LP ASSIGNMENT-4

ENG24CY0159

SHADIK KHAN

1. A system has a file /etc/passwd. How would you use grep + tee to extract usernames and save them to a file while also displaying them on screen?

This is a great question, but for this job, grep isn't the best tool. The /etc/passwd file is structured with colons (:) separating the information. The best tool to slice it up is cut.

Here's how you can do it with cut and tee:

Command:

Bash

```
cut -d':' -f1 /etc/passwd | tee usernames.txt
```

How it works, step-by-step:

1. **cut -d':' -f1 /etc/passwd**: This part does the heavy lifting. We tell the cut command to look at the /etc/passwd file, use the colon as a slicer (-d':'), and give us just the very first field (-f1), which is the username.
2. **|**: This pipe takes the clean list of usernames that cut produced.
3. **tee usernames.txt**: And sends it to tee, which does two things at once: it displays the list on your screen and saves a copy to the file usernames.txt.

---

2. A binary isn’t found in $PATH. How would you use commands (which, find, locate) to troubleshoot and fix the issue?

This is a classic Linux problem. Your shell is like an assistant with a short list of contacts (the $PATH). If a program isn't on that list, the shell says, "Sorry, I don't know who that is."

Here's how you can play detective and fix it:

1. **Confirm the Problem with which**: First, run which the_command. If it's on the list, which will tell you where it is. If it comes back empty, you've confirmed the program isn't in your $PATH.
2. **Find the Missing Program**: Now you need to find where the program is actually located. You have two options:
    - **The Fast Way (locate):** locate the_command. This is super fast because it searches a pre-built index.
    - **The Slow-but-Sure Way (find):** sudo find / -name "the_command". This searches your entire computer in real-time, so it's guaranteed to find it if it exists.
3. **Fix the $PATH**: Once you've found its location (let's say it's in /opt/cool_app/bin), you need to add it to your shell's "contact list."
    - **The Temporary Fix:** export PATH=$PATH:/opt/cool_app/bin This works for your current terminal session but will be forgotten when you close it.
    - **The Permanent Fix:** Add that exact same export line to the bottom of your shell's configuration file (like ~/.bashrc or ~/.zshrc). The next time you open a terminal, your assistant will know exactly where to find your program.

---

3. Write a command pipeline that finds all .log files modified in the last 24 hours in /var/log and saves results into log_report.txt.

You can get this done with a single, powerful find command.

Command:

Bash

```
find /var/log -type f -name "*.log" -mtime -1 > log_report.txt
```

Here's the breakdown:

- **find /var/log**: Tells find to start its search in the /var/log directory.
- **-type f**: We're only interested in **f**iles, not directories.
- **-name "*.log"**: We only want files whose names end with .log.

- **-mtime -1**: This is the magic part. It tells find to only grab files that have been **m**odified in less than (-) **one** (1) 24-hour period.
- **> log_report.txt**: This is a simple redirect. Instead of printing the list of files to the screen, it sends the entire output straight into the log_report.txt file.

---

4. What is the difference between shutdown -r now and reboot?

In the old days, there was a technical difference. But on any modern Linux system, **they effectively do the exact same thing.**

Think of it like two buttons in a car labeled "Turn Off Engine" and "Stop Car." Today, they're both wired to the same safe, modern engine shutdown procedure.

- **shutdown -r now** is the "official" command. It's more powerful because it can be used to schedule restarts (e.g., shutdown -r +10 for a restart in 10 minutes) and send warning messages to users.
- **reboot** is essentially a convenient shortcut. When you type reboot, your system almost always just runs shutdown -r now for you in the background.

So, for an immediate restart, you can use them interchangeably.

---

5. How can you use the tee command to debug a script that generates both standard output and error messages?

This is a fantastic use for tee, but it requires a little trick. By default, the pipe (|) only grabs a script's normal output (stdout) and ignores its error messages (stderr).

