Shells, File Systems, and Emacs

1. File Systems and Links

1.1. File System Overview

The file system is organized hierarchically as a tree, with the root directory `/` at its base.

- Root Directory (`/`): The entry point to the file system.
- Subdirectories: Standard directories include `/bin`, `/usr`, `/home`, etc.
- Types of Entries:
  - Regular files (e.g., `.gitignore`)
  - Directories
  - Symbolic links

Each file or directory is uniquely identified within the file system by its inode number.

1.2 Path Resolution

- Absolute path: begins with `/`.
- Relative path: begins from current working directory.
- The resolution algorithm (namei-like function):
  1. Start from `/` if absolute, otherwise current directory.
  2. Traverse each component, resolving directories.
  3. Resolve symlinks if present.

1.3. iNodes and File Identification

Each file or directory is assigned a unique identifier called an inode number. - Inodes store metadata (ownership, permissions, timestamps), not name. - Commands to view inode numbers: - `ls -i filename` - `ls -li` for long format with inode.

Example:

```
ls -li /bin/sh
1234567 lrwxrwxrwx 1 root root 4 Jan 1 00:00 /bin/sh -> bash
```

Where `1234567` is the inode number.

1.4. Hard Links

Definition: - Links to the same inode from different directory entries. - Equivalent entries pointing to the same file content.

Key Facts: - Hard links to the same file share the same inode. - Cannot make hard links to directories. - Changes made through one hard link are visible through others.

Example:

```
ln file1 file2
```

Name	Inode
file1	97311
file2	97311

1.4. Symbolic (Soft) Links

Definition: A symbolic link (symlink) is a reference to another path in the file system. Unlike hard links, symlinks are a separate file containing a path as a string.

Key Facts: - Symlinks have their own inode. - They can point to non-existent targets (dangling symlinks). - Symlinks can be relative or absolute.

Command:

```
ln -s target symlink
```

Types of symlinks:

Type	Starts with	Interpreted From
Relative	No `/`	Directory containing the link
Absolute	`/`	Interpreted from root `/`

Example:

```
ln -s /bin/sh mysh  # Absolute
ln -s ../bin/sh relsh  # Relative
```

---

2. Pathname Resolution

2.1. Absolute vs. Relative Paths

- Absolute paths start with `/` and begin resolution from the root.
- Relative paths do not start with `/` and are resolved from the current working directory.

Each process maintains a current working directory.

Change directory:

```
cd dirname
```

2.2. Symbolic Link Resolution

- If a pathname includes a symbolic link, the system replaces that component with the content of the symlink.
- If the symlink is:
  - Relative: interpreted relative to the directory containing the symlink.
  - Absolute: restart resolution from the root.

Example resolution:

```
/bin/sh
```

If `/bin` is a symbolic link to `usr/bin`, the path transforms to:

```
/usr/bin/sh
```

2.3. Common Errors and Loops in Symlinks

- Dangling symlink:

```
ln -s nowhere foo
cat foo  # No such file or directory
```

- Infinite Loop:

```
ln -s B A
ln -s A B
cat A  # Fails due to loop
```

To prevent infinite symbolic link loops, the system tracks the number of traversals (typically capped at ~20 in Linux).

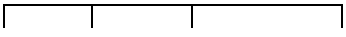
2.4. Hard Links to Symbolic Links

- Multiple hard links to the same symbolic link result in shared link counts.

Example:

```
ln foo bar
```

if `foo` is a symlink.



Name	Type	Link Count
foo	symlink	2
bar	symlink	2

Removing one doesn't delete the symlink, only removes one reference.

2.5. Symbolic Links and Directory Context

Relative symlinks behave differently based on the path used to reference them.

Example:

```
mkdir e
ln foo e/.baz
```

- `e/.baz` may resolve correctly if relative from `e/`
- May not work if accessed from the parent directory.

---

3. The Shell

3.1. Shell Purpose and Use

- First widely used scripting language.
- Shell's primary job is coordination: launching and configuring other programs.
- Common tasks:
  - Configuring standard input/output
  - Variable expansion
  - Command sequencing

3.2. Command Execution and Standard I/O

Redirection operators: - `>`: Redirect standard output. - `<`: Redirect standard input. - `2>`: Redirect standard error.

Example:

```
cat file > output.txt 2> errors.log
```

Piping:

```
cmd1 | cmd2 | cmd3
```

Chains standard output from one command to standard input of the next.

