Javan Pahlavan Mentorship

Course Assignment – T(5)

Prepared By:

Mahdi Siamaki

(siamaki.me@gmail.com)

Instructor:

Mr. Mahdi Bahrami

July 2025

Question 1:

Please explain the use cases of soft and hard links. When should we use a soft link, and when is a hard link more appropriate?

Links in Unix-like systems are pointers to files. They allow a single file to be referenced by multiple names or from multiple locations.

Hard Links:

A **hard link** is a direct reference to a file's data on the disk, identified by its inode number. It's essentially a second directory entry for the exact same file.

- **How it works:** All hard links to a file share the same inode. The filesystem maintains a "link count" within the inode. When a hard link is created, this count is incremented. When a link is deleted (rm), the count is decremented. The actual file data is only removed from the disk when the link count reaches zero.
- **Limitations:**
    1. You cannot create a hard link to a directory (to prevent recursive loops in the filesystem tree).
    2. Hard links cannot span across different filesystems or partitions, as inode numbers are only unique within a single filesystem.

When to use a Hard Link:

- **Data Safety/Backup:** If you have a critical file, you can create a hard link to it in another directory on the same filesystem. If the original file name is deleted, the data still exists and is accessible through the hard link. This is useful for creating simple, space-efficient "snapshots."
- **Consolidating Files:** When you want multiple file names in different locations to refer to the exact same content without duplicating data, saving disk space.

Soft Links (Symbolic Links or Symlinks):

A **soft link** is not a direct pointer to the data but a special type of file whose content is simply the path to another file or directory.

- **How it works:** A soft link has its own unique inode. When the system accesses a soft link, it reads the path stored within it and then follows that path to access the target file.
- **Advantages:**
    1. Can link to directories.
    2. Can span across different filesystems and partitions.
    3. The link can point to a file that does not yet exist (a "dangling" or "broken" link).

When to use a Soft Link:

- **Linking Across Filesystems:** This is the most common reason.

  If /home and /usr/local/bin are on different partitions, you must use a soft link to link a file between them.

- **Linking to Directories:** Creating shortcuts to frequently used directories (e.g., ln -s /var/log/apache2 ~/apache_logs).

- **Version Management:** A common practice is to have a program point to a specific version via a soft link. For example, /usr/bin/java might be a soft link to /opt/jdk-11.0.12/bin/java. To upgrade Java, you just change the soft link to point to the new version directory, and all scripts using /usr/bin/java will automatically use the new version.

- **Clarity:** Soft links clearly show (ls -l) where they point, making them easier to understand and debug than hard links.

Feature	**Hard Link**	**Soft Link (Symbolic Link)**
Concept	Another name for the same inode	A separate file containing a path to the target
Inode	Shares the same inode as the target	Has its own, different inode
Cross-filesystem	No	Yes
Link to Directory	No	Yes
Target Deletion	Data remains until all hard links are gone	Link becomes "broken" or "dangling"
Command	ln target_file link_name	ln -s target_file link_name

Question 2:

Describe the inode relationship of soft links. How do inodes work with soft and hard links? Please show inode numbers and explain how the inode accesses the data on the hard disk.

An **inode** (index node) is a data structure on a filesystem that stores all the metadata about a file except its name and its actual data. This includes:

- File permissions (owner, group, etc.)
- File size
- Timestamps (creation, modification, access)
- The link count (number of hard links)
- **Pointers to the data blocks** on the hard disk where the file's content is stored.

The filename, which resides in a directory entry, simply points to the inode number.

How Inodes Work with Links:

Hard Links:

A hard link creates a new filename that points to the **exact same inode** as the original file.

- They share the same inode number.

- Modifying the content through one link changes it for all links, because they all point to the same data blocks.
- The inode's link count increases with each new hard link.

Soft Links:

A soft link is a completely separate file with its **own unique inode**.

- The data block for the soft link's inode contains the text path of the target file.
- The system recognizes this file as a special "symbolic link" type. When you try to access it, the kernel reads the path from its data block and then follows that path to the target file. It then accesses the *target file's inode* to find its data.

