Principles of Computer Science II

Unix Programming Shell

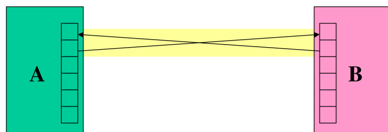
Ioannis Chatzigiannakis

Sapienza University of Rome

Lecture 12

UNIX Pipes

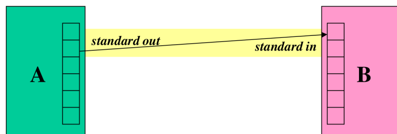
- General idea: The input of one program is the output of the other, and vice versa.



- Both programs run at the same time.

UNIX Pipes

- Often, only one end of the pipe is used.



- This can be done using intermediate files.

UNIX Pipes

- Commands produce an output – using the descriptor > the output is redirected to a file
  ```
  # ls > filelist
  ```
- A new file is created under the name filelist
- If the file already exists, the new file will replace the old one.
- We can use the descriptor >> to redirect the output to an existing file
  ```
  # ls -lt /root/doc >> /root/filelist
  ```
- The commands that require input – using the descriptor < the input is redirected from a file
  ```
  # sort < /root/filelist
  ```

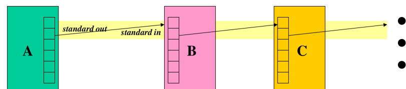
UNIX Pipes

- File approach: Run first program, save output into file.
- Run second program, using file as input.

- Unnecessary use of the disk:
  - Slower,
  - Can take up a lot of space.
- Makes no use of multi-tasking.

UNIX Pipes

- The output of a process is redirected as input to another process.



- The redirection is done using the descriptor |
  `# ls | sort` – *sorting the files of a folder*
  `# ls /root | wc -l` – *counting files*
- Multiple pipes are often chained together.

UNIX Pipes

- What's the difference?
- Both commands send input to command from a file instead of the terminal:
  `# cat file | command`
  `# command < file`
- An extra process !
  `# cat file | command`

`# command < file`

UNIX Pipes

- What if a process tries to read data but nothing is available?
  - UNIX puts the reader to sleep until data available.
- What if a process cannot keep up reading from the process that's writing?
  - UNIX keeps a buffer of unread data.
  - This is referred to as the pipe size.
  - If the pipe fills up, UNIX puts the writer to sleep until the reader frees up space (by doing a read).
- Multiple readers and writers possible with pipes.

UNIX Pipes

- Examples of filters:
  - Sort
    - Input: lines from a file.
    - Output: lines from the file sorted.
  - Grep
    - Input: lines from a file.
    - Output: lines that match the argument.
  - Sed
    - Programmable stream editor.

Processes

- We may execute commands in series by using the delimeter ;
  - Commands are executed one by one. When the first is completed, the next one starts. When the last command is completed, we get a new prompt
  - `# who | sort ; date`
- We may execute commands in the background using the delimeter &
  - The commands are executed and a new prompt is provided immediately
  - `# pr junk | lpr &`
- The execution of a command results to a new process
  - The command *ps* shows up in the list of active processes
  - The command *wait* is active until all the commands executed using the delimeter & complete.

List of processes

```
# ps -a
  PID TTY   TIME CMD
  106  c1   0:01 -sh
 4114  co   0:00 /bin/sh /usr/bin/packman
 2114  co   0:00 -sh
 6762  c1   0:00 ps -a
   87  c2   0:00 getty
   90  c3   0:00 getty
```

- Parameter a – list all the commands created by consoles
- Column PID – unique ID of the process
- Column TTY – the console ID that created the process
- Column TIME – total execution time
- Column CMD – the name of the command

Process management

- To terminate a process we use the command *kill [PID]*
- We may change the priority of a process
  - prefix *nice*
    `# nice pr junk | lpr &`
- We may delay the execution of a command
  - prefix *at*

```
# at 1500
ls -l / /root /dir | wc > allfiles
pr allfiles | lpr ; date > lpr-endtime &
date > lpr-starttime
^D
at: /usr/spool/at/07.111.1500.67 created
#
```

Introduction to Regular Expressions (1)

- A regular expression (regex) describes a set of possible input strings.
- Regular expressions descend from a fundamental concept in
- Computer Science called finite automata theory
- Regular expressions are endemic to Unix
  - vi, ed, sed, and emacs
  - awk, tcl, perl and Python
  - grep, egrep, fgrep
  - compilers

Introduction to Regular Expressions (2)

- The simplest regular expressions are a string of literal characters to match.
- The string matches the regular expression if it contains the substring.

