Piping and Text Processing

CIS 2230 Linux System Administration

Lecture 8a

Steve Ruegsegger

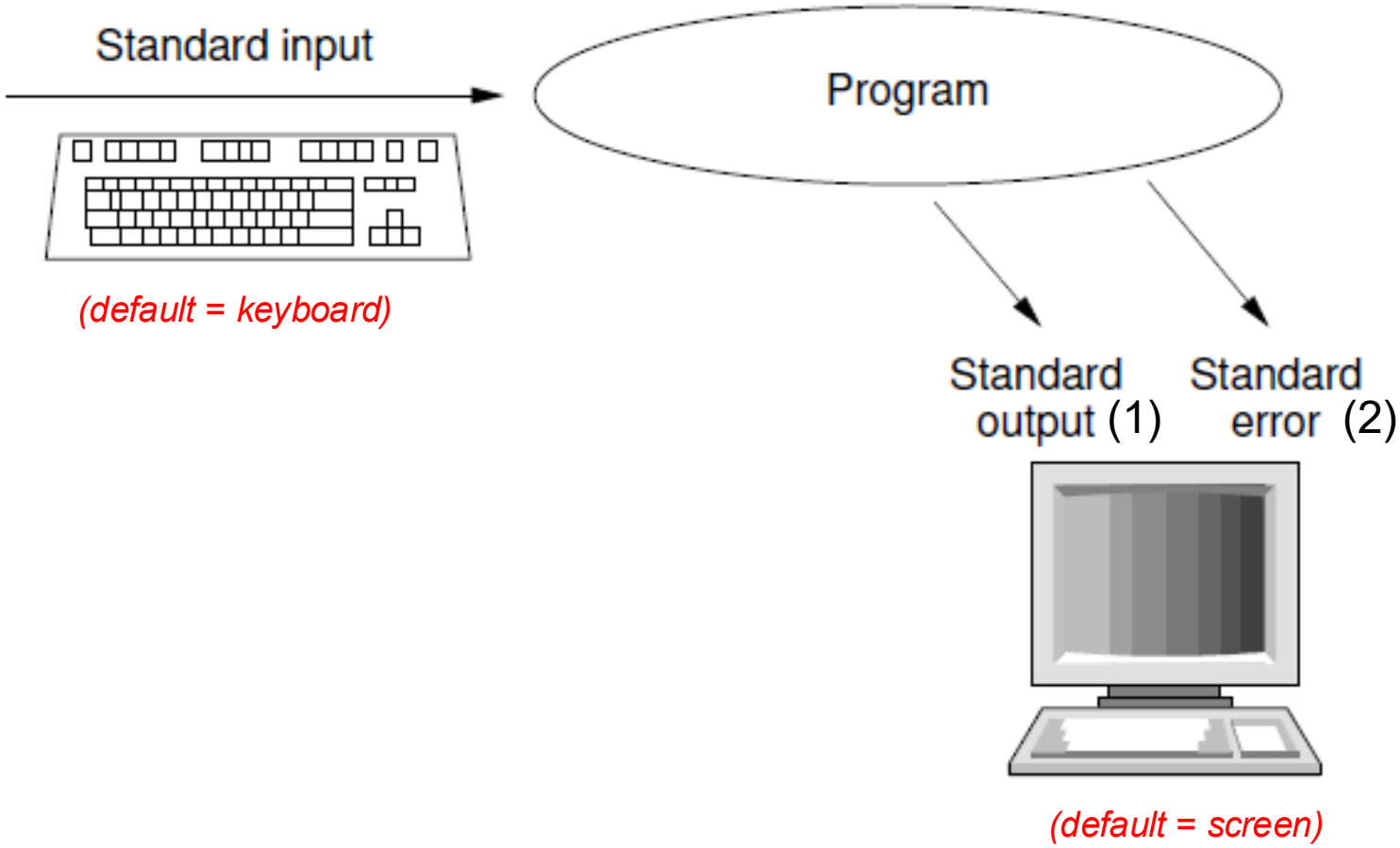
Review

- What is the difference between /home and the “user's home” dir?
- What is the “root” directory and how do we get there?
- What is in /etc?
- What are cwd and $ pwd?
- What are the . and .. directories?
- What is the difference between a relative path and an absolute path?
- What are the 3 short-cuts “home”?
- What is the difference between hard link and soft link?
- What are the 2 main differences between locate and find?
- What is this asking for?

```
$ find ~ -name "*.txt" -mtime -10
```

