04834580 Software Engineering (Honor Track) 2024-25

UNIX Environment

Sergey Mechtaev
mechtaev@pku.edu.cn
School of Computer Science, Peking University

What is the `diff` utility?

北京大学

- `diff` [1] is a command-line tool used to compare two text files line by line.
- It outputs the differences between the files in a readable format.
- Common application: version control, identifying changes between program versions.

Syntax of `diff`

The basic syntax of the `diff` command:

Syntax

```
diff [options] file1 file2
```

Common options:

- `diff -u`: Produces output in Unified Diff format (preferred for readability).
- `diff -c`: Produces output in Context Diff format.

Example Using `diff`

Let's compare two files, `file1.txt` and `file2.txt`:

Contents of file1.txt:

```
apple
banana
cherry
date
```

Contents of file2.txt:

```
apple
blueberry
cherry
date
```

Running `diff file1.txt file2.txt` produces:

```
2c2
< banana
---
> blueberry
```

What is Unified Diff Format?

北京大學

- Unified Diff is a compact and readable format for showing file differences.
- Used extensively in version control systems like Git.
- Highlights context lines along with changes.

Key Elements of Unified Diff

- `@@`: Denotes the location of changes in the file.
- +: Lines added.
- −: Lines removed.

Unified Diff Example

Using `diff -u file1.txt file2.txt` produces:

```
--- file1.txt
+++ file2.txt
@@ -1,4 +1,4 @@
 apple
-banana
+blueberry
 cherry
 date
```

Explanation:

- `--- file1.txt`: Original file.
- `+++ file2.txt`: Changed file.
- `- banana`: Indicates "banana" was removed.
- `+ blueberry`: Indicates "blueberry" was added.

More Complex Unified Diff Example

北京大学

Contents of file1.txt:

```
apple
banana
cherry
date
elderberry
fig
grape
```

Contents of file2.txt:

```
apple
blueberry
cherry
dragonfruit
elderberry
grape
```

More Complex Unified Diff Example

北京大学

Running `diff -u` outputs:

```
--- file1.txt
+++ file2.txt
@@ -1,7 +1,6 @@
 apple
-banana
+blueberry
 cherry
-date
+dragonfruit
 elderberry
-fig
 grape
```

Interpreting Unified Diff Output

北京大学

Key observations:

- Context lines ("`apple`", "`cherry`", etc.) provide surrounding unchanged text for reference.
- Removed lines are prefixed with "`-`".
- Added lines are prefixed with "`+`".
- "`@@`" block indicates where changes occur in the original file.

Command

```
patch < patch-file
```

- Reads the input `patch-file` and applies changes to the appropriate original file.
- Use the `--dry-run` option to simulate patching without actually making changes:

```
patch --dry-run < patch-file
```

Suppose original.txt has:

Original Content

```
Hello, World!
This is a test file.
```

modified.txt has:

Modified Content

```
Hello, Universe!
This is a test file.
```

Generate the patch file:

Command

```
diff -u original.txt modified.txt > patch.txt
```

Applying the Patch

To update original.txt to match modified.txt:

Command

```
patch original.txt < patch.txt
```

After applying the patch, original.txt will now contain:

Result

```
Hello, Universe!
This is a test file.
```

Myers' Algorithm

- Designed by Eugene Myers in 1986 [2].
- Efficient algorithm to find the minimum edit script using a graph-based approach.
- Analyzes **edit graph** where all paths from $(0, 0)$ to (m, n) represent transformations.

Edit Graph

- Construct a 2D grid:
  - Rows correspond to characters of String A.
  - Columns correspond to characters of String B.
- For each cell (i, j) in the matrix where $i > 0$ and $j > 0$, calculate the edit distance from $A[0:i]$ to $B[0:j]$ using the following rules:
  - If $A[i-1]$ equals $B[j-1]$, the value at (i, j) is the same as the value at $(i-1, j-1)$ (no edit needed).
  - If $A[i-1]$ is not equal to $B[j-1]$, the value at (i, j) is the minimum of:
    - The value above $(i-1, j)$ plus 1 (deletion in A).
    - The value to the left $(i, j-1)$ plus 1 (insertion in A).
    - The value diagonally above-left $(i-1, j-1)$ plus 1 (substitution or mismatch).

