Filesystems, Bash, and Scripting

1. File Systems and Links

1.1. File System Overview

The Unix file system is a hierarchical tree, with the root directory `/` at its base. Each entry (file, directory, or link) is uniquely identified by an **inode number**.

- **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 (symlinks)

Table: Common Unix Directories

Directory	Purpose
`/bin`	Essential user binaries
`/usr`	User programs, libraries
`/home`	User home directories
`/etc`	System configuration files
`/tmp`	Temporary files
`/var`	Variable data (logs, spool)
`/dev`	Device files
`/proc`	Process and kernel info (virtual)

Inodes:

- Each file or directory is represented by an inode, which stores metadata (owner, permissions, timestamps, file size, pointers to data blocks) but not the filename.
- Directories are special files that map filenames to inode numbers.

ASCII Diagram of a Simple Filesystem:

```
/
├── bin
│   └── sh
├── usr
│   └── bin
├── home
│   └── user
│       └── .gitignore
```

Each file or directory is identified by its inode number. You can view inode numbers with `ls -i` or `ls -li`.

Example:

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

Where `1234567` is the inode number.

1.2. Hard Links

- **Definition**: Multiple directory entries (names) that point to the same inode (file content).
- **Key Facts**:
  - Hard links share the same inode.
  - Cannot create hard links to directories (to avoid cycles and filesystem corruption).
  - Changes via one hard link are visible through all others.

- Removing one hard link does not delete the file until all links are removed (link count reaches zero).
- Hard links cannot span filesystems (must be on the same device).

Diagram: Hard Links

```
Directory A:           Directory B:
file1  ----+           file2  ----+
           |                      |
        [inode 97311: file data]  |
           +----------------------+
```

Example:

```
ln file1 file2
```

Name	Inode
file1	97311
file2	97311

Why no hard links to directories?

- Allowing hard links to directories could create cycles, making the filesystem a graph instead of a tree, which breaks tools like `fsck` and makes traversals ambiguous.

Effect on Deletion:

- The file's data is only deleted when the last hard link is removed (link count = 0).
- Disk usage is not freed until all hard links are gone.

1.3. Symbolic (Soft) Links

- **Definition**: A symlink is a special file containing a path (string) to another file or directory.
- **Key Facts**:
  - Symlinks have their own inode and metadata.
  - Can point to non-existent targets (dangling symlinks).
  - Can be relative or absolute.
  - Symlinks to directories are allowed.
  - Symlinks can cross filesystem boundaries.

Command:

```
ln -s target symlink
```

Type	Starts with	Interpreted From
Relative	No `/`	Directory containing link
Absolute	`/`	Root `/`

Examples:

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

Table: Hard Links vs Symbolic Links

Feature	Hard Link	Symbolic Link
Inode	Same as target	Own inode
Can cross FS?	No	Yes
Can link dirs?	No	Yes
Dangling?	No	Yes

Feature	Hard Link	Symbolic Link
Link count	Increases target	Own link count
Affected by mv?	No (same inode)	Yes (may dangle)

ASCII Diagram: Hard vs Symbolic Links

```
# Hard Link
/home/user/file1  ----+         (inode 97311)
/home/user/file2  ----+

# Symbolic Link
/home/user/file1  (inode 97311)
/home/user/link1  (inode 97312, contents: 'file1')
```

Why Two Kinds of Links?

- Hard links: For multiple names to the same file (safe renaming, backup, etc.).
- Symbolic links: For linking to directories, creating links to files that may not exist yet, and for more flexible naming (including dangling links).

Restrictions

- **No hard links to directories** (to prevent cycles and confusion in the filesystem).
- **Symlinks can point to directories or non-existent files**.

Symlink and File Operations:

- `rm symlink` removes the link, not the target.
- `rm file` removes the file (if no other hard links).
- `mv symlink newname` renames the link, not the target.
- `cp symlink newcopy` (with `-P` or `-L`) can copy the link or the target, depending on options.

---

2. Pathname Resolution

2.1. Absolute vs. Relative Paths

- **Absolute**: Starts with `/`, resolution begins at root.
- **Relative**: Does not start with `/`, resolution begins at current working directory (CWD).
- Each process has its own CWD, which can be changed with `cd`.

Example:

```
cd dirname
```

2.2. Name Resolution Algorithm (namei)

- Start at `/` if absolute, else at CWD.
- For each path component:
  - If it's a directory, move into it.
  - If it's a symlink, substitute the symlink's contents into the path:
    - If the symlink is **relative**, interpret from the directory containing the link.
    - If **absolute**, restart from root.
- Continue until the final component is resolved to an inode.
- If a symlink is encountered, the system increments a symlink traversal counter; if the limit (e.g., 20) is exceeded, resolution fails with an error.

