Streams and Pipelines

Marek Kozłowski

Faculty of Mathematics and Information Sciences
Warsaw University of Technology

Outline of the Lecture

Outline of the lecture

Streams

- *Stream* is a unidirectional communication channel for transferring ordered bytes.
- *Streams* are used for:
  - reading from and writing to files (`fopen()` creates a stream and associates it with a file),
  - interacting with a user via a keyboard and display (a tty),
  - inter-process communication (so-called *pipelining*),
- A process is unaware if the other end of a stream is associated with:
  - a file,
  - an I/O device,
  - other process.

Standard Streams

- For each process three standard streams are pre-defined. Those are:
  - `stdin` (`0`) – standard input,
  - `stdout` (`1`) – standard output,
  - `stderr` (`2`) – standard error.
- Processes usually don't read from a keyboard directly. Instead they read from `stdin`, by default associated with a keyboard.
- Processes usually don't write on display directly. Instead they use `stdout` and `stderr`, by default associated with a display.

Standard Output and Standard Error Streams

- `stdout` is used for normal output.
- Error messages should be sent to `stderr`.
- Since both are associated with a display they can be easily confused by inexperienced users.

It's easy to confuse `stdout` and `stderr` messages:

Normal output and error messages

```
$ date # outputs to stdout
$ cd non-exeisting # print error message on stderr
```

Stream Redirection

- Standard streams my be redirected to files with the >, >> and < operators.

Stream redirection

```
$ ls > somefile # overwrite somefile with output
$ ls >> somefile # append output to somefile
$ somesetup < somefile # get input from somefile
$ ls >& somefile # redirect output and errors to somefile
$ ls &> somefile # same as above
$ ls &>> somefile # append output and errors to somefile
```

Stream Redirection in Bash

- Bash as well as other sh-compliant shells is able to use numbers as stream identifiers. `2>` can be used for standard error redirection, `2>&1` – for merging standard output and standard error. The latter one is commonly used in bash scripts.

  ✍ *The following commands are equivalent:*

Stdout and stderr redirection

```
$ ls >& file
$ ls > file 2>&1
```

Redirecting to */dev/null*

- `/dev/null` is a special file that acts as a *black hole*.
- Stream content may be ignored by redirecting it to `/dev/null`.
- This technique is commonly used in bash scripts for silent mode effect.

Command is silent mode:

Ignoring output and error streams

```
$ date >/dev/null # ignore stdout
$ cd non-existing 2>/dev/null # ignore stderr
$ date >/dev/null 2>&1 # ignore both
```

Stream Redirection Examples

Clear a history file:

```
$ echo -n > ~/.bash_history
```

Change history settings:

```
$ echo "HISTSIZE=4096" >> ~/.bashrc
$ echo "HISTFILESIZE=4096" >> ~/.bashrc
```

Disable *writes*:

```
$ echo "mesg n" >> ~/.bashrc
```

Prepare a patch file:

```
$ diff file1 file2 > patchfile
```

Ignore *Permission denied* messages:

```
$ find /etc -mtime -30 2>/dev/null
```

Filters

- Numerous commands process files and require file names as arguments.
- Opening a file is performed by creating a stream associated with it.
- Such stream operates the same way as `stdin`.
- If no file name is specified then `stdin` stream may be used instead.
- Commands that process `stdin` if there are no file names given are referred as *filters*.

Pipelines

- `stdout` of a command may be redirected to `stdin` of another command by using the pipe (`|`) symbol.
- If the second command is a filter it processes `stdout` of the preceding command.
- Such technique is referred as *pipelining*.
- Pipelining allows advanced data processing by connecting relatively simple "building blocks".
- One may connect stdout and stderr to stdin of other command by the `|&`. This syntax is very rarely used.

Environment variables are unsorted by default; `wc -l` prints the number of lines in a file/stream:

Pipelines

```
$ env | sort
$ ls | wc -l
```

Command Substitution

- Commands quoted in ` (on the tilde key) are replaced with their `stdout` content with all newlines deleted.

  ✍ *The `date` command displays (by default: current) date. `touch` creates an empty file of a given name if it doesn't exist. Note that double quotes are necessary in the first example to protect blanks:*

Command substitution

```
$ date
  Fri Feb 25 8:09:10 CET 2011
$ touch "`date`"
$ date +%d.%m.%Y
  25.02.2011
$ touch `date +%d.%m.%Y`
```

Command Substitution Examples

Calculate number of environment variables:

```
$ echo "Number of variables: `env | wc -l`"
```

Find Linux kernel modules:

```
$ find /lib/modules/`uname -r`/kernel -name "*.ko.zst"
```

In case you've forgotten:

```
$ echo "My name is: `whoami`"
```

Some dynamic loop:

```
$ for i in `ls`; do ...
```

Outline of the lecture

File Viewers

- All file viewers: `more`, `less`, `head` and `tail` may be used as pipeline filters.

