPS** versus **echo "${GROUPS[@]}"**
  - $DIRSTACK: list of recently visited directories, also use **dirs** to display
- Apart from these, there are standard variables such as $BASH_ENV, $BASHOPTS and more


student@student-virtual-machine:~/bash-scripting$ echo $RANDOM
4117
student@student-virtual-machine:~/bash-scripting$ echo $RANDOM
27060
student@student-virtual-machine:~/bash-scripting$ echo $RANDOM
18904
student@student-virtual-machine:~/bash-scripting$ echo $RANDOM
1768
student@student-virtual-machine:~/bash-scripting$ echo $RANDOM
6670
student@student-virtual-machine:~/bash-scripting$ echo $RANDOM
13870
student@student-virtual-machine:~/bash-scripting$ echo $RANDOM
10453
student@student-virtual-machine:~/bash-scripting$
student@student-virtual-machine:~/bash-scripting$ echo $RANDOM
26896
student@student-virtual-machine:~/bash-scripting$
student@student-virtual-machine:~/bash-scripting$ echo $SECONDS
476
student@student-virtual-machine:~/bash-scripting$ echo $SECONDS
478
student@student-virtual-machine:~/bash-scripting$ echo $DIRSTACK
/home/student/bash-scripting
student@student-virtual-machine:~/bash-scripting$ dirs
~/bash-scripting
student@student-virtual-machine:~/bash-scripting$

```
student@student-virtual-machine:~/bash-scripting$ echo $BASH_ENV

student@student-virtual-machine:~/bash-scripting$ echo $BASHOPT

student@student-virtual-machine:~/bash-scripting$
```

```
student@student-virtual-machine:~/bash-scripting$ vim simplevar
```

student@student-virtual-machine: ~/bash-scripting

```
#!/bin/bash

COLOR=red
TREE=oak

echo the $TREE is $COLOR
~
~
~
```

```
student@student-virtual-machine:~/bash-scripting$ vim simplevar
student@student-virtual-machine:~/bash-scripting$ chmod +x simplevar
student@student-virtual-machine:~/bash-scripting$ ./simplevar
the oak is red
student@student-virtual-machine:~/bash-scripting$
```

DEFINING VARIABLES WITH READ

- When **read** is used, shell script execution will stop to read user input
- The user input is stored in the variable that is provided as an argument to **read**

```
echo enter a value
read value
echo you have entered $value
```

- If no variable name is set, the **read** result is stored in $REPLY
- **read** can also be used without further arguments
- This can be useful for "Press Enter to continue" structures

```
echo press enter to continue
read
echo continuing...
```

DEFINING VARIABLES WITH READ

- **read** can be used to define more than one variable at the same time

```
#!/bin/bash
echo enter firstname, lastname and city
read firstname lastname city
echo nice to meet you $firstname $lastname from $city
```

```
#!/bin/bash
# source

echo which directory do you want to go to?

read DIR

cd ${DIR}
pwd
ls
```

VIEWING VARIABLES

- As variables can be set in different ways, viewing all currently defined variables isn't easy
- **set** shows all current variables, including functions and the values of the variables
- **compgen -v** will show variables only and not their values

HANDLING SCRIPT ARGUMENTS

- Scripts can be started with arguments to provide specific values while executing the script
- *Arguments* are also referred to as *Positional Parameters*
- The first argument is stored in $1, the second argument is stored in $2 and so on
- The script name itself is stored as $0
- By default, a maximum of nine arguments can be defined this way
- When using curly braces, more than nine arguments can be provided

HANDLING SCRIPT ARGUMENTS

- Script arguments can be addressed individually
- To address all arguments, use **$@** or **$***
- Without quotes, **$@** and **$*** are identical
- With quotes, **$@** expands to properly quoted arguments, and **$*** makes all arguments into a single argument

```
#!/bin/bash
#
# ...

echo "Hello $1, how are you today"
echo " hello $2, how are you"
echo " hello $10, how are you"
echo " hello ${10}"
echo " hello ${11}"
shift
echo hi $1
echo "\$0 is $0"
```