ASCII Flowchart: Pathname Resolution

```
START
  |
  v
Is path absolute?
  |--Yes--> Start at /
  |--No-->  Start at CWD
  |
  v
```

```
For each component:
  |
  v
Is component a symlink?
  |--No--> Move into directory
  |--Yes-->
      Is symlink absolute?
        |--Yes--> Restart from /
        |--No-->  Substitute relative path
  |
  v
Repeat until end or symlink limit reached
```

Example: Complex Path with Symlinks
Suppose:

- `/home/user/docs` is a symlink to `/mnt/storage/docs`
- `/mnt/storage/docs/report` is a symlink to `../archive/report`

Resolving `/home/user/docs/report`:

1. `/home/user` (directory)
2. `docs` (symlink) → `/mnt/storage/docs`
3. `/mnt/storage/docs` (directory)
4. `report` (symlink) → `../archive/report`
5. `/mnt/storage/archive/report` (final target)

2.3. Symlink Loops and Dangling Links

- **Dangling symlink**: Points to a non-existent target.

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

- **Symlink loop**: Two or more symlinks refer to each other, causing infinite resolution.

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

- **System limit**: To prevent infinite loops, the system limits symlink traversals (e.g., 20 in Linux). Exceeding this gives an error: `Too many levels of symbolic links`.

2.4. Hard Links to Symlinks

- You can create hard links to a symlink file itself (not to its target).
- All hard links to a symlink share the same inode and link count.
- Removing one hard link does not delete the symlink if others remain.

2.5. Relative Symlinks and Directory Context

- Relative symlinks are interpreted from the directory containing the symlink, not necessarily from where you access it.
- This can cause a symlink to work in one context but not another.

Example:

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

- `e/.baz` may resolve correctly if accessed from `e/`, but not from the parent directory.

---

3. The Shell

3.1. Purpose and Philosophy

- The shell is a simple scripting language for launching/configuring other programs.

- Its main job: set up the environment (I/O, variables, etc.) and run programs.
- Not intended for complex computation (use Python/C for that).
- **Types of shells:** Common ones include `sh`, `bash`, `zsh`, `csh`, `ksh`. Each has its own features and syntax quirks.

Table: Shell Scripting vs. Other Scripting Languages

Feature	Shell Script	Python	Perl
Syntax	Minimal	Rich	Rich
System control	Excellent	Good	Good
Text processing	Basic	Excellent	Excellent
Portability	High	Medium	Medium
Computation	Weak	Strong	Strong

3.2. Command Execution and Standard I/O

- **Redirection**:
  - `>`: Redirect stdout
  - `<`: Redirect stdin
  - `2>`: Redirect stderr
- **Pipes**: `|` connects stdout of one command to stdin of the next.

File Descriptors:

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

Diagram: Pipe Connecting Processes

```
[cmd1] --stdout--> [PIPE] --stdin--> [cmd2]
```

Example:

```
cat file > output.txt 2> errors.log
cmd1 | cmd2 | cmd3
```

3.3. Argument Passing

- Arguments are passed as strings to the program's `main` function.
- `argv[0]` is the command name, `argv[1]` is the first argument, etc.

3.4. Tokenization and Quoting

- The shell splits input into tokens (words) using whitespace and special characters.
- **Special characters**:
  - Whitespace (space, tab, newline)
  - `<`, `>`, `|`, `&`, `;`, `#`, `$`, `` ` ``, `(`, `)`
- **Quoting**:
  - `\char`: Escape a single character
  - `'text'`: Single quotes, literal string (no expansion)
  - `"text"`: Double quotes, allows variable/command expansion

Table: Quoting Effects

Syntax	Variable Expansion	Command Substitution	Field Splitting
'text'	No	No	No
"text"	Yes	Yes	No
unquoted	Yes	Yes	Yes

Examples:

```
echo 'This is $HOME'  # Literal
```

```
echo "This is $HOME"  # Expands $HOME
```

- Newlines can be included in double or single quoted strings.
- **Null bytes**: The shell does not allow embedded null bytes in variables or strings.

Subtle Quoting Points

- To include a single quote inside a single-quoted string, close the quote, escape the quote, and reopen:

  ```
  echo 'Three o'\''clock'
  ```

- Quoting affects argument splitting: `cat $v` vs `cat "$v"` (unquoted may split on spaces, quoted preserves as one argument).

