

GNU Diff and Patch

LinuXpert Systems

WhatsApp: 98841 65649, 90808 01382

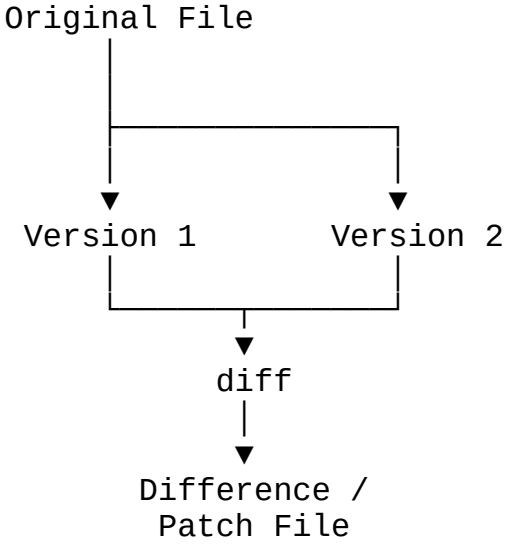
E-mail: baskar@linuxpert.in

1. diff

The **`diff` (difference)** command is a Linux/Unix utility used to **compare two files or two directory trees and report the differences between them**.

It is one of the fundamental tools in Unix/Linux software development and is closely associated with the **`patch`** command.

A typical workflow is:



For example:

```
diff old.c new.c
```

can show which lines were added, removed, or changed.

`Diff` is useful for:

- Comparing source-code versions
- Finding configuration changes
- Generating patch files
- Reviewing changes
- Comparing directories
- Debugging
- Software maintenance
- Version-control workflows
- Linux kernel and open-source development

Syntax

The basic syntax is:

```
diff [OPTIONS] FILE1 FILE2
```

For example:

```
diff old.txt new.txt
```

To compare directories:

```
diff [OPTIONS] DIRECTORY1 DIRECTORY2
```

For example:

```
diff -r project-old project-new
```

1. Compare two files

Suppose **`old.txt`** contains:

```
Linux
Unix
Windows
```

and **`new.txt`** contains:

```
Linux
Unix
FreeBSD
Windows
```

Run:

```
diff old.txt new.txt
```

You may get:

```
3a4
> FreeBSD
```

This means that a line should be **added after line 3** of the first file.

---

2. Compare two identical files

```
diff file1.txt file2.txt
```

If the files are identical, `diff` normally produces **no output**.

You can instead use:

```
diff -s file1.txt file2.txt
```

which reports:

```
Files file1.txt and file2.txt are identical
```

The **`-s`** option means **report when files are identical**.

3. Compare files in normal format

Consider:

```
old.txt
```

```
Linux
Unix
Windows
```

and:

```
new.txt
```

```
Linux
Unix
FreeBSD
Windows
```

Run:

```
diff old.txt new.txt
```

Output:

```
3a4
> FreeBSD
```

The notation:

```
3a4
```

can be interpreted as:

```
3 → line in the first file
a → add
4 → line in the second file
```

Common operations are:

```
a → add
d → delete
c → change
```

For example:

```
3d2
```

means delete a line from the first file.

4. Use unified format

One of the most important options is:

```
diff -u old.txt new.txt
```

This produces **unified diff format**.

Example:

```
--- old.txt
+++ new.txt
@@ -1,3 +1,4 @@
 Linux
 Unix
+FreeBSD
 Windows
```

Unified format is particularly important because it is the format commonly used for **patch files**.

---

5. Generate a patch file

You can redirect the output of `diff` into a file:

```
diff -u old.c new.c > changes.patch
```

Now:

```
changes.patch
```

contains the differences.

It can subsequently be applied using:

```
patch < changes.patch
```

Thus:

```
old.c
  │
  │ diff -u
  ▼
changes.patch
  │
  │ patch
  ▼
new.c
```

This `diff` → `patch` relationship is one of the most important uses of `diff`.

6. Compare directories

Use:

```
diff -r directory1 directory2
```

The `-r` option means **recursive**.

For example:

```
diff -r project-old project-new
```

This compares files throughout both directory trees.