USING SHIFT

- **shift** is used to shift the positional parameters to the left
- **shift** can take a number as its argument to shift the positional parameters to the left by that number
- Using **shift** can be useful to remove arguments after processing them
- As an alternative, consider looping over all arguments using **while**

USING COMMAND SUBSTITUTION

- Command substitution is used to work with the result of a command instead of a static value that is provided
- Use this to refer to values that change frequently:
  - **today=$(date +%d-%m-%y)**
  - **mykernel=$(uname -r)**
- Command substitution can be done in two ways that are not fundamentally different:
  - **today=$(date +%d-%m-%y)**
  - **today=`date +%d-%m-%y`**

```
student@student-virtual-machine:~/bash-scripting$ date +%d-%m-%Y
18-07-2021
```

USING HERE DOCUMENTS

- A *here document* is used as I/O redirection to feed a command list to an interactive program or command
- A here document is used as a scripted alternative that can be provided by input redirection
- Here documents are useful, as all the code that needs to be processed is a part of the script, and there is no need to process any external commands

USING FUNCTIONS

- A function is a small block of reusable code that can be called from the script by referring to its name
- Using functions is convenient when blocks of code are needed repeatedly
- Functions can be defined in two ways:
  - ```
    function_name () {
        commands
    }
    ```
  - ```
    function function_name {
        commands
    }
    ```
- Functions are local to the script where they are executed
- Bash-wide fuctions are initialized from the Bash startup scripts
- Use **set** to show a list of all functions currently available
- Remove a function by using **unset**

USING FUNCTIONS

- Functions can work with arguments, which have a local scope within the function

```
#!/bin/bash
hello () {
  echo hello $1
}
hello bob
```

- Function arguments are not affected by passing positional parameters to a script while executing it

DO IT BY YOURSELF

- Write a script that allows you to install and start any service. The name of the service should be provided as an argument while starting the script.

TRANSFORMING INPUT

- Working with Parameter Substitution
- Calculating
- Using tr

WORKING WITH PARAMETER SUBSTITUTION

- Parameter substitution can be used to deal with missing parameters
- Use it to set a default, or to display a message in case missing parameters were found
- If parameter is not set, use default (does NOT set it)
  - **echo ${username:-$(whoami)}** uses result of whoami if $username is not set
  - **filename=${1:-$DEFAULT_FILENAME}** uses value of $DEFAULT_FILENAME if $filename is not set
- If parameter is not set, set to default
  - **echo ${username:=$(whoami)}** uses result of whoami if $username is not set
  - **filename=${1:=$DEFAULT_FILENAME}** uses value of $DEFAULT_FILENAME if $filename is not set
- If parameter is not set, print error_msg and exit script with exit status 1
  - **echo ${myvar:?error_msg}**

CALCULATING

- Bash offers different solutions for calculation with integers:
  - **let expression**
  - **expr expression**
  - **$((expression))**
- Of these 3, the $((...)) method is preferred
- All work with the following operators
  - +
  - -
  - *
  - /
  - %

CALCULATING

- **let** is a bash internal

  ```
  let a=1+2
  echo $a
  let a++
  echo $a
  ```

- **expr** is an external command that can be used in scripts, but it is not used much anymore
- **$((...))** allows you to put the calculation between parenthesis

  ```
  echo $(( 2 * 3 ))
  ```

ADVANCED CALCULATION TOOLS:

- **bc** is an advanced calculation tool that allows you to work with decimals
  - **bc** is typically used in pipes: **echo "12/5" | bc**
  - to print decimals in the result, use **-l: echo "12/5" | bc -l**
  - **bc** also offers access to built-in mathematical functions:
    echo "sqrt(1000)" | bc -l
- **factor** decomposes an integer into prime factors
  - **factor 399**

USING TR

- **tr** is an external utility that allows for translation of characters
  - **echo hello | tr [a-z] [A-Z]**
  - **echo hello | tr [:lower:] [:upper:]**
  - **echo hello | tr [a-m] [n-z]**
  - **echo how are you | tr [:space:] '\t'**

- Bash variables can modify case using ^^ (uppercase) and ,,(lowercase)
  - **color=red; echo ${color^^}**
  - **color=BLUE; echo ${color,,}**