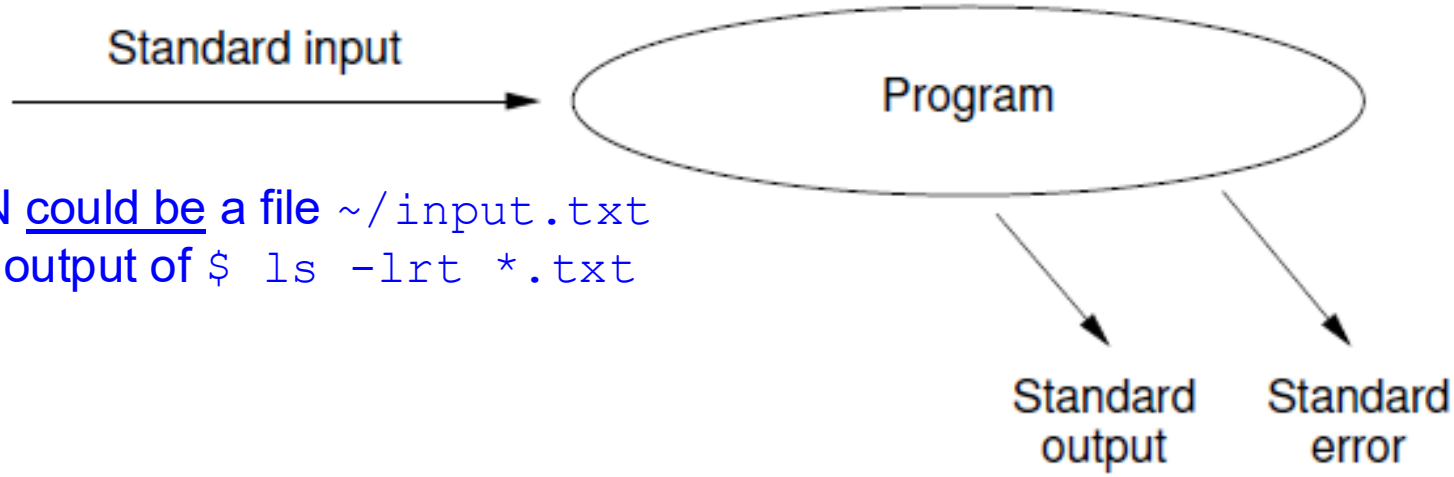
"Standard" Streams (3)

- Processes are connected to three (3) standard "streams"
  - 1 in, 2 out
  - stdin, stdout, stderr



Redirecting streams to files or other commands

- Rather than default of keyboard and screen, these streams could be *redirected*



STDIN could be a file `~/input.txt` or the output of `$ ls -lrt *.txt`

STDERR could be ~/errors.txt or /dev/null

STDOUT could be ~/output.txt or the input to "`sort`"

That darn `cat`

- `cat` = concatenate files together
  - concatenate which files?
  - if not specified… then <u>default</u> from *stdin* to *stdout*
  - no args required
  - Try this… `$ cat`
- We *frequently* redirect output from stdout (screen) to a file :

  `$ cat > lab02.txt`

```
CAT(1)                                User Commands                                CAT(1)

NAME
       cat - concatenate files and print on the standard output

SYNOPSIS
       cat [OPTION]... [FILE]...

DESCRIPTION
       Concatenate FILE(s), or standard input, to standard output.

       -A, --show-all
              equivalent to -vET

       -b, --number-nonblank
              number nonempty output lines

       -e     equivalent to -vE

       -E, --show-ends
              display $ at end of each line
 Manual page cat(1) line 1
```

Connecting Programs to Files

- Redirection connects a program to a named file
- The < symbol indicates redirect into stdin:

```
$ wc < thesis.txt
```

- The > symbol indicates the file to write stdout:

```
$ who > users.txt
```

  - If the file already exists, it is *overwritten*
- Both can be used at the same time:

```
$ filter < input-file > output-file
```

Redirecting multiple standard files

- Open files have numbers, called file descriptors
- These can be used with redirection
- The three standard files *always* have these numbers:
  - stdin - 0
  - stdout - 1
  - stderr – 2
- formats:
  - `n>` = redirect stream n to following location
  - `n>&m` = point stream m to where n is now pointing
  - `&>` = both stderr and stdin are redirected to new location