Edit Graph

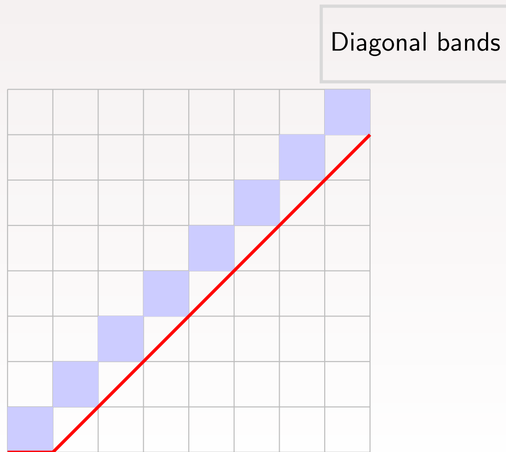
	B[1]	B[2]	B[3]	B[4]	B[5]
A[1]	0,0				
A[2]					
A[3]					
A[4]					
A[5]					

What is a Snake?

- A **snake** is a diagonal segment of the edit graph where no operations are needed (matching characters).
- Myers' algorithm extends paths using snakes whenever possible to reduce computation.

Finding the Shortest Edit Path

- Myers' algorithm divides the problem into **diagonal bands**, making it efficiently find the shortest path.
- Diagonal **k** represents $i - j = k$.



Complexity

- Myers' algorithm runs in $\mathcal{O}(N \times D)$, where:
  - N: Length of the strings.
  - D: Length of the shortest edit script.
- Efficient for real-world applications such as 'diff' and 'git diff'.

The Make Utility

- `make` [3] is a build automation tool used in Unix/Linux systems.
- It is used for compiling programs and managing dependencies.
- A `Makefile` defines build rules, dependencies, and commands.

Basic Syntax of a Makefile

A Makefile consists of:

- **Rules**: Specify targets, dependencies, and commands.
- **Variables**: Represent reusable values.
- **Patterns**: Define generic build rules.

Syntax of a rule:

```
target: dependencies
    command
```

Semantics of Rules

- **Target**: The file or action to create/update.
- **Dependencies**: Files required to build the target.
- **Command**: Shell commands executed to build or update the target.

Example:

```
output.txt: input.txt
    cat input.txt > output.txt
```

Explanation:

- `output.txt` is the target.
- `input.txt` is the dependency.
- The command concatenates the contents of `input.txt` and writes them to `output.txt`.

Using Variables in Makefile

Variables provide reusable values in Makefiles.
Example:

```
CC = gcc
CFLAGS = -Wall

program: program.c
    $(CC) $(CFLAGS) -o program program.c
```

Explanation:

- `CC` sets the compiler (`gcc`).
- `CFLAGS` defines compiler flags (`-Wall` for all warnings).
- `$(CC)` and `$(CFLAGS)` are expanded during execution.

Pattern Rules in Makefile

Pattern rules define generic build instructions.
Example:

```
%.o: %.c
    gcc -c $< -o $@
```

Explanation:

- `%.o`: Target file pattern (object files).
- `%.c`: Dependency file pattern (source files).
- `$<`: The first dependency (a `.c` file).
- `$@`: The target (an `.o` file).
- Compiles `.c` files into `.o` files using the `gcc` compiler.

Phony Targets

Phony targets represent actions, not real files.
Example:

```
.PHONY: clean

clean:
    rm -f *.o program
```

Explanation:

- `.PHONY` marks `clean` as a phony target.
- `clean` will force `make` to execute the command even if a file named `clean` exists.
- Command removes object files and the compiled program.

Understanding Dependencies

Example:

```
all: program

program: program.o utils.o
	gcc -o program program.o utils.o

program.o: program.c
	gcc -c program.c -o program.o

utils.o: utils.c
	gcc -c utils.c -o utils.o
```

Explanation:

- `all` target depends on `program`.
- `program` depends on object files `program.o` and `utils.o`.
- `program.o` depends on `program.c`.
- `utils.o` depends on `utils.c`.

What is `awk`?

- `awk` [4] is a powerful text processing utility in Unix/Linux.
- It is primarily used for pattern matching, processing, and reporting on data.
- `awk` is a programming language as well, with features like:
  - Variables
  - Conditionals
  - Loops
- Commonly used for tasks such as:
  - Extracting columns of data
  - Filtering content based on patterns
  - Performing calculations on data

Basic Syntax of awk

The basic syntax of an awk command is:

Syntax

```
awk 'pattern action ' inputfile
```

- pattern: A condition to match (e.g., a regex or logical test).
- action: Block of code to execute when the pattern matches.
- If no pattern is specified, the action is applied to all lines.
- If no action is specified, matching lines are printed by default.

