Introduction to Bash Programming

Dr. Xiaolan Zhang

Spring 2013

Dept. of Computer & Information Sciences

Fordham University

Outline

- Shell command line syntax
- Shell builtin commands
- Shell variables, arguments
- I/O redirection
- Shell tracing
- Shell initialization

Last class

- Shell:
    - Interactive mode:
    - Scripting mode
- Command line
- File system,
- Some commands

Command line

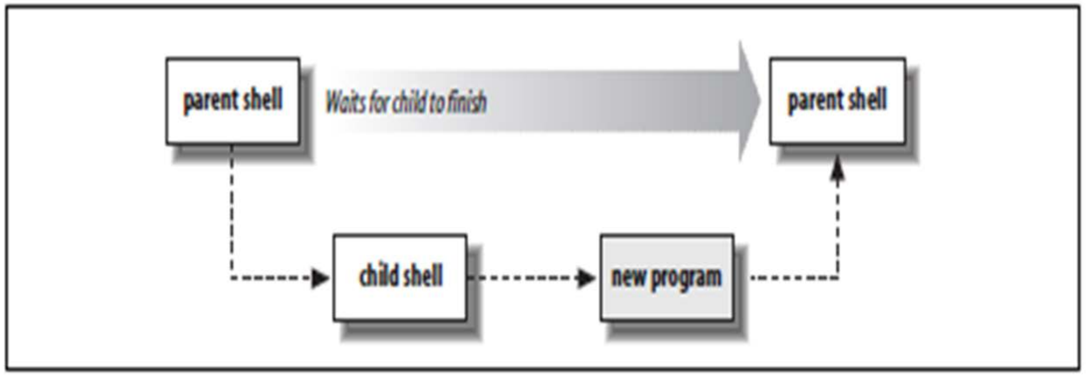
- Short options (-) and long options (--)
- in POSIX, use two dashes (– –) to signify end of options, i.e., remaining arguments on command line that look like options are treated as arguments (for example, as filenames).
  - To delete a file named "-l", rm -- -l
- Semicolons separate multiple commands on same line. The shell executes them sequentially.
- ampersand (&), tell shell to run preceding command in *background, which simply means that shell doesn't* wait for command to finish before continuing to next command.

Shell built-in commands

- Shell recognizes three kinds of commands: **built-in commands, shell functions, and external commands**
- Built-in commands: commands that shell itself executes
  - some from necessity:
    - cd to change current directory,
    - read to get input from the user (or a file) into a shell variable.
  - Other for efficiency:
    - test command, heavily used in shell scripting,
    - I/O commands such as echo or printf.
  - man cd will show all other shell bulit-in commands
- Shell functions are self-contained chunks of code, written in shell language

External commands

- **Implemented by another program**
- Shell runs by creating a separate process.
  1. Create a new process.
  2. In the new process, search directories listed in PATH variable for given command
     - /bin:/usr/bin:/usr/X11R6/bin:/usr/local/bin
     - Note: if command name contains /, skip this step
  3. In the new process, execute found program
  4. When the p
     - reading next



echo

- echo: produce output, prompting or to generate data for further processing.
- printed its arguments to standard output, with each one separated from next by a space and terminated with a newline

```
$ echo Now is the time for all good men
Now is the time for all good men
$ echo to come to the aid of their country.
to come to the aid of their country.
```

- Option: –n, omit trailing newline

```
$ echo -n "Enter your name: "   ##Print prompt
Enter your name: _ Enter data
```

Escape character

- To display special character, use –e option

  echo –e "Hello\tWorld"

- Code for special character
  - \a Alert character, usually the ASCII BEL character.
  - \b Backspace.
  - \c Suppress the final newline in the output. Furthermore, any characters left in the argument, and any following arguments, are ignored
  - \f Formfeed.
  - \n Newline.
  - \r Carriage return.
  - \t Horizontal tab.
  - \v Vertical tab.
  - \\ A literal backslash character.
  - \0*ddd Character represented as a 1- to 3-digit octal value.*

Outline

- Shell command line syntax
- Shell builtin commands
- Shell variables, arguments
- I/O redirection
- Shell tracing
- Shell initialization

Variables

- A variable is a name that you give to a particular piece of information.
- Shell variable **names**: start with a letter or underscore, and may contain any number of following letters,digits,or underscores.
- Shell variables **hold string values**, there is no limit on length of string value
  - variable values can be, and often are, empty—that is, they contain no characters.
  - Empty values are referred to as ***null***

Variable assignment

- Assign value to variable: writing variable name, immediately followed by an = character, and new value, without any intervening spaces.

  myvar=this_is_a_long_string_that_does_not_mean_much

  first=isaac middle=bashevis last=singer ##Multiple assignments allowed on one line

- Shell variable *values are* **retrieved** by prefixing the variable's name with a $ character.

