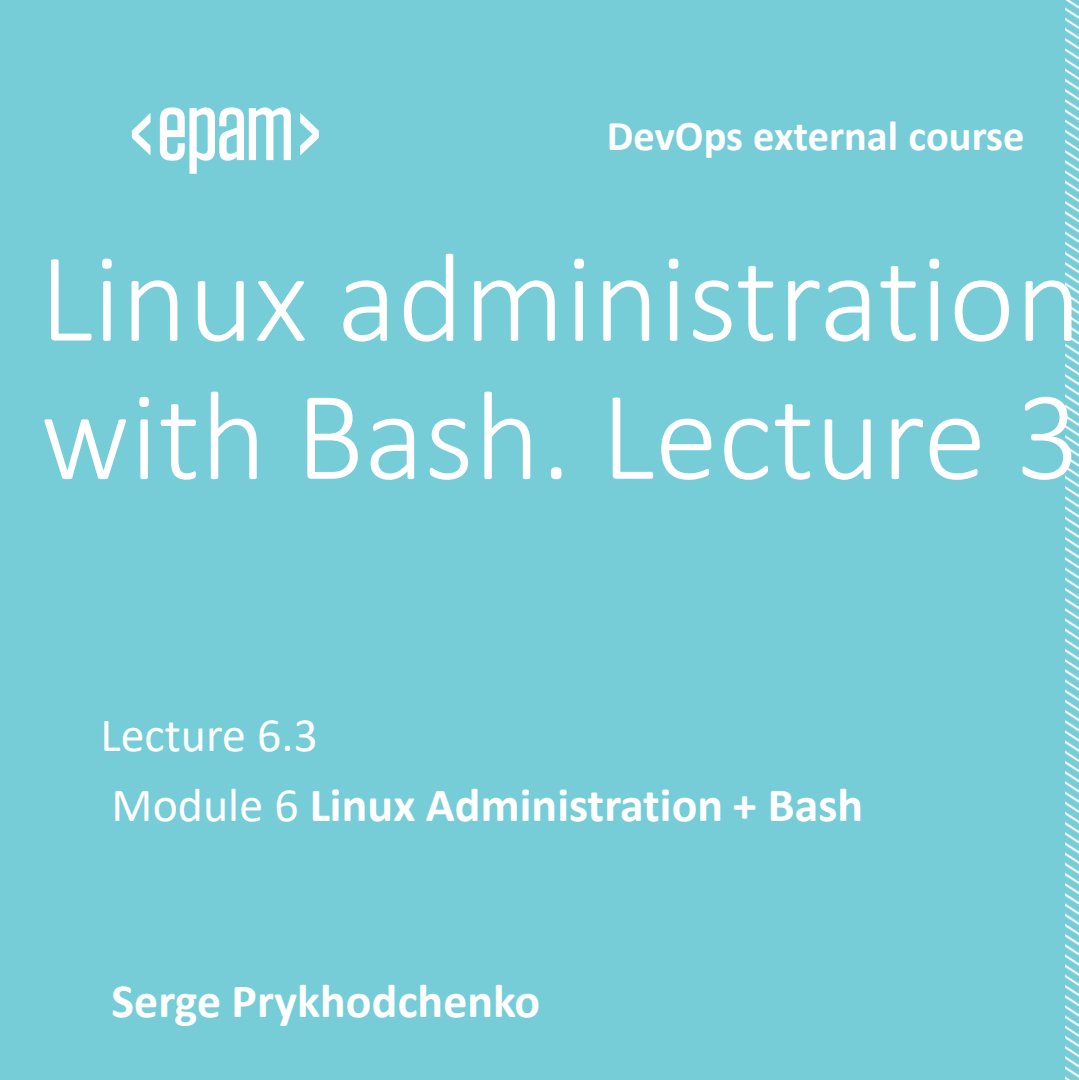


DevOps external course

Linux administration with Bash. Lecture 3

Lecture 6.3

Module 6 **Linux Administration + Bash**

Serge Prykhodchenko

N1tero · 9m · *edited 5m*

Programming is fun, sometimes you may google stuff like:

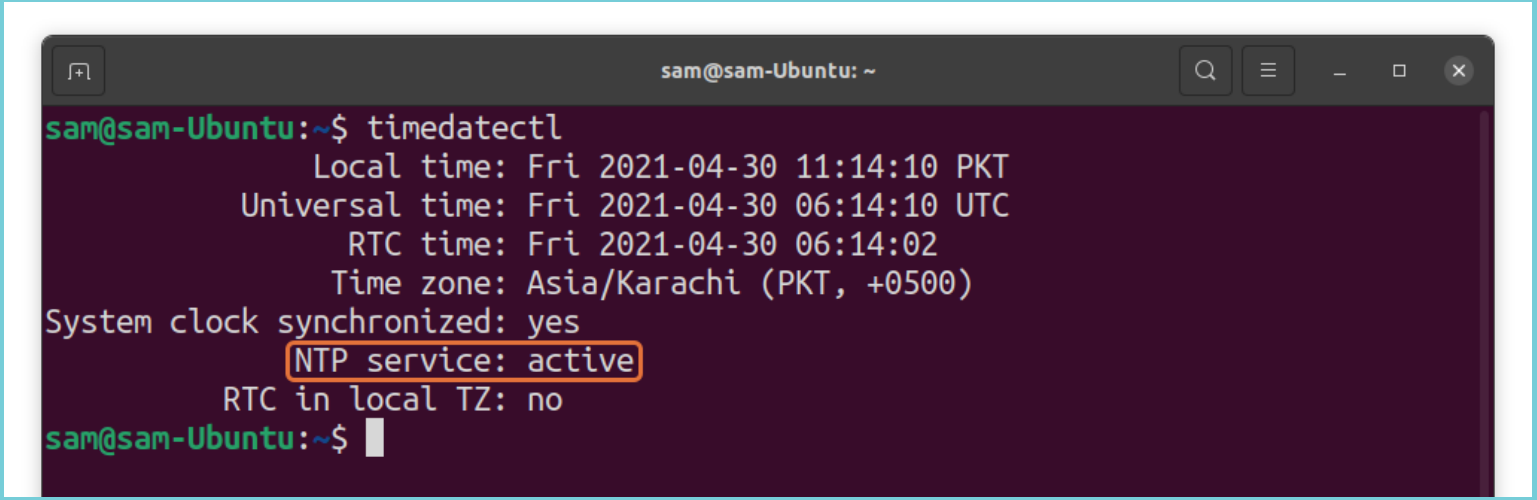
"how to kill child and parent"
"how to kill all children in a class"
"how to re-attach a detached head"
"how to bash cat with pipe"

Agenda

- Administration
- Scripting
- Q&A

ADMINISTRATION

Time sync



timedatectl status
sudo timedatectl set-ntp true

https://linuxhint.com/sync-time-ntp-server-linux/

Time / date and measurement of time intervals

time

Displays detailed statistics on the execution of a command.

touch

The utility sets the time of the last access / modification of the file to the current system time or at a specified time, but it can also be used to create a new empty file. The touch zzz command will create a new empty file named zzz if there was no zzz file before. In addition, such empty files can be used to indicate, for example, the time of the last change in the project.The equivalent of the touch command is: >> newfile or >> newfile (for regular files).

at

The at command is used to start jobs at a specified time. In general terms, it resembles crond, however, at is used to run a set of commands once.at 2pm January 15 - will ask you to enter a set of commands that must be run at the specified time. These commands must be compatible with shell scripts. The entry is completed by pressing the key combination Ctl-D.