Redirecting Multiple 'std' Files (cont)

- 2 things can follow the redirection:
  1. Filename
  2. "file descriptor" – a number
- Examples:
  - To redirect the stderr to a file:

```
$ program 2> file
```

  - To save both output streams to *different files*:

```
$ program > stdout.txt 2> stderr.txt
```

  - To combine stderr with stdout into the *same file*

```
$ program > file 2>&1
$ program &> file
$ sas82 < analyze.sas 2>~/analyze.out 1>&2
```

  - Note: order matters!
- The descriptors 3–9 can also be connected to *normal files*, and are mainly used in shell scripts

The bit bucket!

- Recall:
  - everything in unix is a file
  - *even the devices*, which are in `/dev`
- One device is `/dev/null`, which simply ignores all data
  - *a.k.a.* the bit-bucket
- If a stream is giving lots of output you want to ignore, then redirect to `/dev/null`
- *e.g.* permissions in find:

```
$ find / -name "*.txt"  2> /dev/null
```

stderr goes to bit-bucket

Appending to Files

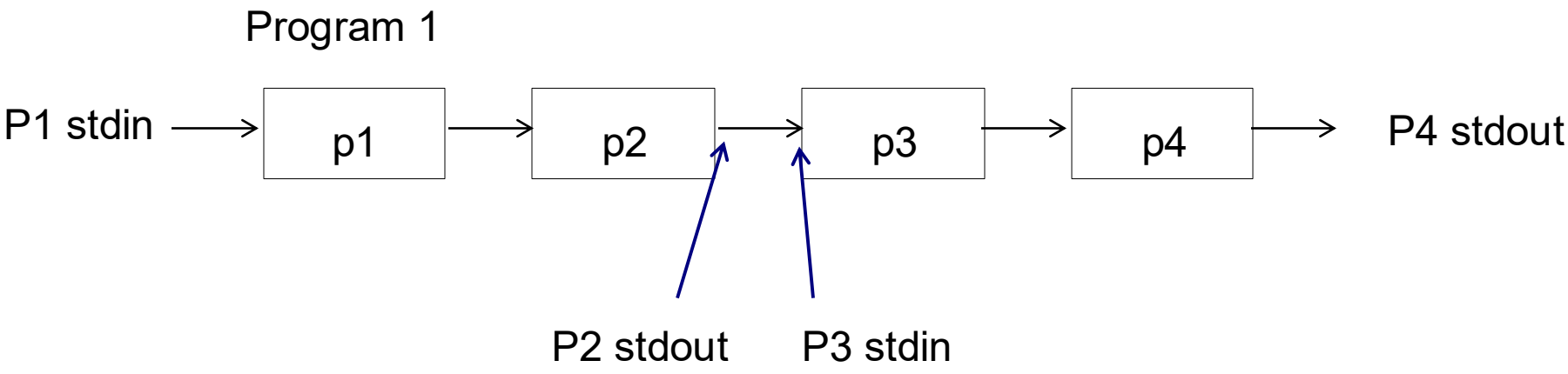
- Use >> to <u>append</u> to a file:

```
$ echo 'whoami' 'hostname' 'date' >> run.log
```

- Appends the stdout of the program to the end of an existing file
- If the file doesn't already exist, it is created

Redirecting streams to another linux command

- The “streams” are connected by “pipes”



Pipe connection

- A pipe (|) channels the output of one program to the input of another
  - Allows programs to be chained together
  - Programs in the chain run immediately after each other
- The OS "channels the streams" – The actual programs don't need to "do" or even "know about" the pipe redirection
- For example, pipe the output of echo into the program rev:

```
$ echo Happy Birthday! | rev
!yadhtriB yppaH
$ cat thesis.txt | more
$ who | wc -l
```

- In these examples, the STDOUT of the 1st program is being sent to the STDIN of the 2nd.
  - The screen/terminal/user does not see the STDOUT of the 1st.

more

- If a file is too long to fit in the terminal, page through it with more

```
$ more <file>
```

- `more` displays one screen at a time and then pauses
  - Waiting for space or enter
- Often used on the end of a pipe

```
$ cat README | more
$ find . -name '*.txt' | more
```

- It's a very 'primitive' and simple program
- Key commands:
  - Searching: /<patt>, n
  - :n, :p – next, prev files
  - = – current line / %age
  - h – help

