Shells

(Also known as: Unix Command Interpreters)

Sistemas Operativos I

Shell as a user interface

- A shell is a special application that provides an interface for running other applications.
- There are many shells, some popular ones are:

Name	Path
Bourne shell	/bin/sh
Bourne-again shell	/bin/bash
C shell	/bin/csh
Korn shell	/bin/ksh

Most popular in Linux systems

Shell as a user interface

- A shell is a command-line interpreter that reads user input (commands and options) and executes programs

- User input is typically read from:
  - A Terminal (interactive shell)
  - Shell scripts (more about this in other set of slides)

Running a Program

When you type in the name of a program and some command line options, the shell:

- 1) reads this line: break it into tokens to identify special symbols (e.g. *, |, >>, <), commands, files, directories, options)
- 2) finds the program and runs it, feeding it the options you specified

Note: Here the steps are simplified. The interpreter execute more steps to perform this action

File Descriptors

- When the shell runs a program it establishes three I/O channels (STDIN, STDOUT and STDERR)

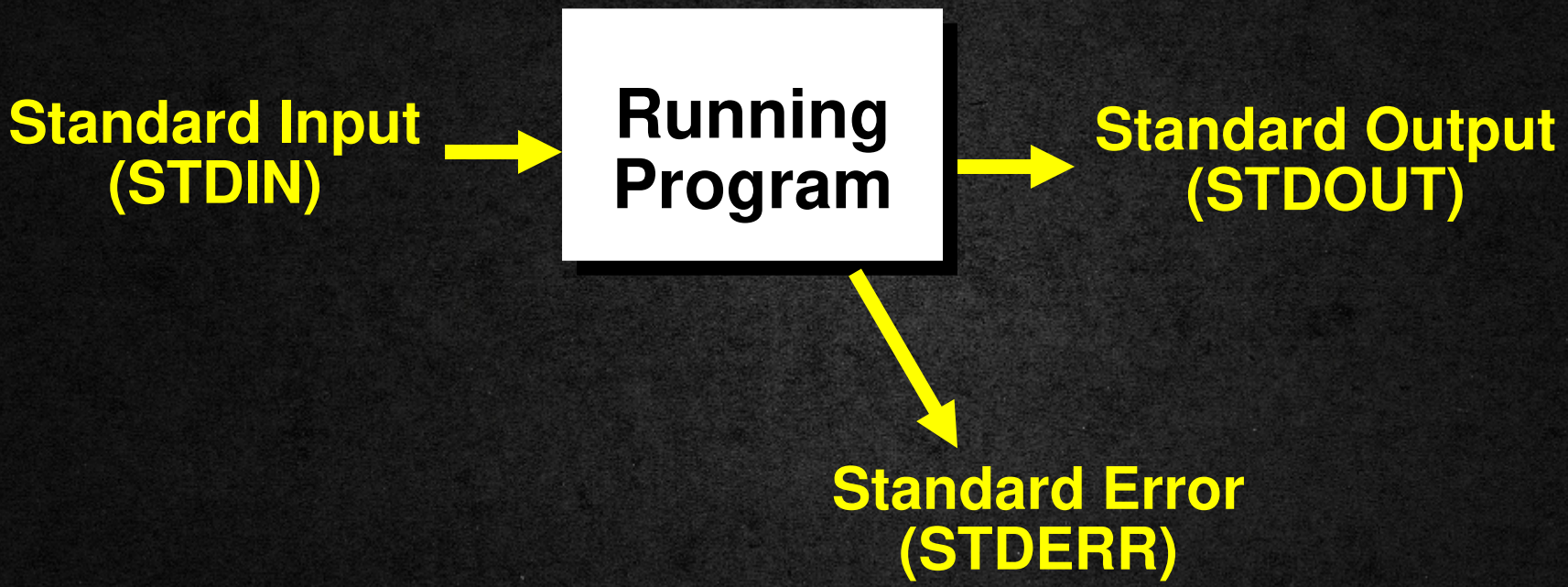
- These channels are files and are represented in unix by small-integers known as file descriptors:

  Standard Input is associated with integer 0

  Standard Output is associated with integer 1

  Standard Error is associated with integer 2

Programs and Standard I/O



Defaults for I/O

When a shell runs a program for you:

- standard input is your keyboard
- standard output is your screen/window
- standard error is your screen/window

Most Unix commands (programs):