Time / date and measurement of time intervals

batch
The batch command, which controls the start of jobs, resembles the at command, but only runs the list of commands when the system load drops below .8. Like the at command, with the -f option, it can read a set of commands from a file.
cal
Prints a neatly formatted calendar for the current month to stdout. Can display a calendar for a specific year.
sleep
Pauses script execution for a specified number of seconds without doing anything. Can be used to synchronize processes running in the background, checking for an expected event as often as needed..
usleep
Microsleep (here the character "u" should be read as a letter of the Greek alphabet - "mu", or the prefix micro). This is the same as sleep, only the time interval is specified in microseconds. Can be used for very fine synchronization of processes.
hwclock, clock
The hwclock command is used to access or correct the computer's hardware clock. Some keys require root privileges. The /etc/rc.d/rc.sysinit script uses the hwclock command to set the system time at boot time.

Time / date and measurement of time intervals

```
real    0m21,422s
user    0m1,757s
sys     0m4,295s
EST  Sat Apr 25 22:30:59 2020 EST
неділя, 26 квітня 2020 03:30:59 +0000
    Квітня 2020
нд пн вт ср чт пт сб
           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
```

```
#!/bin/bash
time ls -R -l /usr
zdump EST
date -u
cal
exit 0
```

Timed tasks management

Traditionally, Unix uses the **crond** daemon to run scheduled tasks. At startup, this program reads user schedule files from the **/var/spool/cron/** directory and the system-wide schedule stored in the **/etc/crontab** file and files located in the **/etc/cron.d/** directory. The system-wide schedule differs from the user one in that each of its records contains a field with the identifier of the user on whose behalf the task is being executed.

After the schedules are loaded into RAM, crond every minute checks for an entry in the schedule corresponding to the current time, and, if such is found, runs the specified command on behalf of the specified user. The output of the command (stdout and stderr) is intercepted and sent by mail specified in the schedule. By default, mail is sent to the local user when a custom schedule is executed, and when a system-wide schedule is executed, it is sent to root.

If crond detects a change in the mtime in the **/etc/crontab** file or in one of the schedules directories, it automatically rereads all the schedule files.

Timed tasks management

System-wide schedule format
 # Setting environment variables
name = value
#schedule line
mm hh DD MM DW ***user*** *cmd [arg...]*

Custom Schedule Format
differs in the absence of the user field
mm hh DD MM DW cmd [arg...]

mm hh DD MM DW – execution time in the format: minute hour day month month day week. Any field can be written as follows:
* - any value
1-7 - range of values from and to
0-10,20,30,40,50 - enumeration of values and ranges
* / 10 - iteration with a step, i.e. 0.10.20 ...0-10 / 2 - in between with a step
The month and day_ of the week are specified by the number or the first three letters of the name.

Timed tasks management

Environment variables that affect command execution:
LOGNAME - username (cannot be changed)
HOME - home directory in which the program will be launched (by default from /etc/passwd)
SHELL - the command interpreter to which the command will be passed for execution (by default from /etc/passwd)
MAILTO - mailing address to send the execution result (by default to the local user)

Editing a personal schedule
crontab file - copy personal schedule from crontab file
-e - launch vi editor to edit schedule crontab
-l - view schedule
crontab -r - delete schedule

Timed tasks management

Example of a personal schedule

```
# some people like the zsh interpreter
SHELL = /bin/zsh
# We send mail to an external address
MAILTO = test@example.com
#
# every day at zero hours five minutes
# intercept the output so that no letters arrive
5 0 * * * $ HOME /bin/daily.job >> /dev/null 2> & 1
# At 14:15 on the first of every month
15 14 1 * * $ HOME /bin/monthly
# check mail every five minutes
* / 5 * * * * $ HOME/bin/check_mail
```

Timed tasks management

One-time execution of scheduled actions – at

To run the command once at a specified time, the atd daemon is used, the tasks for which are generated by the at command. The command is entered from standard input or read from a file. Time is set in various formats, for example 16:00 on 12/31/2014 or "now +3 hours" (minutes, hours, days)

echo touch /tmp/test | at 14:35 # create a file today at 14:35, and if it's too late, then tomorrow at the specified

timeat -f file now + 14days # read command from file and execute after two weeks

atq # show all scheduled tasks

atrm id # delete job with id

Timed tasks management

Anacron service

The anacron service was originally invented for personal computers, which can be turned off at the moment when crond had to run the next command. If anacron detects that a daily, weekly, or monthly command has not been executed, it will launch it. The anacron configuration is stored in / etc / anacrontab in the format:

period delay job-identifier command

Where

period - the frequency of command execution in days

delay - if the command was not executed within the specified period, then a delay of minutes is made and the command is launched

job-identifier - arbitrary text string for writing to log files

command - the command to execute

Timed tasks management

Anacron service

Typical /etc/anacrontab file (run-parts - run all files in a directory):

```
#period in days   delay in minutes   job-identifier   command
1                 5                  cron.daily       nice run-parts /etc/cron.daily
7                 25                 cron.weekly      nice run-parts /etc/cron.weekly
@monthly          45                 cron.monthly     nice run-parts /etc/cron.monthly
```

While and until loop statements

The shell also supports traditional conditional loops with the following syntax:

<operator while> ::=
  while < statement > **do** <commands> **done**

<operator until> ::=
  until <statement> **do** <commands> **done**

The commands specifying the condition are executed and the return code of the last one is checked. If it is zero (true), the commands in the body of the while loop are executed, or the execution of the until loop ends. If it is not zero (false), the while loop ends, or another iteration of the until loop is executed.

While and until loop statements

```
serge@sergeX:~$ ./t2.sh
5
4
3
2
1
serge@sergeX:~$
```

```
#!/bin/bash
var1=5
while [ $var1 -gt 0 ]
do
echo $var1
var1=$[ $var1 - 1 ]
done
```

```
serge@sergeX:~$ ./t2.sh
1
2
3
4
5
6
7
8
9
10
11
```

```
#!/bin/bash
count=0
until [ $count -gt 10 ]
do
(( count++ ))
echo $count
done
```

Subshells

Running the script launches the child shell. Which interprets and executes the list of commands contained in the script file, just as if they were entered from the command line. Any script is launched as a child process of the parent command shell, the one that displays the prompt for input on the console or in the xterm window.

The script can also start another child process in its subshell. This allows scripts to parallelize data processing across multiple tasks that are executed simultaneously.

List of commands in parentheses
(command1; command2; command3; ...)
The list of commands, in parentheses, is executed in a subshell.

Subshells

Subshell processes can run in parallel. This allows you to split a complex task into several simple subtasks that perform parallel information processing.
Running multiple processes in subshells

```
(cat list1 list2 list3 | sort | uniq > list123) &
(cat list4 list5 list6 | sort | uniq > list456) &
# Merging and sorting two lists is done simultaneously.
# Running in the background guarantees parallel execution.
#
# The same effect gives#
cat list1 list2 list3 | sort | uniq > list123 &
cat list4 list5 list6 | sort | uniq > list456 &
wait   # Waiting for subshells to complete.
diff list123 list456
```

sum, cksum, md5sum

These utilities are designed to calculate checksums. Checksum is a certain number calculated based on the contents of the file and serves to control the integrity of the information in the file. The script can perform checksum checks to ensure that the file has not been modified or corrupted. For added security, it is recommended to use the 128-bit sum generated by the md5sum utility (message digest checksum).

bash$ cksum /boot/vmlinuz
1670054224 804083 /boot/vmlinuz

bash$ md5sum /boot/vmlinuz
0f43eccea8f09e0a0b2b5cf1dcf333ba /boot/vmlinuz

shred

Reliable, from the point of view of security, erasing a file by means of preliminary, multiple recording of random information in the file before deleting it.

