Chapter 4
The UNIX Shells

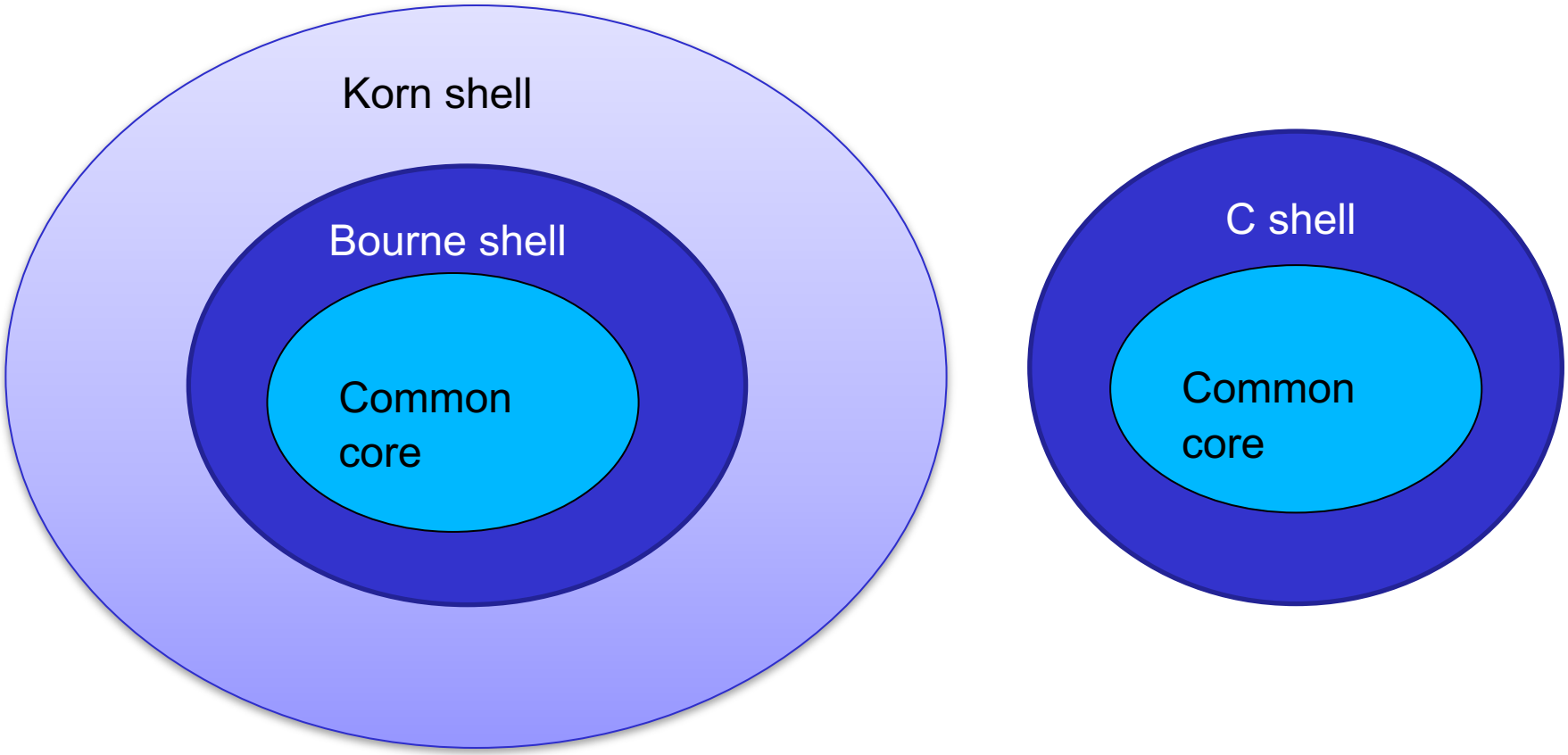
YUAN LONG

CSC 3320 SYSTEM LEVEL PROGRAMMING

SPRING 2019

Updated based on original notes from Raj Sunderraman and Michael Weeks

The Relationship of shell functionality



Which Shell

- To examine your default shell, type:

```
$echo $SHELL
/bin/bash
```

- To change your default shell use the ch**sh** utility

```
yuanlong@yuanlong-VirtualBox:~$ chsh
Password:
Changing the login shell for yuanlong Enter the new value,
or press ENTER for the default
        Login Shell [/bin/bash]: /bin/sh
```

Invoking the Shell

- A shell is invoked, either
  - automatically upon login, or
  - manually from the keyboard or script

What does the shell do?