It can report:

```
Only in project-new: README.md
diff project-old/src/main.c project-new/src/main.c
Only in project-old: old.c
```

This is useful for comparing two versions of an entire software project.

---

7. Compare directories while ignoring certain files

You can use:

```
diff -r --exclude='*.o' project-old project-new
```

This excludes object files such as:

```
main.o
network.o
```

You can exclude multiple patterns:

```
diff -r \
  --exclude='*.o' \
  --exclude='*.a' \
  --exclude='*.so' \
  project-old project-new
```

This is particularly useful for source-code directories containing compiled artifacts.

8. Ignore whitespace differences

Use:

```
diff -w file1.txt file2.txt
```

The `-w` option ignores **all whitespace differences**.

For example:

```
printf("Hello");
```

and:

```
printf( "Hello" );
```

may be considered equivalent for the purposes of this comparison.

---

9. Ignore changes in the amount of whitespace

Use:

```
diff -b file1.txt file2.txt
```

The `-b` option ignores changes in the amount of whitespace.

This differs from `-w`, which ignores whitespace more broadly.

---

10. Ignore blank lines

Use:

```
diff -B file1.txt file2.txt
```

The `-B` option ignores changes involving blank lines.

This is useful when comparing documents or source code where blank-line formatting has changed.

11. Ignore case differences

Use:

```
diff -i file1.txt file2.txt
```

For example:

```
Linux
```

and:

```
linux
```

are treated as equivalent for comparison purposes.

The `-i` option means **ignore case differences**.

---

12. Ignore whitespace and case

You can combine options:

```
diff -iw file1.txt file2.txt
```

This ignores:

- Case differences
- Whitespace differences

---

13. Side-by-side comparison

Use:

```
diff -y old.txt new.txt
```

This displays the files side by side.

Example:

```
Linux                 | Linux
Unix                  | Unix
Windows               | FreeBSD
                      > Windows
```

This can be much easier to read for small files.

14. Control side-by-side width

Use:

```
diff -y -W 120 old.txt new.txt
```

where:

```
-W 120
```

sets the output width to approximately 120 columns.

You can also use:

```
diff --width=120 --side-by-side old.txt new.txt
```

---

15. Show only whether files differ

Use:

```
diff -q file1.txt file2.txt
```

Example:

```
Files file1.txt and file2.txt differ
```

This is useful in shell scripts when you only care whether two files are different.

---

16. Use `diff` in a shell script

`diff` provides an exit status that is particularly useful for automation.

For example:

```
diff file1.txt file2.txt
echo $?
```

Typically:

```
0 → Files are identical
1 → Files are different
2 → Error occurred
```

This allows scripts to test files:

```
if diff -q file1.txt file2.txt > /dev/null
then
    echo "Files are identical"
else
    echo "Files are different"
fi
```

This is an important example of how Unix commands communicate results through **exit status codes**.

---

17. Compare configuration files

`diff` is particularly useful for finding changes in configuration files.

For example:

```
diff /etc/config.old /etc/config.new
```

A more readable form is:

```
diff -u /etc/config.old /etc/config.new
```

This makes it easy to identify:

- Added settings
- Removed settings
- Modified parameters
- Changed paths
- Changed permissions/configuration values

18. Compare source-code versions

Suppose:

```
program-v1.c
program-v2.c
```

Run:

```
diff -u program-v1.c program-v2.c
```

Example:

```
--- program-v1.c
+++ program-v2.c
@@ -10,7 +10,7 @@
 int main()
 {
-    printf("Version 1\n");
+    printf("Version 2\n");
     return 0;
 }
```

The important notation is:

```
- → Line removed from the first version
+ → Line added in the second version
```

This makes unified diff very suitable for **source-code review**.

---

19. Compare two versions of a program

Suppose:

```
program-v1/
program-v2/
```

contain two versions of the same application.

Use:

```
diff -ru program-v1 program-v2
```

Here:

```
-r → recursive
-u → unified format
```

You can save the result:

```
diff -ru program-v1 program-v2 > program.patch
```

Now:

```
program.patch
```

contains the changes between the two versions.