It is part of the GNU fileutils package.

There are a number of technologies with the help of which it is still possible to recover files deleted by the shred utility.
https://www.gnu.org/software/coreutils/manual/html_node/shred-invocation.html

Aliases

Bash aliases are nothing more than hotkeys, a way to avoid typing long lines on the command line. If, for example, insert the line **alias lm = "ls -l | more"** into the ~ /.bashrc file, then later you can save your time and effort by typing the **lm** command instead of the longer **ls -l | more**. By setting alias **rm** = **"rm -i"** (interactive mode for deleting files), you can avoid a lot of trouble, because it will reduce the likelihood of deleting important files inadvertently.

Aliases in scripts can be very limited in scope. It would be great if aliases had the same functionality as C macros, but unfortunately Bash cannot "expand" arguments in the body of an alias. In addition, attempting to access an alias created within "compound constructs" such as if / then, loops, and functions will result in errors. In almost all cases, the actions assigned to the alias can be more efficiently performed using functions.

The **unalias** command removes the previously declared alias.

SCRIPTING

uniq

The uniq command in Linux is a command line utility that reports or filters out the repeated lines in a file.

//...syntax of uniq...//

$uniq [OPTION] [INPUT[OUTPUT]]

Options For uniq Command:

-c – -count : It tells how many times a line was repeated by displaying a number as a prefix with the line.

-d – -repeated : It only prints the repeated lines and not the lines which aren't repeated.

-D – -all-repeated[=METHOD] : It prints all duplicate lines and METHOD can be any of the following:

none : Do not delimit duplicate lines at all. This is the default.

prepend : Insert a blank line before each set of duplicated lines.

separate : Insert a blank line between each set of duplicated lines.

-f N – -skip-fields(N) : It allows you to skip N fields(a field is a group of characters, delimited by whitespace) of a line before determining uniqueness of a line.

-i – -ignore case : By default, comparisons done are case sensitive but with this option case insensitive comparisons can be made.

-s N – -skip-chars(N) : It doesn't compares the first N characters of each line while determining uniqueness. This is like the -f option, but it skips individual characters rather than fields.

-u – -unique : It allows you to print only unique lines.

-w N – -check-chars(N) : It only compares N characters in a line.

https://www.geeksforgeeks.org/uniq-command-in-linux-with-examples/

sort

Syntax:
Sort (options) (file)

https://linuxhint.com/sort-command-in-linux-with-examples/

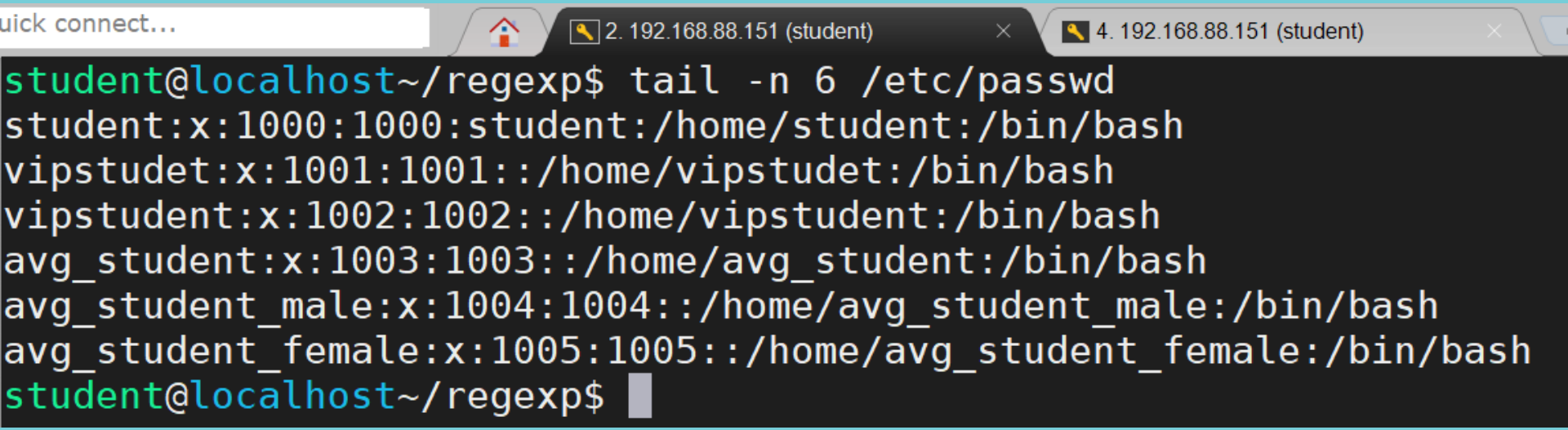
-b, **--ignore-leading-blanks**	Ignore leading blanks.
-d, **--dictionary-order**	Consider only blanks and alphanumeric characters.
-f, **--ignore-case**	Fold lower case to upper case characters.
-g, **--general-numeric-sort**	Compare according to general numerical value.
-i, **--ignore-nonprinting**	Consider only printable characters.
-M, **--month-sort**	Compare (unknown) < `**JAN**' < ... < `**DEC**'.
-h, **--human-numeric-sort**	Compare human readable numbers (e.g., "**2K**", "**1G**").
-n, **--numeric-sort**	Compare according to string numerical value.
-R, **--random-sort**	Sort by random hash of keys.
--random-source=*FILE*	Get random bytes from *FILE*.
-r, **--reverse**	Reverse the result of comparisons.
--sort=*WORD*	Sort according to *WORD*: general-numeric **-g**, human-numeric **-h**, month **-M**, numeric **-n**, random **-R**, version **-V**.
-V, **--version-sort**	Natural sort of (version) numbers within text.

Bash. Using Regular Expressions. Basics

Working with text files is an important skill for a Linux administrator. You not only have to know how to create and modify existing text files, but it is also very useful if you can find the text file that contains specific text.
If will be clear sometimes which specific text you are looking for. Other times, it might not. For example, are you looking for color or colour? Both spellings might give a match. This is just one example of why using flexible patterns while looking for text can prove useful. These flexible patterns are known as regular expressions in Linux.

