Principles of Computer Science II

Unix Programming Shell

Ioannis Chatzigiannakis

Sapienza University of Rome

Lecture 11

What is a Shell?

- The user interface to the operating system
- Functionality:
  - Execute other programs
  - Manage files
  - Manage processes
- A program like any other
- Executed when you "open a Terminal"
- The shell
  - Allows the execution of command scripts
  - Enables alternative methods to carry out complex tasks
  - Provides variables

Shell Interactive Use

- The # is called the "prompt"
- In the prompt we type the name of the command and press "Enter"
- The prompt allows
  - Command history
  - Command line editing
  - File expansion (tab completion)
  - Command expansion
  - Key bindings
  - Spelling correction
  - Job control

Prompt: The Command Line

```
# date
Sat Apr 21 16:47:30 GMT 2007
```

Error Handling

- If we type a wrong command, an error message appears

Prompt: The Command Line

```
# datee
datee: no such file or directory
```

- The error message states that either the file or the folder (directory) was not found
  - In the prompt all commands are assumed to be connected to a file ...
- The arrow keys ↑ ↓ allow to look-up previous commands
- The arrow keys ← → allow to move within the same command line

Terminating Command Execution

- We can interrupt the execution of a command by pressing *ctrl-c*
- We can "freeze" the output of the execution of a command by pressing *ctrl-s*
  - To "un-freeze" the output of a command we use *ctrl-q*
  - Note – only the output is frozen not the actual execution
- To close a terminal we use *ctrl-d*
  - We may need to press multiple times *ctrl-q*
  - All programs currently running will terminate

Manual Pages

- The command *man* allows to access the manual pages
- Manual pages are organized in categories
  1. Commands – *ls, cp, grep*
  2. System Calls – *fork, exit*
  3. Libraries
  4. I/O Files
  5. File Encoding Types
  6. Games
  7. Miscellaneous
  8. Administrator's Commands
  9. Documents
- We can request a page from a specific category *man [category] [topic]*

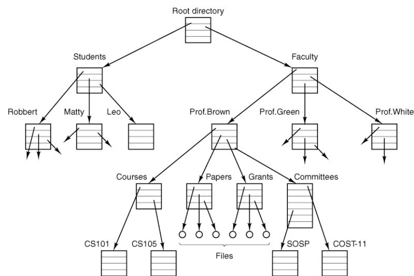
Manual Pages

```
FORK(2)                  Minix Programmer's Manual                  FORK(2)

NAME
     fork - create a new process

SYNOPSIS
     #include <sys/types.h>
     #include <unistd.h>

     pid_t fork(void)

DESCRIPTION
     Fork causes creation of a new process.  The new process  (child  process)
     is an exact copy of the calling process except for the following:

          The child process has a unique process ID.

          The child process has a  different  parent  process  ID  (i.e.,  the
          process ID of the parent process).

          The child process has its own  copy  of  the  parent's  descriptors.
standard-input, 1-24 (Top)_
```

File System

- All system entities are abstracted as files
  - Folders and files
  - Commands and applications
  - I/O devices
  - Memory
  - Process communication
- The file system is hierarchical
  - Folders and files construct a tree structure
  - The root of the tree is represented using the /
- The actual structure of the tree depends on the distribution of Linux
  - Certain folders and files are standard across all Linux distributions

File System Example



Standard Folders

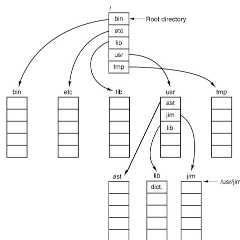
- `/bin` – Basic commands
- `/etc` – System settings
- `/usr` – Applications and Libraries
- `/usr/bin` – Application commands
- `/usr/local` – Applications installed by the local users
- `/sbin` – Administrator commands
- `/var` – Various system files
- `/tmp` – Temporary files
- `/dev` – Devices
- `/boot` – Files needed to start the system
- `/root` – Administrator's folder