---

20. Use `diff -ruN` for directory patches

A particularly useful command is:

```
diff -ruN project-old project-new > project.patch
```

The options mean:

```
-r → Recursive
-u → Unified format
-N → Treat absent files as empty
```

`-N` is useful when comparing directory trees because it allows the differences to represent **newly created or deleted files**.

The resulting patch can often be applied with:

```
patch -p1 < project.patch
```

depending on the paths contained in the patch.

---

21. Compare files while ignoring trailing spaces

GNU `diff` provides:

```
diff -Z file1.txt file2.txt
```

The `-Z` option ignores changes in the amount of **white space at line end**.

This is useful when differences are caused only by trailing spaces.

22. Ignore blank space changes and blank lines

You can combine:

```
diff -wB file1.txt file2.txt
```

This ignores:

```
-w → whitespace differences
-B → blank-line differences
```

This can be useful when comparing source files whose formatting has changed without substantive code changes.

Important `diff` Options

Option	Purpose	Example
`-u`	Unified format	`diff -u old new`
`-r`	Compare directories recursively	`diff -r dir1 dir2`
`-N`	Treat missing files as empty	`diff -N old new`
`-q`	Report only whether files differ	`diff -q a b`
`-s`	Report identical files	`diff -s a b`
`-y`	Side-by-side comparison	`diff -y a b`
`-i`	Ignore case	`diff -i a b`
`-w`	Ignore whitespace	`diff -w a b`
`-b`	Ignore changes in whitespace amount	`diff -b a b`
`-B`	Ignore blank lines	`diff -B a b`
`-Z`	Ignore trailing whitespace changes	`diff -Z a b`
`-c`	Context diff format	`diff -c a b`
`--color`	Colorize output when supported	`diff --color=auto a b`

`diff` Output Formats

There are several important output formats.

Normal format

```
diff file1 file2
```

Unified format

```
diff -u file1 file2
```

Context format

```
diff -c file1 file2
```

Side-by-side format

```
diff -y file1 file2
```

For modern software development, **unified format (`-u`) is especially important**.

Unified Diff Format

Consider:

```
diff -u old.txt new.txt
```

Output:

```
--- old.txt
+++ new.txt
@@ -1,4 +1,5 @@
 Linux
 Unix
+FreeBSD
 Windows
```

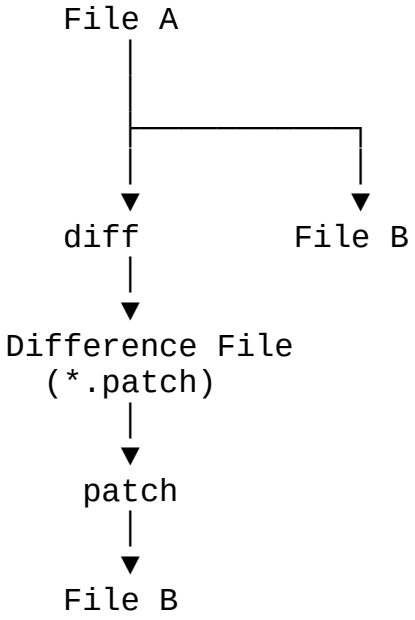
The structure is:

```
--- → Original file
+++ → New file
@@ → Hunk information
-   → Removed line
+   → Added line
```

The surrounding unchanged lines provide **context**, allowing tools such as **`patch`** to locate the correct place to apply the change.

`diff` and **`patch`**

These commands are closely related:



For example:

```
diff -u old.c new.c > changes.patch
```

then:

```
patch old.c < changes.patch
```

The combination provides a simple mechanism for distributing source-code changes.

`diff` vs `cmp`

Both compare files, but they serve different purposes.

Command	Purpose
`diff`	Shows line-oriented differences
`cmp`	Compares files byte-by-byte
`comm`	Compares sorted files line-by-line
`sdiff`	Interactive/side-by-side comparison
`git diff`	Compares versions in a Git repository

For example:

```
cmp file1 file2
```

is useful when you simply want to know whether two files differ at the byte level.

