OPS 102

OPERATING SYSTEMS for PROGRAMMERS

Windows Scripting

Chris Tyler

Windows vs. Linux Scripting

Windows shell scripting is similar to Linux shell scripting in many ways.

However, since Windows and Linux (and other Unix-like operating systems) have different technical heritages, some of the syntax and approaches are very different . . .

In this slide deck, many of the examples are roughly equivalent to the examples in the *Linux Shell Scripting* slides, and it may be useful to compare them side-by-side.

An Abundance of Shells

We've looked at scripting on Linux systems using the *bash* shell – which is the most widely-deployed shell on Linux and other Unix-like systems. There are many other shells on similar systems, including:

- sh – the original Unix shell ("Bourne Shell"), written by Steven Bourne
- ksh – the Korn shell, written by David Korn
- csh – the C shell, which has a C-like syntax
- fish – the Friendly Interactive shell
- Zsh – the Z shell, similar to ksh

Many of these have a syntax based on and similar to the original Bourne shell, which is standardized in the POSIX.1 standard (or IEEE Standard 1003.1), and these shells have a lot in common.

An Abundance of Shells

On Windows, there are two main shells:

- CMD – The Windows command shell, which is the traditional Windows shell. Although based on traditional DOS command syntax, the CMD shell has been considerably expanded, with many new features added over the last few years.

- PowerShell – this is a new Windows command shell, which combines scripting with object-oriented programming. It is pre-installed for interactive use on current Windows systems; however, the execution of PowerShell scripts is disabled by default on all Windows "client" (non-server) systems, including Seneca lab computers.

Due to PowerShell scripting being disabled by default, and because object oriented programming is not taught in the first semester programming courses, we will focus on scripting using the traditional Windows CMD shell.

Cross-Platform Scripting

There are several possible approaches to writing scripts that will work on both Linux and Windows systems, as well as other common operating systems such as Mac OS:

- Bash or Zsh – bash is the default shell on most Linux systems, and zsh is the default shell on most Mac OS systems; either shell can be easily used on either system. Both shells are available as third-party add-ons for the Windows operating system, usually shipped with a collection of GNU utilities compiled for use with Windows (for example, see https://gitforwindows.org/)

- PowerShell – installed by default on current Windows systems, it is also available for Linux and Mac OS systems, although it is not commonly used on those systems yet (see https://github.com/PowerShell/PowerShell)

Other Interpreted Languages

In addition to shell scripting languages, there are other interpreted languages that are well suited for cross-platform development, including Python and Perl.

When should you use a shell scripting langauge?

- Shell scripting languages are ideally suited for process control – executing, managing, and combining external programs to accomplish tasks.
- Shell scripting languages are *not* well-suited for implementing advanced processing algorithms, because they generally lack features such as good floating-point support, typed variables, and strong support for large arrays and hashes.

Windows Shell Scripts vs Batch Files

DOS and early Windows systems were inherently interactive in nature, and early scripts were called "batch files" because they were viewed as similar to non-interactive *batch processing* on mainframe systems.

This terminology has stuck, and Windows shell scripts are still often called "batch files" (hence the occasional use of the .bat extension instead of .cmd).

The concept of a "batch file" and a "shell script" are roughly equivalent.

Basic Requirements for Shell Scripts

Remember these requirements? They apply to Windows scripts too:

1. Create a text file containing shell commands.
2. Tell the operating system *which* shell to use to execute the commands.
3. Ensure that the script file has the appropriate permissions.

1. Create a file containing shell commands

- Use any text editor
- Use the same commands that you would type at the command-line (though in Windows, there are a few small syntax differences between the command-line and script files)
- Save the file

Moving text files between Systems: Line Endings

Windows traditionally places a carriage-return (CR, code 13, \r) and a newline character (NL, code 10, \n) at the end of each line. Linux traditionally places only a newline character at the end of each line. When transferring files between systems, the end-of-line codes will often be translated, but when this does not take place, it can cause confusing issues. The *dos2unix* and *unix2dos* utilities on Linux systems (including Matrix) can be used to force a conversion of the end-of-line characters.

