Administration of Operating Systems

DO2003

http://www.hh.se/do2003

The Shell

The Bourne Again Shell (bash)

HÖGSKOLAN HALMSTAD

Lecture Overview

- At the end of the lecture students are able to:
  - Describe what a Shell is and its main functionalities
  - Use the CLI and describe how commands are executed
  - Redirect stdin, stdout, stderr
  - Control jobs

The Shell

HÖGSKOLAN HALMSTAD

The Shell

- The term shell generally means any program that users employ to type commands
- Is command interpreter and a programming language
- Provide an interface between users and the operating system, i.e., provides access to the kernel services
  - Command-line (CLI) or Graphical (GUI)
- Shells may be used interactively or non-interactively
- Execute utility and application programs and Shell scripts
  - Synchronously and asynchronously
- Several examples
  - sh, dash, bash, ksh, …

The Command line

- Line containing a command and its arguments
- Syntax

```
0          1        2            n-1
command [arg1] [arg2] … [argn] <ENTER>
$ ls -l<ENTER>
```

- Arguments (token or word)
  - Numbered starting with the command itself (arg0)
  - Optional or Mandatory
    - `ls [OPTION]... [FILE]...`
    - `cp [OPTION]... [-T] SOURCE DEST`

HÖGSKOLAN HALMSTAD HH

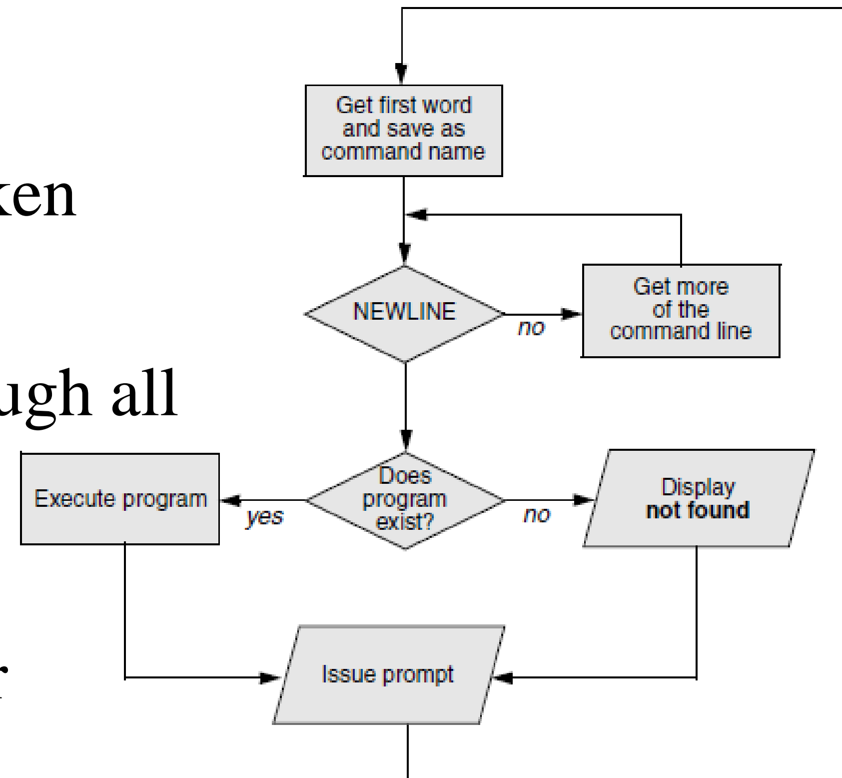
The command Line

- Option arguments
  - Modify the effect of a command
  - Short options `ls -r`
  - Long options `ls --reverse` (GNU specific)
  - Options can be combined, some require arguments
    - `$ ls - r`
    - `$ ls -- r`
    - `$ ls -rx`
    - `$ ls -r x`
    - `$ ls --reverse -x`
    - `$ gcc –o progOut progSourceCode.c`

HÖGSKOLAN HALMSTAD

Processing the Command Line

- Parsing
  - Breaks the command line
  - The command is the first token
- Does the program exist?
  - The shell does not look through all directories
  - PATH
    - Where the Shell looks for programs
- Execute permission
- `$ ./myProgram`



HÖGSKOLAN HALMSTAD

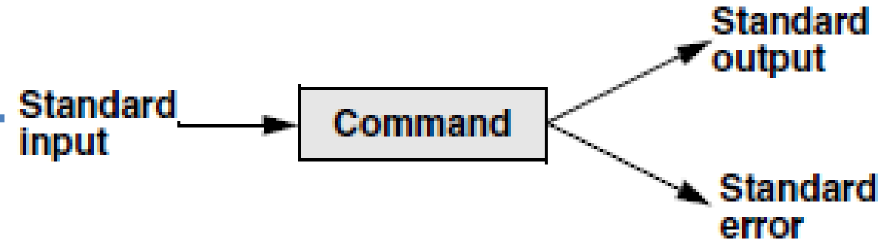
Executing the command line

- If the command (program) is found
  - The shell execute it as a new process
  - The shell waits for the process to finish
    - Sleep state
  - `$`
- Otherwise
  - `x: command not found`
  - `x: invalid option –'y'`