  echo $myvar ## display *the value of myvar*

  this_is_a_long_string_that_does_not_mean_much

Variable assignment

- Use quotes when assigning a literal value that contains spaces:

  fullname="isaac bashevis singer" #*Use quotes for whitespace in value*

  oldname=$fullname #*Quotes not needed to preserve spaces in value*

- To concatenate variables:

  fullname="$first $middle $last" *Double quotes required here*

Command Substitution

- **We can save output of a command into variable**

```
$curr_dir=`pwd`  ##save current directory in a var.
$Curr_time=`date`
$echo $curr_time
Tue Jan 22 09:39:22 EST 2013
```

- **Command substitution**
  - One can embed a command with a backquote (`) in another command line
  - Shell will run embedded command, and use its output to replace the quoted part of original command

```
echo  Time is now `date`
echo  There is `who | wc –l` users online.
```

Example CountFiles script

- Count files/directories in a directory

```
#!/bin/bash
# List the number of files (including those hidden files) and
  directories under the given directory

echo count the number of files under $1
ls –a –L $1 | wc –l
```

Positional/argument parameters

- positional parameters represent a shell script's command-line arguments, also represent a function's arguments within shell functions.

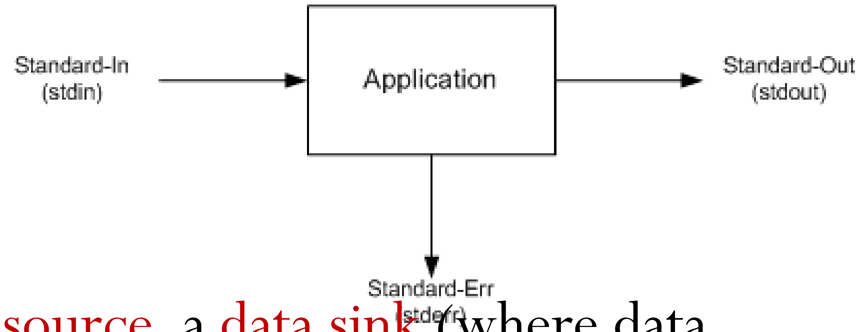
  echo first arg is $1

  echo tenth arg is ${10} ## For historical reasons,you have to enclose number in braces if it's greater than nine

- Other special argument variables:
  - $#: the number of parameters
  - $0: the command/script name
  - $*,$@: the list of all parameters ($1, $2, …), not including $0

Outline

- Shell command line syntax
- Shell builtin commands
- Shell variables, arguments
- Standard I/O, I/O redirection, Pipeline
- Shell tracing
- Shell initialization



Standard I/O

- All programs should have a data source, a data sink (where data goes),and a place to report problems. These are *standard input, standard output, standard error.*
  - Standard input, by default is linked to keyboard
  - Standard output, by default is linked to terminal window
  - Standard error, by default linked to terminal window
- *A program should neither* know, nor care, what kind of device lies behind its input and outputs: disk files,terminals, tape drives,network connections,or even another running program!
- A program can expect these standard places to be already open and ready to use when it starts up.

Simple example

- A very simple C program

```
#include <stdio.h>
main() {
    char yourName[256];

    printf ("Your name ?\n"); // Similar to cout
    if (fgets (yourName,256,stdin)==NULL)  //similar to cin
        fprintf (stderr,"No input");
    else
        printf("hello, %s\n", yourName);
}
```

Input/Output Redirection

- On command line, one can redirect these three files
- To redirect standard output to a disk file:
  - command [[-] option (s)] [option argument (s)] [command argument (s)] > FILENAME
  - Execute the command, sending its standard output to specified file
    - Existing content of the file is deleted
  - E.g.: ls –lt > InfoFilelist.txt
- To append standard output to a file: use >> instead of >
  - grep "tax reform" *.txt > output
  - grep "fuel efficiency" *.txt >> output

Input/Output Redirection (cont'd)

- To redirect standard error to a file

  $ command [[-] option (s)] [option argument (s)] [command argument (s)] 2> ERRORMSGS

- Examples:

```
[zhang@storm ~]$ ls abc
ls: cannot access abc: No such file or directory
[zhang@storm ~]$ ls abc 2> error
[zhang@storm ~]$ more error
ls: cannot access abc: No such file or directory
```

User > and 2> together

- To split error messages from normal output

```
[zhang@storm ~]$ ls research.tex abc
ls: cannot access abc: No such file or directory
research.tex
[zhang@storm ~]$ ls research.tex abc  2> error > output
[zhang@storm ~]$ cat error
ls: cannot access abc: No such file or directory
[zhang@storm ~]$ cat output
research.tex
```

- **This is useful for running a command that might take long time to finish, or generates very long output …**