Components of `awk`

北京大学

- **Patterns:**
  - Define when an action should be applied.
  - Patterns can include:
    - Regular expressions.
    - Relational expressions (e.g., `$1 > 10`).
    - Logical expressions (e.g., `$1 > 10 && $2 < 5`).
- **Actions:**
  - Specify operations to perform when the pattern matches.
  - Actions are enclosed in curly braces {}.
  - Commonly used actions:
    - Print fields using `print`.
    - Perform calculations.
    - Modify fields.
- **Special Patterns:**
  - `BEGIN`: Executes before reading any input.
  - `END`: Executes after processing all input.

Example 1: Printing Specific Columns

Input File (data.txt):

```
John    25   5000
Alice   30   6000
Bob     22   4500
```

Command to print the first and third columns:

```
awk '{ print $1, $3 }' data.txt
```

Output:

```
John 5000
Alice 6000
Bob 4500
```

Example 2: Filtering Rows

北京大学

Input File (`data.txt`):

```
John    25  5000
Alice   30  6000
Bob     22  4500
```

Command to filter rows where the age is greater than 23:

```
awk '$2 > 23 { print $0 }' data.txt
```

Output:

```
John    25  5000
Alice   30  6000
```

Example 3: BEGIN and END

北京大学

Command to calculate and print the sum of the third column in `data.txt`:

```
awk 'BEGIN { sum = 0 }
     { sum += $3 }
     END { print "Total:", sum }' data.txt
```

Output:

```
Total: 15500
```

Explanation:

- `BEGIN`: Initialize the variable `sum` to 0.
- `{ sum += $3 }`: Add the value of the third field to `sum`.
- `END`: Print the final result.

Example 4: Using Regular Expressions

Input File (log.txt):

```
192.168.1.1 - - [01/Jan/2023] "GET /index.html" 200
192.168.1.2 - - [01/Jan/2023] "POST /login" 403
192.168.1.3 - - [01/Jan/2023] "GET /about.html" 200
```

Command to print lines containing GET requests:

```
awk '/GET/ { print $0 }' log.txt
```

Output:

```
192.168.1.1 - - [01/Jan/2023] "GET /index.html" 200
192.168.1.3 - - [01/Jan/2023] "GET /about.html" 200
```

What is jq?

- **jq** [5] is a lightweight and flexible command-line JSON processor.
- It allows you to query, transform, and format JSON data.
- Works by applying filters to JSON data—like SQL for JSON!

Basic Syntax

北京大学

Syntax: `jq '<filter>' <file.json>`

- Filters are expressions that process the JSON input and output the result.
- Example:

```
$ echo '{"name": "Alice", "age": 25}' | jq '.name'
"Alice"
```

Accessing JSON Data

- Use `.key` to access JSON object properties.
- Use `.[index]` to access elements in arrays.
- Example:

```
$ echo '[{"name": "Alice"}, {"name": "Bob"}]' | jq '.[0].
    name'
"Alice"
```

Filters

- Filters are the core feature of **jq**. They transform JSON data.
- Example: Extracting all names from an array.

```
$ echo '[{"name": "Alice"}, {"name": "Bob"}]' | jq '.[] | .
    name'
"Alice"
"Bob"
```

- Use pipes | to chain filters.

```
$ echo '{"a":1,"b":2}' | jq '. | .a'
1
```

- Use `select()` for conditional extraction.

```
$ echo '[1,2,3,4]' | jq '.[] | select(. > 2)'
3
4
```

- Use `map()` for array transformation.

```
$ echo '[1,2,3]' | jq 'map(. * 2)'
[2,4,6]
```

Built-in Functions

北京大学

- **length**: Get the length of an array or string.

```
$ echo '[1,2,3]' | jq 'length'
3
```

- **keys**: Return keys of an object.

```
$ echo '{"a":1,"b":2}' | jq 'keys'
["a", "b"]
```

References I

[1] Free Software Foundation.
Gnu diffutils.
https://www.gnu.org/software/diffutils/, 2025.

[2] Eugene W Myers.
An o (nd) difference algorithm and its variations.
Algorithmica, 1(1):251–266, 1986.

[3] Free Software Foundation.
Gnu make.
https://www.gnu.org/software/make/, 2025.

[4] Free Software Foundation.
Gawk.
https://www.gnu.org/software/gawk/, 2025.

References II

[5] jq Developers.
jq is a lightweight and flexible command-line json processor.
https://jqlang.org/, 2025.