    `Try 'x --help' for more information`

HÖGSKOLAN HALMSTAD HH

Standard I/O



- Standard output (**stdout**)
  - To where a program sends information (Screen)
- Standard input (**stdin**)
  - From where the program gets information (Keyboard)
- Standard error (**stderr**)
  - To where a program sends error messages (Screen)
- stdout, stdin and stderr can be **redirected**
  - File (common to all), device files (/dev)

Redirecting standard I/O

- Redirection
  - By default the shell associates stdin with the keyboard and stdout and stderr with the screen
- Redirecting standard ouput (>)

```
$ command [arguments] > filename
```

- Redirecting standard input (<)

```
$ command [arguments] < filename
```

- Appending standard output to a file (>>)

```
$ command [arguments] >> filename
```

Redirecting standard I/O

- File descriptor
  - Place a program sends its output to and gets its input from
  - When you execute a program, Linux opens three file descriptors for the program
    - 0 (standard input)
    - 1 (standard output)
    - 2 (standard error).
  - > is shorthand for 1>
  - < is short for 0<
  - 2> redirects standard error

Redirecting standard error

- By default, utilities and programs output and error messages are displayed on the screen
- stdout redirection of a command doesn’t affect stderr

```
$ ls
$ echo 'This  is  y' > y
$ cat x y 2> Error.log
This  is  y
$ cat Error.log
cat: x: No such file or directory
$ ls
Error.log    y
```

Redirections

- Good to know
  - Redirecting output can destroy a file

```
$ touch file1.txt
$ echo “line of text” > file1.txt
$ echo “another line of text” > file1.txt
$ echo “more text” > file2.txt
$ cat file1.txt file2.txt > file1.txt
```

  - How to avoid overwriting files
    - noclobber

```
$ set –o noclobber
$ set +o noclobber
```

Redirection operators

Operator	Meaning
`< filename`	Redirects stdin input from filename
`> filename`	Redirects stdout to filename unless filename exists and noclobber is set
`>| filename`	Redirects stdout to filename, even if the file exists and noclobber is set
`>> filename`	Redirects and appends stdout to filename unless filename exists and noclobber is set
`&> filename`	Redirects stdout and stderr to filename
`<&m`	Duplicates stdin from file descriptor m
`[n]>&m`	Duplicates stdout or file descriptor n if specified from file descriptor m
`[n]<&-`	Closes standard input or file descriptor n if specified
`[n]>&-`	Closes standard output or file descriptor n if specified

```
$ echo 3 141592 > pi
$ exec 3<> pi
$ read -n 1 <&3
$ echo -n . >&3
$ exec 3>&-
$ cat pi
3.141592
$
```

Pipe (|)

- Connects stdout of one command to stdin of another command

```
$ command1 [args] | command2 [args]
$ ls | lpr
```

  is equal to

```
$ ls > temp
lpr temp
rm temp
```

- So, what is the advantage of the first example over the second one?
  - No intermediate file is created

HÖGSKOLAN HALMSTAD

Filters

- Command that processes a data stream
- Reads its stddin from stdout of one command and writes its stdout to stdin of another command

```
$ who | tee who.out | grep ’wagner’
$ who | sort | head -1
```

  - …

Foreground and background execution

- So far, all commands run in the foreground
  - The shell waits for the command to finish
- Programs can run in the background so the shell will not wait for them to display the prompt
- You can have only one foreground, but you can have many background jobs
- Job
  - Series of one or more commands that can be connected by pipes (command pipeline)

```
$ who | sort | head -1 > myFile.txt &
[21234]
```


HÖGSKOLAN HALMSTAD

Job control

- To run a command in the background (&)

```
    command [arg1] … [argn] & <ENTER>
```

```
$ sleep 10 &
[1] 25645
```

  - [1] is the job number
  - 25645 is the process identification (PID) number
- You can check the status of processes using `ps`
- You can check status of jobs using `jobs`

```
$ jobs
[1]+ Running          sleep 10
```

Job control

- Suspending a foreground job (stopping it from running)
  - CONTROL-Z
- Restarting a suspended job
  - Running in the background

    ```
    $ bg [job number]
    ```

  - Running in the foreground

    ```
    $ fg [job number]
    $ % [job number]
    ```

- CONTROL-C aborts a process running the foreground it
- To abort a a process running the foreground
  - `$ kill [job number]`

HÖGSKOLAN HALMSTAD HH

The shell handles wildcards

- File name generation
  - ? wildcard

    ```
    $ ls file?.txt
    $ ls file?
    ```
  - * wildcard

    ```
    $ ls file*
    $ ls file1*
    ```
  - [ ] wildcard

    ```
    $ ls file[1234].txt
    $ ls file[1-3].txt
    ```

```
$ ls file[^12].txt
```

Ambiguous file references

```
$ ls file*
file1.txt  file2.txt  file3.txt
$
```


HÖGSKOLAN HALMSTAD HH

The Shell has builtin commands

- Each of the shells has its own set of builtins
- Builtins run more quickly
  - The shell does not fork a new child process
  - The shell does not look for the command
- Examples

```
bg, fg, jobs, kill
cd, pwd
echo
read
```

- **http://www.gnu.org/s/bash/manual/html_node/Bash-Builtins.html**

HÖGSKOLAN HALMSTAD