Operating Systems I
Practice 3: AWK tool

Objectives

Programming and automating tasks over text files using the *awk* tool

Usage and syntax

Awk[1] is a tool employed for processing text files, line by line, performing various actions over each line according to a series of conditions (patterns).

The basic syntax for calling *awk* is as follows:

```
awk [-Fc] [-v var=value …] awk_program text_files
```

where:

`-Fc` Sets character `c` as field separator on the text lines.

`-v var=value` Assigns the given value to the variable `var`. This `-v` option can only assign one variable, but it can be used multiple times.

`awk_program` *awk* program given as `pattern{action}` (one or multiple times). It can be passed as:

> `program`: the program given to *awk* is usually taken from text between single quotes, but it allows combining quoted and non-quoted parts as long as there are no whitespaces in between, for example `'{print $'1}` is equivalent to `'{print $1}'`
>
> `-f file`: the *awk* program is read from a text file.

`text_files` Files (one or more) to apply the *awk* program to. This can be replaced by the output of a previous command by using pipes.

An *awk* program consists of sentences of the form `pattern{action}`. Each line of the input file is matched against each `pattern` in the program, and if it matches, then its corresponding `action` is performed. Once the line does not match any more patterns, the next line is processed. If no `pattern` is specified, all lines are considered to match.

Example file

We will use `/etc/passwd` in the following examples. Each line of this file has 7 fields separated by the *colon* character (:). For example, in the following line:

```
user:XXXXX:1000:1001:username:/home/user:/bin/bash
```

the meaning of each field is:

[1] Fact: The name comes from the initials of its creators' surnames: Alfred **A**ho, Peter **W**einberger, Brian **K**ernighan. Furthermore, Kernighan is one of the co-authors of the C language. Now you have something cool to tell people at parties![2]

[2] That is, if people really want to hear this kind of stuff at such social gatherings.

1	user	Account name
2	XXXXX	Encrypted password
3	1000	Account UID
4	1001	GID of the main group to which this account belongs to
5	username	User name
6	`/home/user`	User working directory
7	`/bin/bash`	User command interpreter (shell)

Separator and fields

The default field separator is one or more whitespaces. The separator divides each record (line)[3] into fields. $0 is used to represent the entire current line, and $1...$*n* represent each individual field in the current line.

Example 1

```
#!/bin/bash
# a) print passwd
# b) print the result of ls -l

echo "passwd file"
awk '{print $0}' /etc/passwd
echo "File list"
ls -l | awk '{print $0}'
```

Example 2

```
#!/bin/bash
#
# Print the first field with separator " "
awk '{print $1}' /etc/passwd
#
# Print the first field with separator ":"
awk -F: '{print $1}' /etc/passwd
```

Example 3

```
#!/bin/bash
#
# Prints the first field, sorted alphabetically
# The separator is ":"
#
awk -F: '{print $1}' /etc/passwd | sort
```

Predefined variables

awk recognises, among others, the following variables:

FILENAME	Name of the file currently being processed.
FNR	Record number in the file currently being processed.
NR	Record number in the set of all files being processed. If only a file is processed, it equals FNR.
NF	Number of fields in a record.
FS	Input field separator (whitespace by default).
OFS	Output field separator (whitespace by default).
RS	Input record separator (`\n` by default).
ORS	Output record separator (`\n` by default).

examen: skip first line -> NR>1

[3] Record and line can be used interchangeably.

Example 4

```
#!/bin/bash
#
# Print the 2 input files numbering each line,
# first globally and then per file.

echo -e "\nGlobal numbering"  # with previous newline
awk '{print NR,$0}' /etc/passwd /etc/group
echo -e "\nPer file numbering"
awk '{print FNR,$0}' /etc/passwd /etc/group
```

Example 5

```
#!/bin/bash
#
# Prints fields 1 to 4 numbering each line
# with output field separator "-"
awk -F: '{OFS="-";print FNR,$1,$2,$3,$4}' /etc/passwd
# with output field separator ":"
awk -F: '{OFS=":";print FNR,$1,$2,$3,$4}' /etc/passwd
```

Patterns

The following are used for creating patterns or conditions:

- Record range given as `pattern1,pattern2`. Matches all input records beginning with a record that matches `pattern1`, and ending with a record matching `pattern2` (both inclusive).
- Comparisons with <, >, ==, >=, <=, ! =
- Logical operands && (AND), || (OR) and ! (NOT)
- Regular expressions given as `/reg_exp/`
- Tilde operator (`~`) to indicate "must be contained in the regular expresion", and its negation (`!~`) to indicate "not contained".