2. Tell the operating system which shell to use

- In Windows, the filename extension is used to associate a file with a program, and this mechanism is used to associate a script with a command interpreter.
- For CMD scripts, the extension ".cmd" is used.
  - For historical reasons, the extension ".bat" is also accepted.
- For PowerShell scripts, the extension ".ps1" is used.
  - The reason that ".ps" isn't used is that that extension was already used for PostScript files.
  - We're not going to write PowerShell scripts in this course.

3. Set the correct permissions on the file

- On Windows, the ability to read the script file is sufficient (and this is the default permission, so no change is usually required for scripts that you create for your own use; the situation may be different for scripts that are shared to other machines over the network or to other users on your system).

A Word about Command Echos

Windows defaults to displaying each command in a script before executing it (the opposite of the default in the bash shell). If you do not want each command to be displayed, you can:

- Add an @ sign in front of each command, or
- Issue the `echo off` command.

Usually, you'll combine these in a script, using this as one of the first lines:

```
@echo off
```

Demo: Basic scripts

- Let's write some simple scripts using commands that we know

- Scripts act like any other executable file, so we can type the script name as a command. It is not necessary to include the `.cmd` extension. By default, the current directory is searched when looking for CMD script files, so as long as the script name does not collide with an existing command name and the script is in the current directory, just typing the name (without `.\`) is sufficient:

```
> scriptname
```

Setting Variables

- To set a variable:

```
set VARIABLE=VALUE
```

- Variable names start with a letter and can contain letters, numbers, and underscores.
- Case does not matter! **A** and **a** are the same variable.
- Don't put spaces on either side of the equal sign
- Note: unlike other languages such as C, you don't need to declare the variable or specify the type of data (e.g., integer, string) which it will hold.

Accessing Variables

- To access a variable value, surround the variable name with percent signs:

```
%VARIABLE%
```

- The value of the variable is substituted into the command.

Variables: A Simple Example

```
> set WHAT=World
> echo Hello %WHAT%
Hello World
> echo Hello %what%
Hello World

> type hello.cmd
@echo off
set WHAT=World
echo Hello %WHAT%
> hello
Hello World
```

Quoting in the Windows Shell

- Quoting in the Windows shell is very different from bash!
- Using single or double quotes causes the quotes themselves to be included as part of the string or argument in most cases, but not when dealing with a filename:

```
> echo "Hello"
"Hello"

> echo test > "test file"
```

- Escaping characters to remove their special meaning is performed using the carat ^ symbol in Windows.

Escaping in the Windows Shell

- Escaping in the Windows shell is very different from bash!
- Escaping characters to remove their special meaning is performed using the carat ^ symbol in Windows:

```
> echo Lost ^& Found
Lost & Found
```

- When piping, a CMD subshell is started for each command in the pipeline, and it is necessary to use triple carat symbols ^^^ to escape characters:

```
> echo Lost ^^^& Found | find "Lost"
Lost & Found
```

Variables vs Environment Variables

- By default, all variables are environment variables, inherited by child processes.
- Environment variables are commonly used to pass configuration information to programs and to configure how programs operate.
- Environment variables are used by all processes, not just the shell!

Common Environment Variables

Environment Variable	Purpose
CD	Current directory
TIME	Current time in HH:MM:SS format
DATE	Current date in local format
ERRORLEVEL	The error code / exit status of the last command executed
PATH	A semi-colon (;) separated list of directories to be searched for commands and scripts
PROMPT	The interactive shell prompt (see online help for special characters that may be included)
RANDOM	A random integer (0-32767)

Viewing Environment Variables

- See all current environment variables and their values:

```
> set | more
```

PATH and PROMPT Environment Variables

- The PATH and PROMPT environment variables are usually set using the corresponding PATH and PROMPT commands instead of the SET command. See the online documentation ('help prompt' or 'help path') for details:

```
> PROMPT COMMAND:
COMMAND:PROMPT $G$S
>

> PATH %PATH%;D:\SomeNewDirectory
```

Reading a Variable Value from Stdin: set /p

- You can read values from standard input (stdin) and assign them to a variable with the `set` command using the `/p` option (aka switch):

```
> set /p NAME=Enter your name:
Enter your name: J. Doe

> echo %NAME%
J. Doe
```

Demo: Variables in a Script

```
@echo off
set /p NAME=Please enter your name:
echo Please to meet you, %NAME%
set /p FILE=Please enter a filename:
echo Saving your name into the file...
echo NAME=%NAME% >> %FILE%
echo Done.
```

Arithmetic!