USING CONDITIONAL STATEMENTS

- Using if and if..then..else
- Using Conditional and Loops

USING IF AND IF..THEN..ELSE

- Using Test
- Using Simple if Statements
- Using Logical Tests
- Testing with [[]]
- Using if...then...else
- Using if...then...else with elif

USING TEST

- **test** is the foundation of many **if** statements
- **test** is an external command that allows you to perform different types of test
  - Expression tests: test can evaluate the binary outcome of an expression, which is a logical test by itself
  - String tests: allow you to evaluate if a string is present or absent, and compare one string to another
  - Integer tests: allow you to compare integers, which includes operations like bigger than, smaller than
  - File tests: allow you to test all kind of properties of files
- **test** is typically used in conditional statements
- **test *condition*** can also be written as **[*condition*]**

```
student@student-virtual-machine:~/bash-scripting$ type [
[ is a shell builtin
student@student-virtual-machine:~/bash-scripting$ type test
test is a shell builtin
student@student-virtual-machine:~/bash-scripting$ which test
/usr/bin/test
student@student-virtual-machine:~/bash-scripting$ help test
```

USING TEST

```
student@student-virtual-machine:~/bash-scripting$ test -f /etc/hosts
student@student-virtual-machine:~/bash-scripting$ echo $?
0
student@student-virtual-machine:~/bash-scripting$ test -f /etc/khkhk
student@student-virtual-machine:~/bash-scripting$ echo $?
1
student@student-virtual-machine:~/bash-scripting$ test -f /usr/bin/passwd
student@student-virtual-machine:~/bash-scripting$ echo $?
0
student@student-virtual-machine:~/bash-scripting$ test -f /dev/sda
student@student-virtual-machine:~/bash-scripting$ echo $?
1
student@student-virtual-machine:~/bash-scripting$ [ -f /etc/hosts ]
student@student-virtual-machine:~/bash-scripting$ [ -f /etc/hjkgkjg ]
student@student-virtual-machine:~/bash-scripting$ echo $?
1
student@student-virtual-machine:~/bash-scripting$
```

USING SIMPLE IF STATEMENTS

- **if** is used to verify that a condition is true

```
if true
then
        echo command executed successfully
fi
```

- The condition is a command that returned an exit code 0, or a test that completed successfully:

```
if [ -f /etc/hosts ]; then echo file exists; fi
```

USING LOGICAL TESTS

- Logical tests are **if then else** tests written in a different way
- Logical AND: the structure is **a && b**, which results in **b** being executed when **a** is successful
- Logical OR: the structure is **a || b**, which results in **b** being executed only if **a** is not successful
- Logical operators can be embedded in **if** statements to test multiple conditions
  - **if [-d $1] && [-x $1]; then echo $1 is a directory and has execute; fi**
  - **if [-f /etc/hosts]**
    then
    echo file exists
    fi
  - **[-f /etc/hosts] && file exists**

TESTING WITH [[]]

- A regular test is written as **[*condition*]**
- The enhanced version is written as **[[*condition*]]**
- **[[*condition*]]** is a Bash internal, and offers features not offered by **test**
- Because it is a Bash internal, you may not find it in other shells
- Because of the lack of compatibility, many scripters prefer using **test** instead
- **[[$VAR1 = yes && $VAR2 = red]]** is using a conditional statement within the test
- **[[1 < 2]]** tests if 1 is smaller than 2
- **[[-e $b]]** will test is $b exists. If $b is a file that contains spaces, using [[ ]] won't require you to use quotes
- **[[$var = img* && ($var = *.png || $var = *.jpg)]] && echo $var starts with img and ends with .jpg or .png**

USING IF...THEN...ELSE

- **else** can be used as an extension to **if** statements to perform an action if the first condition is not true
- Notice that instead of using **else**, independent statements can be used in some cases
- See the differences between **else1** and **else2** in the course Git repo

USING IF...THEN...ELSE WITH ELIF

- **elif** can be added to the **if ... then ... else** statement to add a second condition

```
if [ -d $1 ]; then
        echo $1 is a directory
elif [ -f $1 ]; then
        echo $1 is a file
else
        echo $1 is an unknown entity
fi
```

DO IT YOUR SELF

- Write a script that will alert if available disk space on the / partition or volume is less than 3 GB, or if less than 512 MB RAM is listed as free.
- If both of these conditions are true, the script should print: WARNING: low system resource.
- If only low disk space is available, the script should print: WARNING: low disk space available
- If only low memory is available, the script should print WARNING: low memory available.
- If neither of these low conditions is the case, the script should print the message: all is good.