3.3 Argument Passing

```
command arg1 arg2
```

- Argument count and values appear as:

```
int main(int argc, char** argv)
// argv[0] = command, argv[1] = arg1, ...
```

3.4. Tokenization and Quoting

Special Characters:

Characters with special interpretations unless quoted:

Character	Purpose
Space, Tab, Newline	Separators
`<`, `>`	I/O Redirection
`&`, `|`	Background & Pipe

`;`	Command separator
`#`	Comment
`$`	Variable expansion
`` ` ``	Command substitution
`(, )`	Grouping/Subshells

Quoting Mechanisms:

Method	Syntax	Behavior
Escape	`\\char`	Escape a single character
Single Quotes	`'text'`	Literal string
Double Quotes	`"text"`	Variable/command expansion allowed

Examples:

```
echo 'This is $HOME'  # Literal string
echo "This is $HOME"  # Expands $HOME
```

Multiline:

```
echo "Multi
Line"
```

Null Byte Limitation

Shell does not allow embedded null bytes in variables or strings.

3.5. Reserved Words

Type	Example
Conditional	`if`, `then`, `else`, `fi`
Loops	`for`, `in`, `do`, `done`, `while`, `until`
Case	`case`, `esac`
Grouping	`{`, `}`
Negation	`!`

Examples:

If Statement:

```
if cat file; then
  echo "Success"
else
  echo "Failure"
fi
```

While loop:

```
while cat file; do
  sleep 1
done
```

For loop:

```
for i in a b c; do
  echo $i
done
```

Case Statement:

```
case $x in
  *.c) gcc $x;;
  *.cpp) g++ $x;;
  *) echo "Unknown type";;
esac
```

3.6. Parameters and Variables

Shell scripts receive arguments as: - `$0`: Script name - `$1` to `$9`: Positional args - `$#`: Number of arguments - `$*`: All args as one word - `$@`: All args as distinct words

Key Differences:

Usage	Result
`"$*"`	All args as single string
`"$@"`	All args preserved individually

Special Examples:

```
x=a b c
echo $x  # a b c
echo "$x"  # "a b c"
```

Unset variables:

```
unset x
```

Default substitution:

```
${VAR:-default}  # Use default if VAR isn't set
${VAR:=default}  # Set VAR to default if not set
${VAR:+alt}      # Use alt if VAR is set
${VAR:?error}    # Print error if not set
```

Environment variables: Use `export` to make available in subcommands:

```
export EDITOR=emacs
```

3.7. Redirection and Pipes

Standard file descriptors:

Descriptor	Number	Default
stdin	0	Keyboard
stdout	1	Screen
stderr	2	Screen

Modifiers:

```
2>file     # stderr to file
1>&2       # stdout to stderr
command &  # Run in background
```

Advanced example:

```
A 2>err1 | B 2>err2 | C
```

Splitting Output: Use tee to duplicate output:

```
cmd | tee file | another_cmd
```

3.8 Multiple Writers or Readers

- Running concurrent processes that read/write from shared pipe creates race conditions.

Example:

```
cmd1 | tee output.txt | cmd2
```

3.9. Expansion Rules

Processing order:

1. Tilde expansion (`~`, `~user`)
2. Parameter expansion (`$VAR`)
3. Command substitution (`$(cmd)`)
4. Arithmetic expansion (`$((expr))`)
5. Field splitting (split on whitespace)
6. Pathname expansion (globbing, e.g. `*.c`)
7. Redirection

Example Evaluation:

```
echo $((1 + 2)) # => 3
```

Globbing:

```
ls *.c  # Lists all .c files
```

If no matching files: - The pattern remains unexpanded.

---

4. Summary

This lecture discusses the structure and function of POSIX-compatible file systems, focusing on file naming and linking mechanisms (both hard and symbolic). It explains the interpretation of file names through pathname resolution, including relative vs. absolute paths, and how symbolic links are resolved or can go wrong (e.g., dangling links, symlink loops). It then transitions into a comprehensive exploration of Unix shell behavior, discussing command execution, redirection, quoting rules, reserved words in scripts, and shell expansions. Special attention is paid to scripting constructs such as conditionals (`if`, `else`), loops (`for`, `while`), variables and parameter handling (`$@`, `$*`, `$#`), background process management, process IDs, and the use of pipes for process communication. The lecture concludes with an explanation of shell expansion mechanisms and variable substitution nuances essential for writing robust shell scripts.