Whereas:

```
diff -u file1 file2
```

explains **what changed**.

`diff` vs `git diff`

In a Git repository:

```
git diff
```

is usually more convenient because Git understands:

- Commits
- Branches
- Staging
- Renames
- Repository history
- File states

However:

```
diff -u file1 file2
```

has an important advantage: it works **independently of any version-control system**.

`diff` and Open-Source Development

The traditional Unix/open-source workflow was heavily based on:

```
Source Version 1
      |
      ▼
     diff
      |
      ▼
  patch file
      |
      ▼
   Maintainer
      |
      ▼
    patch
      |
      ▼
Source Version 2
```

This workflow has historically been important in projects such as the **Linux kernel** and many other open-source projects.

Today, Git has replaced much of the manual `diff`/`patch` workflow, but unified diff remains a fundamental representation of source-code changes.

Practical Laboratory Exercise

A useful engineering-student exercise is to create two versions of a C program.

Step 1 - Create Version 1

```
cp program.c program-v1.c
```

Modify `program.c` to create Version 2:

```
cp program.c program-v2.c
```

Step 2 - Compare them

```
diff program-v1.c program-v2.c
```

Step 3 - Use unified format

```
diff -u program-v1.c program-v2.c
```

Step 4 - Save the differences

```
diff -u program-v1.c program-v2.c > program.patch
```

Step 5 — Inspect the patch

```
less program.patch
```

Step 6 — Restore Version 1

```
cp program-v1.c program.c
```

Step 7 — Test the patch

```
patch --dry-run program.c < program.patch
```

Step 8 — Apply the patch

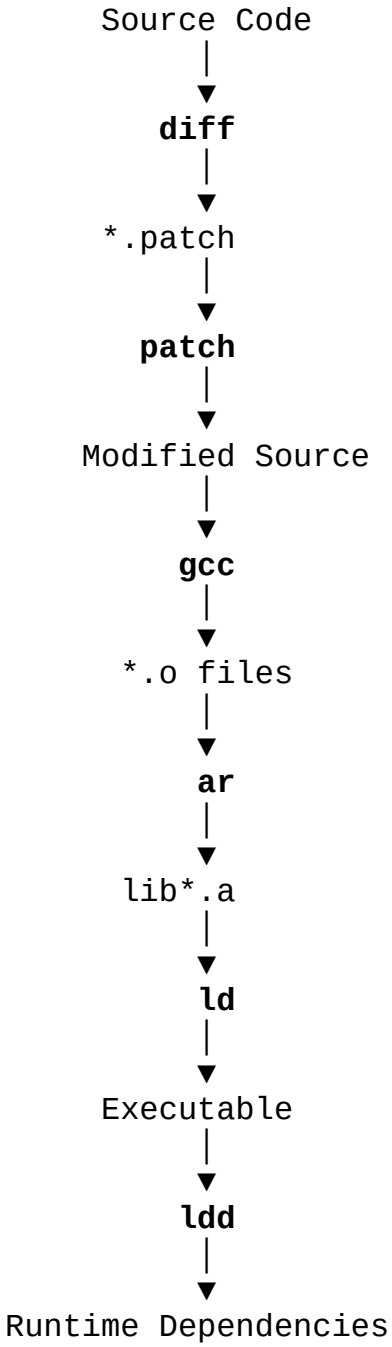
```
patch program.c < program.patch
```

This demonstrates the complete:

Edit
↓
Compare
↓
diff
↓
Patch
↓
patch
↓
Updated Source
↓
gcc
↓
Executable

`diff` in the Linux Software Development Toolchain

`diff` fits naturally into a broader set of Linux development tools:



Why `diff` is important for engineering students ?

`diff` is an excellent command for teaching the fundamental concept of **representing software changes as data**.