- CMD can do *integer* arithmetic (sound familiar?!)
- To evaluate an arithmetic expression and store the results in a variable, use the SET command with the /a option. (When used interactively, the result of the expression evaluation will be output to stdout; this doesn't happen in scripts).

```
> set A=100
> set B=12
```

```
> set /a X=A*B
1200
> echo %X%
1200
```

```
> set /A A+=1 >NUL:
> echo %A%
101

> set /a C=A*B*2 >NUL:
> echo The answer is
%C%
The answer is 2424
```

Arithmetic!

- You can perform more than one arithmetic evaluation and assignment in one SET command by separating the expressions with a comma (,)

- Some characters used in arithmetic expressions, such as the carat symbol, may need to be quoted or escaped to function correctly.

- Percent signs, when used in arithmetic expressions (as the modulo operator), need to be doubled (%%) to avoid confusion with the percent signs placed around variable names

Exit Status Code: ERRORLEVEL

- The special variable %ERRORLEVEL% can be used to find out the exit status of the last command executed:

```
> dir \foo\bar\baz
The system cannot find the path specified.


> echo %ERRORLEVEL%
1
```

Conditional logic: if / else

- The if command takes a test, and uses the result of the test to control the execution of one or more commands. An else clause is optional; if included the first conditional commands should be placed in parenthesis.

```
if test then list
```

```
rem * Note that the parenthesis below are required
if test then (list) else list2
```

Conditional logic: if / else

Examples:

```
> set A=Blue
> set B=Orange
> set C=Blue

> if %A%==%C% echo Strings A and C match
Strings A and C match
> if %A%==%B% (echo Same!) else echo Different!
Different!
```

Tests 1: Filesystem entries

- Tests that a filename exists (regardless of the entry type – file or directory)

  `EXIST` *`filename`* *filename* is an existing file or directory

Tests 2: String Equality

- Test for string equality.

  `string1==string2`   True if the strings match.

Tests 3: String and Numeric Comparisons

- These tests accept two string arguments, both strings or both integers, which are compared. Adding the /i switch will make string comparisons case-insensitive (UPPER/lowercase).

value1 EQU *value2*	True if the values are equal
value1 NEQ *value2*	True if the values are not equal
value1 LSS *value2*	True if the value1 less than value2
value1 LEQ *value2*	True if the value1 less/equal to value2
value1 GTR *value2*	True if the value1 less/equal to value2
value1 GEQ *value2*	True if the value1 less/equal to value2

Tests 4: Variable Definition, Errorlevel

- Test to see if a variable is defined

  `DEFINED` *`variable`* True if variable is defined

- Test to see if the ERRORLEVEL is above a threshold

  `ERRORLEVEL` *`value`* True if ERORRLEVEL>=value

  - Or just use: `%ERRORLEVEL% GEQ` *`value`*

Notes about IF and these Tests

- These tests work only with the IF command

- The IF command can be used with GOTO and a label:

```
    IF test GOTO :skip
    ...
    :skip
```

- Using a GOTO in a loop will make the shell forget about the loop, regardless of where the label is located.

Examples of using test: ERRORLEVEL

```
@echo off
VER | FIND "Version 10" >NUL:
IF ERRORLEVEL 1 (ECHO Not Windows 10.) ELSE ECHO Windows 10.

rem * The test above could be rewitten as %ERRORLEVEL% GTR 0
```

Negating and Combining Tests

- You can negate (invert) a test with the NOT operator:

```
IF NOT EXIST %N% ECHO The file "%N%" does not exist.
```

- You cannot combine tests – there is no and/or operator.