- read something from standard input
- send something to standard output (typically depends on what the input is!)
- send error messages to standard error

Input Redirection

- The shell can attach things other than your keyboard to standard input
  - A file (the contents of the file are fed to a program as if you typed it)
  - A pipe (the output of another program is fed as input as if you typed it)

Output Redirection

- The shell can attach things other than your screen to standard output (or stderr)
  - A file (the output of a program is stored in a file)
  - A pipe (the output of a program is fed as input to another program)

Output Redirection Example

- To tell the shell to store the output of your program in a file, follow the command line for the program with the “>” character followed by the filename:

ls > lsout.txt

the command above will create a file named **lsout.txt** and put the output of the **ls** command in the file.

Input Redirection Example

- To tell the shell to get standard input from a file, use the “<“ character:

sort < nums.txt

- The command above would sort the lines in the file nums.txt* and send the result to stdout

* In this case, using < to redirect input to nums.txt yields the same result as giving a **sort** command with nums.txt as parameter

You can do both!


```
sort < nums > sortednums

tr a-z A-Z < letter > rudeletter
```

Output and Output Append

- The command **ls > foo** will create a new file named foo (deleting any existing file named foo).

- If you use **>>** the output will be **appended** to foo:

```
ls /etc >> foo
ls /usr >> foo
```

All about redirection

There are 3 file descriptors: 0(stdin), 1(stdout) and 2(stderr) -std=standard-

Basically you can:

- redirect stdout to a file
- redirect stderr to a file
- redirect stdout to a stderr
- redirect stderr to a stdout
- redirect stderr and stdout to a file
- redirect stderr and stdout to stdout
- redirect stderr and stdout to stderr

Pipes

- A pipe is a holder for a stream of data
- A pipe can be used to hold the output of one program and feed it to the input of another

Asking for a pipe

- Separate 2 commands with the “|” character
- The shell does all the work!

ls | sort Where the output of sort goes?

ls | sort > sortedls and now?

Building commands

- You can "pipe" together a series of unix commands to do something new!

- Exercises:
  - List all files in the current directory but only use upper case letters.
  - List only those files that have permissions set so that anyone can write to the file.

tee

- Allows to redirect input to standard input and also one or more files. Very useful with pipes!
- Example:

```
find /home -name '*.txt' | tee search.output |
                    grep “poems”
```

```
cut -f2 students.csv | tee students.unordered |
                    sort -n > students.ordered
```

Note: More about find, cut and grep commands in text handling slides

Stderr

- Many commands send error messages to *standard error*.
  - This is a different *stream* than **stdout.**
- The “>” output redirection only applies to **Stdout** (not to **Stderr**).
- Try this:

ls foo blah gork > savedls

(I’m assuming there are no files named foo, blah or gork!).

Redirecting stderr

- To redirect stderr to a file you need to know what shell you are using.
- When using **sh**, **ksh** or **bash** it's easy:

```
ls foo blah gork 2> erroroutput
```

It's not so easy with csh...

Globbing for filename abbreviation

- The shell gives some characters (known as **wildcards** or **metacharacters**) a special treatment
- When the shell finds wildcards performs something known as filename expansion (also pathname expansion)
- Wildcards make it easy to specify filenames

Globbing for filename abbreviation

- Wildcards: *, ?, [abc], [a-c], [!abc]
- The shell expands a wildcard pattern by replacing your command line with one that includes a list of file names matching the pattern in the current directory

Globbing for filename abbreviation

Example: the wildcard expansion of the following ls command in a directory that contains text1.txt, text2.txt, text3.txt:

```
ls text*
```

results in an ls call with all this files as parameters:

```
ls text1.txt text2.txt text3.txt
```

Globbing for filename abbreviation

`*` Matches anything — **`ls *.doc`**

`?` Matches any single character

`ls Test?.doc`

`[abc…]` Matches any of the enclosed characters

`ls T[eE][sS][tT].doc`

`[a-z]` Matches any character in a range

`ls [a-z]*`

`[!abc…]` Matches any character except those listed

`ls [!0-9]*`

Shell Variables
sh / ksh / bash

- The shell also have variables that you can use with your commands
- Some of them are:

PWD	*current working directory*
PATH	*list of places to look for commands*
HOME	*home directory of user*
IFS	internal field separator
TERM	*what kind of terminal you have*
HISTFILE	*where your command history is saved*