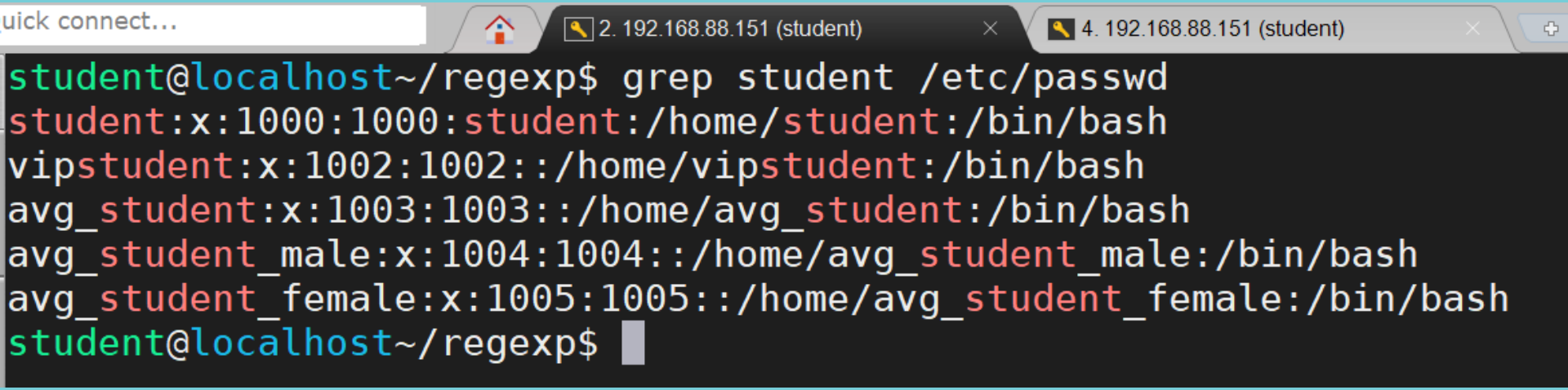
Bash. Using Regular Expressions. Basics

To understand regular expressions a bit better, let's take a look at a text file example. This file contains the last six lines from the /etc/passwd file.



Bash. Using Regular Expressions. Basics

Let's suppose that we are looking for the user ***student***. In that case, we could use the general regular expression parser ***grep*** to look for that specific string in the file ***/etc/passwd*** by using the command ***grep student /etc/passwd***. The results of that command, and as you can see, give us too many results as shown

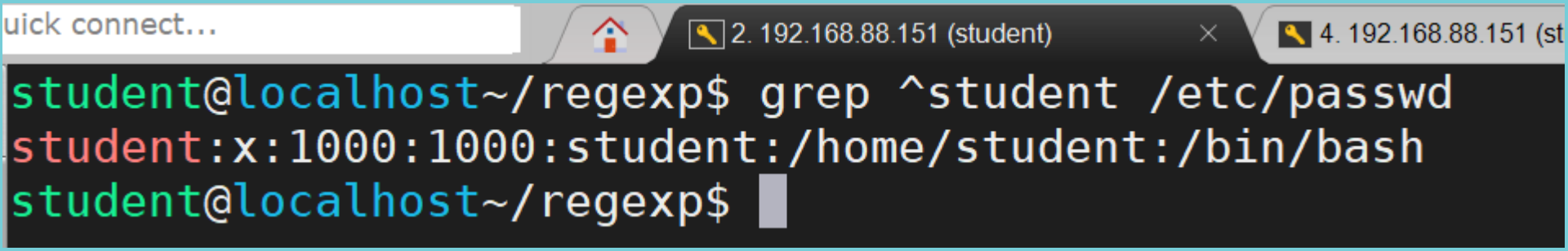


Bash. Using Regular Expressions. Basics

A regular expression is a search pattern that allows you to look for specific text in an advanced and flexible way.

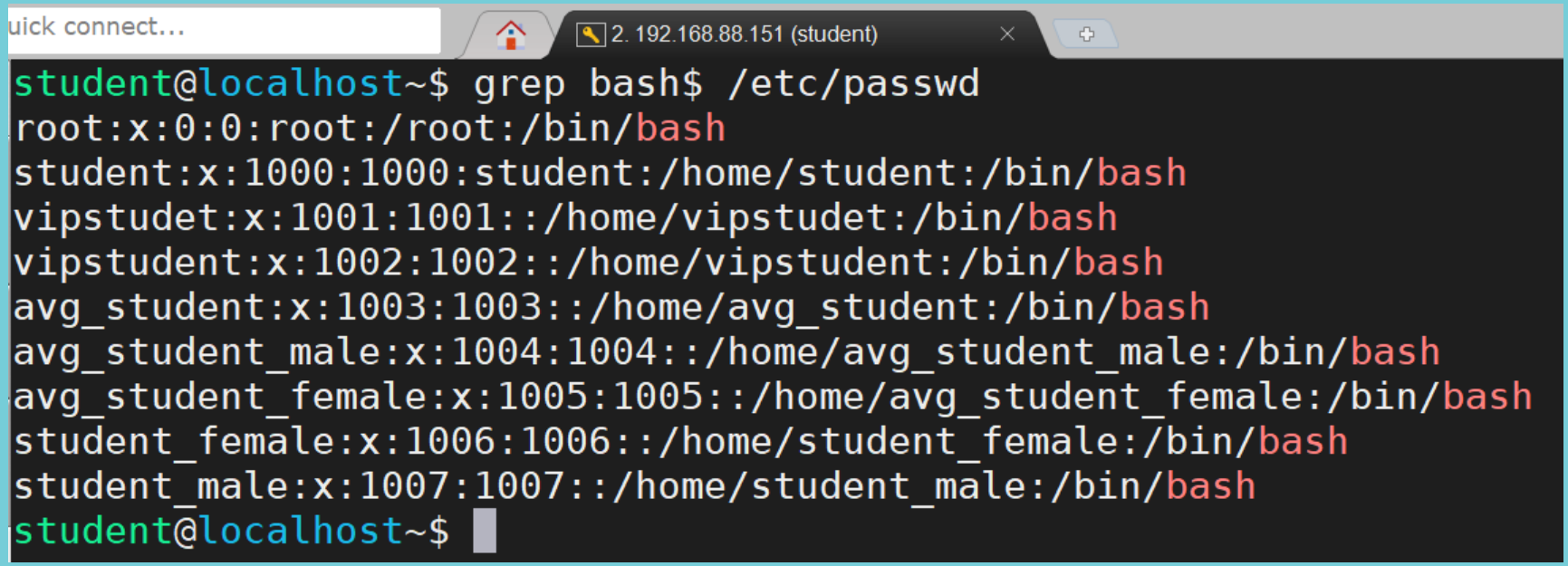
In previous slide, we might want to specify that you are looking for lines that are starting with the text ***student***. The type of regular expression that specifies where in a line of output the result is expected is known as a ***line anchor***.

To show only lines that start with the text you are looking for, you can use the ***regular expression ^*** (in this case, to indicate that you are looking only for lines where ***student*** is at the beginning of the line, as shown on next screenshot).



