Operating Systems

Section 05

Prepared by Eng. Abdelrahman Mahmoud

I/O Redirection

Standard Input, Output, and Error, pipelines

Intro

- The “I/O” stands for input/output
- The output often consists of two types:
  - The program's results, that is, the data the program is designed to produce
  - Status and error messages that tell us how the program is getting along
- Most programs display their results and their error messages on the screen.

Standard Output, and Error

- Programs actually send their results to a special file called standard output (often expressed as stdout) and their status messages to another file called standard error (stderr).
- By default, both standard output and standard error are linked to the screen and not saved into a disk file.

Standard Input

- Many programs take input from a facility called standard input (stdin).
- By default, standard input attached to the keyboard.

I/O redirection

- I/O redirection allows us to change where output goes and where input comes from. Normally, output goes to the screen and input comes from the keyboard, but with I/O redirection, we can change that.

Redirecting Standard Output

- To redirect standard output to another file instead of the screen, use the > redirection operator followed by the name of the file.

```
~$ ls /usr/bin > output.txt
```

- To display the content of output.txt

```
~$ less output.txt
```

Redirecting Standard Output

- Since we only redirected standard output and not standard error, the error message was still sent to the screen.

```
(base) abdo@abdo-HP-EliteBook-820-G1:~$ ls /bin/user > output.txt
ls: cannot access '/bin/user': No such file or directory
```

- When we redirect output with the “>” redirection operator, the destination file is always rewritten from the beginning

```
~$  > output.txt
```

- Using the redirection operator with no command preceding it will truncate an existing file or create a new, empty file.

Redirecting Standard Output

- We can append redirected output to a file instead of overwriting the file from the beginning using the >> redirection operator.

```
~$ ls /usr/bin >> output.txt
```

Redirecting Standard Error

- To redirect standard error to another file instead of the screen, use the 2> redirection operator followed by the name of the file.

```
~$ ls /bin/usr 2> error.txt
```

- We can also append the standard error streams to a single file.

```
~$ ls /bin/usr 2>> error.txt
```

Redirecting Standard Output and Standard Error to One File

- First way
  - we redirect standard output to the file output.txt and then we redirect file descriptor 2 (standard error) to file descriptor 1 (standard output) using the notation 2>&1.
  - Notice that the order of the redirections is significant.

```
~$ ls /usr/bin > output.txt 2>&1
```

Redirecting Standard Output and Standard Error to One File

- Second way
  - we use the single notation &> to redirect both standard output and standard error to the file output.txt

```
~$ ls /usr/bin &> output.txt
```

  - We can also append the standard output and standard error streams to a single file by &>>

```
~$ ls /usr/bin &>> output.txt
```

Disposing of Unwanted Output

- Sometimes we don't want output from a command, we just want to throw it away. This applies particularly to error and status messages. The system provides a way to do this by redirecting output to a special file called “/dev/null”. This file is a system device often referred to as a *bit bucket*, which accepts input and does nothing with it.

```
~$ ls /bin/usr 2> /dev/null
```

Redirecting Standard Input

- Using the < redirection operator, we change the source of standard input from the keyboard to the file.
- cat – Concatenate Files
  - The cat command reads one or more files and copies them to standard output. cat [file…]
  - If no files it read from Standard Input (stdin).
  - And Ctrl-d (i.e., hold down the Ctrl key and press “d”) to tell cat that it has reached end of file (EOF) on standard input.

Redirecting Standard Input

- cat copies standard input to standard output

```
(base) abdo@abdo-HP-EliteBook-820-G1:~$ cat
Hello cat
Hello cat
```

- We can make cat read from file instead of stdin Using the < redirection operator

```
~$ cat < output.txt
```

Pipelines

- Using the pipe operator | (vertical bar), the standard output of one command can be piped into the standard input of another. command1 | command2
- Pipelines are often used to perform complex operations on data. It is possible to put several commands together into a pipeline. Frequently, the commands used this way are referred to as *filters.*
    - command1 | command2 | command3 | command4

```
~$ ls /bin /usr/bin | sort | uniq | wc -l
```

Some commands

- uniq - Report or Omit Repeated Lines
- wc – Print Line, Word, and Byte Counts
- grep – Print Lines Matching a Pattern
- head / tail – Print First / Last Part of Files
- tee – Read from Stdin and Output to Stdout and Files
- echo – Display a line of text

Notes

- (see the man page for details) ex: man grep
- All above commands can use in *filters*

Seeing the World as the Shell Sees It

Expansion, Quoting

Expansion

- *bash* performs several substitutions upon the text before it carries out our command. The process that makes this happen is called expansion.

```
[me@linuxbox ~]$ echo *
Desktop Documents ls-output.txt Music Pictures Public Templates
Videos
```

- Why didn't echo print *? As we recall from our work with wildcards, the * character means match any characters in a filename

Pathname Expansion

- The mechanism by which wildcards work is called pathname expansion

```
(base) abdo@abdo-HP-EliteBook-820-G1:~$ echo /*/*/bin
/opt/wine-stable/bin /usr/local/bin
```

- Pathname Expansion of Hidden Files

```
~$ echo .*
```

Tilde Expansion

- the tilde character (~) has a special meaning. When used at the beginning of a word, it expands into the name of the home directory of the named user or, if no user is named, the home directory of the current user.

Arithmetic Expansion

- The shell allows arithmetic to be performed by expansion. This allows us to use the shell prompt as a calculator.
- Arithmetic expansion uses the following form:
    - $((expression))
  - where expression is
  - an arithmetic expression
  - consisting of values
  - and arithmetic operators.