Example 6

BEGIN{print "***"}
END{print "pattern"}

you can add -v (variables)

counter in lines

```
#!/bin/bash
#
# Prints lines 3 to 12 of the file, numbering each line
#
# FNR is the file record counter
# NR is the global record counter
#
awk 'NR==3,NR==12{print NR,$0}' /etc/passwd
# or also
awk 'FNR>=3 && FNR<=12{print NR,$0}' /etc/passwd
```

Example 7

```
#!/bin/bash
#
# Prints fields 1, 5 and 7 (user, name and shell) of the
# users who use the bash shell.
# The separator is ":"
#
awk -F: '$7=="/bin/bash"{print $1,$5,$7}' /etc/passwd
```

Example 8

```
#!/bin/bash
#
# Prints fields 1 and 5 (user and name) of the users
# whose user accounts begin with "r" or "d".
# The separator is ":"
awk -F: '/^r/{print $1,$5}
         /^d/{print $1,$5} ' /etc/passwd
# or also
awk -F: '/^r/ || /^d/{print $1,$5} ' /etc/passwd
```

Example 9

```
#!/bin/bash
#
# Prints fields 1, 5 and 7 (user, name and shell) of the
# users who use the bash shell and whose user account begins with "r".
# The separator is ":"
#
awk -F: '$7~/^\/bin\/bash$/ && /^r/{print $1,$5,$7}' /etc/passwd
```

Besides, *awk* provides two special patterns: `BEGIN` and `END`, which can be used for performing actions before the first line is read and processed (`BEGIN`), and after the last line is read and processed (`END`).

Example 10

```
#!/bin/bash
#
# Prints the number of lines of the file
#
awk 'BEGIN{num=0}
    {num++}
    END{print "Number of lines:",num}' /etc/passwd
```

Example 11

```
#!/bin/bash
#
# prints fields 1 to 4 numbering each line
# with output separator ":"
awk -F: 'BEGIN{OFS=":"}
         {print FNR,$1,$2,$3,$4}' /etc/passwd
```

Variable passing

Using the option `-v var=value`, you can pass variables to *awk*. This option can only set one variable, but it can be used multiple times.

Example 12

```
#!/bin/bash
#
# Prints field 1 of record 5
#
awk -F: -v l=5 -v c=1 ' NR==l{print $c}
    END{print "Printed field "c" of record "l"}' /etc/passwd
```

Example 13

```
#!/bin/bash
#
# Prints a field of a record (given as parameters to the script:
# record is $1, field is $2)
#
awk -F: -v l=$1 -v c=$2 'NR==l{print $c}
        END{print "Printed field "c" of record "l"}' /etc/passwd

# another option, combining quoted and non-quoted parts
awk -F: 'NR=='$1'{print $'$2'}
        END{print "Printed field "'$2'" of record "'$1'}' /etc/passwd
```

Actions

Actions, appearing between curly braces, have a syntax similar to C:

```
    if (condition) statement [else statement]
    while (condition) statement
    for (initialization;condition;increment) statement
    break
    continue
    variable=expression
    print [expression list] [$>$file]
    printf format [,expression list] [$>$file]
    next (go to next record skipping the current one)
    exit (skip all records in the input file)
```

Example 14

```
#!/bin/bash
#
# Print (with formatted output)
# fields 1 and 5 (user and name).
# Field 1 is 20 characters wide and left-justified.
# The separator is ":"
#
awk -F: '{printf "%-20s %s\n",$1,$5}' /etc/passwd
```

Example 15

```
#!/bin/bash
#
# Print fields 1 and 5 (user and name) of the users whose account
# begins with "r", as long as field 5 is non-empty.
# The separator is ":"
#
awk -F: '/ˆr/ && $5!=""{print $1"\n-->\t"$5}' /etc/passwd
# equivalent form
awk -F: '/ˆr/{if ($5!="") print $1"\n-->\t"$5}' /etc/passwd
```

Example 16

```
#!/bin/bash
#
```

```
# Shows the record of the user passed as a parameter
# or the message "User xxx does not exist"
#
awk -F: -v user=$1 'BEGIN{exists=0}
    user==$1{print $0; exists=1}
    END{if (exists==0) print "User",user,"does not exist"}' /etc/passwd
```

Arrays

awk has matrices and one of its advantages is that indices do not have to be a sequential set of numbers: you can use strings or numbers as indices[4]!

Array operations:

- Create array element: `array[index]` (creates an entry with empty value) or `array[index]=value`
- Remove array element: `delete array[index]`
- Check index existence: `(var in array)`
- Loop array indices: `for (var in array) acciones`

Example 17

```
#!/bin/bash
# unique listing of users with processes
echo "Users with processes"
ps -ef | awk '!($1 in array){array[$1]} # creates array entry with empty value
              END{for (i in array) print i}'
#
# awk '{array[$1]}' also works, without the pattern condition
```

Example 18

```
#!/bin/bash
echo -e "Number of processes per user"
# number of processes each user has
ps -ef | awk '{array[$1]++}
              END{for (i in array) print i,array[i]}'
```

Example 19

```
#!/bin/bash
# Print file line by line in reverse order
echo -e "File in normal order"
cat file
echo -e "\n\nFile in reverse order"
awk '{num++;line[num]=$0}
     END{for (i=num;i>0;i--) print line[i]}' file
```

Functions

Also, *awk* supports a set of predefined functions, among which are:

[4] This is similar to Python syntax for dictionaries[5].

[5] Another fact: the name of the Python language comes from the Monty Python! And its documentation is full of references to their sketches! Another cool fact to tell at parties[6]!