It helps students understand:

- File comparison
- Source-code changes
- Unified diff format
- Patch generation
- Patch application
- Recursive directory comparison
- Software maintenance
- Code review
- Configuration management
- Version-control concepts
- Open-source development workflows

2. patch

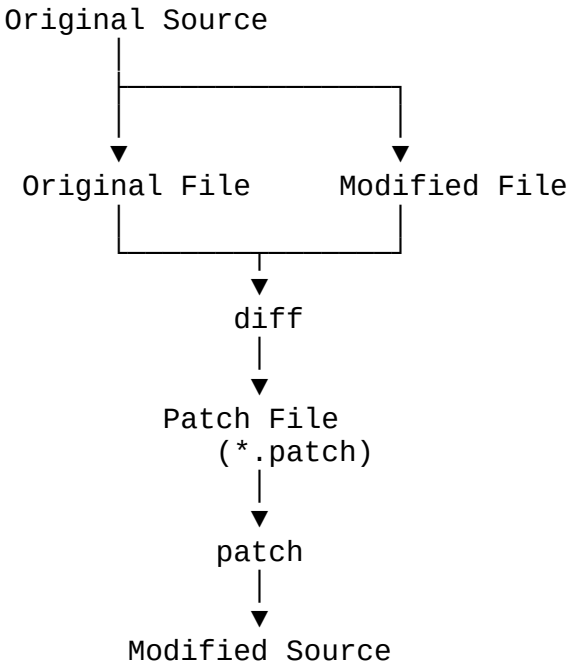
The **`patch`** command is a Linux/Unix utility used to **apply changes from a patch file to source code, configuration files, or other text files**.

A patch file normally contains the differences between an **original file** and a **modified file**. Instead of distributing the entire modified file, developers can distribute only the changes.

`patch` is therefore an important tool for:

- Software development
- Linux kernel development
- Open-source projects
- Source-code maintenance
- Bug fixing
- Code review workflows
- Applying security fixes
- Maintaining different versions of software

A typical workflow is:



Syntax

The most common syntax is:

```
patch [OPTIONS] [ORIGINAL_FILE [PATCH_FILE]]
```

For a patch supplied through standard input:

```
patch [OPTIONS] ORIGINAL_FILE < PATCH_FILE
```

A very common form is:

```
patch < changes.patch
```

The `patch` command examines the patch and determines which file(s) should be modified.

---

1. Create a simple patch

Suppose we have:

```
hello.c
```

Original version:

```
#include <stdio.h>

int main()
{
    printf("Hello\n");
    return 0;
}
```

After modifying it:

```
#include <stdio.h>

int main()
{
    printf("Hello Linux\n");
    return 0;
}
```

We can create a patch using:

```
diff -u hello.c hello-new.c > hello.patch
```

The resulting **`hello.patch`** contains only the differences.

2. Examine a patch file

A patch file is normally a text file.

Display it with:

```
cat hello.patch
```

or:

```
less hello.patch
```

A unified patch may look like:

```
--- hello.c
+++ hello-new.c
@@ -3,5 +3,5 @@
 int main()
 {
-    printf("Hello\n");
+    printf("Hello Linux\n");
     return 0;
 }
```

The important symbols are:

```
---   Original file
+++   Modified file
@@    Location of the change
-     Line removed
+     Line added
```

---

3. Apply a patch

Suppose:

```
hello.c
hello.patch
```

are available.

Apply the patch with:

```
patch < hello.patch
```

The original:

```
printf("Hello\n");
```

becomes:

```
printf("Hello Linux\n");
```

The important point is that `patch` **modifies the original file according to the instructions contained in the patch file**.

---

4. Apply a patch to a specific file

You can explicitly specify the file:

```
patch hello.c < hello.patch
```

This is useful when the patch does not contain sufficient filename information or when you want to control exactly which file is modified.

---

5. Use a patch generated with `diff -u`

The most common combination is:

```
diff -u original.c modified.c > changes.patch
```

Then:

```
patch original.c < changes.patch
```

Conceptually:

```
original.c
    │
    │ diff -u
    ▼
changes.patch
    │
    │ patch
    ▼
modified original.c
```

This **`diff`** + **`patch`** workflow is fundamental to understanding traditional Unix software development.

6. Apply a patch in reverse

Suppose a patch has already been applied:

```
patch < changes.patch
```

You can reverse it using:

```
patch -R < changes.patch
```

`-R` means **reverse**.