More on redirection

- To redirect both output and error to same file:
  - ./a.out > dd 2> dd : does not work. Error output is not captured.
  - sort file.txt > dd 2>&1
    - 2>&1: redirect error output to same place as standard output
  - grep numOfStudents 2>dd >&2
    - >&2: redirect standard output to same place as error output
- To discard output, redirect it to /dev/null
  - /dev/null: a special virtual file, “a black hole”

  ./a.out > /dev/null 2>&1
  - I don’t want to see the output or error message, nor do I want them saved to a file …

Input/Output Redirection (cont'd)

- To read standard input from a file, instead of keyboard

  $ command [[-] option (s)] [option argument (s)] [command argument (s)] < FILENAME

- Examples
  - mail zhang –s "Question" < proj1.cpp
  - ./a.out < values.txt

  //a.out is your program that reads integers from standard input and calculate the sum

Combining commands together

- How many files are there under current directory ?

```
ls > tmp
wc –l < tmp
rm tmp
```

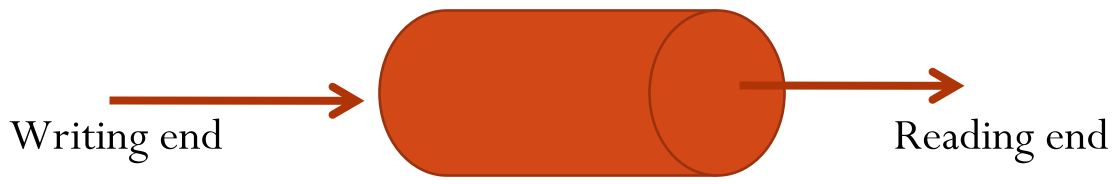
Is file “tmp” listed ?

- Sort current online user by alphabetic order

- Is some user login to the system now ? (using grep)

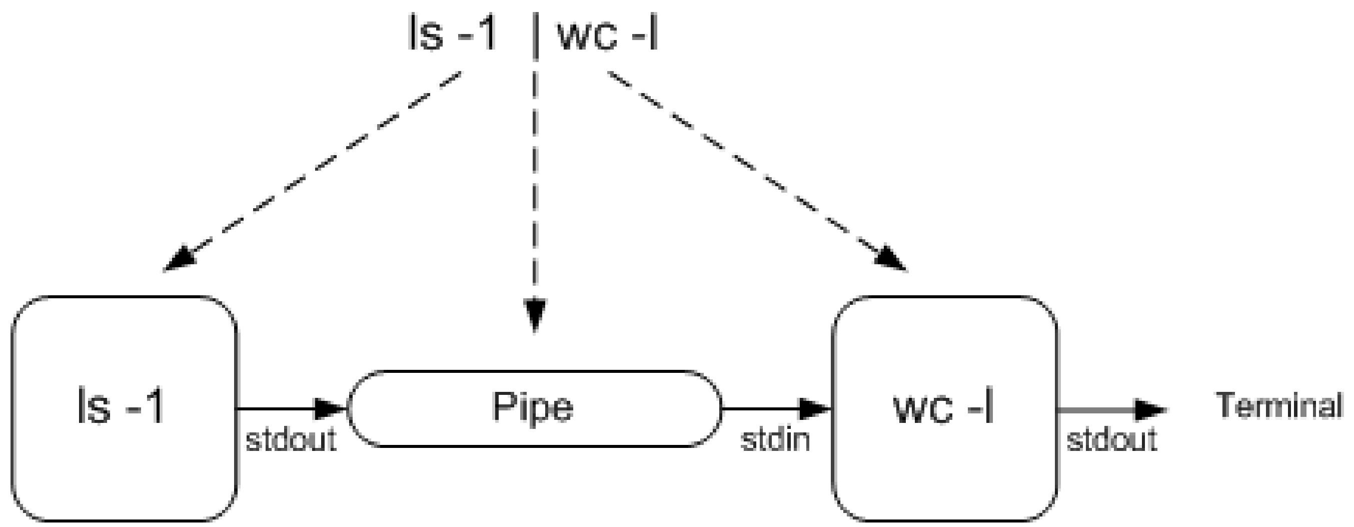
Pipe: getting rid of temporary file

- Pipe: an inter-process communication mechanism provided by kernel
  - Has a reading end and a writing end
  - Any data write to writing end can be read back from reading end
  - Read/write pipe is no different from read/write files, i.e., any prog. that reads from standard input can read from pipe, similarly for the standard output



Command Pipeline

- Shell set things up
  - create a pipe, "start" two programs simultaneously, with the first program's output redirected to writing end of pipe, second program's input redirected to reading end of pipe
  - individual program/command knows nothing about redirection and pipe



Rule of composition

- Design programs to be connected with other programs
  - Read/write simple, textual, stream-oriented formats
  - Read from standard input and write to standard output
- Filter: program that takes a simple text stream on input and process it into another simple text stream on output

The Power of Pipe

- Find out how many subdirectories are there ?