  ✍ *Note that not all ttys allow scrolling up:*

  Listing shell variables

  ```
  $ set | less
  ```

  ✍ *Display yhe 7^{th} line of a file:*

  The last one of the first seven lines

  ```
  $ head -n7 /etc/passwd | tail -n1
  ```

- `cat` can act as a filter too but since it just copies `stdin` to `stdout` with no processing it makes no sense.

Word Count (*wc*)

Command syntax

```
$ wc [-l|-w|-c] file
```

- Prints number of lines (EOLs), words (blank separated strings) and characters in a file as well as the file name.
- Note that pipeline filters see no file names.
- Output can be limited to lines (`-l`), words (`-w`) or characters only (`-c`).
- Numerous commands put each output record on a separate line so the `wc -l` filter is especially useful for calculating them.

Word Count (*wc*) Examples

A trick to skip a file name:

```
$ wc -l /etc/passwd
$ echo "Number of /etc/passwd lines: `cat /etc/passwd | wc -l`"
```

Calculating the number of ...:

```
$ set | wc -l # variables
$ env | wc -l # environment variables
$ ls -A | wc -l # files
$ pacman -Q | wc -l # packages installed in Arch Linux
$ getent passwd | wc -l # local and remote users
```

Line Sorting (*sort*)

- Sorts lines of a file or a stream.
- All options for sorting files can be used for streams too.

env doesn't sort environment variables by default:

Environment variables sorted

```
$ env | sort
```

Filtering ASCII Strings (*strings*)

- Displays printable (lower ASCII) character sequences (strings) from a file / stream.
- Allows displaying string literals from a binary file (a program).
- For text files it does no harm and acts the same as `cat`.

Generic Regular Expression Parser (*grep*)

Command syntax

```
$ grep [options] pattern file
```

- `grep` displays only those lines which contain strings that match the pattern. The most common options include:
  - `-i` – ignore case,
  - `-r` (or `-R`) – scan a directory content recursively,
  - `-v` – invert selection (show only non-matching lines).
- `zgrep` allows grepping over compressed files.

grep Examples

Searching for some pattern in binary file:

```
$ strings /bin/bash | grep License
```

Searching for configuration all files containing some IP address fragment:

```
$ grep -R 194.29.178 /etc 2>/dev/null
```

Outline of the lecture

POSIX Regular Expressions

- Templates/patterns for any set of strings.
- Key concept of automata theory.
- Heart of numerous programming languages, especially:
  - awk,
  - perl,
  - ruby.
- Used for searches in `less` (`man`), `vim`, etc.
- Syntax standardized by POSIX (BRE/ERE) although some extensions do exist (for example: PCRE).

Regular Expression Syntax

- Regular expression syntax:
  - `(`<r>`)` – regular expression (grouped),
  - c – 'c' character,
  - `\`c – special character or meta-character (for example: `\t`),
  - `^` – beginning of the string,
  - `$` – end of the string,
  - `.` – any single character,
  - `[`ab...`]` – any single character – a, b etc.,
  - `[`a`-`z`]` – any character from the range a - z,
  - `[^`ab...`]` – any character except ...,
  - <r>`*` – expression <r> repeated 0 or more times,
  - <r>`+` – expression <r> repeated 1 or more times,
  - <r>`?` – expression <r> 0 or 1 times,
  - <r>`{n,m}` – expression <r> `n` to `m` times,
  - <r1>`|`<r2> – alternative: <r1> or <r2>.

Notes on Regular Expression Syntax

- `[0-9]+` doesn't denote the same digit one or more times. It means: *any non-empty sequence of digits*.
- Several ranges or sets can be defined inside single square brackets, for example: `[A-Za-zĄĆĘŁÓŻŹąćęłóżź]`.
- A dot can be denoted in two ways: as `\.` or `[.]`.

Regular Expressions – Examples

- `ali(baba|gator)` – *alibaba* or *aligator*,
- `[A-Z][a-z]*ski$` – full name that ends with *ski*,
- `(ba|c|k|tc|z)?sh` – any shell,
- `[ \t]+` – one or more blanks,
- `[+-]?[0-9]+[.]?[0-9]*` – signed decimal.

grep and Regular Expressions

- `grep -E` is able to use regular expressions as patterns.
- Due to some specification differences some `grep` implementations don't recognize `\t` as tab in ERE regular expressions. Use `grep -P` (perl regular expressions, PCRE) if you experience this issue.
- In both cases regular expressions have to be placed in single quotes to prevent interpreting special characters by the shell.