- The following takes place:
  - (1) reads a special **startup file** (***.bash_profile*** in the user's home directory) and executes all the commands in that file
  - (2) displays a **prompt** and waits for a user command
  - (3) If user enters **CTRL-D** (end of input) the shell terminates, otherwise it executes the user command(s)

CORE Shell Functionality

- User Commands
- Built-in commands
- Scripts
- Redirection
- Wildcards
- Pipes
- Sequences
- Background processing
- Command substitution
- Variables (local, environment)
- Job control

User Commands

- **ls** (list files), **ps** (process info)
- \ continues line

```
$ ls

$ ps -ef | sort

$ ls \
  | ps -ef
```

Built-in commands

- Most Unix commands invoke utility programs stored in the file hierarchy
  - E.g. ***sed*** *is in* ***/bin/sed***
  - E.g. ***awk*** *is in* ***/usr/bin/awk***
  - E.g. ***wc*** *is in* ***/usr/bin/wc***
- The shell has to locate the utility (using *PATH* variable)
- Shells have built-in commands, e.g.:
  - **echo**
  - **cd**

Built-in commands

- echo arguments
  - $ echo Hi, How are you?
  - Hi, How are you?
- echo by default appends a new line (to inhibit new line use -*n* option)
- cd dir

Finding a command: $PATH

- If the command is a shell built-in such as *echo* or *cd*, it is directly interpreted by the shell.
- If the command begins with a /
  - shell assumes that the command is the absolute path name of an executable file
    - E.g. $/home/yuanlong/**start.sh**
  - error occurs if the executable is not found.
- If not built-in and not a full pathname
  - shell searches the directories in the **PATH**
  - from left to right for the executable
    - E.g. $echo $PATH
    - **/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin**
- Current working directory may not be in PATH

PATH variable

- If PATH is empty or is not set, only the current working directory is searched for the executable.
- Append new directory to PATH
  - E.g. Append /home/ylong4/csc3320 to PATH

```
$echo $PATH
/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin
:/bin:/usr/games:/usr/local/games:/snap/bin
$PATH=/home/ylong4/csc3320:$PATH
$echo $PATH
/home/ylong4/csc3320:/usr/local/sbin:/usr/local/bin:/us
r/sbin:/usr/bin:/sbin:/bin:/usr/games:/usr/local/games:
/snap/bin
```

Shell Programs/Scripts

- Shell commands may be stored in a text file for execution
  - E.g. start.sh, try.csh, example.ksh
- Use the *chmod* utility to set execute permissions on the file
  - chmod **a+x** shellScriptName
- Executing it by simply typing the file name
  - **./**shellScriptName
- When a script runs, the system determines which shell to use

Shell Programs/Scripts

- To determine which shell:
  - if the first line of the script is a pound sign (#)
    - then the script is interpreted by the current shell
  - if the first line of the script is of the form
    - #!/bin/sh or #!/bin/ksh or #!/bin/bash
    - then the appropriate shell is used to interpret the script
  - else the script is interpreted by the Bourne shell
- Note: pound sign on 1st column in any other line implies a comment line

Example of Shell Script


```
$cat start.sh
#! bin/bash
# A simple shell script
echo -n "The date today is "
date;
```

```
$chmod 777 start.sh
```

```
$./start.sh
The date today is Sun Sep 15 12:50:51 EDT 2016
```

Metacharacters - 1

- Output redirection
  - > writes standard output to file
  - >> appends standard output to file
- Input redirection
  - < reads std. input from file
  - **<<tok** read std. input until tok

Redirection

- The shell redirection facility allows you to
  - store the output of a process to a file
  - use the contents of a file as input to a process
- Examples:
  - cat x1.c > y.c
  - cat x2.c >> y.c
  - mail ylong4@gsu.edu < hiMom.txt
- The **<<tok** redirection is almost exclusively used in shell scripts

Here Documents


```
$ cat here.sh
#!/bin/bash
mail $1 << ENDOFTEXT
Dear $1,
Please see me regarding some exciting news!
$USER
ENDOFTEXT
echo mail sent to $1
```

```
$./here.sh ylong4@gsu.edu
mail sent to ylong4@gsu.edu
```

Metacharacters - 2

- File-substitution wildcards:
  - * matches 0 or more characters
  - ? matches any single character
  - [...] matches any character within brackets
- Command substitution:
  - `command` replaced by the output of command
  - e.g. *echo `ls`*

Filename Substitution

- $ *ls *.c* # list .c files
- $ *ls ?.c* # list files like a.c, b.c, 1.c, etc
- $ *ls [ac]** # list files starting with a or c
- $ *ls [A-Za-z]** # list files beginning with a letter
- $ *ls dir*/*.c* # list all .c files in directories starting with dir

Command Substitution

- A command surrounded by **grave accents** (`) is executed and its standard output is inserted in the command's place in the command line.

```
$ echo today is `date`
today is Sat Jun 19 22:23:28 EDT 2007
$ echo there are `who | wc -l` users on the system
there are 2 users on the system
```