Bash. Using Regular Expressions. Basics

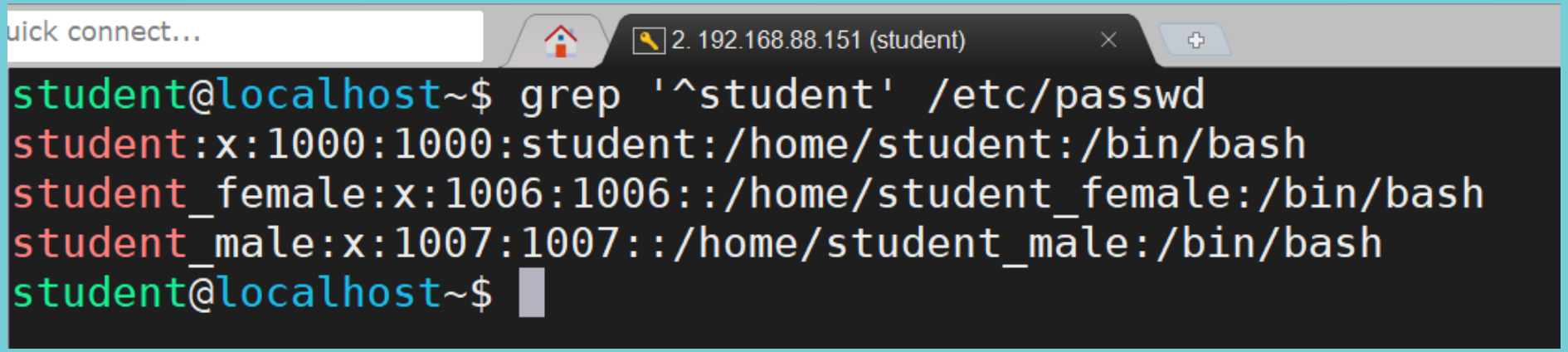
Another regular expression that relates to the position of specific text in a specific line is ***$***, which states that the line ends with some text. For instance, the command ***grep bash$ /etc/passwd*** shows all lines in the ***/etc/passwd*** file that end with the text ***bash***



Bash. Using Escaping in Regular Expressions

Although not mandatory, when using regular expressions it is a good idea to use escaping to prevent the regular expression from being interpreted by the shell. In many cases, it is not really necessary to use escaping; in some cases, the regular expression fails without escaping. To prevent this from ever happening, it is a good idea to put the regular expression between quotes.

So, instead of typing ***grep ^student /etc/passwd***, it is better use ***grep '^student' /etc/passwd***, even if in this case both examples work

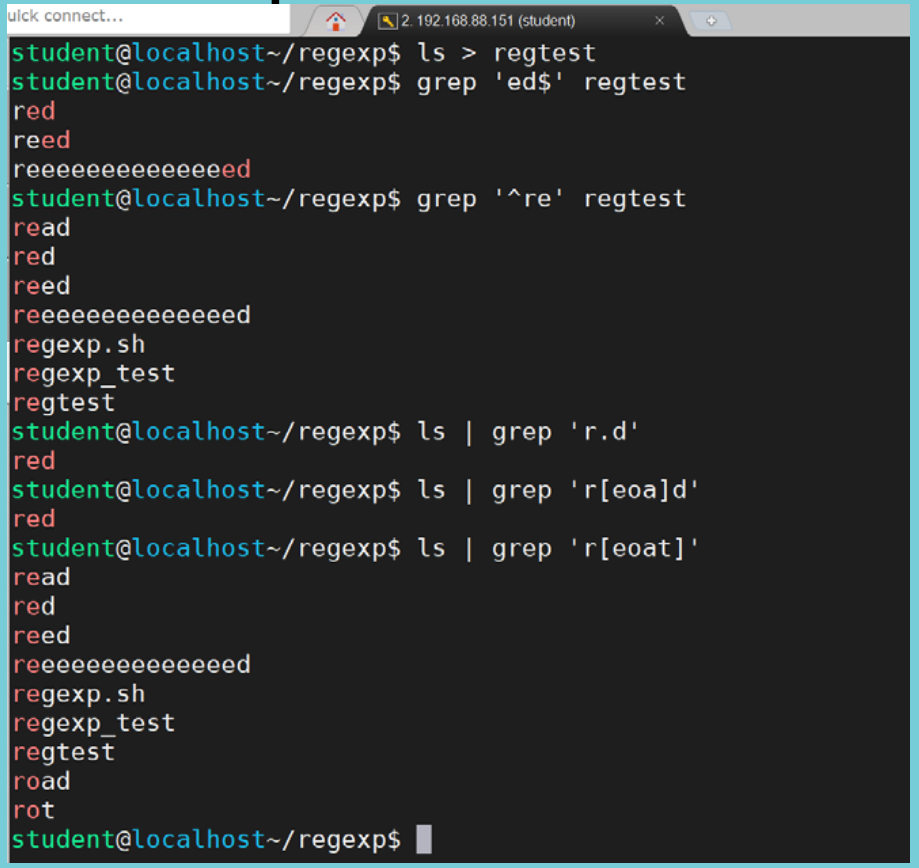


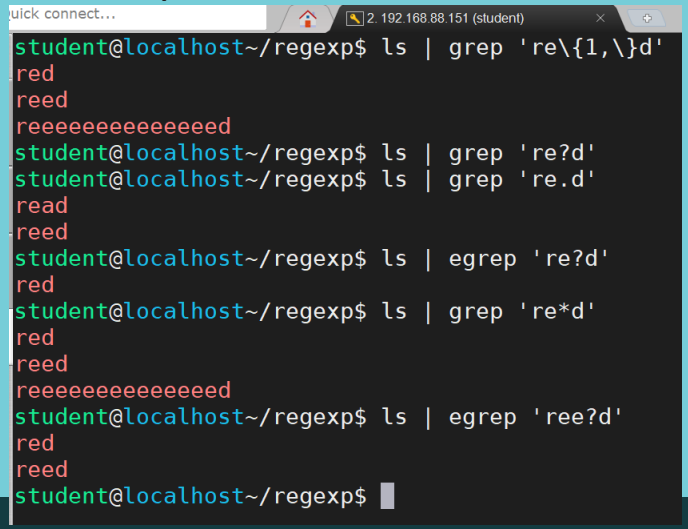
Bash. Using Wildcards and Multipliers.

In some cases, you might know which text you are looking for, but you might not know how the specific text is written. Or you just want to use one regular expression to match different patterns. In those cases, wildcards and multipliers come in handy.
To start with, there is the ***. regular expression***. This is used as a wildcard character to look for one specific character. So, the ***regular expression r.t*** would match the strings ***rat, rot, and rut***.
In some cases, you might want to be more specific about the characters you are looking for. If that is the case, you can specify a range of characters that you are looking for. For instance, the ***regular expression r[aou]t*** matches the strings ***rat, rut, as well as rot***.
Another useful regular expression is the ***multiplier **. This matches zero or more of the previous character. That does not seem to be very useful, but indeed it is, as you will see in the examples at the end of this section.

Bash. Using Wildcards and Multipliers.

If you know exactly how many of the previous character you are looking for, you can specify a number also, as in ***re\{2\}d***, which would match red as well as reed. The last regular expression that is useful to know about is ***?***, which matches zero or one of the previous character.





Bash. Using Wildcards and Multipliers.

Regular Expression	Use
^text	Line starts with text.
text$	Line ends with text.
.	Wildcard. (Matches any single character.)
[abc]	Matches a, b, or c.
*	Match 0 to an infinite number of the previous character.
\{2\}	Match exactly 2 of the previous character.
\{1,3\}	Match a minimum of 1 and a maximum of 3 of the previous character.
colou?r	Match 0 or 1 of the previous character. This makes the previous character optional, which in this example would match both *color* and *colour*.

Bash. Using grep to Analyze Text. Most Useful grep Options

Option	Use
-i	Not case sensitive. Matches uppercase as well as lowercase.
-v	Only show lines that do *not* contain the regular expression.
-r	Search files in the current directory and all subdirectories.
-e	Use this to search for lines matching more than one regular expression.
-A <number>	Show <number> of lines after the matching regular expression.
-B <number>	Show <number> of lines before the matching regular expression.