Table 7-1: Arithmetic Operators

Operator	Description
+	Addition
-	Subtraction
*	Multiplication
/	Division (but remember, since expansion supports only integer arithmetic, results are integers).
%	Modulo, which simply means “remainder.”
**	Exponentiation

Arithmetic Expansion

- To multiply 5 squared by 3

```
(base) abdo@abdo-HP-EliteBook-820-G1:~$ echo $(($((5**2)) * 3))
75
(base) abdo@abdo-HP-EliteBook-820-G1:~$ echo $(((5**2) * 3))
75
```

Brace Expansion

- Perhaps the strangest expansion is called brace expansion. With it, we can create multiple text strings from a pattern containing braces. Here's an example:

```
(base) abdo@abdo-HP-EliteBook-820-G1:~$ echo Front-{A,B,C}-Back
Front-A-Back Front-B-Back Front-C-Back
(base) abdo@abdo-HP-EliteBook-820-G1:~$ echo {001..15}
001 002 003 004 005 006 007 008 009 010 011 012 013 014 015
(base) abdo@abdo-HP-EliteBook-820-G1:~$ echo {Z..A}
Z Y X W V U T S R Q P O N M L K J I H G F E D C B A
(base) abdo@abdo-HP-EliteBook-820-G1:~$ echo a{A{1,2},B{3,4}}b
aA1b aA2b aB3b aB4b
```

Parameter Expansion

- Many of its capabilities have to do with the system's ability to store small chunks of data and to give each chunk a name. Many such chunks, more properly called variables, are available for our examination. For example, the variable named USER contains our username. To invoke parameter expansion and reveal the contents of USER we would do this:

```
(base) abdo@abdo-HP-EliteBook-820-G1:~$ echo $USER
abdo
```

- To see a list of available variables, try this: `~$ printenv | less`

Command Substitution

- Command substitution allows us to use the output of a command as an expansion.

```
[me@linuxbox ~]$ echo $(ls)
Desktop Documents ls-output.txt Music Pictures Public Templates
Videos
```

```
~$ file $(ls -d /usr/bin/* | grep zip)
```

- In this example, the results of the pipeline became the argument list of the file command.
- in older shell programs, It uses backquotes instead of the dollar sign and parentheses

```
~$ file `ls -d /usr/bin/* | grep zip`
```

Quoting

- how we can control expansions.

```
(base) abdo@abdo-HP-EliteBook-820-G1:~$ echo this is a        test
this is a test
(base) abdo@abdo-HP-EliteBook-820-G1:~$ echo The total is $100.00
The total is 00.00
```

- In the first example, word-splitting by the shell removed extra whitespace from the echo command's list of arguments.
- In the second example, parameter expansion substituted an empty string for the value of $1 because it was an undefined variable. The shell provides a mechanism called quoting to selectively suppress unwanted expansions.

Double Quotes

- If we place text inside double quotes, all the special characters used by the shell lose their special meaning and are treated as ordinary characters.
- But parameter expansion, arithmetic expansion, and command substitution still take place within double quotes.

```
(base) abdo@abdo-HP-EliteBook-820-G1:~$ echo "this is a        test"
this is a        test
(base) abdo@abdo-HP-EliteBook-820-G1:~$ echo "The total is $100.00"
The total is 00.00
```

Double Quotes

```
(base) abdo@abdo-HP-Elite‌Book-820-G1:~$ echo $(cal)
مارس2020 ح ن ث ر خ ج س1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
(base) abdo@abdo-HP-Elite‌Book-820-G1:~$ echo "$(cal)"
     مارس2020
 ح  ن  ث  ر  خ  ج  س
 1  2  3  4  5  6  7
 8  9 10 11 12 13 14
15 16 17 18 19 20 21
22 23 24 25 26 27 28
29 30 31
```

- In the first instance, the unquoted command substitution resulted in a command line containing 38 arguments. In the second, it resulted in a command line with one argument that includes the embedded spaces and newlines.

Single Quotes

- If we need to suppress all expansions, we use single quotes.

```
(base) abdo@abdo-HP-EliteBook-820-G1:~$ echo 'The total is $100.00'
The total is $100.00
```

Escaping Characters

- Sometimes we want to quote only a single character. To do this, we can precede a character with a backslash, which in this context is called the escape character. Often this is done inside double quotes to selectively prevent an expansion.

```
(base) abdo@abdo-HP-EliteBook-820-G1:~$ echo "The total is \$100.00"
The total is $100.00
```

- It is also common to use escaping to eliminate the special meaning of a character in a filename. For example, it is possible to use characters in filenames that normally have special meaning to the shell. These would include $, !, &, spaces, and others.

Escaping Characters

- To allow a backslash character to appear, escape it by typing \\.
- Note that within single quotes, the backslash loses its special meaning and is treated as an ordinary character.

```
(base) abdo@abdo-HP-EliteBook-820-G1:~$ echo "\\"
\
(base) abdo@abdo-HP-EliteBook-820-G1:~$ echo '\'
\
```

Escaping Characters

- The backslash is also used as part of a notation to represent certain special characters called control codes
- The first 32 characters in the ASCII coding scheme are used to transmit commands to tele-type-like devices. Some of these codes are familiar (tab, backspace, linefeed, and carriage return), while others are not (null, end-of-transmission, and acknowledge).

Escaping Characters

Escape Sequence	Meaning
\a	Bell (an alert that causes the computer to beep)
\b	Backspace
\n	Newline. On Unix-like systems, this produces a linefeed.
\r	Carriage return
\t	Tab

- Adding the -e option to echo will enable interpretation of escape sequences.You may also place them inside $' '

```
(base) abdo@abdo-HP-EliteBook-820-G1:~$ echo -e "Time's up\a"
Time's up
(base) abdo@abdo-HP-EliteBook-820-G1:~$ echo "Time's up" $'\a'
Time's up
```