Metacharacters - 3

- | Pipe
- Send output of one process to the input of another
  - e.g. list files, then use ***wc*** to count lines
    - ls | wc -l
  - this effectively counts the files

Metacharacters - 4

- # Comment
  - rest of characters ignored by shell
- ; Sequences
- (…) Group commands
- & Run command in background
- $ Expand the value of a variable

Sequences

- Commands or pipelines separated by **semi-colons**
- Each command in a sequence may be individually I/O redirected.
- Example:
  - $date; pwd; ls
  - $date > date.txt; pwd > pwd.txt; ls

Sequences

```
$ date
Sun Sep 15 23:19:23 EDT 2016
$ pwd
/home/local/GSUAD/ylong4/public
$ ls
csc3320          dummy      mountainList.txt
```

```
$ date;pwd;ls
Sun Sep 18 23:20:44 EDT 2016
/home/local/GSUAD/ylong4/public
csc3320          dummy      mountainList.txt
```

```
$ date>t1; pwd>t2; ls
csc3320          dummy      mountainList.txt   t1 t2
```

Grouping commands

- Commands can be grouped by putting them within parentheses
  - a sub shell is created to execute the grouped commands
- Example:
- $ (date; ls; pwd) > out.txt
- $ cat out.txt

Example of Grouping commands


```
$(date; ls; pwd) > out.txt
$cat out.txt
Sun Sep 18 23:20:44 EDT 2016
csc3320          dummy      mountainList.txt
/home/local/GSUAD/ylong4/public
```

Background processing

- An **&** sign at end of a simple command,
  - or pipeline, sequence of pipelines,
  - or a group of commands
- Starts a sub-shell
  - commands are executed as a background process
  - does not take control of the keyboard
- A process id is displayed when it begins

Background processing

- Redirect the output to a file (if desired)
  - prevents background output on terminal
  - E.g. $(pwd; sleep 3; who) > log.txt &
  - [1] 18156
  - When it is finished, you will see
    - [1]+ Done (pwd; sleep 3; who) > log.txt
- Background process cannot read from standard input
  - If they attempt to read from standard input; they terminate.

Variables

- A shell supports two kinds of variables:
  - Local variables
    - E.g. X, y, name
  - Environment variables
    - Pre-defined: HOME PATH USER SHELL
    - User defined: E.g. M, file, val
  - Both hold data in string format
- Accessing variables in all shells is done by prefixing the name with a $ sign.
  - E.g. $PATH, $x

Built-in Variables

- Common built-in variables with special meaning:
  - $$ process ID of shell
  - $0 name of shell script (if applicable)
  - **$1..$9** $n refers to the nth command line argument (if applicable)
  - $* a list of all command line arguments

Example using Built-in variables


```
$cat var.sh
#!/bin/bash
echo the name of this file is $0
echo the first argument is $1
echo the list of all arguments is $*
echo Process ID is $$
```

```
$./var.sh one two three
#!/bin/bash
the name of this file is ./var.sh
the first argument is one
the list of all arguments is one two three
Process ID is 18396
```

Quoting

- **Single quotes '** inhibit wildcard replacement, variable substitution, and command substitution
- **Double quotes "** inhibit wildcard replacement only
- When quotes are nested only the outer quotes have any effect

Example of Quoting


```
$ echo 3 * 4 = 12
3 3.log 3.tex script.csh script2.csh 4 = 12
```

```
$ echo ‘3 * 4’ = 12
3 * 4 = 12
```

```
$ echo "my name is $USER; the date is `date`"
my name is raj; the date is Sun Jun 20 21:59:13
EDT 2007
```

Job Control

- ***ps*** command generates a list of processes and their attributes
  - E.g. $ ps –elf
- ***kill*** command terminates processes based on process ID
  - E.g. $kill -9 18403

Example of Job control


```
$(sleep 20; echo done) &
[1] 3832
```

```
$ps –a
PID   TTY          TIME     CMD
3832 pts/4    00:00:00     bash
3833 pts/4    00:00:00     sleep
3834 pts/4    00:00:00     ps
```

```
$kill -9 3832
$ps -a
PID   TTY          TIME     CMD
3833 pts/4    00:00:00     sleep
3856 pts/4    00:00:00     ps
[1]+  Killed                    ( sleep 20; echo done )
```

Termination and Exit codes

- Every Unix process terminates with an exit value
- By convention, a zero value means success and a non-zero value means failure
- All built-in commands return 1 when they fail

Termination and Exit codes

- Any script written by you should contain the exit command:
  - *exit <number>*
- The special variable ***$?*** contains the exit code of the last command execution.
- If the script does not exit with an exit code, the exit code of the last command is returned by default.

Review

- Covered core shell functionality
  - Built-in commands
  - Scripts
  - Variables
  - Redirection
  - Wildcards
  - Pipes
  - Background processing