- Display the content of last edited file (under current directory)…
  - cat \`ls –t | head -1\`

Shell command line

- A command ends with a newline, or a semicolon (;), or an ampersand (&)
  - date;
  - sleep 4; who
  - sleep 20&who
- What's the output ?
  - date; who | wc
    - | has higher precedence over ;
  - ls –l | grep ^d &
    - | has higher precedence over &
  - Use parenthesis to group commands
    - (date;who) | wc

Outline

- Shell command line syntax
- Shell builtin commands
- Shell variables, arguments
- I/O redirection
- Shell tracing
- Shell initialization

C/C++ topics: command line arguments

- We learnt how to access command line arguments from shell, how about in C/C++ Program?
- Example: write your own echo program
  - echo: display a line of text

    $echo Good morning, everyone !

    Good morning, everyone!
  - In C/C++, command line arguments are passed as parameters to main function
    - main(int argc, char * argv[])
    - argc: number of command line arguments, including command itself
    - argv: the arguments
      - argv[0]: the first word in the command line (the command name)
      - argv[1]: the second word in the command line

Simplified Echo program

- Does not take options yet

```
#include <iostream>
using namespace std;

int main(int argc, char *argv[])
{
        for (int i=1;i<argc; i++)
        {
                cout <<argv[i]<<" ";
        }
        cout <<endl;
}
```

```
char * argv[ ];
char argv[][[];
```

--- argv is an array of “char *”.

In C, there is no string class, and string is represented as an array of char.

```
char myName[256];
char * name;
```

name = myName;
A array variable actually stores the address of the first element.

Outline

- Shell command line syntax
- Shell builtin commands
- Shell variables, arguments
- I/O redirection
- Shell tracing
- Shell Initialization and Termination

User Customization

- shells read certain specified files on startup, and for some shells,also on termination.
- We focus on bash here (different shell behaves differently)
- If you write shell scripts that are intended to be used by others, you *cannot rely on* startup customizations. All of the shell scripts that we develop in this book set up their own environment (e.g., the value of $PATH) so that anyone can run them.

Login Shell versus Non-login Shell

- **Login shell:** The shell that you talks to right after log in (from terminal, or remote log in using ssh command)
- **Nonlogin shell:** the shell that you runs by typing "shell" command, or by running a shell script
- Variable $0: indicates what shell you are in right now. Why?

```
[zhang@storm Codes]$ echo $0
-bash                                    the "-" indicates it's a login shell
[zhang@storm Codes]$ bash ## run a bash program,
[zhang@storm Codes]$ echo $0
bash                                     this is nonlogin shell
[zhang@storm Codes]$ exit
exit                                     exit the bash program
[zhang@storm Codes]$ echo $0
-bash                                    back to login shell
```

Source command

- A shell builtin command
- Usage:

  . filename [arguments]

  source filename [arguments]

  Read and execute commands from filename in current shell environment, and return exit status of last command executed from filename.
- Demo: difference of running a script directly and source it

  ```
  $./CountFiles
  $source CountFiles
  ```
- Why?
  - When running a script directly, a new shell (non-login, non-interactive shell) is started to batch processing script …

Bash: startup initialization

- For login shell:

```
test -r /etc/profile && . /etc/profile
if test -r $HOME/.bash_profile ; then
    . $HOME/.bash_profile
elif test -r $HOME/.bash_login ; then
    . $HOME/.bash_login
elif test -r $HOME/.profile ; then
    . $HOME/.profile
fi
```

Try to read /etc/profile

Try three more possibilities

/etc/profile: System wide default, setting environment for all shell.
/etc/bashrc: System wide function and aliases for bash

Shell: startup initialization

- **non-login interactive shell** :

test -r $HOME/.bashrc && . $HOME/.bashrc *Try to read $HOME/.bashrc*

- **Non-login non-interactive shell**:

test –r "$BASH_ENV" && eval . "$BASH_ENV"

One can set BASH_ENV to point to an initialization file.

Export command

- Take a look at typical settings
- export command: a bulit-in command
  - Puts given variable into environment, a list of name-value pairs available to all programs
    - Will learn how to access environment from C/C++ program
  - A child process inherits environment from parent process
    - Variables not in environment not inherited
- When setting PATH, needs to put it into environment, unless only for current script
  - examples

To test your settings

- To test your changes to login shell initialization setting:
  - Reloggin
  - Run a script from current shell
    - source .bashrc , or . .bashrc
    - Change current shell's settings

Summary

- Shell command line syntax
- Shell builtin commands
- Shell variables, arguments
  - Argument variables
  - Command substitution
- I/O redirection, pipe
- Shell initialization