---

4. Reserved Words and Control Structures

4.1. Reserved Words

- Only reserved at the start of a command.
- Examples: `if`, `then`, `else`, `fi`, `for`, `in`, `do`, `done`, `while`, `until`, `case`, `esac`, `{`, `}`, `!`

4.2. Conditionals and Loops

Table: Shell Control Structures

Structure	Syntax Example
if	if ...; then ...; else ...; fi
while	while ...; do ...; done
for	for x in ...; do ...; done
case	case x in ... esac
group	{ ...; }
negate	! command

If Statement

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

- Exit status: 0 = success, nonzero = failure (opposite of C/C++ convention).

While Loop

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

For Loop

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

```
done
```

- The control variable takes each value in the list.
- Commonly used with expansions (e.g., `for f in *; do ...`).

Case Statement

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

- Patterns can be arbitrary (not just file extensions).
- If no pattern matches, exit status is 0 (success).

Grouping and Negation

- `{ ... }`: Group commands as a single command (semicolon required before `}` if on same line).
- `!`: Negate the exit status of a command.

---

5. Variables and Parameters

5.1. Special Variables

- `$0`: Script/command name
- `$1`, `$2`, ...: Positional arguments
- `$#`: Number of arguments
- `$*`: All arguments as a single word
- `$@`: All arguments as separate words

Usage	Result
"$*"	All args as a single string
"$@"	All args, each as a word

- Quoting matters: `"$@"` preserves argument boundaries, `"$*"` joins them into one string.

5.2. Default and Conditional Substitution

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

Examples:

```
name=${USER:-guest}   # Use $USER if set, else 'guest'
file=${1:?Must provide a filename}
```

5.3. Unsetting and Exporting

- `unset x`: Unset variable x
- `export VAR=value`: Make variable available to subcommands (environment variable)

5.4. The Environment

- The shell maintains a set of environment variables (name-value pairs) inherited by child processes.
- Use `export` to add variables to the environment.
- **Shell variables** are local to the shell; **environment variables** are inherited by child processes.

5.5. Special Built-in Variables

- `$$`: PID of the current shell
- `$!`: PID of the last background command

6. Redirection, Pipes, and Concurrency

6.1. Standard File Descriptors

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

6.2. Redirection

- `2>file`: Redirect stderr to file
- `1>&2`: Redirect stdout to stderr
- `command &`: Run in background

6.3. Advanced Redirection and Pipes

- You can redirect different file descriptors in a pipeline:

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

- To merge stderr and stdout:

```
command 2>&1
```

- To split output to multiple places, use `tee`:

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

6.4. Multiple Writers/Readers and Race Conditions

- If multiple processes read from or write to the same pipe, the data is split in a race condition (nondeterministic).
- Example:

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

- Avoid multiple writers/readers unless you want nondeterministic behavior.

Diagram: Multiple Writers/Readers

```
[Writer1]---+
            |---> [PIPE] ---> [Reader1]
[Writer2]---+               [Reader2]
```

- If a process closes its end of a pipe, the other end receives EOF (readers) or SIGPIPE (writers).

6.5. Pipes as Bounded Buffers

- Pipes are implemented as bounded buffers in the kernel (size varies by OS).
- If the buffer is full, the writer blocks; if empty, the reader blocks.
- No overflow: the OS suspends the process if it tries to write more than the buffer can hold.

ASCII Diagram: Pipe

```
[Program A] --writes--> [PIPE (buffer)] --reads--> [Program B]
```

7. Shell Expansion Rules

Order of processing:

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

Table: Shell Expansion Steps and Examples

Step	Example	Result/Effect
Tilde expansion	`echo ~`	`/home/user`
Parameter expansion	`echo $USER`	`dsun`
Command substitution	`echo $(date)`	`Mon Jun 3 ...`
Arithmetic expansion	`echo $((1+2))`	`3`
Field splitting	`set -- a b c`	3 args: a, b, c
Pathname expansion	`ls *.c`	`foo.c bar.c`
Redirection	`echo hi > out.txt`	Output to file

Example:

```
echo $((1 + 2)) # => 3
ls *.c          # Lists all .c files
```

- If no files match a glob pattern, the pattern remains unexpanded (unless `nullglob` is set in some shells).

---

8. Summary

This lecture covered the structure and function of POSIX file systems, focusing on file naming, hard and symbolic links, and the resolution of file names. It explained the shell's role as a scripting and coordination tool, including command execution, redirection, quoting, reserved words, variables, environment, process management, pipes, and the order of shell expansions. Special attention was paid to subtle points like symlink loops, quoting nuances, and race conditions in pipes.