Demonstration:

Let's create a file and then create both a hard and a soft link to it. We use ls –i to show inode numbers.

```
# 1. Create an original file and see its inode

$ echo "This is the original file data." > original.txt

$ ls -li original.txt

1324083 -rw-r--r-- 1 user user 32 Sep 20 10:00 original.txt

# The inode number is 1324083. The '1' after permissions is the link count.



# 2. Create a hard link

$ ln original.txt hard_link.txt

$ ls -li original.txt hard_link.txt

1324083 -rw-r--r-- 2 user user 32 Sep 20 10:00 hard_link.txt

1324083 -rw-r--r-- 2 user user 32 Sep 20 10:00 original.txt

# Notice: Both files have the SAME inode number (1324083).

# The link count for both is now '2'.



# 3. Create a soft link

$ ln -s original.txt soft_link.txt

$ ls -li original.txt soft_link.txt

1324083 -rw-r--r-- 2 user user 32 Sep 20 10:00 original.txt

1324084 lrwxrwxrwx 1 user user 12 Sep 20 10:01 soft_link.txt -> original.txt

# Notice: soft_link.txt has a DIFFERENT inode number (1324084).

# The file type is 'l' for link, and it shows the path it points to.
```

How the Inode Accesses Data:

1. When you issue a command like cat hard_link.txt:
   - The system looks up hard_link.txt in the current directory.
   - It finds the corresponding inode number: 1324083.
   - The kernel goes to the inode table and retrieves inode 1324083.
   - From the inode, it gets the pointers to the data blocks on the disk.
   - It reads the data from those blocks and displays it.
2. When you issue cat soft_link.txt:
   - The system looks up soft_link.txt and finds its inode number: 1324084.
   - The kernel retrieves inode 1324084 and sees it's a symbolic link.
   - It reads the data from the soft link's data block, which is the path "original.txt".
   - The kernel then starts a new lookup for "original.txt", finds its inode (1324083), and proceeds as in step 1.

Question 3:

What is the /opt directory used for? How does it differ from /etc and /sbin in terms of purpose and usage?

This question relates to the **Filesystem Hierarchy Standard (FHS)**, which defines the main directories and their contents in Linux distributions.

/opt (Optional)

- **Purpose:** Reserved for the installation of add-on application software packages from third-party vendors.
- **Usage:** Software that does not follow the standard Linux file layout (e.g., placing binaries in /usr/bin, libraries in /usr/lib, and configs in /etc) can be installed here. The package will typically create a self-contained directory under /opt, such as /opt/google/chrome or /opt/slack. This makes it easy to install and uninstall the software without cluttering the rest of the system.

/etc (Etcetera)

- **Purpose:** Contains system-wide configuration files and shell scripts that are run at boot time.

- **Usage:** These are almost exclusively static, human-readable text files. You would edit files in /etc to change the behavior of the system or its services. It should not contain any binaries.
- **Examples:** /etc/fstab (filesystem table), /etc/passwd (user database), /etc/ssh/sshd_config (SSH server configuration).

/sbin (System Binaries)

- **Purpose:** Contains essential system administration binaries, which are programs required for booting, restoring, recovering, and/or repairing the system.
- **Usage:** These are executable programs generally intended to be run only by the root user. On many systems, /sbin is not in the default $PATH for regular users.
- **Examples:** fdisk (partition table manipulator), reboot, ifconfig (network interface configurator), mkfs (make filesystem).

Directory	**Purpose**	**Content Type**	**Who Uses It?**
/opt	Optional, self-contained third-party software	Binaries, libraries, configs, etc.	System administrators for installation; all users to run the software.
/etc	System-wide configuration files	Human-readable text files	Primarily system administrators for configuration.
/sbin	Essential system administration commands	Executable binaries	Primarily the root user for system management.

Question 4:

Compare the locate and find commands. Which one is faster, and why?

locate and find are both used to search for files, but they work in fundamentally different ways.

find

- **How it Works:** find searches the filesystem in **real-time**. It traverses the directory tree starting from a specified path and checks every file and directory against the criteria you provide.
- **Pros:**
  - **Real-time:** It always works with the current state of the filesystem. It will find files created seconds ago.
  - **Powerful:** Can search based on a wide range of attributes, including name, size, modification time, permissions, owner, type, and more.
- **Cons:**
  - **Slower:** Because it has to actively read directory and file metadata from the disk, it can be very slow, especially when searching a large filesystem tree like /.

Example: find /home -name "*.log" -mtime -7 (Find all files ending in .log in the /home directory that were modified in the last 7 days).

locate

- **How it Works:** locate does not search the filesystem directly. Instead, it searches a **pre-built database** (usually located at /var/lib/mlocate/mlocate.db). This database is a simple list of file paths on the system.
- **Pros:**
  - **Extremely Fast:** Searching a single, indexed database file is significantly faster than walking the entire filesystem.
- **Cons:**
  - **Not Real-time:** The database is typically updated only once a day by a cron job running the updatedb command. Any files created, moved, or deleted since the last update will not be reflected in locate's results.
  - **Limited:** It can only search by filename/path. It cannot search by size, permissions, modification time, etc.