For example:

```
Original
   │
   │ patch
   ▼
Modified
   │
   │ patch -R
   ▼
Original
```

This is particularly useful for testing or undoing a patch.

---

7. Test a patch without modifying files

Before applying a patch, you can perform a dry run:

```
patch --dry-run < changes.patch
```

or:

```
patch -t --dry-run < changes.patch
```

The dry run checks whether the patch can be applied without actually modifying the files.

This is extremely useful when working with large source trees.

A good workflow is:

```
patch --dry-run < changes.patch
```

```
patch < changes.patch
```

8. Create a backup before patching

By default, `patch` may create backup files depending on the situation and implementation.
You can explicitly request backup behavior with:

```
patch -b < changes.patch
```

For example:

```
hello.c
hello.c.orig
```

The original version may be retained as a backup.

---

9. Specify the backup suffix

You can use:

```
patch -b -z .bak < changes.patch
```

This can produce a backup such as:

```
hello.c.bak
```

The exact backup behavior can depend on the patch format and options.

---

10. Patch multiple files

A patch file can contain changes for multiple files.

For example:

```
src/main.c
src/network.c
src/config.h
```

A single patch can contain changes to all three.

Apply it with:

```
patch < project.patch
```

The patch utility processes each file described in the patch.

This is one of the major advantages of patch files: **a single small file can represent changes across an entire source tree**.

11. Patch an entire directory

Suppose a patch was created using:

```
diff -ruN old-project new-project > project.patch
```

The resulting patch may contain paths such as:

```
old-project/main.c
new-project/main.c
```

To apply it:

```
patch -p1 < project.patch
```

The `-p` option is particularly important when working with directory-level patches.

---

12. Understand the `-p` option

Consider a patch containing:

```
--- a/src/main.c
+++ b/src/main.c
```

If you are already inside the project directory:

```
project/
├── src/
│   └── main.c
```

you would commonly use:

```
patch -p1 < changes.patch
```

Why?

```
a/src/main.c
│
└── remove first path component
      ↓
src/main.c
```

Thus:

```
-p0 → remove 0 path components
-p1 → remove 1 path component
-p2 → remove 2 path components
```

This is extremely common when applying patches from Git-based projects.

13. Create a directory patch

Suppose:

```
project-old/
project-new/
```

contain two versions of a project.

Create a recursive unified patch:

```
diff -ruN project-old project-new > project.patch
```

Here:

```
-r → recursive
-u → unified format
-N → treat absent files as empty
```

You can then apply it to the old project:

```
patch -p1 < project.patch
```

This is a classic Unix technique for distributing source-code changes.

---

14. Patch a newly created file

A patch can also represent the creation of a new file.

For example:

```
--- /dev/null
+++ b/newfile.c
@@ ...
```

Applying the patch can create:

```
newfile.c
```

Similarly, patches can represent deleted files.

15. Handle rejected patches

Sometimes a patch cannot be applied cleanly.

For example:

```
Hunk #1 FAILED at 25.
1 out of 1 hunk FAILED -- saving rejects to file.c.rej
```

The patch may create:

```
file.c.rej
```

The `.rej` file contains the portion of the patch that could not be applied.

You should then inspect:

```
cat file.c.rej
```

and compare it with:

```
diff -u file.c.rej file.c
```

or manually inspect the relevant source code.

---

16. Understand "Hunks"

A **hunk** is a group of related changes within a patch.

For example:

```
@@ -20,7 +20,8 @@
```

identifies a hunk and provides information about where the change occurs.

A patch might contain:

```
Hunk #1
Hunk #2
Hunk #3
```

If the surrounding source code has changed significantly, one or more hunks may fail.

17. Patch with `-p1`

A very common real-world command is:

```
patch -p1 < fix.patch
```

This is particularly common for patches containing Git-style paths:

```
--- a/file.c
+++ b/file.c
```

For example:

```
cd project
patch -p1 < ../fix.patch
```

The `-p1` removes:

```
a/
```

or:

```
b/
```

from the paths.

---

18. Patch interactively

If the patch does not apply cleanly, `patch` may ask what to do.

