Operating Systems Assignment 0: Advanced shell scripting using Awk, Perl and make

February 9, 2011

1 Introduction

Whereas last week we have been introduced to shell programming using the various available utilities, this week we have discussed some more advanced tools that can be used in conjunction with shell scripting during class. Similar to last week, we will work our way through some exercises to gain hands-on experience with what we have seen in class today.

All data files used below are available at `http://www.liacs.nl/~krietvel/courses/os2011/lab00/files/` *and at* `http://132.229.136.243/~kris/lab00/` *(only in room 411).*

2 Awk

We will start off with a couple of exercises around *awk*. As we have seen this morning, *awk* is a very useful language to process text files filled with structured records.

Remember that you can execute *awk* scripts in two ways. The first way is to put your code on the command line:

```
awk 'condition { pattern }' file-to-process.txt
```

When no file is specified, awk will read from `stdin` so you can use pipes and/or redirection.

You can also put your script in a separate file and invoke *awk* as follows:

```
awk -f myscript.awk file-to-process.txt
```

Exercise 1. Recall the file we used in the first exercise last week (named *words*):

```
one two
three four
five six
seven eight
nine ten
```

We asked you to create a command that would remove the first word from each line and sort the result. Now, instead of using the `cut` utility, construct this command using *awk* instead.

Exercise 2. Provided a text file with lines containing 3 numbers each (named *numbers*):

```
1 2 3
4 3 6
123 235 924
35 924 5234
3 1 2
```

Write an *awk* script that will output the average of the three numbers for each line.

Exercise 3. Adapt the script written in exercise 2 to be able to process lines with varying amounts of numbers (named *numbers2*):

```
1 2 3 4 5
4 5
123 235 924 234 6123
35 5234 324 52
69
```

(Hint: the following works to access a field using a field number stored in a variable (it will print the first field): `i = 1; print $i;`).

Exercise 4. Using the data file from exercise 3, which you might want to extend a bit for testing, write an *awk* script that will display the first half of a line's fields (you can round the result of the division down). Also, on termination it should output the average number of fields encountered per line (before division).

Exercise 5. A file can also contain different field and record separators. For example (file available as *numbers3*):

```
1,2,3;4,3,6;123,235,924;35,924,5234;3,1,2
```

Modify the script written in exercise 2 to process this string, but only by adding a `BEGIN` clause with commands. The resulting output should be exactly the same.

Exercise 6. As a final *awk* exercise, write a script that will compute a histogram showing the occurrence count of the numbers in the file. For example, for the data file used in exercise 2, the output would be:

```
2: 2
3: 3
4: 1
6: 1
5234: 1
924: 2
35: 1
235: 1
123: 1
1: 2
```

3 Perl

Similar to *awk* you can either put your Perl script on the command line:

```
perl -e 'for ($i = 0; $i < 10; $i++) { print "$i\n"; }'
```

and in the above example be sure to use the correct bash quotes to avoid variable interpretation! Or simply create a new file containing your script and execute it:

```
perl filename.pl
```

Another good advice is to always use the `-w` command line option to Perl to enable output of all warnings.

We will practice a little Perl by rewriting some of the *awk* scripts we have written in the above exercises. This way, you can also get a sense of which language is more useful for which tasks.

Exercise 7. Write a Perl version of the script written in exercise 3. Remember that Perl will not divide the data into fields by default like *awk* does. Instead use the `split` function we have discussed in class. Before `split`, you will likely want to remove the newline character from the string using `chomp`.

Exercise 8. In exercise 6, we have written a script that can compute a histogram of numbers in a file. Rewrite this script in Perl.

As a nice touch, instead of simply printing the number of occurrences, you can make the output more "graph-like" using:

```
print "=" x $i
```

which will print a sequence of i equal signs.

*** Exercise 9.** An additional exercise for when there's time left: modify the script from exercise 8 to create a histogram of all digits instead of the numbers. To create an array of characters in a string, use: `@characters = split //, $string;` (no space between the slashes).

4 Make

For the following exercises it is easiest to create a new directory for each exercise and put your solution in a file named *Makefile* (note the capital "M" up front). You can test your solution by simply running `make` in that directory.

Exercise 10. Write a simple "Hello world" C program (or use *hello.c* as provided on the website). Construct a *Makefile* that will compile this program. Make smart use of (automatic) variables and generic rules so that your *Makefile* will be easily extensible in the future. Include a `clean` rule which will remove the files that are generated by the *Makefile*.

Exercise 11. Make can also be put to use if you have a project mainly written in a scripting language instead of C. When a script file is using a data file that is usually available on the system, but the exact location depends on the operating system used, then the script file is often modified on the target system before usage.

Consider the following Perl script (available as *word-avg.pl.in*):

```
#!/usr/bin/perl -w

$word_file = '@@WORD_FILE@@';

$sum = 0;
$count = 0;

open (FH, "< $word_file");
while (<FH>) {
        chomp;
        $sum += length;
        $count++;
}
close (FH);

printf "Average word length: %3.2f\n", ($sum / $count);

exit 0;
```

Create a *Makefile* which will create the destination file *word-avg.pl* by substituting `@@WORD_FILE@@` with `/usr/share/dict/words` (the location of the dictionary on these systems) using `sed`. After that it should also make the script executable using `chmod +x`.

Exercise 12. We can also use *make* to write a simple automatic download tool with resume capability. We have provided a webserver with a couple of large files at *http://132.229.136.243/~kris/lab00/*. You can use the following variables in your *Makefile*:

```
BASE = http://132.229.136.243/~kris/lab00/
FILES = 10M.bin 20M.bin 30M.bin 40M.bin
```

Write a *Makefile* which uses `wget` to download these files. Make sure to use a generic rule. Because `wget` will modify the filename if it is invoked for a file which already exists, we suggest you to use the `-O` command line flag and a `mv` command after the `wget`. For example, download *10M.bin* to *10M.bin.tmp* first (using `-O`), then move *10M.bin.tmp* to *10M.bin*.

Test your *Makefile* by interrupting it with Control-C. If you run *make* again, will it start downloading the first file again or continue with the file where it was interrupted?

As a final touch you can include a clean target which will remove all *.bin* files and also all *.bin.tmp* files (you can use a substitution reference for the latter).

Recall the `-j` flag to *make* we have seen during class, can we use this to turn our Makefile into a parallel download utility?