Example: locate mydocument.txt

Which is faster and why?

locate is dramatically faster.

The reason is its method of operation. find performs extensive I/O operations by traversing directories and reading inode information for potentially millions of files across the disk. locate performs a single, highly optimized read operation on one database file. The hard work of scanning the filesystem was already done by updatedb at a time when system load was likely low.

Question 5:

Please explain how your device's updatedb.conf is configured. How does it affect the behavior of the locate command?

The updatedb.conf file, typically located at /etc/updatedb.conf, is the configuration file for the updatedb command. Since locate depends entirely on the database built by updatedb, this configuration file indirectly controls what locate can and cannot find.

The purpose of updatedb.conf is to tell updatedb which parts of the filesystem it should **exclude** from its scan. This is done to:

1. Avoid scanning volatile or temporary directories (e.g., /tmp).
2. Avoid scanning pseudo-filesystems that don't contain real files (e.g., /proc, /sys).
3. Avoid scanning network filesystems (e.g., nfs, cifs) which could be slow and generate unnecessary network traffic.

4. Prevent sensitive directories from appearing in the database.

Key Configuration Directives:

- PRUNEFS: A space-separated list of filesystem types to not scan. The scanner will not cross from one filesystem into another of a type listed here.
    - PRUNEFS = "NFS nfs nfs4 rpc_pipefs afs binfmt_misc proc smbfs autofs iso9660 ncpfs coda devpts ftpfs devfs mfs shfs sysfs cifs lustre tmpfs usbfs udf fuse.glusterfs fuse.sshfs curlftpfs"
- PRUNEPATHS: A space-separated list of absolute directory paths to not scan.
    - PRUNEPATHS = "/tmp /var/spool /media /home/.ecryptfs"
- PRUNE_BIND_MOUNTS: A setting ("yes" or "no") that tells updatedb whether to scan the contents of a directory that is a bind mount point. The default is "no" to avoid indexing the same content multiple times.
    - PRUNE_BIND_MOUNTS = "yes"

How it affects locate:

The configuration has a direct and absolute impact. **If a file or directory is excluded by a rule in updatedb.conf, it will not be added to the mlocate.db database. Consequently, the locate command will be unable to find it.**

For example, if PRUNEPATHS includes /tmp, and you create a file /tmp/important_file.txt, running sudo updatedb will not add this file to the database.

A subsequent locate important_file.txt command will return no results, even though the file exists.

Question 6:

What is the difference between .bashrc and .bash_profile? What is the purpose of /etc/bash_completion?

This relates to how the Bash shell initializes itself. The key is understanding the difference between a **login shell** and a **non-login interactive shell**.

- **Login Shell:** A shell session that starts after you authenticate (log in). Examples include connecting via SSH or logging into a text-based console.
- **Non-Login Interactive Shell:** A shell started from an already logged-in session. The most common example is opening a new terminal window inside a graphical desktop environment.

.bash_profile vs. .bashrc

- **~/ .bash_profile:**
  - **When it's read:** Executed **only by login shells**.

    - **Purpose:** To run commands that should only be executed once per session, at login time. This is the ideal place for setting environment variables that don't change, like PATH, JAVA_HOME, or EDITOR.

- **~/ .bashrc:**
    - **When it's read:** Executed by **non-login interactive shells**.
    - **Purpose:** To run commands for every new interactive shell. This is the correct place for defining shell-specific things like aliases, shell functions, and customizing the command prompt (PS1).

The Common Convention:

Most users want their aliases and functions to be available in both login and non-login shells. To achieve this, a standard practice is to add the following lines to ~/.bash_profile. This code checks if ~/.bashrc exists and, if so, loads it.

```
# in ~/.bash_profile

if [ -f ~/.bashrc ]; then

   . ~/.bashrc

fi
```

This makes the distinction less critical for daily use, but it's vital to understand the underlying mechanism.

/etc/bash_completion

This is not a file that is sourced directly by default. It's a directory (/etc/bash_completion.d/) or script that provides programmable **tab completion** for a huge number of commands.

- **Purpose:** To enhance user productivity by allowing complex command arguments to be completed by pressing the <Tab> key.
- **How it works:** System-wide shell startup files
  (like /etc/profile or /etc/bash.bashrc) source the main bash completion script. This script then defines a framework and sources smaller completion scripts for individual commands (e.g., git, docker, systemctl, apt).
- **Example:** After sourcing bash completion, you can type git comm<Tab> and it will complete to git commit. Or systemctl restart ap<Tab> and it will complete to systemctl restart apache2. This is far more advanced than simple filename completion.