Introduction to Regular Expressions (3)

regular expression → **c k s**

UNIX Tools rocks.

match

UNIX Tools sucks.

match

UNIX Tools is okay.

no match

Introduction to Regular Expressions (4)

- A regular expression can match a string in more than one place.

regular expression → **a p p l e**

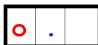
Scrapple from the apple.

match 1 *match 2*

Introduction to Regular Expressions (5)

- The . regular expression can be used to match any character.

regular expression →



For me to poop on.

match 1 *match 2*

Character Classes (1)

- Character classes [] can be used to match any specific set of characters.

regular expression →

beat a brat on a boat

match 1 *match 2* *match 3*

Character Classes (2)

- Character classes can be negated with the [^] syntax.

regular expression →

beat a brat on a boat

match

Character Classes (3)

- [*aeiou*] will match any of the characters a, e, i, o, or u
- [*kK*]orn will match korn or Korn
- Ranges can also be specified in character classes
- [1 – 9] is the same as [123456789]
- [*abcde*] is equivalent to [*a* – *e*]
- You can also combine multiple ranges
- [*abcde*123456789] is equivalent to [*a* – *e*1 – 9]
- Note that the - character has a special meaning in a character class but only if it is used within a range,
- [–123] would match the characters -, 1, 2, or 3

Named Character Classes

- Commonly used character classes can be referred to by name (alpha, lower, upper, alnum, digit, punct, cntrl)
- Syntax [: *name* :]
- $[a-zA-Z]$ is equivalent [[: *alpha* :]]
- $[a-zA-Z0-9]$ is equivalent [[: *alnum* :]]
- $[45a-z]$ is equivalent [45[: *lower* :]]
- Important for portability across languages

Anchor Characters

- Anchors are used to match at the beginning or end of a line (or both).
- ^ means beginning of the line
- $ means end of the line

regular expression → `^ b [eor] a t`

beat a brat on a boat

match (beat)

regular expression → `b [eor] a t $`

beat a brat on a **boat**

match (boat)

`^word$`   `^$`

Repetition

- The * is used to define zero or more occurrences of the single regular expression preceding it.

regular expression → y a * y

I got mail, yaaaaaaaaaay!

match

regular expression → o a * o

For me to poop on.

match

.*

Match Length

- A match will be the longest string that satisfies the regular expression.

regular expression → a . * e

Scrapple from the apple.

no no yes

Repetition Ranges

- Ranges can also be specified
- { } notation can specify a range of repetitions for the immediately preceding regex
- {n} means exactly n occurrences
- {n,} means at least n occurrences
- {n,m} means at least n occurrences but no more than m occurrences
- Example:
  .{0,} same as .*
  a{2,} same as aaa*

Subexpressions

- If you want to group part of an expression so that * or { } applies to more than just the previous character, use ( ) notation
- Subexpresssions are treated like a single character
- a* matches 0 or more occurrences of a
- abc* matches ab, abc, abcc, abccc, . . .
- (abc)* matches abc, abcabc, abcabcabc, . . .
- (abc)2,3 matches abcabc or abcabcabc

Global Regular Expressions Print – grep

- grep comes from the ed (Unix text editor) search command "global regular expression print" or g/re/p
- This was such a useful command that it was written as a standalone utility
- There are two other variants, egrep and fgrep that comprise the grep family
- grep is the answer to the moments where you know you want the file that contains a specific phrase but you can't remember its name

Syntax

- Regular expression concepts we have seen so far are common to grep
- grep: \(and \), \{ and \}

Introduction to sed (1)

sed: Stream Editor:

- Input from a file or from a pipe
- Output to a file or to a pipe
- Filters and edits the input text and produces the modified text as output
- Examines input line-by-line, searches for a pattern and makes a replace
- We usually use it when we know how content is structured (lines, columns)

Introduction to sed (2)

- Sed is very useful for simple operations, such as
  - replace or remove patterns,
  - when the operation is not necessarily related with the formatting of the input.
- We wish to repeat the operation over all the lines of the input text.