Examples of using test: Integers vs Strings

```
@echo off
rem intcmp.cmd - compare as integers
SET /A A=11,B=2
IF %A% GTR %B% (
       ECHO %A% is greater than %B%
) ELSE ECHO %A% is less than or equal to %B%
```

```
@echo off
rem strcmp.cmd - compare as strings
SET /A A=11,B=2
IF "%A%" GTR "%B%" (
       ECHO %A% is greater than %B%
) ELSE ECHO %A% is less than or equal to %B%
```

Examples of using test: Integer Numbers

```
@echo off
SET /A COIN=%RANDOM% %% 2
IF %COIN% EQU 0 (ECHO Heads!) ELSE ECHO Tails
```

Script Parameters

- It's useful to be able to call a script with positional parameters (arguments).
- These can be accessed within a script as %0, %1, %2, %3, and so forth.
- %0 is the name of the script itself.
- The `shift` command gets rid of the first parameter and shifts every parameter to a lower number.

Script Parameters

```
> type params.cmd
@echo off
ECHO PARAM 0: %0
ECHO PARAM 1: %1
ECHO PARAM 2: %2

> params red green blue
PARAM 0: params
PARAM 1: red
PARAM 2: green
```

Script Parameters

```
> type params-shift.cmd
@echo off
ECHO List of all arguments: %*
:start
IF "%1"=="" GOTO :done
ECHO %1
SHIFT
GOTO :start
:done
> params-shift yellow orange red
List of all arguments: yellow orange red
yellow
orange
red
```

Script Parameters

- %* returns ALL of the parameters

- On the command line, parameters may be separated by:
  - Space (or Tab)
  - Comma ,
  - Semicolon ;
  - Equal sign =

Looping

- Loops through list of files:
  - `FOR    `*`%variable`*` IN (`*`files`*`) DO `*`list`*
  - `FOR /D `*`%variable`*` IN (`*`files`*`) DO `*`list`*

- Loop through a range of numbers:
  - `FOR /L `*`%variable`*` IN (start; increment; end ) DO list`

Looping - Variables

- The names of the controlling variables in FOR loops are different from other variables:
  - Must be a single letter
  - The variable name is case sensitive
  - The variable name is written with a single preceding % sign at all times (not just when reading) when used from the command line, or double preceding %% signs inside a script

Looping - Delayed Expansion

- When the body of a FOR loop is executed, the variables are expanded (replaced by their values) before the loop begins. That means that any variables that are contained in the loop have their values *locked in* and they can't be changed while the loop is executing.
- To allow updated variable values to be accessed within a loop:
    1. Set the EnableDelayedExpansion option:
       ```
       SETLOCAL EnableDelayedExpansion
       ```
    2. Change any variables which will be updated during the execution of the loop by replacing the percent-signs (%) with exclaimation-marks (!):
       ```
       %ERRORLEVEL%    ->    !ERRORLEVEL!
       ```

Looping: FOR *variable* IN (*files*)

```
@echo off
SETLOCAL EnableDelayedExpansion
FOR %%F IN (*) DO (
        CHOICE /M "DELETE %%F"
        IF !ERRORLEVEL!==1 (
                ECHO ...Deleting %%F
                DEL %%F
        ) ELSE (
                ECHO ...Skipping %%F
        )
)
```

Looping: FOR *variable* IN *(files)*

- This type of loop accepts a list of files, or a filename pattern.
- The loop will iterate through the list of files, assigning each one sequentially to the control variable.
- Other strings can be used as though they are filenames as long as they do not contain wildcards – there is no test performed to see if the names actually exist or are potentially valid.

Looping: FOR /L (*start, step, end*) DO *list*

- This type of loop counts forward or backwards from *start* to *end* by a given *step*. Example:

```
@echo off
rem Count from 0 to 5 in increments of 1
FOR /L %%I IN (0,  1, 5) DO ECHO ... %%I ...

rem Count from 4 to 0 in increments of -1
FOR /L %%I IN (4, -1, 0) DO ECHO ... %%I ...
```