Example of File Metadata

```
# ls -la
lrwxrwxrwx   1 bin    operator   2880 Jun  1  1993 bin
-r--r--r--   1 root   operator    448 Jun  1  1993 boot
drwxr-sr-x   2 root   operator  11264 May 11 17:00 dev
drwxr-sr-x  10 root   operator   2560 Jul  8 02:06 etc
drwxrwxrwx   1 bin    bin           7 Jun  1  1993 home
lrwxrwxrwx   1 root   operator      7 Jun  1  1993 lib
drwxr-sr-x   2 root   operator    512 Jul 23  1992 mnt
drwx------   2 root   operator    512 Sep 26  1993 root
drwxr-sr-x   2 bin    operator    512 Jun  1  1993 sbin
drwxrwxrwx   6 root   operator    732 Jul  8 19:23 tmp
drwxr-xr-x  27 bin    bin        1024 Jun 14  1993 usr
drwxr-sr-x  10 root   operator    512 Jul 23  1992 var
```

Navigating the File System

- Each folder contains two "virtual" folders
  `ls -la`
  `.   ..`
- The single dot represents the same folder
  `./myfile` ⇒ `myfile`
- The two dots represent the "parent" folder in the tree



File System Security

- For each file we have 16 bit to define authorization
  - 12 bit are used by the operator
  - They are split in 4 groups of 3 bit – 1 octal – each
- The first 4 bit cannot be changed
  - They characterize the type of the file (simple file, folder, symbolic link)
  - When we list the contents of a folder the first letter is used to signify:
    - - – simple files
    - d – folders
    - l – symbolic links
- The next 3 bit are known as the s-bits and t-bit
- The last three groups are used to define the access writes for read 'r', write 'w' and execute 'x'
  - For the file owner, users of the same group, and all other users.

File System Permissions Examples

```
Type Owner Group Anyone
  d  rwx   r-x   ---
```

- Folder
- The owner has full access
- All users that belong to the group defined by the file can read and execute the file – but not modify the contents
- All other users cannot access the file or execute it
- To access a folder we use the command *cd* given that we have permission to execute 'x'

Changing the File Permissions

Examples of File Permissions

Binary	Octal	Text
001	1	x
010	2	w
100	4	r
110	6	rw-
101	5	r-x
-	644	rw-r--r--

- The command *chmod* allows to modify the permissions
- There are 2 way to define the new permissions
  1. Defining the 3 Octal – e.g., *644*
  2. By using text – e.g., *a+r*

Some Examples of *chmod*

```
make read/write-able for everyone
# chmod a+w myfile

add the 'execute' flag for directory
# chmod u+x mydir/

open all files for everyone
# chmod 755 *

make file readonly for group
# chmod g-w myfile

descend recursively into directory opening all files
# chmod -R a+r mydir/
```

Changing the Owner and Group of a File

- The command *chown* allows to change the owner of a file
- The command *chgrp* allows to change the group of a file

```
give ownership to ichatz
# chown ichatz myfile

set group to students
# chgrp students mydir/

give ownership to pcs and group to students
# chgrp pcs:students myfile mydir/

descend recursively into directory opening all files
# chown -R ichatz mydir/
```

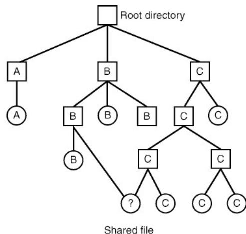
Symbolic Links

- The file system enables to create symbolic links
- Two types are provided
  - Symbolic link
  - Hard link
- The contents and metadata of the original file are used for all operations

```
create a symbolic link to a directory
# ln -s /var/log ./log
# ls -lg
lrwxrwxrwx 1 operator     8 Apr 25   log -> /var/log
```

- The contents and metadata of the original file are used for all operations
  - Except for deletion.

Examples of Symbolic Links



Shared file

Access Dates

- For each file the system keeps track of
  - Date of last usage/access
  - Date of last change

```
check last usage time
# ls -lu
drwxrwxrwx   1 bin     bin           7 Apr 25   1993 home
lrwxrwxrwx   1 root    operator      7 Apr 25   1993 lib
drwx------   2 root    operator    512 Mar 30   1993 root

check last change time
# ls -lc
drwxrwxrwx   1 bin     bin           7 Apr 25   1993 home
lrwxrwxrwx   1 root    operator      7 Oct 27   1993 lib
drwx------   2 root    operator    512 Oct 27   1993 root
```

BASH Script Example

```
$ for dir in $PATH
>do
>  if [ -x $dir/gcc ]
>  then
>     echo Found $dir/gcc
>     break
>  else
>     echo Searching $dir/gcc
>  fi
>done
```

- For each folder within the variable `$PATH`
- Check if the folder contains the file gcc
  - If the file is found, print out the *path* and stop
  - Otherwise continue to the next folder.