Bash. Using grep to Analyze Text. Most Useful grep Options

```
student@localhost~/regexp$ ls | grep -i 'a'
123abc
AAA555
read
road
student@localhost~/regexp$ grep -i 'a' regtest
123abc
AAA555
read
road
student@localhost~/regexp$
```

```
student@localhost~/regexp$ egrep -r 'a' ~/regexp
/home/student/regexp/regexp.sh:#!/bin/bash
/home/student/regexp/regexp.sh:echo -n 'Input regexp ex
/home/student/regexp/regexp.sh:read regexp
/home/student/regexp/regexp.sh:touch 123abc 234cvb 456
/home/student/regexp/regexp.sh:echo -n 'Content of crea
/home/student/regexp/regexp_test:123abc
/home/student/regexp/regtest:123abc
/home/student/regexp/regtest:read
/home/student/regexp/regtest:road
student@localhost~/regexp$
```

```
student@localhost~/regexp$ grep -i -A 5 'a' regtest
123abc
2
22
234cvb
3
33
--
AAA555
FFFBBB
kjfsdhk
oiu
read
red
reed
reeeeeeeeeeeeeed
regexp.sh
regexp_test
--
road
rot
rut
vcbn
student@localhost~/regexp$
```

Bash. Using regulars. More examples

This shows that the file /etc/sysconfig/sshd contains a number of lines that start with the comment sign #

```
uick connect...   2. 192.168.88.151 (student)
root@localhost/home/student/regexp$ grep '^#' /etc/sysconfig/sshd
# Configuration file for the sshd service.
# The server keys are automatically generated if they are missing.
# To change the automatic creation uncomment and change the appropriate
# line. Accepted key types are: DSA RSA ECDSA ED25519.
# The default is "RSA ECDSA ED25519"
# AUTOCREATE_SERVER_KEYS=""
# AUTOCREATE_SERVER_KEYS="RSA ECDSA ED25519"
# Do not change this option unless you have hardware random
# generator and you REALLY know what you are doing
# SSH_USE_STRONG_RNG=1
root@localhost/home/student/regexp$
```

This shows our Linux distributive details (universal for distributives, like RedHat and Debian families)

```
uick connect...   2. 192.168.88.151 (student)
root@localhost/home/student/regexp$ cat /etc/*release
CentOS Linux release 7.6.1810 (Core)
NAME="CentOS Linux"
VERSION="7 (Core)"
ID="centos"
ID_LIKE="rhel fedora"
VERSION_ID="7"
PRETTY_NAME="CentOS Linux 7 (Core)"
ANSI_COLOR="0;31"
CPE_NAME="cpe:/o:centos:centos:7"
HOME_URL="https://www.centos.org/"
BUG_REPORT_URL="https://bugs.centos.org/"

CENTOS_MANTISBT_PROJECT="CentOS-7"
CENTOS_MANTISBT_PROJECT_VERSION="7"
REDHAT_SUPPORT_PRODUCT="centos"
REDHAT_SUPPORT_PRODUCT_VERSION="7"

CentOS Linux release 7.6.1810 (Core)
CentOS Linux release 7.6.1810 (Core)
root@localhost/home/student/regexp$
```

Bash. Using regulars. More examples

The ***grep*** utility is a powerful utility that allows you to work with regular expressions. It is not the only utility, though. Some even more powerful utilities exist, like ***awk*** and ***sed***. Both utilities are extremely rich and merit a book by themselves. As a Linux administrator in the twenty-first century, you do not have to be a specialist in using these utilities anymore. It does make sense, however, to know how to perform some common tasks using these utilities. The most useful use cases are summarized in the following examples:

This command shows the ***fourth column from /etc/passwd.***
awk -F : '{ print $4 }' /etc/passwd

You can also use the ***awk*** utility to do tasks that you might be used to using ***grep*** for.
awk -F : '/student/ { print $4 }' /etc/passwd

What is awk?

- created by: Aho, Weinberger and Kernighan
- scripting language used for manipulating data and generating reports
- versions of awk:
  - awk, nawk, mawk, pgawk, …
- GNU awk: gawk

What can you do with awk?

- awk operation:
  - reads a file line by line
  - splits each input line into fields
  - compares input line/fields to pattern
  - performs action(s) on matched lines
- Useful for:
  - transform data files
  - produce formatted reports
- Programming constructs:
  - format output lines
  - arithmetic and string operations
  - conditionals and loops

Basic awk invocation

```
awk 'script' file(s)
awk –f scriptfile file(s)
```

- common option: **`-F`**
  - to change field separator

Basic awk script

- consists of patterns & actions:

```
pattern {action}
```

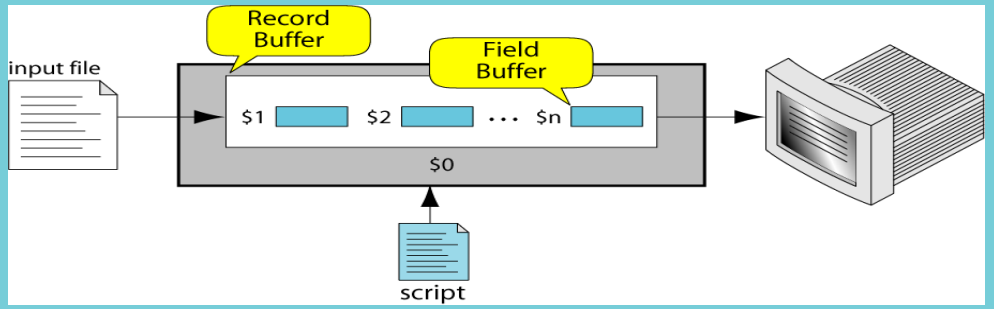
  - if pattern is missing, action is applied to all lines
  - if action is missing, the matched line is printed
  - must have either pattern or action

Example:

```
awk '/for/ { print }' testfile
```

  - prints all lines containing string “for” in testfile

awk variables



- awk reads input line into buffers: record and fields
- field buffer:
  - one for each field in the current record
  - variable names: $1, $2, …
- record buffer:
  - $0 holds the entire record

More awk variables

NR Number of the current record

NF Number of fields in current record

<u>also:</u>

FS Field separator (default=whitespace)

Example: Records and Fields

```
% cat emps
Tom Jones          4424     5/12/66 543354
Mary Adams         5346     11/4/63 28765
Sally Chang        1654     7/22/54 650000
Billy Black        1683     9/23/44 336500

% awk '/Tom/ { print }' emps
Tom Jones       4424     5/12/66 543354
```

Example: Records and Fields

```
% cat emps
Tom Jones        4424    5/12/66 543354
Mary Adams       5346    11/4/63 28765
Sally Chang      1654    7/22/54 650000
Billy Black      1683    9/23/44 336500

% awk '{print NR, $0}' emps
1 Tom Jones      4424    5/12/66 543354
2 Mary Adams     5346    11/4/63 28765
3 Sally Chang    1654    7/22/54 650000
4 Billy Black    1683    9/23/44 336500
```

Example: Space as Field Separator

```
% cat emps
Tom Jones        4424    5/12/66 543354
Mary Adams       5346    11/4/63 28765
Sally Chang      1654    7/22/54 650000
Billy Black      1683    9/23/44 336500

% awk '{print NR, $1, $2, $5}' emps
1 Tom Jones 543354
2 Mary Adams 28765
3 Sally Chang 650000
4 Billy Black 336500
```