Displaying Shell Variables

- Prefix the name of a shell variable with "$" (gets the value of a variable).
- The **echo** command will do:

```
echo $HOME
echo $PATH
```

- You can use these variables on any command line:

```
ls -al $HOME
```

You can also try **env** or **printenv** commands to display values of all shell variables

Setting Shell Variables

- You can change the value of a shell variable with an assignment command (this is a shell *builtin* command):

```
HOME=/etc
PATH=/usr/bin:/usr/etc:/sbin
NEWVAR="blah blah blah"
IFS='
'
```

The PATH

- Each time you give the shell a command line it does the following:
  - Checks to see if the command is a shell built-in.
  - If not - tries to find a program whose name (the filename) is the same as the command.
- The **PATH** variable tells the shell where to look for programs (non built-in commands).

echo $PATH

```
======= [foo.cs.rpi.edu] - 22:43:17 =======
/cs/hollingd/introunix echo $PATH
/home/hollingd/bin:/usr/bin:/bin:/usr/local/
  bin:/usr/sbin:/usr/bin/X11:/usr/games:/usr/
  local/packages/netscape
```

- The **PATH** is a list of ":" delimited directories.
- The **PATH** is a list and a *search order*.
- You can add stuff to your PATH by changing the shell startup file

Job Control

The shell allows you to manage *jobs*.

A job is a set of running programs (e.g. a single command or programs communicated through a pipe)

Managing jobs means:

- place *jobs* in the *background*
- move a job to the *foreground*
- suspend a job
- kill send a signal to kill a job

Background jobs

- If you follow a command line with "&", the shell will run the *job* in the background.
  - you don't need to wait for the job to complete, you can type in a new command right away.
  - you can have a bunch of jobs running at once.
  - you can do all this with a single terminal (window).

```
ls -lR > saved_ls &
```

Listing jobs

- The command *jobs* will list all background jobs:

```
> jobs
[1] Running      ls -lR > saved_ls &
>
```

- The shell assigns a number to each job (this one is job number 1).

Suspending and Killing the Foreground Job

- You can suspend the foreground job by pressing ^Z (Ctrl-Z).
  - Suspend means the job is stopped, but not dead.
  - The job will show up in the **jobs** output.

- You can *kill* the foreground job by pressing ^C (Ctrl-C).

Moving a job back to the foreground

- The **fg** command will move a job to the foreground.
  - You give **fg** a job number (as reported by the **jobs** command) preceeded by a %.

```
> jobs
[1] Stopped                 ls -lR > saved_ls &
> fg %1
ls -lR > saved_ls
```

Quoting - the problem

- We've already seen that some characters mean something special when typed on the command line: **`* ? [] < > | &`**
- What if we don't want the shell to treat these as special?
- For example, we really mean *, not all the files in the current directory:

```
echo here is a star *
```

Quoting - the solution

- To turn off special meaning - surround a string with double quotes:

```
echo here is a star "*"

echo "here is a star *"
```

Careful!

You have to be a little careful. Double quotes around a string turn the string in to a single command line *parameter*.

ls foo fee file?

Lists foo, fee and e.g. file1, file2,...

ls "foo fee file?"

ls: foo fee file?: No such file or directory

Quoting Exceptions

- Some *special* characters are **not** ignored even if inside double quotes:
- $ get variable value
- " the quote character itself
- \ slash is always something special (\n)

You can use \$ to mean $ or \" to mean "

```
echo "This is a quote \" "
```

Single quotes

- You can use single quotes just like double quotes.
  - Nothing (except ') is treated special.

```
> echo 'This is a quote " '
This is a quote "
> echo 'This is a backslash \'
This is a backslash \
```

Backquotes are different!

- If you surround a string with backquotes the string is replaced with the result of running the command in backquotes:

```
> echo `ls`
foo fee file?
> PS1=`date`
Tue Jan 25 00:32:04 EST 2000
```

new prompt!

There is lots more...

- Check the book for more info on job control

- We also didn't talk about command history, it is very useful:
  - the shell keeps a list of commands that you entered
  - you tell the shell to repeat the same command again

Summary

- Bash Scripting

Bash is an acronym for Bourne Again SHell. The Bourne shell is the traditional Unix shell originally written by Stephen Bourne. All of the Bourne shell builtin commands are available in Bash, and the rules for evaluation and quoting are taken from the Posix 1003.2 specification for the 'standard' Unix shell.