Less is more better

- `less` is the *'improved'* version of `more`
- Here's what I *love* about `less`
  - Clears the terminal of other things (helpful for small files) and returns terminal back to previous state
  - Goes 'backwards' (b)
  - Doesn't need input stream to finish, so it starts faster
  - Doesn’t choke on strange characters, so it won’t mess up your terminal
  - -N option adds line numbers
- Used in same way as `more`

```
$ wc *.txt | less
$ less userlog.log
```

Reverse line order -- tac

- `cat` backwards :)
- Displays a file in *reverse* line order
- *i.e.* Prints the last line of the input first and then goes back from there.
- Example:
  - The last command shows a list of logins and logouts
  - It puts the most recent ones at the top, so the most important ones scroll off the terminal.
  - With `tac`, we reverse the order and the most recent are on the bottom.

```
$ last | tac
```

head

- Prints the top lines of a file
- Defaults to ten lines
- -n or --lines to print different # lines
- Two main ways to use:

```
$ head <file>
$ cat <file> | head
```

tail

- Similar to `head`, but prints the *last* lines of the stream/file
- Very helpful for log files where most recent, pertinent data is appended at the end of the file. (*e.g.* log files)
- The option -n is the same as in head (number of lines to print)
- Very cool option:
  - The `-f` option watches the file *forever*
  - Continually updates the display as new entries are appended to the end of the file
  - Kill it with Ctrl+C
- Example: monitor HTTP requests on a webserver:

```
$ tail -f /var/log/httpd/access_log
```

Counting with wc

- `wc` counts characters, words and lines in a file
- If used with multiple files, outputs counts *for each file*, and a combined total
- Options:
  - -c output character count
  - -l output line count
  - -w output word count
- Example

```
$ wc -l /usr/share/dict/words
```

- Display the total number of lines in several text files:

```
$ wc -l *.txt
```

- Often used in a pipe to summarize.

```
$ find . -name *.doc | wc -l
$ who | wc -l
```

Selecting Parts of Lines (cols) with cut

- `/usr/bin/cut` is used to select columns or fields from each line
- Two options for 'cutting' lines into cols by
  1. -c = fixed Character range
  2. -f = Fields
- Field separator (delimiter) is specified with `-d` (defaults to tab)
- This is often used to get one column from a stdout
- Examples:

```
$ who | cut -d" " -f1
$ ls -l | cut -c 29-33
```

Sorting Lines of Text with sort

- The `sort` filter reads lines of text and prints them in order
- For example, to sort a list of words into dictionary order:

```
$ sort words.txt > sorted-words.txt
```

- The -n option sorts *numerically*, rather than lexicographically
- Sorts the entire line (*i.e.* first col)
  - Often combined with cut or awk
- What do these do?

```
$ who | cut -d" " -f1 | sort
$ ls -l | cut -c 29-33 | sort -n | tac | head
```

Sorting 'other' columns (-k)

- The *default* sort is the first col of the stream
- However, sort can sort on *other* columns as well
- It calls them "keys" (use the -k option)

```
-k, --key=KEYDEF
       sort via a key; KEYDEF gives location and type
```

- The column number follows the "k"
  - Space-delimited, multiple spaces OK
  - Numbering is 1-based

```
$ ls -l | sort -n -k5
```

  - (Note: the KEYDEF is a fancier way with custom cols)
- This (sort –k) is *especially* helpful when you want to keep other columns for future info or "xargs"

Removing Duplicate Lines with uniq

- Use `uniq` to find unique lines in a file
- However – MUST REMEMBER
  - `$ uniq` *only* removes consecutive duplicate lines
  - Why do you think "consecutive" is a requirement?
  - Therefore, unique is usually given a *sorted* input
  - (I've been burned by this.)
- Example

```
$ who | cut -d" " -f1 | sort | uniq
```

- Note: sort has a -u option

```
$ who | cut -d" " -f1 | sort -u
```

Translating Sets of Characters with tr

- `tr` translates one set of characters to another
- Usage: `$ tr <start-set> <end-set>`
- Replaces *all* characters in start-set with the <u>corresponding</u> characters in end-set
- Example:
  - Replace all uppercase characters in input-file with lowercase characters:

```
$ cat input-file | tr A-Z a-z
$ tr A-Z a-z < input-file
```