For example:

```
Apply anyway? [y]
```

You may also use:

```
patch -i changes.patch
```

where `-i` specifies the patch input file.

Important `patch` Options

Option	Purpose	Example
`-pN`	Strip N leading path components	`patch -p1 < fix.patch`
`-R`	Reverse the patch	`patch -R < fix.patch`
`--dry-run`	Test without modifying files	`patch --dry-run < fix.patch`
`-b`	Create backups	`patch -b < fix.patch`
`-z`	Set backup suffix	`patch -b -z .bak < fix.patch`
`-i`	Read patch from file	`patch -i fix.patch`
`-o`	Write output to a file	`patch -o new.c old.c < fix.patch`
`-N`	Ignore already-applied/reversed patches	`patch -N < fix.patch`
`-t`	Suppress questions / assume defaults	`patch -t < fix.patch`
`-f`	Force patching	`patch -f < fix.patch`
`-d DIR`	Change to directory first	`patch -d project < fix.patch`
`--verbose`	Produce detailed output	`patch --verbose < fix.patch`

---

19. `-N` — Avoid Reapplying a Patch

Suppose:

```
patch < fix.patch
```

has already been executed.

Running it again may cause the patch to be recognized as reversed/already applied.

You can use:

```
patch -N < fix.patch
```

to tell `patch` not to apply a patch that appears to be reversed or already applied.

This can be useful in automated scripts.

20. Specify the Working Directory

Instead of:

```
cd project
patch -p1 < ../fix.patch
```

you can use:

The `-d` option tells `patch` to change to the specified directory before applying the patch.

---

21. Generate a patch using `diff`

`patch` is commonly paired with **`diff`**.

Create two versions:

```
file-old.txt
file-new.txt
```

Generate:

```
diff -u file-old.txt file-new.txt > file.patch
```

Apply:

```
patch file-old.txt < file.patch
```

This gives the fundamental Unix workflow:

```
diff
  │
  │ Compare two versions
 ▼
patch file
  │
  │ Apply differences
 ▼
Updated file
```

22. `patch` and Git

Modern software development often uses Git instead of manually managing patch files.

For example:

```
git diff > changes.patch
```

creates a patch containing Git changes.

It can then be applied with:

```
patch -p1 < changes.patch
```

However, Git also provides its own commands:

```
git apply changes.patch
```

and:

```
git am changes.patch
```

The distinction is useful:

Tool	Purpose
`diff`	Generate differences
`patch`	Apply traditional patch files
`git diff`	Generate Git-format differences
`git apply`	Apply a Git patch
`git am`	Apply patches as Git commits

---

23. `patch` vs `git apply`

For a simple patch:

```
patch -p1 < fix.patch
```

For a Git repository:

```
git apply fix.patch
```

`git apply` understands Git-specific patch information and integrates more naturally with Git workflows.

Nevertheless, `patch` remains important because it is:

- Portable
- Old and well-established
- Available on virtually every Unix/Linux system
- Independent of Git
- Widely used in traditional source-distribution workflows

---

24. Applying a Linux Kernel Patch

One of the classic uses of `patch` is modifying Linux kernel source code.

A simplified workflow is:

```
cd linux-source
patch -p1 < kernel-fix.patch
```

The patch may contain changes to files such as:

```
drivers/
kernel/
mm/
fs/
net/
include/
```

A patch can therefore modify many files in one operation.

In large kernel-development workflows, however, Git-based tools are now commonly used to manage patches and commits.

---

25. Check a Patch Before Applying It

A good workflow is:

```
patch --dry-run -p1 < fix.patch
```

If the output indicates that the patch can be applied successfully:

```
patch -p1 < fix.patch
```

This is especially important when modifying:

- Operating-system source code
- Production software
- Large source trees
- Configuration files
- Third-party software

---

26. Patch File Lifecycle

A useful conceptual example is:

```
Version 1
    │
    │ modify source
    ▼
Version 2
    │
    │ diff -u
    ▼
patch file
    │
    │ distribute
    ▼
Another system
    │
    │ patch
    ▼
Version 2
```

The patch is therefore a **portable description of changes**, rather than a complete copy of the modified source tree.