Example: Colon as Field Separator

```
% cat emps2
Tom Jones:4424:5/12/66:543354
Mary Adams:5346:11/4/63:28765
Sally Chang:1654:7/22/54:650000
Billy Black:1683:9/23/44:336500
% awk -F: '/Jones/{print $1, $2}'
  emps2
Tom Jones 4424
```

Special Patterns

- BEGIN
  - matches before the first line of input
  - used to create header for report
- END
  - matches after the last line of input
  - used to create footer for report

example input file

```
Jan 13 25 15 115
Feb 15 32 24 22
Mar 15 24 34 228
Apr 31 52 63 420
May 16 34 29 208
Jun 31 42 75 492
Jul 24 34 67 436
Aug 15 34 47 316
Sep 13 55 37 277
Oct 29 54 68 525
Nov 20 87 82 577
Dec 17 35 61 401
Jan 21 36 64 620
Feb 26 58 80 652
Mar 24 75 70 495
Apr 21 70 74 514
```

awk example runs

- `awk '{print $1}' input`
- `awk '$1 ~ /Feb/ {print $1}' input`
- `awk '{print $1, $2+$3+$4, $5}' input`
- `awk 'NF == 5 {print $1, $2+$3+$4, $5}' input`

awk example script

```
BEGIN {
        print "January Sales Revenue"
}
$1 ~ /Jan/ {
        print $1, $2+$3+$4, $5
}
END {
        print NR, " records processed"
}
```

Categories of Patterns

- simple patterns
  - BEGIN, END
  - expression patterns: whole line vs. explicit field match
    - whole line /regExp/
    - field match $2 ~ /regExp
- range patterns
  - specified as from and to:
    - example: /regExp/,/regExp/

awk actions

- basic expressions

- output: **`print, printf`**
- decisions: **`if`**
- loops: **`for, while`**

awk Expression

- consists of: operands and operators
- operands:
  - numeric and string constants
  - variables
  - functions and regular expression
- operators:
  - assignment: = ++ -- += -= *= /=
  - arithmetic: + - * / % ^
  - logical: && || !
  - relational: > < >= <= == !=
  - match: ~ !~
  - string concatenation: space

awk Variables

- created via assignment:

```
var = expression
```

- types: number (not limited to integer)
  string, array
- variables come into existence when first used
- type of variable depends on its use
- variables are initialized to either 0 or “”

awk variables example

```
BEGIN {
        print "January Sales Revenue"
        count = 0
        sum = 0
}
$1 ~ /Jan/ && NF == 5 {
        print $1, $2+$3+$4, $5
        count++
        sum += $5
}
END {
        print count, " records produce: ", sum
}
```

awk output: print

- Writes to standard output
- Output is terminated by newline
- If called with no parameter, it will print $0
- Printed parameters are separated by blank
- Print control characters are allowed:
  - \n \f \a \t \b \\ …

print examples

```
% awk '{print $1, $2}' grades
john 85
andrea 89
jasper 84
% awk '{print $1 "," $2}' grades
john,85
andrea,89
jasper,84
```

printf: Formatting output

Syntax:

```
printf(format-string, var1, var2, …)
```

- each format specifier within “format-string” requires additional argument of matching type

%d, %i	decimal integer
%c	single character
%s	string of characters
%f	floating point number

Format specifier modifiers

- between “%” and letter

  %10s

  %7d

  %10.4f

  %-20s

- meaning:
  - width of field, field is printed right justified (“-” will left justify)
  - precision: number of digits after decimal point

awk Example: list of products

```
101:propeller:104.99
102:trailer hitch:97.95
103:sway bar:49.99
104:fishing line:0.99
105:mirror:4.99
106:cup holder:2.49
107:cooler:14.89
108:wheel:49.99
109:transom:199.00
110:pulley:9.88
111:lock:31.00
112:boat cover:120.00
113:premium fish bait:1.00
```

awk Example: output

```
Marine Parts R Us

Main catalog

Part-id name                      price
====================================

101     propeller                104.99
102     trailer hitch             97.95
103     sway bar                  49.99
104     fishing line               0.99
105     mirror                     4.99
106     cup holder                 2.49
107     cooler                    14.89
108     wheel                     49.99
109     transom                  199.00
110     pulley                     9.88
111     lock                      31.00
112     boat cover               120.00
113     premium fish bait          1.00
====================================

Catalog has 13 parts
```

awk Example: complete

```
BEGIN {
        FS= ":"
        print "Marine Parts R Us"
        print "Main catalog"
        print "Part-id\tname\t\t\t price"
        print "================================="
}
{
        printf("%3d\t%-20s\t%6.2f\n", $1, $2, $3)
}
END {
        print "================================="
        print "Catalog has", NR, "parts"
}
```

Bash. Using awk. Examples

```
root@localhost/home/student/regexp$ awk -F : '{ print $1 " "$4 }' /etc/passwd
root 0
bin 1
daemon 2
adm 4
lp 7
sync 0
shutdown 0
halt 0
mail 12
operator 0
games 100
ftp 50
nobody 99
systemd-network 192
dbus 81
polkitd 998
sshd 74
postfix 89
student 1000
vipstudet 1001
vipstudent 1002
avg_student 1003
avg_student_male 1004
avg_student_female 1005
student_female 1006
student_male 1007
root@localhost/home/student/regexp$
```

```
root@localhost~$ awk -F : '/student/ { print $1 "\t" $4 }' /etc/passwd
student 1000
vipstudent      1002
avg_student     1003
avg_student_male        1004
avg_student_female      1005
student_female  1006
student_male    1007
root@localhost~$
```

Bash. Using awk. Examples

A classic example where we need to use the ***-d*** option to extract the list of users on the current system from the ***/etc/passwd*** file.

```
student@localhost~$ cut -d : -f 1 /etc/passwd
root
bin
daemon
adm
lp
sync
shutdown
halt
mail
operator
games
ftp
nobody
systemd-network
dbus
polkitd
sshd
postfix
student
vipstudet
vipstudent
avg_student
avg_student_male
avg_student_female
student_female
student_male
test
```

