U N I V E R S I T Y ®

CS 50011: Introduction to Systems II

Lecture 2: More *nix

Prof. Jeff Turkstra

I/O redirection - reading

- To redirect input for a program or command,
  `< file` `n < file` n is the file descriptor
  `<< file` `n << file` n is the file descriptor
- Example:
  `mail jeff@purdue.edu < my_document`

I/O redirection - writing

- We can redirect the output from a program or command too!
  `> file` and `n > file` Redirect output to `file`
  `>> file` and `n >> file` Appends output to `file`
  `>| file` and `n >| file` Overrides the noclobber option, if set
  `>& number` Redirects the output to file descriptor `number`

I/O redirection - pipes

- Pipes enable a series of programs to work together
  `command_1 | command_2 | … | command_n`
- Functions a lot like `>` except stdout from `command_n-1` is redirected to stdin of `command_n`.
- Example:

```
$ ls -l | wc -l
46
```

counts how many lines of text `ls` just output

`tee` command

- Check out the unix `tee` command…
  `any_command | tee save_out`
  - Saves a copy of all output (sent to standard out) in the file `save_out`

  `tee save_in | any_command`
  - Saves a copy of all input (sent to standard in) in the file `save_in`

`grep` command

- Used to search files for lines of information. Many, many flags - see the man page.
  `grep -flags regular_expression filename`
- Useful flags…
  `-x` Exact match of line
  `-i` Ignore upper/lower case
  `-c` Only count the number of lines which match
  `-n` Add relative line numbers
  `-b` Add block numbers
  `-v` Output all lines which do not match

Simple regular expressions

- Regular expressions express patterns. They are used to find and/or extract pieces of information from a string.
  `.` Matches any character
  `^` Start of line
  `$` End of line
  `\` Escape character
  `[list]` Matches any character in the list
  `[^list]` Matches any character not in the list
  `*` Match zero or more occurrences of the previous regular expression
  `\{min,max\}` Matches at least min and at most max occurrences of the previous regular expression

Examples

- `grep "^string$" file_name`
  collects all lines which contain only string
- `grep " … " file_name`
  collects all lines which have any three characters surrounded by spaces
- `grep " [0-9]\{1,3\} " file_name`
  collects all lines containing a sequence of one to three digits surrounded by spaces
- `grep "^x*[abc]" file_name`
  collects all lines which start with zero or more `x`'s followed by a single `a`, `b`, or `c`

More examples

- Let's pretend we have a file named `data1`…

```
12
12 345
567
3 abd
asdf
```

- And this script…

```
#! /bin/bash                # begins with 1 or 2
grep "^[0-9]\{1,2\} " data1 # digits followed by
exit 0                      # a space
```

- We should get this output…

```
12 345
3 abd
```

`-v` option, inverting the match

- Let's pretend we have a file named `data1`…

```
12
12 345
567
3 abd
asdf
```

- And this script…

```
#! /bin/bash
grep -v "^[0-9]\{1,2\} " data1
exit 0
```

- We should get this output…

```
12
567
asdf
```

`-c` option, counting the matches

- Let's pretend we have a file named `data1`…

```
12
12 345
567
3 abd
asdf
```

- And this script…

```
#! /bin/bash
grep -c "^[0-9]\{1,2\} " data1
exit 0
```

- We should get this output…

```
2
```

`-n` option, adding line numbers

- Let's pretend we have a file named `data1`…

```
12
12 345
567
3 abd
asdf
```

- And this script…

```
#! /bin/bash
grep -n "^[0-9]\{1,2\} " data1
exit 0
```

- We should get this output…

```
2:12 345
4:3 abd
```

Using `grep` inside a script

```
#! /bin/bash
if (( $# != 1 )); then
  echo "Usage: $0 <user_id>"
  exit 1
fi
USER="$1"
if echo "${USER}" Id_File > /dev/null
then
  echo "Bad way: ${USER} in file"
else
  echo "Bad way: ${USER} not in file"
fi
if grep "^${USER}$" Id_File > /dev/null
then
  echo "Good way: ${USER} in file"
else
  echo "Good way: ${USER} not in file"
fi
exit 0
```

13

Output

```
$ cat Id_File
sam
maryann
john
jeff
jeffrey
bill
william
peterson
```

```
$ Check
Usage: Check <user_id>
$ Check jeff
Bad way: jeff in file
Good way: jeff in file
$ Check son
Bad way: son in file
Good way: son not in file
```

14

`head`

- Collects the first n lines of a file with n defaulting to 10 if unspecified
  `head [-n] file`
- Examples…

```
head -30 yuk # top 30 lines
head yuk     # top 10 lines
head *       # top 10 lines of every file
```

15

`tail`

- `tail [+/-[n] [b|c|l] [-f]] file`
  delivers n units from the file
  `+n` counting from the top
  `-n` counting from the end
  `n` defaults to -10 if unspecified
  counting by
  `b` blocks
  `c` characters
  `l` lines (default)
  `-f` means follow - infinite trailing output (use `ctrl-c` to stop)
- Has buffer limitations - see the man page