[6] If you haven't been kicked out already, of course.

index(s,t) If *t* is a substring of *s*, returns the index in *s* where *t* begins. Otherwise, return 0.

length(s) Returns the length of the *s* string.

substr(s,m,n) Returns a substring of *s* taking *n* characters from position *m*.

getline Reads the next input record and assigns such record to $0.

system(command) Runs the command from a *shell* and returns the resulting *errorlevel*.

Example 20

```
#!/bin/bash
#
# Prints field 1 and its length, as long as it is larger than the
# parameter passed in the call.
#
awk -F: -v long=$1 '{if (length($1)>long) print length($1),$1}' /etc/passwd
```

Example 21

```
#!/bin/bash
#
# For users whose account begins with "r",
# show 5 characters beginning from the third, and then the entire line.
#
awk '/^r/{print substr($0,3,5)," -> ",$0}' /etc/passwd
```

Example 22

```
#!/bin/bash
# Shows the starting position of the "bash" string on each line
awk '{print index($0,"bash")," -> ",$0}' /etc/passwd
# Same, but only for field 7
awk -F: '{print index($7,"bash")," -> ",$7}' /etc/passwd
```

Example 23

```
#!/bin/bash
#
# Show values from 1 to 10
echo "All the lines:"
for ((i=1;i<=10;i++)) do echo $i; done | awk '{print $0}'

# Show only even values from 1 to 10
# Each time a line is read, jump to the next one
echo "Jumping to the next line:"
for ((i=1;i<=10;i++)) do echo $i; done | awk '{getline;print $0}'
```

Example 24

```
#!/bin/bash
#
# Redirects the output to a file and runs a shell command
#
awk '{print $0 >"delete"} END{system("cat delete | more")}' /etc/passwd
```

Exercises

Exercise 1 Create a command that performs the following actions:

- From the list of active processes, only show the proprietary user (UID), the process identifier (PID), and the executed command (CMD).
- Modify the previous command so as to only show the processes belonging to users whose account name begins with “r”.

Tip: use this command: `ps -ef`

Exercise 2 Create a shell-script which accepts a user name (xxxx) as a parameter, and shows the following information[7]:

- For each active process of the user: *User xxxx has a process with PID xxxx.*
- If the user has no active processes: *User xxxx has no active processes.*

Exercise 3 Show on-screen the contents of a file, numbering its lines as specified on each section.

- Showing all the lines of the file (empty and non-empty) with the following format: 4 characters for the line number, a whitespace, and the text line afterwards.

  For example, for the following file:

  ```
  line one
  line two

  line four

  line six
  ```

  you must obtain the following output (whitespaces indicated with ␣):

  ```
  ␣␣␣1␣line␣one
  ␣␣␣2␣line␣two
  ␣␣␣3
  ␣␣␣4␣line␣four
  ␣␣␣5
  ␣␣␣6␣line␣six
  ```

- Showing just the non-empty text lines and respecting the original line number. For the same example file, the output would be:

  ```
  ␣␣␣1␣line␣one
  ␣␣␣2␣line␣two
  ␣␣␣4␣line␣four
  ␣␣␣6␣line␣six
  ```

[7] If you had never thought of it, “informatics” comes from “**infor**mation” and “auto**matic**”: this is, automated information processing. Yet another cool thing to tell at parties[8]!

[8] At this point, all the other attendees will be most likely wondering who is that person spitting out random computer science facts, and who invited them. Conversely, if somebody is willing to engage in conversation following these facts, then congrats, you have met a potential new friend!

- Numbering separately and consecutively the empty and non-empty lines. After showing the file, you must print the number of empty and non-empty lines:

  For the same example file, the output would be:

  ```
  ␣␣␣1␣line␣one
  ␣␣␣2␣line␣two
  ␣␣␣1
  ␣␣␣3␣line␣four
  ␣␣␣2
  ␣␣␣4␣line␣six
  Empty␣lines:2.␣Non-empty␣lines:4
  ```

Exercise 4 Join the three options of the previous exercise in a single file. To do so, create a shell-script which accepts as parameters a filename and an option (from 1 to 3).

You must check that the number of parameters is correct, the file exists, and the option is between 1 and 3. Show the corresponding messages otherwise.

Exercise 5 Write a script that accepts 2 parameters: a filename and a subject name. The file format is `student:subject:mark`. The script must do the following:

1. If a number of parameters different than 2 were given, show this message and finish: "Wrong number of parameters".
2. If the first parameter is not a regular file, show this message and finish: "XXXX is not a regular file".
3. Use *awk* to obtain the following lists (sorted decreasingly by mark):
   - 2 best marks of the subject indicated as second parameter.
   - 3 worst marks of that same subject.
   - 5 best marks of any other subjects that are not the one indicated.

Exercise 6 Write a script that accepts a user name as parameter. This script must do the following:

1. If no parameters are given, show this message and finish: "No parameters were given".
2. Taking the output of `top` as input, use *awk* to obtain the following lists:
   - First 3 processes (by PID number) of the given user. Show only the PID, USER and COMMAND fields. If the user has no active processes, show the message "User xxxx has no active proceses".
   - The 5 processes with highest CPU usage. Only show the fields USER, CPU and COMMAND.

Tip: use this command `top -b -n1`