Command line

```
# bash
bash-4.4.20#
```

- Left part of # can be changed.
- Right part of # is used to type in commands.
- Offers certain built-in commands
  - Implemented within the BASH source code
  - These commands are executed within the BASH process
- Allows to execute scripts
  - For this reason it is called a UNIX programming environment

Built-in Commands

Command	Description	Exception
cd	Change Folder	`cd ..`
declare	Set a variable	`declare myvar`
echo	Print out a text to the standard output	`echo hello`
exec	Replace bash with another process	`exec ls`
exit	Terminate shell process	`exit`
export	Set a global variable	`export myvar=1`
history	List of command history	`history`
kill	Send a message to a process	`kill 1121`
let	Evaluate an arithmetic expression	`let myvar=3+5`

Built-in Commands

Command	Description	Exception
local	Declare a local variable	`local myvar=5`
pwd	The current folder	`pwd`
read	Read a value from standard input	`read myvar`
readonly	Lock the contents of a variable	`readonly myvar`
return	Complete a function call and return a value	`return 1`
set	List declared variables	`set`
shift	Shifts the command parameters	`shift 2`
test	Evaluate an expression	`test -d temp`
trap	Monitor a signal	`trap "echo Signal" 3`

The echo command (1)

- Main way to produce output
- Prints out values of variables
- Recognizes special characters (or meta-characters)

```
bash-4.4.20# echo hello there
hello there
bash-4.4.20# let myvar=1; echo $myvar
1
bash-4.4.20# echo *
junk lpr-starttime temp
bash-4.4.20# echo print '*' "don't"
print * don't
```

The echo command (2)

- May contain more than 1 lines
- May also execute commands

```
bash-4.4.20# echo 'hello
there'
hello
there
bash-4.4.20# echo hello\
 there
hello there
bash-4.4.20# echo `date`
Mon Apr 30 16:12:21 GMT 2007
bash-4.4.20# echo -n `date` " "
Mon Apr 30 16:12:21 GMT 2007 bash-4.4.20#
```

Meta-characters

- The character *?* – defines any single character, e.g., `ls /etc/rc.????`
- The character * – defines multiple characters, e.g., `ls /etc/rc.*`
- The array *[...]* – defines a specific set of characters, e.g. `ls [abc].c`
- The use of the above meta-characters is also called filename substitution
- We may use these meta-characters in any combination within command execution
- The following command is disabled `mv *.x *.y`

Shell Variables

- The shell allows the declaration of variables
- Initial values of variables are defined in the user settings file
- The scope of the variables is connected with the session
  - Or until the user removes them
- The variables with UPPER-case letters are global – they are transfered to all processes executed by the shell
- The variables with LOWER-case letters are local – they are accessible only by the shell process

```
HOME            # The path to your home directory
term            # The terminal type
```

Shell Variables

- We may use variables at the command line
- We use the descriptor $

```
bash-4.4.20# myvar="hello"; echo $myvar
hello
bash-4.4.20# myvar="ls -la"
bash-4.4.20# $myvar
lrwxrwxrwx  1 bin    operator   2880 Jun  1  1993 bin
-r--r--r--  1 root   operator    448 Jun  1  1993 boot
drwxr-sr-x  2 root   operator  11264 May 11 17:00 dev
...
```

Special Variables

- Some special variables are provided

Variable	Description
USER	User name
HOME	Home folder of user
TERM	Type of terminal
SHELL	Name of shell
PATH	List of folders to look for commands
MANPATH	List of folders to look for manual pages
PWD	Active folder
OLDPWD	Previously active folder
HOSTNAME	Name of the system

Variable Handling

- The commands *env*, *printenv* provide a list of GLOBAL variables
- The command *set* provides a list of LOCAL variables
- To declare a new GLOBAL variable we use the command *export*
- Variable type is define by content type
  - String variables – `myvar = "value"`
  - Integer variables – `declare -i myvar`
  - Constant variables – `readonly me="ichatz"`
  - Array variables – `declare -a MYARRAY`
    `MYARRAY[0]="one"; MYARRAY[1]=5; echo ${MYARRAY[*]}`
- The names of the variables are case-sensitive
- The command *unset* removes a variable