`patch` vs `cp`

These commands have fundamentally different purposes.

Command	Purpose
`cp`	Copy an entire file
`patch`	Apply specific changes to an existing file
`diff`	Compare files and produce differences
`rsync`	Synchronize files/directories
`git`	Manage versions and changes

For example:

```
cp new.c old.c
```

replaces the file with a complete copy.

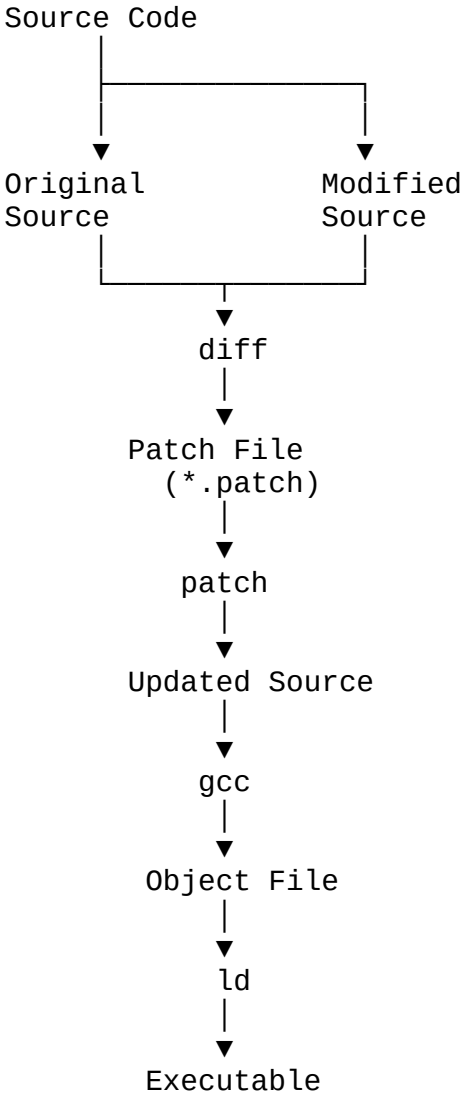
Whereas:

```
patch old.c < changes.patch
```

applies only the changes described by the patch.

`patch` in the Linux Software Development Toolchain

`patch` fits naturally into the traditional Unix development workflow:



This connects **`patch`** with commands such as:

```
diff
patch
gcc
ar
ld
ldd
readelf
objdump
nm
```

Practical Laboratory Exercise

A useful engineering-student exercise is to create two versions of a program.

Step 1 — Create the original

```
cp hello.c hello-old.c
```

Modify `hello.c`.

Step 2 — Generate a patch

```
diff -u hello-old.c hello.c > hello.patch
```

Step 3 — Restore the old version

```
cp hello-old.c hello.c
```

Step 4 — Examine the patch

```
cat hello.patch
```

Step 5 - Test it

```
patch --dry-run hello.c < hello.patch
```

Step 6 - Apply it

```
patch hello.c < hello.patch
```

Step 7 - Verify

```
diff hello.c hello-old.c
```

The final result should show that `hello.c` has been restored to the modified version represented by the patch.

Why `patch` Is Important for Engineering Students ?

`patch` is a particularly useful command for teaching students how **software changes can be represented, distributed, reviewed, and applied without distributing an entire source tree**.

It introduces important concepts such as:

- Difference files
- Unified diff format
- Source-code modification
- Patch hunks
- Context lines
- Patch rejection
- Reverse patches
- Directory-level patches
- Open-source development workflows
- Software maintenance
- Version-control concepts
- Linux kernel development

A useful curriculum sequence would be:

```
diff
  ↓
Generate differences
  ↓
patch
  ↓
Apply differences
  ↓
gcc
  ↓
Compile
  ↓
ar
  ↓
Create library
  ↓
ld
  ↓
Link
  ↓
ldd
  ↓
Inspect runtime dependencies
```

This gives students a much clearer understanding of the **complete Linux software-development lifecycle—from modifying source code to compiling, linking, and executing the resulting program.**