16

Examples

- `tail +10 yuk` # all lines beyond line 10
  `tail -30 yuk` # last 30 lines
  `tail -30c yuk` # last 30 characters
  `tail -30f yuk` # last 30 lines, continuing outputting any added lines
  `tail -30 *` # last 30 lines of all files

17

`cut`

- Used to make vertical cuts across a file
  `cut -flags columns or field filename`
- Useful flags
  `-c` characters
  `-d` field delimiter
  `-f` fields
- See man page for more information

18

Examples

- Given a file `data2`,
```
12345 7890 abcd efgb
This is line one
this is no big deal
```
- …and this script,
```
#! /bin/bash
cut -c1-5,8- data2
echo '------'
cut -d' ' -f2-3 data2
exit 0
```
- We get this output:
```
12345890 abcd efgb
This  line one
this  no big deal
------
7890 abcd
is line
is no
```

Another example

- Here's another example:
```
#! /bin/bash
DAY_OF_WEEK="$(date | cut -d' ' -f1-1)"
MONTH="$(date | cut -d' ' -f2-2)"
DAY="$(date | cut -d' ' -f3-3)"
YEAR="$(date | cut -d' ' -f6-)"
echo "Date: $(date)"
echo "Month: ${MONTH}"
echo "Day: ${Day}"
echo "Year:  ${Year}"
echo "Day of the week: ${DAY_OF_WEEK}"
exit 0
```
- Which outputs…
```
Date: Mon Jul 22 16:01:17 EST 1996
Month: Jul
Day: 22
Year: 1996
Day of the week: Mon
```

`paste`

- Used to combine lines from two files together
  `paste [-dlist] file1 file2 …`
- By default concatenates corresponding lines of the files together using a tab as the separator
- Example:
  `paste -d" " x y z`
  concatenates the corresponding lines of the files `x`, `y`, and `z` together using the list of separators circularly. In this case the list only contains a single space.

More examples

- `paste -s [-d list] file1 file2 …`
  merges lines together serially (one file at a time)
- `paste -s -d" \n" yuk`
  pastes each pair of lines in the file `yuk` together
  - the list specified with `-d` is a space followed by a newline
- See man page for more options and information

`wc`

- Word count
  `wc -[c|w|l] file`
- Used to count
  `-c` characters
  `-l` lines
  `-w` words (separated by whitespace)
- Default is all three

Example

```
#! /bin/bash                Output:
wc x.c                      301   878   8382  x.c
wc -l x.c                   301 x.c
wc -w x.c                   878 x.c
wc -c x.c                  8382 x.c
NL=$(wc x.c)              301 878 8382 x.c
echo ${NL}                "    301    878    8382 x.c"
echo "\"${NL}\""          "    301"
LL=$(wc -l < x.c)         301
echo "\"${LL}\""
echo ${LL}
exit 0
```

A quick look at `sort`

- Read the man page for more information
- A few very useful flags:
  `-u` unique lines only
  `-tx`field separator `x` (default is whitespace)
  `-b` ignore leading blanks
  `-r` reverse sort
  `-n` numbers not characters
  `-k` sort on fields (up to 10 `-k` options allowed)
  - Note: field numbers begin with 1

`sort` example

- This example,

```
#! /bin/bash
cat data
echo
sort data3
exit 0
```

- Yields this output:

```
1  a  5          0  b  3
2  b  4          1  a  5
3  a  4          1  a  5
1  b  4          1  b  4
0  b  3          2  b  4
1  a  5          3  a  4
```

Another example

- This example,

```
#! /bin/bash
cat data
echo
sort -u data3
exit 0
```

- Yields this output:

```
1  a  5          0  b  3
2  b  4          1  a  5
3  a  4          1  b  4
1  b  4          2  b  4
0  b  3          3  a  4
1  a  5
```

`-u` and `-k` example

- This example,

```
#! /bin/bash
cat data
echo
sort -u -k 2 data3
exit 0
```

- Yields this output:

```
1  a  5          3  a  4
2  b  4          1  a  5
3  a  4          0  b  3
1  b  4          1  b  4
0  b  3
1  a  5
```

Another `-u` and `-k` example

- This example,

```
#! /bin/bash
cat data
echo
sort -u -k 2,2 data3
exit 0
```

- Yields this output:

```
1  a  5          1  a  5
2  b  4          0  b  3
3  a  4
1  b  4
0  b  3
1  a  5
```

Specifying field order

- This example,

```
#! /bin/bash
cat data
echo
sort -ur -k 2,2 -k 3,3 -k 1,1 data3
exit 0
```

- Yields this output:

```
1  a  5          2  b  4
2  b  4          1  b  4
3  a  4          0  b  3
1  b  4          1  a  5
0  b  3          3  a  4
1  a  5
```

Beating the dead horse

- This example,

```
#! /bin/bash
cat data
echo
sort -k 3,3 -k 1,1 -k 2,2 data3
echo
sort -k 3bn,3 -k 1bn,1 -k 2b,2 data3
exit 0
```

- Yields this output:

```
1  a  5        3  a  14         11 b  4
11 b  4        11 b  4          1  a  5
12 c  40       12 c  40         3  a  14
2  a  40       2  a  40         2  a  40
21 c  51       1  a  5          12 c  40
3  a  14       21 c  51         21 c  51
```

Questions?