Question 7:

What kind of information is stored in .bash_logout and .bash_login? What are their typical use cases?

~/.bash_login

- **Information/Purpose:** This file serves the same purpose as ~/.bash_profile. It is read and executed by **login shells**.
- **Execution Order:** Bash looks for ~/.bash_profile first. If it exists, ~/.bash_login and ~/.profile are ignored. If ~/.bash_profile does not exist, Bash then looks for ~/.bash_login.
- **Typical Use Case:** It's essentially a fallback for ~/.bash_profile. Most modern Linux distributions create a ~/.bash_profile or ~/.profile by default, so ~/.bash_login is less commonly used today. You would use it for setting environment variables and running startup commands if you were on a system that favored it over ~/.bash_profile.

~/.bash_logout

- **Information/Purpose:** This file is executed when a **login shell exits**. It does not contain stored information but rather commands to be executed upon logout.
- **Typical Use Cases:**

- **Cleanup:** Running commands to clean up the user's session.
    - Deleting temporary files created during the session: rm -f /tmp/my_app_*
- **Logging:** Recording the logout time to a log file.
- **System Messages:** Displaying a message to the user, like "Session ended. Don't forget to backup your files."
- **Clearing the screen:** Running the clear command so that the next user at a physical terminal doesn't see the previous user's screen history.

Example ~/ .bash_logout file:

```
#!/bin/bash

# Commands to be executed upon logout



clear

echo "You have been logged out. The time is $(date)."
```

Question 8:

Please demonstrate how to customize the command prompt (PS1) in the .bashrc file.

The shell prompt is controlled by the PS1 environment variable. You can customize it by setting PS1 in your ~/.bashrc file. This ensures it's applied every time you open a new terminal.

The PS1 string can contain special backslash-escaped characters that are replaced with system information.

Common PS1 escapes:

- \u: The username of the current user.
- \h: The hostname, up to the first dot.
- \H: The full hostname.
- \w: The current working directory (e.g., /home/user/project).
- \W: The basename of the current working directory (e.g., project).
- \$: Displays a $ if you are a regular user, or a # if you are the root user.
- \t: The current time in HH:MM:SS format.
- \d: The date in "Weekday Month Date" format.

Adding Colors:

You can add colors using the format \[\e[COLOR_CODEm\]. The \[...\] part is important as it tells Bash that the characters inside do not take up space on the line.

- \e[0m: Reset all color attributes.
- \e[31m: Red text.
- \e[32m: Green text.
- \e[34m: Blue text.
- \e[1;33m: Bold (1) Yellow (33) text.

Demonstration:

Step 1: Edit ~/ .bashrc

Open the file with your favorite editor: nano ~/.bashrc

Step 2: Add your custom PS1 setting

Go to the end of the file and add your new PS1 export. Here are a few examples from simple to complex.

Example 1: A simple user@host:directory$ prompt

```
# in ~/.bashrc

export PS1='\u@\h:\w\$ '
```

Result: user@my-laptop:~/documents$

Example 2: A colorful, multi-line prompt

This prompt shows the user and host in green, the directory in blue, and puts the $ on a new line.

```
# in ~/.bashrc

# Define colors for readability

GREEN="\[\e[32m\]"

BLUE="\[\e[34m\]"

RESET="\[\e[0m\]"



export PS1="${GREEN}\u@\h${RESET}:${BLUE}\w${RESET}\n\$ "
```

Result:

```
user@my-laptop:/home/user/documents

$
```

Step 3: Apply the changes

To see the new prompt, you can either:

- Source the file in your current shell: source ~/.bashrc
- Open a new terminal window.

Question 9:

Write a Bash script that takes a webpage URL as input and downloads all the URLs (links) found on that page.