Bash. Using awk. Examples. Fragment of Apache log-file.

```
46.29.2.62 - - [30/Sep/2015:00:15:01 +0300] "GET /graffiti HTTP/1.0" 200 146 "-" "Wget/1.12 (linux-gnu)"
5.255.253.45 - - [30/Sep/2015:00:15:02 +0300] "GET /muzykanty HTTP/1.0" 200 38658 "-" "Mozilla/5.0 (compatible; YandexBot/3.0; +http://yandex.com/bots)"
46.29.2.62 - - [30/Sep/2015:00:17:01 +0300] "GET /cats HTTP/1.0" 200 146 "-" "Wget/1.12 (linux-gnu)"
46.29.2.62 - - [30/Sep/2015:00:20:01 +0300] "GET /flowers HTTP/1.0" 200 146 "-" "Wget/1.12 (linux-gnu)"
46.29.2.62 - - [30/Sep/2015:00:22:01 +0300] "GET /dresses HTTP/1.0" 200 146 "-" "Wget/1.12 (linux-gnu)"
46.29.2.62 - - [30/Sep/2015:00:25:01 +0300] "GET /cars HTTP/1.0" 200 146 "-" "Wget/1.12 (linux-gnu)"
46.29.2.62 - - [30/Sep/2015:00:27:01 +0300] "GET /shoes HTTP/1.0" 200 146 "-" "Wget/1.12 (linux-gnu)"
157.55.39.174 - - [30/Sep/2015:00:27:41 +0300] "GET /vote/1279 HTTP/1.0" 302 - "-" "Mozilla/5.0 (compatible; bingbot/2.0; +http://www.bing.com/bingbot.htm)"
157.55.39.174 - - [30/Sep/2015:00:27:43 +0300] "GET /error404 HTTP/1.0" 200 2385 "-" "Mozilla/5.0 (compatible; bingbot/2.0; +http://www.bing.com/bingbot.htm)"
217.69.134.39 - - [30/Sep/2015:00:27:51 +0300] "GET /sitemap1.xml.gz HTTP/1.0" 304 - "-" "Mozilla/5.0 (compatible; Linux x86_64; Mail.RU_Bot/Fast/2.0; +http://go.mail.ru/help/robots)"
46.29.2.62 - - [30/Sep/2015:00:30:01 +0300] "GET /weather.php?get HTTP/1.0" 200 - "-" "Wget/1.12 (linux-gnu)"
46.29.2.62 - - [30/Sep/2015:00:30:01 +0300] "GET /architecture HTTP/1.0" 200 146 "-" "Wget/1.12 (linux-gnu)"
46.29.2.62 - - [30/Sep/2015:00:32:01 +0300] "GET /snowboard HTTP/1.0" 200 146 "-" "Wget/1.12 (linux-gnu)"
46.29.2.62 - - [30/Sep/2015:00:35:01 +0300] "GET /bike HTTP/1.0" 200 146 "-" "Wget/1.12 (linux-gnu)"
37.140.141.30 - - [30/Sep/2015:00:36:29 +0300] "GET /novogodnie/4/ HTTP/1.0" 200 13950 "-" "Mozilla/5.0 (compatible; YandexBot/3.0; +http://yandex.com/bots)"
46.29.2.62 - - [30/Sep/2015:00:37:01 +0300] "GET /skateboard HTTP/1.0" 200 146 "-" "Wget/1.12 (linux-gnu)"
46.29.2.62 - - [30/Sep/2015:00:42:01 +0300] "GET /funny animals HTTP/1.0" 200 146 "-" "Wget/1.12 (linux-gnu)"
```

```
#!/bin/bash
file_out = out_script1
grep -E -o "([0-9]{1,3}[\.]){3}[0-9]{1,3}" $1 | sort | uniq -c | sort -gr > $file_out
{
read line1
} < $file_out
echo $line1
```

From which ***ip*** were the most requests?

```
student@localhost~/regexp$ ./task_apach_2 apache_logs.txt
62 157.55.39.250
student@localhost~/regexp$
```

```
#!/bin/bash
file_out=out_script2
awk '{print $7}' $1 | sort | uniq -c | sort -gr > $file_out
{
read line1
}<$file_out
echo $line1
```

What is the most requested page?

```
uick connect...
2. 192.168.88.151 (student)
student@localhost~/regexp$ ./task_apach_3 apache_logs.txt
8 /sitemap1.xml.gz
student@localhost~/regexp$
```

Example

Forming of a PID array

```
serge     8205  0.0  0.3 220792  6940 ?        Sl   18:09   0:00 /usr/lib/at-spi
serge     8211  0.0  1.3 425364 27244 ?        Ssl  18:09   0:00 /usr/lib/x86_64
serge     8222  0.0  0.8 816924 16752 ?        S<l  18:09   0:00 /usr/bin/pulsea
serge     8249  0.0  1.6 491656 34336 ?        Sl   18:09   0:02 /usr/lib/x86_64
serge     8257  0.0  0.7 180736 15092 ?        S    18:09   0:00 /usr/lib/x86_64
serge     8258  0.0  1.4 402476 28760 ?        Sl   18:09   0:00 /usr/lib/x86_64
serge     8259  0.0  1.1 431920 23916 ?        Sl   18:09   0:00 /usr/lib/x86_64
serge     8265  0.0  1.3 507756 27680 ?        Sl   18:09   0:00 /usr/lib/x86_64
serge     8266  0.0  1.4 406568 30456 ?        Sl   18:09   0:00 /usr/lib/x86_64
serge     8267  0.1  1.5 698756 31964 ?        Sl   18:09   0:25 /usr/lib/x86_64
serge     8296  0.0  0.3 375076  8020 ?        Ssl  18:09   0:00 /usr/lib/x86_64
serge     8335  0.1  1.8 612264 36760 ?        Sl   18:09   0:21 mousepad
serge     8349  0.0  1.7 553588 35656 ?        Sl   18:11   0:01 xfce4-appfinder
root      8898  0.0  0.0      0     0 ?        I    20:59   0:00 [kworker/u2:1]
serge     8921  0.0  1.6 464132 34256 ?        Sl   21:14   0:02 /usr/bin/xfce4-
serge     8925  0.0  0.2  31652  5324 pts/0    Ss   21:14   0:00 bash
root      8995  0.0  0.0      0     0 ?        I    22:04   0:00 [kworker/u2:0]
root      8998  0.0  0.0      0     0 ?        I    22:14   0:00 [kworker/u2:2]
serge     8999  0.0  0.1  46444  3632 pts/0    R+   22:16   0:00 ps aux
serge@sergeX:~$ array=( `ps aux | grep java | awk '{print $2}'` )
serge@sergeX:~$ ./t2.sh
1052 7505 8002 8129 8147 8160 8202 8211 8249 8257 8258 8259 8265 8266 8267 8349
8921 9010
```

/home/serge/t2.sh - Mousepad

File Edit Search View Document Help

```
array=( `ps aux | grep xfce | awk '{print $2}'` )
echo ${array[@]}
```

Примеры

Скрипт для мониторинга дискового пространства

```
Top Ten Disk Space Usage
The /home Directory:
1:      960     /home/user3/.cache/gstreamer-1.0
2:      960     /home/serge/.cache/gstreamer-1.0
3:      960     /home/dummy/.cache/gstreamer-1.0
4:      156     /home/serge/.cache/fontconfig
5:      144     /home/serge
6:      80      /home/user3
7:      76      /home/serge/.config/pulse
8:      60      /home/serge/.config/xfce4/xfconf/xfce-perchannel-
9:      44      /home/serge/cpp/p2/p2/obj/Debug
10:     44      /home/dummy
The /var/log Directory:
1:      40972   /var/log/journal/47df12148e824e8ba2bf19e357801f82
2:      2988    /var/log
3:      712     /var/log/installer
4:      384     /var/log/apt
5:      72      /var/log/lightdm
6:      20      /var/log/unattended-upgrades
7:      20      /var/log/cups
8:      4       /var/log/speech-dispatcher
9:      4       /var/log/journal
10:     4       /var/log/hp/tmp
```

```
#!/bin/bash
MY_DIRECTORIES="/home /var/log"
echo "Top Ten Disk Space Usage"
for DIR in $MY_DIRECTORIES
do
echo "The $DIR Directory:"
du -S $DIR 2>/dev/null |
sort -rn |
sed '{11,$D; =}' |
sed 'N; s/\n/ /' |
awk '{printf $1 ":" "\t" $2 "\t" $3 "\n"}'
done
exit
```

QUESTIONS & ANSWERS

THANK YOU!