

ĐẠI HỌC QUỐC GIA THÀNH PHỐ HỒ CHÍ MINH
TRƯỜNG ĐẠI HỌC BÁCH KHOA



OPERATING SYSTEM (CO2018)

Lab 1

GVHD: Trần Trương Tuấn Phát
Lớp L12 : Nguyễn Huy Hoàng - 2211091



Mục lục

1 Question 1 : List some other popular Linux shells and describe their highlighted features.	2
1.1 Zsh (Z Shell)	2
1.2 Fish (Friendly Interactive Shell)	2
1.3 Ksh (KornShell)	2
2 Question 2 : Compare the Output Redirection (>/») with the Piping () technique.	3
3 Question 3: Compare the sudo and the su command	4
4 Question 4: Discuss about the 777 permission on critical services (web hostings, sensitive database	5
5 Question 5:	5
5.1 What are the advantages of Makefile? Give examples?	5
5.2 Is there any Makefile mechanism for other programming languages? If it has, give an example?	6
5.3 Compiling a program in the first time usually takes a longer time in comparison with the next re-compiling. What is the reason?	7

1 Question 1 : List some other popular Linux shells and describe their highlighted features.

Beside Bash, here are some more Linux Shell:

1.1 Zsh (Z Shell)

Interactive Use: Zsh is highly customizable and designed for interactive use. It offers advanced tab completion, spelling correction, and shared command history among other features.

Plugins and Themes: Zsh supports plugins and themes through frameworks like Oh My Zsh, which enhances its usability and appearance.

Extended Globbing: Zsh offers powerful filename globbing capabilities, making it easier to work with files and directories.

1.2 Fish (Friendly Interactive Shell)

User - Friendly: Fish is designed to be user-friendly with features like syntax highlighting, auto-suggestions, and easy-to-remember commands.

Sane Defaults: Fish has sensible defaults which reduce the need for configuration, making it suitable for beginners.

Scripting Language: Fish has sensible defaults which reduce the need for configuration, making it suitable for beginners.

1.3 Ksh (KornShell)

Powerful Scripting Capabilities:

- Supports variables and arrays, allowing for easy data storage and retrieval.
- Provides comprehensive control flow structures like if, for, while, etc., enabling writing complex scripts.
- Supports functions, allowing for code modularity and reusability.

Effective Job Control:

- Enables running multiple jobs simultaneously and easy switching between them.
- Provides commands like bg, fg, and jobs to manage running jobs.
- Integrates job completion notification functionality.

2 Question 2 : Compare the Output Redirection (>/») with the Piping (|) technique.

Both Output Redirection (> and ») and Piping (|) are techniques used in command lines to manipulate the flow of data, but they achieve different goals:

Output Redirection (> and »):

1. Function: Sends the output (standard output) of a command to a file,
 - > (Output redirection): Overwrites the existing content of the destination file with the output of the command. If the file doesn't exist, it creates a new one.
 - » (Append redirection): Appends the output of the command to the end of the destination file. If the file doesn't exist, it creates a new one and appends the output to it.
2. Syntax:
 - command > filename: Overwrites the content of the file with the command's output (if it exists).
 - command » filename: Appends the command's output to the end of the file (creates the file if it doesn't exist).
3. Use Case: Saves the output of a command for later reference or processing.
4. Example: `ls > hello.txt` - Creates a file named "hello.txt" containing the list of files in the current directory.

Piping (|)

1. Function: Connects the standard output of one command to the standard input of another command.
2. Syntax: `command1 | command2`
3. Use Case: The output from command1 becomes the input for command2. This allows chaining commands together to perform complex tasks in a single line.
4. Example: `ls | grep .txt` - Lists all files with the ".txt" extension in the current directory. Here, `ls` outputs the directory listing, which is then piped to `grep` that filters for lines containing ".txt".

In summary, use redirection when you want to capture the output of a command and save it to a file. Use piping when you want to use the output of one command as the input for another command.