This script will use curl to fetch the webpage content and grep with a regular expression to extract all URLs found in href attributes. It will then attempt to download each unique URL using wget.

```
#!/bin/bash


# A script to find and download all links from a given URL.


# --- Check for input ---

if [ -z "$1" ]; then

  echo "Usage: $0 <URL>"

  echo "Example: $0 https://www.gnu.org"

  exit 1

fi


BASE_URL="$1"
```

```
# Create a directory to store the downloaded files, named after the host

HOST=$(echo "$BASE_URL" | grep -oP 'https?://\K[^/]+')

DOWNLOAD_DIR="downloads_${HOST}"

mkdir -p "$DOWNLOAD_DIR"



echo "Fetching links from: $BASE_URL"

echo "Files will be saved in: $DOWNLOAD_DIR"



# --- Fetch the webpage and extract URLs ---

# -s: silent mode for curl

# -o: output only matching parts for grep

# -P: use Perl-compatible regular expressions (for \K)

# 'href="(\K[^"]+)"': Find href=", then \K discards the href=" part,

#                 then [^"]+ matches everything until the next quote.

# sort -u: Get only unique URLs

URLS=$(curl -s "$BASE_URL" | grep -oP 'href="\K[^"]+' | sort -u)



if [ -z "$URLS" ]; then

  echo "No links found on the page."

  exit 0

fi
```

```
# --- Loop through URLs and download them ---

echo "--- Found URLs ---"

echo "$URLS"

echo "-------------------"



for url in $URLS; do

 # Handle relative URLs (that start with / or just a filename)

 if [[ "$url" =~ ^https?:// ]]; then

  # Absolute URL, use as is

  TARGET_URL="$url"

 elif [[ "$url" =~ ^// ]]; then

  # Protocol-relative URL, prepend https:

  TARGET_URL="https:$url"

 elif [[ "$url" =~ ^/ ]]; then

  # Root-relative URL, prepend base host

  TARGET_URL="$(echo $BASE_URL | grep -oP 'https?://[^/]+')$url"

 else

  # Page-relative URL, prepend full base URL path

  TARGET_URL="${BASE_URL%/}/$url"

 fi
```

```
  echo "Downloading: $TARGET_URL"

  # -P: specify download directory

  # -q: quiet mode for wget

  # --no-check-certificate: useful for sites with self-signed certs

  wget -P "$DOWNLOAD_DIR" -q --no-check-certificate "$TARGET_URL"



  if [ $? -eq 0 ]; then

    echo "  -> Success"

  else

    echo "  -> Failed"

  fi

done



echo "Download process complete. Check the '$DOWNLOAD_DIR' directory."
```

How to use:

1. Save the code above as download_links.sh.
2. Make it executable: chmod +x download_links.sh.
3. Run it with a URL: ./download_links.sh https://example.com.

Question 10:

Please explain the use of the exit command and other common exit-related statements in Bash scripting.

In Bash scripting, controlling the flow and termination of a script is crucial for creating robust and predictable programs.

exit command

The exit command terminates the entire script immediately.

- **Exit Status:** Its most important feature is the ability to provide an **exit status** (or exit code), which is an integer between 0 and 255. By convention:
    - **0** means **success**.
    - Any number from **1 to 255** means **failure** or an error condition.
- **Usage:**
    - exit: Exits the script with the exit status of the last command that was executed.
    - exit n: Exits the script with the specific integer status n.

- **Checking the Exit Status:** The special variable $? always holds the exit status of the most recently executed command.

Example:

```
#!/bin/bash

if ! grep -q "username" /etc/passwd; then

  echo "Error: User 'username' not found." >&2 # Redirect to stderr

  exit 1 # Exit with a failure code

fi



echo "User 'username' was found."

exit 0 # Explicitly exit with a success code
```

After running this script, you could check its success with echo $?.

Other Common Exit-Related Statements

Statement	Scope	Purpose
return	Function	Exits the current **function**, not the whole script. It can also return a numeric status code (0-255) to the calling part of the script, which is then available in $?.
break	Loop	Immediately terminates the current loop (for, while, until). Execution continues with the first command *after* the loop.
continue	Loop	Skips the remaining commands inside the current loop **iteration** and begins the next iteration of the loop.

Code Examples:

return example:

```
check_file() {

 if [ -f "$1" ]; then

  echo "File exists."

  return 0 # Success

 else

  echo "File does not exist."

  return 1 # Failure

 fi

}



check_file "/etc/hosts"

echo "Function exited with status: $?" # Will be 0



check_file "/nonexistentfile"

echo "Function exited with status: $?" # Will be 1
```

break and continue example:

```
for i in {1..10}; do

 if [ $i –eq 3 ]; then

  echo "Skipping number 3."

  continue # Skip the rest of this iteration, go to i=4

 fi



 if [ $i –eq 7 ]; then

  echo "Found number 7. Exiting loop."

  break # Terminate the for loop entirely

 fi



 echo "Processing number $i"

done



echo "Loop finished." # This line will be executed after the break
```

Output:

```
Processing number 1

Processing number 2

Skipping number 3 .

Processing number 4

Processing number 5

Processing number 6

Found number 7 . Exiting loop .

Loop finished .
```