USING CONDITIONALS AND LOOPS

- Applying Conditionals and Loops
- Using for
- Using Case
- Using while and until
- Using Break and Continue

APPLYING CONDITIONALS AND LOOPS

- **if ... then ... else** is used to execute commands if a specific condition is true
  - **if [-z $1]; then echo hello; fi**
- **for** is used to execute a command on a range of items
  - **for i in "$@"; do echo $i; done**
- **while** is used to run a command as long as a condition is true
  - **while true; do true; done**
- **until** is used to run a command as long as a condition is not true
  - **until who | grep $1; do echo $1 is not logged in; done**
- **case** is used to run a command if a specific situation is true

USING FOR

- **for** is used to iterate over a range of items
- This is useful for handling a range of arguments, or a series of files

USING CASE

- **case** is used to check a specific number of cases
- It is useful to provide scripts that work with certain specific arguments
- It has been used a lot in legacy Linux init scripts
- Notice that **case** is case sensitive, consider using **tr** to convert all to a specific case

USING WHILE AND UNTIL

- **while** and **until** are used to run a command based on the exit status of an expression
- **while** will run the command as long as the expression is true
- **until** will run the command as long as the expression is not true

USING BREAK AND CONTINUE

- **break** is used to leave a loop straight away
- Using **break** is useful if an exceptional situation arises
- **continue** is used to stop running through this iteration and begin the next iteration
- Using **continue** is useful if a situation was encountered that makes it impossible to proceed

ADVANCED BASH SCRIPTING OPTIONS

- Using Advanced Scripting Options
- Using Arrays
- Script Best Practices, Debugging and Analysing
- Exploring Scripts For DevOps and Linux

USING ADVANCED SCRIPTING OPTIONS

- Working With Options
- Using Variables in Functions
- Defining Menu Interfaces
- Using trap

WORKING WITH OPTIONS

- An *option* is an argument that changes script behavior
- Use **while getopts "ab:" opts** to evaluate options a and b, and while evaluating them, put them in a temporary variable opts
- Next, use **case $opts** to define what should happen if a specific option was encountered

```
#!/bin/bash
while getopts "hs:" arg; do
case $arg in
        h)
                echo "usage"
                ;;
        s)
                strength=$OPTARG
                echo $strength
                ;;
        esac
done
~
```

USING VARIABLES IN FUNCTIONS

- No matter where they are defined, variables always have a global scope - even if defined in a function
- Use the **local** keyword to define variables with a local scope inside of a function
- See **funcvar** for an example

```
#!/bin/bash
var1=A

my_function () {
        local var2=B
        var3=C
        echo "inside function: var1: $var1, var2: $var2, var3: $var3"
}

echo "before runninng function: var1: $var1, var2: $var2, var3: $var3"

my_function

echo "after running function: var1: $var1, var2: $var2, var3: $var3"
```

DEFINING MENU INTERFACES

- The **select** statement can be used to select a menu
- Use **PS3** to define a menu prompt
- The **select** statement embeds a **case** statement
- After executing a menu option, you will get back to the menu
- Use **break** to get out of the menu
- **$REPLY** is a default variable that contains the string that was entered at the prompt
- The select statement can refer to the choices directly. If the choices contain spaces, you'll need to use an array instead

USING TRAP

- **trap** is used to run a command while catching a signal
- **SIGKILL** (**kill -9**) cannot be trapped
- Signals are specific software interrupts that can be sent to a command
- Use **man 7 signal** or **trap -l** for an overview

- **trap** is useful to run commands upon receiving a signal
- Use it to catch signals that you don't want to happen in the script, like INT
- Or use it on EXIT, to define tasks that should happen when the script properly exits
  - This is stronger than just running a command at the end of the script, because that will only run if the script reaches the end