Main Concepts

pattern space = the data we wish to edit (data buffer)

while (readline) {

1. read the input one line at a time
2. for each line, sed executes a series of commands on the pattern space
3. outputs the resulting/modified text

}

Command Syntax

```
sed <options> '<address><command>'
```

1. address: the line number of the input text, the pattern to search, contained within slashes (/pattern/). Defines where the command will be applied, in which lines or to all lines.
2. The pattern is described using regular expressions,
3. We can provide a range of lines as comma separated values to execute the command over a given range of lines, including the lines defined.
4. ! = NOT (to apply the command to all lines excluding the range provided)

Common Commands

a\	Insert text after current line
c\	Change current text into (new text)
d	Delete text
i\	Insert text before current line
p	Print text
r	Read file
s	Search and replace text
w	Write to file
-e	To set multiple commands
-f SCRIPT_FILE	To use a sed file with commands
-n	Print only the p commands

Replace

Common usage:

```
sed s 'pattern/replacement/<flags>'
```

- pattern: search pattern
- replacement: the string with which to replace the pattern
- flags (optional):
  - **n** (number): number of occurance to replace
  - **g** (global): replace all occurances
  - **p** (print): print the content of the pattern space

Example file

```
bash-3.1$ cat -n example.sed

1 This is the first line of an example text.
2 It is a text with erors.
3 Lots of erors.
4 So many erors, all these erors are making me sick.
5 This is a line not containing any errors.
6 This is the last line.
```

Usage Example 1

```
bash-3.1$ sed 's/erors/errors/g' example.sed

This is the first line of an example text.
It is a text with errors.
Lots of errors.
So many errors, all these errors are making me sick.
This is a line not containing any errors.
This is the last line.
```

What if we replace the command g with number 2?
What if we remove command g completely?

Example 2

ˆ Start of line - **$** End of line

```
bash-3.1$ sed 's/^/> /' example.sed

> This is the first line of an example text.
> It is a text with erors.
> Lots of erors.
> So many erors, all these erors are making me sick.
> This is a line not containing any errors.
> This is the last line.
```

What if we replace the command of ˆ with $?

Example 3

```
bash-3.1$ sed -e 's/erors/errors/g' -e
                 's/last/final/g' example.sed
(or, alternatively)
sed 's/erors/errors/g; s/last/final/g' example.sed

This is the first line of an example text.
It is a text with errors.
Lots of errors.
So many errors, all these errors are making me sick.
This is a line not containing any errors.
This is the final line.
```

Other special characters

- The characters _ or , (comma) may replace / for improved readability
- \: escape character
- **&** Signifies the pattern found (always referring to the current line)
- Take special care on those symbols that are part of the regular expression

One more example

```
bash-3.1$ sed 's/[^ ][^ ]*/(&)/' example.sed

(This) is the first line of an example text.
(It) is a text with erors.
(Lots) of erors.
(So) many erors, all these erors are making me sick.
(This) is a line not containing any errors.
(This) is the last line.
```

What if the pattern was

`[a-z]\+\.` ?

`s/[^ ]` ?

Yet another example

Print only lines that match the pattern after changing it, based on the conditions set:

```
bash-3.1$ sed -n 's/erors/errors/gp' example.sed

It is a text with errors.
Lots of errors.
So many errors, all these errors are making me sick.
```

What if there was a ! before p (print)?

Focus on specific lines (1)

We may focus the changes only in specific lines, declaring the lines with their number.

```
bash-3.1$ sed '1,3 s/erors/errors/g' example.sed

This is the first line of an example text.
It is a text with errors.
Lots of errors.
So many erors, all these erors are making me sick.
This is a line not containing any errors.
This is the last line.
```

Focus on specific lines (2)

We can do the same by providing the common pattern

```
bash-3.1$ sed '/^T/ s/\ is/\ was/g' example.sed

This was the first line of an example text.
It is a text with errors.
Lots of errors.
So many errors, all these erors are making me sick.
This was a line not containing any errors.
This was the last line.
```

What if we use the following command in a python file?

```
sed '/\/\*/,/\*\// s/.\+//' program.py
```

Delete

d	all lines
6d	line 6
/^$/d	all empty lines
/^\./d	all lines starting with .
1,10d	lines 1 -10
1,/^$/d	from the first line until the first empty line
/^$/,$d	from the first empty line until the last line

```
sed -e '/\/\*/,/\*\// s/.\+//' -e 's/^[ \t]\+//' -e'/^$/ d' file.py
```

sed – extra examples

- Replace "foo" with "bar" only in lines containing "baz"

```
sed '/baz/s/foo/bar/g'
```

- Remove empty space from the stand and end of each line

```
sed 's/^[ \t]*//;s/[ \t]*$//'
```

- Add 5 spaces at the start of each line

```
sed 's/^/     /'
```

- Remove all empty lines from a file

```
sed '/^$/d'
```