- Use `-d` to delete characters <u>only</u>:
  - Delete all occurrences of '*' in story.txt

```
$ cat story.txt | tr -d '*'
```

  - To convert a DOS text file to a unix file (real example): (DOS files have extra ^M's)

```
$ tr -d \\015 < file.dos > file.txt
```

Introduction to filtering -- grep

- `grep` filters a text stream (file or stdin) by line, and *only* prints/outputs entire lines with the search string *somewhere* in that line.
- A good resource… https://help.ubuntu.com/community/grep
- Format: `$ grep <pattern> [file(s)]`
  - Is Bob online?

```
$ who | grep bob
```

  - What processes am I running?

```
$ ps -ef | grep steve
```

  - List log entries with Mary

```
$ grep mary *.log
```

4 types of pattern matching in grep

- `grep`
  - 'basic' regexp PATTERN matching; the default (-G)
- `fgrep` = grep -F
  - 'fixed string' (I always thought 'fast')
  - i.e. 'no' regexp
- `egrep` = grep -e
  - 'extended' regexp PATTERN matching
- `grep -P`
  - 'Perl' regexp → experimental

A,B,C's of grep

- Often, we want a few lines AROUND the matched pattern to be printed: context
  - `-A` – n lines after
  - `-B` – n lines before
  - `-C` – n lines on either side (context)
- Example:

```
$ ifconfig | grep -C2 eth0
```

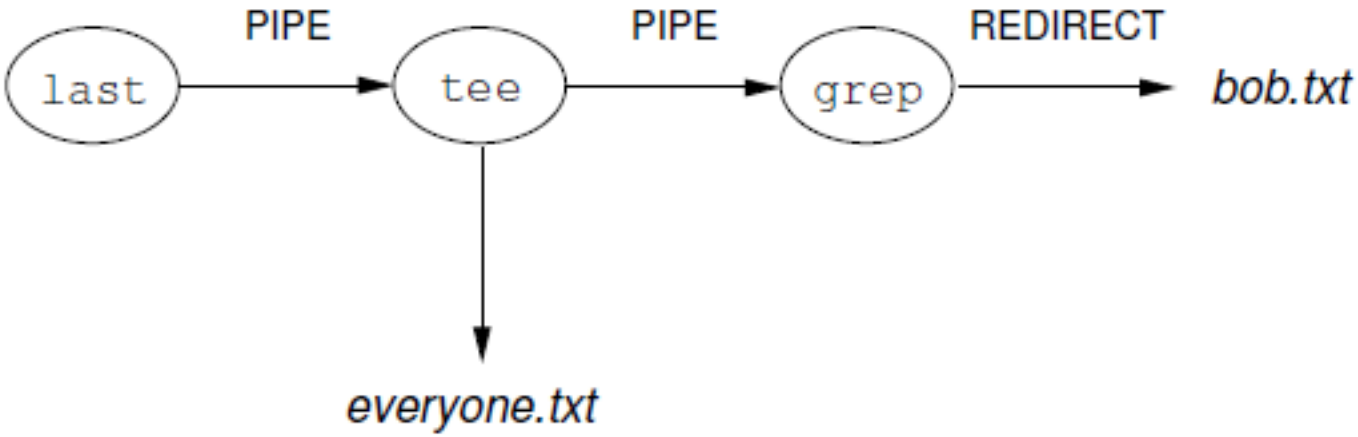
- Used for scripting I need a line "near" a key word.
- Can you explain how this works?

```
IP=$(ifconfig | grep -C1 wlan0 | grep \
"inet addr" | cut -c 21-32)
echo IP=$IP
```

tee

- The `tee` program makes a 'T-junction' in a pipeline
- It copies data from stdin to stdout, and also to a file
- It's like > (redirect) and | (pipe) combined into 1 step
- For example, to save details of everyone's logins, and save Bob's logins in a separate file:

```
$ last | tee everyone.txt | grep bob > bob.txt
```



'z' commands

- Recall gzip?
- You can view a compressed files without having to uncompress to disk!
  - *i.e.* we do not want to uncompress, process, recompress (yuck)
- The compressed file can be uncompressed to the STDOUT stream where it can be processed as though it was not compressed
- Know these commands:

```
$ gunzip -c
$ zcat
$ zmore
$ zgrep
```

- Example

```
$ zmore server.log.gz
$ zgrep steve server.log.gz
$ zcat server.log.gz | cut -f3 | sort | more
```