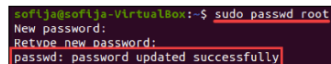
3 Question 3: Compare the sudo and the su command

Both sudo and su are commands used in Linux to execute commands with elevated privileges, typically those requiring root access. However, they differ in terms of security, flexibility, and use cases:

sudo (superuser do)

1. Function: Allows a user to execute a specific command with root privileges.
2. Security:
 - Requires the user's own password, not the root password.
 - More secure as it grants temporary privilege escalation for a single command.
 - System administrators can configure detailed permissions for each user using sudoers file, specifying which commands they can run with sudo and potentially requiring additional authentication (like a password prompt every time).
3. Flexibility: Offers fine-grained control over what commands users can execute with elevated privileges.
4. Use Case: Ideal for everyday administrative tasks where a user needs root access for a single command, like installing software or modifying system files.

Sudo example:



```
sofi@sofijs-VirtualBox:~$ sudo passwd root
New password:
Retype new password:
passwd: password updated successfully
```

su (switch user)

1. Function: Completely switches the current user to the root user account.
2. Security:
 - Requires the root password, which is a significant security risk if compromised.
 - Grants complete root access to the entire system, potentially allowing unintended modifications.
3. Flexibility: Offers full root access but lacks the granular control of sudo.
4. Use Case: Primarily used by system administrators for extensive administrative tasks requiring prolonged root access, or situations where sudo might not be available (e.g., initial system setup).

Su example:

```
soflja@soflja-VirtualBox:~$ su
Password:
su: Authentication failure
soflja@soflja-VirtualBox:~$ su -
Password:
su: Authentication failure
```

4 Question 4: Discuss about the 777 permission on critical services (web hostings, sensitive database

Setting file or directory permissions to 777, which grants full read, write, and execute access to everyone, including the owner, group, and other users, poses significant security risks for critical services such as web hosting and sensitive databases. These risks include unauthorized access, data manipulation, compromised integrity, non-compliance with regulations, vulnerability to attacks, and potential data loss. Instead, it's crucial to follow the principle of least privilege, granting permissions based on necessity and implementing proper access controls to mitigate security threats and maintain the integrity of critical systems.

5 Question 5:

5.1 What are the advantages of Makefile? Give examples?

Automation: Makefiles automate the compilation process by specifying dependencies and rules for building a project. This saves time and effort, especially in larger projects with multiple source files.

Efficiency: Makefiles only compile files that have changed since the last compilation, avoiding unnecessary recompilation of unchanged files.

Dependency Management: Makefiles manage dependencies between source files and automatically rebuild affected files when dependencies change.

Customization: Makefiles allow for customization of build options, such as compiler flags and build targets.

```
# Example Makefile for a C program
CC = gcc
CFLAGS = -Wall -O2
TARGET = myprogram
SOURCES = main.c utils.c

$(TARGET): $(SOURCES)
    $(CC) $(CFLAGS) -o $@ $^

clean:
    rm -f $(TARGET)
```

Hình 1: Example Makefile for a C program

5.2 Is there any Makefile mechanism for other programming languages? If it has, give an example?

Yes, Makefile mechanisms exist for various programming languages besides C. Makefiles can be adapted to compile projects written in languages like C++, Java, Python, and more. Each language may have its own compiler and specific build requirements, but the concept of Makefiles remains the same - specifying dependencies and rules for building the project.

```
# Example Makefile for a Java project
JAVAC = javac
SOURCES = Main.java Utils.java
TARGET = Main

all: $(TARGET)

$(TARGET): $(SOURCES)
    $(JAVAC) $^

clean:
    rm -f *.class
```

Hình 2: Example Makefile for a Java project

5.3 Compiling a program in the first time usually takes a longer time in comparison with the next re-compiling. What is the reason?

Reason for Longer Compilation Time Initially: The initial compilation typically takes longer due to the need to compile all source files from scratch and generate object files and executables. Additionally, the compiler may need to perform various optimizations and generate additional metadata during the first compilation. Subsequent re-compilations are faster because Makefiles track dependencies and only recompile files that have changed since the last build, resulting in fewer files needing compilation.