Creation of scripts

- Scripts are used as if they were commands/applications
  - Defined by a source file
- We execute the script using the command *sh*
  - Or directly by setting execute access permissions

```
bash-4.4.20# pico
who
--> save/exit
bash-4.4.20# cat nu
who
bash-4.4.20# sh nu
ichatz        :0
bash-4.4.20# chmod a+x nu
bash-4.4.20# nu
ichatz        :0
```

Handling (1)

- We may pass parameters to a script at command-line
  - These are called the command-line arguments
- We use arguments as variables

Argument	Description
$0	The name of the script
$1 ... $9	The value of 1st ... 9th argument
$#	Number of arguments
$	All the arguments as string

```
bash-4.4.20# pico nu2
echo Files found: "($1\*)" `ls -la $1*`
--> save/exit
bash-4.4.20# nu2 /b
Files found: (/b*)
...
```

Handling Parameters (2)

- In order to access more than 9 parameters
  - We may not use *$10*
- We need to use command *shift x*
  - Shifts the parameters left-wise by *x* positions
  - Shifted parameters are lost (!)

```
bash-4.4.20# cat ten
shift 10
echo $1
echo $* " -- " $#
bash-4.4.20# ten 1 2 3 4 5 6 7 8 9 10
10
10 -- 1
```

Input from the standard input

- We may use the standard input using *read*
  - The syntax is `read var-name`
  - We may use multiple variables
    `read var1 var2 ...`
  - We may output a message before requesting the input
    `read -p "Enter value:" var`

```
bash-4.4.20# read -p "Enter values:" i j k;\
echo i=$i, j=$j, k=$k
abc d e f
i = abc, j = d, k = e f
```

Mathematical Expressions

- Allows the evaluation of mathematical expressions using integers
  - Similar with C programming language
  - No need to explicitely declare a variable as an integer
  - We use *expr* rather than *int*

```
((a = a + 1))
a=$((a+1))
a=$(($a+1))
let a = a + 1
let a++
a=`expr $a + 1`
```

If Expressions

```
if [ condition 1 ]; then
  if [[ condition 2 && condition 3]]; then
  ...
  fi
elif [ condition 4 ] || [ condition 5 ] ; then
  ...
else
  ...
fi
```

- The command *test* allows the evaluation of an expression
  - Returns either true or false
  - Supports broad range of expressions
  - e.g., we might check if we have write access to a given file

```
if test -w "$1"; then echo "File $1 is writable"
fi
```

Evaluation using test

Expression	Description
-gt	Greater or equal
-ge	Greater
-lt	Smaller
-le	Smaller or equal
-eg	Equal
-ne	Not Equal
-n str	Size of the string bigger than 0
-z str	Empty string
-d file	The file is a folder
-s file	A non empty file
-f file	The file exists
-r file	Read access to file
-w file	Write access to file
-x file	Execution access to file

Evaluation Example (1)

```
bash-4.4.20# cat check.sh
#!/bin/bash
read -p "Enter a filename: " filename
if [[ -w "$filename" ]]
then
  echo "File is writeable"
fi

if [[ ! -r "$filename" ]]
then
  echo "File is not readable"
fi
```

Boolean expressions

```
if [[ condition 1 && condition a]]; then
  if [[ condition 2 || condition b]]]; then
  ...
  fi
elif [[ ! condition 3 ]] ; then
  ...
else
  ...
fi
```

For Loop

```
for VAR in <list>
do
  ...
done
```

```
for i in 6 3 1 2
do
  echo $i
done
```

```
for i in *.c
do
  echo $i
done
```

While Loop

```
while [ expression ];
do
  ...
done
```

```
i = 1
while [[ $i -lt 10 ]];
do
  echo $i
  ((i++))
done
```

```
while true;
do
  echo "alive..."
  sleep 3
done
```

Until Loop

```
until [ expression ];
do
  ...
done
```

```
Stop = "N"
until [[ $Stop = "Y" ]];
do
  ps -ef
  read -p "Do you want to stop? (Y/N)" Stop
done
echo "Stopping..."
```

- Within loops we may use *break, continue*

Functions

```
function name [()]
{
  ...
  [return]
}
```

- All functions declaration must be location at the top of the script
- A function may not have any parameters
- Parameters and Return value can be of any type
- Parameters defined within the function are global!
  - We need to explicitely define them as local

Functions: An Example

```
#!/bin/bash
outside = "a global variable"

function mine() {
  local inside="this is local"
  echo $outside
  echo $inside
  outside = "a global with new value"
}

echo $outside
mine
echo $outside
echo $inside
```