USING ARRAYS

- Understanding why Arrays are useful
- Understanding Array Types
- Using Arrays
- Reading Command output into Arrays- Alternative Approach
- Looping through Arrays

UNDERSTANDING WHY ARRAYS ARE USEFUL

- The string is the default type of parameter
- Integers are parameter types that can be used in calculations
- An array is a parameter that can hold multiple values, stored as key/value pairs, where each value can be addressed individually
- Strings may contain multiple elements, but there is no way that always works to find all these elements
- Particularly, files containing spaces in their names have issues
  - string: **files=$(ls *.doc); cp $files ~/backup**
  - array: **files=(*.txt); cp "${files[@]}" ~/backup**
- To ensure that lists of things that need to be processed always work, use arrays

UNDERSTANDING ARRAY TYPES

- *Index arrays* address a value by using an index number
  - Index arrays are very common
  - The **declare** command does not have to be used to define an index array
- Associative arrays address a value by using a name
  - Associative arrays provide the benefit of using meaningful keys
  - Associative arrays are relatively new
  - While using associative arrays, you must use **declare -A**

USING ARRAYS

- Indexed arrays are the most common, and are defined much like variables, where all elements are put between braces
- In the indexed array, the key is an index value, starting with index value 0
  - my_array=(one two three)
- Don't confuse with command substitution
  - myname=$(whoami)
- Associative arrays exist since Bash 4.0: they use user-defined strings as the key
  - ${value[XLY]}
  - Ordering in associative arrays can not be guaranteed!

USING ARRAYS

- Arrays should always be used with quotes, without quotes you'll lose the array benefits and your script may fail over spaces
- **"${myarray[@]}"** refers to all values in the array
- **"${myarray[1]}"** refers to index value 1 (the second element)
- **"${!myarray[@]}"** refers to all keys in the array

USING DECLARE TO DEFINE ARRAY

- To create an indexed array, you can just start by assigning values to it
  - **my_array=(cow goat)**
- Optionally, you can declare it using **declare -a**
- To create an associative array, you need to use **declare -A**
  - **declare -A my_aaray**
  - **my_aaray=([value1]=cow [value2]=sheep)**

READING COMMAND OUTPUT INTO ARRAYS

- Use **mapfile** to fill an array with the result of a command
  - **mapfile -t my_array < <(my_command)**
- Otherwise, you can use a loop that adds each item in the output item-by-item

```
my_array=()
while IFS= read -r line; do
        my_array+=( "$line" )
done < <( my_command )
```

  - **IFS=** sets the Internal Field Separator to a space for just the **read** command
  - **read -r** tells read not to interpret backslashes as escape characters
  - As long as elements are found in **my_command**, the loop continues

READING COMMAND OUTPUT INTO ARRAYS

- **readarray** is a relatively new solution that allows you to put the result of a command in an array

```
$ readarray -t my_array < <(seq 5)
$ declare -p my_array
declare -a my_array=([0]="1" [1]="2" [2]="3" [3]="4"
[4]="5")
```

SCRIPT BEST PRACTICES, DEBUGGING AND ANALYZING

- Best Practices
- Using Set Options
- Including Debug Information
- Writing Debug Information into a File
- Running bash – X

BEST PRACTICES

- Use a shebang:
- Begin your script with a shebang (#!) to specify the interpreter. For Bash scripts, use:
- #!/bin/bash
- ---
- Add comments:
- Use comments to explain the purpose and functionality of your script.
- # This script demonstrates Bash best practices.
- ---
- Use meaningful variable names:
- Choose variable names that are descriptive and easy to understand.
- user_name="John Doe"
- ---
- Use lowercase for variable names:
- By convention, use lowercase and underscores to separate words in variable names.
- file_count=5

- Use double quotes for variable expansion:
- To avoid issues with spaces and special characters, use double quotes when expanding variables.
- echo "Welcome, ${user_name}!"
- ---
- Use the [[ ]] test construct:
- The [[ ]] construct is more powerful and less error-prone than the [ ] construct.
- if [[-e "$file_path"]]; then
- echo "File exists."
- fi
- ---
- Use (( )) for arithmetic operations:
- Use (( )) for arithmetic expressions and comparisons.
- if ((file_count > 0)); then
- echo "There are files to process."
- fi