Imagine your script has two mouths: one for talking (stdout) and one for shouting errors (stderr). The pipe only listens to the talking mouth. The trick, 2>&1, is like putting a funnel over both mouths so that tee can hear everything.

Command:

Bash

```
./my_buggy_script.sh 2>&1 | tee debug_session.log
```

How it works:

1. ./my_buggy_script.sh: We run our potentially buggy script.
2. 2>&1: This is the magic funnel. It says, "Take the error stream (2) and redirect it to the standard output stream (1)."
3. | tee debug_session.log: Now that both normal output and errors are in one stream, we can pipe it all to tee to see it on our screen AND save a complete record in debug_session.log for later.

---

6. Explain any three real-world applications of Linux in industries.

You're probably using Linux in several ways right now without even realizing it! It runs the world in the background.

1. **The Internet and the Cloud:** The vast majority of websites, online services, and cloud platforms (like Amazon's AWS and Google Cloud) run on Linux servers. Its stability, security, and zero cost make it the undeniable king of the internet's infrastructure.
2. **Smartphones:** If you have an Android phone, you're carrying a device powered by the Linux kernel. Google built the entire Android operating system on top of Linux, making it the most deployed OS kernel on the planet.
3. **Supercomputers:** When scientists need to model the climate, research diseases, or design new technology, they use the world's fastest supercomputers. Over 99% of these massive machines run Linux because it's powerful, efficient, and can be customized for highly specific scientific tasks.

---

7. Differentiate application, system and utility software in the context of Linux environment.

Let's think of your computer like a restaurant.

- **System Software (e.g., the Linux Kernel):** This is the restaurant itself—the building, the foundation, the plumbing, and the electricity. It's the absolute essential core that everything else depends on, but it doesn't actually cook any food.

- **Utility Software (e.g., ls, grep, cp):** These are the kitchen tools—the knives, the ovens, the mixers. They are the essential utilities that the chefs use to manage the kitchen and prepare the food. You don't serve an oven to a customer, but you can't cook without one.
- **Application Software (e.g., Firefox, LibreOffice, GIMP):** This is the actual meal that gets served to the customer. It's the reason the customer came to the restaurant in the first place. These are the programs you interact with to get your work done.

---

8. What are the key differences between open-source and proprietary operating systems?

Think of it like the difference between a community cookbook and a secret family recipe.

- **An Open-Source OS (like Linux)** is the community cookbook.
    - **Code:** The recipe is public. Anyone can view it, use it, share it, and even suggest improvements.
    - **Cost:** It's free for everyone.
    - **Support:** If you have a problem, you ask the community (forums, wikis) or pay a specialist (like Red Hat) for help.
    - **Freedom:** You can change any part of the recipe you don't like.
- **A Proprietary OS (like Windows or macOS)** is the secret family recipe.
    - **Code:** The recipe is a secret, locked in a vault. Only the company that owns it knows what's inside.
    - **Cost:** You have to pay for it, either by buying a license or by buying the company's expensive hardware.
    - **Support:** You can only get official help from the company that owns the recipe.
    - **Freedom:** You have to use the recipe exactly as it's given to you.

---

9. Write the command to display the system's kernel version.

This is a nice and simple one. The command is uname.

Command:

Bash

uname -r

The -r flag specifically asks for the kernel **r**elease version. If you want to see everything uname knows (like the OS, architecture, etc.), you can use uname -a (for **a**ll).

---

10. What is the difference between head and tail commands in text processing?

It's as simple and literal as it sounds! When you're looking at a text file:

- **head**: Shows you the beginning of the file (the top of the page). By default, it shows the first 10 lines.
- **tail**: Shows you the end of the file (the bottom of the page). By default, it shows the last 10 lines.

You use head when you want a quick peek at what a file is about. You use tail when you want to see the most recent entries, which is incredibly useful for looking at log files.

The best feature of tail is tail -f filename.log, which will **f**ollow the file and show you new lines as they are added in real-time. It's like streaming a log file directly to your terminal.