- Use functions:
- Encapsulate code in functions to promote modularity and reusability.
- function greet_user() {
- local user_name="$1"
- echo "Hello, ${user_name}!"
- }
- greet_user "John Doe"
- ---
- Use local variables in functions:
- Declare variables as local within functions to prevent global scope pollution.
- function calculate_sum() {
- local num1="$1"
- local num2="$2"
- local sum=$((num1 + num2))
- echo "The sum is: ${sum}"
- }
- ---
- Use command substitution:
- Use $(command) or `command` for command substitution, with the former being the preferred method.
- current_directory=$(pwd)
- echo "Current directory: ${current_directory}"

- Check for command exit status:
- Check the exit status of commands using $? to handle errors and unexpected results.
- cp source_file target_file
- if [[$? -ne 0]]; then
- echo "Error: File copy failed."
- exit 1
- fi
- ---
- Use set -e and set -u:
- Use set -e to exit the script if a command fails and set -u to exit if an unset variable is used.

- #!/bin/bash
- set -e
- set -u
- ---

- Use trap for cleanup tasks:
- Use trap to execute cleanup tasks when the script exits or receives specific signals.
- trap "echo 'Cleaning up...'; rm -f temp_file" EXIT

- Use getopts for option parsing:
- Use getopts to parse command-line options and arguments.
- while getopts ":hvf:" opt; do
- case $opt in
- h) usage ;;
- v) verbose=1 ;;
- f) input_file="$OPTARG" ;;
- \?) echo "Invalid option: -$OPTARG" >&2; usage ;;
- esac
- done
- shift $((OPTIND - 1))
- ---
- Use printf instead of echo:
- printf is more reliable and versatile than echo. It allows for better control over formatting.
- printf "User: %s, Age: %d\n" "$user_name" $user_age

- Use arrays for multiple values:
- Use arrays to store multiple values and iterate over them.
- files=("file1.txt" "file2.txt" "file3.txt")

- for file in "${files[@]}"; do
- echo "Processing: ${file}"
- done
- ---

- Use case statements for multiple choices:
- Use case statements for branching based on a variable's value, which is often more readable than multiple if statements.
- animal="dog"
- case "$animal" in
- "cat")
- echo "This is a cat."
- ;;
- "dog")
- echo "This is a dog."
- ;;
- "bird")
- echo "This is a bird."
- ;;
- *)
- echo "Unknown animal."
- ;;
- esac

USING SET OPTIONS

- Bash **set** can be used in a script to enable/disable specific functionality
  - **set** options can be specified in two ways: using an option, or using **set -o option-name**
  - compare **set -e** and **set -o errexit**
- Use **set -x** to enable an option
- Use **set +x** to disable an option
- Some useful options include:
  - **-e**: exit the script when a command fails
  - **-i**: runs the script in interactive mode
  - **-v**: runs a script in verbose mode
  - **-x**: runs a script in verbose mode while expanding commands
- The advantage of **set** is that you can enable an option before a specific section and disable it again after that section

INCLUDING DEBUG INFORMATION

- As discussed before, options can be used to make a script more verbose
- Alternatively, the scripter can manually include debug information at critical points
- To do so, use **echo** and **read**
  - **echo just added the user, press enter to continue**
  - **read**
- Notice that **read** is used without further argument, as the answer to the **read** prompt is not used anyway
- Use manually inserted debug information while developing, and don't forget to clean up once done
- While testing scripts, consider using **echo** with all commands you want to use
- Using **echo** avoids your script from making modifications to the system, once confirmed that all works as expected, you can clean up the **echo** statements

RUNNING BASH -X

- Instead of using the **set** option, you can use **bash -x** from the command line
- Using **bash -x** is not as elegant, as it will debug the entire script and you might just want to focus on a specific section
- The benefit however is that you don't have to modify the script code while using **bash -x**

LINK FOR SCRIPTS:

- https://github.com/JaiTrieTree/bashscripts-devops

THANK YOU